On the Computation of Discrete Logarithms in Finite Prime Fields

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.–Ing.)
der Technischen Fakultät
der Universität des Saarlandes
von

Damian Weber

Saarbrücken
1997

Tag des Kolloquiums: 30.10.1997

Dekan: Prof. Dr.–Ing. Alexander Koch

Gutachter:

Prof. Dr. Johannes Buchmann

Prof. Ph. D. Raimund Seidel

Acknowledgements

To write a thesis on discrete logarithms has been a very demanding and a highly fascinating challenge.

First of all, I would like to thank my thesis supervisor, Prof. Dr. Johannes Buchmann, who approved of my proposal to consider the practicability of the asymptotically fastest discrete logarithm algorithm for finite prime fields, the NFS, and who gave me the chance to join his research group at the university of Saarbrücken.

Many thanks to the members of this research group for their support and assistance; especially to Thomas Denny who has been able to solve bigger and bigger matrix equations and who shared his experience with the quadratic sieve factoring algorithm. I am also very grateful to Thomas Papanikolaou for his generosity in investing plenty of his time in answering questions about integer arithmetic and C++ internals. Thank you as well to him and Susanne Wetzel for proofreading the manuscript. Furthermore, I would like to thank Dr. Jörg Zayer for providing many implementation tricks and details of the NFS factoring algorithm.

Special thanks are due to Dr. Oliver Schirokauer (Oberlin College/Ohio) for his constructive feedback on algebraic number theory and for giving insight into his invention of the additive characters. I am also grateful to Marije Elkenbracht–Huizing (Amsterdam/Netherlands) for stimulating discussions about the NFS for factoring.

Many thanks to the GNU software project which provides numerous public domain software tools including the compilers `gcc`, `g++`, the awk–extension `gawk`, the packer `gzip`, and the profiler `gprof`.

Kurzzusammenfassung

In dieser Arbeit berichten wir über praktische Erfahrungen mit der Lösung von Kongruenzen der Form

$$a^x \equiv b \bmod p, \quad a, b, p, x \in \mathbb{Z}, \quad p \text{ Primzahl.}$$

Dies ist das Problem der *Diskreten Logarithmen* in $(\mathbb{Z}/p\mathbb{Z})^*$. Zahlreiche kryptographische Protokolle wie digitale Unterschriften, Verschlüsselung von Nachrichten, Schlüsselaustausch und Identifikation basieren auf der Schwierigkeit dieses Problems. In dieser Arbeit befassen wir uns mit der Praktikabilität verschiedener Index–Calculus Verfahren, die zur Zeit die asymptotisch schnellsten Algorithmen liefern, um dieses Problem zu lösen. Wir präsentieren Berechnungen mit bis zu 85–stelligem p und legen eine partielle Lösung zu McCurley's Challenge vor, die ein 129–stelliges p von spezieller Form benutzt.

Abstract

In this thesis we write about practical experience when solving congruences of the form

$$a^x \equiv b \bmod p, \quad a, b, p, x \in \mathbb{Z}, \quad p \text{ prime.}$$

This is referred to as the *discrete logarithm* problem in $(\mathbb{Z}/p\mathbb{Z})^*$. Many cryptographic protocols such as signature schemes, message encryption, key exchange and identification depend on the difficulty of this problem. We are concerned with the practicability of different index calculus variants, which are the asymtotically fastest known algorithms at present to solve this problem. We present computations for p having up to 85 decimal digits. We include a partial solution to McCurley's challenge with a 129–digit p, which has a special form.

Zusammenfassung

> Insofern sich die Sätze der Mathematik auf die Wirklichkeit beziehen, sind sie nicht sicher, und insofern sie sicher sind, beziehen sie sich nicht auf die Wirklichkeit.
>
> Albert Einstein

Diese Dissertation beschäftigt sich mit praktischen Erfahrungen bei der Benutzung verschiedener Index–Calculus Verfahren zur Lösung des Problems Diskreter Logarithmen (DLP) in der multiplikativen Gruppe endlicher Primkörper $(\mathbb{Z}/p\mathbb{Z})^*$, wobei p Primzahl ist. Diese Gruppe besteht aus $p-1$ Elementen. Das Problem Diskreter Logarithmen kann folgendermaßen formuliert werden. Gegeben seien ganze Zahlen a, b und eine Primzahl p. Zu finden ist eine ganze Zahl x mit der Eigenschaft

$$a^x \equiv b \bmod p,$$

oder ein Beweis dafür, daß ein solches x nicht existiert. Seit der Veröffentlichung des Diffie–Hellman'schen Schlüsselaustausch–Protokolls im Jahre 1978 ist das Interesse am DLP ständig gestiegen; dies ist aus der Erfindung vieler kryptographischer Protokolle, deren Sicherheit von der Schwierigkeit des DLP in bestimmten Gruppen abhängt, ersichtlich. Die Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$ gehört zu denjenigen Gruppen, in denen das DLP allgemein als schwierig angenommen wird, vorausgesetzt, daß p groß ist und in der Primfaktorzerlegung von $p-1$ ein großer Primfaktor q enthalten ist. Es ist daher nicht verwunderlich, daß zahlreiche kryptographische Verfahren die Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$ benutzen; sogar das amerikanische National Institute of Standards and Technology hat ein digitales Signaturverfahren zum Standard erhoben, welches diese Gruppe benutzt [49]. Dieses Verfahren basiert auf Vorschlägen von ElGamal [21] und Schnorr [67]. Selbstverständlich kann das DLP in $(\mathbb{Z}/p\mathbb{Z})^*$ auch mit Algorithmen gelöst werden, die in beliebigen Gruppen funktionieren. Beispielsweise kann das Verfahren von Silver, Pohlig und Hellman [55] benutzt werden, das entweder mit Hilfe der Methode von Shanks oder der von Pollard implementiert

werden kann. Der große Nachteil dieser Methode besteht jedoch in seiner Laufzeit, die exponentiell vom größten Primfaktor der Gruppenordnung $p-1$ abhängt. Dennoch kann für kleine Teiler q von $p-1$ mit diesem Algorithmus immer die partielle Lösung x modulo q erhalten werden.

Wegen der steigenden kryptographischen Bedeutung [51], wurde sehr intensiv nach DL-Algorithmen mit subexponentieller Laufzeit geforscht. Daher kennen wir nun einige deterministische und heuristische Verfahren für $(\mathbb{Z}/p\mathbb{Z})^*$; allen liegt die Index–Calculus Idee zugrunde, die man in nahezu jeder Übersicht zum DLP findet [51, 38, 44, 52, 65]. Da die Grundidee dieser Algorithmen bereits bei Faktorisierungsalgorithmen in Erscheinung trat, erben die DL–Algorithmen in natürlicher Weise die dortige Laufzeitnotation. Wir definieren

$$L_p[s,c] := \exp((c+o(1))\,(\log p)^s(\log\log p)^{1-s}).$$

Im Jahre 1987 bewies Pomerance, daß Diskrete Logarithmen in einem Körper von p^n Elementen mit einer erwarteten Laufzeit von $L_{p^n}[1/2,\sqrt{2}]$ berechnet werden können [60]. Der Artikel von Coppersmith, Odlyzko und Schroeppel [13] enthält drei heuristische Verfahren, die sich die Index–Calculus Idee zunutze machen; sie erreichen eine vermutete Laufzeit von $L_p[1/2,1]$. Im Jahre 1993 wurde ein größerer Durchbruch durch Gordon erzielt, der das beim Faktorisieren erfolgreiche Number Field Sieve (NFS) auf das DLP übertrug und damit eine heuristische erwartete Laufzeit von $L_p[1/3,3^{\frac{2}{3}}]$ erzielte [27].

Die neueste Effizienzsteigerung des NFS geht auf Schirokauer zurück, der die vermutete erwartete Laufzeit auf

$$L_p[1/3,(64/9)^{\frac{1}{3}}]$$

verbesserte [64]. Diese Laufzeit ist identisch mit der schnellsten asymptotischen Laufzeit, die zum Faktorisieren von zusammengeszten Zahlen gleicher Größe benötigt wird.

Dennoch blieb bis zum Jahre 1995 der einzige ernsthafte Versuch einer expliziten Berechnung der von LaMacchia und Odlyzko im Jahre 1991. Zu diesem Zeitpunkt zeigten sie die Unzulänglichkeit des von der Firma Sun verwendeten Kryptosystems zur Sicherung ihres verteilten UNIX Dateisystems auf. Hierbei wurde ein DLP mit einem 58–stelligen p gelöst.

Die vorliegende Dissertation stellt umfangreiches experimentelles Datenmaterial zur Verfügung, welches die Praktikabilität der verschiedenen Index–Calculus Versionen durch die Erweiterung der Methode von [64] aufzeigt. Außerdem wird eine weitere Version des NFS vom Faktorisieren auf das DLP übertragen. Ein Vergleich zwischen

den Verfahren gibt Aufschluß über tatsächliche Laufzeiten, die in den Konstanten des Ausdrucks $L_p[s, c]$ verborgen sind. Das folgende Beispiel vermittelt einen Eindruck davon, wie schwer es ist, die Exponentiation modulo einer Primzahl zu invertieren. Die Berechnung einer Potenz modulo einer 85–stelligen Primzahl dauert 32 Millisekunden auf einem 40–mips Rechner. Der gleiche Rechner benötigt jedoch ungefähr ein Jahr, um den benutzten Exponent aufzufinden – und dies mit Hilfe des besten bekannten DL–Algorithmus für endliche Primkörper dieser Größe.

Kapitel 1 führt in die Notation und die mathematischen Grundlagen ein, die benötigt werden, um die verwendeten Algorithmen zu beschreiben; diese werden in Kapitel 2 kurz skizziert.

Kapitel 3 bildet den Hauptteil dieser Arbeit. Es beginnt mit einer detaillierten Beschreibung der ersten NFS Implementierung für das DLP, greift deren zahlreiche Verfeinerungen mit ihren vier Variationen auf und erklärt, wie die Parameter gewählt werden. Zusätzlich wird eine neue Methode eingeführt, um die beim Faktorisieren erfolgreiche Large Prime Variante auch beim DLP anwenden zu können. Bei der Beschreibung des Gleichungssystems am Schluß des Verfahrens wird hervorgehoben, wie die Lösung des DLP aus dessen Lösung gewonnen wird. Schliesslich werden vier Weltrekorde, die mit unserer Implementierung erzielt wurden, präsentiert, die ihren Höhepunkt in der Berechnung diskreter Logarithmen in einem Primkörper mit über 10^{85} Elementen finden. Weiterhin wird die Vorberechnungsphase zur McCurley Challenge beschrieben; hier besitzt der Primkörper 10^{129} Elemente.

Kapitel 4 stellt die erste allgemeine Siebimplementierung vor, die die aktuellen schnellsten DL– und Faktorisierungsalgorithmen, einen Algorithmus zum Umformen von DL–Problemen und einen neuen Algorithmus zur Berechnung von Klassengruppen algebraischer Zahlkörper als Spezialfälle enthält. Es werden einige Daten zur Verfügung gestellt, die zeigen, daß diese Verallgemeinerung nicht allzuviel an Effizienz kostet.

Introduction

> As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality.
>
> Albert Einstein

The subject of this thesis is to provide practical experience concerning the discrete logarithm problem (DLP) in the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$ of the finite prime field $\mathbb{Z}/p\mathbb{Z}$, p prime, by employing different index calculus algorithms. Evidently, the group in question contains $p-1$ elements. The discrete logarithm problem in this group may be stated as follows. Given integers a, b, and a prime number p find an integer x such that

$$a^x \equiv b \bmod p,$$

or prove that such a solution does not exist. Since the publication of the key exchange protocol by Diffie and Hellman in 1978 [19], the interest in the DLP has constantly been growing as can be seen in the invention of many cryptographic protocols whose security depends on the difficulty of the discrete logarithm in certain groups. The group $(\mathbb{Z}/p\mathbb{Z})^*$ is such a group in which the DLP is widely assumed to be difficult, provided that p is large and the group order $p-1$ contains a large prime factor. Consequently, numerous cryptographic protocols are designed for $(\mathbb{Z}/p\mathbb{Z})^*$; even the National Institute of Standards and Technology adopted a digital signature algorithm [49], which makes use of this group, based on proposals of ElGamal [21] and Schnorr [67]. The DLP in $(\mathbb{Z}/p\mathbb{Z})^*$ can, of course, be solved with algorithms which work in arbitrary groups, for example the one of Silver, Pohlig, and Hellman [55], which may be implemented by using either Shanks's method [63] or Pollard's method [57] as a subroutine. But the main disadvantage of this algorithm is that its running time is $O(\sqrt{q})$ if q largest prime divisor of $p-1$; this is exponential in the length of the input. Nevertheless, for small divisors q of $p-1$ it is possible to obtain the partial information $x \bmod q$ with this algorithm.

Because of the rising cryptographic significance [51], a lot of effort has been put into inventing discrete logarithm algorithms with a sub–exponential running time. As a result, we now know a couple of rigorous and heuristic methods for $(\mathbb{Z}/p\mathbb{Z})^*$, all based on the index calculus idea, which the reader will find in almost every survey article on the DLP [51, 38, 44, 52, 65]. As the basic idea of these algorithms has already appeared in integer factorization algorithms we naturally have the notion of running times introduced there. We define

$$L_p[s, c] := \exp((c + o(1)) (\log p)^s (\log \log p)^{1-s}).$$

In 1987, Pomerance proved in [60] that discrete logarithms in a finite field of p^n elements can be computed with an expected running time of $L_{p^n}[1/2, \sqrt{2}]$. The article of Coppersmith, Odlyzko, and Schroeppel [13] contains three heuristic versions exploiting the index calculus idea, with a conjectured expected running time of $L_p[1/2, 1]$. In 1993, a major breakthrough was achieved by Gordon who came up with an adaption of the Number Field Sieve (NFS) for factoring [27] by improving the (conjectured) expected running time to $L_p[1/3, 3^{\frac{2}{3}}]$. The latest acceleration of the NFS is due to Schirokauer [64] obtaining a conjectured, expected running time of

$$L_p[1/3, (64/9)^{\frac{1}{3}}].$$

This is the same asymptotic running time as for factoring composite integers of the same size as p.

Until 1995, however, the only serious attempt of a practical computation by implementing the method of [13] was made in 1991, when LaMacchia and Odlyzko showed the weakness of the Sun network–file–system cryptosystem by solving a DLP with a prime p with 58 decimal digits involved.

Our thesis is designed to provide substantial experimental data in order to show the practicability of the different heuristic index calculus versions by extending the method of [64] and by adapting another NFS version from factoring, and to give a comparison of the practical running times, which are hidden in the constants occuring in the expression $L_p[s, c]$. The following example gives an idea of how hard it is to invert the exponentiation function modulo a prime at present. Computing a power modulo a 85–digit prime can be done within 32 milliseconds on a 40–mips–machine. But it takes about one year computing time on the same machine to recover the exponent in question with the aid of the most practical discrete log algorithm for finite prime fields of that size.

Chapter one is intended to introduce the notation and the mathematical background needed to give a description of the considered algorithms, which are then roughly presented in chapter two.

Chapter three constitutes the main part of the work. It begins with a detailed description of the first NFS implementation for the DLP, including its numerous refinements and its four variations, and explains how to choose the parameters. Additionally, a new method is introduced to adapt the large prime variation from the factoring algorithms. In the linear algebra step at the end of the algorithm it is emphasized how the discrete logarithm solutions can be obtained with the help of the solution vectors belonging to the linear system. Finally, four world records, achieved with our implementation, are presented – culminating in computing logarithms in a prime field of over 10^{85} elements; furthermore, the precomputation step of attacking McCurley's 129–digit problem, which is stated as a challenge in [44], is described. Occasionally, we will refer to the analogous steps when the NFS is used for factoring integers.

In chapter four we show the first general sieving implementation which covers the fastest discrete logarithm and factoring algorithms known at present. Furthermore, it covers a special algorithm to transform discrete logarithm problems, and a new algorithm to compute class groups of number fields. We give some evidence that we pay for this generalization with only a slight lack of efficiency.

Contents

Chapter 1

Preliminaries

1.1 List of Frequently Used Symbols

At this point, we introduce some symbols which are not defined in the sequel. The reader is assumed to be familiar with most of them.

DL is an abbreviation for 'discrete logarithm'.

DLP is an abbreviation for 'discrete logarithm problem'.

NFS is an abbreviation for 'Number Field Sieve'.

$|M|$ means the cardinality of the set M.

$\mathbb{Z}$ is the set of the rational integers.

$\mathbb{N}$ is the set of natural numbers.

$\mathbb{N}_0$ is the set of natural numbers including 0.

$\mathbb{Q}$ is the set of rational numbers.

$\mathbb{R}$ is the set of real numbers.

$\mathbb{C}$ is the set of complex numbers.

$\mathbb{P}$ is the set of rational primes.

if R is a ring, then R^* means the ring of units of R.

$|a|$ ($a \in \mathbb{R}$) means the absolute value of a.

$\lfloor a \rfloor$ ($a \in \mathbb{R}$) means the greatest integer less than a.

$a|b$ ($a, b \in \mathbb{Z}$) means a divides b.

$a \not| b$ means a does not divide b.

$a \equiv b \bmod p$ means $p|(a-b)$.

$a \operatorname{Mod} b$ means the least non–negative residue when a is divided by b.

$\pi(n)$ means the number of primes up to n.

$\gcd(a, b)$ means the greatest common divisor of $a, b \in \mathbb{Z}$.

$\operatorname{ord}_p a$ ($p \in \mathbb{P}, a \in \mathbb{Z}$) is the exponent of p in the prime factorization of a.

$\operatorname{ord}_G a$ G a group, $a \in G$ means the order of a in G.

$\langle \cdot, \cdot \rangle$ means the standard scalar product.

When referring to computing times, we use the following abbreviations:

- *hsec* means 1/100 second,
- *mips* means 10^6 instructions per second,
- *1 mips year* is the computing time of one year, carried out by a computer rated at 1 mips.

1.2 Presentation of Algorithms

We present algorithms in a pseudo–code, which is related to the programming language C [30]. The LaTeX–style `algo.sty` is used, developed by Papanikolaou and Zayer [53]. They have replaced the bracketed control sequences of the language C by keywords, as follows.

The structure

```
if (condition)
   {
   instruction block 1
   }
   else
   {
   instruction block 1
   }
```

reads as

(1) **if** (condition) **then**
(2) instruction block 1
(3) **else**
(4) instruction block 2
(5) **fi**

The *while*–loop

```
while (condition)
      {
      instruction block
      }
```

is written as

(1) **while** (condition) **do**
(2) instruction block
(3) **od**

The *for*–loop

```
for (initialization; condition; increment)
      {
      instruction block
      }
```

is written as

(1) **for** (initialization; condition; increment) **do**
(2) instruction block
(3) **od**

As usual, the output of a function is indicated by the keyword **return**.

1.3 The Discrete Logarithm Problem

We start by introducing the term *discrete logarithm* for a finite group G.

1.1 Definition *Let $(G, \cdot)$ be a finite group and $a, b \in G$. If there exists $x \in \mathbb{Z}$, $x \geq 0$, such that*

$$a^x = b, \tag{1.1}$$

we call the minimal x satisfying (1.1) the discrete logarithm *of b to the base a. We write* $\log_a b$.

We will use this notation throughout the thesis. Unless stated otherwise, a will be the basis of the logarithm and b will be an element, the logarithm of which has to be computed. The definition tells us that by asking for the existence of x, the problem has to be solved, whether b is member of the subgroup generated by a. This problem is not easy in general but is trivial for a cyclic group G, provided the factorization of $|G|$ is known.

Evidently, if $y \in \mathbb{Z}$ is a solution to (1.1), then the set of all solutions to (1.1) is given by

$$\{y + k \cdot \operatorname{ord}_G a \mid k \in \mathbb{Z}\}.$$

As we shall see soon, the subgroups of G are of great importance for both the decision problem and the computational problem. The basic facts are given in the following Lemma.

1.2 Lemma *Let G be a finite cyclic group of order n with neutral element e and $t \in \mathbb{Z}$ be a divisor of n. Then*

i) $G^t := \{b^t \mid b \in G\}$ is a subgroup of G

ii)

$$a^{\frac{n}{t}} = e \Longleftrightarrow a \text{ is a } t\text{–th power in } G$$

Proof:

i) trivial.

ii) Let g be a generator of g. Choose $x \in \mathbb{Z}$, $x < n$, such that $g^x = a$. It follows $g^{xn/t} = e$. Therefore n divides xn/t whence x is divisible by t. Conversely, assume a is a t–th power in G, then $g^{kt} = a$ for some $k \in \mathbb{Z}$. We have

$$a^{n/t} = g^{ktn/t} = g^{kn} = e.$$

■

1.3 Lemma *Let $(G, \cdot)$ be a cyclic group with neutral element e,*

$$|G| = \prod_{i=1}^{n} p_i^{e_i}$$

be the prime factorization of its cardinality and $a, b \in G$. Then (1.1) is solvable if and only if for each j, $1 \leq j \leq n$

$$a^{|G|/p_j^k} \neq e,$$

whenever

$$b^{|G|/p_j^k} \neq e,$$

with $1 \leq k \leq e_j$.

Proof: Let g be a generator of G. We show that (1.1) is insoluble if there exist j, k with

$$b^{|G|/p_j^k} \neq e,$$

and

$$a^{|G|/p_j^k} = e.$$

Let U be the subgroup of p_j^k–th powers of G. According to Lemma 1.2 $a \in U$ and $b \notin U$. But U is closed under group operation. Hence the equation (1.1) has no solution.

Conversely, assume x is a solution to (1.1). Assume

$$b^{|G|/p_j^k} \neq e.$$

Since $b = a^x$, we have $a^{x|G|/p_j^k} \neq e$. Therefore

$$a^{|G|/p_j^k} = e$$

is impossible. ■

The criterion of the Lemma is sufficient for our purposes since the multiplicative group of prime fields is cyclic as we shall see in the next section (Theorem 1.7). For the use of the Number Field Sieve for computing discrete logarithms in these groups, the factorization of the group order is necessary; therefore we can assume that the prime factorization of $|G|$ is given.

The following two Lemmata show that it suffices to compute the solution of a discrete log problem modulo divisors of $|G|$.

1.4 Lemma *Let $(G, \cdot)$ be a cyclic group of order n with generator g. Let $a, b \in G$ and $t > 1$ be a divisor of n. Let $k, l \in \mathbb{Z}$, with $\gcd(t, l) = 1$, such that*

$$a^k b^l = d^t \tag{1.2}$$

for some $d \in G$.

Then

$$x \equiv -\frac{k}{l} \bmod t$$

is a solution to

$$\overline{a}^x = \overline{b}$$

in G/G^t.

Proof:

We have $t't + l'l = 1$ for some $t', l' \in \mathbb{Z}$. In particular, $l' \equiv l^{-1} \bmod t$. Then

$$d^{tl'} = (a^k b^l)^{l'} = a^{kl'} b^{ll'} = a^{kl'} b^{1-t't}. \tag{1.3}$$

Let

$$\overline{\cdot}: \; G \longrightarrow G/G^t$$

be the quotient map. It follows

$$\overline{1} = \overline{a}^{kl'}\overline{b},$$

giving

$$\overline{a}^{-kl'} = \overline{b},$$

which is the desired result. ■

1.5 Lemma *In the situation of Lemma 1.3, we have the following.*

If

$$\overline{a}^{x_j} = \overline{b} \quad \text{in } G/G^{p_j^{e_j}}, \quad 1 \le j \le n,$$

then for the unique non–negative $x < |G|$ which simultaneously solves

$$\begin{array}{ccccc} x & \equiv & x_1 & \mod & p_1^{e_1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x & \equiv & x_n & \mod & p_n^{e_n} \end{array}, \tag{1.4}$$

we have

$$a^x = b$$

in G.

Proof: We have the (homomorphic and surjective) quotient maps

$$\varphi_j : \; G \longrightarrow G/G^{p_j^{e_j}}, \quad 1 \le j \le n$$

with kernel $G^{p_j^{e_j}}$.

Define the map φ as follows

$$\begin{array}{rccc} \varphi : & G & \longrightarrow & G/G^{p_1^{e_1}} \times \cdots \times G/G^{p_1^{e_1}} \\ & h & \mapsto & (\varphi_1(h), \ldots, \varphi_n(h)). \end{array}$$

We are going to show that φ is an isomorphism of groups. The kernel of φ is

$$H = G^{p_1^{e_1}} \cap \ldots \cap G^{p_n^{e_n}}.$$

Let g be a generator of G and $h \in H$. Then $\varphi_j(g^x) = g^x G^{p_j^{e_j}}$, which is equal to $G^{p_j^{e_j}}$, if and only if $p_j^{e_j} | x$. So $g^x \in H$, if and only if $p_j^{e_j} | x$ for $1 \le j \le n$, whence $|G|$ divides x and $h = g^{|G|} = e$.

Having thus found a solution (1.4), applying φ^{-1} reveals x. ■

1.6 Remark The ability of computing discrete logarithms with respect to a particular base a is equivalent to the ability of computing discrete logarithms with respect to other bases a'. Assume, we want to solve

$$a'^x = b.$$

We compute y, z, such that $a^y = a'$ and $a^z = b$ and obtain

$$a^{xy} = a^z,$$

whence

$$xy \equiv z \bmod |G|.$$

1.4 Finite Prime Fields

In this section we intend to list the basic properties of finite fields, which are well–known and frequently used in the sequel. More details and proofs of the statements can be found in [29] or other books about algebra and number theory.

1.7 Theorem *Let K be a finite field of order q. Then q is a prime power and the multiplicative group of K is cyclic of order $q-1$. If K is a prime field (i.e. contains no proper subfields), then q is a prime number.*

The following lemma gives the method of representing elements of finite prime fields.

1.8 Lemma *For every $q = p^k$ for some $p \in \mathbb{P}$, $k \in \mathbb{N}$, there is a finite field K of order q. The integer multiples of the identity in K form a subfield K' isomorphic to $\mathbb{Z}/p\mathbb{Z}$.*

Given the situation of the preceding Lemma, we call K' *finite prime field* of order p. The algorithmic representation of an element of K'

$$\underbrace{1+1+\cdots+1}_{a} \in K' \tag{1.5}$$

is $a + p\mathbb{Z} \in \mathbb{Z}/p\mathbb{Z}$.

We always choose a to be the least non–negative integer satisfying (1.5).

The index calculus algorithms profit from the fact that on the assumption of the extended Riemann hypothesis (ERH), the representation of the least generator is relatively small. In this context, we know the following Theorem 1.9, which is proved in [71].

1.9 Theorem *(ERH) There exists a constant $c \in \mathbb{R}$ such that, for all $p \in \mathbb{P}$, there exists $a \in \mathbb{Z}$, $0 < a \leq c(\log p)^6$ such that a is generator of $(\mathbb{Z}/p\mathbb{Z})^*$.*

1.10 Remark In practice we usually observe a generator for $(\mathbb{Z}/p\mathbb{Z})^*$ among the first twenty primes.

1.5 Number Fields

In this section we recall some basic properties about algebraic number fields. Our considerations will be specialized on developing only the necessary facts needed for the subsequent chapters. For a few special cases, which are needed for the justification of some steps of the NFS algorithm and which are not to be found in the standard literature, proofs are given. For a more systematic approach we refer the reader to standard textbooks about algebra and algebraic number theory, for example [36, 12, 28]. We now define the structure, in which the main computation in the NFS algorithm takes place.

1.11 Definition *A* number field *is a field K containing $\mathbb{Q}$ which, considered as $\mathbb{Q}$–vector space, is finite dimensional. The number $n = \dim_{\mathbb{Q}} K$ is called the degree of the number field K.*

For the rest of this chapter let K be a number field of degree n. The possibility of representing elements of an algebraic number field, the so–called algebraic numbers, is given by the following Theorem.

1.12 Theorem *There exists $\theta \in K$, such that*

$$K = \mathbb{Q}(\theta) := \{\sum_{i=0}^{n-1} x_i \theta^i | (x_0, \ldots, x_{n-1}) \in \mathbb{Q}^n\}.$$

1.13 Remark In the situation of Theorem 1.12, we call the set

$$\{1, \theta, \theta^2, \ldots, \theta^{n-1}\}$$

standard basis of $Q(\theta)$.

1.14 Lemma *Let $\alpha \in K$. There exists $f \in \mathbb{Z}[X]$, $k := \deg f \leq n$*

$$f(X) = a_k X^k + \cdots + a_1 X + a_0,$$

with $f(\alpha) = 0$. If f has minimal degree and $\gcd(a_0, \ldots, a_n) = 1$, we call f the minimal polynomial *of α.*

We introduce algebraic integers as a generalization of rational integers.

1.15 Definition *Let $\omega \in K$. If the minimal polynomial of ω is of the form*

$$X^k + a_{k-1} X^{k-1} + \cdots + a_1 X + a_0,$$

we call ω an algebraic integer*.*

1.16 Theorem *The set*

$$\mathcal{O}_K := \{\omega \in K \mid \omega \text{ is an algebraic integer}\}$$

forms a ring.

1.17 Definition *We call $\mathcal{O}_K$ the* ring of integers *of K. If the context is clear, we simply write $\mathcal{O}$ instead of $\mathcal{O}_K$.*

1.18 Remark For ω being an algebraic integer, we also write $\mathcal{O}_\omega$ for the ring of integers of $\mathbb{Q}(\omega)$.

The norm map, which will be defined next, is a useful tool to transform problems in K into problems in $\mathbb{Q}$.

1.19 Definition *Let $\alpha \in K$. Let ν be the following map:*

$$\begin{array}{rccc} \nu_\alpha : & K & \longrightarrow & K \\ & \beta & \mapsto & \alpha\beta. \end{array}$$

We call $N(\alpha) := \det(\nu_\alpha)$ the norm *of α.*

1.20 Lemma *The norm $N(\alpha)$ is a rational number. It is a rational integer, provided that α is an algebraic integer. $N(\alpha) = 1$ if and only if α is a unit in $\mathcal{O}$.*

1.21 Definition *Let ω be an algebraic integer with minimal polynomial*

$$f(X) = X^k + a_{k-1}X^{k-1} + \cdots + a_1X + a_0.$$

The value

$$\Delta_f := N(f'(\omega)) := k\omega^{k-1} + (k-1)a_{k-1}\omega^{k-2} + \cdots + a_1$$

is called the discriminant *of f.*

We now derive the formula which will be used to compute norms in the NFS algorithm.

1.22 Lemma *Let $K = \mathbb{Q}(\alpha)$ be a number field of degree $n \geq 2$. Let*

$$f(X) := a_nX^n + \cdots + a_1X + a_0$$

be the minimal polynomial of α. Then the norm of $c + d\alpha$, $c, d \in \mathbb{Q}$ can be computed from

$$N(c + d\alpha) = (-d)^n \frac{f(-\frac{c}{d})}{a_n}.$$

Proof: We compute the norm of $c - d\alpha$ by constructing the representation matrix $A_{c-d\alpha} \in \mathbb{Q}^{n\times n}$ of the map $\nu_{c-d\alpha}$. Note that

$$\alpha^n = -\frac{a_{n-1}}{a_n}\alpha^{n-1} - \cdots - \frac{a_1}{a_n}\alpha - \frac{a_0}{a_n}.$$

After having constructed the matrix of the images of the standard basis of $\mathbb{Q}(\alpha)$ under $\nu_{c-d\alpha}$, we compute its determinant

$$\begin{vmatrix} c & 0 & \dots & \dots & \dots & 0 & d\frac{a_0}{a_n} \\ -d & c & 0 & \dots & \dots & 0 & d\frac{a_1}{a_n} \\ 0 & -d & c & 0 & \dots & 0 & d\frac{a_2}{a_n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & \dots & 0 & -d & c + d\frac{a_{n-1}}{a_n} \end{vmatrix}. \tag{1.6}$$

In order to compute a triangular form of the matrix above, we perform the following operation in step i ($1 \leq i \leq n-1$): we multiply row i by $\frac{d}{c}$ and add the result to row $i+1$. This is clearly a transformation which keeps the value of the determinant invariant. The effect of the operation $i = 1$ is as follows. The first column will only

consist of the entry c in the upper left corner. The second entry in the last column will be replaced by

$$\frac{d^2a_0}{ca_n} + d\frac{a_1}{a_n}.$$

By iterating, after $n-2$ transformations, the entry $n-1$ will show up as

$$\frac{d^{n-1}a_0}{c^{n-2}a_n} + \frac{d^{n-2}a_1}{c^{n-3}a_n} + \cdots + \frac{da_{n-2}}{a_n}.$$

Since the last entry in the last column is $c + d\frac{a_{n-1}}{a_n}$, this entry is replaced by

$$h(c,d) := \frac{d^na_0}{c^{n-1}a_n} + \frac{d^{n-1}a_1}{c^{n-2}a_n} + \cdots + \frac{d^2a_{n-2}}{ca_n} + d\frac{a_{n-1}}{a_n} + c$$

in step $n-1$.

After step $n-1$, the matrix has the following form:

$$\begin{vmatrix} c & 0 & \dots & \dots & \dots & 0 & d\frac{a_0}{a_n} \\ 0 & c & 0 & \dots & \dots & 0 & * \\ 0 & 0 & c & 0 & \dots & 0 & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & \dots & 0 & 0 & h(c,d) \end{vmatrix}.$$

It is clear that the determinant is the product of the diagonal entries. So we have

$$\begin{aligned} N(c-d\alpha) &= c^{n-1}h(c,d) \\ &= \frac{d^na_0}{a_n} + \frac{d^{n-1}ca_1}{a_n} + \cdots + \frac{d^2c^{n-2}a_{n-2}}{a_n} + dc^{n-1}\frac{a_{n-1}}{a_n} + c^n \\ &= d^n\frac{f(\frac{c}{d})}{a_n}. \end{aligned}$$

The result follows by replacing d by $-d$. ■

1.23 Remark We consider two special cases. Firstly, $|N(\alpha)| = |f(0)/a_n|$. Secondly, the norm of an element $c \in \mathbb{Q}$ is $N(c) = c^n$ (set $d = 0$ in (1.6)).

1.24 Definition *Let R be a ring. A non–empty subset $I \subset R$ is an* ideal *of R if and only if the following two conditions are satisfied:*

1. $a, b \in I \Longrightarrow a - b \in I$; *and*

2. $r \in R$, $a \in I \Longrightarrow ra \in I$.

In order to define the norm of an ideal we first note the following Lemma.

1.25 Lemma *Let K be an algebraic number field with ring of integers $\mathcal{O}$ and let I be an ideal of $\mathcal{O}$. Then $\mathcal{O}/I$ is a finite ring.*

1.26 Definition *In the situation of the lemma above, we call*

$$N(I) := |\mathcal{O}/I|$$

the norm *of the ideal I. In case of $\mathcal{O}/I$ being a finite field, we will say that I is a prime ideal of $\mathcal{O}$. Then $|\mathcal{O}/I| = p^m$ for some natural number m. We call m the degree of the prime ideal I.*

Ideals can be decomposed (multiplicatively) into prime ideals. Thus we give the multiplication operation.

1.27 Definition *Let K be an algebraic number field with ring of integers $\mathcal{O}$ and let I, J be ideals of $\mathcal{O}$. Then the* ideal product *of I and J is defined as*

$$IJ := \{\sum_{i=1}^{k} x_i y_i | k \in \mathbb{N}, x_i \in I, y_i \in J, 1 \leq i \leq k\}.$$

1.28 Definition *Let K be an algebraic number field with ring of integers $\mathcal{O}$. A* fractional ideal *I in $\mathcal{O}$ is a non–zero submodule of K such that there exists a non–zero integer d with dI ideal of $\mathcal{O}$. An ideal I of $\mathcal{O}$ is said to be a* principal ideal *if there exists $x \in \mathcal{O}$ such that $x\mathcal{O} = I$. Finally, $\mathcal{O}$ is said to be a principal ideal domain if every ideal of $\mathcal{O}$ is a principal ideal.*

The following Theorem is fundamental to the functioning of the NFS algorithm.

1.29 Theorem *Given the situation of definition 1.28. Every fractional ideal I in $\mathcal{O}$ can be written in a unique way as*

$$I = \prod_{\mathfrak{p}} \mathfrak{p}^{e_\mathfrak{p}},$$

the product being over a finite set of prime ideals and the exponents being in $\mathbb{Z}$. In particular $I \subset \mathcal{O}$ if and only if all the $e_\mathfrak{p}$ are non–negative.

1.30 Remark As to Theorem 1.29, we write

$$ord_{\mathfrak{p}} I := e_{\mathfrak{p}}.$$

If $\xi \in \mathcal{O}$, $I = \xi\mathcal{O}$, we write

$$ord_{\mathfrak{p}} \xi := e_{\mathfrak{p}}.$$

1.31 Remark Let $K := \mathbb{Q}(\omega)$ be defined by fixing a root ω of a monic polynomial $f \in \mathbb{Z}[X]$. Then $(\mathbb{Z}[\omega], +)$ has finite group index in $\mathcal{O}$. We will call the subset of $p \in \mathbb{P}$, which divides this index, *index divisors*. If the index is not divisible by $p \in \mathbb{P}$, we say that $\mathbb{Z}[\omega]$ is p–maximal.

In order to compute a superset of the index divisors, we note

1.32 Lemma *Let $K = \mathbb{Q}(\omega)$ be a number field, $f(X) \in \mathbb{Z}[X]$ the monic minimal polynomial of ω, $p \in \mathbb{P}$. If p is divisor of the index $[\mathcal{O}_\omega : \mathbb{Z}[\omega]]$ then $p^2 | \Delta_f$.*

1.33 Remark In particular, the set of index divisors is finite.

In order to actually recognize the index divisors, we apply the following theorem due to Dedekind[1] (Dedekind's criterion).

1.34 Theorem *Let $K = \mathbb{Q}(\omega)$ be a number field, $f(X) \in \mathbb{Z}[X]$ the monic minimal polynomial of ω, $p \in \mathbb{P}$. Let*

$$f(X) \equiv \Pi_{i=1}^{k} g_i(X)^i \bmod p$$

be the factorization of f modulo p, where g_i is the product of monic irreducible polynomials dividing f exactly with exponent i. Define

$$h(X) := \frac{1}{p}(f(X) - \prod_{i=1}^{k} g_i(X)^i).$$

Then $\mathbb{Z}[\omega]$ is p–maximal if and only if

$$\gcd(h(X), \prod_{i=2}^{k} g_i(X)) = 1.$$

[1] Richard Dedekind 1831–1916

The following Theorem gives the prime ideal decomposition of a rational prime p from the factorization of the field polynomial modulo p, for all but finitely many p.

1.35 Theorem *Let $K := \mathbb{Q}(\omega)$ be a number field, where ω is an algebraic integer, whose (monic) minimal polynomial is denoted $T(X)$. Let f be the index of ω, i.e. $f = [\mathcal{O}_\omega : \mathbb{Z}[\omega]]$. Then for any prime not dividing f one can obtain the prime decomposition of $p\mathcal{O}_\omega$ as follows. Let*

$$T(X) \equiv \prod_{i=1}^{g} T_i(X)^{e_i} \bmod p$$

be the decomposition of T into irreducible factors in $\mathbb{Z}/p\mathbb{Z}[X]$, where the T_i are taken to be monic. Then

$$p\mathcal{O}_\omega = \prod_{i=1}^{g} \mathfrak{p}_i^{e_i},$$

where

$$\mathfrak{p}_i = (p, T_i(\omega)) = p\mathcal{O}_\omega + T_i(\omega)\mathcal{O}_\omega.$$

1.36 Remark As to Theorem 1.35, we say that $\mathfrak{p}_i$ lies above p.

Referring to the NFS, we start with choosing a polynomial, take a root α of this polynomial and then perform computations in $\mathbb{Z}[\alpha]$. If α is an algebraic integer, it is clear that $\mathbb{Z}[\alpha] \subset \mathcal{O}_\alpha$. In case of α not being an algebraic integer, some adjustments have to be made. In particular, an algebraic integer related to α is constructed. The following Lemma collects some facts about this case.

1.37 Lemma *Let $f(X) \in \mathbb{Z}[X]$ irreducible,*

$$f(X) = a_n X^n + a_{n-1}X^{n-1} + \cdots + a_0, \quad a_n \neq 0$$

and define

$$\begin{aligned} g(X) &:= a_n^{n-1} f(X/a_n) \\ &= X^n + a_{n-1}X^{n-1} + a_n a_{n-2}X^{n-2} + a_n^2 a_{n-3}X^{n-3} + \cdots + a_n^{n-1}a_0. \end{aligned} \tag{1.7}$$

Let $\alpha \in \mathbb{C}$, such that $f(\alpha) = 0$, $\omega := a_n\alpha$. Then

i) $g(\omega) = 0$

ii) ω *is an algebraic integer*

iii) $\mathbb{Q}(\alpha) = \mathbb{Q}(\omega)$

iv) if $p \nmid a_n$ *and* p *is not index divisor of* $[\mathcal{O}_\omega : \mathbb{Z}[\omega]]$, *there is a bijection between the prime ideals of* $\mathcal{O}_\omega$ *lying above* p *and the irreducible factors of* $f(X) \bmod p$.

v) if $n \geq 3$, *every prime* p *dividing* a_n *is an index divisor of* $[\mathcal{O}_\omega : \mathbb{Z}[\omega]]$.

Proof:

i) $g(\omega) = a_n^{n-1} f(\omega/a_n) = a_n^{n-1} f(\alpha) = 0$

ii) ω is root of the monic polynomial g

iii) clear, because $a_n \in \mathbb{Q}$

iv) There is a bijection between the prime ideals of $\mathcal{O}_\omega$ lying above p and the irreducible factors of $g(X) \bmod p$, by Theorem 1.35. Let the polynomial factorization of $g(X)$ modulo p be given as

$$g(X) \equiv \prod g_i(X)^{e_i} \bmod p.$$

Then

$$f(X/a_n) \equiv \frac{1}{a_n^{n-1}} g(X) \equiv \frac{1}{a_n^{n-1}} \prod g_i(X)^{e_i} \bmod p.$$

The result now follows by substituting $Y := X/a_n$ and applying Theorem 1.35.

v) We apply Dedekind's criterion. Assume $p \nmid a_{n-1}$. From (1.7) we see that

$$g(X) \equiv X^{n-1}(X + a_{n-1}) \bmod p.$$

Consequently, in terms of Theorem 1.35, $g_1(X) = X + a_{n-1}$, $g_{n-1}(X) = X$ and

$$h(X) = -\frac{1}{p}(a_{n-2}a_n X^{n-2} + \cdots + a_0 a_n^{n-1}).$$

But p divides a_n and therefore divides $a_0 a_n^{n-1}/p$, if $n \geq 3$, whence

$$\gcd(g_{n-1}, h) \equiv X \bmod p.$$

Therefore $\mathbb{Z}[\omega]$ is not p–maximal, so p is index divisor. Now assume $p | a_{n-1}$. Then g factors as

$$g(X) \equiv X^n \bmod p,$$

and $g_n(X) = X$. We obtain

$$h(X) = -\frac{1}{p}(a_{n-1}X^{n-1} + a_{n-2}a_n X^{n-2} + \cdots + a_0 a_n^{n-1}).$$

As above, p divides the constant term of h, and $\mathbb{Z}[\omega]$ is not p–maximal, so p is again an index divisor.

■

We note that the norm map is extended to ideals in a natural way.

1.38 Lemma *Given the situation of definition 1.28. For $\beta \in \mathcal{O}$, the following equality holds:*

$$|N(\beta)| = N(\beta\mathcal{O}).$$

The following lemma is needed for determining which prime ideals occur in the decomposition of $c + d\alpha$.

1.39 Lemma *Let ω be an algebraic integer and let $c+d\omega \in \mathcal{O}_\omega$, $c, d \in \mathbb{Z}$ be coprime and $\mathfrak{p} = (p, \omega - c_p)$ be a prime ideal of norm $p \in \mathbb{P}$ not dividing the index $[\mathcal{O}_\omega : \mathbb{Z}[\omega]]$. Then $\mathfrak{p}$ divides the principal ideal $(c + d\omega)$ if and only if*

- *$N(c + d\omega)$ is divisible by p, and*
- *$-c/d \equiv c_p \bmod p$.*

The exponent of $\mathfrak{p}$ in the prime ideal factorization of $(c+d\omega)$ is equal to the exponent of p in the prime factorization of $N(c + d\omega)$.

The correctness of the NFS algorithm for DL depends on a condition which the class number of the employed number field has to satisfy. We first note the following result.

1.40 Theorem *Given the situation in definition 1.28 and let $\mathcal{I}$ be the set of fractional ideals of $\mathcal{O}$. Let $\mathcal{P} \subset \mathcal{I}$ be the set of principal fractional ideals of $\mathcal{O}$. Then $\mathcal{I}$ is an abelian group with respect to multiplication and $Cl_K := \mathcal{I}/\mathcal{P}$ is a finite group.*

Now we are ready to introduce the class number of a number field.

1.41 Definition *The group Cl_K of Theorem 1.40 is called the* class group *of K and its cardinality h_K is called the* class number *of K.*

1.6 Miscellaneous

In order to measure the quality of a decomposition of integers into a product of smaller integers, we need a notion of *smoothness*.

1.42 Definition *Let $n, t \in \mathbb{Z}$. The integer n is said to be* t–smooth *or* smooth with respect to t*, if all prime divisors of n are at most t.*

To estimate the likelihood of smoothness, which is essential in the index calculus algorithms, we use the ρ–functions, which are examined in an article of Knuth and Trabb Pardo [31].

For $n, k \in \mathbb{N}$ let $P_k(x, n)$ be the number of positive integers less than n, whose k–th largest prime divisor is at most n^x. Knuth and Trabb Pardo proved that

$$F_k(x) := \lim_{n\to\infty} \frac{P_k(x,n)}{n}$$

exists.

1.43 Definition *With the notation of the preceding paragraph, we define*

$$\rho_k(x) := F_k(1/x).$$

As a special case, we obtain an approximation of the probability of n being $n^{1/x}$–smooth by evaluating $\rho_1(x)$.

Knuth and Trabb Pardo give the following formulae for $\rho_k(x)$, which are convenient for numerical integration:

$$\rho_k(x) = \begin{cases} 0 & \iff \quad x \le 0 \vee k = 0 \\ 1 & \iff \quad 0 < x \le 1, k \ge 1 \\ 1 - \int_1^x (\rho_k(t-1) - \rho_{k-1}(t-1))\frac{dt}{t} & \iff \quad x > 1, k \ge 1. \end{cases}$$

Chapter 2

Discrete Logarithm Algorithms

This chapter is a survey on the ideas of the current significant discrete logarithm algorithms for finite prime fields, which we are going to discuss in this thesis: the Pohlig–Hellman–Algorithm [55], the Coppersmith–Odlyzko–Schroeppel–Algorithm [13], and the Number Field Sieve Algorithm [27, 64]. As stated in section 1.3, we consider

$$a^x \equiv b \bmod p, \qquad a, b \in \mathbb{Z},\ p \in \mathbb{P}.$$

2.1 The Pohlig–Hellman–Algorithm

The Pohlig–Hellman–Algorithm (PH) actually works in any cyclic group. Unfortunately, it has an exponential running time in terms of the group order. Let $q_{\max}$ be the greatest prime divisor of $|G|$, then PH needs $O(\sqrt{q_{\max}})$ group operations. In its original version, PH also needs space for storing $O(\sqrt{q_{\max}})$ group elements, which for increasing $q_{\max}$ rapidly exhausts main memory.

2.1 Example Assume $p \approx 2 \cdot 10^{14}$ with $(p-1)/2$ prime. Then the representation of a number in $\mathbb{Z}/p\mathbb{Z}$ consumes 7 digits to the base 2^{32}, plus 32 bit length and sign information, that is 32 bytes in total. The original PH stores $\sqrt{q}$ numbers in $\mathbb{Z}/p\mathbb{Z}$; the amount of main memory needed is therefore

$$32 \cdot 10^7 \text{ bytes} = 320 \text{ MB}.$$

Fortunately, PH can be combined with an idea of Pollard [57] such that its space requirements are bounded by a small constant. Now we will sketch how the combination of PH and Pollard's method works.

Let q be any prime dividing $p-1$ and h an integer with $h \geq 1$. Let q^h be a power of q dividing $p-1$; we are going to compute x modulo q^h provided that we know the value of x modulo q^{h-1}. Assume

$$x \equiv x_0 + x_1 q + x_2 q^2 + \cdots + x_{h-2} q^{h-2} \bmod q^{h-1}.$$

Set

$$\begin{aligned} a' &\equiv a^{(p-1)/q} \bmod p, \\ b' &\equiv \left(\frac{b}{a^{x_0 + x_1 q + x_2 q^2 + \cdots + x_{h-2} q^{h-2}}} \right)^{(p-1)/q^h} \bmod p. \end{aligned}$$

Because of

$$\begin{aligned} a'^q &\equiv 1 \bmod p \quad \text{and} \\ b'^q &\equiv 1 \bmod p \end{aligned}$$

both a' and b' are members of the unique subgroup U_q of order q in $(\mathbb{Z}/p\mathbb{Z})^*$.

Solving

$$a'^{x_{h-1}} \equiv b' \bmod p \quad \text{for } x_{h-1},$$

we obtain

$$a'^{x_{h-1}} \equiv \left(\frac{b}{a^{x_0 + x_1 q + x_2 q^2 + \cdots + x_{h-2} q^{h-2}}} \right)^{(p-1)/q^h} \bmod p.$$

With

$$a'^{x_{h-1}} \equiv (a^{x_{h-1}})^{\frac{p-1}{q}} \equiv a^{q^{h-1} x_{h-1} \frac{p-1}{q^h}} \bmod p,$$

it follows that

$$\left(a^{x_0 + x_1 q + x_2 q^2 + \cdots + x_{h-1} q^{h-1}} \right)^{\frac{p-1}{q^h}} \equiv b^{\frac{p-1}{q^h}} \bmod p,$$

which is equivalent to

$$x \equiv x_0 + x_1 q + x_2 q^2 + \cdots + x_{h-2} q^{h-2} + x_{h-1} q^{h-1} \bmod q^h.$$

The main idea for computing x_{h-1} is to produce iteratively a sequence of elements $(d_i)_{i \geq 1}$, where all the d_i's are of the form

$$a'^k b'^l \bmod p.$$

As U_q contains exactly q elements, (d_i) will become periodic after at most q iterations. This, however, can be expected to happen in an expected number of $O(\sqrt{q})$ steps (birthday paradoxon). The computation stops when $d_{i'} \equiv d_i \bmod p$ for $i' > i$ is recognized. In this case,

$$a'^{k-k'} b'^{l-l'} \equiv 1 \bmod p$$

and therefore

$$(k - k' + x_{h-1}(l - l')) \equiv 0 \bmod q,$$

which reveals x_{h-1}, provided that $\gcd(l - l', q) = 1$.

Pollard's method finds the pair (i, i') by computing the sequence twice as (d_j) and (d_{2j}), waiting for $(d_j) = (d_{2j})$. This will be the case when j is a positive multiple of the period length. Pollard's ϱ–method for factoring works analogously. In our implementation, we were able to improve the running time by adapting Brent's improvement [7] concerning Pollard's ϱ–method to the discrete logarithm case. Instead of computing the sequence twice, Brent's algorithm remembers the sequence elements d_{2^i} and compares them to d_{2^i+j}, $1 \leq j \leq 2^i$. It can be shown that comparison is not needed for $1 \leq j \leq 3 \cdot 2^{i-1}$.

By combining PH and the two improvements, we got the running times shown in table 2.1 on a Sparc ELC workstation. The 22–digit example (*) has been computed on a Sparc Ultra workstation. The number of digits refers to the decimal digits of p, where p is a prime such that $(p-1)/2$ is a prime of the same size.

Table 2.1: Running times Pohlig–Hellman–Pollard–Brent

# digits	CPU min:sec	# examples	factor
10	0:08	250	
11	0:27	250	3.38
12	1:31	250	3.37
13	4:38	250	3.05
14	13:36	250	2.93
15	42:32	250	3.13
16	142:43	150	3.36
17	345:16	65	2.42
18	1006:56	5	2.92
19	4131:50	1	4.10
20	8886:09	1	2.15
21	10718:36	1	1.21
22*	10195:36	1	

As one would expect, the running time increases by a factor of magnitude $\sqrt{10} \approx 3.16$ when p is enlarged by one decimal digit. Extrapolating this from the surprisingly good running time for the 21–digit number, the running time for an 85–digit prime would be $2 \cdot 10^{29}$ years in practice.
Fortunately, we can do better due to the practicability of index calculus methods, the idea of which is sketched within the next section.

2.2 The Number Field Sieve

The Number Field Sieve (NFS) was eminently successful in attacking the problem of factoring integers. Many people have contributed to this algorithm, see for example [58, 14, 23, 24, 48, 78, 37]. The NFS factoring algorithm is closely related to the NFS DL algorithm. On the one hand, some steps are similar, one the other hand, some steps are completely new in the latter. With regard to the handling of the similar steps, we will comment on the difference to the factoring case. In the following, we will briefly explain the idea of the Number Field Sieve to compute discrete logarithms, as we will give a detailed description in the next chapter. The Gaussian–Integer–Method[1], published by Coppersmith, Odlyzko and Schroeppel (COS) may be viewed as a special case of the NFS; we will outline this in the next chapter.

The NFS consists of the following steps:

1. reduce original problem (1.1) to congruences
$$a^x \equiv s \bmod p,$$
$s \in S$ where S is a set of "sufficiently" small natural numbers

2. choose two polynomials $g_1(X), g_2(X) \in \mathbb{Z}[X]$ of degree n_1, n_2 respectively with common root m mod p;
for $j = 1, 2$:
   - let $h_j \in \mathbb{Z}$ be the coefficient of X^{n_j} in polynomial g_j,
   - let $\alpha_j \in \mathbb{C}$ be a root of g_j
   - let $\mathcal{O}_j \supset Z[h_j\alpha_j]$ be the ring of integers of $K_j := \mathbb{Q}(\alpha_j)$

3. choose factor bases
$F_j = \{\text{prime ideals of } \mathcal{O}_j \text{ with norm below some bound } B_j\} \cup \{h_j\}$

4. find set of pairs $C := \{(c, d)\} \subset \mathbb{Z} \times \mathbb{Z}$ with
$h_1 \cdot (c + d\alpha_1)$ smooth over F_1
$h_2 \cdot (c + d\alpha_2)$ smooth over F_2 by sieving, with $|C| > |F_1| + |F_2|$

5. for each $s \in S$ find special relations:
$h_1 \cdot (c + d\alpha_1)/\mathfrak{p}_s$ smooth over F_1
$h_2 \cdot (c + d\alpha_2)$ smooth over F_2
for each prime ideal $\mathfrak{p}_s$ of $\mathcal{O}_1$ lying above s

[1] Carl Friedrich Gauß 1777–1855

6. for every big prime divisor q dividing $p-1$:

   - compute additive characters of $h_j \cdot (c + d\alpha_j)$
   - matrix A over $\mathbb{Z}/q\mathbb{Z}$
     $\dim A \approx |F_1| + |F_2|$
     A consists of exponent vectors in the decomposition of the $h_j \cdot (c + d\alpha_j)$ and the additive characters
   - compute elements $\gamma_1 \in \mathcal{O}_1$, $\gamma_2 \in \mathcal{O}_2$, which are q–th powers by solving $Ax \equiv 0 \bmod q$
   - obtain k, l with $a^k b^l \bmod p$ being a q–th power in $(\mathbb{Z}/p\mathbb{Z})^*$
   - compute x modulo q by applying Lemma 1.4

From Theorem 1.9 and Remark 1.6 we may assume that a is smooth with respect to either B_1 or B_2.

The basic idea of using auxiliary number rings $\mathcal{O}_1$, $\mathcal{O}_2$ containing $\mathbb{Z}[h_1\alpha_1]$, $\mathbb{Z}[h_2\alpha_2]$ respectively is that the number rings are constructed in such a way that

$$\begin{array}{rccc} \varphi_j : & \mathbb{Z}[h_j\alpha_j] & \longrightarrow & \mathbb{Z}/p\mathbb{Z} \\ & h_j\alpha_j & \mapsto & h_j m \end{array} \qquad j = 1, 2$$

are ring homomorphisms.

In particular, if we construct a q–th power $\gamma_j \in \mathbb{Z}[h_j\alpha_j]$, then $\varphi_j(\gamma_j)$ is a q–th power in $(\mathbb{Z}/p\mathbb{Z})^*$. During the execution of the algorithm, we obtain the prime ideal factorization of many principal ideals. Merely given these factorizations, the linear algebra step would compute an element

$$\gamma \in V := \{\xi \in \mathcal{O} \mid q \,\text{divides}\, \mathrm{ord}_{\mathfrak{p}}(\xi), \text{ for all prime ideals } \mathfrak{p} \subset \mathcal{O}\}.$$

Although the group V contains the subgroup of q-th powers of K, it is not equal to $(K^*)^q$ in general. The quotient $V/(K^*)^q$ may be viewed as an obstruction group. It is finite in the Gaussian integer method but not in the general Number Field Sieve. This is the reason for employing the additive characters computed in step 6. we will have a close look at that in section 3.6.

For the sake of convenience we introduce some terms to be used when talking about *relations* which are produced by the NFS algorithm. For each f_j $(j = 1, 2)$, choose $L_j \in \mathbb{Z}$, a bound called the *large prime bound* for f_j. A relation is a pair (c, d), where $c + d\alpha_1$ and $c + d\alpha_2$ satisfy the following property:

The prime ideal decomposition of the two ideals $(h_1 \cdot (c + d\alpha_1))$, $(h_2 \cdot (c + d\alpha_2))$ may be expressed as

$$\begin{array}{rcl} h_1 \cdot (c + d\alpha_1) & = & \mathfrak{q}_1\mathfrak{q}_2 \prod_{\mathfrak{p}\in F_1} \mathfrak{p}^{e_\mathfrak{p}} \\ h_2 \cdot (c + d\alpha_2) & = & \mathfrak{q}_1'\mathfrak{q}_2' \prod_{\mathfrak{p}'\in F_2} \mathfrak{p}'^{e_{\mathfrak{p}'}}, \end{array} \tag{2.1}$$

where $N(\mathfrak{q}_1) \leq N(\mathfrak{q}_2) \leq L_1$, $N(\mathfrak{q}_1') \leq N(\mathfrak{q}_2') \leq L_2$.

The *exponent vector* of the relation is defined as the vector consisting of all exponents in the decomposition of $(c+d\alpha_1)$, $(c+d\alpha_2)$, including the exponent 1 of all occuring large primes.

2.2 Remark Allowing non–monic polynomials, i.e. $h_1 \neq 1$ and $h_2 \neq 1$, causes $h_1 \cdot (c + d\alpha_1)$ and $h_2 \cdot (c + d\alpha_2)$ to be mapped to different elements:

$$\begin{array}{rcl} \varphi_1(h_1 \cdot (c + d\alpha_1)) & = & h_1(c + dm) \\ \varphi_2(h_2 \cdot (c + d\alpha_2)) & = & h_2(c + dm). \end{array}$$

In order to get rid of the factors h_1, h_2, we enforce them to show up as q–th powers, too. This is achieved by storing their common exponent – which is always 1 – in an extra column of A.

We call a relation *full, single, double, triple, quadruple*, or *quintuple* relation according to whether it contains no, one, two, three, four, or five large primes. A relation containing at least one large prime will also be called *partial* relation. A relation with the condition $c \in \mathbb{P}$, $d = 0$ is called *free* relation since it can be obtained from the construction of the factor bases.

Chapter 3

The Number Field Sieve

This chapter contains a close examination of the NFS steps sketched at the end of the previous chapter. So we stick to the notation introduced there. To illuminate the steps of the NFS and its different variations, we now introduce an example, which we will use throughout the chapter. The continuation of this example will be marked with **DL–Example**.

3.1 DL–Example We start by picking the first prime $p > 10^4$, with $(p-1)/2$ prime. Having thus found $p = 10007 = 2 \cdot 5003 + 1$, we start looking for a generator of the group $(\mathbb{Z}/p\mathbb{Z})^*$. The first element $a \in \mathbb{N}$ with $a^{5003} \equiv -1 \bmod p$ is $a = 5$. For b, we choose the prime $b = 5039$.
Our example DL problem then looks as follows:

$$5^x \equiv 5039 \bmod 10007.$$

3.1 The Reduction Step

This section is devoted to step 1 of the survey at the end of the previous chapter. We are interested in reducing the original task of solving

$$a^x \equiv b \bmod p$$

to several tasks

$$a^{x_i} \equiv s_i \bmod p, \quad s_i \in \mathbb{Z},$$

where the s_i's are relatively small integers.

Let s be one of the s_i's. Such a reduction is necessary because in the NFS step 4, the simultaneous smoothness of the terms

$$(c + d\alpha_1)/\mathfrak{p}_s \quad \text{and} \quad (c + d\alpha_2), \tag{3.1}$$

is required. Here, $\mathfrak{p}_s$ is a prime ideal lying over s in $\mathcal{O}_1$.

Let the prime ideal $\mathfrak{p}_s$ be generated by $(s, \alpha_1 - r)$ over $\mathcal{O}_1$, $r \in \mathbb{Z}$. Then $\mathfrak{p}_s$ divides $(c + d\alpha_1)$ if and only if $-c/d \equiv r \bmod s$, by Lemma 1.39. So we expect either c or d to be of size s. Therefore the difficulty in finding a relation of type 3.1 raises with the size of s. The subject of this section is a new sieving method, which can be used to minimize the maximal value of s for a given DL problem.

In their implementation [35], LaMacchia and Odlyzko transformed the DL problem as follows:

- compute $c_l :\equiv a^l \cdot b \bmod p$ for many different l,
- express
$$c_l \equiv t_l/u_l \bmod p, \quad t_l, u_l \approx \sqrt{p}, \tag{3.2}$$
by applying the extended Euclidean algorithm to c_l and p,
- compute the factorization of t_l, u_l, hoping for smoothness of both terms.

This way, they broke the challenge of the Sun cryptosystem. The transformation above worked well because the prime in the Sun challenge consisted of only 58 decimal digits. As a consequence, they had to find 10^{10}–smooth numbers among numbers of size 10^{29}. Such numbers show up quite frequently (6.0%). So simultaneous smoothness can be expected to happen after 400 trials.

For a prime of 75 or 85 digits, however, this method is not advisable because the probability that a 43–digit number is 10^9–smooth is only 0.065 %. As we need simultaneous smoothness of two numbers, we expect to make at least $2 \cdot 10^6$ trialdivision steps before we encounter an appropriate pair (t_l, u_l).

For this reason, we propose a sieving method like the residue list sieve to perform this task. The residue list sieve is outlined in [13]. We present a variation of it, which is appropriate to our situation. For a number t of size $\sqrt{p}$ as in (3.2), we find a B–smooth representation of t as follows. Without loss of generality, we may assume $\sqrt{p}/2 \leq t \leq \sqrt{p}$. The lower bound is not really a restriction since we can

multiply t by an appropriate power of 2. Set $t' := \lfloor \frac{p}{t} \rfloor + 1$ and define homogeneous polynomials of degree 1 as

$$\begin{aligned} f_1(X,Y) &= X + t'Y \\ f_2(X,Y) &= tX + (tt' - p)Y. \end{aligned}$$

Search for $(x,y) \in \mathbb{Z}^2$, such that f_1, f_2 are simultaneously B–smooth for some bound B. This can be achieved by sieving. Having found such a pair, we have

$$\begin{aligned} x + t'y &= \prod_{p \le B} p^{e_p} \\ tx + tt'y &\equiv t(x + t'y) \\ &\equiv \prod_{p \le B} p^{e'_p} \bmod p \end{aligned}$$

and therefore

$$t \equiv \frac{\prod_{p \le B} p^{e'_p}}{\prod_{p \le B} p^{e_p}} \bmod p \tag{3.3}$$

which is a B–smooth representation of $t \bmod p$. The size of the numbers tested for smoothness is given by the following Lemma.

3.2 Lemma *Let $C > 0$ be a constant, $x, y \le C$, and f_1, f_2 as above. Then*

$$\begin{aligned} f_1(x,y) &\le C(1 + 2\sqrt{p}), \\ f_2(x,y) &\le 2C\sqrt{p}. \end{aligned}$$

Proof: With $x, y \le C$, we have

$$f_1(x,y) \le C(1 + t') \le C(1 + 2\sqrt{p}),$$

and

$$\begin{aligned} f_2(x,y) &= tx + (tt' - p)y \\ &\le tx + (t(p/t + 1) - p)y \\ &= tx + ty \\ &\le 2C\sqrt{p}. \end{aligned}$$

■

The numbers tested for simultaneous smoothness are of magnitude $O(\sqrt{p})$ as before, but the factoring subroutine is replaced by an efficient sieving method.

3.3 DL–Example For $p = 10007$ and $b = 5039$, we find

$$b \equiv 71/2 \bmod p, \tag{3.4}$$

by using Euclid's algorithm. Later, we will choose a factor base with maximal element 11, and large prime bound 19, so the numerator is not smooth over this factor base. Therefore we are going to sieve the two polynomials f_1, f_2, where

$$\begin{aligned} f_1(x,y) &:= x + 141y \\ f_2(x,y) &:= 71x + 4y, \end{aligned}$$

for $-100 \leq x \leq 100$ and $y = 1$. We find that $(-31, 1)$ is a good pair, because

$$\begin{aligned} f_1(-31,1) &= 2 \cdot 5 \cdot 11 \\ f_2(-31,1) &= -13^3. \end{aligned}$$

Equation (3.3) now tells us that

$$71 \equiv -\frac{13^3}{2 \cdot 5 \cdot 11} \bmod 10007. \tag{3.5}$$

So our task is reduced to finding the logarithms of $2, 5, 11, 13$, which are in our factor base.

In the sequel we may therefore restrict to computing logarithms of elements of moderate size. We consider the problem of computing a solution to

$$a^x \equiv s \bmod p,$$

where s is "sufficiently" small. Our experiments show that for p having at most 85 decimal digits, one can expect to find at least one relation which reduces the original DLP modulo p to problems where the right hand side consists of maximal ten decimal digits within acceptable time (table 3.1).

Table 3.1: Reduction Step

$\log_{10} p$	max $\lvert x \rvert$	max y	B	# rels	time (s)	time/rel (s)
65	10^6	200	10^9	127	2606	21
75	$2.5 \cdot 10^6$	5040	$1.5 \cdot 10^9$	203	246597	1215
85	$5.0 \cdot 10^6$	735	$2.0 \cdot 10^9$	6	111059	18510

Let us take a look at the row corresponding to the 85–digit number. The numbers t, t' consist of 40 and 45 digits respectively, the maximal value of x is below 10^7, while the maximal value of y is about 10^3. So we split 47–digit and 48–digit numbers into numbers below $B = 2.0 \cdot 10^9$. Sieving the two polynomials f_1, f_2 was done by using a factor base of primes below $3.5 \cdot 10^7$ and allowing one large prime up to $2.0 \cdot 10^9$. So it is required that the second largest prime factor is at most $3.5 \cdot 10^7$. The ρ–function tells us that we can expect this to happen with probability

$$1.90 \cdot 10^{-4} \cdot 3.20 \cdot 10^{-1} \cdot 1.39 \cdot 10^{-4} \cdot 3.11 \cdot 10^{-1} \approx 2.62 \cdot 10^{-7}.$$

Therefore it can be estimated that there is one B–smooth value among $3.8 \cdot 10^8$ coprime pairs (x, y). There are approximately $3.42 \cdot 10^9$ coprime pairs within the rectangle

$$\{(x, y) \in \mathbb{Z} \times \mathbb{Z} | -5 \cdot 10^6 \leq x \leq 5 \cdot 10^6, 1 \leq y \leq 735\}.$$

We conclude that the ρ function gives quite a good approximation, since

$$3.42 \cdot 10^9 / 3.8 \cdot 10^8 = 9;$$

we have found six relations. Similar considerations predict 876 for the 75–digit prime and 354 for the 65–digit prime, which is not too far from what we have obtained by sieving.

The method of [35], applied for finding smooth representations of b modulo a 85–digit number takes approximately one second on the same machine for trial division up to 10^6 and the Elliptic Curve Method for the range 10^6 up to 10^9. The search for one successful exponent would take $3.8 \cdot 10^8$ seconds= 105555 hours on average instead of 5 hours when using our new reduction method.

3.2 Constructing Polynomials

The construction of polynomials is a crucial point since all values, which are tested for smoothness during the NFS algorithm, depend on the two polynomials chosen at the beginning of the algorithm. First, the two polynomials must satisfy at least the following two conditions

- they must be irreducible over $\mathbb{Q}$, and
- they must have a common root modulo p.

Additionally, there are several obstructions to overcome, which have already led to difficult problems in the factoring case ($j = 1, 2$):

1. the ring $\mathbb{Z}[h_j\alpha_j]$ is not the ring of integers of the number field $\mathbb{Q}(\alpha_j)$
2. the ring of integers $\mathcal{O}_j$ of $\mathbb{Q}(\alpha_j)$ is not a principal ideal ring
3. the group of units of $\mathcal{O}_j$ is an infinite group
4. α_j is not an algebraic integer

From these obstructions, only the first one appears in the Gaussian integer variant of the Coppersmith–Odlyzko–Schroeppel algorithm (COS) [13], which is the reason for beginning our description with COS as a special case of the NFS algorithm. Actually, the invention of the NFS algorithm was inspired by the Gaussian integer variant of the COS algorithm, which in turn was inspired by an algorithm of ElGamal [22].

3.2.1 The COS Algorithm

There are three sub–exponential discrete logarithm algorithms described in the article of Coppersmith, Odlyzko, and Schroeppel. One of them, namely the Gaussian integer method, is apparently the most practical one, and this is the variation, which we denote by COS. The use of the COS algorithm is convenient in the sense that the number rings involved behave in a friendly way. One chooses a quadratic imaginary number field of the form $\mathbb{Q}(\sqrt{-r})$, such that its maximal order is a principal ideal ring. This is exactly the case for

$$r \in M := \{1, 2, 3, 7, 11, 19, 43, 67, 163\}.$$

The particular choice of r directly avoids obstruction 2 from above. As there are maximal 6 units in imaginary quadratic number rings, obstruction 3 does not matter either. Since α is an algebraic integer, obstruction 4 does not occur.

Obstruction 1 occurs when $r \equiv 3 \bmod 4$. In this case, we do not compute in $\mathbb{Z}[\sqrt{-r}]$ but in $\mathbb{Z}[(1 + \sqrt{-r})/2]$, which is the ring of integers of $\mathbb{Q}(\sqrt{-r})$. Consequently, we choose g_1 to be

$$g_1(X) = \begin{cases} X^2 + X + \frac{r+1}{4}, & \text{if } r \equiv 3 \bmod 4 \\ X^2 + r, & \text{otherwise} \end{cases}$$

and

$$\alpha = \alpha_1 = \begin{cases} \frac{1+\sqrt{-r}}{2}, & \text{if } r \equiv 3 \bmod 4 \\ \sqrt{-r}, & \text{otherwise} \end{cases}.$$

Here r is set to the minimal r' in the set M with $-r'$ being a quadratic residue modulo p. Let $w \in \mathbb{Z}$ with $w^2 \equiv -r \bmod p$. Then, a representation $w \equiv T/V$ is found, with T, V of size $\sqrt{p}$. We have a ring homomorphism

$$\begin{array}{rccl} \varphi: & \mathbb{Z}[\alpha] & \longrightarrow & \mathbb{Z}/p\mathbb{Z} \\ & \alpha & \mapsto & \begin{cases} \frac{1+T/V}{2}, & \text{if } r \equiv 3 \bmod 4 \\ T/V, & \text{otherwise.} \end{cases} \end{array}$$

For each relation (c, d), we therefore get the following equality

$$\varphi(c + d\alpha) \equiv \begin{cases} c + d\frac{1+T/V}{2} \bmod p, & \text{if } r \equiv 3 \bmod 4 \\ c + d\frac{T}{V} \bmod p, & \text{otherwise} \end{cases}$$

or equivalently

$$\begin{array}{ll} 2V\varphi(c + d\alpha) \equiv 2cV + d(T + V) \bmod p, & \text{if } r \equiv 3 \bmod 4 \\ V\varphi(c + d\alpha) \equiv cV + dT \bmod p, & \text{otherwise.} \end{array} \tag{3.6}$$

An element $c + d\sqrt{-r}$ is smooth when its norm

$$N(c + d\alpha) = \begin{cases} c^2 + cd + (r + 1)d^2/4, & \text{if } r \equiv 3 \bmod 4 \\ c^2 + rd^2, & \text{otherwise} \end{cases}$$

is smooth. From the polynomial g_1 we construct the bivariate homogeneous polynomial

$$f_1(X, Y) = \begin{cases} X^2 + XY + \frac{r+1}{4}Y^2 & \text{if } r \equiv 3 \bmod 4 \\ X^2 + rY^2 & \text{otherwise} \end{cases}, \tag{3.7}$$

the values of which are tested for smoothness.

Obviously, the second homogeneous polynomial tested for smoothness is

$$f_2(X, Y) = \begin{cases} 2V\varphi(c + d\alpha) \equiv 2VX + (T + V)Y \bmod p, & \text{if } r \equiv 3 \bmod 4 \\ V\varphi(c + d\alpha) \equiv VX + TY \bmod p, & \text{otherwise} \end{cases} \tag{3.8}$$

according to equation (3.6).

3.4 DL–Example For the prime $p = 10007$, we find that $r = 7 \in M$ is the smallest value, for which $-r$ is a square mod p. We compute $T = 7$ and $V = -100$, which results in the polynomials

$$\begin{array}{rcll} f_1(X, Y) & = & X^2 + XY + 2, & \text{and} \\ f_2(X, Y) & = & -200X - 93Y. & \end{array}$$

3.2.2 Standard NFS

We call the following method for picking a number field "standard" NFS, because the basic idea of it has widely been used in factoring large integers, culminating in the latest record of factoring a general 130 digit number [15]. Prescribing the degree n of the number field, the NFS starts by taking an integer

$$m \in [p^{\frac{1}{n+1}}, p^{\frac{1}{n}}],$$

and computing a modified m–adic representation of p as

$$p = a_n m^n + a_{n-1} m^{n-1} + \cdots + a_1 m + a_0, \quad |a_i| \leq \frac{m}{2}, 1 \leq i \leq n.$$

Taking

$$\begin{aligned} g_1(X) &= X - m, \\ g_2(X) &= \sum_{j=0}^{n} a_j X^j \end{aligned}$$

then yields a valid pair of NFS polynomials. Here m plays the role of α_1 in the notation of section 2.2. This is performed several thousand times for random m's of the interval above. Note that for $m \in [\frac{1}{2}p^{\frac{1}{n}}, p^{\frac{1}{n}}]$ a monic polynomial g_2 shows up.

3.5 DL–Example Choose the degree of the number field to be 4. For $p = 10007$, we can take m with $7 \leq m \leq 10$. Take $m = 10$ and obtain

$$\begin{aligned} g_1(X) &= X - 10, \quad \text{and} \\ g_2(X) &= X^4 + X - 3. \end{aligned}$$

The polynomial $g_2(X) \in Z[X]$ has to be chosen in such a way, that q does not ramify in $\mathcal{O}_2$ for each prime factor q of $p - 1$ we want to apply the algorithm to. The reason for this condition will become obvious in section 3.6; we check the condition by applying the distinct degree factorization algorithm, which is part of the Cantor–Zassenhaus algorithm to factor polynomials modulo a prime [12, Algorithm 3.4.3]. Distinct degree factorization modulo q can be done in polynomial time and is quite fast in practice. In table 3.2, the running times are shown for q running through 500 primes of 130 decimal digits. The degree of g_2 varies between 3 and 6, which is the usual range of degrees for NFS polynomials. Because the number of bit operations depends on how g_2 splits, we calculated the average running time with respect to the number of roots of g_2 modulo q. The percentage indicates how likely it is for g_2 to have the corresponding number of roots.

Table 3.2: Distinct Degree Factorization

degree g_2	# of roots	percentage	avg. running time (s)
3	0	33.4	567
3	1	50.4	282
3	3	16.2	92
4	0	36.3	773
4	1	33.1	569
4	2	26.1	283
4	4	4.8	94
5	0	37.5	919
5	1	36.3	604
5	2	16.7	455
5	3	8.3	226
5	5	1.2	77
6	0	36.7	1367
6	1	35.5	1096
6	2	20.4	745
6	3	5.1	566
6	4	2.0	284
6	6	0.2	88

In order to achieve small coefficients, for each m–adic representation of p, it is worthwhile performing an LLL–reduction on the coefficient vectors of g_2 [78]. The difference between straightforward m–adic representation in the monic case and the application of LLL–reduction is to be seen in table 3.3 in section 3.3 on page 37.

We need simultaneous smoothness of elements $c + dm$ and $h_2 \cdot (c + d\alpha_2)$. The norm of $h_2 \cdot (c + d\alpha_2)$ is given by

$$N(h_2 \cdot (c + d\alpha_2)) = (-d)^n g_2(-\frac{c}{d}) \cdot h_2^{n-1}.$$

The prime ideal decomposition of $h_2 \cdot (c + d\alpha_2)$ is determined by factoring

$$N'(h_2 \cdot (c + d\alpha_2)) = (-d)^n g_2(-\frac{c}{d}), \tag{3.9}$$

according to the Lemmata 1.35 and 1.39. When using a monic polynomial g_2, $N'(c + d\alpha_2) = N(c + d\alpha_2)$. We postpone the situation for non–monic polynomials until the next subsection, which is devoted to that subject.

To compare the generated polynomials, $N(h_2 \cdot (c + d\alpha_2))$ is computed for many pairs (c, d) within the sieving range.

3.2.3 Non–monic Polynomials

As in the case of factoring integers with the NFS, requiring g_2 to be monic is not really necessary if one employs the facts of Lemma 1.37 by making use of the polynomial $T(X) := g_2(\frac{X}{h_2})h_2^{n-1} \in \mathbb{Z}[X]$ with root $\omega := h_2\alpha_2$. With non–monic polynomials, it is possible to choose a slightly smaller m and therefore to get a better probability for decompositions over the rational factor base.

3.6 Example Take $p \approx 10^{85}$, $n = 5$. With monic polynomials, the smallest possible m is $p^{1/n}/2 = 5 \cdot 10^{16}$. With non–monic polynomials, the smallest possible m is $p^{1/(n+1)} \approx 1.5 \cdot 10^{14}$.

Furthermore, and this is more important, the use of non–monic polynomials allows to adapt the two–quadratics–version of the NFS, which we will describe in the next subsection 3.2.4.

According to Lemma 1.37ii), ω is an algebraic integer and generates the same field as α over $\mathbb{Q}$, by iii). Let $T(X) = g_2(\frac{X}{h_2})h_2^{n-1} \in \mathbb{Z}[X]$, such that $\alpha_2 \cdot h_2 = \omega$. As before, we obtain the true decomposition into prime ideals of $\mathcal{O}_2$ by Lemma 1.37iv). In order to compare the non–monic with the monic version, we present table 3.3 in section 3.3 on page 37 showing another effect concering the factor bases associated to the polynomials.

3.2.4 NFS with Two Quadratics

Originally invented by Montgomery [48], the use of two quadratic polynomials in the NFS algorithm has been exploited to achieve impressive factorizations of large integers by Elkenbracht–Huizing [23]. As we shall see, for discrete log problems within the currently solvable range, it turned out to be the preferred method compared to the standard NFS method from subsections 3.2.2 and 3.2.3.

Given an integer m, it is easy to find two quadratic polynomials g_1, g_2 with

$$g_1(m) = g_2(m) \equiv 0 \bmod p.$$

The point is that both polynomials should have small coefficients. Montgomery's construction of the two polynomials is based on the observation that by using the standard inner product, the two vectors

$$\begin{pmatrix} c_{12} \\ c_{11} \\ c_{10} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} c_{22} \\ c_{21} \\ c_{20} \end{pmatrix}$$

are orthogonal to the vector

$$\vec{m} := \begin{pmatrix} 1 \\ m \\ m^2 \end{pmatrix}$$

modulo p, where the quadratic polynomials g_1, g_2 are of the form

$$g_i(X) = c_{i2}X^2 + c_{i1}X + c_{i0}.$$

His method starts by fixing a prime $r < \sqrt{p}$, then solving

$$c^2 \equiv p \bmod r$$

for c and defining $m :\equiv c/r \bmod p$. Let s be the inverse of c in $(\mathbb{Z}/r\mathbb{Z})^*$. Then

$$\vec{a} := \begin{pmatrix} rm \\ -r \\ 0 \end{pmatrix} \quad \text{and} \quad \vec{b} := \begin{pmatrix} (rm(m \operatorname{Mod} r) - rm^2)/r \\ -m \operatorname{Mod} r \\ 1 \end{pmatrix}$$

are orthogonal to $\vec{m}$ and consist of entries of magnitude $\sqrt{p}$. The vectors $\vec{a}$ and $\vec{b}$ even span the sublattice of $\mathbb{Z}^3$ orthogonal to $\vec{m}$. Let $\vec{a'}$, $\vec{b'}$ be the result of the lattice reduction algorithm applied to $\vec{a}$, $\vec{b}$. One can show that for the length of $\vec{a'}$, $\vec{b'}$, we have

$$||\vec{a'}|| \cdot ||\vec{b'}|| = O(\sqrt{p}).$$

In practice, however, the lengths of the two vectors are of magnitude $p^{1/4}$. Different values of r produce different polynomials.

To obtain useful polynomials g_1 and g_2 for DL by Montgomery's method, we add the condition that a and s split in either of the corresponding number rings. Assume that $(a) = \mathfrak{p}\mathfrak{p}'$ in $\mathcal{O}_1$, and (s) splits either in $\mathcal{O}_1$ or $\mathcal{O}_2$. Without loss of generality, assume $(s) = \mathfrak{q}\mathfrak{q}'$ in $\mathcal{O}_2$. The method works similar in case of (s) splitting in $\mathcal{O}_1$.

3.7 DL–Example For $p = 10007$ we may take $r = 227$ and obtain

$$\begin{aligned} g_1(X) &= 7X^2 + 20X - 7 \\ g_2(X) &= 4X^2 - 21X + 8, \end{aligned}$$

with common root $m = 5599$ modulo p, where both $a = 5$ and $s_1 := 2$ split in $\mathcal{O}_1$ and $s_2 := 71$ splits in $\mathcal{O}_2$.

3.3 Choosing Factor Bases

In the NFS algorithm, factor bases consist of a finite subset of the first degree prime ideals of the ring $\mathcal{O}_j$. According to Theorem 1.35, a first degree prime ideal of norm q, which does not divide the index $[\mathcal{O}_2 : \mathbb{Z}[\omega]]$ is generated by $(q, \alpha_j - r)$ over $\mathcal{O}_j$, for every root r of g_j modulo q. Avoiding index divisors has two main advantages. Firstly, the whole factor base can be computed by finding roots of our polynomials modulo small primes. Secondly, the prime ideal factorization of $(h_j \cdot (c + d\alpha_j)) \in \mathcal{O}_j$ can be constructed by decomposing $N(h_j \cdot (c + d\alpha_j)) \in \mathbb{Z}$, according to Lemma 1.39.

To recognize index divisors r, we search for quadratic divisors of the discriminant of f and apply the Dedekind test to them (see Lemma 1.32 and Theorem 1.34). Avoiding index divisors restricts the choice of f considerably, but surprisingly enough, this does not prevent us from finding polynomials which lead to elements with small norms. In table 3.3 on page 37, we list experimental data concerning discrete log problems, where p has 50, 65 and 75 decimal digits. Here we have examined 4000 polynomials for each p.

The column poly-type contains the information, whether only monic or non–monic polynomials are considered. The third column lists the degree of the polynomials tested. The next column reports the number of polynomials having square discriminant divisors below the factor base bound. After testing by means of the Dedekind-criterion, which of the square discriminant divisors are index divisors, we get the number of good polynomials. The others are called bad in the sense that

Table 3.3: Comparison of Polynomials

digits	poly-type	degree	r^2|disc	good	bad	norm (all)	norm (good)
50	non–monic	3	3844	237	3763	$4.2 \cdot 10^{24}$	$4.9 \cdot 10^{24}$
50	non–monic	4	3944	123	3877	$2.2 \cdot 10^{28}$	$7.2 \cdot 10^{28}$
50	non–monic	5	3927	157	3843	$5.1 \cdot 10^{32}$	$1.0 \cdot 10^{33}$
50	monic	3	2390	2650	1350	$8.1 \cdot 10^{29}$	$8.1 \cdot 10^{29}$
50	monic	4	2555	3024	976	$3.3 \cdot 10^{31}$	$3.4 \cdot 10^{31}$
50	monic	5	2060	2902	1098	$6.6 \cdot 10^{34}$	$6.6 \cdot 10^{34}$
65	non–monic	3	3922	120	3880	$9.9 \cdot 10^{28}$	$4.3 \cdot 10^{29}$
65	non–monic	4	3920	143	3857	$1.3 \cdot 10^{31}$	$5.6 \cdot 10^{31}$
65	non–monic	5	3941	123	3877	$9.0 \cdot 10^{34}$	$4.4 \cdot 10^{35}$
65	monic	3	2915	2391	1609	$1.2 \cdot 10^{35}$	$1.3 \cdot 10^{35}$
65	monic	4	2551	2850	1150	$6.7 \cdot 10^{34}$	$6.7 \cdot 10^{34}$
65	monic	5	2427	2747	1253	$1.4 \cdot 10^{38}$	$1.4 \cdot 10^{38}$
75	non–monic	3	3922	134	3866	$2.3 \cdot 10^{31}$	$5.3 \cdot 10^{31}$
75	non–monic	4	3912	146	3854	$1.3 \cdot 10^{33}$	$6.3 \cdot 10^{33}$
75	non–monic	5	3915	165	3835	$5.7 \cdot 10^{36}$	$1.1 \cdot 10^{37}$
75	monic	3	2540	2432	1568	$1.9 \cdot 10^{38}$	$1.9 \cdot 10^{38}$
75	monic	4	2560	2867	1133	$3.1 \cdot 10^{37}$	$3.1 \cdot 10^{37}$
75	monic	5	2396	2788	1212	$1.3 \cdot 10^{40}$	$1.3 \cdot 10^{40}$

they cannot be used without having a more time–consuming procedure to recognize the correct exponents in the prime ideal factorization of $h_2c + d\omega$ corresponding to $(c, d) \in S$. As table 3.3 shows, the norms are merely slight worse as if the irreducible polynomials could be chosen without restriction.

Because the size of the factor base directly affects the size of the linear system to be solved in the last step of the NFS algorithm, we choose it maximal with respect to the current capability of solving large sparse linear systems modulo a big prime. The multiprecision arithmetic performed in this step is the reason, why we choose considerably smaller factor bases than in the factoring case where the linear systems are solved modulo 2. We compare ours to Zayer's examples [78] in table 3.4 on page 38.

The factor bases of the 85–digit example (size 70000) and the 129–digit McCurley

Table 3.4: Factor Bases Factoring/DL

digits	factoring	DL
50	17200	3491
65	24752	19954
75	34941	25058

challenge (size 40000) are not comparable to Zayer's factoring examples because we did the 85–digit problem with the two–quadratics–method which he did not use and the prime of the McCurley's challenge is of a special form. For the choice of the large prime bound, we have adapted the successful heuristics from factoring [78]. Since the large primes do not enlarge the matrix A but only its weight, we can use large prime bounds in magnitude comparable to factoring. But we need more large prime relations in our case because many large prime relations reduce the weight of resulting matrix rows as will be explained in section 3.5.

3.4 Sieving

Once the Number Field Sieve is properly initialized for computing discrete logarithms according to the previous sections, sieving is identical with the factoring case. Since this part of the whole computation can be cheaply distributed over independent clusters of workstations in contrast to linear algebra, sieving will dominate the running time. It is typical for this sort of algorithms that we are not interested in minimizing the total CPU time; instead, we are inclined to minimize the elapsed real time, which accumulates from the distributed sieving and the one–machine linear algebra step. For two reasons we will not give a detailed treatment of the sieving procedure at this point. On the one hand, in chapter 4 we will describe a more general sieving device covering NFS, COS, the Quadratic Sieve, and many more sieving algorithms. On the other hand, there are excellent descriptions of the NFS sieving step to be found in literature [78, 37, 23].

3.4.1 The Classical Sieve

In the sieving step, we need to find many pairs $(c, d) \in \mathbb{Z}\times\mathbb{Z}$, where $h_1\cdot(c+d\alpha_1) \in \mathcal{O}_1$ as well as $h_2 \cdot (c + d\alpha_2) \in \mathcal{O}_2$ are smooth with respect to factor bases F_1, F_2

respectively. For the following description we set $\alpha := \alpha_j$, $h := h_j$, $\mathcal{O} := \mathcal{O}_j$ for $j = 1, 2$.

Suppose, we want to find smooth values in the set

$$M := \{h \cdot (c + d\alpha) \mid (c, d) \in \mathbb{Z} \times \mathbb{Z}\}.$$

Let $\mathfrak{r}$ be a prime ideal of norm $r \in \mathbb{P}$. The classical sieve is based on the observation that

$$\mathfrak{r} | h \cdot (c + d\alpha) \Longleftrightarrow \mathfrak{r} | h \cdot (c + r + d\alpha).$$

Given $d' \in \mathbb{Z}$, it suffices to know one location $c' \in \mathbb{Z}$, where $\mathfrak{r} | h \cdot (c' + d'\alpha)$. In this case the subset of all elements of M, which are divisible by $\mathfrak{r}$, is

$$\{h \cdot (c + d'\alpha) \mid c = c' + k \cdot r,\ k \in \mathbb{Z}\}.$$

Evidently, pairs (c, d') with $\gcd(c, d') > 1$ can be omitted.

In table 3.5 on page 39, we show how many relations we got after having finished the sieving step. The first letter at the beginning of a row (S,Q,C) refers to whether we have used the standard NFS algorithm (S), the two–quadratics NFS (Q), or the COS Gaussian–Integer–Method (C). The factor base size is given as $|F_1| + |F_2|$, the large prime bound is taken from the polynomial, which produces the far greater values. For the Standard NFS, this is the polynomial of degree greater than one; for the COS, this is the polynomial of degree one. For the two–quadratics version, the two large prime bounds are equal, since the coefficients of the two polynomials have approximately the same size.

Table 3.5: Collected Partials

	$\log_{10} p$	#FB	LP	# relations					mips
				full	single	double	triple	quad	years
S	65	19954	$6 \cdot 10^6$	745	12695	84904	266067	316388	5.3
Q	65	26135	$6 \cdot 10^6$	13708	177474	760106	1341965	679587	9.1
S	75	25058	10^7	714	13879	107723	385844	534231	70.0
C	75	24980	$2 \cdot 10^7$	5970	103738	493017	733439	205812	11.4
Q	85	70339	$8 \cdot 10^7$	5415	109082	8114437	2563554	5015662	44.5
C	85	69981	10^7	15115	136803	335890	280808	59078	30.6

3.4.2 The Lattice Sieve

As with the classical sieving, Pollard's lattice sieve method is also to be found in various publications [59, 23, 25]. For the sake of completeness, we present its idea. For a given prime ideal $\mathfrak{r}$ of norm r, the lattice sieve tries to find smooth elements in the set

$$M_{\mathfrak{r}} := \left\{ \frac{(h) \cdot (c + d\alpha)}{\mathfrak{r}} \mid (c,d) \in \mathbb{Z} \times \mathbb{Z}, (h) \cdot (c + d\alpha) \subset \mathfrak{r} \right\}.$$

Denote by $L_{\mathfrak{r}}$ the set of pairs (c,d), which determine the elements of $M_{\mathfrak{r}}$. The set $L_{\mathfrak{r}}$ actually is a lattice over $\mathbb{Z}$. In order to sieve over small elements in the lattice, one computes a basis of $L_{\mathfrak{r}}$ consisting of two vectors $\vec{v_1} := (c_1, d_1)$, $\vec{v_2} = (c_2, d_2)$ having a short Euclidean length. This is achieved by means of a straightforward lattice reduction method. Analogously to the classical NFS, sieving is then based on the fact that

$$\begin{aligned} &\mathfrak{q} \mid h \cdot (\lambda \cdot (c_1 + d_1\alpha) + \mu \cdot (c_2 + d_2\alpha)) \\ &\iff \mathfrak{q} \mid h \cdot ((\lambda + q) \cdot (c_1 + d_1\alpha) + \mu \cdot (c_2 + d_2\alpha)) \quad (\lambda, \mu) \in \mathbb{Z} \times \mathbb{Z} \end{aligned}$$

when $\mathfrak{q}$ is a prime ideal of norm q. As before, pairs (λ, μ) with $\gcd(\lambda, \mu) > 1$ can be omitted.

3.4.3 Individual Relations

In order to determine the DL of s, it is necessary to find relations which involve s in some way. In theory, the polynomials are constructed in such a way that a relation involving s is automatically obtained [27, 64]. These methods only apply for the standard NFS and prescribe the polynomial g_1 of degree 1 and search for a polynomial g_2 defining the number field $\mathbb{Q}(\alpha_2)$, such that the ideal (α_2) splits over the factor base belonging to g_2.

In practice, there are usually many logarithms to compute in the same prime field. As we cannot afford to repeat the sieving step for each of them within different number fields, we take a different approach, which is convenient for the two–quadratics–method. According to subsection 3.2.4, we may assume that $a = \mathfrak{p}\mathfrak{p}'$, $s = \mathfrak{q}\mathfrak{q}'$ in $\mathcal{O}_2$.

Additionally to the collected set of relations, we find relations $(c_a^{(1)}, d_a^{(1)})$, $(c_a^{(2)}, d_a^{(2)})$, $(c_s^{(1)}, d_s^{(1)})$, $(c_s^{(2)}, d_s^{(2)})$, with

- $h_1 \cdot (c_a^{(1)} + d_a^{(1)}\alpha_1)/\mathfrak{p}$ smooth over F_1,

- $h_2 \cdot (c_a^{(1)} + d_a^{(1)}\alpha_2)$ smooth over F_2,
- $h_1 \cdot (c_a^{(2)} + d_a^{(2)}\alpha_1)/\mathfrak{p}'$ smooth over F_1,
- $h_2 \cdot (c_a^{(2)} + d_a^{(2)}\alpha_2)$ smooth over F_2.
- $h_2 \cdot (c_s^{(1)} + d_s^{(1)}\alpha_2)/\mathfrak{q}$ smooth over F_2,
- $h_1 \cdot (c_s^{(1)} + d_s^{(1)}\alpha_1)$ smooth over F_1,
- $h_2 \cdot (c_s^{(2)} + d_s^{(2)}\alpha_2)/\mathfrak{q}'$ smooth over F_2,
- $h_1 \cdot (c_s^{(2)} + d_s^{(2)}\alpha_1)$ smooth over F_1.

This is no problem in case of $\mathfrak{p}, \mathfrak{p}' \in F_1$ and $\mathfrak{q}, \mathfrak{q}' \in F_2$.

Otherwise do lattice sieving with the prime ideals $\mathfrak{p}$, $\mathfrak{p}'$, $\mathfrak{q}$, $\mathfrak{q}'$ to find at least one partial relation involving the corresponding prime ideal.

Enlarge S by the four relations $(c_a^{(1)}, d_a^{(1)})$, $(c_a^{(2)}, d_a^{(2)})$, $(c_s^{(1)}, d_s^{(1)})$, $(c_s^{(2)}, d_s^{(2)})$.

In practice, finding such individual relations is quite fast; from section 3.1 we already know that it is no problem to achieve $s \approx 10^9$. The lattice sieve then produces many relations of the type denoted above. This is confirmed by experimental data taken from a DL computation with a 65–digit p (table 3.6). The sieving rectangle was set to

$$-8000 \leq \lambda \leq 8000, \quad 1 \leq \mu \leq 900.$$

Sieving took about 2 minutes per $\mathfrak{q}$ on a Sparc 20 workstation.

Now pick relations (c_a, d_a), (c_s, d_s) from S and change them by multiplying with (a) and (s) respectively:

$$\mathfrak{p}\mathfrak{p}'(h_1) \cdot (c_a + d_a\alpha_1) = (a) \cdot (h_1) \cdot (c_a + d_a\alpha_1) \tag{3.10}$$

$$\mathfrak{q}\mathfrak{q}'(h_2) \cdot (c_s + d_s\alpha_2) = (s) \cdot (h_2) \cdot (c_a + d_a\alpha_2). \tag{3.11}$$

3.8 DL–Example Assume, we were faced with equation (3.4) of DL–example 3.3 on page 28 and would not have found equation (3.5). When using two quadratics, we are left with $s_1 = 2$, $s_2 = 71$. (s_1) is the square of a factor base prime ideal of F_1, so we can expect to get many relations involving s_1. The same holds for $a = 5$,

Table 3.6: Individual Relations 65–digit p

$\mathfrak{q}$	single	double	triple	quadruple	quintuple
(233201,82509)	1	15	40	71	43
(233201,90266)	0	4	23	49	23
(507119,3691)	0	13	35	49	33
(507119,478728)	1	10	45	62	30
(746047,190901)	0	7	23	39	27
(746047,369716)	1	11	21	55	28
(825509,22278)	1	11	32	53	46
(825509,351888)	0	4	26	46	27
(865043,549560)	0	2	3	16	16
(865043,675391)	0	7	30	58	28
(1667917,1314755)	1	6	22	41	36
(1667917,219882)	1	8	32	58	40
(1879849,317870)	0	3	17	30	34
(1879849,873774)	0	6	26	33	24
(2061361,382437)	1	4	25	47	38
(2061361,928562)	0	7	29	51	35
(2090069,100653)	1	7	22	39	29
(2090069,2030854)	1	10	26	59	37
(7299247,5408589)	1	4	14	33	27
(7299247,6299489)	0	5	13	24	16

because $5\mathcal{O}_1 = \mathfrak{p}\mathfrak{p}'$. For s_2, we do lattice sieving in $\mathcal{O}_2$ for the prime ideal identified by $(71, 28)$ and $(71, 66)$ and obtain

$$\begin{aligned} 4(17+7\alpha_2)/\mathfrak{q}_{71,28} &= \mathfrak{p}_{2,0}\mathfrak{p}_{3,1}\mathfrak{p}_{19,3} \\ 4(5+1\alpha_2)/\mathfrak{q}_{71,66} &= \mathfrak{p}_{2,0}\mathfrak{p}_{3,1} \\ 7(17+7\alpha_1) &= \mathfrak{p}_{2,1}^2\mathfrak{p}_{5,4}\mathfrak{p}_{7,0}\mathfrak{p}_{7,7} \\ 7(5+1\alpha_1) &= \mathfrak{p}_{2,1}^2\mathfrak{p}_{7,0}\mathfrak{p}_{17,12}. \end{aligned}$$

3.5 Combining Large Prime Relations

A common technique of reducing the amount of main memory which is needed in the subsequent steps is the *filtering step*. Filtering means to remove those partials, which contain a large prime that does not occur a second time in the set of partials. Of course, this is what we do at this point, but we will not go into any detail because it has already been discussed in reports about implementations of factoring algorithms. Table 3.7 shows that also in the DL case, where the factor base size is smaller, we spare a substantial amount of main memory.

Table 3.7: Filtering Step

	$\log_{10} p$	#relations				after filtering			
		single	double	triple	quad	single	double	triple	quad
S	65	12695	84904	266067	316388	11907	74183	218665	248539
Q	65	177474	760106	1341965	679587	147545	536583	–	–
S	75	13879	107723	385844	534231	12471	87473	286887	367262
C	75	103738	493017	733439	205812	86906	378678	541111	151145
Q	85	154235	1167955	3748694	5015662	102657	549756	1337556	–
C	85	136803	335890	280808	59078	94344	163065	98891	15459

The large prime variation of the NFS now combines relations containing at least one large prime so that relations without any large primes will be the result. For each successfully combined set of partials the exponent vector only consists of exponents of factor base elements and therefore fits into the relation matrix A. When the NFS is used for factoring it does not matter whether the partials are multiplied or divided by each other. This is because the exponent vectors are considered modulo 2. In the discrete log case we are in a slightly different situation, since we are interested in combining products to q–th powers, so the exponents of large primes have to be combined mod q.

To recognize sets of partial relations, which can be combined in such a way that all large primes occur as squares, a well–known graph algorithm exists and has been widely used in the Quadratic Sieve algorithm and the NFS for factoring. First developed by Lenstra and Manasse [39] for the use of two large primes, it was extended by Zayer up to four large primes [78]. The lattice sieve variant even produces relations with five large primes when sieving with large primes. Therefore we unify the description of cycle finding when faced with a set of partial relations R, where each relation of R contains at most $l \geq 2$ large primes, for fixed l. With the generalization, we cover all the special cases mentioned before.

It remains to explain how we cope with our different situation in the discrete log case: finding sets of partials, in which the large prime exponents can be combined mod q. We postpone this subject to section 3.5.2.

3.5.1 Cycle Finding with l Large Primes

The output of a sieve algorithm contains many relations, which consist of elements that split either completely or partially over a factor base. Full relations can be transported to the matrix step without any changes. Partial relations will be composed in an appropriate way so that a complete splitting is the result. Before this can be done, it is essential to know, which subsets of partials will contribute to a full relation.

3.9 Example Assume, we have collected the following set of partials.

$$\begin{array}{lll}
(c_1, d_1) & \mathfrak{P}_1, & \mathfrak{Q}_1 \\
(c_2, d_2) & \mathfrak{P}_2, & \mathfrak{Q}_2 \\
(c_3, d_3) & \mathfrak{P}_3, & \mathfrak{Q}_1 \\
(c_4, d_4) & \mathfrak{P}_1, & \mathfrak{Q}_3 \\
(c_5, d_5) & 1, & \mathfrak{Q}_2 \\
(c_6, d_6) & \mathfrak{P}_4, & \mathfrak{Q}_1 \\
(c_7, d_7) & \mathfrak{P}_3, & 1 \\
(c_8, d_8) & \mathfrak{P}_2, & \mathfrak{Q}_3
\end{array}$$

This is a compact representation of the large primes occuring in eight relations. For example, the first row indicates that there is a relation

$$\begin{array}{lllll}
c_1 + d_1\alpha_1 & = & \mathfrak{P}_1 & \prod_{\mathfrak{p}\in F_1} & \mathfrak{p}^{e_\mathfrak{p}} \\
c_1 + d_1\alpha_2 & = & \mathfrak{Q}_1 & \prod_{\mathfrak{q}\in F_2} & \mathfrak{q}^{e_\mathfrak{q}},
\end{array}$$

and analogously for the other rows. It is convenient to introduce a pseudo large prime 1, which fills the place of a large prime if the relation does not consist of the maximal number of large primes.

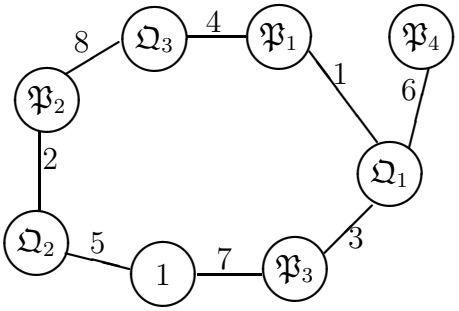
The cycle finding algorithm will dynamically change the directed graph $G = (V, E)$, where V is a set of large primes and E is a set of labeled edges.

Denote by R the set of relations found by one of the NFS algorithms. Then the labeling is given by the map

$$\Phi : \quad E \quad \longrightarrow \quad 2^R.$$

That is, edges are labeled by sets of relations. The graph belonging to the partials of example 3.9 is depicted in figure 3.1. When maximal two large primes are used, the labeling simplifies to one relation number.

Figure 3.1: Graph Example



It will be useful for the description of the cycle finding algorithm to extend the map Φ to paths. Let $v, v' \in V$ and the path in figure 3.2 from v to v' be given.

Then we define Φ for the path from v to v' to be $\Phi(v, v') := \Phi(e_1) \cup \ldots \cup \Phi(e_k)$. To keep this well–defined, we ensure during the algorithm that there can be only one path between two vertices and that the condition outdegree$(v) = 1$ holds for all $v \in V$. This is done as follows. The graph gets initialized by one vertex and no edges. Each operation of the algorithm creates either a new edge not connected to the graph or keeps the structure of a tree. So in each step of the algorithm, the graph is in fact a union of trees. A basic subroutine of the algorithm is given by computing the function

$$\begin{array}{rccc} \text{root}: & V & \longrightarrow & V \\ & v & \mapsto & \text{root}(v), \end{array}$$

Figure 3.2:

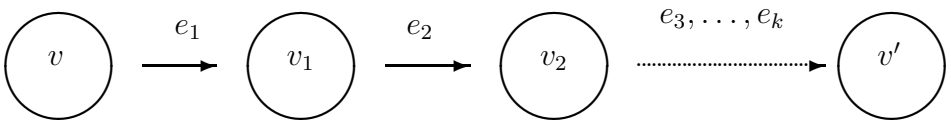
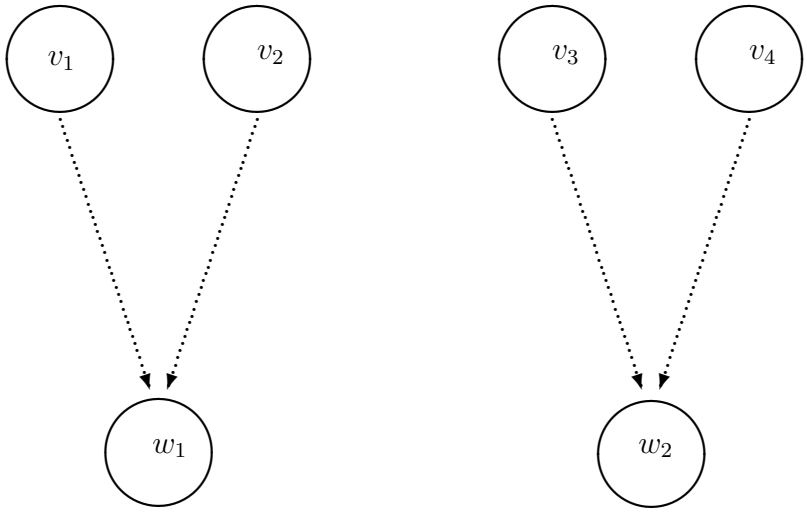


Figure 3.3: Quadruple Yields Full



where root(v) is the vertex at the end of the path starting in v. In particular, this is useful to compute a cycle in G when G is considered as an undirected graph. For example, if we intend to insert the edge (v, v') and we are faced with the condition root(v) = root(v'), we have encountered a path

$$v' \longrightarrow \text{root}(v') = \text{root}(v) \longrightarrow v,$$

which actually is a cycle.

As a first application of the root–function, we sketch how a relation with four large primes will be reduced to a relation without any, or one, or two large primes – provided that appropriate partial cycles can be found.

A quadruple relation yields a full relation if two different pairs of large primes have the same root in the graph (figure 3.3).

Figure 3.4 shows how a quadruple relation can be reduced to a single large prime relation, whereas in figure 3.5 the case of a quadruple relation yielding a double relation is depicted.

From figure 3.5 we see the possibility of violating the condition outdegree$(v) = 1$. This obstruction is eliminated by turning all edges on the path $(v_1, \ldots, w_1)$ into direction v_1 so that after this operation we actually have a path

$$w_1 \rightarrow v_1 \rightarrow v_2 \rightarrow w_2.$$

Figure 3.4: Quadruple Yields Single

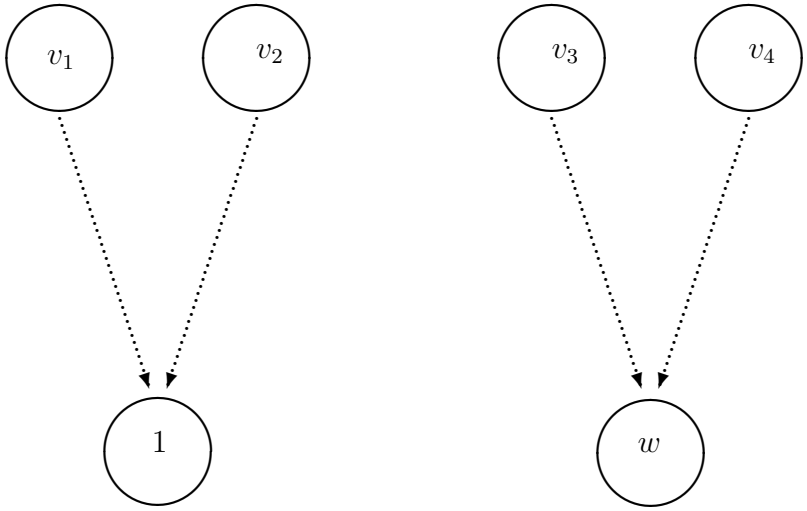
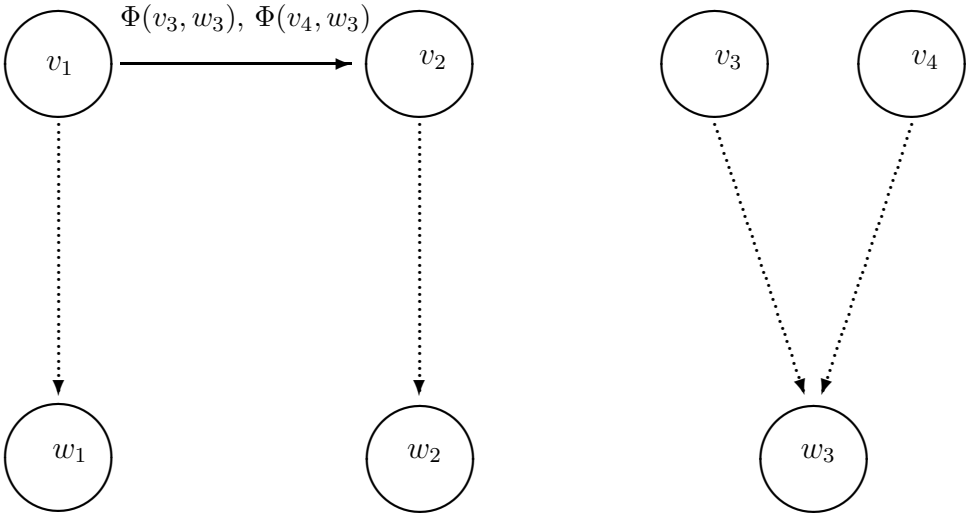


Figure 3.5: Quadruple Yields Double



Operations of this type are collected in a function called `adjust_E`, which is called by the cycle finding procedure.

The following lines of pseudo–code show, how the general cycle finding algorithm is implemented.

Finding Cycles

INPUT: partials, consisting of at most l large primes
OUTPUT: sets of partials, which can be combined to fulls

INITIALIZATION

(1)	$V = \{1\}$;	pseudo large prime
(2)	**for** (each partial $r = \{P_1, \ldots, P_l\} \in R$) **do**	consider all partials

FIND PARTIAL CYCLES (DOUBLES)

(3)	$H = \{r\}$;	partials of cycle
(4)	**for** (all pairs $(P, P') \in r \times r$) **do**	
(5)	**if** ($\text{root}(P) = \text{root}(P')$) **then**	common root found
(6)	$H = H \cup \Phi(P, \text{root}) \cup \Phi(P', \text{root})$	collect partial cycle
(7)	$r = r \setminus \{P, P'\}$	remove LP from rel
(8)	**fi**	
(9)	**od**	

FIND PARTIAL CYCLES (SINGLES)

(10)	**for** (all $P \in r$) **do**	
(11)	**if** ($\text{root}(P) = 1$) **then**	single cycle found
(12)	$H = H \cup \Phi(P, 1)$	collect partial cycle
(13)	$r = r \setminus \{P\}$	remove LP from rel
(14)	**fi**	
(15)	**od**	

Test for Double

```
(16)  if (|r| = 2 (r = {P, P'})) then          double found
(17)    V = V ∪ {P, P'}                        insert large primes
(18)    E = E ∪ {(P, P')}                      insert edge
(19)    Φ(P, P') = H                           update labeling
(20)    adjust_E(P, root(P), P', root(P'))     outdegree(v)=1
(21)  fi
```

Test for Single

```
(22)  if (|r| = 1, (r = {P})) then             single found
(23)    V = V ∪ {P}                            insert large primes
(24)    E = E ∪ {(P, 1)}                       insert edge
(25)    Φ(P, 1) = H                            update labeling
(26)    adjust_E(P, root(P), 1, 1)             outdegree(v)=1
(27)  fi
```

Test for Full

```
(28)  if (|r| = 0) then                        cycle found
(29)    output H                               print new cycle
(30)  fi
(31)  od
```

Table 3.8: Cycle Counting

method	# digits p	cycles needed	cycles found
S	65	19114	208985
Q	65	9125	361831
S	75	24139	224551
C	75	19018	637170
Q	85	56816	539238
C	85	54880	113858

Table 3.8 shows the number of cycles needed, which is the number of factor base elements minus the number of free relations minus the number of full relations. Sieving continues even when enough cycles are found, in order to improve the quality of the cycles, which can be measured by its length. This is the reason why the number of cycles found considerably exceeds the number of cycles needed when constructing the relation matrix.

3.5.2 Cycle Building

From the algorithm in section 3.5.1, we know sets of relations whose large prime exponents can be combined to 0 modulo 2. The large prime exponents of such a relation set may or may not be combined to 0 modulo q, where q is a very big prime number. As q is big, the partials have to be appropriately multiplied and divided by each other such that the large primes vanish after that operation.

The following example shows that not all subsets of relations which are suitable for the factoring case are also suitable for the DL case.

3.10 Example Assume we have three partial relations and the exponents of the three large primes $\mathfrak{Q}_1, \mathfrak{Q}_2, \mathfrak{Q}_3$ are as follows.

Element	$\mathfrak{Q}_1$	$\mathfrak{Q}_2$	$\mathfrak{Q}_3$
$c_1 + d_1\alpha$	1	1	0
$c_2 + d_2\alpha$	0	1	1
$c_3 + d_3\alpha$	1	0	1

Computing modulo 2 as in the factoring case we could multiply the three relations and get three large prime squares. But when computing modulo q as in the discrete

log case, this does not suffice; in fact the determinant is 2 which is $\not\equiv 0 \bmod q$ for $q > 2$.

Surprisingly, in most other cases the determinant is $0 \in \mathbb{Z}$. So the strategy is to construct cycles as in factoring and to try to combine them modulo q.

Generalizing example 3.10, we consider the corresponding matrix of large prime exponents for each cycle:

Element	$\mathfrak{Q}_1$	$\mathfrak{Q}_2$	...	...	$\mathfrak{Q}_l$
$c_1 + d_1\alpha$	$\mathrm{ord}_{\mathfrak{Q}_1}\, c_1 + d_1\alpha$	$\mathrm{ord}_{\mathfrak{Q}_2}\, c_1 + d_1\alpha$	...	...	$\mathrm{ord}_{\mathfrak{Q}_l}\, c_1 + d_1\alpha$
$c_2 + d_2\alpha$	$\mathrm{ord}_{\mathfrak{Q}_1}\, c_2 + d_2\alpha$	$\mathrm{ord}_{\mathfrak{Q}_2}\, c_2 + d_2\alpha$	...	...	$\mathrm{ord}_{\mathfrak{Q}_l}\, c_2 + d_2\alpha$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$c_k + d_k\alpha$	$\mathrm{ord}_{\mathfrak{Q}_1}\, c_k + d_k\alpha$	$\mathrm{ord}_{\mathfrak{Q}_2}\, c_k + d_k\alpha$	...	...	$\mathrm{ord}_{\mathfrak{Q}_l}\, c_k + d_k\alpha$

Set $m_{ij} := \mathrm{ord}_{\mathfrak{Q}_j}\, c_i + d_i\alpha$ and consider the matrix $M := (m_{ij})$, $1 \le i \le k$, $1 \le j \le l$. Sieving with large primes produces not more than a prescribed amount of entries per row of M – for example when using the quadruple large prime variation, at most four entries per row. Additionally, we know from the graph algorithm that in a cycle, every large prime (represented as a vertex in the graph) is contained in exactly two relations. This means every column of M contains exactly two entries. We are searching for a linear combination of the rows,

$$v_1 m_1 + v_2 m_2 + \cdots + v_k m_k = (0, \ldots, 0),$$

where each v_i is either 1 or -1. As a consequence of the latter property of M, prescribing one of the v_i determines the value of all other v_j, $1 \le j \le k$, $j \ne i$. The v_j can be considered sequentially. In our algorithm, we will store those row numbers j, where v_j has been set, in a queue. In case the entry v_i affects the entry v_j, two cases have to be considered

- v_j is not set – in this case we set v_j to $-v_i$,
- v_j has already been set – we test for $v_j = -v_i$; if this is true, we can proceed with the next queue entry, otherwise we have encountered a contradiction – no linear combination is possible, and we process the next cycle.

The following algorithm gives a formal description of this idea.

Combination of Cycles

INPUT: matrix m[][],rows,columns
OUTPUT: return 0: v[] linear combination of rows
return 1: no linear combination found

INITIALIZATION

```
(1)  q_in=q_out=0;                          queue empty
(2)  for (i := 2;i ≤ rows;i++) do           init linear combination
(3)    v[i]:= 0;
(4)  v[1]=1;                                set factor of row 1
(5)  row_queue[q_in++]=1;                   insert row 1
```

COLLECT ROW NUMBERS IN QUEUE

```
(6)  while (q_in> q_out) do                 queue not empty
(7)    i= row_queue[q_out++];               read next row number
(8)    for (j:= 1;j≤ columns;j++) do        for every column j
(9)      if (m[i][j]) then                  with entry in row i
(10)       for (k:= 1;k≤ rows;k++) do       find another row
(11)         if (k ≠ i) then
(12)           if (m[k][j]) then            with entry in col j
(13)             if (v[k]= 0) then          factor not set?
(14)               row_queue[q_in++]=k;     append row to queue
(15)               v[k]=-v[i];              and set factor
(16)               break;                   next row from queue
(17)             else
(18)               if (v[k]≠ -v[i]) then    parity violation
(19)                 return (1)             unsuccessful
(20)               fi
(21)             fi
(22)           fi
(23)         fi
(24)       od
(25)     fi
(26)   od
(27) od
(28)  return (0)                            successful
```

In table 3.9, we list the running time of this algorithm on a Sparc 20 workstation. As usual, the first column gives the NFS method, the second column the number of digits of p, the third column the maximal cycle length, the fourth the total number of cycles processed, the fifth the number of cycles which cause the above algorithm to return a successful linear combination, and the sixth the percentage of the successful returns.

Because of the high percentage, we may rely on the experimental evidence that most cycles are suitable for DL. We conclude that if the maximal cycle length is not too big, almost every cycle leads to a full relation. The running time of the cycle building step is only a fraction compared to the sieving or the linear algebra step.

Table 3.9: Statistic of Combining Partials

method	$\log_{10} p$	cycle max. length	#cycles	#cycles suitable	%	running time (h)
S	65	50	23015	19364	84.14	2.0
Q	65	3	112035	112035	100.00	0.7
C	75	3	26197	26190	99.97	0.1
Q	85	19	158883	158841	99.97	17.0
C	85	19	104864	104850	99.99	2.6

Full relations which are the result of more than 40 partial relations are not of practical use because their weight is too high. There is a correlation of the number of partials, which is denoted as cycle length, and the resulting weight – the weight of the corresponding row in the relation matrix – as shown in table 3.10. The DL problem considered here is the 85–digit problem solved by using the two–quadratics version of the NFS. With the data collected in table 3.10 we have a justification for the intuitive assumption that when reducing the cycle length, we automatically obtain a reduction of the row weight. As one would expect, this is true for the average weight, but we observe that the intervals overlap: pick for example a full relation with a row weight of 177. This could be the result of a cycle of length 11, 12, 13, or 14.

From the use of the NFS for factoring, we already know that further sieving greatly reduces the number of non–zero entries in the relation matrix. This is especially necessary for computing discrete logarithms, since the matrix equation is to be solved modulo a big prime. Every non–zero entry here leads to additional arithmetic operations with multiprecision integers. We therefore strongly recommend to continue with sieving even if enough relations have been found in order to build the matrix A.

Table 3.10: Correlation of Cycle Length and Row Weight

length	min weight	max weight	avg weight	# cycles
2	28	44	36.2	4322
3	43	63	52.6	3911
4	57	77	66.3	4241
5	70	93	81.3	5064
6	76	107	94.4	5935
7	96	121	108.7	6963
8	106	136	121.2	8312
9	120	150	135.0	9680
10	129	163	147.3	11046
11	144	177	160.5	12330
12	155	192	172.5	13012
13	169	205	185.5	13354
14	176	214	197.2	13595
15	186	228	209.9	12846
16	199	242	221.4	11825
17	213	258	233.9	9969
18	226	264	245.1	7774
19	232	278	257.4	4662

We illustrate this phenomenon in table 3.11, where once again the data are taken from the 85–digit DL problem solved with the two–quadratics NFS. The table lists the number of cycles of a given length which can be obtained after a computing time of 20 mips years and after 44 mips years. With a cycle–reducing strategy from Denny and Müller [18], we could further reduce the weight of the relation matrix as can be seen from the "cycle–reducing" column.

Table 3.11: Reduction of Cycle Length by Further Sieving

length	20 mips (y)	44 mips (y)	cycle–reducing
2	2260	4321	4322
3	1456	3911	3911
4	1224	4208	4241
5	1068	4928	5064
6	947	5740	5935
7	915	6458	6963
8	849	7433	8313
9	880	8323	9683
10	795	8975	11049
11	822	9833	12335
12	802	10331	13016
13	900	10897	13358
14	908	11385	13602
15	839	11773	12856
16	910	12349	11828
17	844	12393	9970
18	934	12872	7775
19	885	12753	4662

3.6 Computing Additive Characters

To compute a q-th power in $\mathcal{O}_j$ by multiplying elements of the form $c + d\alpha_j$, it is required that the product of the ideals $(c + d\alpha_j)$ is the q–th power of an ideal I of $\mathcal{O}_j$. Provided, we can construct a product

$$\prod_{(c,d)} (c + d\alpha_j)^{e_{(c,d)}} = I^q,$$

we know that I is principal unless the class number of $\mathcal{O}_j$ is divisible by q.

So in fact we have

$$\prod_{(c,d)} (c + d\alpha_j)^{e_{(c,d)}} = (\gamma_j)^q,$$

for some $\gamma_j \in \mathcal{O}_j$, which is equivalent to

$$\prod_{(c,d)} (c + d\alpha_j)^{e_{(c,d)}} = \eta_j \gamma_j^q,$$

for some $\eta_j \in \mathcal{O}_j^*$. What is needed is $\eta_j \in (\mathcal{O}_j^*)^q$. Since every unit of $\mathbb{Z}$ is an odd power, this condition is automatically satisfied on the rational side of the standard NFS relations.

The details of the technique to achieve this for general algebraic number rings have been described by Schirokauer [64]. In the following we give a brief overview of his technique in order to emphasize the tasks, which our implementation must carry out. For the rest of this section, let $\mathcal{O} := \mathcal{O}_j$ for $j = 1, 2$.

Since q does not ramify in $\mathcal{O}$, the prime ideal factorization of (q) can be written as

$$q\mathcal{O} = \prod_{\rho=1}^{r} \mathfrak{p}_\rho,$$

where the $\mathfrak{p}_\rho$ are mutually distinct. Consequently, we have the following decomposition of the ring $\mathcal{O}/(q)$ into fields:

$$\mathcal{O}/(q) = \prod_{\rho=1}^{r} \mathcal{O}/\mathfrak{p}_\rho.$$

Define an integer ϵ to be

$$\epsilon := \mathrm{lcm}_\rho\{N(\mathfrak{p}_\rho) - 1\}.$$

For each $\gamma \in \mathcal{O}$, it follows that

$$\gamma^{\epsilon} \equiv 1 \bmod \mathfrak{p}_{\rho} \quad 1 \leq \rho \leq r.$$

In particular $\gamma^{\epsilon} - 1 \in q\mathcal{O}$.

Define λ to be the following map:

$$\begin{array}{rccc} \lambda: & (\mathcal{O}, \cdot) & \longrightarrow & q\mathcal{O}/q^2\mathcal{O}(+) \\ & \gamma & \longrightarrow & \gamma^{\epsilon} - 1. \end{array}$$

This is actually a homomorphism of semi groups and a homomorphism on the group of units of $\mathcal{O}$.
We consider a special case of the main result of [64].

3.11 Proposition *Let γ be an element of $\mathcal{O}$ whose norm is not divisible by q. Let U be the group of units of $\mathcal{O}$. Let*

$$U' = \{\eta \in U \mid \eta \equiv 1 \bmod q\mathcal{O}\}.$$

Then γ is a q–th power in $\mathcal{O}$ if

i) the class number of K is not divisible by the prime q,

ii) $U' \subset U^q$,

iii) $ord_{\mathfrak{p}}(\gamma) \equiv 0 \bmod q$ for all prime ideals $\mathfrak{p}$ of $\mathcal{O}$,

iv) $\lambda(\gamma) = 0$.

We can write each cycle found by algorithm 3.5.2 in this way:

$$\frac{\gamma}{\gamma'} := \frac{\prod_{(c,d)} c + d\alpha}{\prod_{(c,d)} c + d\alpha}, \tag{3.12}$$

and compute the images of γ, γ' under λ modulo q^2 using the α–power basis of $\mathbb{Z}[\alpha]$ modulo $q^2\mathbb{Z}[\alpha]$.

We obtain

$$\begin{aligned}\lambda(\gamma) &= \sum_{j=0}^{n-1} b_j \alpha^j \bmod q^2\mathcal{O} \\ \lambda(\gamma') &= \sum_{j=0}^{n-1} b'_j \alpha^j \bmod q^2\mathcal{O} \\ \lambda(\frac{\gamma}{\gamma'}) &= \sum_{j=0}^{n-1} (b_j - b'_j) \alpha^j \bmod q^2\mathcal{O}.\end{aligned}$$

But all the images under λ are multiples of q. Therefore we can divide each $b_j - b'_j$ by q and then take the sum modulo q instead of computing modulo q^2.

With the help of this argument the coefficients $b_j - b'_j$ of the image under λ can be supplied to the exponent vector of the prime ideals. As a consequence, the exponent vector gets extended by n entries.

Now let us shortly comment on the conditions *i) – iv)* of proposition 3.11. In [64], it is referred to the Cohen–Lenstra heuristics, that *i)* is satisfied with probability $1-1/q$, and a heuristic argument is given that also *ii)* can be assumed with probability $1 - 1/q$. Next, the purpose of the solution of the matrix equation in the linear algebra step is to provide exactly conditions *iii)* and *iv)*.

This concludes the construction of q–th powers in number fields. To achieve this, it is merely necessary to be able to compute powers of algebraic elements, and this is straightforward.

3.12 DL–Example From

$$\begin{aligned}g_1(X) &= 7X^2 + 20X - 7 \equiv 7(X - 2977)(X - 4882) \bmod 5003 \\ g_2(X) &= 4X^2 - 21X + 8, \text{ irreducible mod } 5003,\end{aligned}$$

we obtain $\epsilon_1 = 5002$, $\epsilon_2 = 5003^2 - 1 = 25030008$.

For the relation corresponding to the pair $(c, d) := (-1, 2)$, we compute the additive character of $-1 + 2\alpha_1$, $-1 + 2\alpha_2$ by evaluating

$$\begin{array}{lllllll}(-1+2\alpha_1)^{5002} & -1 & \equiv & 14673799 & +1580948\alpha_1 & \text{mod} & 5003^2\mathcal{O}_1 \\ (-1+2\alpha_2)^{25030008} & -1 & \equiv & 8319989 & +19386625\alpha_2 & \text{mod} & 5003^2\mathcal{O}_2.\end{array}$$

In the linear algebra step, the exponent vector corresponding to $(-1, 2)$ will then be extended by the four entries

$$\frac{1}{5003}(14673799, 1580948, 8319989, 19386625).$$

As for the running times in table 3.12, it is clear that the smaller the degree of the field polynomial is, the faster ϵ can be computed. The size of ϵ depends on how the polynomial splits modulo q. The running time per cycle strongly depends on the size of ϵ. The size of ϵ, however, is not a criterion for the choice of the polynomial, since computing the additive characters merely consumes a fraction of the time for the whole DL computation.

Table 3.12: Average Running Time for Additive Characters

method	$\log_{10} p$	$\log_{10} \epsilon$	running time (s)	
			computing ϵ	per cycle
S	65	257	8.6	9.2
Q	65	65	0.2	0.2
S	75	149	4.5	4.1
Q	85	85	0.4	0.8

The cycle length does not at all affect the running time of the computation of the additive characters corresponding to this cycle. This is because we first compute the algebraic element γ/γ' of (3.12). Then we apply the map λ by computing the ϵ–th power of them. But to evaluate the ϵ–th power of an algebraic integer is far more expensive than to compute a product of a few algebraic integers. So we may expect that the powering dominates the running time. A short experiment with a 626–digit ϵ taken from the McCurley challenge and cycle lengths between 2 and 30 shows that one cannot even distinguish the different cycle lengths by considering the average running time. For each cycle length, 100 cycles have been considered (table 3.13).

We achieved a speed–up of about 16 % of the running time for polynomials of the special form $X^n + c$. This was especially useful for computing the additive characters of the McCurley challenge. For the same cycles as considered in table 3.13, we observed the following average running times per cycle (table 3.14).

Table 3.13: Running Time Additive Characters for Different Cycle Lengths

cycle length	running time / cycle (hsec)
2	6355
3	6330
4	6462
5	6438
6	6442
7	6445
8	6800
9	6477
10	6517
11	6403
12	6287
13	6047
14	6447
15	6767
16	6467
17	6361
18	6671
19	6428
20	6459
21	6452
22	6346
23	6472
24	6361
25	6093
26	6396
27	6103
28	6237
29	6219
30	6051

Table 3.14: Speed Up Additive Characters for Special Polynomials

cycle length	running time / cycle (hsec)
2	4943
3	4891
4	4425
5	4438
6	4827
7	4774
8	4991
9	4989
10	4880
11	4935
12	4876
13	4923
14	4973
15	4877
16	4989
17	4928
18	4927
19	4856
20	4887
21	4846
22	4889
23	4881
24	4924
25	4925
26	4966
27	5051
28	4859
29	4878
30	5001

3.7 Linear Algebra

In contrast to factoring by means of the number field sieve, linear algebra modulo a prime takes a considerable amount of CPU time within the whole DL algorithm. We briefly summarize the results from [17], which lead to the final form of the implementation. Solving linear systems still cannot be distributed among the comparatively cheap power of independent workstation clusters because of the heavy communication involved. We did our matrix computations mainly on the massively parallel Paragon machine with 136 nodes at the Kernforschungszentrum Jülich/Germany. At the beginning, we were provided with a structured Gauss implementation combined with parallelized ordinary Gaussian elimination for dense matrices. Unfortunately, pursuing this route consumes too much main memory, when building the dense matrices. Nevertheless, we started using these algorithms for matrices up to 3754×3494, a sufficient size when solving 50–digit DL problems.

3.13 Example The structured Gaussian elimination is usually expected to reduce the number of columns to one third. This means, a 24000×24000 sparse system can be transformed into a 8000×8000 dense system. Let the entries of the dense matrix be numbers up to 10^{87}. Such numbers consume 40 bytes each. This would require $8000 \cdot 8000 \cdot 40$ bytes, that is 2.56 GB of main memory.

A substantial speed–up of the linear algebra step is gained when the Lanczos method is used. The best performance has been achieved with a parallelized Lanczos implementation from Denny [17]. In table 3.15 we list the running–times on a Sparc 20 workstation and on the parallel Paragon machine combined with the information about how many nodes were used.

Table 3.15: Running Time of Linear Algebra Step

	$\log_{10} p$	dimension	method	time	machine	# nodes
S	50	3754 × 3494	S–Gauss	74 m	Paragon	136
S	65	20442 × 19957	Lanczos	38 h	Paragon	50
Q	65	20340 × 20330	Lanczos	19 h	Sparc 20	1
S	75	25085 × 25070	Lanczos	25 h	Paragon	64
Q	85	175046 × 70342	Lanczos	64 h	Paragon	64
C	85	119951 × 69984	Lanczos	23 d	Sparc 20	1
S	129	40015 × 40000	Lanczos	30 d	Sparc 20	1

3.7.1 Sketch of the Lanczos Algorithm

Although applying the Lanczos Algorithm is standard in modern factoring and discrete log computation we give a brief sketch here, for the sake of completeness. The reader will find a more detailed discussion about this topic in [34], [47]. The parallel implementation, which is used to compute the solutions to the DL problems discussed in this thesis, is extensively described in [17].

Let K be a finite prime field. Given a symmetric matrix $A \in K^{n \times n}$ with $\det(A) \neq 0$ and a vector $w \in K^n$, the Lanczos algorithm solves the system

$$Ax = w, \quad x \in K^n.$$

The algorithm iteratively computes a sequence of vectors as follows. Let

$$\begin{aligned} w_0 &:= w, \quad v_1 = Aw_0 \\ w_1 &:= v_1 - \frac{\langle v_1, v_1 \rangle}{\langle w_0, v_1 \rangle} w_o \end{aligned}$$

and then for $i \geq 1$, inductively

$$\begin{aligned} v_{i+1} &:= Aw_i \\ w_{i+1} &:= v_{i+1} - \frac{\langle v_{i+1}, v_{i+1} \rangle}{\langle w_i, v_{i+1} \rangle} w_i - \frac{\langle v_{i+1}, v_i \rangle}{\langle w_{i-1}, v_i \rangle} w_{i-1}. \end{aligned}$$

It can be proved that there exists $j \leq n$, such that $\langle w_j, Aw_j \rangle = 0$. If $w_j = 0$, then

$$x = \sum_{i=0}^{j-1} \frac{\langle w_i, w \rangle}{\langle w_i, v_{i+1} \rangle} w_i$$

is a solution of the system. In every iteration $i > 1$ the vector $v_{i+1} = Aw_i$ and three inner products are to be computed. In [17], it is shown how to avoid the computation of one of the three inner products; the effect is that the running time and the requirements of main memory are improved substantially. The trick makes use of the identity $\langle v_{i+1}, v_i \rangle = \langle w_i, v_{i+1} \rangle$ in the Lanczos iteration i.

Assume, we are faced with a linear system, which is not symmetric, for instance

$$Bx' = w', \quad B \in K^{m \times n}, w' \in K^m.$$

Assuming B having full rank, the transformation

$$B^t B x' = B^t w'$$

meets our requirements, with $A = B^t B$ and $w = B^t w$. In this case, A is not computed explicitely; instead, the matrix vector multiplication Aw_i in iteration i is replaced by evaluating $B^t(Bw_i)$.

3.7.2 Applying the Lanczos Algorithm

The Lanczos algorithm has the advantage of computing a solution of an inhomogeneous linear system very fast. Though, at first sight, it seems to have two drawbacks. For our purposes it is convenient to get a solution to a homogeneous system[1]. Furthermore, the NFS computes one logarithm per solution of the linear system, therefore we normally need more than one solution to our system. In the sequel of this section, we describe how to meet these two conditions.

Assume, we compute in $K := \mathbb{Z}/q\mathbb{Z}$, $B \in K^{m \times n}$, $m > n$, needing l solutions to

$$xB = 0.$$

We start by splitting B into two matrices, such that

$$B = \begin{pmatrix} B' \\ B'' \end{pmatrix}, \quad B' \in K^{m-l \times n}, B'' \in K^{l \times n}.$$

Let $b''_j \in K^{1 \times n}$, $1 \leq j \leq l$ be the rows of B''. We proceed by solving the n inhomogeneous systems

$$xB' = b''_j$$

simultaneously. According to the running times of [17], computing l simultaneous solutions with the Lanczos algorithm is far cheaper than computing l solutions sequentially. Although the value of l is limited by main memory constraints, this is not very severe restriction; on a parallel machine we can compute at least 40 simultaneous solutions without difficulty. For K with a characteristic of 65 decimal digits, it takes only twice as much time to compute 40 solutions simultaneously than to compute one solution to the linear system. These are sufficient to compute the logarithm of an arbitrary element in a prime field of such a size.

3.7.3 Computing Logarithms from Linear Algebra Solutions

Now we are ready to explain how to obtain the final results of the discrete log computation from our solutions obtained in the linear algebra step.

Let m be the total number of cycles, i.e. the number of rows of the relation matrix A, r be the number of primes we want to compute the discrete log of. Furthermore, let s be the number of columns with heavy weight, say more than 90 % of the entries are non-zero. Note that $s \geq n$ because the additive character columns are

[1] see the sketch of the NFS algorithm at the end of section 2.2

heavy (see section 3.6). Then we define the number of the rest of the columns as $t := |FBR| + |FBA| + n + 1 - s - r$.

Then A is of the form

$$A = (A'|A''|A''')$$

with $A' \in K^{m \times r}$, $A'' \in K^{m \times t}$, $A''' \in K^{m \times s}$.
A' consists of the exponents of elements $p_1, \ldots, p_r$ we want to compute the discrete logarithm of. We are interested in $r-1$ solutions of

$$x(A''|A''') = 0^T \tag{3.13}$$

in K.

As indicated in subsection 3.7.2, the Lanczos algorithm computes $s+r-1$ solutions of $xA'' = 0^T$ simultaneously; A'' plays the role of B there. We write the solutions as vectors $s_i := (s_{i1}, \ldots, s_{im})$, for $1 \leq i \leq s+r-1$. Define $S \in K^{s+r-1 \times m}$ to be the matrix consisting of the s_i.

We compute $B = SA''' \in K^{s+r-1 \times s}$ and $r-1$ solutions of

$$xB = 0^T$$

of the form $x_j := (x_{j1}, \ldots, x_{j,s+r-1})$, which we write as a matrix $X \in K^{(r-1) \times (s+r-1)}$.

Then the rows of $XS \in K^{r-1 \times m}$ are solutions to the original equation (3.13):

$$\begin{aligned} X \cdot S \cdot A'' &= X \cdot 0 &= 0 \\ X \cdot S \cdot A''' &= X \cdot B &= 0 \end{aligned}$$

We now construct linear combinations $XSA' = L \in K^{(r-1) \times r}$. For every row i of L, $(1 \leq i \leq r-1)$, we have

$$p_1^{l_{i1}} p_2^{l_{i2}} \cdots p_r^{l_{ir}} \equiv d_i^q \bmod p$$

for some $d_i \in (\mathbb{Z}/p\mathbb{Z})^*$.

Therefore

$$\begin{aligned} l_{11} \log p_1 + \ldots + l_{1r} \log p_r &\equiv 0 \bmod q \\ \vdots \qquad\qquad \vdots \qquad\qquad & \vdots \\ l_{r-1,1} \log p_1 + \ldots + l_{r-1,r} \log p_r &\equiv 0 \bmod q \end{aligned}$$

and we compute one non–trivial solution to

$$Ly \equiv 0 \bmod q, \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{pmatrix}.$$

Now choose a generator of $(\mathbb{Z}/p\mathbb{Z})^*$ among the $p_j,\ 1 \le j \le r$, say p_k and define

$$y' := \frac{1}{y_k} y = \begin{pmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_r \end{pmatrix}.$$

We end up with

$$\log_{p_k} p_j \equiv y'_j \bmod q, \qquad 1 \le j \le r \qquad y'_k = 1.$$

3.7.4 The Two–Quadratics Version

In this subsection we present how the DL solutions will be obtained, when applying the method of the two–quadratics adaption, which is discussed in section 3.2.4. We stick to the notation introduced there. In the two–quadratics version of the NFS the matrix A' of the preceding subsection is empty. Instead, r relations are changed. For the ease of exposition, assume $r = 2$; the logarithm of s to the base a shall be computed. Assume further we have replaced two relations (c_a, d_a), (c_b, d_b) by the relations (ac_a, ad_a), (bc_b, bd_b) respectively.

The solution to the linear system gives exponents $e_{c,d}$ of the elements $c + d\alpha_1$, $c + d\alpha_2$, such that the power products give two q–th powers simultaneously (see (3.10), (3.11)):

$$(a(c_a + d_a\alpha_1))^{e_{c_a,d_a}} \prod_{(c,d)} (c + d\alpha_1)^{e_{c,d}} = \gamma_a^q \tag{3.14}$$

$$(s(c_b + d_b\alpha_2))^{e_{c_b,d_b}} \prod_{(c,d)} (c + d\alpha_2)^{e_{c,d}} = \gamma_b^q. \tag{3.15}$$

Applying φ_1, φ_2, we get (3.16) from (3.14) and (3.17) from (3.15); note also Remark 2.2.

$$a^{e_{c_a,d_a}} \prod_{(c,d)} (c + dm)^{e_{c,d}} \equiv g_a^q \bmod p \tag{3.16}$$

$$s^{e_{c_b,d_b}} \prod_{(c,d)} (c + dm)^{e_{c,d}} \equiv g_b^q \bmod p, \tag{3.17}$$

for some $g_a, g_b \in \mathbb{Z}$.

For simplicity set $k := e_{c_a,d_a}$, $l := e_{c_b,d_b}$ and obtain

$$a^k s^{-l} \equiv (g_a/g_b)^q \bmod p$$

by dividing (3.16) by (3.17).

As before, Lemma 1.4 now tells us that the discrete log of s to the base a modulo q is $x \equiv k/l \bmod q$.

3.7.5 The COS Version

When computing logarithms with the COS method (subsection 3.2.1), the linear algebra step is a little bit easier. The reason is that each *ideal* relation of the NFS output already gives rise to an *element* relation modulo p:

$$\prod p^{e_p} \equiv cV + dT \equiv V\varphi(c + d\alpha) \equiv V \prod \pi^{e_\pi} \bmod p$$

with prime *elements* $\pi \in \mathcal{O}$. Note that α is an algebraic integer here.

Let $A \in K^{n \times m}$ be the matrix, where the element a_{ij} is defined by the exponent of prime element i in relation j. Then the vector

$$x := (\log p_1, \ldots, \log p_k, \log(\varphi(\pi_1)), \ldots, \log(\varphi(\pi_{n-k})))$$

solves

$$xA = 0. \tag{3.18}$$

When A has maximal rank $n - 1$, the solution space is one–dimensional, so all solutions are given by ax, $a \in K$. Therefore, given an arbitrary solution of (3.18), say

$$y := (y_1, \ldots, y_n) \in K^{1,n},$$

it yields the logarithm of each factor base element with respect to factor base element number i by the normalization

$$\frac{1}{y_i} \cdot y.$$

We see, the COS algorithm has the significant advantage of computing the logarithm of every factor base element from only *one* solution of the linear system. Furthermore, the partial relations allow to compute iteratively the logarithms of almost all large prime elements, which occur in the set of partial relations.

3.8 Computational Results

The purpose of this section is to collect the outstandingly successful discrete logarithm computations in finite prime fields of large order. We begin with the 129–digit challenge of McCurley in 3.8.1, then describe the biggest example with the standard NFS method in 3.8.2, which has been beaten by the two–quadratics adaption from factoring (description in 3.8.3). Finally, we present a new record with the COS Gaussian integer method in 3.8.4.

3.8.1 The McCurley Challenge

Our implementation, instantiated by the standard NFS method, has been used to compute the logarithms of 3, 5, 11, 23, 31, 67, 7351, and 11287 to the base 7 in $\mathbb{Z}/p\mathbb{Z}$, where

$$\begin{aligned} p &= (739 \cdot 7^{149} - 736)/3 \\ &= 204706270385532838059744535166974274803608394340123459695798674591526591372685229510652847339705797622075505069831043486651682279. \end{aligned}$$

The difficulty of computing discrete logarithms in this field is due to the 126–digit prime factor q occuring in the factorization of

$$p - 1 = 2 \cdot 739 \cdot q,$$

with

$$\begin{aligned} q &= 138502212710103408700774381033135503926663324993317631729227790657325163310341833227775945426052637092067324133850503035623601. \end{aligned}$$

This is the field presented by McCurley in his challenge problem [44].

The practical experience obtained so far suggests that one should consider number fields of degree $n = 3, 4, 5, 6$ and examine the probability of finding relations over two factor bases of optimal size. In order to construct suitable polynomials of such degrees, we may use the identities

$$\begin{aligned} 21p &= 739 \cdot (7^{50})^3 - 5152 \\ 3p &= 5173 \cdot (7^{37})^4 - 736 \\ 21p &= 739 \cdot (7^{30})^5 - 5152 \\ 21p &= 739 \cdot (7^{25})^6 - 5152. \end{aligned}$$

We therefore have the choice between the pairs of polynomials

$g_1(X)$	$=$	$X - 7^{50}$	$g_2(X)$	$=$	$739X^3 - 5152$
$g_1(X)$	$=$	$X - 7^{37}$	$g_2(X)$	$=$	$5173X^4 - 736$
$g_1(X)$	$=$	$X - 7^{30}$	$g_2(X)$	$=$	$739X^5 - 5152$
$g_1(X)$	$=$	$X - 7^{25}$	$g_2(X)$	$=$	$739X^6 - 5152$

Starting from the previous 65–digit record, we decided to end up with a 40000×40000 system in the linear algebra step. In order to compare the four different possible choices of the degree, we look at the values $N(c+d\alpha_1)$, $N(c+d\alpha_2)$ to be decomposed over the factor bases. On the rational side $\mathbb{Z} = \mathcal{O}_1$, we get $c + dm \approx dm$; on the algebraic side, we obtain $h_2 \cdot c^n - a_{2,0}d^n \approx h_2 \cdot c^n$. As we expected to need a sieving rectangle of $10^6 \times 10^6$, we got table 3.16 with the aid of the ρ–function.

Table 3.16: Comparing different degrees – McCurley challenge

degree	m	dm	h_2c^n	$\lvert FB_1\rvert$	$\lvert FB_2\rvert$	# trials per full
3	$1.8 \cdot 10^{42}$	$1.8 \cdot 10^{48}$	10^{21}	19800	20200	$3.7 \cdot 10^{11}$
4	$1.9 \cdot 10^{31}$	$1.9 \cdot 10^{37}$	10^{28}	19900	20100	$6.2 \cdot 10^{9}$
5	$2.3 \cdot 10^{25}$	$2.3 \cdot 10^{31}$	10^{33}	19600	20400	$3.7 \cdot 10^{9}$
6	$1.3 \cdot 10^{21}$	$1.3 \cdot 10^{27}$	10^{39}	16400	23600	$1.1 \cdot 10^{10}$

As the expected number of trials for a full relation do not differ very much when choosing the degrees 4 and 5, we started sieving with both degree 4 and degree 5. After 4 mips years sieving, degree 5 turned out to have slightly better chances.

So the polynomial used in this case was $g_2(X) := 739X^5 - 5152$. Notice that it is the special form of p that allows the construction of a suitable polynomial with such small coefficients. Using the notation introduced in section 2.2, let α_2 be a root of $g_2(X)$, φ_2 be the corresponding homomorphism from $\mathbb{Z}[739\alpha_2]$ to $\mathbb{Z}/p\mathbb{Z}$. The small size of the coefficients of f are clearly a great advantage. The smaller the values of $d^n f(c/d)$ are, the more likely it is that they are divisible by only small primes.

For the computation of the logarithms listed below, two factor bases containing 20,000 elements each were used. Each element whose logarithm was computed was in the factor base. As a result, there was no need to find smooth pre-images under φ_2 of these elements. The sieving interval for c and d was

$$\begin{aligned} -15 \cdot 10^6 \leq c \leq & \ 15 \cdot 10^6 \\ 1 \leq d \leq & \ 10^6. \end{aligned}$$

After using 48.5 mips years of idle time on 110 Sparc workstations (nearly all of type ELC rated at 21 mips), the following amount of relations has been found:

type of rel	# after sieving	# after filtering
fulls	2826	2826
singles	37046	32261
doubles	183383	141120
triples	410843	283197
quadruples	332133	210381
total	966231	666959
# of lp	797794	476883

This led to 190077 full relations. As only 40000 full relations were needed we have had a bound of maximal 57 partials per full relation (average 36). The resulting matrix, however, consisted of too many non–zero entries to compute a linear algebra solution within acceptable time.

Further sieving, reaching 110.6 mips years, led to more than 300000 cycles as an effect of the cycle explosion phenomenon ([20], [78]).

type of rel	# after sieving	# after filtering
smalls	3199	3199
singles	42407	38446
doubles	211888	176307
triples	478543	369116
quadruples	388685	282832
total	1124722	869900
cycles	306717	

The large amount of new cycles has also the advantage of getting far more short cycles – here a new algorithm of combining cycles shows its worth [18]. The resulting matrix is more sparse; a full relation consisted of maximal 22 partials (15 on average).

It took 1 min per cycle on a Sparc 20 to get the additive character columns. This was parallelized in a trivial way by distributing the cycles among several machines. The special form of f allowed to reduce the number of bit operations when multiplying two algebraic numbers by exploiting the fact that $(739\alpha)^5 = 1536574451494432$; see section 3.6.

The computation of the 15 solution vectors of the 40015×40000 linear system modulo the 126–digit prime factor of $p - 1$ was done on three Sparc 20 stations within a month by using the Lanczos algorithm [17].

Proceeding as described in subsection 3.7.3 we then derived the logs of the values of the equation (3.19) below, which are in our factor base.

What follows is a list of these logarithms computed on March 4, 1996.

$\log_7 3 =$
68860094399350245342602688357969433483445840039152871728566347529
56578596486372578418538224292870419125241916672093665622728752 0

$\log_7 5 =$
11879221270981906956222381382324106470112756903551534511004275082
35003882863259426051929551146640354628139096587652974388356561 26

$\log_7 11 =$
14172715990276841722156670601045288532560462777922377368290431150
17790364274980056844894488159939005533847954985398015726177955 60

$\log_7 23 =$
15511605859077805410959549453221030604260697443327496808471308380
90775543263809825008675103756835085289602612181051956155343558 04

$\log_7 31 =$
99355352742967588716252377905072644977502899916228947274361609866
91477915956266874779141451027144753634888847530469721807314238 0

$\log_7 67 =$
18524527685763603567792984693345199476969677751442057713807059 66
98587572970022624767383788051783719066798580382374055617496228 94

$\log_7 7351 =$
14463196894490829567073226898360380983979979583855805954303228348
52079492445878021058205245682070769476999126034278024170206345 93

$\log_7 11287 =$
79051812480562894353242540763178671604799913001076931312515301149
37529830547665373448844397449013712704980829834525824668962996 5.

It is the McCurley challenge that remains, namely to compute the logarithm to the

base 7 of

$$\begin{aligned} b &= 12740218011997394682426924433432284974938204258693162165455 7 \\ &\quad 73529032291467909599868186097881304659516645545814428058807 6 \\ &\quad 766033781. \end{aligned}$$

Using the congruence

$$7^{83} \cdot b \equiv \frac{s}{t} \equiv \frac{3 \cdot 5 \cdot 11 \cdot 23 \cdot 11287 \cdot p_8 \cdot p_{10} \cdot p_{17} \cdot p_{23}}{31 \cdot 67 \cdot 7351 \cdot 402869 \cdot p_{13} \cdot p'_{13} \cdot p_{26}} \bmod p \tag{3.19}$$

where

$$\begin{aligned} p_8 &= 10547587 \\ p_{10} &= 2916781859 \\ p_{13} &= 2599909498829 \\ p'_{13} &= 3598631011739 \\ p_{17} &= 51337921071904669 \\ p_{23} &= 22761868782949840132373 \\ p_{26} &= 77731271923481246820848221, \end{aligned}$$

the problem was reduced to computing the logarithms of the relatively small factors of s and t. Indeed, as we have seen, the logarithms of the smallest factors have already been computed. In order to compute the logarithm of any of the bigger prime factors by means of the number field sieve with $\mathbb{Z}[\alpha_2]$ given as above, one must find a smooth element in $\mathbb{Z}[\alpha_2]$ which is mapped to that prime by φ_2. No good method has yet been found to accomplish this. If one is willing to give up the attractive polynomial $f(X) = 739X^5 - 5152$, then certainly there is the method used for arbitrary integers, to find an alternative polynomial which will generate a suitable number ring. It is not yet feasible, however, to pursue this route for the primes listed here. The problem is that the coefficients of the polynomials under consideration are too big. As a result, the factor base has to be increased in order to find enough smooth pairs in the corresponding number ring. Unfortunately, then the size of the factor base is too big for the available implementations of linear algebra modulo a large prime. We note that current NFS factoring efforts do, in fact, use number rings given by polynomials with coefficients of the size we are discussing. In this case, however, the linear algebra is done mod 2.

3.8.2 The Standard–NFS Record

With the standard version of the NFS (section 3.2.2), our implementation set two records in computing logarithms of factor base elements – one with p having 65 decimal digits on September 29, 1995 and one with p having 75 decimal digits on March 25, 1996. The parameters, the NFS was configured with, are shown in table 3.17 together with the resulting amount of relations and cycles. In order to make the discrete logarithm computation difficult, we chose p such that $(p-1)/2$ is prime. The choice of the polynomials, the factor base sizes and the large prime bounds is a straightforward application of the discussion in the sections 3.2 and section 3.3.

We present the discrete log problems solved modulo the 65–digit prime computed on September 29, 1995.

$$p_{65} = 31081938120519680804196101011964261019661412191103091971180537759.$$

$$\begin{aligned}
\log_7 2 &= 12947465376923824724957499951503053332437571430268512704320339344\\
\log_7 3 &= 22080187724931255875760876853515374478171170414218024970175409792\\
\log_7 5 &= 90203601220544716377527643224213256109908926455294991774140561 96\\
\log_7 11 &= 99455510732446733201773881405622850169029168883281944745441 06942\\
\log_7 13 &= 15156730731943267081963372032905052327723905195485355154769047594\\
\log_7 17 &= 81526596391616298526166602372244543966918059690321824435206700 07\\
\log_7 19 &= 84294389427220421836113045200830183852429877608312278878698274 58\\
\log_7 23 &= 29153020481930701437148309607402611581505734463034646141828747593\\
\log_7 29 &= 22997906266006136529682973437413418001682127605800489536168728807
\end{aligned}$$

We append the discrete log problems solved modulo the 75–digit prime on March 25, 1996.

$$p_{75} = 310819381205196808041961141219110101196426101966030919711805194127121999327$$

$$\log_5 2 = 247304965037952295066692348548903196491045004838332188604053856130900104778$$

$$\log_5 3 = 960297796467834075340248339229935989153611666307790134329861078196676200 6$$

$$\log_5 7 = 147001657721304571107460703308056871315960324818746059568618613166225193632$$

$$\log_5 11 = 253762437654824910964035857039763668258075514706017456372999633994434110941$$

$$\log_5 13 = 204890434527986803740486893000176797293960297959843422287988074270476059103$$

$$\log_5 17 = 994367469189707170348518718066396458166481490556625301852160022209620983 91$$

$$\log_5 19 = 304826238078240870953965599601565525277161778098196660032717603407232004336$$

Table 3.17: The 65– and 75–digit Standard NFS

prime	p_{65}	p_{75}
poly g_1		
m	13277827521354825	4198817734636290744
poly g_2		
coeff. X^4	-5796988	41440163
coeff. X^3	-1040988700418	8899586579547
coeff. X^2	-1599410033377	50013054105621
coeff. X^1	2467898905167	-385158712921327
coeff. X^0	2804774217242	-226856042090363
# rat. FB	3000	5000
# alg. FB	16954	20058
lp bound rat	$2 \cdot 10^6$	$5 \cdot 10^6$
lp bound alg	$6 \cdot 10^6$	10^7
max c	$4 \cdot 10^6$	10^7
max d	$5 \cdot 10^5$	$1.2 \cdot 10^6$
time sieve (mips)	5 y 116 d	70 y 16 d
collected rels		
full rels	745	714
single rels	12695	13879
double rels	84904	107723
triple rels	266067	385844
quadruple rels	316388	534231
filtered rels		
single rels	11907	12471
double rels	74183	87473
triple rels	218865	286887
quadruple rels	248539	367262
cycles	208985	224551
cycles needed	19114	24139

3.8.3 The Two–Quadratics Record

The new two–quadratics version for DL set a record on September 23, 1996, when we solved

$$\begin{aligned} 59^x &= 29 \bmod p \\ 59^y &= 53 \bmod p, \end{aligned}$$

where

$$\begin{aligned} p = {} & 31081938080419611412191112051968261019660101196403 \\ & 09197118051941271219700607191207059 \end{aligned}$$

is a prime of 85 decimal digits with $q := (p-1)/2$ prime.

We computed the two solutions

$$\begin{aligned} x = {} & 30510320398109765754475052908348052559852331892660 \\ & 22096531322524429784944990676327395 \end{aligned} \quad \bmod \; p-1$$

$$\begin{aligned} y = {} & 12445261273448784489646035237768063038577529049189 \\ & 59081249797500704003169233661409571 \end{aligned} \quad \bmod \; p-1.$$

The two quadratic polynomials, which defined the auxiliary number rings were chosen as

$$\begin{aligned} g_1(X) &= 12088651913597925810 \cdot X^2 \\ &\quad +905452079113038068089 \cdot X \\ &\quad +8749043915900881108603 \\ g_2(X) &= 1146890895334804811 \cdot X^2 \\ &\quad +5297984501155169639345 \cdot X \\ &\quad -3247049136460419754715. \end{aligned}$$

They have the common root

$$\begin{aligned} m = {} & 90032406615008104576059194778390117845110494177770 \\ & 74681938816180778489604342897798 43 \end{aligned} \quad \bmod p.$$

The factor bases were of size 35346 and 34995 respectively.

The sieving procedure was done on 100 workstations using their idle time of 44.5 mips years.

Computing 10 dependencies among the rows of the resulting 175046×70342 linear system modulo q was done by the following steps:

- a compactification step, which constructed a 54866×54856 system
- the Lanczos algorithm on the parallel PARAGON machine at the KFA in Jülich/Germany within 64 hours on 64 nodes.

Due to the compactification step, the computing time for the linear algebra step was reduced by more than 30%. We refer the reader, who is interested in the subject of minimizing the running time of a parallel Lanczos implementation by compactification, to [17].

3.8.4 The COS Record

On November 24, 1996, a logarithm of an "arbitrary" element was computed.

We recomputed our NFS example from September 23, 1996, when we obtained two logarithms of factor base elements log(29) and log(53) to the base 59. We configured our NFS implementation by the COS method in order to obtain a comparison of the COS and the NFS.

With

$$p_{85} = 31081938080419611412191112051968261019660101196403091971180519412712197006071912070 59$$

(85 decimal digits, 281 bits), $q := (p-1)/2$ prime,

we solved the following problem

$$2^x \equiv 31415926535897932384626433832795028841971693993751058209749445923078164062862089 9862 \bmod p.$$

Note that the right hand side consists of the first 84 digits of π.

We found the solution

$$x \equiv 75682328830687872815850309300288240821108757674368163695803006547760748172040286919 2 \bmod p-1.$$

We used the Number Field $\mathbb{Q}(\sqrt{-2})$, so the relations consisted of simultaneously smooth values of

$$f_1(X,Y) = 1323274340819980392303558671985532821598359 * X + 8237532479357539733978757237386763941839 67 * Y$$

and

$$f_2(X,Y) = X^2 + 2Y^2$$

with common root

$$(m,1) \equiv (21623229451881471841110284273561032206560208349063574889053122828729253965356256119 03,1) \bmod p.$$

The factor bases were of size 58000 and 11981 respectively.

The sieving procedure was done on 120 workstations using their idle time of about 30.6 mips years. This is faster than using NFS with two quadratic polynomials (44.5 mips years with equal total factor base size).

Computing one dependency among the columns of the resulting 119951×69984 linear system modulo q was done by the following steps:

- a compactification step, which resulted in a 51855×51855 system
- the Lanczos algorithm on a Sparc 20 station within 23.2 CPU days using 35 MB of main memory.

As decribed in subsection 3.7.5, the solution of the linear system almost immediately gives the logarithm of the factor base elements. But we did not get the logarithm of all factor base elements at once because the compactification step removes some relations from the original system for the sake of efficiency. This was the case for 1088 factor base elements. With the aid of the full relations, the logarithms of these elements have been computed in a negligible amount of time.

In order to be able to compute logs of arbitrary elements, we extended our table of the 69981 factor base logs by creating a data base of 626419 logs of elements with norm up to 10^7.

These were obtained within less than two hours on a Sparc 20 workstation from the partial relations collected during the sieving step.

The log of the element above was derived from the following identities:

$$\begin{aligned} & 3141592653589793238462643383279502884197169399375105820974944592307816406286208998 62 \\ \equiv\ & -1107911020245284271895336948767925749763403 \\ & /12383853456341283587269734548824840418395 9 \\ \equiv\ & -7 \cdot 61 \cdot 25946393916751388100593371165525193202 89 \\ & /13 \cdot 2207 \cdot 3779 \cdot 5053313 \cdot 38665007 \cdot 78959357 \cdot 74034701813 \bmod p. \end{aligned}$$

Here, the 40–digit factor

$$p_{40} = 2594639391675138810059337116552519320289$$

and the 11–digit factor

$$p_{11} = 74034701813$$

were replaced by 10^{10}–smooth expressions.

By applying the reduction step of section 3.1 (8 hours on Sparc 20), we found that

$$p_{40} \equiv \frac{33613 \cdot 40829 \cdot 83617 \cdot 851761 \cdot 2115961 \cdot 2443219 \cdot 4287211 \cdot 4976687}{2^2 \cdot 19 \cdot 6803 \cdot 8387 \cdot 59387 \cdot 152239 \cdot 586501 \cdot 628997 \cdot 18636193 \cdot 210112139} \bmod p.$$

By sieving (33 minutes on Sparc 20), we found that

$$p_{11} \equiv \frac{-3 \cdot 17 \cdot 37 \cdot 1109 \cdot 6199 \cdot 24989 \cdot 46957 \cdot 120661 \cdot 936667 \cdot 4133219 \cdot p_9}{2^{30} \cdot 5^{29} \cdot 13 \cdot 727 \cdot 1303 \cdot 2399 \cdot 9157 \cdot 32251 \cdot 630299 \cdot 3862493 \cdot 5308663 \cdot p_{9'}} \bmod p,$$

where $p_9 = 515357041$ and $p_{9'} = 422591069$.

The logarithms of a prime number s, with $10^7 < s < 10^{10}$ were found by lattice sieving (1:15 min on Sparc 20 each). The lattice sieve tested the expressions

$$\frac{f_1(c,d)}{r} \text{ and } f_2(c,d)$$

for smoothness over the factor base.

Table 3.18: Lattice Sieve for p_{40}

type	# relations
single	0
double	9
triple	35
quadruple	51
quintuple	5

Table 3.19: Lattice Sieve for p_{11}

type	# relations
single	2
double	16
triple	35
quadruple	43
quintuple	2

In order to find the logarithm of p_{40}, the lattice sieve was carried out for 8 different r's by setting the sieving rectangle to

$$-4000 \leq \lambda \leq 4000, \quad 1 \leq \mu \leq 1800.$$

With these parameters, we found the amount of relations shown in table 3.18.

By the use of the same sieving rectangle, for the prime p_{11}, the lattice sieve was carried out for 7 different r's and produced the output as shown in table 3.19.

Chapter 4

A General Sieving Device

Within the last decade, many implementations of different sieving algorithms applicable to the discrete logarithm problem and to the factoring problem have been developed [5, 78, 23, 16, 76]. At first sight, these algorithms require a different sort of input; this is probably the reason for specialized implementations. Every implementation usually suffers from the well–known software problems when it is done from scratch. For each implementation, it was required to invent a design, to debug code tediously, to test executables for different input, and to speed up code in the most time consuming parts of the program. Within this chapter, we present a generalized implementation, which covers all the current sieving techniques both for DL and factoring, and which is suitable for at least two new sieve applications. Of course, it is necessary to compare the running time to the existing, specialized implementations. For each algorithm, we will describe its configuration within our sieving device and show that we do not invest too much running time for the generalization.

The sieving algorithms in question are the Coppersmith–Odlyzko–Schroeppel method for DL, the Quadratic Sieve for factoring and the Number Field Sieve for both DL and factoring. Here we add the sieving method of section 3.1, used for the reduction step in the DL problem. Furthermore, we add a Number Field Sieve application for computing class groups of Number Fields [10].

We now proceed by reviewing the purpose of the different sieving procedures. For a bound B, the Quadratic Sieve searches for values $f(x)$ of a quadratic polynomial $f \in \mathbb{Z}[X]$, which are smooth over a set of *prime numbers* with absolute value below B. For bounds B_1, B_2, the Number Field Sieve aims to find principal ideals $(x + y\alpha_1)$, $(x + y\alpha_2)$ of number rings which are simultaneously smooth over a set of *prime ideals* with norm below B_1, B_2. Given bounds as above, the Gaussian integer (COS) method aims to find algebraic integers $x + y\sqrt{-r}$, which are smooth over a

set of *prime elements* with norm below B_1, B_2. The two latter methods make use of the fact that the norm of elements $x + y\alpha$ of a number ring can be expressed by a *homogeneous* polynomial

$$f(X,Y) = a_n X^n + a_{n-1} X^{n-1} Y + \cdots + a_1 X Y^{n-1} + a_0 Y^n \in \mathbb{Z}[X,Y]. \tag{4.1}$$

The sieving region is bounded by four values $x_{\min}, x_{\max}, y_{\min}, y_{\max} \in \mathbb{Z}$, such that we are looking for $f(x,y)$ smooth within the rectangle

$$x_{\min} \le x \le x_{\max},\ y_{\min} \le y \le y_{\max}.$$

The sieving usually proceeds by keeping y fixed while obtaining suitable x's by sieving the univariate polynomial $f(X,y)$.

4.1 Specification

To unify the interfaces of the sieving algorithms, we choose the input of the generic sieving device to be:

1. the number of simultaneous polynomials k
2. homogeneous polynomials $f_1, \ldots, f_k \in \mathbb{Z}[X,Y]$
3. (a) factor base bounds $B_1, \ldots, B_k \in \mathbb{N}$, or
   (b) factor bases $F_1, \ldots, F_k \subset \mathbb{N} \times \mathbb{N}$
4. large prime bounds $L_1, \ldots, L_k \in \mathbb{N}$
5. values $x_{\min}, x_{\max}, y_{\min}, y_{\max} \in \mathbb{Z}$.

4.1 Remark Within the factorization of $f_j(x,y)$, $1 \le j \le k$, the data structure is prepared to allow arbitrary many large prime factors r with $B_j \le r \le L_j$. But the experimental data collected in this chapter is gained by restricting to one large prime factor for each $f(x,y)$, This is because one large prime for each polynomial covers the range up to at least 60 decimal digits for factoring and discrete log. This is enough for testing the reliability and the performance of the involved data structures. By adding a factorization procedure designed for composites up to 20 decimal digits such as Shanks's square–form–factorization method [63] or Pollard's ρ–method [56], one can use two or more large primes per polynomial.

The output of our sieving device shall be a set S of pairs $(x, y) \in \mathbb{Z} \times \mathbb{Z}$, with $f_j(x, y)$ is L_j–smooth in the sense of remark 4.1, together with a prime factorization of $f_1(x, y), \ldots, f_k(x, y)$. Trading rigor for speed, the set S will usually be a true subset of the set of all smooth values within the sieving region. Omitting some of the smooth values is not a severe restriction, as has already been shown by experiments of the specialized implementations mentioned at the beginning of this chapter.

4.2 Factor Bases

Of course, a factor base contains primes up to a bound B, alternatively, the number of elements in the factor base can be adjusted. It is essential to know where a prime q divides $f(x, 1)$. Given this information, the locations where q divides $f(x, y)$, can be determined. Assume $f(x, 1) \equiv 0 \bmod q$. From (4.1), we see that

$$f(xy + kq, y) \equiv y^n f(x, 1) \equiv 0 \bmod q, \quad k, y \in \mathbb{Z}.$$

For different roots of $f(X, 1) \bmod q$, we therefore get different sets $\{(x, y) \mid f(x, y) \equiv 0 \bmod q\}$, for which $f(x, y)$ is divisible by q. Consequently, for each prime q with $q \leq B$, we compute the roots $r_1, r_2, \ldots, r_q$ of $f(X, 1) \bmod q$ and hold this information in the factor base. It is worthwile to distinguish which root of $f(X, 1)$ contributes to the smoothness of $f(x, y)$. This is exploited in the NFS algorithm, because there is a bijection between the roots of f and the prime ideals of the corresponding number ring. However, when $q|a_n$ and $q|y$, we are in a special situation. In this case (4.1) tells us that $f(x, y) \equiv 0 \bmod q$ for all $x \in \mathbb{Z}$. For all q dividing a_n, we therefore add a special factor base element which is used in place of the (q, r_i) if and only if $q|y$.

Therefore, the factor base F connected to a homogeneous polynomial $f(X, Y)$ and a smoothness bound B is of the form

$$\begin{aligned} F &= \{(q, r) \in \mathbb{P} \times \mathbb{N}_0 \mid q \in \mathbb{P},\, q \leq B, r \leq q,\, f(r, 1) \equiv 0 \bmod q\} \\ &\quad \cup \{(q, q) \mid a_n \equiv 0 \bmod q\}. \end{aligned}$$

We choose r as the least non-negative integer satisfying $f(r, 1) \equiv 0 \bmod q$.

If q is bounded by the integer B, one can expect that $|F| \approx \pi(B)$. This is a consequence of the Tauberian theorem [36, Th. XV.5.4]. As this statement is valid asymptotically, we want to know, how close the factor base size actually is to the value of $\pi(B)$ when computing the average size over 1000 polynomials of a certain degree. The following table 4.1 shows that at least for polynomials of degree less than 6, which occur in the NFS, $\pi(B)$ is a good guess for the size of the factor base.

Table 4.1: Average Factor Base Size

degree	B	$\pi(B)$	avg. $\lvert F\rvert$
3	547	100	105
3	7927	1000	1004
3	104743	10000	10008
3	611957	50000	50001
4	547	100	103
4	7927	1000	1004
4	104743	10000	10005
4	611957	50000	50003
5	547	100	104
5	7927	1000	1003
5	104743	10000	10010
5	611957	50000	50010
6	547	100	103
6	7927	1000	1004
6	104743	10000	10004
6	611957	50000	50011

4.3 Configuration

We now give the concrete configuration of the sieving device to meet the requirements of the sieving methods mentioned at the beginning of this chapter. As there is enough experimental data available in literature, we can focus our attention on the choice of the homogeneous polynomials, instead of dealing with the factor base bounds, the large prime bounds, and the size of the sieve array.

4.3.1 The Number Field Sieve

Let $g_1, g_2 \in \mathbb{Z}[X]$ be the polynomials defining the Number Fields $\mathbb{Q}(\alpha_1)$ and $\mathbb{Q}(\alpha_2)$ respectively.

As usual, we take α_j to be root of

$$g_j(X) = \sum_{l=0}^{n_j} a_{jl} X^l, \qquad j = 1, 2, \tag{4.2}$$

and h_j the highest coefficient of g_j.

Proceeding with the general description in 4.1, we set the number of homogeneous polynomials $k = 2$. We recall the expression, which is used in the algorithm instead of the actual norm of elements $h_j \cdot (x + y\alpha_j)$, $x, y \in \mathbb{Z}$ from formula 3.9.

$$\begin{aligned} |N'(x + y\alpha_j)| &= y^{n_j} g_j(-\frac{x}{y}) \\ &= \sum_{k=0}^{n_j} (-1)^k a_{jk} x^k y^{n_j - k} \\ &=: f_j(x, y). \end{aligned}$$

Smoothness of $f_j(x, y)$ then is equivalent to $h_j \cdot (x + y\alpha_j)$ splitting into prime ideals of small norm.

When computing the class group of a number field K with the NFS, we are interested in getting relations among the prime ideals of only one number ring. The number ring is given by one polynomial of the form 4.2. So we simply let $g_1(X)$ be a polynomial defining K, compute $f_1(X, Y)$ and omit the polynomial $g_2(X)$. An explicit discussion of computing class groups with the NFS is intended to appear in [10].

4.3.2 The Gaussian Integer Method

Now we will explain, how to instantiate the Gaussian integer variant of the COS algorithm described in subsection 3.2.1 in terms of our sieving device. From the discussion at this place, we already know how to initialize the two bivariate polynomials of our sieving device. By reusing the notation, let $\mathbb{Q}(\sqrt{-r})$ be the chosen number field, and

$$(\frac{T}{V})^2 \equiv -r \bmod p,$$

where $T, V \in \mathbb{Z}$ with $|T|, |V| < \sqrt{p}$. We recall the result from (3.7), (3.8).

$$f_1(X, Y) = \begin{cases} X^2 + XY + \frac{r+1}{4} Y^2 & \text{if } r \equiv 3 \bmod 4 \\ X^2 + rY^2 & \text{otherwise} \end{cases}$$

$$f_2(X, Y) = \begin{cases} 2V\varphi(c + d\alpha) \equiv 2VX + (T + V)Y \bmod p, & \text{if } r \equiv 3 \bmod 4 \\ V\varphi(c + d\alpha) \equiv VX + TY \bmod p, & \text{otherwise} \end{cases}$$

From 3.2.1, it is clear that $k = 2$ and the polynomials f_1, f_2 are suitable.

Serving as an example, we recomputed the Sun challenge, which has been solved in 1991 by LaMacchia and Odlyzko [35] – their sieving time was 100 hours on a 25 mips computer. By using a rational factor base of size 150000 and an algebraic factor base size of 30000, we obtained 180000 relations within 12.6 hours on a Sparc 20 workstation, which is rated at about 80 mips. The computation was carried out this way in order to get a comparison to the computation of 1991. Of course, it can be substantially improved by employing the large prime variation.

4.3.3 The Quadratic Sieve

Assume we want to factor $N \in \mathbb{Z}$. The Quadratic sieve algorithm starts by computing many polynomials $Q(X) = AX^2 + BX + C \in \mathbb{Z}[X]$, with $B^2 - 4AC = kN$, $k \in \mathbb{Z}$ a (small) multiplier. We construct a homogenous quadratic polynomial

$$f(X, Y) = AX^2 + BXY + CY^2,$$

which we specialize with the condition $Y \equiv 1$. From the theoretical point of view, this specialization is not necessary, as for arbitrary y

$$Q_y(X) = AX^2 + (By) \cdot X + y^2 \cdot C,$$

is also a valid quadratic sieve polynomial for N, since

$$(By)^2 - 4A \cdot y^2 \cdot C = y^2(B^2 - 4AC) = k'N,$$

with $k' := y^2 k$.

Practical experience, however, shows that one should prefer to generate a new polynomial instead of using different values of y. The quadratic sieve is a method, in which the additional factor base information about roots of f mod q is only needed for sieving, but not for postprocessing the output. So we may simplify the output by merging the exponents of the two different factor base elements $(q, r), (q, r')$ for each prime q. Different output for different sieving algorithms is the reason why the output procedure is prepared to be changed at run–time (see subsection 4.5.1).

Compared to the specialized quadratic sieve implementation of LiDIA (Version 1.2) [6] for a 40–digit composite, we observe a running time of 43.4 seconds on a Pentium 100 computer using 400 polynomials for the sieving device and a running time of 29.1 seconds for LiDIA's quadratic sieve. In both cases the large prime variation has not been employed.

In table 4.2, we list the running times of the sieving device when factoring a 60–, 70– and 80–digit number with large prime variation; the parameters were not quite optimal, nevertheless this range of composites is already managable.

Table 4.2: Sieving Device (Quadratic Sieve)

# digits n	# factor base	LP bound	# relations	time sieve (sec)
60	8000	10^7	50000	19894
70	15000	$2 \cdot 10^7$	80000	76208
80	50000	$2 \cdot 10^7$	200000	729069

4.3.4 Reduction step for DL

For a detailed description of this step, we refer to section 3.1. Naturally, we stick to the notation of that paragraph.

Given the prime p, $t \in \mathbb{Z}$, $t \leq \sqrt{p}$, assume we want to compute the logarithm of t to some base, we may wish to express t by a product of small positive residues in $\mathbb{Z}/p\mathbb{Z}$. With $t' = \lceil \frac{p}{t} \rceil$, we find that the two homogeneous polynomials of degree one meet our requirements, by Lemma 3.2:

$$\begin{aligned} f_1(X,Y) &= X + t'Y \\ f_2(X,Y) &= tX + (tt' - p)Y. \end{aligned}$$

Information about the running times and the number of relations in the case of p having 65, 75 and 85 digits, is listed in table 3.1 of section 3.1 on page 28.

4.4 The Object Model

The implementation of the sieving device is carried out in the programming language C++, as it is specified in the reference manual of [73]. The reason for choosing C++ is the possibility to combine both the convenience of a powerful object model and the efficiency of the C language. In particular, this is the main reason for numerous C++ implementations in number theory. Additionally, it is possible to make it part of the big class library for computational number theory LiDIA [54, 6].

The heart of the model is the class `sieving_device`, which contains the polynomials, the factor bases, the large prime bounds, the sieve array, and the list of hits. To keep things simple, it is the only class, the user has to deal with. The polynomials are represented in a standard way as a dynamic vector of their coefficients. A *factor base* is a dynamic vector of factor base elements, whereas a *factor base element*

consists of the prime q, a root r of the corresponding polynomial mod q and an approximation of $\log q$ to some base. We call a pair (q, r) *factor base prime*, so that a factor base element is actually represented as factor base prime plus approximation of the logarithm. This is because we need factor base primes in order to represent large prime factors, too. Large prime bounds are simply single precision integers. The sieve array, which is repeatedly used, when sieving the different polynomials $f_j(x, y)$, holds a vector of bytes, bounds for x and the current y. When a hit (a probably smooth value) is found, its coordinates (x, y) are appended to a simple list of sieve hits. A *sieve hit* is identified by the pair (x, y) where the hit has occured. Further information is given by the decompositions of $f_j(x, y)$. Consequently, they are organized as simple lists of (prime) factors and large prime factors. A *factor* is of the form index/exponent, where the index refers to the number of the factor base element in factor base j. Large prime factors are simply represented by the type `factor_base_prime`.

This concludes the description of the object model.

4.5 The Classes

In C++ the abstract data types are called *classes*. We present the classes of the sieving device, with their class members and member functions. For each class, we present its definition and proceed with the description.

4.5.1 Sieving Device

As mentioned in section 4.4 above, this is the only class the user has to interact with in order to find smooth values of bivariate homogeneous polynomials simultaneously. For the sake of exposition, we use the top down approach when describing the classes.

```
typedef void (*sd_output_func)(ostream &, sieving_device &);

class sieving_device
      {
      bigint ring_char;                  // characteristic of ring

      short int n_poly;                  // # biv_hom_polynomials
```

```
biv_hom_polynomial <bigint> *f;   // f_i(x,y)
biv_hom_polynomial <double> *fd;  // f_i(x,y) type double

factor_base *fb;                  // factor bases
long        *lp_bound;            // large prime bounds

sieve_array s;                    // the sieve array

int *rels;                        // type of relations

int avail_mem;                    // main memory usage (bytes)

double log_base;                  // basis of logarithms
double log_log_base;              // log(log_base)

// list for collecting sieving hits
simple_list <sieve_hit> l;
// list iterator for l
simple_list_iterator <sieve_hit> li;

sd_output_func output_func;

char verbose_mode;                // verbose_mode mode yes/no
char timing_mode;                 // do timing yes/no

// -------------------------------------------------------
enum {MAX_POLY=2, AVAIL_MEM=1000000 };
// -------------------------------------------------------

int *pow3_tab;
int pow3(int k) const;
void pow3_init(int max);

void error(char *s);

public:

sieving_device();                    // constructor
~sieving_device();                   // destructor

// set user-defined output function
void set_output_func(const sd_output_func f);
```

```
void set_ring_characteristic(const bigint &n);
bigint get_ring_characteristic();

void set_timing_mode(int on_or_off);
void set_verbose_mode(int on_or_off);
int  set_avail_mem(int mem);

int  get_n_poly();
long get_lp_bound(int np);           // return lp_bound[np]

// return &l
simple_list <sieve_hit> *get_hit_pointer();

// return &f[np]
biv_hom_polynomial <bigint> *get_poly_pointer(int np);

// return &fb[np]
factor_base *get_fb_pointer(int np);

// insert polynomial f, compute fb for f, set lp_bound
void insert(const biv_hom_polynomial <bigint> &f,
           int num_elements, long lp_max,
           const prime_list &pl);

// insert polynomial f, factor base fb0, lp_bound
void insert(const biv_hom_polynomial <bigint> &g,
                    const factor_base &fb0, long lp_max);

// replace polynomial f[np] by g, semantic like insert()
void update(int np, const biv_hom_polynomial <bigint> &g,
            int num_elements, long lp_max,
            const prime_list &pl);

// find hits in the rectangle
// x_min<=x<x_max, y_min<=y<=y_max
void sieve(int x_min, int x_max, int y_min, int y_max);

// li points to the head of l
void reset_rel_iterator();

// get current (where li points to) relation,
```

```
      // increment li
      int get_relation(int &,  int &, int &, int *, int *);

      int count_hits();        // return number of elements of l
      void delete_hits();      // assign the empty list to l

      // print statistic of hits in l
      void output_statistic(ostream &);

      // print all elements of l
      void output_hits(ostream &out);

      // print all components of s
      friend ostream & operator<<(ostream &out, sieving_device &s);
      };

void sd_default_output_func(ostream &, sieving_device &);
void sd_qs_output_func(ostream &, sieving_device &);
```

Recalling the purpose of the class, namely to provide a user interface for finding smooth polynomials, the first definitions are straightforward. We dynamically allocate storage for a given number of polynomials `f[]`, factor bases `fb[]`, and large prime bounds `lp_bound[]`. Within the sieving process, one often needs to merely compute approximations of the values $f(x, y)$. Then it is faster to do this with 52–bit floating point approximations. So we make sure that we keep a copy of our multiprecision polynomials, where an approximation of the coefficients is stored as a polynomial of type `double`. Only one sieve array is needed for all polynomials. We shall concentrate on that point when examining the class `sieve_array`. The variable `avail_mem` will be set to restrict the use of main memory for the sieve array. This is especially reasonable to limit the sieve array length to fit in the cache memory. When a hit is found, it simply gets appended to the simple list `l`.

Although sieve algorithms have much in common, different sieve algorithms need different output formats. This may be seen by comparing the NFS with QS. With respect to the factor base of a polynomial f the NFS requires different roots of f modulo the same (rational) prime q to be treated separately. This is because they represent different prime ideals of norm q. In contrast, the output of the QS can be substantially simplified, since both roots mod q represent the prime q. In order to let the user choose, what should be the preferred output format for his (perhaps new invented) sieving method, we decided to enable the user to invoke his own output function at run time by the use of the member `set_output_func()`. An example for QS is our `sd_qs_output_func(ostream &, sieving_device &)`. Of course,

he can also use the default output function `sd_default_output_func(ostream &, sieving_device &)`, write the relations into a file, and modify the output later.

4.5.2 Factor Base

Having determined the bivariate polynomials, the values of which are searched for smoothness, each sieving algorithm starts with creating a factor base consisting of several small primes, which are used for sieving. As mentioned above, a prime q possibly occurs more than once in the factor base, because k roots of $f \bmod q$ refer to k factor base elements representing q. A factor base for a polynomial f of arbitrary degree with maximal element Q consists of roughly $\pi(Q)$ elements (see section 4.2).

We organize the factor base as dynamic array. The data structure is trivial though we give the listing for sake of completeness.

```
class factor_base
      {
      int      size;      // # elements
      fb_element *e;      // vector of FB elements

      void error(char *s);

      public:

      factor_base();
      ~factor_base();

      int get_size() const;
      int is_empty() const;  // (size==0) ?

      // compute roots of f modulo primes in pl
      void compute(const biv_hom_polynomial <bigint> &f,
                   int num_elements, const prime_list &pl);


      fb_element get_element(int index)  const; // return e[index]

      factor_base & operator=(const factor_base &fb0);
```

```
// Input / Output
friend ostream & operator<<(ostream &out, factor_base &fb);
friend istream & operator>>(istream &in, factor_base &fb);
};
```

When looking at the member functions, we see that the only non–trivial procedure here is the one solving the following problem.

Given a polynomial

$$f(X) = a_n X^n + a_{n-1} X^{n-1} + \ldots + a_1 X + a_0,$$

we must find all roots $\bmod q$ for all primes q below some bound Q within acceptable running time. This is done by a well known procedure, which is part of the Cantor–Zassenhaus polynomial factorization algorithm modulo primes. One attempts to split f by computing

$$\gcd(f(X), (X-a)^{(q-1)/2})$$

in $\mathbb{Z}/q\mathbb{Z}$ for sufficiently many (random) $a \in \mathbb{Z}$. The Cantor–Zassenhaus algorithm is described in many books about computational number theory; see for example [12].

4.5.3 Factor Base Element and Factor Base Prime

In a sieve algorithm, primes show up in two different situations. On the one hand, large primes happen to show up after dividing a function value by factor base primes. On the other hand, if q is a prime of the factor base, it is used to subtract the value $\log q$ from the sieve locations. Therefore it is reasonable to have a base class representing prime elements by a pair (q, r) and to derive a class `fb_element` from it by simply adding the information of the logarithm. Note that the class `prime_element` needs a virtual destructor since it serves as a base class for `fb_element`.

```
class prime_element
      {
      protected:

      long p,cp;   // (prime, start in sieve array)
```

```
      private:

      void error(char *s)
           { cout << "error in class prime_element: " << s << endl;
             exit(1); };

      public:

      prime_element(int prime=0, int cprime=0)
          { p=prime; cp=cprime; };

      virtual ~prime_element()
               { };

      inline void set_p(long p0)
                  { p=p0; };

      inline void set_cp(long cp0)
                  { cp=cp0; };

      inline long get_p() const
             { return p; };

      inline long get_cp() const
             { return cp; };

      friend int operator==(const prime_element &p,
                            const prime_element &q);


      // Input / Output
      friend ostream & operator<<(ostream &out,
                      const prime_element &p);
      friend istream & operator>>(istream &in, prime_element &p);
      };


class fb_element : public prime_element
      {
      char log_p;  // approx. to log(p)

      void error(char *s)
```

```
          { cout << "error in class fb_element: " << s << endl;
            exit(1); };

      public:

      inline void set_log_p(long lp)
                  { log_p=lp; };

      inline char get_log_p() const
             { return log_p; };

      // Input / Output
      friend ostream & operator<<(ostream &out, const fb_element &p);
      friend istream & operator>>(istream &in, fb_element &p);
      };
```

4.5.4 Sieve Array

Having determined polynomials and factor bases, the sieve process may start. We encapsulate the `char`–based sieve array and all corresponding maintenance and sieving functions in the class `sieve_array`, which we are going to present now.

```
class sieve_array
      {
      enum { NOHIT=0x7f,  HITB_TOLERANCE=3 };

      int  x_min, x_max;  // bounds first variable
      int  y;             // second variable
      char *s;            // sieve location
                          // s[0]          <-> x_min
                          // s[x_max-xmin] <-> x_max

      void error(char *s);

      void do_gcd_sieve(int p);

      public:
      enum modus {INIT, INIT_SURVIVE};
```

```
sieve_array();
~sieve_array();

// set sieving bounds
void set_bounds(int x_min, int x_max, int y);

inline int get_x_min();
inline int get_x_max();
inline int get_y();

// return next hit location in s[]
inline int next_hit(int start);

// remove pairs (x,y) with gcd(x,y)>1
void gcd_sieve();

// set s[] to init value
int init(const biv_hom_polynomial <double> &f,
          const bigint &leading_coeff,
          long lp_bound, modus m);

// called by init()
int init_recursive(const biv_hom_polynomial <double> &f,
                   const bigint &leading_coeff,
                   long lp_bound, int x0, int x1, modus m);

// subtract log(p) from s[] for each fb_element p
void log_sieve(const factor_base &fb);

// divide value of each hit location by p
void div_sieve(int np,
               sieving_device *sdp,
               hash_table <sieve_hit> &shh,
               simple_list <int> &shl);

// find smooth values after division sieve
void survive_div(int np,
                 const biv_hom_polynomial <bigint> *f,
                 long lp_bound,
                 hash_table <sieve_hit> &shh,
                 simple_list <int> &shl);
```

```
int count_hits(simple_list<int> &shl);

char & operator[](int index);

// Output
friend ostream &
operator<<(ostream &out, const sieve_array &sa);
};
```

To begin with the member variables, we store the actual x–bounds and y, whereas the `char`–pointer s points to a dynamically allocated sieve array. We will describe the sieving procedure very briefly – it is closely related to J. Zayer's [78]. First, we remove pairs (x, y) for which $\gcd(x, y) > 1$. This is done by sieving the array by the primes dividing y and setting each hit location to `NOHIT`, which is the maximal positive `char`–value. For each polynomial f, the initializing of the sieve array and the `log_sieve` step are done. Let C_f be a constant to be explained in a few moments. We initialize the array recursively with a crude approximation of $\log f(x, y) - C_f$ by the following method:

When preparing the interval $[x_0; x_2]$ for sieving, we compute

$$f(x_0, y), f(x_1, y), f(x_2, y),$$

where x_0, x_1, x_2 are the minimal, middle and maximal x–value respectively. In case of $\log f(x_i, y)$ and $\log f(x_{i+1}, y)$ differing by a value of at least `HITB_TOLERANCE` $(i = 0, 1)$, the interval gets bisected, and the procedure recursively affects the two intervals $[x_0; x_1]$ and $[x_1; x_2]$. Otherwise, the whole array gets initialized with value

$$(\log f(x_0, y) + \log f(x_2, y))/2 - C_f.$$

The `log_sieve` function subtracts $\log q$ for each factor base element (q, r) from the sieve locations $x \equiv yr \bmod q$. In previous implementations [78, 16] it turned out, to be wise not to sieve with prime powers and not to sieve with the smallest primes. We balance the missing subtractions of $\log q$ by increasing C_f appropriately, instead. For each large prime allowed, $\log$ `lp_bound` is added to C_f, too.

Note that the primes q' dividing the highest coefficient of f divide $f(x, y)$ when $\gcd(q', y) > 1$. So in this case we do not sieve with these primes; instead, $\log q'$ is added to C_f.

After the `log_sieve` step has been performed for each polynomial f, we look for locations of the sieve array which are < 0; these are *candidates* for smoothness.

The candidates are found by simultaneously testing the sign bit of 16 bytes. As the candidate locations are needed several times, they are stored in a simple list `shl`. The actual factorization is found by a division sieve then, where for each candidate x, the value $f(x, y)$ is computed. The `div_sieve` function behaves similar to the `log_sieve` function except for two changes:

- when a prime q hits a non–candidate location, nothing happens
- when a prime q hits a candidate location, $f(x, y)$ is replaced by $f(x, y)/q^k$, where k is the exact power of q dividing $f(x, y)$.

In order to efficiently find partial factorizations of candidate locations, the sieve hits are stored in a hash table. By walking through the simple list of hit locations, the `survive_div` function checks whether a rest <`lp_bound` remains after $f(x, y)$ has been divided by all factor base prime powers. If not, the sieve hit is removed from the `hash_table`. This concludes the description of the class `sieve_array`. The two functions that are most time critical are not part of the class but held globally, to keep the possibility of replacing them easily by assembly code. These are the functions

- `do_log_sieve(char *s,int stop,int p,char log_p)`
  which subtracts the amount of `log_p` from every location
  `s[stop-k*p]`, $0 \leq k \leq stop/p$
- `get_next_hit(char *s, int start)` which returns the location $i \geq$ `start`, with `s[i]<0`.

4.5.5 Sieve Hit

A sieve hit is a pair (x, y), where $f(x, y)$ is probably smooth over the factor base connected to f. The sieve hits are collected in a simple list in the sieving device. Usually, we want to know more than merely the coordinates. Therefore we decided to store the factorization with the coordinates. Here is the definition of the class `sieve_hit`.

```
class sieve_hit
      {
      sieving_device *sdp; // pointer to the sieving device,
                           // which uses sieve_hit
```

```
// smooth pair f_i(x,y)=value[i]*llp[i]*lfb[i]
int x,y;

// value[i] to be decomposed over factor base i
bigint *value;

simple_list <fb_factor> *lfb;     // see (x,y)
simple_list <prime_element> *llp; // see (x,y)

void error(char *s);              // error messages

public:

sieve_hit(sieving_device *s=0);
~sieve_hit();

void set_x(int x0);
void set_y(int y0);

void   set_value(int np, const bigint &v); // value[np]=v
bigint get_value(int np);

// insert element lf in list llp[np]
void insert_lp(int np, const prime_element &lf);

// return length of llp[np]
int count_lp(int np);

// (value[np]<=smoothness_bound)
int is_smooth(int np,long smoothness_bound);

// divide value[np] by p, insert (p,exponent) into lfb
void trialdiv(short int np,long p, int index);

// store values of x,y,lfb into ind[], e[], return length lfb
int store_into_array(int &n,  int &ret_x, int &ret_y,
                     int *ind, int *e);

sieve_hit & operator=(const sieve_hit &sh);

// Input / Output
```

```
friend ostream & operator<<(ostream &out, const sieve_hit &s);
friend istream & operator>>(istream &in, sieve_hit &s);

// compare (x,y) of s1 to s2
friend int
operator==(const sieve_hit &s1, const sieve_hit &s2);

// output a sieve_hit in qs format
friend void output_qs(ostream &out, const sieve_hit *s,
         const bigint &offset,        // offset=-b/2a mod n
         const bigint &a,             // h.c. of Q
         const bigint &n,             // characteristic of ring
         const factor_base *fbp       // factor base pointer
         );
};
```

At the beginning of the division sieve step, `value[i]` is equal to $f_i(x, y)$. When a factor q^k of $f_i(x, y)$ is found by `trialdiv`, $value_i$ is replaced by $value_i/q^k$ and the pair (j, k) is appended to the simple list `lfb[i]`, where j is the index of the pair (q, r) in the factor base. Recognition of possible large prime factors of the form (q, r) is achieved by the `is_smooth` function. These factors are appended then to the simple list `llp[i]` by calling `insert_lp`.

4.5.6 Prime List

Finally, the generation of primes remains to be explained. The fast and useful method for our purpose, namely producing the first primes below some bound, is the 2000–year old sieve of Eratosthenes[1]. Due to its fame, there is no need to explain it here. A substantial speed up is gained by sieving the linear functions $6X \pm 1$, because every prime except from 2 and 3 must be of this form since $6X + k$ is divisible by 2 or 3 unless $k \equiv 1, 5 \bmod 6$. This leads to saving up to 50 % of the running time, as can be seen in the following table, where we generated all primes up to n on a Sparc ELC workstation (21 Mips).

Furthermore, 75% of the main memory can be saved by not storing the absolute value of the primes. Instead, the differences between the primes divided by 2 are stored. This is sufficient since every operation involving primes from the prime number list reads the list sequentially. As a matter of fact, the halved differences fit into 1 byte for primes less than 10^9 [61].

[1] Eratosthenes von Kyrene 276–194 a. Chr.

Table 4.3: Running Time Erathostenes Sieve

n	Time Eratosthenes (hsec)	Time $6k \pm 1$ (hsec)	$\pi(n)$
1000000	102	53	78498
2000000	209	109	148933
5000000	538	313	348513
10000000	1106	980	664579
14000000	1570	1334	910077
15000000	1688	1431	970704

Conclusion

The computational results achieved by our implementation show that we are able to compute discrete logarithms in $(\mathbb{Z}/p\mathbb{Z})^*$, when p is a 85–digit prime (281 bits). Although the new two–quadratics–NFS version for DL has turned out to be highly efficient, a practical comparison has shown that up to prime fields of that size, the Gaussian integer method is the method of choice.

The data concerning McCurley's prime field with p having 129 digits show that under certain circumstances, even 416 bits are insecure. Although there is still a gap of 96 bits to the Digital Signature Standard, one is tempted to assume that with further refinements of theory and practical implementations, with further improvement of hardware and a concerted effort, a today's digital signature might be forgeable at some time in the future.

Apparently, sieving techniques are still the only way to break cryptographic schemes, the security of which is based on the difficulty of computing discrete logarithms in $(\mathbb{Z}/p\mathbb{Z})^*$ or factoring large integers. With our general sieving device, we provide a powerful tool, which covers all these methods and which is almost as efficient as specialized implementations. Furthermore, it is helpful to quickly experience the practicability of new methods for breaking such schemes.

List of Tables

Bibliography

[1] L. M. Adleman, *A subexponential algorithm for the discrete logarithm problem with applications to cryptography*, Proc. 20th IEEE Found. Comp. Sci. Symp., pp. 55–60, 1979

[2] L. M. Adleman, *The function field sieve*, Algorithmic number theory, Lecture Notes in Computer Science 877, pp. 108–121, Springer, 1994

[3] L. M. Adleman, J. DeMarrais, *A subexponential algorithm for discrete logarithms over all finite fields*, Math. Comp. 61, pp. 1–155, 1993

[4] E. Bach, *Explicit bounds for primality testing and related problems*, Math. Comp. 55, pp. 355–380, 1990

[5] D. Bernstein, A. K. Lenstra, *A general number field sieve implementation*, in [37], pp. 103–126, 1993

[6] I. Biehl, J. Buchmann, Th. Papanikolaou *LiDIA – A library for computational number theory*, Universität des Saarlandes, Tech. Report, 1995

[7] R. P. Brent, *An improved Monte Carlo factorization algorithm*, Nordisk Tidskrift för Informationsbehandling (BIT) 20, pp. 176–184, 1980

[8] J. Buchmann, *Number theoretic algorithms and cryptology*, Proceedings FCT '91, Lecture Notes in Computer Science 529, pp. 16–21, Springer, 1991

[9] J. Buchmann, J. Loho, J. Zayer, *An implementation of the general number field sieve*, Advances in Cryptology – Crypto '93, Lecture Notes in Computer Science 773, pp. 159–165, Springer, 1993

[10] J. Buchmann, St. Neis, D. Weber, *Computing class groups with the NFS*, in preparation

[11] J. P. Buhler, H. W. Lenstra, Jr., C. Pomerance, *Factoring integers with the number field sieve*, The development of the number field sieve, Lecture Notes in Mathematics 1554, pp. 50–94, Springer, 1993

[12] H. Cohen, *A course in computational algebraic number theory*, Graduate Texts in Mathematics 138, Springer, 1993

[13] D. Coppersmith, A. Odlyzko, R. Schroeppel, *Discrete logarithms in GF(p)*, Algorithmica 1, pp. 1–15, 1986

[14] D. Coppersmith, *Modifications to the number field sieve*, J. Cryptology, 1990

[15] J. Cowie, B. Dodson, M. Elkenbracht-Huizing, A. K. Lenstra, P. L. Montgomery, J. Zayer, *A world wide number field sieve factoring record: on to 512 bits*, ASIACRYPT 1996

[16] Th. Denny, *Faktorisieren mit dem Quadratischen Sieb*, Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1993

[17] Th. Denny, *Lösen großer dünnbesetzter Gleichungssysteme über endlichen Primkörpern*, PhD Thesis, Universität des Saarlandes, Saarbrücken, to appear

[18] Th. Denny, V. Müller, *On the reduction of composed relations from the number field sieve* , Proceedings ANTS II, Lecture Notes in Computer Science 1122, pp. 75–90, Springer, 1996

[19] W. Diffie, M. Hellman, *New directions in cryptography.* IEEE Trans. Information Theory 22, pp. 472–492, 1976

[20] B. Dodson, A. K. Lenstra, *NFS with four large primes: an explosive experiment*, Advances in Cryptology – Crypto '95, Lecture Notes in Computer Science 963, pp. 372–385, Springer, 1995

[21] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Trans. Information Theory 31, pp. 469–472, 1985

[22] T. ElGamal, *A subexponential-time algorithm for computing discrete logarithms over $GF(p^2)$*, IEEE Trans. Information Theory 31, pp. 473–481, 1985

[23] M. Elkenbracht–Huizing, *An implementation of the number field sieve*, Technical Report, Centrum for Wiskunde en Informatica, Amsterdam, 1993

[24] M. Elkenbracht–Huizing, *A multiple polynomial general number field sieve*, Proceedings ANTS II, Lecture Notes in Computer Science 1122, pp. 99–114, Springer, 1996

[25] R. A. Golliver, A. K. Lenstra, K. S. McCurley, *Lattice sieving and trial division*, Algorithmic Number Theory (ANTS), Lecture Notes in Computer Science 877, Springer, 1994

[26] G. H. Golub, C. F. Van Loan, *Matrix computations*, The Johns Hopkins University Press, 1993

[27] D. Gordon, *Discrete logarithms in GF(p) using the number field sieve*, SIAM J. Discrete Math. 6, pp. 124–138, 1993

[28] Th. Hungerford, *Algebra*, Graduate Texts in Mathematics 73, Springer, 1974

[29] K. Ireland, M. Rosen, *A classical introduction to modern number theory*, Graduate Texts in Mathematics 84, 2nd edition, Springer, 1990

[30] B. W. Kernighan, D. M. Ritchie, *The C programming language*, 2nd ed., Prentice Hall, 1988

[31] D. E. Knuth, L. Trabb Pardo, *Analysis of a simple factorization algorithm*, Theoretical Computer Science 3, pp. 321–348, 1976

[32] M. Kraitchik, *Théorie des nombres*, Vol. 1, Gauthier–Villars, 1922

[33] M. Kraitchik, *Recherches sur la théorie des nombres*, Gauthier–Villars, 1924

[34] M. LaMacchia, A. Odlyzko, *Solving large sparse linear systems over finite fields*, Advances in Cryptology – Crypto '90, Lecture Notes in Computer Science 537, pp. 109–133, Springer, 1991

[35] M. LaMacchia, A. Odlyzko, *Computation of discrete logarithms in prime fields*, Designs, Codes and Cryptography 1, pp. 46–62, 1991

[36] Serge Lang, *Algebraic number theory*, Graduate Texts in Mathematics 110, Springer, 1986

[37] A. K. Lenstra, H. W. Lenstra, Jr. (eds.), *The development of the number field sieve*, Lecture Notes in Mathematics 1554, Springer, 1993

[38] A. K. Lenstra, H. W. Lenstra, Jr., *Algorithms in number theory*, Technical Report 87-008, University of Chicago, 1987

[39] A. K. Lenstra, M.S. Manasse, *Factoring with two large primes*, Math. Comp. 63, pp. 77–82, 1994

[40] R. Lovorn, *Rigorous, subexponential algorithms for discrete logarithms over finite fields*, PhD Thesis, University of Georgia, 1992.

[41] R. Lovorn Bender, C. Pomerance *Rigorous discrete logarithm computations in finite fields via smooth polynomials*, preprint, 1995

[42] U. Maurer, *Towards the equivalence of breaking the Diffie–Hellman protocol and computing discrete logarithms*, Advances in Cryptology – Crypto '94, Lecture Notes in Computer Science 839, pp. 271–281, Springer, 1994

[43] U. Maurer, St. Wolf, *Diffie–Hellman Oracles*, Advances in Cryptology – Crypto '96, Lecture Notes in Computer Science, to appear

[44] K. McCurley, *The discrete logarithm problem*, Cryptology and Computational Number Theory, Proc. Symp. in Applied Mathematics 42, American Mathematical Society, pp. 49–74, 1990

[45] A. Menezes, T. Okamoto, S. A. Vanstone, *Reducing elliptic curve logarithms to logarithms in a finite field*, Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing, pp. 80–89, 1991

[46] A. Menezes, P. C. v. Oorschot, S. A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997

[47] Peter L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, Advances in Cryptology – Eurocrypt'95, Lecture Notes in Computer Science 921, pp. 106–120, Springer, 1995

[48] Peter L. Montgomery, *Number field sieve with two quadratic polynomials*, Centrum for Wiskunde en Informatica, Amsterdam, 1993

[49] National Bureau of Standards, *Digital signature standard*, FIPS Publication 186, 1994

[50] R. Needham, M. Schroeder, *Using encryption for authentication in large networks of computers*, Communications of the ACM 21, pp. 993–999, 1978

[51] A. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*, Advances in Cryptology – Eurocrypt '84 Lecture Notes in Computer Science 209, pp. 224–314, Springer, 1985

[52] A. Odlyzko, *Discrete logarithms and smooth polynomials*, Finite Fields: Theory, Applications, and Algorithms (Las Vegas, NV, 1993), Contemp. Math 168, Amer. Math. Soc., pp. 269–278, 1994

[53] Th. Papanikolaou, Jörg Zayer, *Algo.sty – ein TeX-Style für Algorithmen*, Universität des Saarlandes, 1994

[54] Th. Papanikolaou, *Software–Entwicklung in der Computer–Algebra am Beispiel einer objektorienten Bibliothek für algorithmische Zahlentheorie*, Universität des Saarlandes, PhD thesis, to appear

[55] S. Pohlig, M. Hellman, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*, IEEE Trans. on Information Theory 24, pp. 106–110, 1978

[56] J. M. Pollard, *A Monte Carlo method for factorization*, BIT 15, pp. 331–335, 1975

[57] J. M. Pollard, *Monte Carlo methods for index computation (mod p)*, Math. Comp. 32, pp. 918–924, 1978

[58] J. M. Pollard, *Factoring with cubic integers*, Manuscript, in [37], pp. 4–10, 1988

[59] J. M. Pollard, *The lattice sieve*, Manuscript, in [37], pp. 43–49, 1991

[60] C. Pomerance, *Fast rigorous factorization and discrete logarithms algorithms*, Discrete algorithms and complexity (D.S. Johnson, T. Nishizeki, A. Nozaki and H. Wilf, eds.), Academic Press, pp. 119–143, 1987

[61] Riesel, *Prime numbers and computer methods for factorization*, Birkhäuser, 1996

[62] R. Rivest, A. Shamir, L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Communications of the ACM 21, pp. 120–126, 1978

[63] D. Shanks, *Class Number, a Theory of Factorization and Genera*, Proc. Symposium Pure Mathematics Vol. 20, American Mathematical Society, Providence, R. I., pp. 415–440, 1970

[64] O. Schirokauer, *Discrete logarithms and local units*, Phil. Trans. R. Soc. Lond. A 345, pp. 409–423, 1993

[65] O. Schirokauer, D. Weber, Th. F. Denny, *Discrete logarithms: the effectiveness of the index calculus method*, Algorithmic Number Theory – ANTS II, Lecture Notes in Computer Science 1122, pp. 337–361, Springer, 1996

[66] O. Schirokauer, *Using number fields to compute logarithms in finite fields*, in preparation

[67] C. P. Schnorr, *Efficient signature generation by smart cards*, J. Cryptology 4, pp. 161–174, 1991

[68] E. Scholz, *Geschichte der Algebra*, Bibl. Inst., 1990

[69] Th. Setz, R. Roth, *LiPS: a system for distributed processing on workstations*, SFB 124 TP D5, Universität des Saarlandes, 1992

[70] Th. Setz *LiPS*, PhD Thesis, Universität des Saarlandes, 1996

[71] V. Shoup, *Searching for primitive roots in finite fields*, Proc. 22nd Annual ACM Symp. on Theory of Computing (STOC), pp. 546–554, 1990

[72] D. R. Stinson, *Cryptography in theory and practice*, CRC Press, 1995

[73] B. Stroustrup, *The C++ programming language*, Addison–Wesley, 2nd Edition, 1994

[74] B. Taylor, D. Goldberg, *Secure networking in the Sun environment*, Proc. USENIX Assoc. Summer Conference, Atlanta, pp. 28–37, 1986

[75] D. Weber, *An implementation of the number field sieve to compute discrete logarithms mod p*, Advances in Cryptology – Eurocrypt'95, Lecture Notes in Computer Science 921, pp. 95–105, Springer, 1995

[76] D. Weber, *Computing discrete logarithms with the number field sieve*, Algorithmic Number Theory – ANTS II, Lecture Notes in Computer Science 1122, pp. 390–403, Springer, 1996

[77] J. Zayer, *Die Theorie des Number Field Sieve*, Diplomarbeit, Saarbrücken, 1991

[78] J. Zayer, *Faktorisieren mit dem Number Field Sieve*, PhD Thesis, Saarbrücken, 1995