

Algorithms for Factoring Integers

Johannes Buchmann & Volker Müller

Technische Hochschule Darmstadt
Institut für theoretische Informatik

March 15, 2005

Contents

1	Introduction	5
1.1	Notation and Basic Concepts	5
1.2	The RSA cryptosystem	6
2	Basic Algorithms	11
2.1	Perfect Power Testing	11
2.2	Testing Pseudoprimality	12
2.2.1	The Fermat Pseudoprimality Test	12
2.2.2	Fast Exponentiation	13
2.2.3	The Miller Rabin Pseudoprimality Test	16
2.3	Trial Division	17
2.3.1	The Sieve of Erathostenes	18
2.4	The Pollard ρ -Method	19
3	Factoring Algorithms Using Smooth Group Orders	23
3.1	The Pollard $(p - 1)$ Factoring Method	23
3.2	Continuations	25
3.2.1	The Standard Continuation	25
3.2.2	The Improved Standard Continuation	26
3.2.3	The Prime Pairing Continuation	26
3.2.4	The Birthday Paradoxon Continuation	27
3.2.5	Montgomery's FFT - Continuation	27
3.3	The Principle Behind the $(p - 1)$ Method	28
3.4	The $(p + 1)$ Factoring Method	29
3.5	The Elliptic Curve Method (ECM)	33
3.5.1	Elliptic Curves over $\mathbf{Z}/N\mathbf{Z}$	34
3.5.2	Practical Experiences with ECM	37

4	Factoring Algorithms Using Smooth Elements	39
4.1	The Main Idea	39
4.2	Factoring Algorithms Using Continued Fractions	41
4.2.1	Shanks' SQUFOF Algorithm	44
4.2.2	The CFRAC Algorithm	45
5	The Quadratic Sieve Algorithm	47
5.1	Finding the Starting Points for Sieving	50
5.1.1	Testing whether N is a Square modulo a Prime	50
5.1.2	Extracting a Square Root modulo a Prime	51
5.2	Computing the Factor Basis	53
5.3	The Multi Polynomial Quadratic Sieve (MPQS)	54
5.4	The Self Initializing Variant of the MPQS	55
5.5	The Large Prime and Double Large Prime Variation	57
5.6	Practical Successes with the PPMPQS	58
6	The Number Field Sieve (NFS)	59
6.1	The Basic Idea of the NFS	59
6.2	Algebraic Background	61
6.3	Finding Relations	62
6.4	Extracting a Square Root in $\mathbf{Z}[\omega]$	64
6.5	Practical Improvements	66
6.5.1	The LLL Variant for Finding Polynomials	66
6.5.2	The Lattice Sieve	66
6.5.3	The Quadruple Large Prime Variation	67
6.6	Successes with the NFS	68
7	Solving Large Sparse Linear Systems	69
7.1	Structure of the Matrices	69
7.2	Structured Gaussian Elimination	70
7.3	The Coordinate Recurrence Method of Wiedemann	71
7.4	The Lanczos Algorithm	77

Chapter 1

Introduction

The goal of algorithmic number theory is to produce efficiently the objects whose mere existence is proved in the classical theorems of number theory. With growing importance of public key cryptography, the following three problems became especially important:

- Primality testing: is a given number N prime or composite ?
- Factoring: find all prime factors of a given composite number N .
- Computing discrete logarithms.

In this report, we will mainly concentrate on the second problem, but some remarks concerning primality testing will be made.

After a presentation of basic concepts, we will describe the RSA public key cryptosystem, which is perhaps the most well known cryptosystem based on a number theoretic problem.

1.1 Notation and Basic Concepts

We denote the set of rational integers by \mathbb{Z} . If $a, b \in \mathbb{Z}$, then there is precisely one pair $q, r \in \mathbb{Z}$ such that

$$a = q \cdot b + r \quad \text{with} \quad 0 \leq r < |b|.$$

We say that the division of a by b leaves **remainder** r . We write $q = \lfloor a/b \rfloor$ and $r = a \bmod b$ (note that \bmod starts with a capital M). If $r = 0$, then b is called **divisor** of a . We write $b|a$. A divisor b of a is called **proper divisor** of a , if $b \neq \pm a$ and $b \neq \pm 1$. A number $p \in \mathbb{N}_{>1}$, which has no proper divisors, is called **prime number**. There are infinitely many prime numbers.

The fundamental theorem of number theory states that each $N \in \mathbb{Z}$ can be decomposed into a power product of units and prime numbers p_i :

$$N = (-1)^{l_0} \cdot \prod_{i=1}^k p_i^{l_i} \tag{1.1}$$

with $l_0 \in \{0, 1\}$ and $l_i \in \mathbb{Z}_{\geq 1}$. This decomposition is uniquely determined up to the ordering of the prime numbers p_i . We say that we **factor** a number $N \in \mathbb{Z}_{>1}$ if we find

two numbers $b, c \in \mathbb{N}_{>1}$, such that $N = b \cdot c$. We **factor** a number $N \in \mathbb{Z}$ **completely**, if we find the decomposition given in (1.1), i.e. we find all prime numbers dividing N and the corresponding exponents.

1. Remark The determination of the exact exponent l_i for a prime number p_i which divides N can be done with the following trivial algorithm: as long as $N \bmod p_i$ is 0, set $N = N/p_i$ and increment l_i by one. Obviously, this algorithm works correctly, such that the main problem in the computation of (1.1) is the determination of the prime numbers p_i which divide N .

Let $a \in \mathbb{Z}$ and $N \in \mathbb{Z}_{\geq 1}$. Then

$$a + N \cdot \mathbb{Z} = \{a + N \cdot k \mid k \in \mathbb{Z}\}$$

is called the **residue class** of a modulo N , which we denote by $a \bmod N$. There are N residue classes modulo N . On the residue classes modulo N we can introduce an addition

$$(a \bmod N) + (b \bmod N) = (a + b) \bmod N$$

and a multiplication

$$(a \bmod N) \cdot (b \bmod N) = (a \cdot b) \bmod N.$$

With those operations, the residue classes modulo N form a commutative ring with unit element $1 \bmod N$, the **residue class ring** modulo N , denoted by $\mathbb{Z}/N\mathbb{Z}$. If $a \bmod N = b \bmod N$, we also write

$$a \equiv b \bmod N.$$

An element $b \in (a \bmod N)$ is called **representative** of the residue class $a \bmod N$. We can uniquely choose a representative of $a \bmod N$ in the set $\{0, \dots, N - 1\}$, namely the number $a \bmod N$. It is called the **least (non negative) residue** of $a \bmod N$. In the following, we do not explicitly specify the representative for a residue class, but usually least non negative representatives are used.

1.2 The RSA cryptosystem

The first so called public key cryptosystem was the RSA cryptosystem described in [RSA78]. We describe this cryptosystem with the help of a small example.

1.1. Example *Suppose that Bob wants to receive secret messages from Alice. He chooses a modulus N , say $N = 33$, and an encryption exponent e , say $e = 3$. Both those pieces of information are sent to Alice. Alice wishes to send him the secret message "I love you". Each letter can be encoded by a number $m \in \{1, \dots, 26\}$. The separating blank is $m = 27$. Thus Alice encrypts her message*

$$9 \ 27 \ 12 \ 15 \ 22 \ 5 \ 27 \ 25 \ 15 \ 21$$

by computing for each encoding number m the least residue of m^e modulo N . Then the encrypted message is

$$3 \ 15 \ 12 \ 9 \ 22 \ 26 \ 15 \ 16 \ 9 \ 21.$$

In order to recover the original information, Bob only needs to calculate for each encrypted number m the least residue of m^d modulo N with decryption exponent $d = 7$. Of course, Bob keeps the decryption exponent d secret and hopes that nobody can find it out.

In practice, the choice of these parameters would give no security since we could easily check all possibilities. Therefore the following questions arise: How does one choose the modulus N and the exponents e and d such that the system is secure? To answer this question, we must study the residue class ring $\mathbb{Z}/N\mathbb{Z}$ a little bit more. For $a, b \in \mathbb{Z}$, the largest $d \in \mathbb{Z}_{\geq 1}$, which divides both a and b , is called the **greatest common divisor** of a and b . We write

$$d = \gcd(a, b).$$

The gcd of two numbers can be computed with an algorithm going back to Euclid (see the exercises of this chapter). Each common divisor d' of a and b divides $\gcd(a, b)$. If $\gcd(a, b) = 1$, we call a and b **coprime**. The gcd of a and b has a representation

$$\gcd(a, b) = x \cdot a + y \cdot b \quad (1.2)$$

with integers $x, y \in \mathbb{Z}$. This representation of the gcd can also be computed with an extended version of Euclid's algorithm (see again the exercises). If $\gcd(a, N) = 1$, then $a \bmod N$ is called **primitive residue class** modulo N . Since by (1.2) there exists a representation

$$1 = x \cdot a + y \cdot N,$$

we see that $(1 \bmod N) = (x \bmod N) \cdot (a \bmod N)$ (or, in other words, $x \cdot a \equiv 1 \bmod N$). Therefore the primitive residue classes are **invertible** in $\mathbb{Z}/N\mathbb{Z}$. This means that the primitive residue classes modulo N form a finite abelian multiplicative group, the **primitive residue class group** $(\mathbb{Z}/N\mathbb{Z})^*$.

1.2. Example Let $N = 10$. Then there are 10 residue classes modulo 10. Among those there are 4 primitive ones, namely $(1 \bmod 10)$, $(3 \bmod 10)$, $(7 \bmod 10)$, $(9 \bmod 10)$. We have claimed that they are all invertible in $\mathbb{Z}/10\mathbb{Z}$. In fact, we have $3 \cdot 7 = 21 \equiv 1 \bmod 10$ and $9 \cdot 9 = 81 \equiv 1 \bmod 10$.

The number of primitive residues modulo N , i.e. the order of the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^*$, is denoted by $\varphi(N)$. The function

$$\begin{array}{ccc} \varphi: \mathbb{Z}_{\geq 1} & \longrightarrow & \mathbb{Z}_{\geq 1} \\ N & \longmapsto & |(\mathbb{Z}/N\mathbb{Z})^*| \end{array}$$

is called the **Euler φ -function**. As an application of Lagrange's theorem, we know that for every primitive class a modulo N

$$(a \bmod N)^{\varphi(N)} = (1 \bmod N) \quad \text{or, equivalently,} \quad a^{\varphi(N)} \equiv 1 \bmod N. \quad (1.3)$$

The congruence (1.3) shows us how to choose the encryption and decryption exponents e and d . The encryption exponent e is chosen "at random" such that $\gcd(e, \varphi(N)) = 1$. Then, by (1.2), we can compute a number d , such that

$$1 = e \cdot d + y \cdot \varphi(N) \quad (1.4)$$

for some $y \in \mathbb{Z}$. If m is the encoded letter, then $m^e \bmod N$ is the residue class of the encrypted letter. We have

$$\begin{aligned} (m^e)^d &= m^{e \cdot d} = m^{1-y \cdot \varphi(N)} \\ &= m \cdot m^{-\varphi(N) \cdot y} = m \cdot (m^{\varphi(N)})^{-y} \\ &\equiv m \pmod{N}. \end{aligned}$$

This argument only covers the case where $\gcd(m, N) = 1$. In general,

$$m^{e \cdot d} \equiv m \pmod{N} \tag{1.5}$$

is not even true. But if $N = p \cdot q$ is the product of two distinct prime numbers p and q , then (1.5) remains valid.

In order to apply this encryption scheme in practice, Bob must be able to determine $\varphi(N)$ and then solve the equation (1.4), which can be done by computing the extended gcd of e and $\varphi(N)$. If

$$N = \prod_{i=1}^k p_i^{l_i} \tag{1.6}$$

is the decomposition of N into powers of distinct primes, then

$$\varphi(N) = N \cdot \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right).$$

Hence, $\varphi(n)$ can be computed if the factorization (1.6) of N is known. No other efficient way of computing $\varphi(N)$ is known so far.

In order to use the RSA-scheme, Bob will use a modulus $N = p \cdot q$ with two large prime numbers (each 100 digits). Then he gets $\varphi(N) = (p-1) \cdot (q-1)$. Choosing a large random encryption exponent e , he can easily determine the decryption exponent d . He publishes the **public keys** N and e and keeps his **secret key** d private.

Alice only needs to encode her message using numbers $m \in \{0, \dots, N-1\}$. Then she has to determine the least positive residue in $m^e \bmod N$. We will describe later how she can solve this problem efficiently. As shown above, Bob can decrypt the encrypted message by computing the d -th power modulo N .

In order to break the scheme, one has to recover $m \bmod N$ from $m^e \bmod N$, N and e . If an intruder can compute the factorization of N , then he can compute $\varphi(N)$ and therefore the private key d . Therefore the factorization problem for integers is of importance for the security of RSA. The RSA modulus N must be chosen in a way that it is infeasible to factor N . Unfortunately, it is not known whether breaking RSA is equivalent to factoring the modulus N , but there exist other public key cryptosystems similar to RSA which have this property.

In this report, we will present an overview on currently used algorithms for factoring integers and give some practical information. We will try to be precise enough that a

reader can implement these algorithms and test their practical behavior.

1.1. Exercises

1. Euclid's algorithm for computing the gcd of two integers a_1 and a_2 is based on successively carrying out divisions with remainder

$$a_i = q_i \cdot a_{i+1} + a_{i+2} \quad \text{with } 0 \leq a_{i+2} < |a_{i+1}|.$$

Show that the sequence of divisions with remainder terminates and yields the gcd of the two numbers a_1 and a_2 . Formulate Euclid's idea in form of an algorithm.

2. There exist several variants for computing gcds. One practical observation is the fact that division by 2 can be done with a shift and is therefore much cheaper than a general division. This fact leads to two binary gcd algorithms. Assume that we want to compute the gcd of two odd integers a_1 and a_2 . The so called right-to-left binary gcd algorithm is based on the equation

$$\gcd(a_1, a_2) = \gcd\left(\frac{a_1 - a_2}{2}, a_2\right),$$

the left-to-right binary algorithm uses

$$\gcd(a_1, a_2) = \gcd(a_1 + 2^k \cdot a_2, a_2)$$

for all $k \geq 0$. Prove these two equations and show how they can be used to get a gcd algorithm for arbitrary integers.

3. The Extended Euclid Gcd Algorithm uses exactly the same iteration as the "normal" algorithm, but additionally it guarantees the following invariant: we always know integers s_1, s_2 and t_1, t_2 such that

$$a_i = s_1 \cdot a_1 + t_1 \cdot a_2 \quad \text{and} \quad a_{i+1} = s_2 \cdot a_1 + t_2 \cdot a_2.$$

Show how the algorithm can find such a representation for a_{i+2} if it knows representations for a_i and a_{i+1} . How do we get a representation of the gcd of a_1 and a_2 in this way?

4. In Part 2 of this exercise, we described the basic ideas of two binary gcd algorithms. There also exist variants of these binary algorithms which compute the representation of the gcd. These variants hold exactly the same invariant as the Extended Euclid Gcd Algorithm. Show how such representations can be computed during the algorithm.

Chapter 2

Basic Algorithms

In this report, we always assume that we want to factor a positive integer N bigger than one. Obviously, the first test before starting a “serious” factoring algorithm should be a check whether N is even (then we have the factor 2), is a prime number or a perfect power of some number. In the next sections we will explain algorithms for solving the latter two problems. Then we describe basic factoring algorithms as Trial Division and the Pollard ρ -algorithm.

2.1 Perfect Power Testing

Assume that we want to test for a positive odd number $N > 1$ whether N is a perfect power, i.e. whether $N = m^k$ for some $k \geq 2$ and $m \in \mathbb{N}$. Obviously, we have the trivial bound $k \leq \lfloor \log_2(N) \rfloor$. Therefore, we can do this perfect power test by checking for all $2 \leq k \leq \lfloor \log_2(N) \rfloor$ whether the polynomial $X^k - N$ has an integer root. Additionally, the following observation limits the number of possible exponents even more:

2.1. Fact *If $k = k_1 \cdot k_2$ and $N = m^k$, then there exists an integer l such that $N = l^{k_1}$.*

Therefore we can restrict ourself to check exponents, which are prime numbers. These candidates can be precomputed and stored. An algorithm for finding all prime numbers below some bound K will be described in Section 2.3.

Suppose that we want to check for a prime exponent k whether $X^k - N$ has an integer solution. There exists an algorithm of Newton to solve this problem. It makes use of the result that the sequence defined by $x_0 = N$ and

$$x_{i+1} = \frac{(k-1) \cdot x_i}{k} + \frac{N}{k \cdot x_i^{k-1}}$$

for $i \geq 0$ converges against $N^{1/k}$. Note that all x_i are rational numbers if N is an integer. We compute the sequence until the integer part of two subsequent sequence elements is the same. In this case, we have found the integer part of $N^{1/k}$ or $N^{1/k} \pm 1$. These three

cases can be checked by an exponentiation.

2.1. Exercises

1. *Develop an algorithm based on the Newton sequence which only uses integers (Idea: Substitute the rational numbers in the sequence by their integer part and find a suitable stopping condition).*
2. *Describe a “divide-and-conquer” algorithm for testing whether polynomial $X^k - N$ has an integer root.*

2.2 Testing Pseudoprimality

Let N be an odd integer which we want to check for primality. Since proving the primality of a number is a complicated problem (for an overview on this problem, see [BuMü92]), we use a so called **Pseudo Primality Test**. If such a test outputs “number is composite”, then we have a proof that the input really is composite (but without knowing a factor of the input). On the other hand, we get no proof for the primality of the input number, but we know with “high probability” that a number described as “prime” indeed is a prime number, i.e. the probability that a number N is composite but the test outputs “input is prime” is very small. Nevertheless such tests are commonly used since they are very fast. We present two such tests, the Fermat Test and the Miller-Rabin Test.

2.2.1 The Fermat Pseudoprimality Test

The Fermat Test is based on the following observation: If N is a prime number, then we know that $\varphi(N) = N - 1$, and from (1.3) we can conclude that for every $a \in \mathbb{Z}$ with $\gcd(a, N) = 1$ we have

$$a^{N-1} \equiv 1 \pmod{N}. \quad (2.1)$$

Thus we have proven the following proposition.

2.1. Proposition *Let $a \in \mathbb{Z}$. If $\gcd(a, N) = 1$, but $a^{N-1} \not\equiv 1 \pmod{N}$, then N is composite.*

Proposition 2.1 provides a sufficient criterion for testing a number N for compositeness. There are however composite numbers N , which satisfy (2.1) for a certain base a (so called **Pseudo primes** for the base a) or even for all bases a with $\gcd(a, N) = 1$ (**Carmichael Numbers**). We note that Carmichael numbers are rather scarce (2163 below $25 \cdot 10^9$) and that below $25 \cdot 10^9$ there are only 21853 pseudo primes for the base 2. The chance that a test constructed on the basis of Proposition 2.1 fails is therefore very small.

If we have checked (2.1) for “a few” bases and have found no contradiction, then we suspect that N is prime and we apply a primality test to prove its primality. Otherwise we know that N is composite and we attempt to factor it.

2. Algorithm (Fermat Pseudoprimality Test)

INPUT: $N \in \mathbb{N}_{>1}$.

OUTPUT: “ N is composite” or “ N is a pseudo prime”.

```

(1) for (number of desired tests) do
(2)   Choose  $1 < a < N$  at random.
(3)   if  $\text{gcd}(a, N) > 1$  then
(4)     return (“ $N$  is composite”)
(5)   fi
(6)   Compute  $a^{N-1} \text{ Mod } N$ .
(7)   if  $(a^{N-1} \neq 1 \text{ Mod } N)$  then
(8)     return (“ $N$  is composite”)
(9)   fi
(10) od
(11) return (“ $N$  is a pseudo prime”)

```

In Step (6) of this algorithm, we have to compute $a^{N-1} \text{ Mod } N$ for a “possibly large” number N . Computing this value by successive multiplications is therefore not possible. In the next section we describe two methods to perform this exponentiation.

2.2.2 Fast Exponentiation

Suppose that we want to determine $a^d \text{ mod } N$ for some number $d \neq 0$. For negative exponents d , we use the transformation $a^d = (a^{-1})^{|d|}$, such that we can assume that d is positive. We compute the binary decomposition of the exponent

$$d = \sum_{i=0}^s \beta_i \cdot 2^i, \quad \beta_i \in \{0, 1\}, \quad \beta_s = 1.$$

Then we have

$$a^d = a^{\sum_{i=0}^s \beta_i \cdot 2^i} = \prod_{i=0}^s a^{\beta_i \cdot 2^i} = \prod_{i=0, \beta_i=1}^s a^{2^i}. \quad (2.2)$$

If we want to compute $a^d \text{ Mod } N$, then all intermediate computations can be done modulo N . Note further that the number of factors in this representation of a^d is at most $s + 1 = \lfloor \log_2 d \rfloor + 1$. In the so called right-to-left binary exponentiation algorithm, the decomposition of the exponent d is done in ascending order (from the low order bit β_0 to the high order bit β_s) while computing the successive powers $a^{2^i} \text{ Mod } N$. We illustrate the idea in an example.

2.1. Example We want to check whether $N = 21$ is a prime number. We use the Fermat Test with base $a = 2$, and we have to test whether $2^{20} = 1 \text{ Mod } 21$. We decompose the exponent as $20 = 2^4 + 2^2$. Therefore

$$2^{20} = 2^{16} \cdot 2^4 \text{ Mod } 21 .$$

We successively compute $2^1 = 2 \text{ Mod } 21$, $2^2 = (2^1)^2 = 4 \text{ Mod } 21$, $2^4 = (2^2)^2 = 16 \text{ Mod } 21$, $2^8 = 4 \text{ Mod } 21$ and $2^{16} = 16 \text{ Mod } 21$, and we obtain the final result

$$2^{20} = 16 \cdot 16 = 4 \text{ Mod } 21 .$$

Thus we have proven that 21 is a composite number.

This idea leads to the following algorithm.

3. Algorithm (Right-to-Left Binary Exponentiation)

INPUT: $a, d, N \in \mathbb{N}$.
OUTPUT: $b = a^d \text{ Mod } N$.

- (1) Initialize $b = 1, c = a \text{ Mod } N$.
- (2) **while** ($d \geq 1$) **do**
- (3) **if** (d is odd) **then**
- (4) Set $b = b \cdot c \text{ Mod } N$ and $d = d - 1$.
- (5) **fi**
- (6) Set $d = d/2$ and $c = c^2 \text{ Mod } N$.
- (7) **od**
- (8) **return** (b)

A second look at (2.2) shows the idea for another fast exponentiation algorithm which scans the bits of the exponent d from “the left to the right” (i.e. from the high order bit β_s to the low order bit β_0). We have the equation

$$a^d = \left(\dots \left(\left(a^{\beta_s} \right)^2 \cdot a^{\beta_{s-1}} \right)^2 \dots \cdot a^{\beta_1} \right)^2 \cdot a^{\beta_0} .$$

2.2. Example We solve the same problem as in Exercise 2.1. Again, we observe $20 = 16 + 4$ and

$$2^{20} = \left(\left(\left(\left(2^1 \right)^2 \right)^2 \cdot 2 \right)^2 \right)^2 \text{ Mod } 21 .$$

We start the evaluation of this expression in the middle and get $(2^1)^2 = 4 \text{ Mod } 21$, $((2^1)^2)^2 = 16 \text{ Mod } 21$ and $((2^1)^2)^2 \cdot 2 = 11 \text{ Mod } 21$. Squaring these values twice, we get the result $2^{20} = 4 \text{ Mod } 21$.

This idea leads to a second fast exponentiation algorithm, the so called left-to-right binary exponentiation algorithm.

4. Algorithm (Left-to-Right Binary Exponentiation)

INPUT: $a, d, N \in \mathbb{N}$.
 OUTPUT: $b = a^d \text{ Mod } N$.

```
(1) Initialize  $b = 1$ .
(2) Compute the binary decomposition  $d = \sum_{i=0}^s \beta_i 2^i$ ,  $\beta_s = 1$ .
(3) for ( $i = s$  downto 0) do
(4)   Set  $b = b^2 \text{ Mod } N$ .
(5)   if ( $\beta_i = 1$ ) then
(6)     Set  $b = b \cdot a \text{ Mod } N$ .
(7)   fi
(8) od
(9) return ( $b$ )
```

Concerning the running time of both these algorithms, we have the following proposition:

2.2. Proposition *Using Algorithm 3 or Algorithm 4, we can compute $a^d \text{ Mod } N$ in $O(\log d)$ elementary operations on numbers of size $O(\log N)$.*

It should be mentioned that there exist several other variants of fast exponentiation algorithms which support special situations. In cryptography, it is sometimes possible to precompute some information, which can lead to a significant practical running time improvement.

2.2. Exercises

1. *The algorithms for fast exponentiation are not restricted to primitive residue classes modulo N , they can be generalized to arbitrary abelian groups. Are there situations where scanning the bits of an exponent from the high order bit to the low order bit is superior to scanning the bits in the opposite direction ?*
2. *Develop a block version of Algorithm 3, i.e. a version which uses a m -adic representation of the exponent d .*
3. *In practice, the number of ones in the binary decomposition of the exponent d greatly affects the running time of both these algorithms. If inversion is as fast as multiplication, we can reduce the number of 1's by allowing -1's in the decomposition of d . Describe a left-to-right or a right-to-left binary algorithm which uses this idea (Hint: Try to substitute a "block of 1's" by a corresponding block with 1's, -1's and 0's (note for example that $15 = 16 - 1$)).*

2.2.3 The Miller Rabin Pseudoprimality Test

The Miller Rabin Pseudoprimality Test is an extension of the Fermat Test. In addition to the test whether $a^{N-1} \equiv 1 \pmod N$, it uses the fact that for a prime number N there exist exactly two square roots of 1 modulo N . If we find an integer x with $x^2 \equiv 1 \pmod N$ and $x \not\equiv \pm 1 \pmod N$, then N can not be prime. Additionally, such an integer x directly leads to a factor of N : since $x^2 \equiv 1^2 \pmod N$ and $x \not\equiv \pm 1 \pmod N$, we know that N divides $x^2 - 1^2 = (x+1) \cdot (x-1)$, but N is no divisor of $x-1$ and $x+1$. Therefore $\gcd(x-1, N)$ is a proper divisor of N .

2.3. Example *Assume that we use the Fermat Test on $N = 15$ with base $a = 4$. We compute $4^{14} = 1 \pmod{15}$, and the Fermat Test does not declare N to be composite. But the exponent 14 is even, and therefore we have the equation $(4^7)^2 = 4^{14} = 1 \pmod{15}$. We compute $4^7 = 4 \pmod{15}$. Obviously, $4^7 \not\equiv \pm 1 \pmod{15}$ and therefore $\gcd(4^7 - 1, 15) = 3$ is a proper divisor of 15.*

This example clarifies the idea of the Miller Rabin Pseudoprimality Test. There are two possibilities, where we can prove the compositeness of an integer. First the Fermat test might fail, i.e. $a^{N-1} \not\equiv 1 \pmod N$. If it does not fail, then $a^{N-1} \equiv 1 \pmod N$, and – since $N-1$ is even – there is a chance that $a^{(N-1)/2} \not\equiv \pm 1 \pmod N$, which would prove N to be composite (and directly give us a factor of N). If $a^{(N-1)/2} \equiv 1 \pmod N$, then we recursively do the same test again. In the implementation of this algorithm, one should test these cases “bottom-up” as following:

Let $N-1 = 2^s \cdot m$ with an odd integer m , and assume that $a \in \mathbb{Z}$ is a random integer coprime to N . If we find $0 \leq i < s$ with $a^{2^i m} \not\equiv \pm 1 \pmod N$, but $a^{2^{i+1} m} \equiv 1 \pmod N$, then N must be composite and – as shown above – we find a proper divisor of N . These tests

can be done on the fly in the fast exponentiation.

5. Algorithm (Miller Rabin Pseudoprimality Test)

INPUT: $N \in \mathbb{N}_{>1}$.

OUTPUT: “Factor of N found”, “ N is composite” or “ N is a pseudo prime”.

```

(1) Compute  $s$  such that  $N - 1 = 2^s \cdot m$  with  $m$  odd.
(2) for (number of desired tests) do
(3)   Choose  $1 < a < N$  at random.
(4)   if  $\text{gcd}(a, N) > 1$  then
(5)     return (“Factor  $\text{gcd}(a, N)$  found”)
(6)   fi
(7)   Compute  $b = a^m \text{ Mod } N$ .
(8)   for ( $i = 1$  to  $s$ ) do
(9)     if  $(b^2 = 1 \text{ Mod } N \text{ and } b \neq \pm 1 \text{ Mod } N)$  then
(10)      return (“Factor  $\text{gcd}(b - 1, N)$  found”)
(11)     fi
(12)     Set  $b = b^2 \text{ Mod } N$ .
(13)   od
(14)   if  $(b \neq 1 \text{ Mod } N)$  then
(15)     return (“ $N$  is composite”)
(16)   fi
(17) od
(18) return (“ $N$  is a pseudo prime”)

```

2.1. Exercise *Is the number $2^{17} + 1$ a prime number (Hint: first think about it!!) ?*

2.3 Trial Division

Assume that we know that the odd integer N is composite and no perfect power, and we want to find a proper factor of N . In practice, the first factoring method which we should use for N is the so called **Trial Division**. The basis of this method for factoring N is the following trivial observation.

2.2. Fact *N is prime if it has no prime divisor $p \leq \sqrt{N}$.*

Since we know that N is not prime, there must exist a prime number smaller than \sqrt{N} which divides N . If we check all primes numbers smaller than this bound, we will find a factor of N . We therefore precompute a table of all prime numbers $p \leq C$ and use this

table to completely factor integers $N \leq C^2$. Since the number of primes $p \leq C$ is roughly $C/\log C$, this is quite a time and memory expensive method. In practice, it is almost never possible to completely factor a large number of about 100 decimal digits only with trial division. Nevertheless trial division is very fast for finding small factors (up to about 10^6) in N . Instead of reading the used small prime numbers from a table, we can compute these primes with the sieve of Erathostenes.

2.3.1 The Sieve of Erathostenes

We want to find all prime numbers smaller than some given bound C . With Fact 2.2 we can conclude that all composite numbers smaller than C have a prime divisor smaller than \sqrt{C} . The sieve of Erathostenes finds all these composite numbers with a sieving strategy (sieving is a very important trick in factoring algorithms as we will see).

6. Algorithm (Sieve of Erathostenes)

INPUT: $C \in \mathbb{N}_{>1}$.

OUTPUT: List of prime numbers smaller C .

- (1) Initialize an array $a[i] = 0$ for $1 \leq i < C$.
- (2) Set $L = \emptyset$, $p = 2$.
- (3) **while** (1) **do**
- (4) Set $L = L \cup \{p\}$.
- (5) Set $a[i \cdot p] = 1$ for all $1 \leq i \leq \lfloor C/p \rfloor$.
- (6) **if** (there exists index $p < j \leq \lfloor \sqrt{C} \rfloor$ with $a[j] = 0$) **then**
- (7) Set p to the smallest such index $j > p$.
- (8) **else**
- (9) Set $L = L \cup \{q\}$ for all $\sqrt{C} \leq q < C$ with $a[q] = 0$.
- (10) **return** (L)
- (11) **fi**
- (12) **od**

In practice, one can gain an improvement in memory and running time by packing the array a into a bit field. It is obvious that we do not have to store all even numbers greater 2, these can not be prime. Then we can use precomputed masks to efficiently mark all multiples of a found prime number. This improvement is substantial for finding all multiples of “very small” prime numbers (up to about the word size, commonly 32). A detailed description of this idea can be found in [Mü96]. With this implementation, the computation of all prime numbers smaller than 10^6 takes about one second on a sparc ELC computer.

2.2. Exercise Find a set of masks for marking all multiples of the prime number 5 on a 32-bit word architecture (assume that i -th bit in the bit field denotes the number $2i + 1$).

2.4 The Pollard ρ -Method

In this section, we explain a factoring algorithm due to Pollard ([Po75]) which uses a principle which can be found in many other factoring algorithms, too. Assume that p is a prime factor of the odd composite number N , which we want to factor. The idea of the Pollard ρ -method is based on the so called **birthday paradoxon**:

There are a large number of people in a room. You sample their birthdays, hoping to find two with the same birthday. You stop as soon as you find a duplicate. How many people do you have to choose such that you find a duplicate with probability greater $\frac{1}{2}$? We can answer this question by solving the inequality

$$\left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdot \left(1 - \frac{3}{365}\right) \cdots \left(1 - \frac{n-1}{365}\right) < \frac{1}{2}$$

for n , which leads to the result $n > 23$ (most people are surprised by this small number).

Let us translate this birthday question into a factoring question. Suppose that we choose a sequence x_i of random integers modulo N . In [Ri85], it is shown that after

$$\sqrt{2 \ln 2} \cdot \sqrt{p} \approx 1.18 \cdot \sqrt{p} \tag{2.3}$$

random choices we can expect with probability greater $\frac{1}{2}$ to find two elements x_i, x_j , which are congruent modulo p . In this case, $\gcd(x_i - x_j, N)$ is divided by p and hopefully this gcd will lead to a proper factor of N .

In Pollard's ρ -method, we generate the "random integers" x_i with the help of a pseudo random generator. Experience shows that this substitution does not affect the success of the method in practice. Often the pseudo random sequence is generated by the following deterministic description: choose an arbitrary $x_0 \in \{1, \dots, N-1\}$ and iterate $x_{i+1} = x_i^2 + a \text{ Mod } N$ for $i \geq 0$, where a is some randomly chosen integer unequal to 0 or -2 ([GoSa83] suggest that this choice is "optimal" in practice, concerning computation time and randomness).

Since pseudo random number generators are deterministic methods, finding two integers x_i, x_j which are congruent modulo p means to find a cycle in the sequence of all $x_i \text{ mod } p$. The trivial algorithm for finding such a cycle would be to store all integers $x_k \text{ Mod } N$ and compute $\gcd(x_r - x_s, N)$ for all indices $r \neq s$. Obviously, this strategy is very slow (we have lots of gcd computations) and has a huge memory need. Fortunately, we can restrict ourselves to the comparison of elements $x_r \text{ Mod } N$ and $x_{2r} \text{ Mod } N$, as was first shown by Floyd.

2.1. Lemma *Let $(x_k \text{ Mod } N)_{k \geq 0}$ be a pseudo random sequence in $\mathbb{Z}/N\mathbb{Z}$. If there exist indices $i < j$ such that $x_i \equiv x_j \text{ mod } p$, then there also exists an index $i \leq r < j$ such that $x_r \equiv x_{2r} \text{ mod } p$.*

Proof: Since the sequence $(x_k \text{ Mod } N)_{k \geq 0}$ of pseudo random elements is determined in a deterministic way, we know that $x_{i+s} \equiv x_{i+s+k \cdot (j-i)} \text{ mod } p$ for all $0 \leq s < j-i$ and $k \geq 1$. Therefore we can assume that $i \leq r < j$. If we set $r = i + s$, we have to find an integer s such that the equation $2 \cdot r \equiv r \text{ mod } (j-i)$ holds. Obviously, an integer $0 \leq s < j-i$ with $s \equiv -i \text{ mod } (j-i)$ satisfies all the desired conditions.

Note that the disadvantage of this very memory efficient extension of Floyd is the fact that the number of sequence elements which we have to test before we find a factor increases (Lemma 2.1 shows that this increase can at most be equal to the length of the cycle). Nevertheless this trick is used in practice. In the following Algorithm 7, the variable x is used to hold the i -th element, y holds the $2i$ -th element of the pseudo random sequence generated by the polynomial $g(X)$. Therefore in each iteration, y has to be changed to $g(g(y)) \text{ Mod } N$.

7. Algorithm (Pollard's ρ -Method, Floyd's variant)

We use the pseudo random sequence generated by the polynomial $g(X) = X^2 + a \text{ Mod } N$.

INPUT: composite $N \in \mathbb{N}_{>1}$, starting value x_0 , $a \in \mathbb{Z}$.

OUTPUT: Factor of N .

- (1) Set $x = x_0 \text{ Mod } N$ and $y = x_0 \text{ Mod } N$.
- (2) **repeat**
- (3) Set $x = x^2 + a \text{ Mod } N$.
- (4) Set $y = y^4 + 2y^2 a + a^2 + a \text{ Mod } N$.
- (5) **until** $(\gcd(x - y, N) > 1)$
- (6) **return** $(\gcd(x - y, N))$

2.1. Remark

1. It is possible to find the trivial divisor N . In this case, we have even found a cycle modulo N . We can restart the computation with another generating polynomial $g(X)$.
2. If all prime divisors of N are large, then Algorithm 7 has an enormous expected running time. Therefore it is a good solution to insert a break condition into the loop in Algorithm 7 and to stop after a predefined number of loop iterations.
3. Algorithm 7 performs a lot of gcd computations. Since these operations are expensive, we should use a blocking technique. If x_i, y_i are the values of the variables x, y , in iteration i , respectively, we "block" r gcd computations starting with iteration s by setting

$$h = \prod_{k=0}^{r-1} (x_{s+k} - y_{s+k}) \text{ Mod } N$$

and computing $\gcd(h, N)$. In this way, we have reduced the number of gcd computations by $r - 1$. If $\gcd(h, N) = N$, we should go back and check each of the elements $x_{s+k} - y_{s+k}$ separately. Note that the optimal size of the blocking factor is a difficult question. A small blocking factor induces lot's of gcd computations and therefore slows down the algorithm; with a huge blocking factor, we can do a lot of unnecessary iterations before we realize that we have found a factor. In practice, the blocking factor should be between 20 and 50, depending on the actual implementation.

2.4. Example Let $N = 31613 = 101 \cdot 313$. We use the function $g(X) = X^2 + 1$ with starting value $x_0 = 5$ as generating pseudo random function. Then we get

i	x_i	y_i	$\gcd(x_i - y_i, N)$
0	5	5	–
1	26	677	1
2	677	27133	1
3	15748	26820	1
4	27133	19308	313

Algorithm 7 stops if we find an index r such that $x_r \equiv x_{2^r} \pmod{p}$, i.e. if the length of the cycle in the sequence of pseudo random elements $(x_k \pmod{p})_{k \geq 0}$ divides the index r . In a variant of the algorithm proposed by Brent ([Br80]), a different method is used for finding two elements $x_i, x_j, i < j$, such that the cycle length divides $j - i$. Brent suggests to compare for all $k \geq 1$ the sequence element $x_{2^k - 1}$ only with the elements x_j for $2^k + 2^{k-1} \leq j \leq 2^{k+1} - 1$. After we have done these tests, we have checked all possible cycle lengths between $2^{k-1} + 1$ and 2^k .

2.5. Example We try to factor the same integer $N = 31613$ as in Example 2.4. Again we use the starting value $x_0 = 5$ and the generating function $g(X) = X^2 + 1$. This choice leads to the following table:

k	$x_{2^k - 1}$	$x_{2^k + 2^{k-1}}, \dots, x_{2^{k+1} - 1}$	$\gcd(x_{2^k - 1} - x_j, N)$
1	26	15748	1
2	15748	26820, 21812	1, 1
3	21812	29011, 5223, 29324, 23377	313, ...

The advantage of Brent's variant versus Algorithm 7 is the fact that the number of iterations until a cycle is found can be up to 25 % smaller. Obviously, this can lead to a big running time effect. Again, all the remarks made to Floyd's variant of the Pollard ρ -Algorithm remain valid; especially it is advisable to use a blocking techniques (see Remark 2.1) to reduce the number of gcd computations.

8. Algorithm (Pollard's ρ -Method, Brent's variant)

We use the pseudo random sequence generated by the polynomial $g(X) = X^2 + a \text{ Mod } N$.

INPUT: composite $N \in \mathbb{N}_{>1}$, starting value x_0 , $a \in \mathbb{Z}$.

OUTPUT: Factor of N .

```
(1) Set  $x = x_0 \text{ Mod } N$ ,  $y = x_0^2 + a \text{ Mod } N$  and  $k = 1$ .
(2) if  $(\text{gcd}(x - y, N) > 1)$  then
(3)   return  $(\text{gcd}(x - y, N))$ 
(4) fi
(5) while (1) do
(6)   Set  $x = y$ .
(7)   for ( $j = 1$  to  $k$ ) do
(8)     Set  $y = y^2 + a \text{ Mod } N$ .
(9)   od
(10)  for ( $j = 1$  to  $k$ ) do
(11)    Set  $y = y^2 + a \text{ Mod } N$ .
(12)    if  $(\text{gcd}(x - y, N) > 1)$  then
(13)      return  $(\text{gcd}(x - y, N))$ 
(14)    fi
(15)  od
(16)  Set  $k = 2k$ .
(17) od
```

Chapter 3

Factoring Algorithms Using Smooth Group Orders

In the previous chapters, we have introduced some basic algorithms which should be used as a first step in a factoring procedure. Both the trial division and the Pollard ρ -method can be used to find very small factors in a composite, odd integer N . In this chapter we describe three other algorithms for finding small factors. All these algorithms are based on the same idea, they employ the smoothness of the order of some abelian group. An integer $k \in \mathbb{N}_{>1}$ is called **\mathcal{B} -smooth** if all prime factors of k are smaller than \mathcal{B} ; it is called **\mathcal{B} -power smooth**, if all prime powers which divide k are smaller than \mathcal{B} . Let in the remainder of this chapter, N be the odd, composite integer which we want to factor, and let p be a prime divisor of N .

3.1 The Pollard $(p - 1)$ Factoring Method

In the first section of this chapter, we describe the $(p - 1)$ method due to Pollard ([Po74]). This method was used to factor the eighth Fermat number $F_8 = 2^{2^8} + 1$ (see [BrPo81]). Pollard's algorithm works in the abelian group $(\mathbb{Z}/N\mathbb{Z})^*$. Using the Theorem of Lagrange, we know that for every $a \in \mathbb{Z}$ with $\gcd(a, p) = 1$

$$a^{p-1} - 1 \equiv 0 \pmod{p}.$$

And even for exponents $e = k \cdot (p - 1)$ for some $k \in \mathbb{N}$, we have

$$a^e - 1 \equiv 0 \pmod{p}.$$

On the other hand, if $\gcd(a, N) = 1$, and if the order of a modulo N is not a divisor of e , then

$$a^e - 1 \not\equiv 0 \pmod{N},$$

which means that $d = \gcd(a^e - 1, N)$ is a proper divisor of N . Of course, we do not know neither p nor $p - 1$. But if $p - 1$ factors into a power product of prime numbers smaller or equal to some bound \mathcal{B}_1 (i.e. $p - 1$ is \mathcal{B}_1 -smooth), a multiple of $p - 1$ can be computed as the product of all prime numbers bounded by \mathcal{B}_1 “with small exponents”. We illustrate this fact with an example:

3.1. Example Let $N = 437$. We randomly choose the base $a = 2$ and use the exponent $e = 2^4 \cdot 3^4$. Then

$$a^e = a^{1296} = 305 \text{ Mod } N \quad \text{and} \quad \gcd(305 - 1, N) = 19.$$

In fact, we have $N = 19 \cdot 23$. Note that e is a multiple of $19 - 1$ (18 is 3-smooth), but it is no multiple of $23 - 1$ since 22 is not 3-smooth.

It remains to find a “good” way to choose the exponents e . If we want to catch all numbers below \mathcal{B}_2 which are composed of prime numbers below \mathcal{B}_1 , we have to determine for any prime $p \leq \mathcal{B}_1$ the maximal exponent e_p such that $p^{e_p} \leq \mathcal{B}_2$; then we set

$$e = \prod_{p \leq \mathcal{B}_1} p^{e_p}.$$

Of course, e is not computed explicitly, but we rather store the list of prime numbers $p_i \leq \mathcal{B}_1$ with their multiplicities e_{p_i} , or we compute these primes with the Sieve of Erathostenes (Algorithm 6). If we first compute $a^e \text{ Mod } N$ and test afterwards, whether $\gcd(a^e - 1, N) > 1$, it might be that we do a lot of extra work. On the other hand, computing a gcd is an expensive operation, such that a gcd test after each exponentiation with a prime power $p_i^{e_i}$ slows down the algorithm. Therefore we again use a blocking technique similar to Algorithm 7.

9. Algorithm (Pollard’s $(p - 1)$ Algorithm)

INPUT: $N \in \mathbb{N}_{>1}$, smoothness bounds $\mathcal{B}_1, \mathcal{B}_2$.
 OUTPUT: Factor of N or “no factor found”.

- (1) Compute all prime numbers p_1, \dots, p_k smaller or equal than \mathcal{B}_1 .
- (2) Compute the multiplicities $e_i = \lfloor \log_{p_i} \mathcal{B}_2 \rfloor$ for $1 \leq i \leq k$.
- (3) Choose $b \in \{2, \dots, N - 2\}$ at random.
- (4) **if** $(\gcd(b, N) > 1)$ **then**
- (5) **return** $(\gcd(b, N))$
- (6) **fi**
- (7) **for** $(j = 1 \text{ to } k)$ **do**
- (8) Compute $b = b^{p_j^{e_j}} \text{ Mod } N$ with fast exponentiation.
- (9) **if** $(j = 0 \text{ Mod } 100)$ **then**
- (10) **if** $(\gcd(b - 1, N) > 1)$ **then**
- (11) **return** $(\gcd(b - 1, N))$
- (12) **fi**
- (13) **fi**
- (14) **od**
- (15) **return** (“no factor found”)

Algorithm 9 already uses a blocking strategy whose blocking rate is determined by the divisibility test for j in step (8). Note that a small blocking factor slows down the algorithm since you have to compute a lot of gcds, but with a large blocking factor you can “loose” a factor (i.e. the gcd in step (9) is equal to N). Fortunately, it is possible to use a backtracking strategy to almost always find a factor of N in the “last” block. Experience shows that blocking factors of size approximately 100 are a good choice.

If it happens that Algorithm 9 does not output a factor of N , then we have chosen an integer a whose order modulo every prime divisor p of N was not \mathcal{B}_1 -smooth or not \mathcal{B}_2 -power smooth. We can restart the algorithm with bigger bounds \mathcal{B}_1 and \mathcal{B}_2 or choose another random element $a \in \mathbb{Z}/N\mathbb{Z}$. Nevertheless it is better in practice to use another algorithm, since the probability of choosing an element with \mathcal{B}_1 -smooth order if the group order $p - 1$ is not \mathcal{B}_1 -smooth is very small.

3.2 Continuations

It is well known that the biggest prime factor of a random number k is expected to have about 60 % of the decimal length of k . This observation explains one of the main problems of the $(p - 1)$ -method. It often happens that exactly one prime factor of $p - 1$ (the biggest one) is missing in the exponent e . We could increase the bound \mathcal{B}_1 , but this would drastically increase the running time. The solution to this problem was found in so called **continuations**, a second step in the $(p - 1)$ -algorithm. Assume that we know $a' = a^e \text{ Mod } N$, where e is chosen as described above and that the order of $a' \text{ mod } p$ is equal to a prime q . We explain five different continuations, which will factor N , if $\mathcal{B}_1 < q < \mathcal{B}_3$ for a third bound $\mathcal{B}_3 > \mathcal{B}_1$. A practical comparison of these continuations will be given in Section 3.5, where we present running times for these continuations for the elliptic curve factoring algorithm.

3.2.1 The Standard Continuation

The trivial solution for “catching” q would be to test for all prime numbers $\mathcal{B}_1 < q_i < \mathcal{B}_3$, $1 \leq i \leq r$, whether $\text{gcd}((a')^{q_i} - 1, N) > 1$. It has however an important drawback: we need a lot of extra multiplications for testing all these prime numbers $\mathcal{B}_1 < q_1 < q_2 < \dots < q_r < \mathcal{B}_3$. The main observation for preventing this drawback is the fact that the difference between two such primes q_i and q_{i+1} is very often small. Therefore we gain an advantage if we precompute the elements $(a')^{r_i} \text{ Mod } N$ for all the differences $r_i = q_{i+1} - q_i$. Then we can test successive primes q_i, q_{i+1} with one multiplication

$$(a')^{q_{i+1}} = (a')^{q_i + (q_{i+1} - q_i)} = (a')^{q_i} \cdot (a')^{r_i} \text{ Mod } N .$$

Obviously, we should again not compute a gcd for testing each prime q_i separately, but we should use a blocking technique.

3.2.2 The Improved Standard Continuation

Assume that the order of $a' \bmod p$ is equal to a prime q with $\mathcal{B}_1 < q < \mathcal{B}_3$. Setting $w = \lceil \sqrt{\mathcal{B}_3} \rceil$, there exist integers $u, v \in \mathbb{N}$ satisfying

$$q = v \cdot w - u \quad \text{and} \quad 0 \leq u \leq w, \quad \left\lfloor \frac{\mathcal{B}_1}{w} \right\rfloor \leq v \leq \left\lceil \frac{\mathcal{B}_3}{w} \right\rceil.$$

Then it is easy to see that

$$(a')^{v \cdot w} \equiv (a')^u \pmod{p},$$

such that $\gcd((a')^{v \cdot w} - (a')^u, N) \geq p$.

Obviously, such a decomposition $q_i = v_i \cdot w - u_i$ can be done for all primes $\mathcal{B}_1 < q_i < \mathcal{B}_3$. The Improved Standard Continuation uses these decompositions in its two parts: First we compute and store for all values u_i the elements $(a')^{u_i} \bmod N$ in a table. Then compute the element $b = (a')^w \bmod N$, compute for all possible values v_i the elements $b^{v_i} \bmod N$ and check – again with a blocking technique – whether $\gcd(b^{v_i} - (a')^{u_i}, N)$ is a proper factor of N . Note that it is an advantage to check the primes q_i in ascending order, since the difference between v_{i+1} and v_i is usually very small (zero or one). Knowing b^{v_i} , the computation of $b^{v_{i+1}}$ can therefore be done with “a few” multiplications.

One disadvantage of the Improved Standard Continuation is the large memory amount which is necessary for storing the elements computed in the first part of the algorithm. We can drastically reduce the amount of needed memory by choosing w “near” to $\sqrt{\mathcal{B}_3}$ and congruent to $0 \pmod{30}$. Since we are looking for a prime number q , we immediately know that u can not be divisible by 2, 3 and 5 and we do not have to compute and store corresponding elements. This trick reduces the memory need by about 75 %.

3.2.3 The Prime Pairing Continuation

The Prime Pairing Continuation is a further improvement to the Improved Standard Continuation. Its idea is quite simple: Assume that there exist for some $w \in \mathbb{Z}$ integers $v, u \in \mathbb{Z}$ such that

$$q_1 = v \cdot w - u \quad \text{and} \quad q_2 = v \cdot w + u$$

for two primes $\mathcal{B}_1 < q_1, q_2 < \mathcal{B}_3$. Then we can check these two primes simultaneously by computing $\gcd\left((a')^{(vw)^2} - (a')^{u^2}, N\right)$. Since

$$(a')^{(vw)^2} - (a')^{u^2} = \left((a')^{(vw)^2 - u^2} - 1\right) \cdot (a')^{u^2} = \left((a')^{(vw-u) \cdot (vw+u)} - 1\right) \cdot (a')^{u^2},$$

this gcd will be greater than 1 if either q_1 or q_2 is the order of $a' \bmod p$.

Usually the bounds $\mathcal{B}_1, \mathcal{B}_3$ used in the continuations are fixed. Then we can precompute the optimal value for w , such that we find the maximal number of prime pairs, and store these values together with the bounds $\mathcal{B}_1, \mathcal{B}_3$. The Prime Pairing Continuation starts similar as the Improved Standard Continuation: for all primes $\mathcal{B}_1 < q_i < \mathcal{B}_3$, we compute integers v_i, u_i such that either $q_i = v_i \cdot w - u_i$ or $q_i = v_i \cdot w + u_i$ (of course, we should try to hold the set of u_i 's as small as possible). The first part of the continuation consists of computing (and storing) the elements $(a')^{u_i^2} \bmod N$ for all possibilities u_i . Note that

$$(a')^{(u+1)^2} = (a')^{u^2} \cdot (a')^{2u} \cdot a',$$

such that we can compute the element with exponent $(u + 1)^2$ with 2 multiplications, if we know $(a')^{2u}$. After this computation of the so called Babysteps, we compute $b = (a')^{w^2} \text{ Mod } N$ and use the same strategy with b for testing all possible values for v_i .

Obviously, it can happen that there exist no pairing prime for several primes in the interval $[\mathcal{B}_1, \mathcal{B}_3]$. In this case, we test a pseudo pair, i.e. a prime together with a composite number. Again we can use a trick in choosing w similar the Improved Standard Continuation to reduce the memory amount for the table computed in the Babystep part of the continuation.

3.1. Exercise Describe an algorithm which finds for given bounds $\mathcal{B}_1, \mathcal{B}_3$ the optimal value for w (Hint: Look at all primes between \mathcal{B}_1 and \mathcal{B}_3 modulo $2w$).

3.2.4 The Birthday Paradoxon Continuation

The Birthday Paradoxon Continuation is based on the same principle as Pollard's ρ -factoring method (see Algorithm 7 in Section 2.4). We already know that the order of $a' \text{ mod } p$ is equal to some prime $\mathcal{B}_1 < q < \mathcal{B}_3$. If we generate approximately $\sqrt{\mathcal{B}_3}$ random elements in the subgroup generated by $a' \text{ mod } p$ (a cyclic subgroup of order q), at least two of these elements will be equal with probability greater $\frac{1}{2}$.

We generate a pseudo random sequence $(s_i)_{i \geq 1}$ of integers and compute the corresponding elements $x_i = (a')^{s_i} \text{ Mod } N$. After generation of approximately $\sqrt{\mathcal{B}_3}$ random exponents s_i , we expect to find two exponents such that $s_i \equiv s_j \text{ mod } q$ and therefore $x_i \equiv x_j \text{ mod } p$. But then $\text{gcd}(x_i - x_j, N)$ gives a factor of N . As shown in Section 2.4, we can use Floyd's trick to restrict ourself to compare x_i and x_{2i} . Note that the size of the random exponents s_i should be bounded. Moreover, it is advisable to use different pseudo random number generators as proposed in Section 2.4, such that the computation of $(a')^{s_{i+1}} \text{ Mod } N$ with knowledge of $(a')^{s_i} \text{ Mod } N$ becomes easier (for a suitable pseudo random number generator, see [Mo92]). In practice it is again preferable to use a blocking technique.

3.2.5 Montgomery's FFT - Continuation

We briefly mention another continuation first published by Peter L. Montgomery (see [Mo87]), which is an extension of the Birthday Paradoxon Continuation. In the Birthday Paradoxon Continuation we always compare two elements of a randomly generated sequence of elements in the subgroup generated by $a' \text{ mod } p$. Obviously, it would be much better if we could compare several elements of this sequence in parallel. An algorithm for doing this test is based on the following theorem.

10. Theorem Assume that for a sequence $(x_k \text{ Mod } N)_{k \geq 0}$ of elements in $\mathbb{Z}/N\mathbb{Z}$ we define the two polynomials

$$f(X) = \prod_{i=0}^{r_f} (X - x_i) \text{ Mod } N \quad \text{and} \quad g(X) = \prod_{i=0}^{r_g} (X - x_{i+r_f+1}) \text{ Mod } N .$$

If there exist two indices $0 \leq i \leq r_f$ and $r_f < j \leq r_g + r_f + 1$ such that $x_i \equiv x_j \text{ mod } p$, but $x_i \not\equiv x_j \text{ mod } N$, and all pairs of elements are not congruent modulo any prime divisor of

N/p , then Euclid's algorithm for computing the gcd of $f(X)$ and $g(X)$ fails and gives us a factor of N .

Montgomery's FFT - Continuation consists of three steps: first we compute a pseudo random sequence $(x_k \bmod N)_{k \geq 0}$ as explained in the last subsection. Then we compute the polynomials $f(X)$ and $g(X)$ described in Theorem 10. In the third step, we try to compute the gcd of $f(X)$ and $g(X)$ – if the gcd computations fails, we are happy since we found a factor of N . The name of this continuation is based on the fact that the polynomial operations should use the FFT method for multiplication, since then the degree of the polynomials can be quite large. In practice, one chooses the degree of the polynomials up to 16384.

3.1. Exercises

1. Prove Theorem 10. Hint: Examine the result if Euclid's algorithm would be successful.
2. Extend the FFT algorithm such that we find a factor of N if we have $x_i \equiv x_j \pmod{p}$, but $x_i \not\equiv x_j \pmod{N}$ for $0 \leq i, j \leq r_f$, i.e. both elements x_i, x_j are roots of $f(X)$. (Hint: Look at the formal derivative of $f(X)$.)
3. What can we do if there exist no indices $0 \leq i \leq r_f$ and $r_f < j \leq r_g + r_f + 1$ such that $x_i \equiv x_j \pmod{N}$, but the degree of $\gcd(f(X), g(X))$ is non zero ?
4. Assume that $\deg(g) > \deg(f)$. Show that we do not have to compute $g(X)$ explicitly, but that we can reduce $g(X)$ with $f(X)$. Therefore we can increase the number of sequence elements "stored in $g(X)$ " without increasing the degree of the polynomials involved in the computation.

3.3 The Principle Behind the $(p - 1)$ Method

Let us analyze Pollard's $(p - 1)$ -method more abstract. We are working in the abelian group $\mathcal{G}_N = ((\mathbb{Z}/N\mathbb{Z})^*, \cdot)$. For each prime divisor p of N we can define a homomorphism

$$\phi: \mathcal{G}_N \longmapsto \mathcal{G}_p,$$

which is reduction modulo p . In the $(p - 1)$ -algorithm, the abelian group \mathcal{G}_p is given as $((\mathbb{Z}/p\mathbb{Z})^*, \cdot)$. If we find an element of the kernel of ϕ which is not the one-element in \mathcal{G}_N , we find a factor of N . Such an element is found if we find an element $g \in \mathcal{G}_N$ such that the order of $\phi(g)$ is composed of small prime numbers only, whereas this is false for the order of g in \mathcal{G}_N .

3.2. Example Let $N = 491389$. We would like to find a non trivial divisor of N . Now $N = 383 \cdot 1283$ and $383 - 1 = 2 \cdot 191$, $1283 - 1 = 2 \cdot 641$. Both multiplicative groups $(\mathbb{Z}/383\mathbb{Z})^*$ and $(\mathbb{Z}/1283\mathbb{Z})^*$ have an order, which does not decompose into small prime factors. We are therefore stuck with the $(p - 1)$ -method because we have no other choice for \mathcal{G}_p .

In the following sections we will describe two algorithms which use the same principle as the $(p - 1)$ algorithm, but work in different abelian groups \mathcal{G}_N .

3.4 The $(p + 1)$ Factoring Method

Hugh C. Williams used the idea of the $(p - 1)$ factoring algorithm to develop a $(p + 1)$ algorithm which we describe in this section (see [Wi82]).

For a prime number p , we wish to find a multiplicative group with order $p + 1$. Williams proposed to use the group $\mathcal{G}_p = \mathbb{F}_{p^2}^* / \mathbb{F}_p^*$, where $\mathbb{F}_{p^2}^*$, \mathbb{F}_p^* is the multiplicative group of the finite field with p^2 , p elements, respectively. Obviously, this is a multiplicative group with the desired group order. How can we represent this group? Let $\alpha \in \mathbb{F}_{p^2}^*$. Then α is the root of a polynomial $f_\alpha(X) \in \mathbb{F}_p[X]$ of degree 2 and $\alpha \notin \mathbb{F}_p$ if and only if $f_\alpha(X) = X^2 - aX + b$ is irreducible in \mathbb{F}_p . Note that such a polynomial is irreducible in \mathbb{F}_p if and only if the discriminant $\Delta(f_\alpha) = a^2 - 4b$ is no square in \mathbb{F}_p .

Williams $(p + 1)$ algorithm now principally works as follows: choose random integers $a, b \in \mathbb{Z}/N\mathbb{Z}$ and set $\mathcal{G}_N = \mathbb{Z}/N\mathbb{Z}[X] / (f(X) \cdot \mathbb{Z}/N\mathbb{Z}[X])$, where $f(X) = X^2 - a \cdot X + b \in \mathbb{Z}/N\mathbb{Z}[X]$. Then we choose smoothness bounds $\mathcal{B}_1, \mathcal{B}_2$ and set

$$e = \prod_{p \leq \mathcal{B}_1} p^{e_p},$$

where e_p is chosen as $e_p = \lfloor \log_p \mathcal{B}_2 \rfloor$. For random integers $k_1, k_2 \in \mathbb{Z}/N\mathbb{Z}$, we compute $(k_1 \cdot X + k_2)^e \equiv s \cdot X + t \pmod{f(X)}$. Let $\alpha \in \overline{\mathbb{F}_p}$ be a root of $f(X) \pmod{p}$ in the algebraic closure of \mathbb{F}_p . There are two possible situations:

1. If $\alpha \in \mathbb{F}_{p^2}^* - \mathbb{F}_p^*$ (i.e. $f(X)$ is irreducible modulo p) and $p + 1$ is \mathcal{B}_2 -power smooth, but the order of $k_1 \cdot \alpha + k_2$ in \mathcal{G}_N is not \mathcal{B}_2 -power smooth, then $(k_1 \cdot \alpha + k_2)^e = s \cdot \alpha + t \equiv 1_p$ in \mathcal{G}_p , but $s \cdot \alpha + t \not\equiv 1_N$ in \mathcal{G}_N . Therefore $\gcd(s, N)$ will be a proper factor of N .
2. For $\alpha \in \mathbb{F}_p^*$ (i.e. $f(X)$ factors modulo p) and $(p - 1)$ being \mathcal{B}_2 -power smooth, we have $(k_1 \cdot \alpha + k_2)^e = s \cdot \alpha + t \equiv 1 \pmod{p}$. If s is not invertible modulo N , then we have found a factor of N , otherwise we set $\alpha \equiv (1 - t) \cdot s^{-1} \pmod{N}$. Since α is a root of $f_\alpha(X)$ modulo p , $\gcd(f_\alpha(\alpha), N)$ will be a proper factor of N , if the order of $(k_1 \cdot \alpha + k_2)$ in \mathcal{G}_N is not \mathcal{B}_2 -power smooth.

Note that we do not know whether indeed $\alpha \notin \mathbb{F}_p^*$. For deciding this question, we would have to check whether the discriminant of $f_\alpha(X)$ is a square modulo the unknown factor p or not. In fact, the algorithm of Williams is either a $(p + 1)$ or a $(p - 1)$ factoring algorithm, depending on the “quadratic behavior” of the discriminant $\Delta(f_\alpha)$.

3.3. Example Let $N = 473$. Assume that we choose the generating polynomial $f_\alpha(X) = X^2 - 2 \in \mathbb{Z}/N\mathbb{Z}[X]$. Moreover, we decide to take the bounds $\mathcal{B}_1 = \mathcal{B}_2 = 5$, such that $e = 60$. We exponentiate the random polynomial $X + 1$ and obtain

$$(X + 1)^e \equiv 319X + 428 \pmod{f_\alpha(X)}.$$

Since we do not know whether $f_\alpha(X)$ is irreducible modulo a prime divisor p of N , we compute $\gcd(319, N) = 11$ and find the complete factorization $N = 11 \cdot 43$. In fact, we can check that $f_\alpha(X)$ is irreducible modulo any prime divisor of N and that e is a multiple of $p + 1 = 12$.

Assume that we choose the polynomial $g_\alpha(X) = X^2 - 3 \in \mathbb{Z}/N\mathbb{Z}[X]$. If we now exponentiate $X + 1$, we obtain

$$(X + 1)^e \equiv 198 \cdot X + 276 \pmod{g_\alpha(X)}.$$

We try to invert $198 \pmod{N}$, which fails and again we have found the prime factorization of N . Note that $g_\alpha(X)$ is reducible modulo 11, but irreducible modulo 43. Moreover, e is a multiple of $11 - 1$, such that the $(p - 1)$ criterion succeeds.

The original description of Williams was slightly different than the idea given above, since the polynomial $f_\alpha(X)$ is chosen in a special form. Assume that the modulus has the form $f_\alpha(X) = X^2 - aX + 1$, where a is chosen at random. In order to be irreducible modulo p , the discriminant $a^2 - 4$ of $f_\alpha(X)$ must be a non square modulo p . Over finite fields, the roots of polynomials of this form satisfy the following conditions:

3.1. Lemma *Let $\alpha \in \mathbb{F}_{p^2}^*$ be a root of the polynomial $X^2 - aX + 1 \in \mathbb{F}_p[X]$. Then $\alpha + \alpha^{-1} = a$ and $\alpha^{p+1} = 1$.*

Proof: Assume that α and β are the two roots in $\mathbb{F}_{p^2}^*$ of the given polynomial. By comparing coefficients, we get $\alpha \cdot \beta = 1$ and $\alpha + \beta = a$. Therefore $\beta = \alpha^{-1}$, and the first part of the lemma is proven.

The second part of the lemma follows from the observation, that the roots of the polynomial $X^2 - aX + 1$ are given by α and $\beta = \alpha^p$. This statement follows from the fact that for a polynomial $g(X) \in \mathbb{F}_p[X]$ we have $g(X^p) = g(X)^p$. Therefore $\beta = \alpha^{-1} = \alpha^p$, and so $\alpha^{p+1} = 1$.

Williams had the idea how the facts of Lemma 3.1 can be used to do all computations with elements of $\mathbb{Z}/N\mathbb{Z}$. It is therefore not necessary to consider polynomials defined over $\mathbb{Z}/N\mathbb{Z}$ as we did in the explanation of the idea. Before we start with the description of the factoring algorithm, we explain several basics for finite fields.

Assume that $\alpha \in \mathbb{F}_{p^2}$ is a root of the polynomial $f_\alpha(X) = X^2 - aX + 1 \in \mathbb{F}_p[X]$. By Lemma 3.1, the element $\alpha + \alpha^{-1} \in \mathbb{F}_p$. The following lemma will prove the fact that even for all $n \geq 0$ the elements $V_n(\alpha) = \alpha^n + \alpha^{-n}$ are elements of the prime field \mathbb{F}_p . Moreover, Lemma 3.2 explains the idea of Algorithm 11, which can be used to compute $V_n(\alpha)$ for given n and $V_1(\alpha)$.

3.2. Lemma *Let $V_n(\alpha) = \alpha^n + \alpha^{-n}$ for $\alpha \in \mathbb{F}_{p^2}^*$. The Lucas function $V_n(\alpha)$ satisfies the following rules:*

1. $V_0(\alpha) = 2$,
2. $V_{n+m}(\alpha) = V_m(\alpha) \cdot V_n(\alpha) - V_{n-m}(\alpha)$ for $n \geq m$.

Proof: Both claims of Lemma 3.2 follow by a direct calculation.

With the help of this Lemma, it is possible to develop a binary algorithm for computing $V_n(\alpha)$ for given $n \in \mathbb{N}$ and $V_1(\alpha) \in \mathbb{F}_p$. Assume that we know $V_k(\alpha)$ and $V_{k-1}(\alpha)$. Then

we use rule 2 of Lemma 3.2 with suitable integers m to get the formulas

$$\begin{aligned} V_{2k}(\alpha) &= V_k(\alpha)^2 - 2, \\ V_{2k-1}(\alpha) &= V_k(\alpha) \cdot V_{k-1}(\alpha) - V_1(\alpha), \\ V_{2k+1}(\alpha) &= V_{k+1}(\alpha) \cdot V_k(\alpha) - V_1(\alpha) \\ &= \left(V_k(\alpha) \cdot V_1(\alpha) - V_{k-1}(\alpha) \right) \cdot V_k(\alpha) - V_1(\alpha). \end{aligned}$$

Similar to Algorithm 3, we scan the bits of n from the high order bit to the left order bit. Note that it is necessary to know $V_k(\alpha)$ and $V_{k-1}(\alpha)$ to be able to compute $V_s(\alpha)$ for $s = 2k - 1, 2k, 2k + 1$. Therefore we always compute the two values $V_{k-1}(\alpha)$ and $V_k(\alpha)$ in parallel for suitable indices k .

11. Algorithm (Left-to-Right Binary Lucas Computation)

INPUT: $V_1(\alpha) \in \mathbb{F}_p, n \in \mathbb{N}$.
 OUTPUT: $V_n(\alpha) \in \mathbb{F}_p$.

- (1) Initialize $b = V_1(\alpha)$ and $c = 2 \text{ Mod } p$.
- (2) Compute the binary decomposition $n = \sum_{i=0}^s \beta_i 2^i, \beta_s = 1$.
- (3) **for** ($i = s - 1$ **downto** 0) **do**
- (4) **if** ($\beta_i = 1$) **then**
- (5) Set $h = c$ and $c = b^2 - 2 \text{ Mod } p$.
- (6) Set $b = V_1(\alpha) \cdot (b^2 - 1) - b \cdot h \text{ Mod } p$.
- (7) **else**
- (8) Set $h = b$ and $b = b^2 - 2 \text{ Mod } p$.
- (9) Set $c = h \cdot c - V_1(\alpha) \text{ Mod } p$.
- (10) **fi**
- (11) **od**
- (12) **return** (b)

The correctness of Algorithm 11 follows from the invariant that after iteration i of the **for**-loop the value of the variables b, c are $b = V_k(\alpha)$ and $c = V_{k-1}(\alpha)$ for $k = \sum_{j=i}^s \beta_j \cdot 2^{j-i}$.

Assume that we choose the index n , such that $p + 1$ is a divisor of n . By Lemma 3.1, we know that $\alpha^{p+1} = 1$ and therefore we immediately see that

$$V_n(\alpha) = (\alpha^{p+1})^{n/(p+1)} + (\alpha^{p+1})^{-n/(p+1)} = 1 + 1 = 2 \text{ Mod } p.$$

This equation brings us back to the original problem, the factorization of some integer N . In this situation, we obviously do not know a prime factor p of N , but we can

do the computations modulo N . We start by choosing a random polynomial $f_\alpha(X) = X^2 - aX + 1 \in \mathbb{Z}/N\mathbb{Z}[X]$. Then we choose some “exponent” e which is a product of all primes below some bound \mathcal{B}_1 with some suitable exponents (determined as usual by the second exponent bound \mathcal{B}_2). For a root α of $f_\alpha(X) = X^2 - aX + 1$, we compute the e -th Lucas function $V_e(\alpha)$ by using Algorithm 11. Again we have the same uncertainty as in the first description of this idea. We can not check whether the polynomial $f_\alpha(X)$ is irreducible modulo any prime divisor p of N , since we cannot determine whether the discriminant of $f_\alpha(X)$ is a square modulo p or not. But there are again two good situations:

1. If $f_\alpha(X)$ is irreducible for some prime divisor p of N , and if $p+1$ is \mathcal{B}_2 -power smooth, then $V_e(\alpha) \equiv 2 \pmod{p}$, and therefore $\gcd(V_e(\alpha) - 2, N) \geq p$.
2. If $f_\alpha(X)$ factors modulo some prime divisor p of N , and if $p-1$ is \mathcal{B}_2 -power smooth, then by the Theorem of Lagrange $V_e(\alpha) = \alpha^e + \alpha^{-e} \equiv 2 \pmod{p}$, and again $\gcd(V_e(\alpha) - 2, N) \geq p$.

As already mentioned earlier, it might happen that the gcd-computation does not reveal a proper factor of N , but N itself. In this case, we should start a backtracking procedure with lower bounds \mathcal{B}_1 and \mathcal{B}_2 . This backtracking step will return a proper factor with high probability. Before we formally describe the Williams $(p+1)$ -algorithm, we show an example.

3.4. Example *We try to factor the same integer $N = 473$ as in Example 3.3. We choose the random polynomial $f_\alpha(X) = X^2 - 3X + 1$, and exponent $e = 60$. With Algorithm 11, we compute $V_{60}(\alpha)$ for a root α of $f_\alpha(X)$ and obtain the result $V_{60}(\alpha) = 409 \pmod{N}$. This elements leads to a factorization of N , since $\gcd(409 - 2, N) = 11$. We test that the chosen polynomial $f_\alpha(X)$ factors modulo 11, such that the second “good” case happened – we used the smoothness of $11 - 1$.*

On the other hand, if we choose $f_\alpha(X) = X^2 - 5X + 1$, then $V_{60}(\alpha) = 277 \pmod{N}$. With a gcd-computation, we again find the factor 11 of N , but now using the smoothness of $11 + 1$ (this choice of $f_\alpha(X)$ is irreducible modulo 11).

Collecting the ideas, we have the following algorithm:

12. Algorithm (William's ($p + 1$) Algorithm)

INPUT: $N \in \mathbb{N}$, smoothness bounds $\mathcal{B}_1, \mathcal{B}_2$.

OUTPUT: Factor of N or “no factor found”.

- (1) Compute all prime numbers p_1, \dots, p_k smaller or equal than \mathcal{B}_1 .
- (2) Compute the multiplicities $e_i = \lfloor \log_{p_i} \mathcal{B}_2 \rfloor$ for $1 \leq i \leq k$.
- (3) Choose $a \in \{2, \dots, N - 1\}$ at random and set $b = a$.
- (4) **if** ($\gcd(b, N) > 1$) **then**
- (5) **return** ($\gcd(b, N)$)
- (6) **fi**
- (7) **for** ($j = 1$ **to** k) **do**
- (8) Set $r = p_j^{e_j}$ and compute $b = V_r(b) \text{ Mod } N$ with Algorithm 11.
- (9) **if** ($j = 0 \text{ Mod } 100$) **then**
- (10) **if** ($\gcd(b - 2, N) > 1$) **then**
- (11) **return** ($\gcd(b - 2, N)$)
- (12) **fi**
- (13) **fi**
- (14) **od**
- (15) **return** (“no factor found”)

Note that in the Lucas computation in step (7) the element $V_1(\alpha) = a$ is part of the input of Algorithm 11. Again we have used a gcd-blocking factor of 100 in step (8) of this description.

3.2. Exercises

1. Prove that in fields K of characteristic p we have $(a + b)^p = a^p + b^p$ for all $a, b \in K$. Use this fact to show that for a polynomial $g(X) \in \mathbb{F}_p[X]$ we have $g(X^p) = g(X)^p$.
2. Design a binary algorithm for computing $V_n(\alpha)$ which scans the bits of n from the low order bit to the high order bit (compare Algorithm 3).
3. Describe a $p^2 + p + 1$ factoring algorithm. Does such an algorithm make sense in practice?

3.5 The Elliptic Curve Method (ECM)

Hendrik W. Lenstra used the principle described in Section 3.3 to develop a new factoring algorithm, the **Elliptic Curve Method (ECM)** (see [Le87]). This factoring algorithm is the most important algorithm for finding small factors of an odd, composite number

N . Several people have implemented ECM and have found lot of practical improvements (for descriptions of practical aspects of the algorithm and implementations, see [Be93], [Mü96], [Mo92]).

3.5.1 Elliptic Curves over $\mathbb{Z}/N\mathbb{Z}$

It was Lenstra's idea to replace \mathcal{G}_N by the group of points on an elliptic curve over $\mathbb{Z}/N\mathbb{Z}$. Before we explain the elliptic curve factoring algorithm, we explain elliptic curves over $\mathbb{Z}/N\mathbb{Z}$ and $\mathbb{Z}/p\mathbb{Z}$.

Assume that $6 \nmid N$ (a trivial assumption, in our context). For a divisor $d > 1$ of N , consider a polynomial congruence of the form

$$y^2 z \equiv x^3 + axz^2 + bz^3 \pmod{d} \quad (3.1)$$

with $a, b \in \mathbb{Z}$. Assume that for $\Delta = 4a^3 + 27b^2$ we have $\gcd(\Delta, N) = 1$. Consider the set

$$\tilde{\mathcal{E}}_d = \left\{ (x, y, z) \in \mathbb{Z}/d\mathbb{Z} ; (x, y, z) \text{ satisfy (3.1)} \right\}.$$

We define an equivalence relation on $\tilde{\mathcal{E}}_d$, which identifies the solutions in $\tilde{\mathcal{E}}_d$, which are trivially equal:

$$(x, y, z) \sim (x', y', z') \iff (x, y, z) \equiv u \cdot (x', y', z') \pmod{d} \text{ with } u \in (\mathbb{Z}/d\mathbb{Z})^*.$$

Denote by \mathcal{E}_d the set of equivalence classes and the elements of \mathcal{E}_d by $(x : y : z)$.

3.1. Theorem \mathcal{E}_d is an abelian group (usually additively written) with zero element $(0 : 1 : 0)$. The canonical mapping $\mathcal{E}_N \mapsto \mathcal{E}_d$ is a homomorphism.

If we manage to find a point $(x : y : z)$ on an elliptic curve \mathcal{E}_N defined over $\mathbb{Z}/N\mathbb{Z}$, which is non-zero but whose image in \mathcal{E}_d is zero for some proper divisor d of N , then $\gcd(z, N)$ is a proper divisor of N .

In order to factor N by this method, we must first choose an elliptic curve \mathcal{E}_N defined over $\mathbb{Z}/N\mathbb{Z}$ and a point on this curve. Although it is easy to write down such an elliptic curve, it is much harder to find a point on a given curve. One therefore starts by choosing the point: Choose $x, y, a \in \{0, \dots, N-1\}$ and define the point $P_N = (x : y : 1)$. In order for P_N to be a point on the elliptic curve $\mathcal{E}_N = (a, b)$, we must have $y^2 \equiv x^3 + ax + b \pmod{N}$, i.e. we have to compute b as

$$b = y^2 - x^3 - ax \pmod{N}. \quad (3.2)$$

Then we must check whether $\gcd(\Delta, N) = 1$, where $\Delta = 4a^3 + 27b^2$. If we have $1 < \gcd(\Delta, N) < N$, then a proper divisor of N is found; for $\gcd(\Delta, N) = N$ we choose a new triple (x, y, a) (but note that this event is very unlikely).

Then we choose bounds $\mathcal{B}_1, \mathcal{B}_3$ and set

$$e = \prod_{p \leq \mathcal{B}_1} p^{e_p},$$

where e_p is maximal such that $p^{e_p} \leq \mathcal{B}_3$. We try to compute the point $e \cdot P_N$ on the elliptic curve \mathcal{E}_N . If the order of the reduction of P_N modulo d is \mathcal{B}_3 -power smooth, then one of

the additions will fail and this failure will lead to a factor of N (we will explain formulas for addition on an elliptic curve in just a moment). Note that we can use a variant of the fast exponentiation Algorithm 3 to compute $e \cdot P_N$. Furthermore we should not compute the integer e in practice, but we will successively multiply a point with p^{e_p} .

A necessary condition for the success of this algorithm is the fact that the order of the reduced curve \mathcal{E}_d is “not too big”. In fact, the famous Theorem of Hasse states that for prime numbers d the order of \mathcal{E}_d is approximately d :

3.2. Theorem *Let d be a prime, and let \mathcal{E}_d be an elliptic curve over $\mathbb{Z}/d\mathbb{Z}$. Then we have the following bounds for the group order $\#\mathcal{E}(\mathbb{Z}/d\mathbb{Z})$:*

$$d + 1 - 2\sqrt{d} \leq \#\mathcal{E}(\mathbb{Z}/d\mathbb{Z}) \leq d + 1 + 2\sqrt{d}.$$

Before we can formulate the ECM factoring algorithm, it remains to explain the addition of points on \mathcal{E}_N . The general method is quite complicated and not suited for an efficient computer implementation. We rather use a restricted method, which enables us to treat points of the form $P_i = (x_i : y_i : 1)$, $i = 1, 2$. If $P_1 + P_2$ is of the form $P_3 = (x_3 : y_3 : 1)$, then the algorithm will yield P_3 . If P_3 is not of this form, then $P_3 = (x_3 : y_3 : z_3)$, where $d = \gcd(z_3, N) > 1$. In this case the algorithm will yield a divisor of N , which in case $d < N$ will be a non trivial one. Here is the method:

1. If $x_1 = x_2 \pmod{N}$ and $y_1 = -y_2 \pmod{N}$, then $P_1 = -P_2$, and therefore $P_3 = (0 : 1 : 0)$.
2. If $x_1 \neq x_2 \pmod{N}$, then try to calculate the inverse k of $x_2 - x_1 \pmod{N}$. If this calculation fails, then $\gcd(x_2 - x_1, N) = d > 1$, and we have found a factor of N , otherwise set $\lambda = (y_2 - y_1) \cdot k \pmod{N}$.

If $x_1 = x_2 \pmod{N}$ and $y_1 = y_2 \pmod{N}$, (so $P_1 = P_2$), then try to calculate the inverse k of $2y_1 \pmod{N}$. If this calculation fails, then $\gcd(2y_1, N) = d > 1$, and we have factored N , otherwise set $\lambda = (3x_1^2 + a) \cdot k \pmod{N}$.

If no error occurred during the inversions, then P_3 is of the form $P_3 = (x_3 : y_3 : 1)$ with

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \pmod{N}, \\ y_3 &= \lambda \cdot (x_1 - x_3) - y_1 \pmod{N}. \end{aligned}$$

3.5. Example *Assume that $N = 14227$. We randomly choose the point $P_N = (1 : 1 : 1)$ and the elliptic curve $\mathcal{E}_N = (1, -1)$ such that $P_N \in \mathcal{E}_N(\mathbb{Z}/N\mathbb{Z})$. The bounds are given as $\mathcal{B}_1 = \mathcal{B}_2 = 11$, thus we get the multiplier $e = 27720$. We try to compute $e \cdot P_N$ with the binary multiplication algorithm, but the computations fails and we find the zero divisor 41 of N . In fact, we have $N = 41 \cdot 347$.*

Why does the algorithm fail ? If we compute the group order of \mathcal{E}_d for $d = 41$, then we get $\#\mathcal{E}(\mathbb{Z}/d\mathbb{Z}) = 35$. Therefore e is a divisor of the order of \mathcal{E}_d , but it is not a divisor of $\#\mathcal{E}_{347}(\mathbb{Z}/347\mathbb{Z}) = 364$.

The big advantage of ECM in comparison to the $p - 1$ or $p + 1$ method is the fact that we have a large set of possible groups at hand. After a test with one elliptic curve failed,

we can choose another elliptic curve and have another chance to find a curve whose group order modulo p is smooth. This observation leads to the following algorithm:

13. Algorithm (Elliptic Curve Method)

INPUT: $N \in \mathbb{N}$, smoothness bounds $\mathcal{B}_1, \mathcal{B}_2$.
 OUTPUT: Factor of N or “no factor found”.

- (1) Compute all prime numbers p_1, \dots, p_k smaller or equal than \mathcal{B}_1 .
- (2) Compute the multiplicities $e_i = \lfloor \log_{p_i} \mathcal{B}_2 \rfloor$ for $1 \leq i \leq k$.
- (3) **repeat**
- (4) Choose $a, x, y \in \{1, \dots, N-1\}$ at random and set $P_N = (x : y : 1)$.
- (5) Set $b = y^2 - x^3 - ax \pmod N$ and $\Delta = 4a^3 + 27b^2 \pmod N$.
- (6) **until** $(\gcd(\Delta, N) < N)$
- (7) **if** $(\gcd(\Delta, N) > 1)$ **then**
- (8) **return** $(\gcd(\Delta, N))$
- (9) **fi**
- (10) **for** $(j = 1 \text{ to } k)$ **do**
- (11) Compute $P_N = p_j^{e_j} \cdot P_N$ with “fast multiplication” on \mathcal{E}_N .
- (12) **if** (one of these additions fails) **then**
- (13) **return** (found factor)
- (14) **fi**
- (15) **od**
- (16) **return** (“no factor found”)

As mentioned before, we can restart this algorithm when we have found no factor. The number of curves which should be tried and the “optimal” bounds \mathcal{B}_1 and \mathcal{B}_2 can be precomputed for the search for a factor with prescribed decimal size. As for the Pollard $p-1$ algorithm, a continuation as second step of the algorithm is of great importance in practice. Moreover there exist several other improvements, we describe a few of these:

1. In general we have to try several curves until we find “a good one” which factors N . Instead of doing the computations sequentially, we can do them in parallel. The main advantage of this parallel computation is the fact that we can use a gcd-trick to reduce the costs for gcds. The computation of the inverse of an element modulo N is by far the most expensive operation in a point addition. This trick works as follows: assume that we want to invert $x_1, \dots, x_k \pmod N$. For $2 \leq i \leq k$ we compute and store the elements $z_i = \prod_{j=1}^i x_j \pmod N$. Then we invert z_k modulo N with the help of an extended gcd algorithm. Then we compute for $j = k, \dots, 1$ the two elements $x_j^{-1} = z_j^{-1} \cdot z_{j-1} \pmod N$ and $z_{j-1}^{-1} = z_j^{-1} \cdot x_j \pmod N$. If we use this trick with parallel computations for 20 elliptic curves, we get a speed up of 2.9 (see [Be93]).

2. It is possible to choose another parameterization for the given elliptic curve. In this case, addition of points can be done faster as in the Weierstrass from which we presented here (see [Mo92] or [Mü96]).
3. Instead of choosing a random elliptic curve over $\mathbf{Z}/N\mathbf{Z}$, it is possible to choose elliptic curves (and points) over $\mathbf{Z}/N\mathbf{Z}$, such that the group order of \mathcal{E}_d for every divisor d of N is divisible by at least 16. This fact might enlarge the chance that the group order of \mathcal{E}_d is smooth for at least one divisor d of N (see [AtMo92]).
4. Similar to the $p-1$ -method, it is advisable to use a continuation as a second part of the algorithm. Variants of all continuations described in Section 3.2 can be used for elliptic curves, too. We comment on their practical behavior in the next Section.

3.5.2 Practical Experiences with ECM

ECM can be used to find factors of up to 30 digits “with high probability”. Practical tests showed that the Improved Standard continuation is superior to all other continuations for the search for factors with up to 30 digits. If we try to find even larger factors, then the FFT - continuation of Montgomery is the method which should be used. Here are a few timings gained with the implementation described in [Mü96], first ECM with improved standard continuation is used:

#digits	time per curve	total time
15	32 s	11 m 12 s
20	169 s	3 h 11 m 32 s
25	707 s	42 h 13 m 25 s
30	2 802 s	476 h 20 m 24 s
35	10 247 s	4613 h 59 m 47 s

Similarly, we have the following running times for ECM with FFT - continuation:

#digits	time per curve	total time
20	230 s	3 h 15 m 0 s
25	1 145 s	38 h 10 m 0 s
30	5 782 s	449 h 42 m 40 s
35	20 138 s	3412 h 16 m 20 s

Here is a table of the biggest factors, which were found with ECM up to October 1996 and some information on the factors. Most of these factors were found during the factorization of **Partition Numbers** or numbers of the **Cunningham Project** (a description of partition numbers and factoring successes for these numbers can be obtained per email from challenge-administrator@rsa.com, information on the extended Cunningham project can be found in [BrRi92] or on the WWW-page of Richard Brent). The actual list of biggest ECM factors can be found per anonymous ftp on [nimbus.anu.edu.au:/pub/Brent/champs.ecm](ftp://pub/brent/champs.ecm).

# digits of factor	Factor	Factor of	Found by	Date
47	282079783177872995198818\ 83345010831781124600233	$30^{109} - 1$	P.L. Montgomery	02/25/96
47	120257020000651838057515\ 13732616276516181800961	$5^{256} + 1$	P.L. Montgomery	11/27/95
46	734143962229749943072823\ 1447393744853967297899		P.L. Montgomery	05/06/96
44	278858730440424497775406\ 26664487051863162949	P(19069)	F.D. Berger, A. Müller	06/21/95
43	749775407128676917417312\ 3707569003284024777	$491^{37} - 1$	R.P. Brent	09/14/96
43	568886430504865370279175\ 2405107044435136231	P(19997)	F.D. Berger, A. Müller	03/20/93
43	457837649784554574240798\ 7348754758713076681		H. Kuwakado	09/25/95
43	434396339987830233431769\ 0940709972284426861	$60^{73} - 1$	R.P. Brent	04/07/96
43	210876349664797860369291\ 3892519856162800073	P(14957)	F.D. Berger, A. Müller	03/05/96

Chapter 4

Factoring Algorithms Using Smooth Elements

In the previous chapter, we described factoring algorithm well suited for finding small factors of an integer. In addition, there exists another class of factoring algorithms which depends on smooth elements in different groups. The running time of these algorithms does not depend on the size of the factor we are looking for. In this chapter we will describe some of these algorithms. Again we assume that N is a composite, odd integer which we want to factor. Moreover, N is assume to be no perfect power.

4.1 The Main Idea

The main idea of all these algorithms is based on the following observation: Suppose that for $x, y \in \mathbb{Z}$ we know that

$$x^2 \equiv y^2 \pmod{N}, \quad (4.1)$$

but

$$x \not\equiv \pm y \pmod{N}. \quad (4.2)$$

Then N divides $(x^2 - y^2) = (x + y) \cdot (x - y)$, but N does not divide neither $x + y$ nor $x - y$. Hence, $d = \gcd(x - y, N)$ is a proper divisor of N . Finding $x, y \in \mathbb{Z}$ which satisfy (4.1) and (4.2) therefore means factoring N . The different algorithms described in this chapter use different ways to find two such integers x, y . Before we describe these methods, we prove the existence of such integers x, y .

4.1. Proposition *If $N \in \mathbb{N}_{>1}$ is odd, no perfect power and composite, then there always exist $x, y \in \mathbb{Z}$ satisfying (4.1) and (4.2).*

Proof: Let $N = N_1 \cdot N_2$, where $N_1, N_2 \in \mathbb{N}_{>1}$ are coprime. Pick $y \in \mathbb{Z}$ with $\gcd(y, N) = 1$ and choose $x \in \mathbb{Z}$ with $x \equiv y \pmod{N_1}$ and $x \equiv -y \pmod{N_2}$. Then obviously $x^2 \equiv y^2 \pmod{\text{lcm}(N_1, N_2) = N}$, but we will show that $x \not\equiv \pm y \pmod{N}$: Suppose that $x \equiv y \pmod{N}$, then $-y \equiv y \pmod{N_2}$ and so $2 \cdot y \equiv 0 \pmod{N_2}$. Since $\gcd(y, N_2) = 1$, this means that $N_2 = 2$, but N was odd, a contradiction. Similarly we get a contradiction for the situation $x \equiv -y \pmod{N}$.

4.1. Remark *The proof of Proposition 4.1 shows that the number of pairs $(x, y) \in \{1, \dots, N-1\}^2$ satisfying (4.1) and (4.2) is at least $\varphi(N)$.*

Thus there remains the question how we can find two such integers x, y . We illustrate the main idea for finding such integers in the following example.

4.1. Example *Let $N = 4633$. Assume that we know the following congruences:*

$$\begin{aligned} 67^2 &\equiv -144 \equiv -2^4 \cdot 3^2 \pmod{N}, \\ 68^2 &\equiv -9 \equiv -3^2 \pmod{N}. \end{aligned}$$

Multiplying these congruences we find that

$$(67 \cdot 68)^2 \equiv (2^2 \cdot 3^2)^2 \pmod{N},$$

i.e. $77^2 \equiv 36^2 \pmod{N}$. Since $77 \not\equiv \pm 36 \pmod{N}$, we find the factor $41 = \gcd(77-36, 4633)$ of N .

This idea can be formalized as follows: Choose a factor base

$$\mathcal{F} = \{p_0, p_1, \dots, p_l\},$$

where the p_i are pairwise distinct prime numbers, except that $p_0 = -1$ is allowed. In Example 4.1 we have chosen $\mathcal{F} = \{-1, 2, 3\}$. Determine congruences

$$x_i^2 \equiv z_i \pmod{N}, \quad 1 \leq i \leq k, \quad (4.3)$$

such that the integers z_i factor completely over \mathcal{F} as

$$z_i = \prod_{j=0}^l p_j^{e_{ji}} \quad \text{with } e_{ji} \in \mathbb{Z}_{\geq 0}. \quad (4.4)$$

One such congruence is called **relation**. Then we try to find a subset of all found relations such that the subproduct of all these z_i is a perfect square, i.e. find $f_i \in \{0, 1\}$ such that

$$z = \prod_{i=1}^k z_i^{f_i} = y^2 \quad \text{for some } y \in \mathbb{Z}. \quad (4.5)$$

The exponents f_i work as switch: If $f_i = 1$, then the i -th relation is used in the product, otherwise it is not used. If we set

$$x = \prod_{i=1}^k x_i^{f_i},$$

then $x^2 \equiv y^2 \pmod{N}$ and – if we are lucky – also $x \not\equiv \pm y \pmod{N}$. In this case we have found a factorization of N , otherwise we try to find different values for f_i .

In order to practically use this method, we have to discuss how to produce relations (solutions of (4.3), (4.4)) and how to find exponents f_i such that (4.5) holds.

Let us start with the latter question. First observe that z is a perfect square if and only if all the exponents in the prime decomposition of z are even. Since we know the prime

decomposition of all factors z_i of z , we can easily determine the prime decomposition of z as

$$z = \prod_{i=1}^k z_i^{f_i} = \prod_{i=1}^k \left(\prod_{j=0}^l p_j^{e_{ji}} \right)^{f_i} = \prod_{j=0}^l p_j^{\sum_{i=1}^k f_i \cdot e_{ji}}.$$

z is a perfect square if

$$\sum_{i=1}^k e_{ji} \cdot f_i \equiv 0 \pmod{2}$$

for all $0 \leq j \leq l$. This equation describes an algorithm for finding the f_i . For each relation, we compute an exponent vector $\underline{e}_i = (e_{i0}, \dots, e_{il}) \pmod{2}$. As soon as we have found more than $l + 1$ such vectors (i.e. more than $l + 1$ relations), there must be a linear dependency

$$\sum_{i=1}^k f_i \cdot \underline{e}_i \equiv \underline{0} \pmod{2}$$

and thus, the f_i can be determined by solving a linear system over \mathbb{F}_2 , the field of two elements. The linear systems which have to be solved are huge, but they have a special structure: most of the entries in an exponent vector are 0. We will describe special algorithms for solving such linear systems in Chapter 7.

There remains the question how to find enough relations. The first simplest idea for producing relations works as follows: Choose $x \in \{\lceil \sqrt{N} \rceil, \dots, N - 1\}$ at random and set $z = x^2 \pmod{N}$. Then we try whether z decomposes over the given factor basis \mathcal{F} . If we are successful, we have found a relation, otherwise we choose another x . This algorithm is called **Dixon's random square method**. Unfortunately, this strategy is not practical because the numbers z that have to be factored over \mathcal{F} are quite large (order of magnitude N). The first algorithm which could produce smaller integers z in a systematic way was the continued fraction algorithm which we describe in the next section.

4.2 Factoring Algorithms Using Continued Fractions

There exist two factoring algorithms which use special properties of continued fractions. We briefly introduce some basic facts about continued fractions. Instead of representing a real number x by its decimal expansion it can also be written as an (infinite) regular continued fraction. A continued fraction expansion for $x \in \mathbb{R}$ is a sequence of rational integers $b_0 \in \mathbb{Z}$, $b_i \in \mathbb{Z}_{\geq 1}$ for $i \geq 1$ such that

$$x = b_0 + \frac{1}{b_1 + \frac{1}{b_2 + \frac{1}{b_3 + \frac{1}{\ddots}}}} \quad (4.6)$$

As abbreviation we also write $x = [b_0, b_1, b_2, \dots]$. b_0 and the **partial denominators** b_i can be calculated via the following recursion (set $x_0 = x$)

$$b_i = \lfloor x_i \rfloor, \quad \text{and, if } x_i \neq b_i, \quad x_{i+1} = \frac{1}{x_i - b_i}. \quad (4.7)$$

4.2. Example Let $x = \sqrt{2}$. Then $x_0 = \sqrt{2}$ and $b_0 = 1$. Using (4.7), we get $x_1 = 1/(\sqrt{2} - 1) = \sqrt{2} + 1$ and so $b_2 = 2$. For $i \geq 1$ we then get

$$x_i = \sqrt{2} + 1 \quad \text{and} \quad b_i = 2.$$

Therefore we have the continued fraction expansion $\sqrt{2} = [1, 2, 2, \dots]$.

Note that the continued fraction expansion of $\sqrt{2}$ is periodic with period length 1. In general, the continued fraction expansion of x is periodic if and only if x is a quadratic irrationality, i.e. x is a root of a quadratic polynomial with integer coefficients, a fact, which we do not prove here.

The n -th **convergent** or the n -th **partial quotient** of a continued fraction expansion $[b_0, b_1, \dots]$ is the finite part

$$\frac{\mathcal{A}_n}{\mathcal{B}_n} = [b_0, b_1, \dots, b_n].$$

4.2. Proposition If we define $\mathcal{A}_{-1} = 1$, $\mathcal{B}_{-1} = 0$ and $\mathcal{A}_0 = b_0$, $\mathcal{B}_0 = 1$, then for $n \in \mathbb{Z}_{>0}$

$$\begin{aligned} \mathcal{A}_n &= b_n \cdot \mathcal{A}_{n-1} + \mathcal{A}_{n-2}, \\ \mathcal{B}_n &= b_n \cdot \mathcal{B}_{n-1} + \mathcal{B}_{n-2}. \end{aligned}$$

We also mention without further discussion that the sequence of convergents of x is in fact convergent with limit x .

4.3. Proposition For $n \geq 0$ we have $\mathcal{A}_{n-1} \cdot \mathcal{B}_n - \mathcal{A}_n \cdot \mathcal{B}_{n-1} = (-1)^n$.

4.4. Proposition For $n \geq 1$, we have

$$x = \frac{\mathcal{A}_{n-2} + \mathcal{A}_{n-1} \cdot x_n}{\mathcal{B}_{n-2} + \mathcal{B}_{n-1} \cdot x_n}.$$

Proof: We examine the function $R(t) = [b_0, b_1, \dots, b_{n-1}, t]$ for $t > 0$. It is easy to see by the definition of partial convergents that there exist integers a, b, c, d such that $R(t) = \frac{a+bt}{c+dt}$ (proof per induction on n). If we consider the definition of partial quotients, we directly see that

$$\lim_{t \rightarrow \infty} R(t) = \frac{\mathcal{A}_{n-1}}{\mathcal{B}_{n-1}}.$$

In order to evaluate $\lim_{t \rightarrow 0} R(t)$, we note that

$$\lim_{t \rightarrow 0} \frac{1}{b_{n-1} + \frac{1}{t}} = 0,$$

so we can use the same argument to achieve $\lim_{t \rightarrow 0} R(t) = \mathcal{A}_{n-2}/\mathcal{B}_{n-2}$. Hence

$$R(t) = \frac{\mathcal{A}_{n-2} + k \cdot \mathcal{A}_{n-1} \cdot t}{\mathcal{B}_{n-2} + k \cdot \mathcal{B}_{n-1} \cdot t}$$

for some $k \neq 0$. But since

$$R(b_n) = \frac{\mathcal{A}_n}{\mathcal{B}_n} = \frac{\mathcal{A}_{n-2} + k \cdot \mathcal{A}_{n-1} \cdot b_n}{\mathcal{B}_{n-2} + k \cdot \mathcal{B}_{n-1} \cdot b_n},$$

it follows from Proposition 4.2 that $k = 1$. Finally, taking $t = x_n$, proves the assertion (note that by construction that $x = [b_0, b_1, \dots, x_n]$).

For the purpose of factoring N , the regular continued fraction expansion of \sqrt{N} is employed. This expansion is examined in the next proposition.

4.5. Proposition *If $x = \sqrt{N}$, then we can write $x_n = \frac{\sqrt{N} + P_n}{Q_n}$ with $P_n, Q_n \in \mathbb{Z}$ and $P_0 = 0, Q_0 = 1$ and*

$$P_{n+1} = b_n \cdot Q_n - P_n, \quad Q_{n+1} = \frac{N - P_{n+1}^2}{Q_n}.$$

Moreover, we have the estimates $|P_n| < \sqrt{N}$ and $0 < Q_n < 2\sqrt{N}$.

Proof: The proof employs mathematical induction. For $n = 0$, we have $x_0 = \sqrt{N} = (\sqrt{N} + P_0)/Q_0$ and

$$x_1 = \frac{1}{\sqrt{N} - b_0} = \frac{\sqrt{N} + b_0}{N - b_0^2},$$

so P_1 and Q_1 satisfy the condition. Assume therefore that the claim of the proposition is proven for all $n \leq k$. We get

$$\begin{aligned} x_{k+1} &= \frac{1}{x_k - b_k} = \frac{1}{\frac{\sqrt{N} + P_k}{Q_k} - b_k} \\ &= \frac{Q_k}{\sqrt{N} + P_k - b_k \cdot Q_k} = \frac{Q_k}{\sqrt{N} - P_{k+1}} \\ &= \frac{Q_k \cdot (\sqrt{N} + P_{k+1})}{N - P_{k+1}^2}. \end{aligned}$$

We must now prove that $Q_k \mid (N - P_{k+1}^2)$. Since $Q_k = (N - P_k^2)/Q_{k-1} \in \mathbb{Z}$, we have

$$\frac{N - P_{k+1}^2}{Q_k} = \frac{N - (b_k \cdot Q_k - P_k)^2}{Q_k} \in \mathbb{Z}.$$

Finally, we prove the estimates for the size of P_n and Q_n : For $n = 0$ the assertion is true. Suppose that the assertion is true for $n \leq k$. We note that $x_k - b_k = (\sqrt{N} - P_{k+1})/Q_k$. Hence

$$0 < \frac{\sqrt{N} - P_{k+1}}{Q_k} < 1 \quad \text{or} \quad 0 < \sqrt{N} - P_{k+1} < Q_k < 2\sqrt{N}.$$

This implies that $|P_{k+1}| < \sqrt{N}$. Now $0 < \frac{1}{x_{k+1}} = \frac{Q_{k+1}}{\sqrt{N} + P_{k+1}} < 1$, which shows that

$$0 < Q_{k+1} < \sqrt{N} + P_{k+1} < 2\sqrt{N}.$$

Note that it follows from Proposition 4.5 that the continued fraction expansion of \sqrt{N} is periodic. The essential result which links the continued fraction expansion of \sqrt{N} with the factoring problem for N is shown in the next proposition.

4.6. Proposition *If $x = \sqrt{N}$, P_n, Q_n as in Proposition 4.5, $\mathcal{A}_n, \mathcal{B}_n$ as in Proposition 4.2, then for all $n \geq 0$*

$$\mathcal{A}_{n-1}^2 - N \cdot \mathcal{B}_{n-1}^2 = (-1)^n \cdot Q_n .$$

Proof: By Proposition 4.4, we have $x = \frac{\mathcal{A}_{n-2} + \mathcal{A}_{n-1} \cdot x_n}{\mathcal{B}_{n-2} + \mathcal{B}_{n-1} \cdot x_n}$ for all $n \geq 1$. Since by Proposition 4.5 $x_n = \frac{\sqrt{N} + P_n}{Q_n}$, we find

$$\sqrt{N} = \frac{Q_n \cdot \mathcal{A}_{n-2} + P_n \cdot \mathcal{A}_{n-1} + \mathcal{A}_{n-1} \cdot \sqrt{N}}{Q_n \cdot \mathcal{B}_{n-2} + P_n \cdot \mathcal{B}_{n-1} + \mathcal{B}_{n-1} \cdot \sqrt{N}} .$$

Hence

$$\sqrt{N} \cdot (Q_n \cdot \mathcal{B}_{n-2} + P_n \cdot \mathcal{B}_{n-1}) + N \cdot \mathcal{B}_{n-1} = Q_n \cdot \mathcal{A}_{n-2} + P_n \cdot \mathcal{A}_{n-1} + \mathcal{A}_{n-1} \cdot \sqrt{N} .$$

Comparing coefficients yields the two equations

$$\begin{aligned} Q_n \cdot \mathcal{B}_{n-2} + P_n \cdot \mathcal{B}_{n-1} &= \mathcal{A}_{n-1} , \\ Q_n \cdot \mathcal{A}_{n-2} + P_n \cdot \mathcal{A}_{n-1} &= N \cdot \mathcal{B}_{n-1} . \end{aligned}$$

Eliminating P_n in these equation yields

$$Q_n \cdot (\mathcal{A}_{n-1} \cdot \mathcal{B}_{n-2} - \mathcal{A}_{n-2} \cdot \mathcal{B}_{n-1}) = \mathcal{A}_{n-1}^2 - N \cdot \mathcal{B}_{n-1}^2 .$$

The assertion follows then from Proposition 4.3.

If we consider the equation of Proposition 4.6 modulo N , then we have

$$\mathcal{A}_{n-1}^2 \equiv (-1)^n \cdot Q_n \pmod{N} .$$

4.1. Exercises

1. Prove Proposition 4.2 and 4.3.
2. Find an infinite family of non square numbers $N \in \mathbb{Z}_{\geq 1}$ such that the continued fraction expansion of \sqrt{N} has period length 1.

4.2.1 Shanks' SQUFOF Algorithm

Shanks' SQUFOF algorithm is directly based on this equation. Assume that you find for an even index n that Q_n is a square in \mathbb{Z} . Then we have found the equivalence $\mathcal{A}_{n-1}^2 \equiv R^2 \pmod{N}$ and hopefully $\gcd(\mathcal{A}_{n-1} - R, N)$ is a proper factor of N . Therefore the SQUFOF algorithm uses the equations given in Proposition 4.4 and 4.5 to successively compute the integers \mathcal{A}_{n-1} and Q_n . For even n , we test whether Q_n is a square. Since computing square roots is a relatively expensive operation, it is advisable to check whether Q_n is a square modulo several small primes. Only if Q_n survives all these tests, we should try to extract a square root in \mathbb{Z} .

It might happen that the period length of the expansion of \sqrt{N} is very small. In this case, one instead considers the continued fraction expansion of $\sqrt{f \cdot N}$ for a suitable small integer $f \in \mathbb{Z}_{\geq 1}$ (a so called multiplier).

The original description of Shanks is different to the description given above. Shanks used the theory of binary quadratic forms and reduction theory for such forms to show that knowledge of a special reduced form might lead to a factor of N . His approach explains the name of the method, since SQUFOF stands for Square Form Factorization Method. There is a strong connection between reduction of binary quadratic form and continued fractions, but we don't want to go into it here.

In practice, the SQUFOF method is of some importance for factoring integers with up to 20 decimal digits. Moreover, it is the method of choice as a subroutine in the quadratic sieve algorithm and the Number field sieve, as we will see later.

4.2.2 The CFRAC Algorithm

The **Continued Fraction Method** was first published in [LePo31], but it only became of greater importance after the practical use of computers. In 1970, Morrison and Brillhart were able to factor the seventh Fermat number with the CFRAC method (for a description of their implementation and several tricks, see [MoBr75]). Again the equation

$$A_{n-1}^2 \equiv (-1)^n \cdot Q_n \pmod{N}$$

is used. By Proposition 4.5, we have $0 < Q_n < 2\sqrt{N}$. The continued fraction expansion of \sqrt{N} therefore yields solutions of (4.3) with small $|z_n|$. Here we try to decompose Q_n over the chosen factor basis. If we succeed, then we have found a relation.

The amount of computing time for computing these elements z_n is comparatively small compared to the time for (hopefully) decomposing z_n over the factor base F .

It might however happen that the period length of the expansion of \sqrt{N} is very small. In this case, one instead considers the continued fraction expansion of $\sqrt{f \cdot N}$ for a suitable small integer $f \in \mathbb{Z}_{\geq 1}$. In this way, Brillhart and Morrison were able in 1970 to factor the 7-th Fermat number $F_7 = 2^{2^7} + 1$. They calculated 1330000 Q_n 's in the continued fraction expansion of $\sqrt{257 \cdot F_7}$, decomposing 2059 Q_n 's over the factor basis.

Though the quadratic residues produced by the continued fraction method are quite small, trial dividing them takes a long time. It might be that lot's of Q_n do not decompose over the chosen factor basis and lot's of work is done without getting a relation. Therefore the continued fraction method is limited to factoring numbers with up to 50 digits.

This problem of the continued fraction algorithm was overcome with another algorithm, the so called **quadratic sieve algorithm**. The main advantage of the quadratic sieve algorithm over CFRAC is the improvement that it uses a sieve to find candidates z_n which indeed decompose over the factor basis. Since sieving is much cheaper than trial dividing lot of numbers, this trick let to a big practical improvement, as we will see in Section 5.6. Additionally, the size of the elements z_n is approximately the same as the size of the z_n generated in the CFRAC algorithm.

Chapter 5

The Quadratic Sieve Algorithm

The Quadratic Sieve Algorithm (QS) for factoring integers was first described by Carl Pomerance [Po85]. This algorithm has been intensively used for factoring and lot's of practical improvements were found (see for example [De93]). Using this method, numbers with more than 100 digits could be factored (see Section 5.6).

Again, the integer to be factored is called N , it is assumed to be odd, composite and no perfect power. We calculate $m = \lfloor \sqrt{N} \rfloor$, and we define the polynomial

$$z(X) = (X + m)^2 - N \in \mathbb{Z}[X]. \quad (5.1)$$

Note that for any $x \in \mathbb{Z}$

$$\begin{aligned} |z(x)| &\leq (|x| + \sqrt{N})^2 - N \\ &= |x|^2 + 2|x| \cdot \sqrt{N} + N - N \\ &= |x| \cdot (|x| + 2\sqrt{N}), \end{aligned}$$

i.e. for absolutely small integers x we have $|z(x)| \approx 2\sqrt{N}$. Moreover, we have

$$z(x) \equiv (x + m)^2 \pmod{N}.$$

Hence $z(x)$ is a quadratic residue modulo N which is not too large. We therefore try to factor $z(x)$ over the chosen factor base

$$\mathcal{F} = \{p_0, p_1, \dots, p_l\}.$$

Suppose that we know for some integer x that the factor basis element $p \in \mathcal{F}$ divides $z(x)$. Hence, x is a zero of the quadratic polynomial $z(X)$ in the finite field $\mathbb{Z}/p\mathbb{Z}$.

The quadratic polynomial $z(X) \pmod{p}$ has at most two zeros modulo p . Assume that we know the zeros x_p and x'_p of $z(x)$ modulo p . Both x_p, x'_p can be found in the interval $\{0, 1, \dots, p-1\}$. All the other zeros of $z(X)$ modulo p are of the form

$$x = x_p + k \cdot p, \quad x' = x'_p + k \cdot p, \quad k \in \mathbb{Z}. \quad (5.2)$$

So we do not have to trial divide all the polynomial values $z(x)$ by p but we can predict by (5.2) for which arguments x the value $z(x)$ is divisible by p .

5.1. Example Let $N = 323$ and $\mathcal{F} = \{2, 3, 5, 7\}$. Then we have $m = \lfloor \sqrt{323} \rfloor = 17$ and the polynomial is given as $z(X) = (X + 17)^2 - 323 = X^2 + 34X - 34$. We try to find the roots of $z(X)$ modulo some of the primes in the factor basis \mathcal{F} :

$p = 2$: We know $z(X) \equiv X^2 \pmod{2}$. There is precisely one zero modulo 2, namely $x_2 = 0$.

All the other places x , where $z(x)$ is divisible by 2, are therefore given as $x = k \cdot 2$.

$p = 3$: We have $z(X) \equiv (X - 1)^2 + 1 \pmod{3}$. To solve $z(X) \equiv 0 \pmod{3}$, we must solve

$$(X - 1)^2 \equiv -1 \pmod{3},$$

i.e. we must determine whether -1 is a square modulo 3. Since the only squares modulo 3 are 0 and 1, it follows that $z(X)$ is never divisible by 3 and we may as well remove 3 from the factor base \mathcal{F} .

$p = 7$: Here we have $z(X) \equiv (X + 3)^2 - 1 \pmod{7}$. Again we transform $z(X) \equiv 0 \pmod{7}$ to

$$(X + 3)^2 \equiv 1 \pmod{7} \Leftrightarrow (X + 3) \equiv \pm 1 \pmod{7}.$$

We get the solutions $x_7 = 3$ and $x'_7 = 5$. Therefore all the places x , where the value $z(x)$ is divisible by 7, are given as either $x = 3 + k \cdot 7$ or $x' = 5 + k \cdot 7$ for $k \in \mathbb{Z}$.

The previous example has shown how we can find all places x where a prime of the factor basis divides the value $z(x)$. We can use all these information in a sieving procedure. In this way, we find all the places x in some sieving interval, where $z(x)$ completely decomposes over the chosen factor basis \mathcal{F} . We illustrate the sieving procedure with an example.

5.2. Example In order to use the information, which we found in Example 5.1, we write down a one-dimensional array of values of $z(x)$ for $x = 1, \dots, 13$:

x	1	2	3	4	5	6	7	8	9	10	11	12	13
$z(x)$	1	38	77	118	161	206	253	302	353	406	461	518	577

We already know that $z(0)$ is divisible by 2 and that we have to increment $x_2 = 0$ by multiples of 2 to find the other values of $z(x)$, which are divisible by 2. We divide the values $z(x)$ by 2 for all $x = k \cdot 2$ and get (note that values $z(x)$ in circles remain unchanged):

$$z(x)/2 \quad \textcircled{1} \quad 19 \quad \textcircled{77} \quad 59 \quad \textcircled{161} \quad 103 \quad \textcircled{253} \quad 151 \quad \textcircled{353} \quad 203 \quad \textcircled{461} \quad 259 \quad \textcircled{577}$$

The same procedure is applied to $p = 7$. Here we have two starting values for our sieve, $x = 3$ and $x' = 5$. Again, we divide the values $z(x)$ for $x = 3 + k \cdot 7$ and $x = 5 + k \cdot 7$ by 7:

$$z(x)/7 \quad \textcircled{1} \quad \textcircled{19} \quad 11 \quad \textcircled{59} \quad 23 \quad \textcircled{103} \quad \textcircled{253} \quad \textcircled{151} \quad \textcircled{353} \quad 29 \quad \textcircled{461} \quad 37 \quad \textcircled{577}$$

If we add 11 to our factor base, then we see that $z(3)$ can be completely factored over \mathcal{F} : $z(3) = 77 = 7 \cdot 11$, and we have found a relation, namely $z(3) = 7 \cdot 11 \equiv 20^2 \pmod{323}$.

The following sieving procedure can be used to find all places x in a **sieving interval** $\{-\mathcal{M}, \dots, \mathcal{M}\}$ where the polynomial value $z(x)$ completely decomposes over the chosen factor basis \mathcal{F} .

14. Algorithm (Sieving in QS, basic version)

INPUT: $\mathcal{M} \in \mathbb{N}$, polynomial $z(X) \in \mathbb{Z}[X]$, factor basis \mathcal{F} .
 OUTPUT: set $\{|x| \leq \mathcal{M}; z(x) \text{ decomposes completely over } \mathcal{F}\}$.

```

(1) compute and store  $z(x)$  for every  $x \in \{-\mathcal{M}, \dots, \mathcal{M}\}$ .
(2) for (every prime number  $p \in \mathcal{F}$ ) do
(3)   compute  $x_p, x'_p \in \{0, \dots, p-1\}$  with  $z(x_p) = 0 \pmod p$ ,  $z(x'_p) = 0 \pmod p$ .
(4)   if (there is no such number  $x$ ) then
(5)     remove  $p$  from  $\mathcal{F}$ 
(6)     continue
(7)   fi
(8)   for (every  $x = x_p + k \cdot p$  with  $|x| \leq \mathcal{M}$ ) do
(9)     replace  $z(x)$  by  $z(x)/p$ .
(10)  od
(11)  if ( $x'_p \neq x_p$ ) then
(12)    for (every  $x = x'_p + k \cdot p$  with  $|x| \leq \mathcal{M}$ ) do
(13)      replace  $z(x)$  by  $z(x)/p$ .
(14)    od
(15)  fi
(16) od
(17) return (all  $x \in \{-\mathcal{M}, \dots, \mathcal{M}\}$  with  $z(x) = 1$ )

```

We note that this procedure only yields those numbers x in the sieving interval for which $z(x)$ is a product of pairwise distinct prime numbers in \mathcal{F} . It can however be generalized to prime powers. If we have obtained $z(x) = 1$ for some x , then we can easily reconstruct the decomposition of $z(x)$ over \mathcal{F} by trial division.

5.1. Exercise *Generalize the sieving procedure 14 to prime powers. (Hint: Assume we know x_k with $x_k^2 \equiv N \pmod{p^k}$. In order to solve $x_{k+1}^2 \equiv N \pmod{p^{k+1}}$, we set $x_{k+1} = x_k + c \cdot p^k$. Try to find an equation to compute c .)*

Practical Considerations in Sieving

In the sieving procedure 14, we find the places x in the sieving interval where the value $z(x)$ completely decomposes over the factor basis \mathcal{F} . This sieving procedure is much cheaper than searching these places with trial division (note that in the QS we do trial division only at places x where we know that $z(x)$ really decomposes). Nevertheless there are quite

a few divisions in Algorithm 14. In the **logarithm variant** of Algorithm 14, we replace $z(x)$ by an approximation to $\log z(x)$. Then the divisions in the lines (8) and (11) can be replaced by subtraction of approximations for $\log p$.

Since we work with approximations, we will make small errors, such that we do not find places x with $\log(z(x)) = 0$. Nevertheless, we can test for all places x where $\log(z(x))$ is “approximately zero” whether $z(x)$ really decomposes over \mathcal{F} . This trick leads to two more improvements: Since we expect that only small primes in the factor basis will divide $z(x)$ as a perfect power, we find these prime powers “with high probability” by choosing the final test approximation of 0 “slightly larger than usual”. Moreover practical experience shows that it is not worthwhile to use very small primes in the sieving procedure. Logarithms of such primes are small, but there are lot’s of places x in the sieving interval where these primes “match” (therefore lot’s of subtractions and memory operations). Again an appropriate change of the final test approximation can lead to a running time improvement ([De93] gives an example where starting the sieving step with the 50-th prime leads to a running time reduction of approximately 28 %). For a description of all these tricks and the choice of approximations, we refer to [De93].

5.1 Finding the Starting Points for Sieving

The main problem that has to be dealt with now is: How can we decide whether $z(X) \equiv 0 \pmod{p}$ is solvable and, in the case of solvability, how can we determine the absolute smallest solutions. Since $z(X) = (X+m)^2 - N$, checking the solvability of $z(X) \equiv 0 \pmod{p}$ for a prime number p means checking whether N is a square modulo p .

5.1.1 Testing whether N is a Square modulo a Prime

5.1. Definition For $a \in \mathbb{Z}$ and a prime p , we define the **Legendre symbol**

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } \gcd(a, p) = 1 \text{ and } a \text{ is a square mod } p, \\ -1 & \text{if } \gcd(a, p) = 1 \text{ and } a \text{ is not a square mod } p, \\ 0 & \text{if } p \mid a. \end{cases}$$

If $\left(\frac{a}{p}\right) = 1$, then a is called **quadratic residue modulo p** and if $\left(\frac{a}{p}\right) = -1$, a is called **quadratic non residue modulo p** .

Therefore we have to compute the Legendre symbol $\left(\frac{N}{p}\right)$ to decide whether N is a square modulo p . The following Lemma induces an algorithm to compute a Legendre Symbol. In practice, this algorithm is not the best possible algorithm, but in Exercises 5.1 we will explain the idea for better algorithms.

5.1. Lemma Let p be an odd prime and $a \in \mathbb{Z}$. Then we have

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Proof: Note that $(\mathbb{Z}/p\mathbb{Z})^*$ is a cyclic group of order $p-1$. Let ζ be a primitive element, i.e. a generator of $(\mathbb{Z}/p\mathbb{Z})^*$, and let $a = \zeta^k$ with $0 \leq k \leq p-1$. The claim of the lemma follows from the observation that the exponent k is even if and only if a is a square.

5.1. Exercises The **Jacobi Symbol** $\left(\frac{a}{n}\right)$ is a generalization of the Legendre symbol to non prime integers n . If $n = \prod_{i=1}^s p_i^{e_i}$ is the prime factorization of n , then

$$\left(\frac{a}{n}\right) = \prod_{i=1}^s \left(\frac{a}{p_i}\right)^{e_i},$$

where $\left(\frac{a}{p_i}\right)$ are Legendre symbols. The Jacobi Symbol satisfies the following rules (let n, m be odd integers):

$$\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}, \quad \left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8}, \quad \left(\frac{m}{n}\right) = (-1)^{(n-1)(m-1)/4} \cdot \left(\frac{n}{m}\right).$$

1. Show how these equations can be used to find an algorithm which uses divisions with remainder to evaluate a Legendre symbol (Hint: Note that $\left(\frac{a}{n}\right) = \left(\frac{a \bmod n}{n}\right)$ and use the third equality).
2. Describe a binary algorithm for computing Legendre symbols (again there exist two variants of binary algorithms: a left-to-right or a right-to-left algorithm).

5.1.2 Extracting a Square Root modulo a Prime

Now we know how we can check whether the polynomial $z(X) = (X + m)^2 - N \bmod p$ has a root in $\mathbb{Z}/p\mathbb{Z}$ or not. Assume in the following that there exists a square root of N , i.e. an element $\lambda \in \mathbb{Z}/p\mathbb{Z}$ with $\lambda^2 \equiv N \bmod p$. Then the two roots of $z(X) \bmod p$ are given as

$$x_p = -m + \lambda \bmod p \quad \text{and} \quad x'_p = -m - \lambda \bmod p.$$

Therefore we can find the starting points for the sieve algorithm 14 if we can extract square roots modulo p . Assume that we want to extract a square root for a quadratic residue $a \in (\mathbb{Z}/p\mathbb{Z})^*$. Note that if we know one square root λ of a , the second one is directly given as $-\lambda$. First we mention the easy case for extracting square roots:

5.1. Proposition *If $p \equiv 3 \bmod 4$, then $a^{(p+1)/4} \bmod p$ is a square root of $a \bmod p$.*

Proof: Easy computation and Lemma 5.1:

$$\left(a^{(p+1)/4}\right)^2 \equiv a^{(p+1)/2} \equiv a^{(p-1)/2} \cdot a \bmod p \equiv a \bmod p.$$

This case is a special part of an algorithm due to Shanks (see [Sh72]) for extracting square roots modulo primes. We will explain this algorithm in detail. Let p be a square and $a \in (\mathbb{Z}/p\mathbb{Z})^*$ be a quadratic residue. We want to calculate a square root λ of a .

The idea of Shanks is to calculate a sequence (λ_i, κ_i) such that

$$\lambda_i^2 \equiv a \cdot \kappa_i \bmod p, \tag{5.3}$$

and the order of $\kappa_i \bmod p$ is strictly decreasing. Then we will eventually have $\kappa_i \equiv 1 \bmod p$ and $\lambda_i^2 \equiv a \bmod p$ as desired.

For this purpose we compute a quadratic non residue $z \in (\mathbb{Z}/p\mathbb{Z})^*$ by trial. Since half of the primitive residues mod p are non squares, the probability of success is quite high. In fact, it can be shown under the assumption of the generalized Riemann hypothesis that the first quadratic non residue is of magnitude $O((\log p)^2)$.

Now we determine $s_0, k \in \mathbb{Z}_{\geq 0}$ such that

$$p - 1 = 2^{s_0} \cdot (2k + 1).$$

Then the order of $(\mathbb{Z}/p\mathbb{Z})^*$ is $2^{s_0} \cdot (2k + 1)$. The subgroup \mathcal{G}_1 of elements of two power order is of order 2^{s_0} and the subgroup \mathcal{G}_2 of elements of odd order is of order $2k + 1$. Both these groups are cyclic, because $(\mathbb{Z}/p\mathbb{Z})^*$ is cyclic.

We try to hold the following invariants in the main iteration of the algorithm: We know an element $c_i \in (\mathbb{Z}/p\mathbb{Z})^*$ of order 2^{s_i} and an integer $t_i > 0$ such that

$$\kappa_i \equiv c_i^{2^{t_i}(2m_i-1)} \pmod{p}$$

for some unknown integer m_i . Moreover, equation (5.3) is valid. The sequence of the orders of c_i is decreasing, such that for some index s we have $c_s = \kappa_s = 1 \pmod{p}$ and the algorithm terminates with output λ_s . Note that by (5.3) λ_s indeed is a square root of a modulo p .

We initialize the iteration by setting

$$\lambda_0 = a^{k+1} \pmod{p}, \quad \kappa_0 = a^{2k+1} \pmod{p}.$$

Then (5.3) is easily verified. Additionally, we set $c_0 = z^{2k+1} \pmod{p}$. Then the order of c_0 is even and, because z is a quadratic non residue, c_0 generates the subgroup \mathcal{G}_1 of two power order, i.e. the order of c_0 is exactly 2^{s_0} . It is easily verified that the order of κ_0 is a power of two and that therefore $\kappa_0 \in \mathcal{G}_1$. Note that $t_0 > 0$ since a is a square modulo p . We can determine t_0 if we compute the order of κ_0 by repeated squaring. Then we set $t_0 = s_0 - l$ (where l is the number of squarings we need until we have found the order of κ_0). With these initialization, all parts of the chosen invariant are satisfied.

Assume that we know $\lambda_i, \kappa_i, c_i, s_i$ and t_i . It was Shanks' idea to set

$$\lambda_{i+1} = \lambda_i \cdot c_i^{2^{t_i-1}} \pmod{p}.$$

With this choice, we have

$$\lambda_{i+1}^2 \equiv \lambda_i^2 \cdot c_i^{2^{t_i}} \equiv a \cdot \kappa_i \cdot c_i^{2^{t_i}} \pmod{p},$$

such that $\kappa_{i+1} = \kappa_i \cdot c_i^{2^{t_i}} \pmod{p}$ is necessary to satisfy (5.3). We evaluate κ_{i+1} as

$$\kappa_{i+1} = c_i^{2^{t_i} \cdot (2m_i-1)} \cdot c_i^{2^{t_i}} = c_i^{2^{t_i} 2m_i} \pmod{p}.$$

Therefore we can set

$$c_{i+1} = c_i^{2^{t_i}} \pmod{p} \quad \text{and} \quad s_{i+1} = s_i - t_i.$$

Since $t_i > 0$, s_{i+1} is smaller than s_i , and the order of the c_i -sequence is decreasing. Finally, we compute t_{i+1} with exactly the same algorithm as explained above. If $\kappa_{i+1} \not\equiv 1 \pmod{p}$, then $t_{i+1} > 0$ is also satisfied.

This description leads to the following algorithm for extracting square roots modulo prime numbers.

15. Algorithm (Computing Square Roots modulo a Prime)

INPUT: square $a \in (\mathbb{Z}/p\mathbb{Z})^*$.
 OUTPUT: a square root $\lambda \in (\mathbb{Z}/p\mathbb{Z})^*$ of a .

```

(1)  if ( $p = 3 \text{ Mod } 4$ ) then
(2)    return ( $a^{(p+1)/4} \text{ Mod } p$ )
(3)  fi
(4)  Compute  $s, k \in \mathbb{N}$  such that  $p - 1 = 2^s (2k + 1)$ .
(5)  repeat
(6)    Choose  $z \in (\mathbb{Z}/p\mathbb{Z})^*$  at random.
(7)  until ( $z$  is a quadratic non residue modulo  $p$ .)
(8)  Set  $\lambda = a^{k+1} \text{ Mod } p$ ,  $\kappa = a^{2k+1} \text{ Mod } p$  and  $c = z^{2k+1} \text{ Mod } p$ .
(9)  Compute order  $2^l$  of  $\kappa \text{ Mod } p$  and set  $t = s - l$ .
(10) while ( $t > 0$ ) do
(11)   Set  $\lambda = \lambda \cdot c^{2^t-1} \text{ Mod } p$  and  $\kappa = \kappa \cdot c^{2^t} \text{ Mod } p$ .
(12)   Set  $s = s - t$ .
(13)   Compute order  $2^l$  of  $\kappa$  and set  $t = s - l$ .
(14) od
(15) return ( $\lambda$ )

```

One should mention that this algorithm only works if we take a square root modulo a prime. In fact, it is known that extracting square roots modulo composite numbers which are no prime powers is equivalent to factoring the modulus. Note that the algorithm we have described here can be generalized to extracting m -th roots for $m > 2$. There exist several other algorithms for computing square roots, for an overview on these algorithms, see [BaSh96].

5.2 Computing the Factor Basis

We have already mentioned that the factor basis \mathcal{F} should consist of -1 and the primes smaller than some given bound, which depends on the size of the number N we want to factor. We can compute these small primes with the sieve of Erathostenes (Algorithm 6). Moreover, we have already found another condition for factor basis elements: the polynomial $z(X)$ will only have a root modulo a prime p , if N is a square modulo p . Therefore we should only include primes p into the factor basis \mathcal{F} which satisfy this condition. The optimal size of the factor basis depends on the size of N and on the implementation. For a practical list of factor basis sizes, we refer to [De93].

However there can be a disadvantage of this fact: it might happen that N is a square for very few small prime numbers. Since it is desirable to have lot's of small primes in the

factor basis, we can “change” the number N which we want to factor a little bit. Instead of factoring N , we try to factor kN for a small non square integer k (for example, we may choose $1 \leq k \leq 100$). If we find a factor of kN , this factor perhaps leads to a factor of N (or if we are unlucky, we find a factor which divides k , but then we can restart the linear algebra step and we have another chance). We choose the **multiplier** k in such a way that kN is a square modulo lot’s of very small prime numbers.

5.3 The Multi Polynomial Quadratic Sieve (MPQS)

The basic version of the quadratic sieve algorithm can be used to factor integers of up to 50 decimal digits in reasonable time. The problem of the single polynomial version is the size of the sieving interval, which grows enormously. For finding enough relations, we have to use large sieving bounds \mathcal{M} , which lead to large values $z(x)$ if $x \approx \mathcal{M}$. The probability for these values to decompose over the factor basis decreases and finding enough relations becomes difficult and time expensive. Silverman found a solution to this problem [Si87]: He suggested to use several different polynomials with a fixed small sized sieving interval instead of one polynomial with a large sieving interval. We will explain his idea how to compute these polynomials.

Assume that

$$\begin{aligned} Q(X) &= AX^2 + BX + C \\ &= A \cdot \left(X + \frac{B}{2A}\right)^2 - \frac{B^2 - 4AC}{4A}. \end{aligned} \quad (5.4)$$

In order to make $Q(X)$ generate quadratic residues modulo N , we ask that $B^2 - 4AC = kN$ for the chosen odd, non square multiplier $k \in \mathbb{Z}$, and that A is a quadratic residue modulo N . It is advantageous to keep the absolute values of $Q(x)$ small for all $x \in \{-\mathcal{M}, \dots, \mathcal{M}\}$. Without loss of generality we assume that $A, B \geq 0$. The optimal values for the coefficients of $Q(X)$ are

$$A_{opt} = W_1 \mathcal{M} \sqrt{kN}, \quad B_{opt} = 0, \quad C_{opt} = W_2 \mathcal{M} \sqrt{kN}$$

with $W_1 \in [0.7, 0.75]$, $W_2 \in [-0.35, 0.30]$ and $W_1 W_2 = -0.25$ (see [De93]).

The idea is to choose a value for A which is close to those optimal choices: Let $A = \mathcal{D}^2$ where \mathcal{D} is a (pseudo) prime with $\left(\frac{kN}{\mathcal{D}}\right) = 1$ such that $A \approx A_{opt}$. To find the other coefficients of $Q(X)$, we compute an integer B with

$$B^2 \equiv kN \pmod{4A}.$$

We can compute such an integer B in three steps:

1. Use Algorithm 15 to compute a square root B_0 of kN modulo \mathcal{D} (note that \mathcal{D} is assumed to be a pseudo prime). If we choose $\mathcal{D} \equiv 3 \pmod{4}$, then this step is extremely easy.
2. Set $h = (kN - B_0^2)/\mathcal{D} \in \mathbb{Z}$ and compute $B_1 = (2B_0)^{-1} h \pmod{\mathcal{D}}$. Then we set $B = B_0 + B_1 \mathcal{D} \pmod{A}$.
3. If B is even, then set $B = A - B$.

Finally we can compute C as $C = (B^2 - kN)/(4A)$. With those choices, one has by (5.4)

$$Q(X) \equiv \left(\frac{2AX + B}{2\mathcal{D}} \right)^2 \pmod{kN},$$

which is easy to evaluate and a square modulo kN . The roots of $Q(X) \pmod{p}$ for a prime p are

$$\frac{-B \pm \sqrt{kN}}{2A} \pmod{p}. \quad (5.5)$$

The cost of finding the coefficients is dominated by the probable prime test for finding a suitable \mathcal{D} and by the square root extraction in step 1.

5.2. Exercise *Show that the three steps we mentioned above really compute a square root of kN modulo $4A$.*

5.4 The Self Initializing Variant of the MPQS

The Self Initializing Variant of the MPQS is based on ideas of Peter L. Montgomery; it was first described by W.R. Alford and C. Pomerance in [AlPo93]. This variant describes another method for finding many suitable polynomials, but the computation of these polynomials is cheaper than in Silverman's algorithm.

Again we try to find a polynomial $Q(X)$ as given in (5.4). Instead of choosing the coefficient A as a square of a (pseudo) prime number, we choose A as a product of several prime numbers which are elements of the factor basis, i.e. for some bound b we choose

$$A = \prod_{i=1}^b q_i, \quad q_i \in \mathcal{F}.$$

In general, A is no more a square modulo kN . But note that kN is a square modulo all primes q_i , since q_i is an element of the factor basis. Then with (5.4) and the condition that $B^2 - 4AC \equiv 0 \pmod{kN}$, we see that $4 \cdot A \cdot Q(X)$ evaluates to a square modulo kN . Assume that all primes q_i are elements of the given factor basis \mathcal{F} . Since we then know the decomposition of $4A$ over \mathcal{F} , a decomposition of the polynomial value $Q(x)$ for some $x \in \{-\mathcal{M}, \dots, \mathcal{M}\}$ leads to a relation. The main difference to Silverman's method is the fact that now the congruence

$$B^2 \equiv kN \pmod{A} \quad (5.6)$$

has exactly 2^b solutions instead of 2 (this fact follows directly from the Chinese Remainder Theorem).

Let B_{q_i} be a solution to $B_{q_i}^2 \equiv kN \pmod{q_i}$ for $1 \leq i \leq b$. We set for $1 \leq i \leq b$

$$\tilde{B}_i = B_{q_i} \cdot \frac{A}{q_i} \cdot \left(\left(\frac{A}{q_i} \right)^{-1} \pmod{q_i} \right).$$

These integers \tilde{B}_i satisfy the following condition:

$$\tilde{B}_i \equiv \begin{cases} B_{q_i} \pmod{q_j} & \text{for } i = j, \\ 0 \pmod{q_j} & \text{for } i \neq j. \end{cases}$$

Therefore

$$B = \sum_{i=1}^b \pm \tilde{B}_i$$

is a solution to (5.6). The coefficient C of $Q(X)$ can be easily computed from the equation $B^2 - 4AC \equiv 0 \pmod{kN}$.

Now we will describe an efficient way of computing all possibilities for B . Assume that A is fixed and that B_j , $1 \leq j \leq 2^b$, are the 2^b different possible values for B which lead to polynomials $Q_j(X)$. By construction, we can write

$$B_j = \sum_{i=1}^b (-1)^{e_{i,j}} \tilde{B}_i$$

for $e_{i,j} \in \{0, 1\}$, $1 \leq i \leq b$. Therefore we identify the integers B_j with the exponent vector $\hat{e}_j := (e_{b,j}, \dots, e_{1,j})$. If we interpret these exponent vectors as integers, then enumerating all possible B_j 's reduces to the problem to enumerate all integers $0, \dots, 2^b - 1$. Our goal is to find an algorithm which minimizes the cost for computing B_{j+1} if we already know B_j . Moreover, this algorithm should produce all 2^b possible B -values in 2^b iterations.

The trivial idea leads to the solution to increment \hat{e}_j in each iteration by one. Unfortunately, this is no good idea since ‘‘carries’’ correspond to multiple changes in the B_j . A much better solution is based on a so called Gray code. Gray Codes describe an order how we can enumerate all bit strings of a given length with exactly one change per iteration.

Let a bit string of length r be given as $\underline{b} = (b_r, \dots, b_1)$, where $b_i \in \{0, 1\}$, $1 \leq i \leq r$. For enumerating all such bit strings, we start with $\underline{b}_0 = (0, \dots, 0)$. The next bit string is determined in the following way: assume that we know \underline{b}_{i-1} . Let l be the maximal power of 2 that divides i . Then we obtain the next bit string \underline{b}_i by flipping the $(l+1)$ -th bit in \underline{b}_{i-1} .

5.3. Example Assume that $r = 3$. Then we start with $\underline{b}_0 = (0, 0, 0)$. The next bit string can be determined by flipping the bit with index 1, i.e. $\underline{b}_1 = (0, 0, 1)$. If we iterate this procedure, we get $\underline{b}_2 = (0, 1, 1)$, $\underline{b}_3 = (0, 1, 0)$, $\underline{b}_4 = (1, 1, 0)$, $\underline{b}_5 = (1, 1, 1)$, $\underline{b}_6 = (1, 0, 1)$ and $\underline{b}_7 = (1, 0, 0)$.

5.3. Exercise Show that a Gray Code really enumerates all bit strings of length r .

We can use the same principle to find all B -values with minimal changes. We start with

$$B_0 = \sum_{i=1}^b \tilde{B}_i$$

and compute the following recursion

$$B_{j+1} = B_j - 2 \cdot (-1)^{d(j+1)} \cdot \tilde{B}_{\nu(j+1)+1}, \quad (5.7)$$

where $\nu(k)$ is the maximal exponent r such that 2^r divides k , and $d(k) = \lfloor (k/2^{\nu(k)} + 1)/2 \rfloor - \lfloor k/(2^{\nu(k)+1}) \rfloor$. The advantage of this method is the fact that only have to do one addition/subtraction of one of the integers \tilde{B}_i . Therefore this procedure is an extremely cheap method for enumerating all possibilities for B .

Another advantage of this method for enumerating all possible B 's is the fact that this recursion induces a relationship between the zeros modulo p (for a prime $p \in \mathcal{F}$) of subsequent polynomials $Q_j(X)$. Let $z_{p,1}^{(j)}, z_{p,2}^{(j)}$ be the roots of the polynomial $Q_j(X)$ modulo a prime $p \in \mathcal{F}$. Then the roots of the $(j+1)$ -th polynomial $Q_{j+1}(X)$ can be computed using the roots of the j -th polynomial $Q_j(X)$. In detail, we have by (5.5) and (5.7)

$$\begin{aligned} z_{p,1}^{(j+1)} &\equiv \frac{1}{2A} \cdot \left(-B_{j+1} + \sqrt{kN} \right) \pmod{p} \\ &\equiv \frac{1}{2A} \cdot \left(-\left(B_j - 2 \cdot (-1)^{d(j+1)} \cdot \tilde{B}_{\nu(j+1)+1} \right) + \sqrt{kN} \right) \pmod{p} \\ &\equiv \frac{1}{2A} \cdot \left(-B_j + \sqrt{kN} \right) + \frac{1}{2A} \cdot 2 \cdot (-1)^{d(j+1)} \cdot \tilde{B}_{\nu(j+1)+1} \pmod{p} \\ &\equiv z_{p,1}^{(j)} + \frac{1}{A} \cdot (-1)^{d(j+1)} \cdot \tilde{B}_{\nu(j+1)+1} \pmod{p}. \end{aligned}$$

Similarly, we can compute the second root $z_{p,2}^{(j+1)}$ of $Q_{j+1}(X)$ modulo p . This connection between the roots of different polynomials $Q_j(X)$ is of great advantage in practice. The running time for the computation of the different polynomials and the starting points for the sieving procedure decreases to less than 1 % of the total running time. Therefore the self initializing variant of the QS is about 1.5 times faster than Silverman's MPQS version. Another advantage of the self initializing variant is the fact that it can be easily implemented on a parallel computer as a Intel Hypercube. For an implementation report for the QS on a Hypercube, see [So95].

5.5 The Large Prime and Double Large Prime Variation

In the basic version of the quadratic sieve, we find (full) relations, if the value of the chosen quadratic polynomial $Q(x)$ at the place x in the sieving interval completely decomposes over the chosen factor basis \mathcal{F} . A practical observation is the fact that the search for a full relation is time expensive. This limits the practical use of the MPQS to factorizations of integers with up to 50 decimal digits. Therefore the so called **Large Prime Variation of the MPQS (PMPQS)** was developed. In this variation, we allow partial decomposition of polynomial values $Q(x)$, i.e. we consider factorizations

$$Q(x) = L \cdot \prod_{p \in \mathcal{F}} p^{e_p}.$$

In such a **partial relation** L (the so called **large prime**) is a prime which is bigger than the maximal prime in the factor basis \mathcal{F} , but smaller than some large prime bound \mathcal{B}_L . Note that in the logarithm sieving variant, we have to change the final test approximation which is used for a decision whether we should test a value $Q(x)$ with trial division or not. In practice, we can choose the large prime bound as p_{max}^2 , where p_{max} is the maximal prime in the factor basis. This choice has the advantage that we do not have to check primality. If we have trial divided all primes in the factor basis and the remaining part is smaller than p_{max}^2 , then this remaining part must be a prime number.

Of course, we have to show the advantage of the large prime variation. Assume that we have found two partial relations with the same large prime. Then the multiplication of these two partial relations yields a full relation. Moreover, it is much easier to find partial relations than full relations (see Section 5.6). In [De93], the author gives an example of

a factorization of a 102-digit integer where the combination of 180 879 partial relations leads to 11 433 additional full relations. The large prime multi polynomial version of the quadratic sieve needs about half the time of the MPQS.

After the publication of the PMPQS the natural question was, whether it is possible to allow two large primes. In [LeMa91], the authors present the **Double Large Prime Variation (PPMPQS)** of the quadratic sieve, which answers this question. A **partial-partial relation** is the decomposition of $Q(x)$ at some place $x \in \{-\mathcal{M}, \dots, \mathcal{M}\}$ of the form

$$Q(x) = L_1 \cdot L_2 \cdot \prod_{p \in \mathcal{F}} p^{e_p},$$

where L_i are primes satisfying $p_{max} < L_1, L_2 < \mathcal{B}_L < p_{max} x^2$. Assume that we find an integer $R < \mathcal{B}_L^2$ after trial division. If $R < \mathcal{B}_L$, then R must be prime and we have found a partial relation. Note that we can treat partial relations as partial-partial relations if we allow $L_2 = 1$. If $R \geq \mathcal{B}_L$, we use a pseudo primality test to check whether R is composite. If R is prime, we discard it, since it is bigger than \mathcal{B}_L . Otherwise we try to factor R . We immediately know that it is composed of exactly two primes smaller than \mathcal{B}_L . In practice, it is advisable to either use the SQUFOF algorithm or the Pollard ρ -algorithm (see Algorithm 7). It is possible (and a good idea in practice) to restrict the number of iterations in both these algorithms and to combine the two algorithms. If they do not succeed to factor R after a prescribed number of iterations, we discard R because of time considerations. In practice, almost 99 % of all R can be factored with the SQUFOF algorithm.

After describing the method for finding partial-partial relations, we have to show how to combine these partial-partial relations to full relations. A detailed description of an algorithm for this problem can be found in [LeMa91] or [De93], we just mention the main idea. We reduce the given problem to a graph problem: Construct a graph where nodes are given as large primes and an edge links to nodes if and only if there exists a partial-partial relation with these two large primes. Finding a combination of partial-partial relations, such that all large primes of these relations combine to squares is equivalent to find a cycle in the prescribed graph. There exist several algorithms to compute such cycles.

5.6 Practical Successes with the PPMPQS

Detailed description of several factorizations of integers with more than 100 decimal digits can be found in [De93]. The sieving part of all these computations was done on a local area network of computers. Usually, sieving is the most time consuming step.

The two biggest integers which were factored with the quadratic sieve are so called RSA challenge numbers (for a description and a list of these numbers, see [RSA]). The factorization of the RSA-120 number with 120 decimal digits was finished in 1993 by a group of 4 people (see [DeDoLeMa93]), the 129 digit number RSA-129 was factored by a group of people. In the latter factorization, the sieving step was even distributed over the Internet (see [AtGrLeLe95]).

Chapter 6

The Number Field Sieve (NFS)

The **Number Field Sieve (NFS)** is the most modern and most advanced general factoring algorithm. The NFS is supposed to be superior to the PMPQS for integers with more than 115 decimal digits. The original description of the NFS (the so called **Special Number Field Sieve (SNFS)**) was restricted to factorizations numbers of a special form (see [LLMP90]). But soon after the publication of the SNFS, the ideas of this algorithm were generalized to arbitrary integers, and practical implementations have proven the practicability of the algorithm. Assume in the following that we want to factor an integer N , which is odd, composite and no perfect power.

6.1 The Basic Idea of the NFS

In the QS, we generate relations with the help of a quadratic polynomial and a sieve. With the sieve, we can test for a large number of integers of size approximately \sqrt{N} , whether these integers decompose over a factor basis. The NFS extends these ideas by exchanging the quadratic polynomial by a polynomial of larger degree. Unfortunately, this change aggregates the algorithm significantly.

Let $f(X) \in \mathbb{Z}[X]$ be a polynomial of degree d (in practice, d should be chosen between 3 and 6) which is irreducible over the rational numbers, and let $m \in \mathbb{N}$ be an integer with

$$f(m) \equiv 0 \pmod{N}.$$

The coefficients of the polynomial $f(X)$ should be as small as possible, since the size of the coefficients influences the running time of the most important part of the algorithm. One of the main practical difficulties of the number field sieve is the problem of finding a “good” polynomial $f(X)$ and a suitable integer m for given N . If N has a special form, for example $N = b^r - c$ with “small” integers b and c (N is a so called **Cunningham number**), we find $f(X)$ and m easily:

6.1. Example *The seventh Fermat number $F_7 = 2^{2^7} + 1 = 2^{128} + 1$, a number with 39 digits, was first factored in 1970 by Brillhart and Morrison with the CFRAC algorithm. If we look for a suitable polynomial $f(X)$ for this number, we get some problems. Thus we use a trick and try to factor the number $N = 2 \cdot F_7 = 2^{129} + 2$. Now we can use the polynomial*

$$f(X) = X^3 + 2 \quad \text{and} \quad m = 2^{43}.$$

Note that the SNFS was restricted to numbers of this form. In general however, this problem generates a lot of difficulties (for example, nobody has yet found a “good” algorithm for computing m and $f(X)$ for RSA-moduli, i.e. integers, which are the product of two approximately equal sized primes). Nevertheless the following construction always works:

Let d be the degree of $f(X)$. We look for a polynomial

$$f(X) = f_d \cdot X^d + f_{d-1} \cdot X^{d-1} + \dots + f_0$$

and a number m with $f(m) = f_d \cdot m^d + f_{d-1} \cdot m^{d-1} + \dots + f_0 = kN$ for some small integer $k \in \mathbb{N}$. We set $m = \lfloor (kN)^{1/d} \rfloor$. Then we can quickly find f_0, \dots, f_d with $0 \leq f_j < m$ by computing the m -adic representation of kN . Next we check whether the corresponding polynomial is irreducible. In [BuLePo93], it is shown that this question can be answered in a probabilistic fashion by checking the irreducibility of $f(X) \pmod q$ for “a few” small primes q . Moreover, the authors give a method for testing irreducibility modulo q (see also [Co93]).

If the polynomial $f(X)$ is not irreducible in \mathbb{Z} , then we use the fact that for $r \in \mathbb{Z}$ and $1 \leq i \leq d$ the polynomial

$$\tilde{f}(X) = f(X) + r \cdot (X^i - m X^{i-1})$$

satisfies $\tilde{f}(m) = f(m)$. Trying a few small integers r , we find a suitable irreducible polynomial with high probability.

Since $f(X)$ is in general not monic, we define

$$g(X) = f\left(\frac{X}{f_d}\right) \cdot f_d^{d-1}.$$

Then $g(X)$ is defined over \mathbb{Z} , it is irreducible and monic. In the following, we assume that $\rho \in \mathbb{C}$ is a (formal) complex root of $f(X)$.

6.1. Lemma *The element $\omega = f_d \cdot \rho$ is a root of $g(X)$. The two number fields $\mathbb{Q}(\rho)$ and $\mathbb{Q}(\omega)$ are equal. Moreover, the map $\psi : \mathbb{Z}[\omega] \rightarrow \mathbb{Z}/N\mathbb{Z}$, which maps ω to $f_d \cdot m$, is a ring homomorphism.*

The idea of the NFS is as following: If we succeed to find a set S of pairs $(a, b) \in \mathbb{Z}^2$, such that the cardinality of S is even and

$$\begin{aligned} \prod_{(a,b) \in S} (a + b \cdot m) &= \tilde{x}^2 \in \mathbb{Z}, \\ \prod_{(a,b) \in S} (a + b \cdot \rho) &= \tilde{\delta}^2 \in \mathbb{Q}(\rho), \end{aligned}$$

then we have a chance to construct integers x and y with $x^2 \equiv y^2 \pmod N$. If we are lucky and $x \not\equiv \pm y \pmod N$, then $\gcd(x - y, N)$ is a proper divisor of N . The construction of these integers can be done in the following way: Note first that

$$f_d^{|S|} \cdot \prod_{(a,b) \in S} (a + b \cdot \rho) = \prod_{(a,b) \in S} (a \cdot f_d + b \cdot \omega)$$

is equal to a square δ^2 . Additionally, it is an element of the ring $\mathbb{Z}[\omega]$. The square root δ is an element in the algebraic number field $\mathbb{Q}(\omega)$. Unfortunately, it is not obvious whether

this square root δ is also an element of the ring $\mathbb{Z}[\omega]$. If however this is true, then we can use the map ψ given in Lemma 6.1 to construct the integers x and y :

$$\begin{aligned}
y^2 &= \psi(\delta)^2 &\equiv &\psi(\delta^2) \\
&&&\equiv \psi\left(\prod_{(a,b)\in S} (a \cdot f_d + b \cdot \omega)\right) \\
&&&\equiv \prod_{(a,b)\in S} \psi(a \cdot f_d + b \cdot \omega) \\
&&&\equiv \prod_{(a,b)\in S} (a \cdot f_d + b \cdot f_d \cdot m) \\
&&&\equiv f_d^{|S|} \cdot \prod_{(a,b)\in S} (a + b \cdot m) \\
&&&\equiv f_d^{|S|} \cdot \tilde{x}^2 \equiv x^2 \pmod{N}.
\end{aligned}$$

Therefore it remains to answer the following questions:

1. How do we find a suitable set S ?
2. How can we check whether $\delta \in \mathbb{Z}[\omega]$, and how can we extract a square root in $\mathbb{Z}[\omega]$?

The answer to the first of these questions is similar to the methods which we used in the quadratic sieve. We generate a set of **relations**, and we try to combine these relations “in a clever way”. Each relation is linked to a pair $(a, b) \in \mathbb{Z}^2$ and an exponent vector. The search for those relations which should be combined can again be done by solving a linear system over \mathbb{F}_2 . Before we describe the generation of relations, we review several facts from algebraic number theory.

6.2 Algebraic Background

We review several facts from algebraic number theory without proof. Detailed information about these statements can be found in most algebraic number theory books, see for example [We76].

Let $g(X) \in \mathbb{Z}[X]$ be a monic and irreducible polynomial of degree d , and let ω be a (complex) root of $g(X)$. It is easy to see that the set $\mathbb{Z}[\omega]$ forms a ring, and that $\mathbb{Q}(\omega)$ is a field.

Obviously, the polynomial $g(X)$ splits completely over the complex numbers. All other zeros of $g(X)$ are denoted conjugates of ω . For an element $\xi \in \mathbb{Q}(\omega)$, we get a conjugate of ξ by replacing ω by one of its conjugates. The product of all conjugates of an element ξ is called the norm $N(\xi)$ of ξ ; it is a rational number. If we compute a formula for the norm of elements of the form $\xi = a + b\omega$, then we obtain $N(\xi) = N(a, b)$, where $N(a, b)$ is given as in (6.3).

Let $\wp \subset \mathbb{Z}[\omega]$ be a proper prime ideal of $\mathbb{Z}[\omega]$. Then the factor group $\mathbb{Z}[\omega]/\wp$ is a finite field. The order of this field is called the norm of the prime ideal \wp , we write $N(\wp)$. The degree of \wp is the degree of this field as extension field over the prime field. We say that

\wp is a first degree prime ideal if the degree of \wp is one. Obviously, the norm of first degree prime ideals is p , where p is a prime number. It should be mentioned that there exists exactly one rational prime number in any prime ideal; this prime number is equal to the characteristic of the prime field lying below $\mathbb{Z}[\omega]/\wp$.

Let \wp be a first degree prime ideal of $\mathbb{Z}[\omega]$. Then the factor group $\mathbb{Z}[\omega]/\wp$ is isomorphic to $\mathbb{Z}/p\mathbb{Z}$ for some prime number p . Thus there exists a homomorphism

$$\begin{aligned} \mathbb{Z}[\omega] &\longrightarrow \mathbb{Z}/p\mathbb{Z}, \\ \omega &\longmapsto c_p, \end{aligned}$$

with kernel \wp . Here, c_p is a root of the reduced polynomial $f(X) \bmod p$. On the other hand, if c_p is a zero of $f(X) \bmod p$, we can define a homomorphism which maps ω to c_p . The kernel of this homomorphism is a prime ideal of $\mathbb{Z}[\omega]$. Therefore there is a one-to-one correspondence between prime ideals of $\mathbb{Z}[\omega]$ and pairs (p, c_p) , where $0 \leq c_p < p$ is a root of $f(X) \bmod p$. This correspondence can be made precise, a pair (p, c_p) corresponds to the ideal generated by $\langle p, \omega - c_p \rangle$.

6.3 Finding Relations

The idea for finding relations is similar to the Quadratic Sieve, where we decomposed integers over a chosen factor basis. In contrary to the quadratic sieve, we are now working in two different rings, namely \mathbb{Z} and $\mathbb{Z}(\omega)$. Therefore we have to choose two factor bases, and we do the decompositions in both these rings. The **rational factor basis** \mathcal{F}_1 consists of all prime numbers of \mathbb{Z} up to some constant C_1 :

$$\mathcal{F}_1 = \{p \in \mathbb{P}_{\leq C_1}\} \cup \{-1\} = \{p_1, \dots, p_k\}. \quad (6.1)$$

In addition to \mathcal{F}_1 , there exists a second factor basis for $\mathbb{Z}[\omega]$, the so called **algebraic factor basis**:

$$\begin{aligned} \mathcal{F}_2 &= \left\{ (p, c_p); p \in \mathbb{P}_{\leq C_2}, f(c_p) \equiv 0 \pmod{p}, p \nmid f_D \right\} \\ &\cup \left\{ (p, \infty); p \in \mathbb{P}_{\leq C_2}, p \mid f_D \right\}. \end{aligned} \quad (6.2)$$

The algebraic meaning of the special form of \mathcal{F}_2 will be explained in Section 6.2. We define for two integers $a, b \in \mathbb{Z}$ the norm function

$$N(a, b) = f\left(\frac{-a}{b}\right) \cdot (-b)^d. \quad (6.3)$$

We say that a pair $(a, b) \in \mathbb{Z}^2$ is a **good pair** if it satisfies three conditions:

$$\begin{aligned} \gcd(a, b) &= 1, \\ a + bm &= \prod_{p \in \mathcal{F}_1} p^{e_p}, \quad e_p \geq 0, \\ |N(a, b)| &= \prod_{(p, c_p) \in \mathcal{F}_2} p^{f(p, c_p)}, \quad f(p, c_p) \geq 0. \end{aligned}$$

Every good pair will give rise to a relation. For finding good pairs, the NFS uses three different sieving phases. Therefore all three tests can be accomplished very fast. We describe here the linear sieve idea; another sieving variant is described in Section 6.5.2.

Assume that we want to find all good pairs (a, b) in a given set $[A_{min}, A_{max}] \times [1, B_{max}]$. We fix a b -value b' in the admissible range for b . Then we initialize a sieve array for all $A_{min} \leq a \leq A_{max}$. First we find all values for a such that (a, b') satisfies the first condition. This can be accomplished with sieving with all prime factors of b' . For all such primes $q_{b'}$, we mark all multiples of $q_{b'}$ in $[A_{min}, A_{max}]$ as “bad” (more precisely, we mark the corresponding array entries as bad). If we have performed this procedure for all prime factors of b' , then all non marked sieving places correspond to values of a such that (a, b') passes the first condition. This gcd-sieve is about 50 times faster than the naive algorithm, which would test each pair with a gcd-run.

The second sieve starts with a reinitialization of all “good” a -values of the sieve array T as $T[a] = a + b' m$. Exactly as in the Quadratic Sieve, we sieve with all elements of the rational factor basis \mathcal{F}_1 . For every prime $q \in \mathbb{F}_1$, we determine a starting point a_q as the least integer greater A_{min} such that $T[a_q] \equiv 0 \pmod{q}$. Then we find all other array entries where $T[a]$ is divisible by q as $T[a_q + kq]$ for $k \geq 1$. Note that the situation in the NFS is even simpler as in the Quadratic Sieve, since we do not have to extract a square root modulo q . After the second sieve we know the subset of all pairs (A, b') which fulfill condition one and two.

The third sieve is the algebraic sieve, which uses the algebraic factor basis \mathcal{F}_2 . Again we reinitialize the sieve array T for all “surviving” values of a as $T[a] = |N(a, b')|$. Sieving with elements of \mathcal{F}_2 splits into two cases. We sieve with a factor basis element (p, c_p) with $c_p \neq \infty$ exactly as in the rational sieve: we determine the smallest integer $a_p \geq A_{min}$ such that $a_p + b' c_p \equiv 0 \pmod{p}$. Then we sieve with the prime p as usual, we divide at all places $T[a_p + kp]$ for $k \geq 0$ by p . For a factor basis element (p, ∞) , the situation is different. If p is no divisor of b' , we are done; otherwise we have to determine for all “surviving” values $A_{min} \leq a \leq A_{max}$ the maximal power of p which divides $T[a]$. Note that the exponents $f_{(p, c_p)}$ are given as the maximal power of p which divides $|N(a, b')|$.

It should be mentioned that we can use the same tricks as in the quadratic sieve for speeding up the sieving phase. The most important tricks are the use of logarithms and sieving only with factor basis of medium size. Of course, we have to check the correctness of the computation by a trial division (or a fourth correct sieve). A description of an implementation for this sieving phase and a lot of practical aspects can be found in [Za95].

Another difference to the quadratic sieve is the construction of the exponent vectors. Assume that we have found a good pair (a, b) which leads to a decomposition over \mathcal{F}_1 and \mathcal{F}_2 . Then we know exponents e_p and $f_{(p, c_p)}$ such that

$$a + bm = \prod_{p \in \mathcal{F}_1} p^{e_p} \quad \text{and} \quad |N(a, b)| = \prod_{(p, c_p) \in \mathcal{F}_2} p^{f_{(p, c_p)}} .$$

Exponent vector entries corresponding to rational primes are given in exact the same way as in the quadratic sieve. For algebraic primes (p, c_p) , we have to change these entries: if $c_p \neq \infty$, define $l_{(p, c_p)} = f_{(p, c_p)}$, otherwise set $l_{(p, \infty)} = f_{(p, \infty)} + \text{ord}_p(f_d)$. Then we find the following lemma in [BuLePo93].

16. Lemma *If $S \subset \mathbb{Z}^2$ is a subset of coprime pairs (a, b) with*

$$\prod_{(a,b) \in S} (a + b\rho) = \delta^2$$

for some $\delta \in \mathbb{Q}(\rho)$, then

$$\sum_{(a,b) \in S} l_{(p,c_p)}(a + b\rho) \equiv 0 \pmod{2}.$$

Unfortunately, this condition is not sufficient, i.e. if we find a non trivial combination of the algebraic exponent vectors, we are not sure that we really have found a square in $\mathbb{Q}(\rho)$. To assure this property, we have to use another trick, the so called **quadratic symbols**. Basis for this trick is the following theorem (see [BuLePo93]).

17. Theorem *Let $S \subset \mathbb{Z}^2$ is a subset of coprime pairs (a, b) such that $\prod_{(a,b) \in S} (a + b\rho)$ is a square in $\mathbb{Q}(\rho)$. Let (q, c_q) correspond to a first degree prime ideal, such that $a + bc_q \not\equiv 0 \pmod{q}$ for all pairs $(a, b) \in S$ and let $f'(c_q) \not\equiv 0 \pmod{q}$. Then the following product of Legendre Symbols satisfies*

$$\prod_{(a,b) \in S} \left(\frac{a + b\rho}{q} \right) = 1.$$

This Theorem is the basis for the following extension of exponent vectors, which assure with “very high probability” that a solution of the linear system really produces a square in $\mathbb{Q}(\rho)$. We test the statement of Theorem 17 for “a few” additional prime ideals (q, c_q) which are not elements of the chosen algebraic factor basis. For each of these prime ideals, we extend the exponent vector for (a, b) by an entry $g_q(a, b) = 0$, if $(a + bc_q)$ is a square modulo q and 1 otherwise. It is easy to see that for given (q, c_q) the product is one if and only if

$$\sum_{(a,b) \in S} g_q(a, b) \equiv 0 \pmod{2}.$$

In practice, it is sufficient to choose about 100 additional prime ideals for determining the quadratic characters. The size of the linear systems mainly depend on the size of the two factor bases.

The next sections handles the remaining problem, then extraction of a square root in the ring $\mathbb{Z}[\omega]$.

6.4 Extracting a Square Root in $\mathbb{Z}[\omega]$

Assume that we have found a subset S of good pairs, such that

$$\prod_{(a,b) \in S} (a + b \cdot \rho) = \delta^2,$$

and $\delta \in \mathbb{Z}[\omega]$ is satisfied. The next step is the computation of this square root δ . If we evaluate the given product on the left hand side of this equation, the size of the coefficients

are huge (note that the computation has to be done over the rational integers). This size makes computations impossible in practical sense.

In [BuLePo93], the authors propose the following practical algorithm for extracting the square root. In the first step of the square root algorithm, we choose a prime q such that the given polynomial $f(X)$ is irreducible modulo q . Such primes are called inert primes. For such primes, the “reduced” ring $\mathbf{Z}/q\mathbf{Z}[\omega]$ is a finite field with q^d elements. But we have already seen how we can extract square roots in finite fields. The RESSOL algorithm (compare Section 5.1.2) can easily be generalized to arbitrary finite fields. Therefore we can compute the reduction δ_q of δ , which satisfies $\delta_q \equiv \delta \pmod{q}$. Obviously, the size of all coefficients in this computation is bounded by q^2 .

Then we can use Hensel Lifting to lift these solutions to a solution modulo q^i for arbitrary large integers i . We repeat this operation until we have found the correct solution over \mathbf{Z} . Obviously, this method has the disadvantage that again lot’s of operations with huge numbers are necessary.

Couveignes gives an extension of this idea, which behaves much better in practice [Cv93]. Instead using Hensel lifting, he uses the Chinese Remainder Theorem. First he develops an upper bound for the absolute size of the coefficients. Then we extract the reduced square roots δ_q of δ^2 modulo a lot of inert primes q . If we use the homomorphism

$$\begin{aligned} \psi_q : \mathbf{Z}/q\mathbf{Z}[\omega] &\longmapsto \mathbf{Z}/q\mathbf{Z}, \\ \omega &\longmapsto f_d m \pmod{q}, \end{aligned}$$

then we can compute the image of a square root by Chinese remaindering the elements $\psi_q(\Delta_q)$. The advantage of this method is the following trick, which reduces the size of elements in the Chinese remainder step significantly.

Assume that we want to use the Chinese Remainder Algorithm with input $a_i \pmod{m_i}$, $1 \leq i \leq r$ with coprime moduli m_i and reduce the result modulo N . The trivial implementation would compute the unique integer $a \pmod{\prod_{i=1}^r m_i}$ which satisfies all the given congruence condition (application of CRT) and reduce a modulo N . In this way, we would use a lot of computations with integers of size roughly $M = \prod_{i=1}^d m_i$. The following trick computes an approximation for the division with remainder in the final reduction step.

Set $M_i = M/m_i$ and $b_i = M_i^{-1} \pmod{m_i}$. The usual Chinese remainder equation is given as

$$a = \sum_{i=1}^r a_i b_i M_i.$$

The value of a is only uniquely determined modulo M ; therefore we have to reduce this integer modulo M . A division with remainder shows that

$$a = \left\lfloor \frac{a}{M} \right\rfloor \cdot M$$

is the absolute smallest residue of a modulo M . The important fact is that we can determine $\lfloor \frac{a}{M} \rfloor$ on the fly as

$$\left\lfloor \sum_{i=1}^r \frac{a_i \cdot b_i}{m_i} \right\rfloor.$$

Finally, we are interested in the result modulo N , such we have to perform another reduction modulo N . But this reduction is almost straightforward. The only critical part is the sum over $(a_i \cdot b_i)/m_i$, which is a sum of integers of size approximately m_i .

One problem of Couveignes method is the non uniqueness of square roots. For each prime q , there are two square roots, and we have to determine a way to combine the correct one. We can overcome this problem, if we restrict the degree d of the NFS polynomial to be odd. In this case, we can use the norm to classify the correct square root, since now $N(-\alpha) = -N(\alpha)$ for $\alpha \in \mathbb{Z}[\omega]$. Obviously, the norm of the square root we want to compute is the square root of the norm of the square element. The norm of the square is easy to compute by the norm of the combined elements. Finally, we determine that reduced square root whose norm is equivalent to the square root norm modulo q .

6.5 Practical Improvements

As in the quadratic sieve, there exist several variants and practical improvements to the basic idea of the NFS. In this section, we briefly mention three of these ideas. A better overview on these improvements can be found in [Za95]. Moreover, the development of the NFS is a popular working area, such that further improvements can be expected.

6.5.1 The LLL Variant for Finding Polynomials

We already described one method for finding a polynomial $f(X) \in \mathbb{Z}[X]$ of degree d which can be used in the NFS. We used the m -adic expansion of a small multiple of N , where $m \approx N^{1/d}$. Since in the third part of the sieving phase, the norm of elements is decomposed over a factor basis, we should try to find polynomials for which “small pairs” (a, b) have small norm. It is easy to see that this condition can be satisfied, if the coefficients of the polynomial $f(X)$ are as small as possible.

There exists another idea how we can find such a polynomial. This idea links the determination of $f(X)$ to the search for short vectors in a lattice. Fix an integer $m \approx N^{1/d}$ and define

$$L = \left\{ (f_0, \dots, f_d) \in \mathbb{Z}^{d+1} ; \sum_{i=0}^d f_i \cdot m^i \equiv 0 \pmod{N} \right\}.$$

The set L forms a $(d + 1)$ dimensional lattice in \mathbb{Z}^{d+1} . A basis for this lattice can be written down very easily: the vectors $(N, 0, \dots, 0), (1, -m, 0, \dots, 0), \dots, (0, \dots, 0, 1, -m)$ form such a basis. There exists a polynomial time algorithm which determines with input of a lattice basis another basis, which is usually shorter than the input basis. We apply this LLL algorithm [LeLeLo82] on the given lattice. The output of this algorithm gives other polynomials which can be used in the NFS.

The disadvantage of this method is the fact that we have to fix m in advance. The quality of the polynomials produced by LLL is generally slightly better than the polynomials of the m -adic algorithm, but the difference is tiny. In fact, the biggest problem of the NFS is currently the search for a good polynomial $f(X)$. At the moment, there is no really satisfying algorithm known for this problem.

6.5.2 The Lattice Sieve

We have described the sieving phase of the basic NFS. This procedure is usually called line sieving procedure. There exists a variant of this idea, the so called lattice sieve, which

goes back to Pollard (see [Po91]).

The idea of the lattice sieve is to fix a rational factor basis element q and to determine all pairs (a, b) such that q divides $a + b m$, i.e. to determine a set

$$L_q = \{(a, b) \in \mathbb{Z}^2 ; a + b m \equiv 0 \pmod{q}\} .$$

This set L_q forms a lattice in \mathbb{Z}^2 , a basis for this lattice is given by the two elements $(q, 0)$ and $(-m \pmod{q}, 1)$. Using the Gauss reduction algorithm (see [LeLeLo82]) on these two elements, we can compute a shortest basis $\underline{u} = (u_1, u_2)$, $\underline{v} = (v_1, v_2)$ for this lattice. Then all pairs in L_q can be written as

$$(a, b) = \lambda \cdot \underline{u} + \eta \cdot \underline{v} ,$$

where $\lambda, \eta \in \mathbb{Z}$. We directly deduce that $a = \lambda u_1 + \eta v_1$ and $b = \lambda u_2 + \eta v_2$. Since we are looking for pairs (a, b) in the set L_q , we obtain for $u_1 + u_2 m \not\equiv 0 \pmod{q}$ the equation

$$\lambda \equiv -\eta \cdot \frac{v_1 + v_2 m}{u_1 + u_2 m} \pmod{q} . \quad (6.4)$$

If $u_1 + u_2 m \equiv 0 \pmod{q}$, then we deduce that $v_1 + v_2 m \not\equiv 0 \pmod{q}$ (otherwise the basis would only generate a sub lattice of L_q), and therefore q has to be a divisor of η . Obviously, we can use this connection between λ and η for a sieve: we fix a η -value η' in some given range, compute a starting value for λ with (6.4), and then we sieve with distance q . In this way, we find all integers λ in a given interval such that λ, η' induces elements in L_q .

Note that we can use exactly the same idea for elements of the algebraic factor basis. For a factor basis element (p, c_p) with $c_p \neq \infty$, we only have to substitute m by the integer c_p . Details of this procedure can be found in [Za95].

In practice, one should not use all factor basis elements with the lattice sieve. As in the linear sieve, we do not use small factor basis elements, but use only medium size and large elements. The advantage of the lattice sieve is then the fact that we have to examine less pairs to find enough relations. Unfortunately, the lattice sieve needs a lot of main memory, since the sieving areas are usually larger than in the line sieve. Therefore the lattice sieve is restricted to machines with at least 16 MByte main memory. This fact restricts the use of the lattice sieve.

6.5.3 The Quadruple Large Prime Variation

The practical behavior of the quadratic sieve was greatly improved by the use of so called partial relations. Exactly the same idea was used in early implementations of the NFS. Here, we allow the use of two large primes on the rational side. Then we try to combine partial relations to eliminate the large primes on the rational side.

Later this double large prime variation was extended to even a quadruple large prime variation. In this variation, we additionally allow the use of at most large primes on the algebraic side. Thus we use partial relations with up to two rational and two algebraic large primes. In [Za95], a graph algorithm is described which determines a set of partial relations with up to four large primes which can be combined. In the combination, all large primes are eliminated, and a full relation is found.

A practical experience is the fact, that partial relations are found very often. Therefore the memory amount needed for storing these relations is enormous. However there happens

to be an “explosion effect” after we have found enough partial relations. If we have found “enough” partial relations, we find a huge number of combinations and thus a huge number of full relations. Therefore the quadruple large prime is nowadays used in all big factoring attempts done with the NFS.

6.6 Successes with the NFS

Chapter 7

Solving Large Sparse Linear Systems

One important subproblem of the factoring algorithms which we introduced in Chapter 4 is the solution of a large, sparse linear system over the finite field \mathbb{F}_2 with two elements. Before we explain three algorithms for solving these linear systems, we explain several facts about the special structure of the matrices which arise in these systems.

7.1 Structure of the Matrices

We explain some facts about the linear systems which have to be solved in factoring integers. It should be mentioned that a lot of these facts are true for several other algorithmic problems for which sieving algorithms exist (the discrete logarithm problem or computing class groups). Therefore we explain the algorithms in a more general way, and we assume that we want to solve a linear system over a finite prime field \mathbb{F} . There are however some practical improvements for the solution of linear systems over \mathbb{F}_2 (which is necessary in factoring, as we have seen). The main trick is the use of a block techniques, where always blocks of 32 bits are used instead of single bits. Obviously, this trick is especially well suited for 32 bit computers. Technical details of this blocking trick can be found in [Mo95].

The matrices which we want to solve are build up in a special way, as we have seen in the previous chapters. The columns of these matrices are the exponent vectors of relations. Therefore the size of these matrices is approximately equal to the size of the factor bases used in the quadratic sieve of the number field sieve. For integers N with about 100 decimal digits, the dimension of the matrix is approximately $10^5 \times 10^5$, for record NFS computations this size even increases to up to 3 million. Since exponent vectors of relations mostly consist of zero entries, the given matrices are sparse (i.e. the number of non zero entries per column is very small). If we would store such huge matrices in “the usual compact way”, the memory amount would be enormous. Fortunately, we can store the matrix in a sparse representations, if we only store the non zero entries with corresponding indices (note that for matrices defined over \mathbb{F}_2 only the indices of non zero entries have to be stored).

The matrices even have some more structure. Since small factor basis elements correspond

to the “first rows” of the matrix, these rows have a higher probability of being non-sparse as the last rows. The structure of the matrices can be described as in the following picture:

$$\begin{pmatrix} \text{small dense part} \\ \hline \text{large sparse part} \end{pmatrix}.$$

Another important observation is the fact that usually we have more relations than factor basis elements, i.e. the number of columns is greater than the number of rows. Assume that $A \in \mathbb{F}^{m \times n}$ and $\underline{b} \in \mathbb{F}^m$, where $n \geq m$, and we want to solve the linear system

$$A \cdot \underline{x} = \underline{b}. \quad (7.1)$$

We can reduce the solution of (7.1) to the solution of the linear system

$$(A \cdot A^t) \cdot \underline{x}' = \underline{b}.$$

If we know a solution \underline{x}' of this system, then obviously $A^t \cdot \underline{x}'$ is a solution of (7.1). The advantage of this reduction is the fact that $A \cdot A^t$ is a symmetric, square matrix of smaller dimension.

In factoring algorithms, we are interested in the solution of homogeneous linear systems, i.e. we have $\underline{b} = \underline{0}$. With the following trick, we can reduce the solution of homogeneous systems to the solution of inhomogeneous linear systems. Assume that for a matrix $B = [\underline{b}_1, \dots, \underline{b}_n]$ (\underline{b}_i are the columns of the matrix B) we want to solve $B \cdot \underline{x} = \underline{0}$. Then we solve the inhomogeneous system $B' \cdot \underline{x}' = \underline{b}_n$, where $B' = [\underline{b}_1, \dots, \underline{b}_{n-1}]$. A solution $\underline{x}' = (x_1, \dots, x_{n-1})^t$ for this inhomogeneous system gives rise to the solution $\underline{x} = (x_1, \dots, x_{n-1}, -1)^t$ of the original homogeneous system.

After these preliminary observations, we start the description of the following methods, which are well suited for solving linear systems with matrices of this special form:

1. Structured Gaussian Elimination,
2. The Coordinate Recurrence Method of Wiedemann,
3. The Lanczos Algorithm.

7.2 Structured Gaussian Elimination

We want to solve a large linear system over \mathbb{F}_2 which is sparse. Moreover we assume that we have more relations as unknowns (i.e. more columns as rows). This condition can easily be obtained in the QS or the NFS by extending the sieving step a bit.

The most common method for solving linear systems is ordinary Gaussian elimination (see for example [Co93]). If we would use this method from the beginning, the matrix would rapidly fill with new entries, and after a few iterations the matrix would no more be sparse. The size of the matrix would lead to huge memory problems such that this

strategy is impossible in practice. Therefore we first try to do Gaussian elimination steps which preserve the sparseness of the matrix.

The idea of the modified Gaussian elimination algorithm is the following: First we can remove all rows of the matrix which have only one non zero entry. Note that rows correspond to factor basis elements, and that the factor basis element corresponding to such a row can't be combined to a square. In addition, we can eliminate the column which corresponds to the one entry (i.e. the corresponding relation, since the "exponent" of this relation must be zero). After this first step which reduces the size of the matrix slightly, we split the matrix into two parts: the (small) heavy part consists of all rows with "many" nonzero entries, the rest of the matrix is defined to be the "light part". Odlyzko ([LaOd91]) suggests to choose the heavy part of the matrix as the first $\approx n/30$ rows, where n is the total number of rows of the matrix.

Now we perform column operations which reduce the number of non zero entries in the light part (i.e. they generate no new non zero entries in the light part), but perhaps several new entries in the heavy part can be produced. Let a_i be a column which has only one non zero entry in the light part (let (j, i) be the index of this non zero entry). We add column i to all other columns a_k with a one entry in row j (i.e. $a_{j,k} \neq 0$). Obviously, this operation do not generate new one entries in the light part. New entries can only be created in the heavy part of the matrix. We iterate this process until no more such columns exist in the matrix. Note that we have to store all column operations which we perform in this precomputation. If we find no more suitable columns, we enlarge the heavy part and start over.

In this way, we can generate a linear system which is more dense, but smaller as the original system. This new system can be solved with ordinary Gauss elimination. After solving the small system, the remaining variables (of the light part) can be computed by backward-solving and using the stored operations of the precomputation.

Experimentally, Odlyzko has determined that the heavy part should be enlarged as minimal as possible to gain the best solution. If the heavy part becomes too large, you should replace some of the worst columns by some not yet used relations.

Odlyzko has used this method to transform a linear system with 288017 relations and 95697 unknowns with only ≈ 18.2 non zero entries per column (on average) into a system with 7262 relations of 6006 unknowns, but ≈ 80 non zero entries per column.

It should be said that Structured Gaussian Elimination is not suitable for matrices of the enormous size which nowadays occur in NFS computations. Nevertheless it was the method of choice for a long time, especially for QS computations.

7.3 The Coordinate Recurrence Method of Wiedemann

The Coordinate Recurrence Method of Wiedemann ([Wi86]) is an algorithm for solving sparse linear systems over arbitrary finite fields \mathbb{F} . There exists however a block variant for the field with two elements due to Coppersmith ([?]).

Assume without loss of generality that A is a square matrix in $\mathbb{F}^{n \times n}$, $\underline{b} \in \mathbb{F}^n$, and we want to solve the linear system (7.1). We consider the subspace $S = \text{span}\{A^i \cdot \underline{b} ; i \in \mathbb{Z}_{\geq 0}\}$ of

\mathbb{F}^n . On this subspace, we define the endomorphism

$$\begin{aligned} \varphi : S &\longrightarrow S, \\ \underline{v} &\longmapsto A \cdot \underline{v}. \end{aligned} \tag{7.2}$$

Let $f(X) = c_n + c_{n-1}X + \dots + c_1X^{n-1} + X^n$ be the minimal polynomial of the endomorphism φ . By the Theorem of Cayley-Hamilton, we know that $f(\varphi)(\underline{v}) = \underline{0}$ for every $\underline{v} \in S$. In particular, we know that

$$\underline{0} = f(\varphi)(\underline{b}) = c_n \cdot \underline{b} + A \cdot \sum_{i=1}^n c_{n-i} \cdot A^{i-1} \cdot \underline{b} \quad (c_0 := 1).$$

We can assume that A is a regular matrix. If A is not regular, then we do ??. Then the constant coefficient $c_n \neq 0$, because otherwise $\sum_{i=1}^n c_{n-i} \cdot A^{i-1} \cdot \underline{b}$ would be a non trivial solution of the homogeneous system $A \cdot \underline{x} = \underline{0}$. Therefore

$$\underline{x} = -c_n^{-1} \cdot \sum_{i=1}^n c_{n-i} \cdot A^{i-1} \cdot \underline{b} \tag{7.3}$$

is a solution of the given linear system $A \cdot \underline{x} = \underline{b}$.

Thus it is sufficient to find the minimal polynomial of the endomorphism φ in order to solve (7.1). Once we have found this minimal polynomial, we can find a solution of the linear system with n matrix-vector multiplications. Assume that the number of non-zero entries of the matrix A is ω . Then one matrix-vector multiplication can be done in $O(\omega)$ elementary operations in the finite field \mathbb{F} , such that the computation of \underline{x} in (7.3) takes $O(n\omega)$ many elementary operations in \mathbb{F} .

Therefore we can concentrate on the computation of the minimal polynomial $f(X)$ of φ . Pick any vector $\underline{u} \in \mathbb{F}^n$ at random, and consider for $j \geq 0$ the sequence

$$s_j = \langle \underline{u}, A^j \cdot \underline{b} \rangle \in \mathbb{F}$$

(here $\langle \cdot, \cdot \rangle$ denotes the ordinary inner product on \mathbb{F}^n). Again by the Theorem of Cayley-Hamilton, we know

$$\begin{aligned} 0 &= \langle \underline{u}, f(\varphi)(A^j \cdot \underline{b}) \rangle \\ &= \left\langle \underline{u}, A^{n+j} \cdot \underline{b} + \sum_{i=1}^n c_i \cdot A^{n-i+j} \cdot \underline{b} \right\rangle \\ &= \langle \underline{u}, A^{n+j} \cdot \underline{b} \rangle + \sum_{i=1}^n c_i \cdot \langle \underline{u}, A^{n-i+j} \cdot \underline{b} \rangle \\ &= s_{n+j} + \sum_{i=1}^n c_i \cdot s_{n+j-i}, \end{aligned}$$

which is a linear recurrence relation of length n . We will show that the linear recurrence of minimal length L_∞ , which generates the sequence $(s_j)_{j \geq 0}$ is uniquely determined. It coincides with the recurrence of minimal length L_N , which generates $(s_j)_{0 \leq j < N}$ for any $N > 2n$. Such a linear recurrence of minimal length L_N can be inductively computed by the Algorithm of Berlekamp-Massey.

Let $(s_j)_{0 \leq j < k}$ be a sequence of length k , and let L_k be the minimal length of a linear recurrence which generates $(s_j)_{0 \leq j < k}$, i.e. for all $j = L_k, \dots, k-1$ we know coefficients $c_i^{(k)} \in \mathbb{F}$ such that

$$s_j = - \sum_{i=1}^{L_k} c_i^{(k)} \cdot s_{j-i}.$$

The generating polynomial of this sequence is defined as

$$p_k(X) = 1 + c_1^{(k)} X + c_2^{(k)} X^2 + \dots + c_{L_k}^{(k)} X^{L_k}.$$

We describe an inductive procedure due to Berlekamp and Massey for computing the generating polynomial of a sequence $(s_j)_{0 \leq j < k}$ for arbitrary length k . The initialization of the algorithm is done as $L_{-1} = -1$, $L_0 = 0$ and $p_{-1}(X) = p_0(X) = 1$. Moreover we define an integer $d_{-1} = 1$. One invariant of the procedure is the fact that the degree of the generating polynomial $p_{k+1}(X)$ satisfies $L_{k+1} \leq \max\{L_k, k+1-L_k\}$. The iteration step is done as following: Assume that we have found $p_j(X)$ and L_j for all $j \leq k$, and that we want to compute $p_{k+1}(X)$ and L_{k+1} , such that $L+k+1$ satisfies the given invariant. Define a value d_k , which expresses the “error” if we try to generate the $(k+1)$ -th sequence element with the given generating polynomial $p_k(X)$ as

$$d_k = s_k + \sum_{i=1}^{L_k} c_i^{(k)} s_{k-i}.$$

If $d_k = 0$, then the known generating polynomial $p_k(X)$ already generates the $(k+1)$ -th sequence element s_k of the given linear recurrence. Therefore we set $L_{k+1} = L_k$ and $p_{k+1}(X) = p_k(X)$ (note that the invariant is fulfilled). Otherwise, let $m \geq -1$ be the biggest index smaller than k , where the degree of a generating polynomial increased, i.e. we have $L_m < L_{m+1} = L_{m+2} = \dots = L_k$. Then

$$0 \neq d_m = s_m + \sum_{i=1}^{L_m} c_i^{(m)} s_{m-i}$$

and, by the invariant $L_{i+1} \leq \max\{L_i, i+1-L_i\}$ for $i < k$, we have

$$L_k = L_{m+1} \leq m+1-L_m. \quad (7.4)$$

We define the new generating polynomial as $p_{k+1}(X) = p_k(X) - d_k \cdot d_m^{-1} \cdot X^{k-m} \cdot p_m(X)$. The degree L_{k+1} of this polynomial is at most

$$\max\{L_k, k-m+L_m\} = \max\{L_k, k+1-L_k\},$$

because by (7.4) we have $L_m \leq m+1-L_k$. To see that this polynomial is a valid choice (i.e. generates all sequence elements up to the $(k+1)$ -th element s_k), we distinguish two cases: for all $L_{k+1} \leq j < k$, we directly get (note that $p_k(X)$ and $p_m(X)$ are already generating polynomials)

$$\sum_{i=1}^{L_k} c_i^{(k)} \cdot s_{j-i} = -s_j \quad \text{and} \quad \sum_{i=1}^{L_m} c_i^{(m)} \cdot s_{j-k+m-i} = -s_{j-k+m}.$$

For $j = k$, we have

$$s_k + \sum_{i=1}^{L_k} c_i^{(k)} \cdot s_{k-i} = d_k \quad \text{and} \quad s_{k-k+m} + \sum_{i=1}^{L_m} c_i^{(m)} \cdot s_{k-k+m-i} = d_m.$$

These two equations show that the new generating polynomial $p_{k+1}(X)$ generates the sequence $(s_j)_{0 \leq j < k+1}$. Therefore we have finished the description how we can compute a generating polynomial for the linear recurrence $(s_j)_{0 \leq j < k}$ for any given index k . We illustrate the algorithm in the following example.

7.1. Example *Assume that we have the following linear recurrence sequence of numbers, and we want to find a generating polynomial for this sequence:*

$$s_0 = 1, s_1 = 2, s_2 = 3, s_3 = 6, s_4 = 11, s_5 = 20, s_6 = 37.$$

We start the Algorithm of Berlekamp-Massey with $k = 0$ and $L_{-1} = L_0 = 0$ and $p_{-1}(X) = p_0(X) = 1$. We compute d_0 as

$$d_0 = s_0 + \sum_{i=1}^{L_0} c_i^{(0)} \cdot s_{0-i} = s_0 = 1.$$

With $m = -1$, we compute the generating polynomial

$$p_1(X) = p_0(X) - d_0 \cdot (d_{-1})^{-1} \cdot X^{0-(-1)} \cdot p_{-1}(X) = 1 - X.$$

Therefore we have $L_1 = 1$. In the same way, we compute d_1 as

$$d_1 = s_1 + \sum_{i=1}^{L_1} c_i^{(1)} \cdot s_{1-i} = 1,$$

and with $m = 0$ we obtain

$$p_2(X) = p_1(X) - d_1 \cdot d_0^{-1} \cdot X^{1-0} \cdot p_0(X) = 1 - 2X.$$

Using the inductive formulas for p_{k+1} , we can compute the following generating polynomials:

- $k = 2, m = 0$: $d_2 = -1, p_3(X) = 1 - 2X + X^2,$
- $k = 3, m = 2$: $d_3 = 2, p_4(X) = 1 - 3X^2,$
- $k = 4, m = 2$: $d_4 = 2, p_5(X) = 1 - X^2 - 4X^3$
- $k = 5, m = 4$: $d_5 = 2, p_6(X) = 1 - X - X^2 - X^3,$
- $k = 6, m = 4$: $d_6 = 0, p_7(X) = p_6(X).$

Thus we have computed the following minimal linear recurrence of the sequence $(s_j)_{0 \leq j \leq 6}$ for $r = 0, \dots, 3$

$$s_{r+3} = s_r + s_{r+1} + s_{r+2}.$$

It remains to prove that the Algorithm of Berlekamp-Massey terminates. Moreover, we have to show that we only have to consider more than $2n$ sequence elements to find a generating polynomial for a linear recurrence sequence with recurrence length n . We show that the construction must reproduce the same generating polynomial for any $k > 2n$.

If $k \geq 2n$, then $k + 1 - L_k \geq n + 1$, because $L_k \leq (k + 1)/2$ (observe that $L_k \leq L_{k+1} \leq k + 1 - L_k$). Hence the construction must yield a recursion of the same length for all $k \geq 2n$.

Assume that c_1, \dots, c_{L_k} and c'_1, \dots, c'_{L_k} are the coefficients of two different recursions which generate the sequence $(s_j)_{0 \leq j < k}$. Then we consider $c''_i = c_i - c'_i$ for $1 \leq i \leq L_k$. Let c''_{L_k-j} be the first of those coefficients which is non zero and define for $0 \leq i \leq j$

$$d_i = (c''_{L_k-j})^{-1} \cdot c''_{L_k-j+i}.$$

Then we have for an appropriate index l which will be defined later and for $L_k - j + l \leq k - 1$ (i.e. $l \leq k - 1 - L_k + j = k^* - 1$)

$$\begin{aligned} \sum_{i=0}^j d_i \cdot s_{l-i} &= (c''_{L_k-j})^{-1} \cdot \sum_{i=0}^j (c_{L_k-j+i} - c'_{L_k-j+i}) \cdot s_{l-i} \\ &= (c''_{L_k-j})^{-1} \cdot \sum_{i=L_k-j}^{L_k} (c_i - c'_i) \cdot s_{L_k-j+l-i} \\ &= (c''_{L_k-j})^{-1} \cdot \sum_{i=0}^{L_k} (c_i - c'_i) \cdot s_{L_k-j+l-i} \\ &= 0. \end{aligned}$$

We will explain these steps: the first equation follows just from the definition of the d_i and the c''_i . Then we have changed the running index by $L_k - j$. The next equality follows by the fact that $c''_i = 0$ for $i = 1, \dots, L_k - j - 1$. Hence we have $L_{k^*} \leq j$. But on the other hand

$$L_{k^*} = \max\{L_{k^*-1}, k^* - L_{k^*-1}\} \geq k^* - L_{k^*-1} = k - L_k + j - L_{k^*-1}.$$

For proving this equality, note that $L_{k^*-1} \leq L_{k^*} \leq j < L_k \leq n$ and we have found the contradiction $L_{k^*} > j$ for $k \geq 2n$. Therefore we have proven that the recursion which generates the sequence $(s_j)_{j \leq k-1}$ is uniquely determined.

Let us come back to the solution of the linear system given in (7.1). We have shown how we can compute the generating polynomial $p_\infty(X)$ for the sequence $(s_j) = (\underline{u}, A^j \cdot \underline{b})_{j \geq 0}$ for a random vector $\underline{u} \in \mathbb{F}^n$. The connection between the generating polynomial of this sequence and the minimal polynomial $f(X)$ of the endomorphism φ defined in (7.2) is given in the next lemma.

7.1. Lemma *Let $p_\infty(X)$ be the generating polynomial for the sequence $(s_j)_{j \geq 0} = (\underline{u}, A^j \cdot \underline{b})_{j \geq 0}$, where $\underline{u} \in \mathbb{F}^n$ is chosen at random. Then $p_\infty(X)$ is a divisor of the minimal polynomial of the endomorphism φ defined in (7.2).*

Proof:

One way of determining the minimal polynomial $f(X)$ of φ is therefore to compute $p_\infty(X)$ for a randomly chosen vector $\underline{u} \in \mathbb{F}^n$ and to hope that $p_\infty(X) = f(X)$. We try to estimate the probability for a random \underline{u} that $p_\infty(X) = f(X)$ holds indeed.

7.2. Lemma *The probability that for a random vector $\underline{u} \in \mathbb{F}^n$ the generating polynomial of the linear recurrence sequence $(s_j)_{j \geq 0} = (\underline{u}, A^j \cdot \underline{b})_{j \geq 0}$ coincides with the minimal polynomial $f(X)$ of the endomorphism φ given in (7.2) is approximately $\frac{1}{\log(n)}$.*

Proof: It is easy to see that for $\rho = X + f(X)\mathbb{F}[X]$ the map

$$\begin{aligned} S = \text{span}\{A^i \cdot b \mid i \in \mathbb{Z}_{\geq 0}\} &\longrightarrow \mathbb{F}[X]/f(X)\mathbb{F}[X] =: R \\ A^i \cdot b &\longmapsto \rho^i \end{aligned}$$

is an isomorphism. If we moreover define

$$\eta(1) \left(\sum_{i=1}^d \lambda_i \cdot \rho^{i-1} \right) = \lambda_d$$

and for $\alpha, \beta \in R$

$$\eta(\alpha)(\beta) = \eta(1)(\alpha \cdot \beta),$$

then the map

$$\begin{aligned} R &\longrightarrow \text{Hom}_{\mathbb{F}}(R, \mathbb{F}) \\ \alpha &\longmapsto \eta(\alpha)(\cdot) \end{aligned}$$

is an isomorphism. Hence, in order to count the number of linear recurrences $(\underline{u}, A^j \cdot \underline{b})_{j \geq 0}$ with minimal polynomial $f(X)$ we may, as well, count the number of linear recurrences $(\eta(\alpha)(\rho^j))_{j \geq 0}$ with minimal polynomial $f(X)$. Let $h(X)$ be the minimal polynomial of this recurrence, then we have for $j \geq 0$

$$\begin{aligned} 0 &= h(\eta(\alpha)(\rho^j)) = \eta(\alpha)(h(\rho^j)) \\ &= \eta(1)(\alpha \cdot h(\rho^j)), \end{aligned}$$

which means that $\alpha \cdot h(\rho^j) = 0$. If α is a unit in R , then $h(\rho^j) = 0$. But then $f(X)$ is a divisor of $h(X)$ and hence $f = h$.

One can show that the fraction of units in R is at least $1/(6j)$, where j is determined as the least exponent such that $(q := \#\mathbb{F})$

$$\deg(f) \leq 1 + q + q^2 + \dots + q^j.$$

Thus we have $j \approx \log(n)$ and the probability for finding a vector \underline{u} which determines the minimal polynomial $f(X)$ is approximately $\frac{1}{\log(n)}$.

We return to our method for solving the equation and analyze its complexity. We pick $\underline{u} \in \mathbb{F}^n$ at random, we determine the first $2n+1$ elements of the linear recurrence $(\underline{u}, A^i \cdot \underline{b})$ which requires $O(n(\omega + n))$ operations. Note that the vector u can be chosen sparse (i.e. of the form $(0, \dots, 0, 1, 0, \dots, 0)$), which improves the running time in practice. The construction of the minimal polynomial using the Berlekamp-Massey algorithm requires $O(n^2)$ operations. In order to determine the characteristic polynomial of A , one must choose several \underline{u} and compute the least common multiple of the resulting polynomials. Hence we have proven the following Proposition.

7.1. Proposition *Let $A \in \mathbb{F}^{n \times n}$ be non singular with ω non zero entries, and let $\underline{b} \in \mathbb{F}^n$. There is a probabilistic algorithm which calculates a solution of $A \cdot \underline{x} = \underline{b}$ in expected time $O(n(\omega + n) \cdot \log n)$.*

7.4 The Lanczos Algorithm

The third algorithm which can be used for solving sparse linear systems over \mathbb{F}_2 is the Lanczos algorithm. This algorithm was originally described for the real numbers [?], but later it was generalized to finite fields. A description of the Block Lanczos algorithm for the field \mathbb{F}_2 can be found in [Mo95], a version for finite prime fields of characteristic greater two is described in [De97]. We assume without loss of generality that we want to solve the linear system (7.1) with a symmetric matrix $A \in \mathbb{F}^{n \times n}$ and a n -dimensional vector \underline{b} .

invertible W ??

The idea of the Lanczos algorithm is as follows: Assume that we find an invertible matrix $W = [\underline{w}_1, \dots, \underline{w}_n]$ such that $W^t \cdot A \cdot W = D$, where D is a diagonal matrix of full rank. Then a solution \underline{y} of the linear system $D \cdot \underline{y} = W^t \cdot \underline{b}$ leads to a solution of (7.1): By the defining equation for the matrix D , the vector $\underline{x} = W \cdot \underline{y}$ is a solution of (7.1). Since D is a diagonal matrix, a solution of $D \cdot \underline{y} = W^t \cdot \underline{b}$ can be computed very easy. Therefore we have reduced the solution of (7.1) to the determination of a suitable matrix W . The Lanczos algorithm uses the linear independency of the column vectors of the so called Krylow-Matrix to find such a matrix W . We describe an iterative method for computing the columns \underline{w}_i of this matrix W .

The algorithm starts by setting $\underline{w}_1 = \underline{b}$. Then we iterate

$$\underline{w}_i = A \cdot \underline{w}_{i-1} - \sum_{j=0}^{i-1} c_{ij} \underline{w}_j \quad \text{with} \quad c_{ij} = \frac{\underline{w}_j^t \cdot A^2 \cdot \underline{w}_{i-1}}{\underline{w}_j^t \cdot A \cdot \underline{w}_j}, \quad (7.5)$$

until we find an index m with $\underline{w}_m = \underline{0}$. In [Mo95], it is shown that for $i \neq j$ we have $\underline{w}_j^t \cdot A \cdot \underline{w}_i = \underline{0}$. Using this fact, one can show that for $j < i - 2$

$$\begin{aligned} \underline{w}_j^t \cdot A^2 \cdot \underline{w}_{i-1} &= (A \cdot \underline{w}_j)^t \cdot A \cdot \underline{w}_{i-1} \\ &= \left(\underline{w}_{j+1} + \sum_{k=0}^j c_{j+1,k} \underline{w}_k \right)^t \cdot A \cdot \underline{w}_{i-1} = \underline{0}, \end{aligned}$$

and therefore $c_{ij} = 0$ for $j < i - 2$. Thus equation (7.5) simplifies for $i \geq 3$ to

$$\underline{w}_i = A \cdot \underline{w}_{i-1} - c_{i,i-1} \underline{w}_{i-1} - c_{i,i-2} \underline{w}_{i-2}.$$

If the iteration stops with $\underline{w}_m = \underline{0}$, then

$$\underline{x} = \sum_{j=0}^{m-1} \frac{\underline{w}_j^t \cdot \underline{b}}{\underline{w}_j^t \cdot A \cdot \underline{w}_j} \cdot \underline{w}_j$$

is a solution of the linear system (7.1). It should be noted that solution can be computed during the iteration.

These ideas can be combined to the following compact description of the Lanczos algorithm. We start with

$$\begin{aligned} \underline{w}_1 &= \underline{b}, \\ \underline{v}_2 &= A \cdot \underline{w}_1, \\ \underline{w}_2 &= \underline{v}_2 - \frac{\langle \underline{v}_2, \underline{v}_2 \rangle}{\langle \underline{w}_1, \underline{v}_2 \rangle} \cdot \underline{w}_1 \end{aligned}$$

and iterate for $i \geq 2$

$$\begin{aligned} \underline{v}_{i+1} &= A \cdot \underline{w}_i, \\ \underline{w}_{i+1} &= \underline{v}_{i+1} - \frac{\langle \underline{v}_{i+1}, \underline{v}_{i+1} \rangle}{\langle \underline{w}_i, \underline{v}_{i+1} \rangle} \cdot \underline{w}_i - \frac{\langle \underline{v}_{i+1}, \underline{v}_i \rangle}{\langle \underline{w}_{i-1}, \underline{v}_i \rangle} \cdot \underline{w}_{i-1}. \end{aligned}$$

Der Algorithmus stoppt, wenn ein \underline{w}_m gefunden wird, das A -konjugiert zu sich selber ist ($\langle \underline{w}_m, A\underline{w}_m \rangle = 0$). Dies passiert für ein $m \leq n$. Falls $\underline{w}_m = 0$ ist, dann ist

$$x = \sum_{i=0}^{m-1} \frac{\langle \underline{w}_i, b \rangle}{\langle \underline{w}_i, \underline{v}_{i+1} \rangle} \underline{w}_i \quad (7.6)$$

eine Lösung von Gleichung (7.1). Wird solch ein \underline{w}_m nicht gefunden, dann liegt b nicht in dem von den Spalten von A erzeugten Unterraum.

Verwendet man den Lanczos-Algorithmus zum Lösen linearer Gleichungssysteme über endlichen Primkörpern, so überträgt man die Formeln (7.1) - (7.6) auf die Situation in endlichen Körpern. Dabei muss man einige Probleme lösen :

- In endlichen Körpern folgt (im Gegensatz zur Situation in \mathbb{R} , wo A positiv definit ist) aus $w_j^t A w_j = 0$ nicht notwendig, da $w_j = 0$ ist. Löst man Gleichungssysteme über Primkörpern \mathbb{F}_p mit großer Charakteristik ($p \gg n$), so stellt die sogenannte Selbstkonjugiertheit kein Problem dar. Mögliche Lösungen für dieses Problem in Körpern mit kleiner Charakteristik finden sich in ([?], [LaOd91]).
- Mit dem Lanczos Verfahren kann man nicht nur eine Lösung $Bx = u$, sondern auch mehrere Lösungen $Bx_j = u_j$, $1 \leq j \leq r$, berechnen. In [LaOd91] wird ein Verfahren beschrieben, das dies ermöglicht. Man berechnet $z_j = B^t u_j$ und startet den Algorithmus mit $w_0 = z_1$. Zur Berechnung der r verschiedenen Lösungen genügt es Schritt (7.6) durch

$$x_j = \sum_{l=0}^{m-1} \frac{\langle w_l, z_j \rangle}{\langle w_l, v_{l+1} \rangle} w_l, \quad 1 \leq j \leq r$$

zu ersetzen. Einen Beweis dafür findet sich in [LaOd91].

18. Example We try to solve the linear system

$$\begin{pmatrix} 1 & 3 & -1 & 1 & 1 & 1 \\ 2 & 4 & 2 & 1 & -1 & -1 \\ 3 & 1 & -1 & -1 & 2 & 1 \\ 4 & -1 & 4 & 3 & -2 & -1 \\ 5 & -2 & -1 & 2 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

in the finite prime field $\mathbb{F} = \mathbb{Z}/29\mathbb{Z}$. First we transform this system to a square system

$$\begin{pmatrix} 26 & 0 & 11 & 22 & 12 & 3 \\ 0 & 2 & 2 & -1 & -3 & -1 \\ 11 & 2 & 23 & 12 & 13 & 20 \\ 22 & -1 & 12 & 16 & -2 & -2 \\ 12 & -3 & 13 & -2 & 19 & 9 \\ 3 & -1 & 20 & -2 & 9 & 5 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 9 \\ 2 \\ -3 \\ 2 \\ 6 \\ 3 \end{pmatrix}.$$

Dabei wird $W = [w_0, \dots, w_4]$ berechnet als

$$W = \begin{pmatrix} 9 & 20 & 2 & 6 & 0 \\ 2 & 23 & 17 & 9 & 16 \\ 26 & 8 & 15 & 5 & 18 \\ 2 & 19 & 6 & 6 & 14 \\ 6 & 25 & 8 & 2 & 3 \\ 3 & 15 & 18 & 21 & 13 \end{pmatrix}.$$

Dann gilt

$$W^t B^t B W y = \begin{pmatrix} 19 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 28 & 0 \\ 0 & 0 & 0 & 0 & 24 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} 27 \\ 0 \\ 5 \\ 28 \\ 5 \end{pmatrix} = W^t B^t u.$$

Somit findet man als Lsungen

$$y = \begin{pmatrix} 6 \\ 0 \\ 10 \\ 1 \\ 28 \end{pmatrix} \quad \text{und damit} \quad W y = x = \begin{pmatrix} 22 \\ 1 \\ 3 \\ 6 \\ 28 \\ 3 \end{pmatrix}.$$

Bibliography

- [AlPo93] W.R. Alford, C. Pomerance: *Implementing the self initializing quadratic sieve on a distributed network*, Manuscript, 1993.
- [AtMo92] A.O.L. Atkin, F. Morain: *Finding Suitable Curves For The Elliptic Curve Method Of Factorization*, Preprint, Jan. 1992.
- [AtGrLeLe95] D. Atkins, M. Graff, A.K. Lenstra, P.C. Leyland: *THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE*, Proceedings Asiacrypt'94, LNCS 917, (1995) 263-277.
- [BaSh96] E. Bach, J. Shallit: *Algorithmic Number Theory*, MIT Press, 1996.
- [Be93] F.D. Berger: *ECM — Faktorisieren mit elliptischen Kurven*, Diploma Thesis, Universität des Saarlandes, 1993.
- [Br80] R.P. Brent: *An Improved Monte Carlo Factorization Algorithm*, Nordisk Tidsskrift för Informationsbehandling (BIT) **20**, 1980, 176-184.
- [BrPo81] R.P. Brent, J.M. Pollard: *Factorization of the eighth Fermat number*, Math. Computation **36**, 1981, 627-630.
- [BrRi92] R.P. Brent, H.J.J. te Riele: *Factorizations of $a^n \pm 1$, $13 \leq a \leq 100$* , CWI Report, Department of Numerical Mathematics NM-R9212, Juni 1992
- [BuMü92] J. Buchmann, V. Müller: *Primality Proving*, Informatik Fachbericht 02/92, Universität des Saarlandes, 1992.
- [BuLePo93] J.P. Buhler, H.W. Lenstra Jr., C. Pomerance: *Factoring integers with the number field sieve*, Lecture Notes in Mathematics, **1554**, 1993, 50-94.
- [Bru66] N.G. de Bruijn: *On the number of integers $\leq x$ and free of prime factors $> y$* , Indag. Math., **38**, 1966, 239-247.
- [Br81] J. Brillhart: *Fermat's factoring method and its variants*, Congressus numeratum, **32**, 1981, 29-48.
- [Co93] H. Cohen: *A Course in Computational Algebraic Number Theory*, Graduate Texts in Mathematics **138**, Springer-Verlag, 1993.
- [Cv93] J.M. Couveignes: *Computing a square root for the the number field sieve*, Lecture Notes in Mathematics, **1554**, 1993, 95-102.

- [RSA130] J. Cowie, M. Elkenbracht-Huizing, W. Furmanski, A.K. Lenstra, P.L. Montgomery, D. Weber, J. Zayer: *Factorization of RSA-130*, 1995, available on <http://www.npac.syr.edu/factoring/status.html>
- [De93] T.F. Denny: *Faktorisieren mit dem quadratischen Sieb*, Diploma Thesis, Universität des Saarlandes, 1993.
- [De97] T.F. Denny: *Der Lanczos Algorithmus*, PHD Thesis, Universität des Saarlandes, 1997.
- [DeDoLeMa93] T.F. Denny, B. Dodson, A.K. Lenstra, M.S. Manasse: *On the factorization of RSA-120*, Advances in Cryptology — Crypto '93, LNCS 773, 1993, 166-174.
- [GoSa83] R. Gold, J. Sattler: *Modifikationen des Pollard-Algorithmus*, Computing **30**, 1983, 77-89.
- [Kn74] D. Knuth: *The Art of computer programming*, vol. 2, Seminumerical algorithms, Addison Wesley, 1973.
- [Ko87] N. Koblitz: *A course in number theory and cryptography*, Springer-Verlag, 1987.
- [LaOd91] B.A. LaMacchia, A.M. Odlyzko: *Solving large sparse systems over finite fields*, Advances in Cryptology – Crypto '90, LNCS 537, 1991, 109-133.
- [LePo31] D.H. Lehmer, R.E. Powers: *On Factoring Large Numbers*, Bulletin American Mathematical Society, **37**, 1931, 770-776.
- [LLMP90] A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse, J.M. Pollard: *The number field sieve*, Proc. 22nd ACM Symp. on Theory of Computing, 1990, 564-572.
- [LLMP93] A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse, J.M. Pollard: *The factorization of the ninth Fermat number*, Mathematics of Computation, **61**, 1993, 319-349.
- [LeMa91] A.K. Lenstra, M.S. Manasse: *Factoring with two large primes*, Advances in Cryptology — Eurocrypt '90, 1990, 72-82.
- [Le87] H.W. Lenstra Jr.: *Factoring integers with elliptic curves*, Ann. of Math., **126**, 1987, 649-673.
- [LeLeLo82] A.K. Lenstra, H.W. Lenstra Jr., L. Lovasz: *Factoring polynomials with rational coefficients*, Math. Ann., **261**, 1982, 515-534.
- [Mo87] P.L. Montgomery: *Speeding the Pollard and Elliptic Curve Method of Factorization*, Mathematics of Computation, **177**, 1987, 243-264.
- [Mo92] P.L. Montgomery: *An FFT Extension of the Elliptic Curve Method of Factorization*, PHD Thesis, University of California, 1992.
- [Mo95] P.L. Montgomery: *A block Lanczos algorithm for finding dependencies over $GF(2)$* , Advances in Cryptology – Eurocrypt '95, LNCS 921, 1995, 106-120.
- [MoBr75] M.A. Morrison, J. Brillhart: *A method for factoring and the factorization of F_7* , Math. Comp., **29**, 1975, 183-205.

- [Mü96] A. Müller: *Eine FFT-Continuation für die elliptische Kurvenmethode*, Diploma Thesis, Universität des Saarlandes, 1996.
- [Po74] J.M. Pollard: *Theorems on factorization and primality testing*, Proc. Camb. Phil. Soc. **76**, 1974, 521-528.
- [Po75] J.M. Pollard: *A Monte Carlo method for factorization*, BIT **15**, 1975, 331-334.
- [Po91] J.M. Pollard: *The Lattice Sieve*, Lecture Notes in Mathematics, **1554**, 1991, 43-49.
- [Po85] C. Pomerance: *The quadratic sieve factoring algorithm*, Advances in Cryptology, Springer Lecture Notes, **209**, 1985, 169-182
- [Ri85] H. Riesel: *Prime Numbers and Computer Methods for Factorization*, Birkhäuser Verlag (Progress in Mathematics 57), 1985
- [RSA78] R.L. Rivest, A. Shamir, L. Adleman: *A method for obtaining digital signatures and public-key cryptosystems*, Comm. ACM, **21**, 1978, 120-126.
- [RSA] RSA Data Security Inc., sci.crypt, May 18, 1991: information available by sending electronic mail to `challenge-rsa-list@rsa.com`.
- [Sey87] M. Seysen: *A probabilistic factorization algorithm with quadratic forms of negative discriminant*, Math. Comp., **48**, 1987, 757-780
- [Sh72] D. Shanks: *Five Number Theoretic Algorithms*, Congressus Numerantium No. VII, Proceedings 2nd Manitoba Conference on Numerical Mathematics, 1972, 51-70.
- [Si87] R.D. Silverman: *The multiple quadratic sieve*, Mathematics of Computation, **48**, 1987, 329-339.
- [So95] T. Sosnowski: *Faktorisieren mit dem Quadratischen Sieb auf dem Hypercube*, Diploma Thesis, Universität des Saarlandes, 1995.
- [We76] E. Weiss: *Algebraic number theory*, McGraw-Hill, New York, 1963, reprint Chelsea, 1976.
- [Wi86] D.H. Wiedemann: *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory, IT **32**, 1986, 54-62
- [Wi82] H.C. Williams: *A $(p+1)$ Method of Factoring*, Mathematics of Computation, **39**, 1982, 225-234.
- [Za95] J. Zayer: *Faktorisieren mit dem Number Field Sieve*, Dissertation, Universität des Saarlandes, 1995.