NTNU – Trondheim
Norwegian University of
Science and Technology

Number Field Sieve

Ruben Grønning Spaans

Master of Science in Mathematics
Submission date: June 2013
Supervisor: Kristian Gjøsteen, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Acknowledgements

This master thesis was carried out within the Department of Mathematical Sciences (IME) at the Norwegian University of Science and Technology (NTNU) between August 2012 and May 2013.

I would like to thank my supervisor Kristian Gjøsteen for his valuable feedback and suggestions.

Lastly, I want to thank the domain of integer factorization for being a very interesting challenge to work on. It has stayed a fascinating topic for me since my early teens.

Abstract

The Number Field Sieve (NFS) is the fastest known general method for factoring integers having more than 120 digits. In this thesis we will will study the algebraic number theory that lies behind the algorithm, describe the algorithm in detail, implement it and use our implementation to perform some experiments.

Sammendrag

Algoritmen "Number Field Sieve" (tallkroppssålden) er den raskeste generelle algoritmen for faktorisering av heltall med flere enn 120 sifre som vi kjenner i dag. I denne avhandlingen kommer vi til å studere matematikken (algebraisk tallteori) som ligger til grunn for algoritmen, beskrive algoritmen i detalj, implementere denne samt utføre eksperimenter med vår implementasjon.

Contents

Chapter 1

Introduction

1.1 Goal

The goal of this thesis is to study the Number Field Sieve (NFS) algorithm, including the mathematics required in order to understand the algorithm. The mathematics mainly consists of algebraic number theory.

In addition we will implement the complete algorithm and perform some experiments.

1.2 Background

The Number Field Sieve (NFS) is an algorithm for factoring integers, and it's currently the fastest known algorithm for factoring integers of more than 120 digits.

A more specialized version of the algorithm exists, and was actually developed before the general variant. This variant is usually referred to as the Special Number Field Sieve (SNFS), and it is capable of factoring numbers of the form $r^e \pm s$, where r and s are small integers, and e is an integer which is allowed to be large. One of the early factorization successes of the SNFS was that of the 9th Fermat number, $2^{512}+1$ which was fully factored in 1991 [len91].

The generalised variant is sometimes called "General Number Field Sieve" (GNFS), but we will refer to the general algorithm as the Number Field Sieve (NFS) throughout this thesis.

1.3 The RSA algorithm and integer factorization

The RSA algorithm for public-key encryption is based on the fact that it is trivial to multiply two integers, but significantly more difficult to perform the reverse operation: given a product, find the factors.

The person (let's call her Alice) who wants to send, receive and decrypt messages generates a private and a public key. The public key is distrubuted freely, while Alice keeps the private key secret. Anyone (Bob, for example) who wishes to send encrypted messages to Alice can use the public key to encrypt their message. Alice is the only one who can decrypt and read these messages by using her private key.

The first step in the algorithm is to generate the private and public keys. This is done by performing the following steps:

1. Choose two distinct prime numbers p and q, both having roughly the same number of digits.

2. Compute $n = p \cdot q$.

3. Compute $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$ where ϕ is Euler's totient function.

4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n))) = 1$.

5. Find the unique integer d satisfying $1 < d < \phi(n)$ and $d^{-1} \equiv e \pmod{\phi(n)}$.

The public key consists of the values n and e, and the private key consists of the values n and d. Naturally, the factorization of n and the value of $\phi(n)$ are also kept secret.

Assume that Bob wants to send a message, and that the message can somehow be represented as an integer m such that $0 \leq m < n$. The encrypted text (the ciphertext) is then calculated by

$$c \equiv m^e \pmod{n}.$$

Alice can retrieve the original message by computing

$$m \equiv c^d \pmod{n}.$$

There are a number of possible attacks against the RSA algorithm, but in particular the private key can be directly obtained if we can factor n into p and q. When p and q are known, we can easily calculate $\phi(n)$ and calculate d which enables us to decrypt messages. Therefore it is important to choose n large enough so that it is infeasible to factor it.

1.4 Organization of this thesis

This chapter contains the introduction.

In Chapter 2 the Quadratic Sieve (QS) algorithm for factoring integers is described. This chapter can be skipped, but it is recommended if the reader is not familiar with the algorithm. The exception is Section 2.2 which should be read as it contains some necessary definitions.

In Chapter 3 the necessary algebra needed to understand the NFS is reviewed. It can be skipped if the reader is familiar with field theory, number fields and factorization of ideals in rings of algebraic integers.

Chapter 4 contains a thorough description of the NFS algorithm.

Chapter 5 contains algorithms for subtasks that are performed by the NFS. It's not required reading, but is recommended for anyone who wishes to implement the NFS.

Our implementation is described in Chapter 6. It also contains many implementation tips for those who would like to implement the algorithm.

In Chapter 7 we describe some experiments we conducted with our NFS implementation.

Finally, Chapter 8 contains the conclusion of the thesis.

1.5 Some notes on notation

In this section we clarify the use of our notation where ambiguity can occur.

The symbol $\subset$ can mean either proper subset or any subset, depending on the author. In this thesis we will use the following symbols with the following meanings:

$A \subset B$	A is a proper subset of B
$A \subseteq B$	A is a subset of B
$A \not\subset B$	A is not a proper subset of B
$A \not\subseteq B$	A is not a subset of B

Throughout this thesis we will refer to numbers and their magnitude. The magnitude of a given integer n is commonly given by the number of bits in its binary expansion, or the number of digits in its decimal expansion. We will often refer to the number of digits of an integer n. When we say digits we always refer to the number of digits in the decimal expansion.

The *Quadratic Sieve* and *Number Field Sieve* algorithms will mainly be referred to as QS and NFS respectively.

Chapter 2

Quadratic sieve

Before describing the NFS, we will describe the *Quadratic Sieve* algorithm which is a much simpler algorithm that uses many of the same ideas as the NFS.

The QS is currently the second fastest method known for factoring integers, and is the algorithm of choice for integers between around 50 and 120 digits. For smaller integers Pollard's *rho method* or Lenstra's *elliptic curve factorization method* (ECM) are preferred, while for larger integers the NFS is the best choice.

2.1 Quadratic residues

A significant part of the QS algorithm is to find integers $u \not\equiv v \pmod{n}$ satisfying

$$u^2 \equiv v^2 \pmod{n}. \tag{2.1}$$

This idea is based in the idea that we can write the factors of n as

$$n = (u - v)(u + v).$$

From this we get

$$u^2 - v^2 = n$$

from which we can get the congruence (2.1). Having found such $u \neq v$ that satisfies this congruence there is a chance that we can find a non-trivial factor $\gcd(n, u - v)$. If we wish to factor $n = 1649$, $u = 114$ and $v = 80$ satisfy (2.1):

$$114^2 \equiv 80^2 \pmod{1649}$$

and $\gcd(1649, 114-80) = 17$ which is a non-trivial factor of 1649. Indeed, the factorization of 1649 into primes is $1649 = 17 \cdot 97$.

2.2 The sieve

In order to find u, v that satisfies the congruence in (2.1) we use a *sieving process* to find *smooth* integers. We will define smooth integers and a few more terms before describing the sieve process. The following definitions are common for both the QS and NFS methods.

Definition 2.2.1. *A positive integer n is B-*smooth *if none of the prime factors of n is larger than B.*

Example 2.2.1. $20 = 2 \cdot 2 \cdot 5 \cdot 5$ is 5-smooth, while $21 = 3 \cdot 7$ isn't.

Definition 2.2.2. *A* factor base *P is a set of prime numbers less than or equal to B (not necessarily every eligible prime number).*

Definition 2.2.3. *An* exponent vector *is a vector of $|P|$ nonnegative integers e_i which can be used to represent a B-smooth number $m = \prod_{i=1}^{|P|} p_i^{e_i}$.*

Example 2.2.2. Let $P = \{2, 3, 5, 7, 11\}$. Here are some examples of 11-smooth numbers represented by exponent vectors. Assume that each prime $p \in P$ is considered in increasing order.

$$\begin{aligned}
6 &= 2 \cdot 3 = 2^1 \cdot 3^1 \Rightarrow (1, 1, 0, 0, 0) \\
7 &= 7 = 7^1 \Rightarrow (0, 0, 0, 1, 0) \\
10 &= 2 \cdot 5 = 2^1 \cdot 5^1 \Rightarrow (1, 0, 1, 0, 0) \\
64 &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^6 \Rightarrow (6, 0, 0, 0, 0) \\
32340 &= 2 \cdot 2 \cdot 3 \cdot 5 \cdot 7 \cdot 7 \cdot 11 = 2^2 \cdot 3^1 \cdot 5^1 \cdot 7^2 \cdot 11^1 \Rightarrow (2, 1, 1, 2, 1)
\end{aligned}$$

In both the QS and NFS algorithms, the exponent vector is usually augmented to also hold the sign of the smooth number. In this case we add -1 to the factor base.

Example 2.2.3. Let the factor base P contain the elements $\{-1, 2, 3, 5, 7, 11\}$. Here are some additional examples of 11-smooth numbers and their respective exponent vectors.

$$\begin{aligned}
6 &= 2 \cdot 3 = 2^1 \cdot 3^1 \Rightarrow (0, 1, 1, 0, 0, 0) \\
-7 &= (-1) \cdot 7 = (-1)^1 \cdot 7^1 \Rightarrow (1, 0, 0, 0, 1, 0) \\
-32340 &= (-1) \cdot 2 \cdot 2 \cdot 3 \cdot 5 \cdot 7 \cdot 7 \cdot 11 = (-1)^1 \cdot 2^2 \cdot 3^1 \cdot 5^1 \cdot 7^2 \cdot 11^1 \Rightarrow (1, 2, 1, 1, 2, 1)
\end{aligned}$$

It should be obvious that all elements e_i of an exponent vector are even if and only if $m = \prod_{i=1}^{|P|} p_i^{e_i}$ is a square.

2.2.1 The sieving process

In this section we describe the basic variant of the QS algorithm which uses the polynomial $x^2 - n$ to find solutions to the congruence $u^2 \equiv v^2 \pmod{n}$. The goal is to find a set of integers $x_1, x_2, \ldots$ such that for each i, $x_i^2 - n$ is B-smooth and the product $\prod(x_i^2 - n)$ is a square. Then

$$\prod x_i^2 \equiv \prod (x_i^2 - n) \pmod{n}, \tag{2.2}$$

and hopefully the same set of numbers satisfy

$$\prod x_i \not\equiv \sqrt{\prod (x_i^2 - n)} \pmod{n}, \tag{2.3}$$

leading to a non-trivial factor. As we will see soon, the B-smoothness of each $x_i^2 - n$ allows us to use linear algebra to find such a subset.

Assume we have a factor base of size K, with $K-1$ primes less than B, as well as the unit -1. The aim of the sieve phase is to find at least $K+1$ integers x_i, enabling us to find a subset satisfying (2.2). From each $(x_i^2 - n)$ we obtain an exponent vector. In order to find a square we can find a linear combination of the $K+1$ exponent vectors that sum to 0 modulo 2. This resulting exponent vector will have all elements even and hence we have a square.

The actual sieving can be done as follows. Let $N = \lceil \sqrt{n}\, \rceil$, this value will be the "center" of our sieve interval. We initialize an array which has one element for each integer a in the interval $N - M \leq a \leq N + M$ for some bound $M > 0$. Initialize each element with the value $a^2 - n$. For each prime p in the factor base and for each a within our interval, we check if p divides $a^2 - n$. If it does, we divide the array element by p^k, the highest prime power that divides $a^2 - n$.

This procedure is done efficiently by processing each p in turn. First, check if the array element is negative. If it is, update the exponent vector accordingly and set the array element to its absolute value. Then, solve the equation $a^2 - n \equiv 0 \pmod{p}$ which has two solutions for $0 \leq a < n$. Find the two smallest values of $a_1, a_2 \geq N - M$ that satisfy the equation. Then, divide array element $(a_i + bp)^2 - n$ by p^k for $i = 1, 2$ and for all $b \geq 0$ such that $a_i + bp \leq N + M$. All elements that are equal to 1 after this procedure are divisible by primes less than or equal to B, so they are the B-smooth numbers we are searching for.

If we have less than $K+1$ smooth integers after this procedure, we need to increase the bound M and perform sieving in the new intervals.

We end this section with a non-rigorous discussion about the density of the smooth numbers. We will assume (without proof) that a small integer is more likely to be smooth than a large integer. Therefore the sieving interval is chosen so that it contains as small integers as possible. The center of the sieve interval is $N = \lceil \sqrt{n}\, \rceil$, which is close to the value of x that minimizes $x^2 - n$. This interval is extended in the positive and the negative directions by an equal amount (the M bound mentioned above). In this way we maximize the density of smooth numbers within an interval of size $2M + 1$.

If we only considered positive $x^2 - n$ ($x \geq \lceil \sqrt{n}\, \rceil$) we could get rid of -1 from the factor base, but then we would need to include the interval from $N + M + 1$ to $N + 2M$ which has lower density of smooth numbers than $N - M$ to $N - M - 1$.

2.3 The linear algebra

We form a matrix A where row i consists of an exponent vector

$$\mathbf{e} = (e_1, e_2, \ldots, e_K),$$

where the e_i are the prime exponents in the factorization of $x_i^2 - n$. That is,

$$x_i^2 - n = \prod_{i=1}^{K} p_i^{e_i}$$

where p_i are the elements of the factor base (where one element is the unit -1). We seek a non-zero vector $\mathbf{y}$ satisfying the system of equations

$$\mathbf{y}^{\intercal} A \equiv 0 \pmod 2. \tag{2.4}$$

A solution to (2.4) will give us a set of integers x, each having an exponent vector which describe the factorization of $x^2 - n$ into primes in our factor base. Denote this set $\mathcal{S}$. This set $\mathcal{S}$ satisfies

$$\prod_{x \in \mathcal{S}} x^2 \equiv \prod_{x \in \mathcal{S}} (x^2 - n) \pmod n, \tag{2.5}$$

and both sides of the congruence are squares.

2.4 Square roots and factorization

When we have a subset $\mathcal{S}$ of integers such that each $x \in \mathcal{S}$ leads to a B-smooth integer $x^2 - n$, we can calculate

$$\sqrt{\prod_{x \in \mathcal{S}} (x_i^2 - n)} \pmod n \tag{2.6}$$

from the known factorization of $x^2 - n = \prod p_i^{e_i}$ for each $x \in \mathcal{S}$ by halving the prime exponents in the final product. If we let

$$\begin{aligned} u &= \prod_{x \in \mathcal{S}} x \pmod n \quad \text{and} \\ v &= \sqrt{\prod_{x \in \mathcal{S}} (x_i^2 - n)} \pmod n, \end{aligned}$$

we can calculate $g = \gcd(n, u - v)$. If g is a non-trivial factor, then we are done and g and n/g are two non-trivial factors. If g is 1 or n we need to find another solution to (2.4) which leads to a different linear combination of exponent vectors leading to a different square. If we run out linear combinations, more sieving is required.

Chapter 3

Mathematical preliminaries

The purpose of this chapter is to go through the mathematics needed in order to understand the NFS, and list all the needed definitions and results.

In the NFS we will work with numbers of the form $a - b\alpha$ with $a, b \in \mathbb{Z}$, where $\alpha \in \mathbb{C}$ is a root of an irreducible monic polynomial $f(x) \in \mathbb{Z}[x]$. We recall that a monic polynomial has 1 has its highest degree coefficient. These numbers belong to a larger class of numbers called a *number ring*, which contains elements of the form $a_0 + a_1\alpha + a_2\alpha^2 + \cdots$ with $a_i \in \mathbb{Z}$. Number rings will be defined in section 3.2. The reason for looking at numbers of the form $a - b\alpha$ rather than $a + b\alpha$ is that the norm calculations that we will encounter later will be slightly easier.

We will assume that the reader is familiar with basic abstract algebra, including group theory and knowledge of rings, fields and factor groups, as well as elementary number theory.

The material in this chapter is mainly based on Bhattacharya, et al [bha94] and Stewart and Tall [ste02].

3.1 Basic abstract algebra

This section is mainly a refresher of definitions and results in basic algebra, and will include fields, field extensions, ideals and unique factorization domains.

3.1.1 Fields and field extensions

We recall the definition of subfields and field extensions:

Definition 3.1.1. *If F is a subfield of E, then E is called an* extension field *or an* extension *of F.*

If E is an extension of F, then E is a vector space over F. The dimension of the vector space of E over F can be written $[E : F]$; this dimension can be infinite.

Definition 3.1.2. *Let E be an extension of F. The dimension of the vector space of E over F is called the* degree *of E over F.*

Hence, the degree of E over F is $[E : F]$. If $[E : F]$ is finite, then E is a *finite* extension over F. Otherwise, E is an *infinite* extension over F. In the following discussion we will only look at finite extensions.

Next, we will define the notion of *algebraic* elements. In the following definitions, F is a field and E is a field extension of F.

Definition 3.1.3. *Let $\alpha \in E$. If there exists a non-zero polynomial $p(x) \in F[x]$ such that $p(\alpha) = 0$, then α is said to be* algebraic *over F. The root α of a polynomial $p(x) \in F[x]$ is also called an* algebraic element*.*

If no polynomial $p(x) \in F[x]$ exists such that $p(\alpha) = 0$, then α is *transcendental* over F. We will not consider transcendental numbers in this thesis.

We have the following results for finite extensions:

Theorem 3.1.1. *Let E be an extension field over F, and let $\alpha \in E$ be algebraic over F. Let $p(x) \in F[x]$ be a polynomial of the least possible degree such that $p(\alpha) = 0$. Then:*

a. $p(x)$ is irreducible over F.

b. If $g(x) \in F[x]$ is such that $g(\alpha) = 0$, then $p(x)|g(x)$.

c. There is exactly one monic polynomial $p(x) \in F[x]$ of least possible degree having $p(\alpha) = 0$.

The polynomial mentioned in point c in Theorem 3.1.1 is of particular importance.

Definition 3.1.4. *Let E be an extension field over F, let $p(x) \in F[x]$ be a non-zero, irreducible polynomial and let $\alpha \in E$ be algebraic over F. If $p(x)$ is monic with $p(\alpha) = 0$ and having the least possible degree, then it is called the* minimal polynomial *of α over F.*

Example 3.1.1. Let $F = \mathbb{Q}$ and let E be the smallest extention field of $\mathbb{Q}$ containing $\sqrt{2}$. Let $p(x) = x^2 - 2$. Then $p(\alpha) = 0$ and therefore $\sqrt{2}$ is algebraic over $\mathbb{Q}$. Also, $p(x)$ is the minimal polynomial of $\sqrt{2}$ over $\mathbb{Q}$.

We want a simple notation for extensions of a field, given an algebraic algebraic α. The following results are helpful:

Definition 3.1.5. *An extension field E of F is called* algebraic *if each element of E is algebraic over F.*

Theorem 3.1.2. *If E is a finite extension of F, then E is an* algebraic *extension of F.*

Let $F(\alpha)$ denote the smallest field containing all elements of F and the element α which is algebraic over F.

Theorem 3.1.3. *If E is an extension of F and $\alpha \in E$ is algebraic over F, then $F(\alpha)$ is an algebraic extension of F.*

If $E = F(\alpha)$ is a finite extension of F with degree $[E : F] = n$ for some algebraic element α, then a basis for the vector space of E over F is $\{1, \alpha, \alpha^2, \dots \alpha^{n-1}\}$. This basis will come in handy later when we look at ways to calculate the norm of elements in a number field.

We need some additional definitions that specify which fields we will be working with.

Definition 3.1.6. *The* characteristic *of a field F is the smallest positive integer $p \in F$ such that $px = 0$ for any $x \in F$. If no such p exists, the characteristic is 0.*

Example 3.1.2. The fields $\mathbb{Z}_p$ have characteristic p, and $\mathbb{Q}$ and finite extension E over $\mathbb{Q}$ have characteristic 0.

In the NFS we will be working with subfields of $\mathbb{C}$ which have characteristic 0. The following theorem is useful in our setting.

Theorem 3.1.4. *Let K be a field of characteristic 0. A non-zero polynomial f over K is divisible by the square of a polynomial of degree ≥ 0 if and only if f and f' have a common factor of degree ≥ 0.*

Some more concepts will be needed later, so let us define them as well.

Definition 3.1.7. *Let $f(x)$ be a polynomial over some field K of degree ≥ 1. An extension L of K is called a* splitting field *if $f(x)$ factors into linear factors in $L[x]$ and $L = K(\alpha_1, \alpha_2, \ldots, \alpha_n)$ where $\alpha_1, \ldots, \alpha_n$ are the roots of $f(x)$ in L.*

Definition 3.1.8. *An irreducible polynomial $f(x) \in K[x]$ is called a* separable polynomial *if all its roots have multiplicity 1.*

Definition 3.1.9. *Let L be an extension of a field K. An algebraic element $\alpha \in L$ is called* separable *over K if its minimal polynomial over K is separable.*

An algebraic field extension L over K is called a separable extension *if each element in L is separable over K.*

Definition 3.1.10. *A field K is* algebraically closed *if it has no proper algebraic extensions. That is, every algebraic extension of K coincide with K. If E is a subfield of K, then K is* algebraic *of E.*

Theorem 3.1.5. *Given a field K, the following are equivalent:*

i) K is algebraically closed.

ii) Every irreducible polynomial in $K[x]$ has degree 1.

iii) Every polynomial in $K[x]$ of positive degree factors completely into linear factors.

iv) Every polynomial in $K[x]$ of positive degree has at least one root in K.

Example 3.1.3. $\mathbb{C}$ is a field which is algebraically closed, so every polynomial in $\mathbb{C}[x]$ of degree ≥ 1 splits into linear factors.

The concept of embeddings is important in order to define the norm of an element in a number field, which we will get to in Section 3.2. But first we recall the following definition:

Definition 3.1.11. *Let f be a mapping from a ring R to a ring S such that*

a. $f(a+b) = f(a) + f(b), a, b \in R$

b. $f(ab) = f(a)f(b), a, b \in R.$

Then f is called a ring homomorphism *of R into S.*

Definition 3.1.12. *Let F be a field, K be a field extension of F, and let L be a field extension of K. Then a nonzero homomorphism $\sigma : K \mapsto L$ such that $\sigma(a) = a$ for all $a \in F$ is called an* embedding *of K in L over F.*

Example 3.1.4. Let $F = \mathbb{Q}$, $K = \mathbb{Q}(\sqrt{2})$ (where $\sqrt{2}$ is the root of some polynomial $p(x) \in \mathbb{Q}[x]$) and $L = \mathbb{C}$. These are two embeddings, $\sigma_1(a + b\sqrt{2}) = a + b\sqrt{2}$ and $\sigma_2(a + b\sqrt{2}) = a - b\sqrt{2}$. It is clear that $\sigma_1(a) = \sigma_2(a) = a$ for all $a \in \mathbb{Q}$. In other words, σ preserves all elements in $\mathbb{Q}$, but can send roots of $p(x)$ to different roots.

3.1.2 Prime and irreducible elements

In the NFS we will be working in subrings of fields in which we will perform factorization. In this section we will review some basic definitions, and our setting is commutative integral domains with unity. We recall that an integral domain is a commutative ring which has no zero divisors (that is, if $ab = c$ and $c \neq 0$, then $a \neq 0$ and $b \neq 0$).

Let R be an integral domain, and $a, b \in R$. An element a is a *divisor* of b if there exists a $c \in R$ such that $ac = b$. An element $u \in R$ is a *unit* if u is a divisor of 1. Two elements a, b are *associates* if there is a unit $u \in R$ such that $a = ub$. An element a is an *improper divisor* of b if a is a unit or if a and b are associates.

Definition 3.1.13. *A non-zero element a in R is called* irreducible *if it is not a unit and every divisor is improper. That is, $a = bc$ implies that either b or c is a unit.*

Definition 3.1.14. *A non-zero element p in R is called a* prime *if it is not a unit, and if $p|ab$, then $p|a$ or $p|b$.*

Theorem 3.1.6. *If $a \in R$ is prime, then a is also irreducible.*

Example 3.1.5. The converse of Theorem 3.1.6 is not true. The ring $\mathbb{Z}[\sqrt{-5}]$ is not a UFD, since for example $2 \cdot 3$ and $(1 + \sqrt{-5})(1 - \sqrt{-5})$ are two different factorizations of 6 into irreducible factors. It is clear that 2 divides 6, and it is possible to show that 2 does not divide either of $(1 + \sqrt{-5})$ and $(1 - \sqrt{-5})$. Hence, in $\mathbb{Z}[\sqrt{-5}]$, 2 is irreducible but not prime.

Definition 3.1.15. *An integral domain R is a* unique factorization domain *(or* UFD*) if the following conditions are satisfied:*

a. Every nonunit of R is a finite product of irreducible factors.

b. Every irreducible element is prime.

Theorem 3.1.7. *If R is a UFD, then the factorization of any element in R is a finite product of irreducible factors is unique up to order and unit factors.*

Example 3.1.6. $\mathbb{Z}$ and $F[x]$ over a field F are UFDs.

3.1.3 Ideals

We recall some basic definitions and results about ideals. For the following definitions and theorems, assume that R is a commutative ring.

Definition 3.1.16. *A subset $\mathfrak{a}$ of a ring R is called an* ideal *if $a, b \in \mathfrak{a}$ implies $a - b \in \mathfrak{a}$, and $a \in \mathfrak{a}, r \in R$ implies $ra \in \mathfrak{a}$.*

We write aR for the ring with elements $\{ab|b \in R\}$.

Example 3.1.7. Let $R = \mathbb{Z}$. Then $n\mathbb{Z} = \{na|a \in \mathbb{Z}\}$ is an ideal for every $n \in \mathbb{Z}$. In particular, $2\mathbb{Z}$ is the ideal of even numbers in $\mathbb{Z}$.

Let the smallest ideal $\mathfrak{a}$ containing the elements $a_1, a_2, \ldots, a_m$ be denoted $\mathfrak{a} = \langle a_1, a_2, \ldots, a_m\rangle$.

Definition 3.1.17. *An ideal $\mathfrak{a}$ of a ring R is called* finitely generated *if $\mathfrak{a} = \langle a_1, a_2, \ldots, a_m\rangle$ for some $a_i \in R$, $1 \leq i \leq m$.*

Definition 3.1.18. *An ideal $\mathfrak{a}$ of a ring R is called* principal *if $\mathfrak{a} = \langle a\rangle$ for some $a \in R$.*

Definition 3.1.19. *A commutative integral domain with 1 in which every ideal is principal is called a* principal ideal domain *or* PID*.*

Example 3.1.8. $2\mathbb{Z} = \langle 2\rangle$ is a principal ideal, and $\mathbb{Z}$ is a principal ideal domain.

Definition 3.1.20. *Let $\mathfrak{a}, \mathfrak{b}$ be ideals in R. Then the set*

$$\{a + b|a \in \mathfrak{a}, b \in \mathfrak{b}\}$$

(which is an ideal in R) is called the sum *of $\mathfrak{a}$ and $\mathfrak{b}$ and is written $\mathfrak{a} + \mathfrak{b}$.*

Definition 3.1.21. *Let $\mathfrak{a}, \mathfrak{b}$ be ideals in R. Then the set*

$$\{a_1b_1 + a_2b_2 + \cdots + a_nb_n|a_i \in \mathfrak{a}, b_i \in \mathfrak{b}, n \geq 1 \in \mathbb{Z}\}$$

(which is an ideal in R) is called the product *of $\mathfrak{a}$ and $\mathfrak{b}$ and is written $\mathfrak{a}\mathfrak{b}$.*

Definition 3.1.22. *An ideal $\mathfrak{a}$ in R is called* maximal *if $\mathfrak{a} \neq R$ and $\mathfrak{b} \supset \mathfrak{a}$ for an ideal $\mathfrak{b} \subseteq R$ implies $\mathfrak{b} = R$.*

Definition 3.1.23. *An ideal $\mathfrak{p}$ in a ring R is called a* prime ideal *if the following holds: If $\mathfrak{a}$ and $\mathfrak{b}$ are ideals in R such that $\mathfrak{a}\mathfrak{b} \subseteq \mathfrak{p}$, then $\mathfrak{a} \subseteq \mathfrak{p}$ or $\mathfrak{b} \subseteq \mathfrak{p}$.*

Theorem 3.1.8. *If R is a ring with unity, then each maximal ideal is prime.*

Theorem 3.1.9. *If R is a ring, then an ideal $\mathfrak{p}$ in R is prime if and only if $ab \in \mathfrak{p}, a \in R, b \in R$ implies $a \in \mathfrak{p}$ or $b \in \mathfrak{p}$.*

Definition 3.1.24. *Let $\mathfrak{a}$ and $\mathfrak{b}$ be ideals in R. $\mathfrak{a}|\mathfrak{b}$ ($\mathfrak{a}$ divides $\mathfrak{b}$) if and only if $\mathfrak{a} \supseteq \mathfrak{b}$.*

3.2 Algebraic number theory

Algebraic number theory is the study of algebraic structures that arise from finite field extensions of $\mathbb{Q}$. An important structure is the *number field*:

Definition 3.2.1. *Let $K = \mathbb{Q}(\alpha)$ be an algebraic extension of $\mathbb{Q}$. Then K is called an* algebraic number field *or simply a* number field.

An element in a number field is called an *algebraic number*.

Definition 3.2.2. *An algebraic number α is an* algebraic integer *if there is a monic polynomial $p(x)$ with integer coefficients such that $p(\alpha) = 0$.*

Example 3.2.1. $\alpha = \sqrt{2}$ is an algebraic integer, since it satisfies $\alpha^2 - 2 = 0$.

Example 3.2.2. $\alpha = \sqrt{-2}$ is an algebraic integer, since it satisfies $\alpha^2 + 2 = 0$.

Example 3.2.3. $\alpha = \frac{1+\sqrt{5}}{2}$ is an algebraic integer, since it satisfies $\alpha^2 - \alpha - 1 = 0$.

Example 3.2.4. It can be shown that $\alpha = \frac{1}{2}$ is not an algebraic integer. Some polynomial equations having α as a root include $\alpha - \frac{1}{2} = 0$ (coefficients not in $\mathbb{Z}$) and $2\alpha - 1 = 0$ (not a monic polynomial).

We now want to define a concept that will be important for the sieve stage of the NFS. The norm often allows us to transform a problem from the domain of algebraic integers to rational integers.

We need a result about embeddings in order to define the norm.

Lemma 3.2.1. *Let K be a subfield of $\mathbb{C}$ and $f(x) \in K[x]$ be an irreducible polymomial. Then $f(x)$ has no roots of multiplicity 2 or higher. That is, $f(x)$ is a separable polynomial.*

Proof. Since $f(x)$ is irreducible over K, then $f(x)$ and $f'(x)$ are relatively prime by Theorem 3.1.4. Hence, there exist polynomials a, b over K such that $af(x) + bf'(x) = 1$ and this equation interpreted over $\mathbb{C}$ shows that $f(x)$ and $f'(x)$ are relatively prime over $\mathbb{C}$. By applying Theorem 3.1.4 again, $f(x)$ cannot have repeated zeros. □

Theorem 3.2.1. *Let $K = \mathbb{Q}(\alpha)$ be a number field of degree n and a field extension of $\mathbb{Q}$. Then there are exactly n distinct embeddings $\sigma_i : K \mapsto \mathbb{C}$. The elements $\sigma_i(\alpha) = \alpha_i$ are the distinct roots in $\mathbb{C}$ of the minimal polynomial of α over $\mathbb{Q}$.*

Proof. This proof follows Stewart and Tall [ste02, page 38-39]. By Lemma 3.2.1, the minimal polynomial $p(x)$ of K over $\mathbb{Q}$ has no roots of multiplicity ≥ 2, so its n unique roots are $\alpha_1, \alpha_2, \ldots, \alpha_n$. Each root α_i also has a minimal polynomial, and by Theorem 3.1.1, each of them must divide the irreducible $p(x)$. Hence there is a unique field isomorphism $\sigma_i : \mathbb{Q}(\alpha) \mapsto \mathbb{Q}(\alpha_i)$ such that $\sigma_i(\alpha) = \alpha_i$.

If $\beta \in \mathbb{Q}(\alpha)$, then $\beta = r(\alpha)$ for a unique $r \in \mathbb{Q}[x]$ with $\deg(r) < n$ and we must have that $\sigma_i(\beta) = r(\alpha_i)$. (For references to the proof of this claim, see Stewart and Tall [ste02] page 39, proof of Theorem 2.4.)

Conversely, if $\sigma : K \mapsto \mathbb{C}$ is a monomorphism (an injective homomorphism) then σ is the identity on $\mathbb{Q}$. Then,

$$\sigma(p(\alpha)) = p(\sigma(\alpha)) = 0.$$

Then $\sigma(\alpha)$ is one of the α_i, hence σ is one of the σ_i. □

Now we are ready to define the norm.

Definition 3.2.3. *Let $K = \mathbb{Q}(\alpha)$ be a number field of degree n, and let $\sigma_1, \sigma_2, \cdots, \sigma_n$ be the n embeddings $K \mapsto \mathbb{C}$. We define the* norm *of an algebraic integer a as*

$$N(a) = \prod_{i=1}^{n} \sigma_i(a).$$

Since the σ_i are ring homomorphisms we have $N(ab) = N(a)N(b)$ and $N(a) \neq 0$ if and only if $a \neq 0$.

The norm is a concept of great importance for the NFS, and it shows up in several of the stages of the algorithm. Therefore we will include multiple examples.

Example 3.2.5. Let $K = \mathbb{Q}(\sqrt{2})$. K is a number field of degree 2, with the following embeddings of K into $\mathbb{C}$ over $\mathbb{Q}$:

$$\begin{aligned}\sigma_1(a + b\sqrt{2}) &= a + b\sqrt{2},\\ \sigma_2(a + b\sqrt{2}) &= a - b\sqrt{2}.\end{aligned}$$

The norm is $N(a + b\sqrt{2}) = (a + b\sqrt{2})(a - b\sqrt{2}) = a^2 - 2b^2$.

Example 3.2.6. Let $\mathbb{Q}(\alpha)$ be an extension of $\mathbb{Q}$ such that the minimal polynomial over $\mathbb{Q}$ is $f(x) = x^2 + 2x + 2$ having α as a root. We take for granted that the roots of the quadratic equation $x^2 + ax + b$ are given by $x = -\frac{a}{2} \pm \frac{\sqrt{a^2-4b}}{2}$. Then the roots are $\alpha_1 = -1 + \sqrt{-1}$ and $\alpha_2 = -1 - \sqrt{-1}$. The embeddings are:

$$\begin{aligned}\sigma_1(a + b\alpha_1) &= a + b(-1 + \sqrt{-1}),\\ \sigma_2(a + b\alpha_2) &= a + b(-1 - \sqrt{-1}).\end{aligned}$$

The norm is

$$\begin{aligned}N(a + b\alpha) &= (a + b[-1 + \sqrt{-1}])(a + b[-1 - \sqrt{-1}])\\ &= a^2 + ab(-1 + \sqrt{-1} - 1 - \sqrt{-1}) + b^2(-1 + \sqrt{-1})(-1 - \sqrt{-1})\\ &= a^2 - 2ab + 2b^2.\end{aligned}$$

Example 3.2.7. In this example we will develop a general expression for the norm of an element where the extension has degree 2. Let $\mathbb{Q}(\alpha)$ be an extension of $\mathbb{Q}$ such that the minimal polynomial over $\mathbb{Q}$ is $f(x) = x^2 + cx + d$ having α as a root. Let c, d be arbitrary integers in $\mathbb{Z}$ such that $f(x)$ is irreducible. The roots are

$$\begin{aligned}\alpha_1 &= -\frac{c}{2} + \frac{\sqrt{c^2 - 4d}}{2},\\ \alpha_2 &= -\frac{c}{2} - \frac{\sqrt{c^2 - 4d}}{2}.\end{aligned}$$

The embeddings are:

$$\begin{aligned}\sigma_1(a + b\alpha) &= a + b\left(-\frac{c}{2} + \frac{\sqrt{c^2 - 4d}}{2}\right),\\ \sigma_2(a + b\alpha) &= a + b\left(-\frac{c}{2} - \frac{\sqrt{c^2 - 4d}}{2}\right).\end{aligned}$$

The norm is

$$\begin{aligned}
N(a+b\alpha) &= \left(a+b\left[-\frac{c}{2}+\frac{\sqrt{c^2-4d}}{2}\right]\right)\left(a+b\left[-\frac{c}{2}-\frac{\sqrt{c^2-4d}}{2}\right]\right)\\
&= a^2+ab\left(-\frac{c}{2}+\frac{\sqrt{c^2-4b}}{2}-\frac{c}{2}-\frac{\sqrt{c^2-4b}}{2}\right)\\
&\quad + b^2\left(-\frac{c}{2}+\frac{\sqrt{c^2-4d}}{2}\right)\left(-\frac{c}{2}-\frac{\sqrt{c^2-4d}}{2}\right)\\
&= a^2+ab\left(-\frac{c}{2}-\frac{c}{2}\right)+b^2\left(\frac{c^2}{4}-\frac{c^2-4d}{4}\right)\\
&= a^2-cab+db^2.
\end{aligned}$$

Example 3.2.8. Lastly, we include an example where the norm of a degree 3 extension is determined. Let $f(x)=x^3+2$ with root α, and let $\mathbb{Q}(\alpha)$ be a finite extension. The roots of $f(x)$ are:

$$\begin{aligned}
\alpha_1 &= \sqrt[3]{-2},\\
\alpha_2 &= \omega\sqrt[3]{-2},\\
\alpha_3 &= \omega^2\sqrt[3]{-2},
\end{aligned}$$

where $\omega=e^{2\pi i/3}$, the cube root of unity. Let $\alpha=\sqrt[3]{-2}$. The embeddings are:

$$\begin{aligned}
\sigma_1(a+b\alpha+c\alpha^2) &= a+b\alpha+c\alpha^2,\\
\sigma_2(a+b\alpha+c\alpha^2) &= a+b\omega\alpha+c\omega^2\alpha^2,\\
\sigma_3(a+b\alpha+c\alpha^2) &= a+b\omega^2\alpha+c\omega(=\omega^4)\alpha^2.
\end{aligned}$$

By the definition of the norm:

$$\begin{aligned}
N(a+b\alpha+c\alpha^2) &= \sigma_1(a+b\alpha+c\alpha^2)\sigma_2(a+b\alpha+c\alpha^2)\sigma_3(a+b\alpha+c\alpha^2)\\
&= (a+b\alpha+c\alpha^2)(a+b\omega\alpha+c\omega^2\alpha^2)(a+b\omega^2\alpha+c\omega^4\alpha^2)
\end{aligned}$$

Let's expand and group by coefficients in a,b,c:

$$\begin{aligned}
N(a+b\alpha+c\alpha^2) = \quad & a^3\\
&+b^3\omega^3\alpha^3\\
&+c^3\omega^3\alpha^6\\
&+a^2b(\alpha+\omega\alpha+\omega^2\alpha)\\
&+a^2c(\alpha^2+\omega^2\alpha^2+\omega^4\alpha^2)\\
&+b^2a(\omega\alpha^2+\omega^2\alpha^2+\omega^3\alpha^2)\\
&+b^2c(\omega^2\alpha^4+\omega^3\alpha^4+\omega^4\alpha^4)\\
&+c^2a(\omega^2\alpha^4+\omega^4\alpha^4+\omega^6\alpha^4)\\
&+c^2b(\omega^3\alpha^5+\omega^2\alpha^5+\omega^4\alpha^5)\\
&+abc(\omega^2\alpha^3+\omega^4\alpha^3+\omega\alpha^3+\omega^2\alpha^3+\omega^2\alpha^3+\omega\alpha^3)
\end{aligned}$$

Use that $\omega^3 = 1$, $\alpha^3 = -2$, $\omega + \omega^2 = -1$, $1 + \omega + \omega^2 = 0$ and $\omega^{k+3} = \omega^k$. Most of the terms above vanish because they are multiples of $1 + \omega + \omega^2$. The last term is shortened as follows:

$$\begin{aligned}
abc(\omega^2\alpha^3 + \omega^4\alpha^3 + \omega\alpha^3 + \omega^2\alpha^3 + \omega^2\alpha^3 + \omega\alpha^3) &= 3abc\alpha^3(\omega + \omega^2) \\
&= 3abc(-2)(-1) \\
&= 6abc
\end{aligned}$$

Finally we arrive at the expression

$$N(a + b\alpha + c\alpha^2) = a^3 - 2b^3 + 4c^3 + 6abc.$$

We see that even for "innocent-looking" polynomials like $x^3 + 2$ a fair amount of work is needed in order to determine the norm of the extension. We cannot hope to use this method for arbitrary extensions, so another method is desired. We will look at another method, but first we will see how we can calculate the norm if the algebraic numbers are of a simpler form.

In the NFS we will mainly deal with the norm of elements of the form $a - b\alpha$ that are members of a number field of degree n. Therefore we seek an expression for this that is easy to implement. The following theorem is very useful.

Theorem 3.2.2. *Let $\mathbb{Q}(\alpha)$ be a number field of degree n. The norm of an element $a - b\alpha \in \mathbb{Q}(\alpha)$ is*

$$N(a - b\alpha) = b^n f(a/b).$$

Proof. By Theorem 3.2.1, there are n embeddings σ_i, and the elements $\sigma_1(\alpha), \sigma_2(\alpha), \dots, \sigma_n(\alpha)$ are identical, for some ordering, to the roots $\alpha_1, \alpha_2, \dots, \alpha_n$ of the minimal polynomial of α over $\mathbb{Q}$. Starting with Definition 3.2.3, we get

$$\begin{aligned}
N(a - b\alpha) &= \prod_{i=1}^{n} \sigma_i(a - b\alpha) \\
&= (a - b\alpha_1)(a - b\alpha_2) \cdots (a - b\alpha_n) \\
&= b^n(a/b - \alpha_1)(a/b - \alpha_2) \cdots (a/b - \alpha_n) \\
&= b^n f(a/b)
\end{aligned}$$

□

The norm of $a - b\alpha$ can also be written as $N(a - b\alpha) = F(a, b)$ where

$$F(x, y) = x^n + a_{d-1}x^{d-1}y + \cdots + a_0 y^n = y^d f(x/y).$$

From this form it is immediately clear that the norm is an integer whenever a, b are integers.

In Section 5.2 we will give an algorithm for calculating the norm in an arbitrary finite extension that avoids determining the expression for the norm.

3.2.1 Factorization of algebraic integers and ideals

In the NFS algorithm we need to factorize numbers on the algebraic side into prime factors so that they can be represented with an exponent vector (the exponent vector was defined in Section 2.2). If factorization was guaranteed to be unique in $\mathbb{Z}[\alpha]$ all would be good. Unfortunately, this is not generally the case.

While some number rings like $\mathbb{Z}[i]$ are unique factorization domains, $\mathbb{Z}[\sqrt{-5}]$ is a number ring which is not a UFD. We have that $6 = 2 \cdot 3 = (1+\sqrt{-5})(1-\sqrt{-5})$. It can be shown that all of the factors $2, 3, (1+\sqrt{-5})$ and $(1-\sqrt{-5})$ are irreducible in both $\mathbb{Z}[\sqrt{-5}]$ and $\mathcal{O}_{\mathbb{Q}(\sqrt{-5})}$ (defined below), and none are associates of each other since the only units in both rings are 1 and -1.

Our quest for the remainder of this section is to find a setting where we have unique factorization. We will begin by exploring a new kind of ring.

Definition 3.2.4. *Let α be the root of an irreducible polynomial $f(x) \in \mathbb{Z}[x]$. The ring $\mathbb{Z}[\alpha]$ is called a* number ring. *Let $K = \mathbb{Q}(\alpha)$ be a number field. Let the ring of algebraic integers in K be denoted as $\mathcal{O}_K$ (or $\mathcal{O}_{\mathbb{Q}(\alpha)}$).*

The rings $\mathcal{O}_{\mathbb{Q}(\alpha)}$ and $\mathbb{Z}[\alpha]$ are not necessarily equal. For example, consider the rings $\mathcal{O}_{\mathbb{Q}(\sqrt{5})}$ and $\mathbb{Z}[\sqrt{5}]$. The element $\frac{1+\sqrt{5}}{2}$ is a root of the polynomial $x^2 - x - 1$. Since the polynomial is monic and has integer coefficients, $\frac{1+\sqrt{5}}{2}$ is an algebraic integer by definition and a member of $\mathcal{O}_{\mathbb{Q}(\sqrt{5})}$. However, it is not a member of $\mathbb{Z}[\sqrt{5}]$, as $\frac{1+\sqrt{5}}{2} = \frac{1}{2} + \frac{1}{2}\sqrt{5}$ and $\frac{1}{2} \notin \mathbb{Z}$.

Even though none of $\mathbb{Z}[\alpha]$ and $\mathcal{O}_{\mathbb{Q}(\alpha)}$ are not guaranteed to be UFDs, all hope is not lost. Instead of factoring elements of the form $a - b\alpha$ we could try to factor the ideal $\langle a - b\alpha \rangle$ of $\mathcal{O}_{\mathbb{Q}(\alpha)}$ into prime ideals instead. We want to factor ideals in in $\mathcal{O}_{\mathbb{Q}(\alpha)}$ rather than $\mathbb{Z}[\alpha]$ because of the following important result:

Theorem 3.2.3. *In the ring of integers $\mathcal{O}_{\mathbb{Q}(\alpha)}$, every proper non-zero ideal can be written uniquely as the product of prime ideals.*

The proof is omitted here, see Stewart and Tall [ste02] pages 107-110 for the full proof.

In addition, factoring ideals instead of elements has another nice property. We don't have to care about units. If u is a unit, then $\langle a \rangle = \langle au \rangle$.

Here follow some examples of factorizations of ideals into prime ideals, presented without proof.

Example 3.2.9. $\langle 10 \rangle = \langle 2 \rangle \langle 5 \rangle$ in $\mathbb{Z} = \mathcal{O}_{\mathbb{Z}}$.

Example 3.2.10. $\langle 100 \rangle = \langle 2 \rangle \langle 2 \rangle \langle 5 \rangle \langle 5 \rangle$ in $\mathbb{Z}$.

Example 3.2.11. $\langle 16 \rangle = \langle 2 \rangle \langle 2 \rangle \langle 2 \rangle \langle 2 \rangle$ in $\mathbb{Z}$.

For the three previous examples, the factorization of elements in $\mathbb{Z}$ can be said to be identical to the factorization of ideals in $\mathbb{Z}$, since the prime ideals and prime numbers correspond. The next example is more interesting, and shows our problematic factorization mentioned in the beginning of this subsection, and the relation between the two factorizations:

Example 3.2.12. $\langle 6\rangle = \langle 2, 1+\sqrt{-5}\rangle\langle 2, 1+\sqrt{-5}\rangle\langle 3, 1+\sqrt{-5}\rangle\langle 3, 1-\sqrt{-5}\rangle$ in $\mathcal{O}_{\mathbb{Q}(\sqrt{-5})}$. The ideals generated by each of the irreducible factors of $6 \in \mathcal{O}_{\mathbb{Q}(\sqrt{5})}$ are products of different combinations of the prime ideals that are factors of $\langle 6\rangle$:

$$\begin{aligned}
\langle 2\rangle &= \langle 2, 1+\sqrt{-5}\rangle^2 \\
\langle 3\rangle &= \langle 3, 1+\sqrt{-5}\rangle\langle 3, 1-\sqrt{-5}\rangle \\
\langle 1+\sqrt{-5}\rangle &= \langle 2, 1+\sqrt{-5}\rangle\langle 3, 1+\sqrt{-5}\rangle \\
\langle 1-\sqrt{-5}\rangle &= \langle 2, 1+\sqrt{-5}\rangle\langle 3, 1-\sqrt{-5}\rangle
\end{aligned}$$

In particular, we notice that none of the prime ideals are principal, and that none of the ideals generated by the irreducible elements (factors of $6 \in \mathcal{O}_{\mathbb{Q}(\sqrt{5})}$) are prime.

As with algebraic integers, we can define the *norm* of an ideal. The norm will be essential for helping us find the the prime factorization of an ideal. Before we can define the norm, we need one more result.

Theorem 3.2.4. *If $\mathfrak{a}$ is a non-zero ideal in $\mathcal{O}_K$, then the quotient ring $\mathcal{O}_K/\mathfrak{a}$ is finite.*

Proof. Let $\mathfrak{a}$ be a non-zero ideal in $\mathcal{O}_K$, and let $\theta \in \mathfrak{a}$ be different from zero. Let

$$N = N(\theta) = \theta_1\theta_2\cdots\theta_n \in \mathfrak{a},$$

where the θ_i are the conjugates of θ (including θ itself). Then we have $\langle N\rangle \subseteq \mathfrak{a}$ and hence $\mathcal{O}_K/\mathfrak{a}$ is a quotient ring that is a subring of $\mathcal{O}_K/\langle N\rangle$. $\mathcal{O}_K/\langle N\rangle$ is a finitely generated abelian group (when viewed as a group) where each element is of finite order, and is therefore finite. Hence $\mathcal{O}_K/\mathfrak{a} \subseteq \mathcal{O}_K/\langle N\rangle$ is finite. $\square$

Definition 3.2.5. *The* norm *of a non-zero ideal $\mathfrak{a}$ of $\mathcal{O}_{\mathbb{Q}(\alpha)}$ is the size of the quotient ring $\mathcal{O}_{\mathbb{Q}(\alpha)}/\mathfrak{a}$. That is, $\mathfrak{N}(\mathfrak{a}) = |\mathcal{O}_{\mathbb{Q}(\alpha)}/\mathfrak{a}|$. In addition, $\mathfrak{N}(\langle 0\rangle) = 0$.*

It follows form the definition that the norm of a non-zero ideal is a positive integer. To distinguish the norms from each other, we will use $\mathfrak{N}$ for the norm of ideals and N for norm of algebraic integers.

Now follow a series of theorems that will help us calculate the norm of a given ideal. First of all, there is a connection between the norm of an algebraic integer and the norm of an ideal generated by the same algebraic integer. Some of the proofs are omitted, as they depend on topics not covered in this thesis, such as discriminants, free abelian groups and fractional ideals.

Theorem 3.2.5. *Let $\alpha \in \mathcal{O}_K$. Then $\mathfrak{N}(\langle\alpha\rangle) = |N(\alpha)|$.*

See Stewart and Tall [ste02] page 116 (proof of Corollary 5.10) for the full proof.

Theorem 3.2.6. *If $\mathfrak{a}$ and $\mathfrak{b}$ are non-zero ideals of $\mathcal{O}_K$, then $\mathfrak{N}(\mathfrak{a}\mathfrak{b}) = \mathfrak{N}(\mathfrak{a})\mathfrak{N}(\mathfrak{b})$.*

See Stewart and Tall [ste02], pages 116-118 (proof of Theorem 5.12) for the full proof.

We remind ourselves that our goal is to factor the ideal $\langle a - b\alpha\rangle$ into prime ideals. These prime factors have special properties; we will prove later that they are of a certain *degree*.

Definition 3.2.6. *Let $\mathfrak{p}$ be a prime ideal in $\mathcal{O}_{\mathbb{Q}(\alpha)}$. The* degree *of $\mathfrak{p}$ is the integer k such that $\mathfrak{N}(p) = |\mathbb{Z}[\alpha]/\mathfrak{p}|^k$ is satisfied. In particular, $\mathfrak{p}$ is a* first degree *prime ideal if $\mathfrak{N}(p) = |\mathbb{Z}[\alpha]/\mathfrak{p}|$.*

The following results give us more information about the form of the prime factors of the ideals $\langle a - b\alpha \rangle$ in $\mathcal{O}_{\mathbb{Q}(\alpha)}$.

Theorem 3.2.7. *Every ideal in the ring of integers $\mathcal{O}_{\mathbb{Q}(\alpha)}$ has at most two generators. That is, every ideal is of the form $\langle \beta \rangle$ or $\langle \beta, \gamma \rangle$ for $\beta, \gamma \in \mathcal{O}_{\mathbb{Q}(\alpha)}$.*

See Stewart and Tall [ste02] page 81 (proof of Theorem 4.7) and page 121 (proof of Theorem 5.20) for the full proof.

The following four theorems are not proven here, look in Lenstra, et at [len91, pages 58-59] for the proofs.

Theorem 3.2.8. *The prime ideals in $\mathcal{O}_{\mathbb{Q}(\alpha)}$ that divide $\langle a - b\alpha \rangle$ are of the form $\langle p, \alpha - r \rangle$, where p is a prime in $\mathbb{Z}$ and r is an integer satisfying $f(r) \equiv 0$ (mod p).*

Theorem 3.2.9. *Let $a - b\alpha \in \mathbb{Z}[\alpha]$ be an algebraic integer. Then the prime factorization of the ideal $\langle a - b\alpha \rangle$ in $\mathcal{O}_{\mathbb{Q}(\alpha)}$ is*

$$\langle a - b\alpha \rangle = \prod \langle p_i, \alpha - r_i \rangle^{e_i}.$$

That is, each prime ideal is a first degree ideal and is generated by two elements.

Theorem 3.2.10. *A prime ideal $\langle p, \alpha - r \rangle$ divides $\langle a - b\alpha \rangle$ if and only if $a - br \equiv 0$ (mod p).*

Theorem 3.2.11. *Let $\mathfrak{p} = \langle p, \alpha - r \rangle$ be a first degree prime ideal. Then $\mathfrak{N}(\mathfrak{p}) = p$.*

We now have what we need in order to factor the ideal $\langle a - b\alpha \rangle$ in $\mathcal{O}_{\mathbb{Q}(\alpha)}$. There are still some obstructions that need to be overcome when going from prime ideals in $\mathcal{O}_{\mathbb{Q}(\alpha)}$ to elements in $\mathbb{Z}[\alpha]$. We will present a solution to this missing step in the chapter about the NFS.

Chapter 4

Number field sieve

Like most modern factoring methods, the goal of the number field sieve (NFS) is to find two integers u, v such that $u^2 \equiv v^2 \pmod{n}$ and $u \not\equiv v \pmod{n}$. Then there is a chance that $\gcd(n, u-v)$ and $\gcd(n, u+v)$ are nontrivial factors of the integer n we wish to factor. n should be an odd composite number which is not a power (n should not be of the form a^k for integers a and $k \geq 2$).

In order to achieve a faster runtime than QS, the NFS uses a *number ring* instead of searching for u, v in $\mathbb{Z}_n$ only. Given an irreducible polynomial $f(x)$ with coefficients in $\mathbb{Z}_n$ with a root $\alpha \in \mathbb{C}$ and an integer m such that $f(m) \equiv 0 \pmod{n}$, we attempt to find a square in the number ring $\mathbb{Z}[\alpha]$. In addition, we use a ring homomorphism

$$\sigma : \mathbb{Z}[\alpha] \mapsto \mathbb{Z}_n \tag{4.1}$$

induced by $\sigma(\alpha) = m$ to take us back into $\mathbb{Z}_n$ again. This is accomplished by finding many integer pairs (a, b) such that $a - bm$ is smooth with regard to a rational factor base (that is, prime numbers in $\mathbb{Z}$), and $a - b\alpha$ is smooth with regard to an algebraic factor base. Then, we try to find a subset $\mathcal{S}$ of these pairs so that

$$u^2 = \prod_{(a,b)\in\mathcal{S}} (a - bm) \text{ is a square in } \mathbb{Z}_n$$

and

$$\gamma^2 = \prod_{(a,b)\in\mathcal{S}} (a - b\alpha) \text{ is a square in } \mathbb{Z}[\alpha].$$

Then, we take the square root on both sides, apply the homomorphism $\sigma(\gamma) = v$ and evaluate $\gcd(n, u - v)$, hopefully finding a nontrivial factor of n.

The NFS algorithm consists of the following phases:

- **Polynomial selection**: Determine the polynomial $f(x)$ and an integer m such that $f(m) \equiv 0 \pmod{n}$. Let $\alpha \in \mathbb{C}$ be a root of $f(x)$. $\mathbb{Z}[\alpha]$ is the number ring we are working in.

- **Sieving**: Find many pairs (a, b) such that $a - bm$ and $a - b\alpha$ are both smooth with regard to their respective factor bases.

- **Linear algebra**: Combine the pairs from the sieving phase, and solve a system of linear equations in order to find a subset of these pairs such that their products are rational and algebraic squares.

- **Square root and gcd**: Take the square root of the products from the last phase, and take the gcd of n and the difference of the square roots.

These phases will be described in more detail below. We will focus on the basic variant of the algorithm here.

4.1 Polynomial selection

The following is a simple way that works well. First, pick the degree d of the polynomial. Setting $d = \lfloor (3 \ln n / \ln \ln n)^{1/3} \rfloor$ is asymptotically optimal [cra05, page 287]. $d = 5$ or $d = 6$ works fine for integers with between 100 and 250 digits. However, an odd degree allows an easier algorithm for the square root phase, which we will describe in Section 4.5.2. Then, let $m = \lfloor n^{1/d} \rfloor$ and let the coefficients of $f(x)$ be the base-m expansion of n. That is,

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$$

such that $f(m) = n$. Because of the choice of m, we always have $a_d = 1$ and hence $f(x)$ is monic.

If $f(x)$ is irreducible, we obtain a number ring $\mathbb{Z}[x]/\langle f(x) \rangle$. An element in this ring can be written

$$a_0 + a_1 \alpha + a_2 \alpha^2 + \cdots a_{d-1} \alpha^{d-1}.$$

An element can be considered a vector with d coordinates a_i in a d-dimensional vector space with basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{d-1}\}$. Addition in this ring is simply vector addition. Multiplication of two elements can be viewed as multiplication of polynomials in α, which are then reduced to a new polynomial of degree at most $d-1$ using the identity $f(\alpha) = 0$. See Section 5.1 for a more detailed description.

While the base-m algorithm described above is asymptotically optimal, it is possible to do better in practice.

The quality of the polynomial is determined by its *yield*, which is the number of smooth values it produces for a given smoothness bound and sieving range.

According to Murphy [mur99], there are two main factors that influence the yield, *size* and *root* properties. The size properties has to do with the magnitude of the coefficients of f, while the root properties has to do with the number of roots of f modulo p^k for small primes p and $k \geq 1$.

Small [sma03] lists the following polynomial properties as desirable.

- The polynomial has small coefficients.
- The polynomial has many real roots. A randomly selected polynomial would most likely have very few real roots.
- The polynomial has many roots modulo lots of small prime numbers.
- The Galois groups of the polynomial is small.

It can be beneficial to spend some computer time trying some polynomials and do some experimental sieving and pick the one with the highest yield.

The construction of $f(x)$ actually provides an opportunity for an early exit of the algorithm with a nontrivial factor of n. If we can write $f(x)$ as $g(x)h(x)$ where none of the factors are units, then $n = g(m)h(m)$ is a factorization of n. Another early exit opportunity which doesn't require polynomial factorization can be achieved by checking if $\gcd(f'(m), n)$ is a non-trivial factor. This method only works if $f(x)$ has a square factor. Factoring $f(x)$ is sufficient and will also cover all cases that will be detected by the gcd method.

Polynomial factorization is non-trivial to implement, and a deterministic polynomial-time algorithm (in the degree of the polynomial and the logarithm of its coefficients) is described in Lenstra, et al [len82]. An algorithm which is faster in practice and easier to implement, but with no rigorous polynomial-time guarantee is given by Knuth [knu98, Section 4.6.2]. A simple algorithm which only works for $f(x)$ of degree 3 or lower is given in Section 5.4. For n up to 60 digits, $f(x)$ of degree 3 is a good choice.

4.2 The sieve

In the sieve phase we are only concerned with rational numbers of the form $a - bm$ and algebraic numbers of the form $a - b\alpha$ (to be more specific, the ideals generated by $a - b\alpha$).

We are interested in finding pairs a, b such that $a - bm$ is B_1-smooth in $\mathbb{Z}$ and the ideal $\langle a - b\alpha \rangle$ is B_2-smooth in $\mathcal{O}_{\mathbb{Q}(\alpha)}$. Here B_1 is the upper bound for rational primes (the rational factor base) and B_2 is the upper bound of the norm of prime ideals (the algebraic factor base). This will be achieved by searching through all $|a| \leq M$ and $0 < b < N$ with $\gcd(a, b) = 1$ using a sieving process similar to the one used in the Quadratic Sieve. From Section 3.2.1 we have that the factorization of $\langle a - b\alpha \rangle$ into prime ideals can be deduced from $N(a - b\alpha)$.

Here we will describe a method known as the *line sieve*. The following description is from a more mathematical perspective; a description with implementation details is described in Chapter 6.

Let b be fixed. Create an array with one entry for each possible value of a such that $|a| \leq M$ for some bound M. For each a, initialize the corresponding array element with $(a - bm) \cdot N(a - b\alpha)$ (the product of the rational number and the algebraic norm).

First, for each a, check if it is negative. If it is, we note us that a was negative, and the we set a to $-a$ to make it positive.

For each rational prime p in the factor base and for each a within our interval such that p divides $a - bm$, divide the corresponding array element by p (possibly more than once if a power of p is a divisor).

For each algebraic prime in the factor base represented by the pair p, r and for each a such that the ideal $\langle p - r\alpha \rangle$ divides the ideal $\langle a - b\alpha \rangle$ (or equivalently, $a - br \equiv 0 \pmod{p}$), divide the corresponding array element by p (again, powers can occur).

After this procedure, all array elements equal to 1 correspond to pairs a, b such that both $a - bm$ and $\langle a - b\alpha \rangle$ are smooth. We only care about a, b such that $\gcd(a, b) = 1$ to avoid redundant pairs.

For each smooth pair a, b we associate an *exponent vector* (see Section 2.2). The exponent vector has the following elements:

- One entry for the sign of $a - bm$ (0 if positive, 1 if negative).
- One entry for each prime p in the rational factor base.

- One entry for each prime in the algebraic factor base, represented by the pair p, r.

For each rational prime p, the corresponding entry in the exponent vector is the highest power of p that divides $a - bm$ reduced modulo 2. For each algebraic prime represented by p, r, the corresponding entry in the exponent vector is the highest power of $\langle p - r\alpha \rangle$ that divides $\langle a - b\alpha \rangle$ reduced modulo 2. It follows that each element in the exponent vector is either 0 or 1.

The exponent vector for each smooth pair a, b is stored for later use. Each row of the matrix in the linear algebra step (section 4.3) consists of the exponent vector for one pair a, b.

The ultimate goal of the sieve phase is to gather enough a, b pairs to be able to find a subset of them such that the product of each element in the subset is both a rational square and an algebraic square. Such a subset can be found if we have more a, b pairs than there are entries in the exponent vector. In this case we are guaranteed to find a non-zero linear combination of the exponent vectors such that each entry is 0 reduced modulo 2. Since each entry is 0 (mod 2), we know that each prime occurs as a factor an even number of times and hence we have rational and algebraic squares.

In addition to the line sieve method we have just described, there is another method called the *lattice sieve*, which is described in Lenstra, et al [len93, pages 43-49]. This method is more complicated and theoretically faster by a factor of $\log q$ where q is a prime number "in the middle" of the factor base. A description of this method is out of scope for this thesis.

4.3 The linear algebra

From the sieve step we have a matrix A where each row in A is an exponent vector from a pair a, b. In this step we seek a non-zero vector $\mathbf{x}$ satisfying the system of equations

$$\mathbf{x}^{\intercal} A \equiv \mathbf{0} \pmod 2.$$

Let $\mathcal{T}$ be the set of all a, b pairs found in the sieve stage. Then, $\mathbf{x}$ represents a subset of pairs in $\mathcal{T}$. In particular, element i specifies whether exponent vector i is part of the solution. Let us denote the actual subset by $\mathcal{S}$. Then,

$$\prod_{(a,b)\in\mathcal{S}} (a - bm) \tag{4.2}$$

is a rational square and

$$\prod_{(a,b)\in\mathcal{S}} (a - b\alpha) \tag{4.3}$$

is seemingly an algebraic square (see the next section for a more thorough explanation).

The size of the matrix depends on the number of elements in the factor base. For large integers n (say over 150 digits), the matrix can have millions of rows and columns. Methods like Gaussian elimination will be too slow for matrices of this size. Efficient algorithms for this step are outside the scope of this thesis. For the interested reader, we suggest checking out faster algorithms such as *block Wiedemann* [wie86] and *block Lanczos* [cop93].

4.4 Some obstructions

So far in this chapter we have assumed that (4.3) is an algebraic square. In this section we will see that this need not be true, and we will come up with way to fix this problem.

Assume that we have performed the matrix step and have a subset of pairs a, b such that the sum of their exponent vectors are 0 modulo 2. Let $\mathcal{S}$ denote this set of a, b pairs and let

$$\beta = \prod_{(a,b)\in S} (a - b\alpha). \tag{4.4}$$

Then $\langle\beta\rangle$ is an ideal in $\mathcal{O}_{\mathbb{Q}(\alpha)}$.

However, what we really want is an element that is a square in $\mathbb{Z}[\alpha]$, not an ideal in $\mathcal{O}_{\mathbb{Q}(\alpha)}$. Therefore we need to consider the following obstructions.

(1) The ideal $\langle\beta\rangle$ is not necessarily the square of an ideal $\mathfrak{a}$ that lies in $\mathbb{Z}[\alpha]$.

(2) Even if $\langle\beta\rangle = \mathfrak{a}^2$ for some ideal $\mathfrak{a}$ in $\mathcal{O}_{\mathbb{Q}(\alpha)}$, it may happen that $\mathfrak{a}$ is not a principal ideal.

(3) Even if $\langle\beta\rangle = \langle\gamma\rangle^2$ for some $\gamma \in \mathcal{O}_{\mathbb{Q}(\alpha)}$, it may not be that $\beta = \gamma^2$.

(4) Even if $\beta = \gamma^2$ for some $\gamma \in \mathcal{O}_{\mathbb{Q}(\alpha)}$, if may not be that $\gamma \in \mathbb{Z}[\alpha]$.

These obstructions might look very daunting, but they can actually be overcome with two simple modifications to the algorithm.

The following result helps us overcome obstruction (4).

Theorem 4.4.1. *Let $f(x)$ be a monic irreducible polynomial over $\mathbb{Z}$ with a root $\alpha \in \mathbb{C}$. Let $\mathcal{O}_{\mathbb{Q}(\alpha)}$ be the ring of algebraic integers in $\mathbb{Q}(\alpha)$ and let $\beta \in \mathcal{O}_{\mathbb{Q}(\alpha)}$. Then $f'(\alpha)\beta \in \mathbb{Z}[\alpha]$.*

See Crandall, et al [cra05, page 288] for the proof.

We can then use the following as our squares, replacing (4.2) and (4.3):

$$f'(m)^2 \prod_{(a,b)\in\mathcal{S}} (a - bm) \tag{4.5}$$

is the new rational square and

$$f'(\alpha)^2 \prod_{(a,b)\in\mathcal{S}} (a - b\alpha) \tag{4.6}$$

is the new algebraic square. Because of Theorem 4.4.1, (4.6) and its square root are elements in $\mathbb{Z}[\alpha]$ which is what we want.

The remaining obstructions (1), (2) and (3) can be circumvented using a very simple, but probabilistic idea known as *quadratic characters* first introduced by Adleman.

We explain the idea first using rational integers. Let's pretend that we cannot determine the sign of an integer, but we can determine the prime factorization. Then both $4 = 2^2$ and $-4 = -2^2$ would look like squares, although -4 is not a square. However, by using the Legendre symbol with the correct moduli we can tell that -4 is not a square, as $\left(\frac{-4}{7}\right) = -1$. Given an arbitrary integer x and k different primes $p_1, p_2, \ldots, p_k$, if $\left(\frac{x}{p_i}\right) = 1$

for all i then the probability that x is not a square is 2^{-k} (heuristically). For a large enough set of primes, this is a robust test that x is a square. Naturally, if at least one of the legendre symbols are -1, then x is not a square.

Consider β from Equation 4.4. We can use a similar test to check if β is a (probable) square. The following result allows us to use Legendre symbols in the same way as described above.

Theorem 4.4.2. *Let $f(x)$ be a monic, irreducible polynomial over $\mathbb{Z}$ and let $\alpha \in \mathbb{C}$ be a root. Assume that q is an odd prime number and s is an integer satisfying $f(x) \equiv 0 \pmod q$ and $f'(x) \not\equiv 0 \pmod q$. Let $\mathcal{S}$ be a set of pairs (a, b) such that $\gcd(a, b) = 1$ and $q \nmid a - bs$, and $f'(\alpha)^2 \prod_{(a,b)\in\mathcal{S}}(a - b\alpha)$ is a square in $\mathbb{Z}[\alpha]$. Then*

$$\prod_{(a,b)\in\mathcal{S}} \left(\frac{a - bs}{q}\right) = 1.$$

See Crandall, et al [cra05, page 290] for the proof.

With this result, we can use the idea from the example with integers above. Assume that we have k different pairs (q_i, s_i) for $i = 1, 2, \ldots, k$ satisfying the conditions in Theorem 4.4.2, and an algebraic integer $f'(\alpha)^2 \prod_{(a,b)\in\mathcal{S}}(a - b\alpha)$ we wish to test where $\mathcal{S}$ is a set of different pairs (a, b) where $\gcd(a, b) = 1$. If $\prod_{(a,b)\in\mathcal{S}} \left(\frac{a-bs_i}{q_i}\right) = 1$ for each i, then $f'(\alpha)^2 \prod_{(a,b)\in\mathcal{S}}(a - b\alpha)$ is a square with probability $1 - 2^{-k}$.

This information needs to be incorporated in the algorithm. We add a third factor base which we will call the *quadratic character factor base*. This factor base contains k pairs q, s satisfying the conditions in Theorem 4.4.2. In particular, all q are larger than the primes in the algebraic factor base.

In addition, we add k entries to the exponent vector that is created during the sieve stage, one entry for each pair q_i, s_i. For a given pair a, b we will set entry i as follows:

- If $\left(\frac{a-bs_i}{q_i}\right) = -1$, set the entry to 1.
- If $\left(\frac{a-bs_i}{q_i}\right) = 1$, set the entry to 0.

Finding a subset $\mathcal{S}$ of pairs a, b such that the sum of the corresponding exponent vectors is 0 modulo 2 ensures that we will have

$$\prod_{(a,b)\in\mathcal{S}} \left(\frac{a - bs_i}{q_i}\right) = 1 \quad \text{for } i = 1, 2, \ldots, k,$$

which implies that

$$\prod_{(a,b)\in\mathcal{S}} (a - b\alpha) = \gamma^2 \text{ for some } \gamma \in \mathcal{O}_{\mathbb{Q}(\alpha)}$$

with heuristic probability $1 - 2^k$.

Crandall, et al [cra05] suggests using $k = \lfloor 3 \lg n \rfloor$ different pairs (q, s) in the factor base.

4.5 Square roots and factorization

From the last step we have found a rational square $f'(m)^2 \prod_{(a,b)\in\mathcal{S}}(a - bm)$, and an algebraic integer $f'(\alpha)^2 \prod_{(a,b)\in\mathcal{S}}(a - b\alpha)$ which we now assume is a square.

4.5.1 Finding the rational square root

Taking the rational square root of the rational square is easy, since we can use the known factorization of each $a - bm$ for each $(a, b) \in \mathcal{S}$ where $\mathcal{S}$ is the set of pairs (a, b) found in the linear algebra step. This product is of the form $\prod p_i^{e_i}$ where i runs over all prime numbers in the rational factor base, and all e_i are even. We are interested in the square root modulo n, which is

$$f'(m) \prod p_i^{e_i/2} \pmod{n}.$$

4.5.2 Finding the algebraic square root

Taking the square root of our algebraic square $\beta = f'(\alpha)^2 \prod_{(a,b)\in\mathcal{S}}(a - b\alpha)$ is not as straightforward as in the rational case. We know the factorization of the square into prime ideals of $\mathcal{O}_{\mathbb{Q}(\alpha)}$, but since we don't know the generators of these prime ideals we need a different approach.

Here we describe an algorithm given by Couveignes [cou93] which only works for $f(x)$ where the degree d is odd.

Let us express our square as $\beta = \sum_{i=0}^{d-1} b_i \alpha^i$. Then we seek $\gamma = \sum_{i=0}^{d-1} a_i \alpha^i \in \mathbb{Z}[\alpha]$ such that $\gamma^2 = \beta$.

The main idea is to consider β as an element in F_{p^d}. Let $\beta_q = \sum_{i=0}^{d-1} c_i \alpha^i$ where $c_i \in \mathbb{Z}_p$ is b_i reduced modulo p. We can then find the square root γ_p of β_p using standard algorithms for square roots in finite fields (see Section 5.3.1). Let's assume that we always pick the correct γ out of the two possible square roots (we will address the problem of picking the correct square root later). Assume we have several primes p_i (such that f remains irreducible modulo p_i, otherwise we are not in a finite field) for which we calculate the square root γ_{p_i} of β_{q_i}. Then we can obtain $\gamma = \sqrt{\beta}$ by applying the Chinese Remainder Theorem:

$$\begin{aligned}
\gamma &\equiv \gamma_{p_1} \pmod{p_1} \\
\gamma &\equiv \gamma_{p_2} \pmod{p_2} \\
\gamma &\equiv \gamma_{p_3} \pmod{p_3} \\
&\vdots \\
\gamma &\equiv \gamma_{p_k} \pmod{p_k}.
\end{aligned}$$

This assumes that we have been using enough primes. An upper bound for the product of the primes is given by Couveignes [cou93]:

$$\prod_{i=1}^{k} p_i \leq d^{(d+5)/2} \cdot n \cdot \left(2u\sqrt{d}n^{1/d}\right)^{|\mathcal{S}|/2} \tag{4.7}$$

where

$$u = \max_{(a,b)\in\mathcal{S}} \left(\max\left(|a|, |b|\right)\right),$$

that is, the maximal value of $|a|$ and $|b|$ among the smooth pairs a, b in $\mathcal{S}$. This bound also assumes that d is chosen to satisfy $d^{2d^2} < n$.

There exists a tighter upper bound [cou93], but it requires calculating approximations to alle roots α_i of $f(x)$. The bound given in (4.7) will ensure that this algorithms runs efficiently for n with 50-60 digits.

As mentioned earlier, β_p has two different square roots. Since the norm function is multiplicative ($N(ab) = N(a)N(b)$) and the degree d of the extension $\mathbb{Z}[\alpha]/\mathbb{Z}$ is odd, we have $N(-a) = -N(a)$. Therefore we can use the sign of the norm to determine which square root to pick. Let γ_1 and γ_2 be the two roots of β_p for a modulus p. The correct square root is the one that is congruent to the square root of the norm of β modulo p, which is possible to calculate since we know its factorization into prime ideals whose norms are known.

The algorithm described here only works for $f(x)$ with odd degree. See Nguyen [ngu98] for a description of an efficient algorithm that works for any degree.

4.5.3 Getting a non-trivial factor

Now that we have the square roots, we can finally try to obtain a non-trivial factorization of n by taking $\gcd(n, u - v)$.

We have the rational square root

$$u = f'(m) \prod p_i^{e_i/2} \pmod{n},$$

and the algebraic square root mapped into $\mathbb{Z}_n$ by using our homomorphism (4.1)

$$v = \sigma(\gamma).$$

Then we evaluate $g = \gcd(n, u - v)$. If $1 < g < n$ we have found a non-trivial factor. If not, we must either find another linear combination such that (4.5) and (4.6) are squares, or do more sieving to find more smooth pairs a, b.

4.6 Summary

Here we present detailed pseudocode for the entire NFS algorithm.

Input: An integer n.

1. **Setup**

   a. Ensure that n is odd, composite and not a power. If any of these conditions fail, abort the algorithm and give an appropriate error message.

   b. Set a degree d such that $d^{2d^2} < n$, let $m = \lfloor n^{1/d} \rfloor$, then find a degree d polynomial $f(x)$ using the base-m algorithm.

   c. Check if $f(x)$ has non-trivial factors. If yes, then $f(x) = g(x)h(x)$ for some non-constant polynomials $g(x), h(x)$. Return the non-trivial factorization $n = g(m)h(m)$. To factor $f(x)$, the algorithm described in Lenstra, et al [len82] can be used, or if the degree of $f(x)$ is at most 3, use the algorithm described in Section 5.4.

   d. Determine the upper bounds B_1, B_2 for the rational and algebraic factor bases, respectively. To achieve asymptotically optimal run-time, choose $B_1 = B_2 = e^{(8/9)^{1/3}(\ln n)^{1/3}(\ln \ln n)^{2/3}}$.

e. Calculate all rational primes up to B_1. This can be done using algorithms like the sieve of Eratosthenes [era13] or the sieve of Atkin [atk13]. The latter is faster, but more complicated to implement.

f. Calculate all algebraic primes represented by the two integers (p, r) such that $p \leq B_2$. For each p, find the set $R(p) = \{r | f(r) \equiv 0 \pmod{p} \text{ and } r \in \{0, 1, \ldots, p-1\}\}$. Use an efficient algorithm for finding the roots of $f(x)$ modulo p, for instance the one described in Section 5.5.

g. Let $k = \lfloor 3 \ln n \rfloor$. Find the first k primes $q_1, \ldots, q_k > B_2$ such that there is an s_k satisfying $f(s_k) \equiv 0 \pmod{q_k}$ and $f'(s_k) \not\equiv 0 \pmod{q_k}$. The pairs (q_i, s_i) comprise the quadratic character factor base.

h. Let $V = 1 + \pi(B_1) + B' + k$ be the size of the exponent vector. Here $\pi(B_1)$ is the number of rational primes $\leq B_1$ and $B' = \sum_{p \text{ prime}, p \leq B_2} |R(p)|$ is the number of primes in the algebraic factor base.

2. **The sieve**

a. Pick an integer M, the max line width in the sieve.

b. For each integer $b \geq 1$, sieve the interval $-M \leq a \leq M$ and find values a, b such that $\gcd(a, b) = 1$ and $(a - bm) \cdot N(a - b\alpha)$ is smooth with respect to both factor bases. Proceed until we have at least V smooth pairs.

c. For each smooth element (a, b), create an exponent vector. The first element is the sign of $a - bm$, 1 for negative, 0 for positive. For the next $\pi(B_1)$ elements, set the bit for p to 1 if $a - bm$ is divisible by p_i^e for an odd e. For the next B' elements, set the bit for (p, r) to 1 if $N(a - b\alpha)$ is divisible by the prime ideal represented by (p, r) raised to an odd power. For the last k elements, set the bit for (q, s) to 1 if the Legendre symbol $\left(\frac{a-bs}{q}\right) = -1$ and set the bit to 0 otherwise.

3. **The linear algebra**

a. Create a matrix A where each exponent vector found in the sieve step has its own row.

b. Solve the system $\mathbf{x}^\intercal A \equiv \mathbf{0} \pmod{2}$ for the unknown vector $\mathbf{x}$ using some suitable algorithm (block Wiedemann or block Lanczos, or even Gaussian elimination if the matrix is small enough).

c. Let $\mathcal{S}$ be the set of a, b pairs found from the vector x

4. **Square root**

a. Use the known factorization of the square $u^2 = f'(m)^2 \prod_{(a,b) \in \mathcal{S}} (a - bm)$ to find v modulo n.

b. Use a suitable algorithm, such as the algorithm by Couveignes, to find the square root γ of $f'(\alpha)^2 \prod_{(a,b) \in \mathcal{S}} (a - b\alpha)$. Then calculate $v = \phi(\gamma) \pmod{n}$ using the ring homomorphism that maps $\alpha \in \mathbb{Z}[\alpha]$ to $m \in \mathbb{Z}_n$.

5. **Find a factor**

a. Return $\gcd(u-v, n)$. If this is a trivial factor, find another linearly dependent vector from the matrix step and do the square root step again. If this fails, do more sieving to find more smooth pairs, raising the factor base bounds B_1 and B_2 if necessary.

Chapter 5

Algorithms used in the NFS

This chapter contains descriptions of algorithms that are used to solve subtasks in the number field sieve. These algorithms perform common tasks such as factoring polynomials and taking square roots, and are not specific to the NFS. In section 4 we will describe the NFS including the underlying algorithms that are specific to the NFS.

5.1 Arithmetic in a number field

This section describes arithmetic in number fields, but it is also valid for the number ring $\mathbb{Z}[\alpha]$.

Assume that $f(x)$ is a monic irreducible polynomial of degree n and α is a root. Then, $\mathbb{Q}(\alpha)$ is a number field of degree n, and the elements are of the form

$$a_0 + a_1\alpha + a_2\alpha^2 + \cdots + a_{n-1}\alpha^{n-1}$$

where the a_i are elements in $\mathbb{Q}$.

The result of the addition of two elements $\chi, \upsilon \in \mathbb{Q}(\alpha)$ given by

$$\begin{aligned}\chi &= a_0 + a_1\alpha + a_2\alpha^2 + \cdots + a_{n-1}\alpha^{n-1},\\ \upsilon &= b_0 + b_1\alpha + b_2\alpha^2 + \cdots + b_{n-1}\alpha^{n-1}\end{aligned}$$

is simply pointwise addition of the coefficients:

$$\chi + \upsilon = (a_0 + b_0) + (a_1 + b_1)\alpha + \cdots + (a_{n-1} + b_{n-1})\alpha^{n-1}.$$

To multiply two elements $\chi, \upsilon \in \mathbb{Q}(\alpha)$ as above, we first perform regular multiplication as one would multiply two polynomials, followed by a reduction based on the fact that $f(\alpha) = 0$ (since α is a root of $f(x)$). The multiplication before reduction gives

$$\psi = \chi \cdot \upsilon = \sum_{i=0}^{2n-2} \left(\prod_{0 \le j,k < n, j+k=i} a_j b_k \right) \alpha^i.$$

The result of the multiplication, ψ, can be viewed as a polynomial in α having degree of up to $2n - 2$. By adding or subtracting multiples of $f(\alpha)$ (which is equal to 0), we can bring the degree down to $n - 1$. For each $i = 2n - 2, 2n - 1, \ldots, n + 1, n$ subtract $-a_i\alpha^{i-n}f(\alpha)$ from ψ. The resulting ψ will have degree of no more than $n - 1$. This procedure can also be used to reduce a polynomial $f(x)$ modulo a monic polynomial $g(x)$, and it works for polynomials over $\mathbb{Q}$, $\mathbb{Z}$ and $\mathbb{Z}_p$ for primes p.

Example 5.1.1. Let $f(x) = x^2 + x + 1$ be an irreducible polynomial over $\mathbb{Q}$ and let α be a root. Then $\mathbb{Q}(\alpha)$ is a number field having elements of the form $a + b\alpha$ with $a, b \in \mathbb{Q}$. Let $\chi = 2 + \alpha$ and $\upsilon = 1 + 3\alpha$. Then the product is

$$\psi = \chi \cdot \upsilon = 2 + 7\alpha + 3\alpha^2,$$

which has degree 2. We can reduce this product to degree 1 by subtracting $3f(\alpha) = 3\alpha^2 + 3\alpha + 3$. After doing this we end up with the reduced element

$$\psi = -1 + 4\alpha.$$

5.2 Norm of an algebraic number

Throughout this section, let $\mathbb{Q}(\alpha)$ be a number field of degree n, and let α be a root of the minimal polynomial of degree n.

Theorem 3.2.2 gave us a short and implementation-friendly expression for the norm of an algebraic number of the form $a - b\alpha \in \mathbb{Q}(\alpha)$. This expression was derived from the definition (Definition 3.2.3) of the norm in terms of the *conjugates* of the number field; the set of embeddings of $\mathbb{Q}(\alpha)$ into $\mathbb{C}$ over $\mathbb{Q}$.

However, attempting to use the same definition on a general element, an element of the form

$$\beta = a_0 + a_1\alpha + a_2\alpha^2 + \cdots + a_{n-1}\alpha^{n-1}, \tag{5.1}$$

gets unwieldy very fast. We already saw in Example 3.2.8 under Definition 3.2.3 that the expression of the norm in a degree 3 number field with a very simple minimal polynomial needed a fair amount of work to determine. We would rather not repeat this procedure for higher degree number fields with more complex minimal polynomials, so we seek an easier approach.

It turns out that we can use techniques from linear algebra to calculate the norm. Let $B = \{b_1, b_2, \ldots, b_n\}$ be a linearly independent basis of the vector space $\mathbb{Q}(\alpha)$ over $\mathbb{Q}$ and let (5.1) be the element we wish to calculate the norm of. Then, for each element in the basis, create a column vector $\mathbf{c}_i = (c_{i,0}, c_{i,1}, \ldots, c_{i,n-1})^\intercal$ where $c_{i,j}$ is the coefficient in front of the term α^j of the product $\beta \cdot b_i$. Then the norm of β is

$$N(\beta) = \det \begin{bmatrix} \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_n} \end{bmatrix}$$

where the columns of the matrix consists of the $\mathbf{c}_i$ column vectors.

This method is valid for any choice of linearly independent basis, for instance $B = \{1, \alpha, \alpha^2 \ldots, \alpha^{n-1}\}$.

5.3 Calculate square root modulo a prime p

In this section we will present an efficient algorithm for finding an integer x such that $x^2 \equiv a \pmod{p}$ where p is an odd prime. This algorithm is used as part of the algebraic square root stage.

First, we must check that the square root exists. Roughly half the possible values of a have no square root modulo p. To check whether a square root exists for a given a, we compute the *Legendre symbol* $\left(\frac{a}{p}\right)$, which is defined as follows:

$$\left(\frac{a}{p}\right) = 1 \quad \text{if the square root of } a \text{ modulo } p \text{ exists.}$$

$$\left(\frac{a}{p}\right) = -1 \quad \text{if the square root of } a \text{ modulo } p \text{ doesn't exist.}$$

$$\left(\frac{a}{p}\right) = 0 \quad \text{if } p \text{ divides } a.$$

The following congruence allows us to calculate the Legendre symbol:

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

By considering Z_p^* as a group with multiplication of order p, we easily see that the Legendre symbol has to be -1, 0 or 1.

This should be implemented using a fast exponentiation algorithm running in time $O(\log p)$, such as Algorithm A described by Knuth [knu98, page 462].

We break down the calculation of the square root into three cases, depending on p:

$$\begin{aligned} &p \equiv 3 \pmod{4}, \\ &p \equiv 1 \pmod{8} \text{ and} \\ &p \equiv 5 \pmod{8}. \end{aligned}$$

The case where $p \equiv 3 \pmod{4}$

This is the easy case, and the solution is given by

$$x \equiv a^{(p+1)/4} \pmod{p}.$$

Since a is a quadratic residue, we have that $a^{(p-1)/2} \equiv 1 \pmod{p}$. This gives

$$\begin{aligned} x^2 &\equiv a^{(p+1)/2} \pmod{p} \\ &\equiv a \cdot a^{(p-1)/2} \pmod{p} \\ &\equiv a \pmod{p}. \end{aligned}$$

It is tempting to choose p satisfying $p \equiv 3 \pmod{4}$ if we have any choice in the matter, which we actually do when computing the square root of an algebraic number using Couveignes' algorithm (see section 4.5.2).

The case where $p \equiv 5 \pmod{8}$

For this case there are two subcases, depending on whether $a^{(p-1)/4}$ is -1 or 1 modulo p. If it is 1, the desired answer is $x \equiv a^{(p+3)/8} \pmod{p}$. If it is -1, the answer is $x \equiv 2a \cdot (4a)^{(p-5)/8} \pmod{p}$. Consult Cohen [coh93, page 31] for the proof.

The case where $p \equiv 1 \pmod 8$

This is the most difficult case and here we give an algorithm which is due to Tonelli and Shanks. The pseudocode is shown in Algorithm 1. See Cohen [coh93, pages 32-33] for the proof.

Algorithm 1 Tonelli-Shanks' algorithm for finding the square root of a modulo p

function TONELLI-SHANKS(a,p)
  Write $p-1$ as $2^e \cdot q$ for odd q
  Try random integers n, $0 < n < p$ until we find one that satisfies $\left(\frac{n}{p}\right) = 1$
  $z \leftarrow n^q \pmod p$ ▷ Initialize a few intermediate variables
  $y \leftarrow z$
  $r \leftarrow e$
  $x \leftarrow a^{(q-1)/2} \pmod p$
  $b \leftarrow ax^2 \pmod p$
  $x \leftarrow ax \pmod p$
  loop ▷ Loop until we find (or fail to find) the square root
    if $b \equiv 1 \pmod p$ **then**
      return x ▷ We found the square root
    end if
    Find the smallest $m \geq 1$ such that $b^{2^m} \equiv 1 \pmod p$
    if $m = r$ **then** ▷ Can be skipped if we ensure that $\left(\frac{a}{p}\right) = 1$ beforehand
      return "a is not a quadratic residue"
    end if
    $t \leftarrow y^{2^{r-m-1}} \pmod p$ ▷ Reduce the exponent
    $y \leftarrow t^2 \pmod p$
    $r \leftarrow m$
    $x \leftarrow xt \pmod p$
    $b \leftarrow by \pmod p$
  end loop
end function

5.3.1 Square root in finite fields F_{p^n}

The procedure is essentially the same as the one desribed in Section 5.3. We use the Tonelli-Shanks algorithm, as $F_{p^n}^*$ (for odd p) is a cyclic group with even order which is the same setting as for F_p^*. For more details read Briggs [bri98, pages 45-48].

5.4 Find the factors of a polynomial over $\mathbb{Z}$ of degree 3

When the number n we want to factor is "small" (say, 60 digits or less), it is fine to let the number field be generated by a polynomial of degree 3. For such small degrees we don't have to resort to the more complicated algorithms of Lestra, et al [len82]. We will describe a simple algorithm that works for degrees no larger than 3.

First, we notice that we only need to check for linear factors. If $f(x)$ is reducible and of degree 3, there are either 3 linear factors or 2 factors with degrees 1 and 2.

The following theorem is helpful in order to make an algorithm.

Theorem 5.4.1. *Let $f(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + x^n \in \mathbb{Z}[x]$ be a monic polynomial. If $f(x)$ has a root $a \in \mathbb{Q}$, then $a \in \mathbb{Z}$ and $a|a_0$.*

Proof. This proof follows Bhattacharya, et al [bha94]. Let $a = \alpha/\beta$ where $\alpha, \beta \in \mathbb{Z}$ and $\gcd(\alpha, \beta) = 1$. Then

$$a_0 + a_1\left(\frac{\alpha}{\beta}\right) + \cdots + a_{n-1}\left(\frac{\alpha^{n-1}}{\beta^{n-1}}\right) + \frac{\alpha^n}{\beta^n} = 0.$$

Multiply the above equation by β^{n-1}, and move all terms with fractions to the right-hand side to obtain

$$a_0\beta^{n-1} + a_1\alpha\beta^{n-2} + \cdots + a_{n-1}\alpha^{n-1} = -\frac{\alpha^n}{\beta}.$$

Since $\alpha, \beta \in \mathbb{Z}$, the entire left-hand side of the above equation is in $\mathbb{Z}$. Then the right-hand side, $-\alpha^n/\beta$, must also be in $\mathbb{Z}$ and β must be ± 1. The last equation also shows that $\alpha|a_0$. α divides every term except one, so it must divide the last one. Hence, $a = \pm\alpha \in \mathbb{Z}$ and $a|a_0$. $\square$

Then, we can use the following simple algorithm for checking for a root:

1. Find the prime factorization of $a_0 = \prod_i p_i^{r_i}$ where p_i are primes and r_i are exponents. $a_0 < m = \lfloor n^{1/3} \rfloor$ is small enough that algorithms like Pollard's rho algorithm can find the factors quickly, assuming that n is not larger than 60 digits.

2. Generate all $b = \prod_i p_i^{s_i}$ such that $0 \leq s_i \leq r_i$. b is then a divisor of a_0 (easy). For each such number, check if $f(b) = 0$ or $f(-b) = 0$. If that happens, we found a root.

If we assume that n has at most 60 digits we can actually skip the factorization of a_0 and get away with a slower algorithm that is even easier to implement: Try all a such that $1 < a \leq \sqrt{m}$. For all a dividing m, check if $f(a), f(-a), f(m/a)$ or $f(-m/a)$ is 0. If $f(b) = 0$ where b is the value we tested, then $x - b$ is a linear factor. For n with 60 digits this is a loop with 10^{10} iterations, which is still less work than what will be done in the sieve stage.

When all linear factors are found, what remains of $f(x)$ after dividing out the linear factors is either 1, a degree 2 polynomial or an irreducible degree 3 polynomial in the case where no factors were found.

5.5 Find the roots of a polynomial in $\mathbb{Z}_p[x]$

Let $f(x)$ be a polynomial of degree n with coefficients in $\mathbb{Z}_p$. We seek an efficient algorithm for finding all the roots modulo p. This algorithm is required in order to find the prime ideals in the algebraic factor base efficiently.

A naïve way is to evaluate $f(i)$ for all $i = 0, 1, \ldots, p-1$ and check whether $f(i) \equiv 0 \pmod p$. However, this method requires p evaluations, and is infeasible if p is large. We want a method that is sublinear in p.

A better method involves the polynomial $g(x) = x^p - x$, also over $\mathbb{Z}_p[x]$. We will show that this polynomial is identical to $f(x) = \prod_{i=0}^{p-1}(x-i)$. $f(x)$ has degree p and by construction it has the roots $0, 1, \ldots, p-1$. $g(x)$ has degree p. By Fermat's little theorem we have for any prime number p, $a^p \equiv a \pmod p$ for every integer a, so therefore x^p must equal x for each $x \in \mathbb{Z}_p$. Therefore $g(x) = 0$ for all $a = 0, 1, 2, 3, \ldots, p-1$. Hence $g(x)$ has p different roots. Since $g(x)$ has degree p, all the roots have multiplicity 1. $g(x)$ and $f(x)$ have the same degree and the same roots so they must be equal.

Algorithm 2 uses the above polynomial, and works for primes p larger than 2. Is faster as it doesn't scale linearly in p. Let $f(x)$ be the polynomial we want to find the roots of. First, we divide out square factors of $f(x)$ by dividing out $\gcd(f(x), f'(x))$. Then, linear factors are isolated by calculating $a(x) = \gcd(f(x), x^p - x)$. After that we attempt to split the remaining polynomial by taking the greatest common divisor of the isolated factors $a(x)$ and $(x-a)^{(p-1)/2} - 1$ for some random integer a, which has a $1 - (1/2)^{d-1}$ chance of splitting $a(x)$ (since $(x-a)^{(p-1)/2} - 1$ has exactly half the numbers between 1 and $p-1$ as roots). Lastly, we perform the split again on each half until we have split the polynomial into smaller polynomials of degree at most 2, from which we can obtain the roots easily (for degree 2 polynomials, we use the well-known formula for the roots of a quadratic equation).

One can omit the handling of $a(x)$ of degree 2 for an even easier (and possibly slightly slower) implementation, since a degree 2 polynomial will eventually be split into two linear factors.

This algorithm does not work for $p = 2$, but in this case we use the naïve algorithm since there are only two values to test. For small values of p the naïve version is likely to be faster. It is advisable to use experimentation in order to find the crossover point of p for when to use which algorithm.

5.6 Check if polynomial in $\mathbb{Z}_p[x]$ is irreducible

In section 5.5, we saw that we could isolate linear factors by calculating $\gcd(f(x), x^p - x)$. In particular, if $\gcd(f(x), x^p - x) = 1$, $f(x)$ had no linear factors. This method can be generalised: $x^{p^k} - x$ is the product of all monic, irreducible polynomials of degree dividing k. To check if an arbitrary polynomial of degree n is irreducible, it suffices to check if $\gcd(f(x), x^{p^k} - x) = 1$ for every $1 \leq 1 \leq \lfloor \frac{n}{2} \rfloor$. This algorithm is described by Menezes [men01, page 155], and the pseudocode is given in algorithm 3. The algorithm requires that $f(x)$ is monic, but $f(x)$ is easily converted to a monic polynomial by dividing the polynomial by a_{n-1}.

This algorithm is used in Couveignes' algorithm for taking the square root of an algebraic integer, see Section 4.5.2.

Algorithm 2 Find roots of a polynomial modulo p. All arithmetic is done in Z_p.

1: **function** FIND-ROOTS($f(x)$,p)
2: $a(x) \leftarrow \gcd(f(x), f'(x))$ ▷ Ensure that no root have multiplicity ≥ 2
3: $a(x) \leftarrow \gcd(x^p - x, a(x))$ ▷ Isolate linear factors
4: **if** $a(x) = 0$ **then** ▷ Check if 0 is a root
5: Output the root 0
6: $a(x) \leftarrow a(x)/x$
7: **end if**
8: **if** degree($a(x)$)=0 **then** ▷ Output root and terminate if $a(x)$ has small degree
9: **return**
10: **else if** degree($a(x)$)=1 **then**
11: Output the root $-a_0 a_1^{-1}$ ▷ $a(x) = a_1 x + a_0$
12: **return**
13: **else if** degree($a(x)$)=2 **then**
14: $d \leftarrow a_1^2 - 4a_0 a_2$ ▷ $a(x) = a_2 x^2 + a_1 x + a_0$
15: $\epsilon \leftarrow \sqrt{d}$ ▷ Use the algorithm from section 5.3
16: Output the roots $(-a_1 + \epsilon)(2a_2)^{-1}$ and $(-a_1 - \epsilon)(2a_2)^{-1}$
17: **return**
18: **end if**
19: **repeat** ▷ Random splitting
20: Choose a random $a \in F_p$
21: $b(x) \leftarrow \gcd((x-a)^{(p-1)/2} - 1, a(x))$
22: **until** degree($b(x)$) > 0 **and** degree($b(x)$)<degree($a(x)$)
23: Recursively call the algorithm with $b(x)$ and $a(x)/b(x)$, starting at line 8.
24: **end function**

Algorithm 3 Checks if the monic polynomial $f(x)$ is irreducible modulo p.

1: **function** TEST-IRREDUCIBILITY($f(x)$,p)
2: $u(x) \leftarrow x$
3: **for** $i \leftarrow 1$ **to** $\lfloor \frac{\text{degree}(f(x))}{2} \rfloor$ **do**
4: $u(x) \leftarrow u(x)^p \bmod f(x)$ ▷ Use fast exponentiation
5: $d(x) = \gcd(f(x), u(x) - x)$
6: **if** degree($d(x)$) > 1 **then**
7: **return** "reducible"
8: **end if**
9: **end for**
10: **return** "irreducible"
11: **end function**

Chapter 6

Implementation

In this chapter we will describe our NFS implementation in more depth.

Our NFS program is written in C, with the GNU Multiple Precision Arithmetic Library[1] (GMP) as the only external dependency. The source code is given in Appendix A.

This chapter will be divided into several sections, one for each phase of the algorithm.

6.1 Initialization and polynomial selection

We chose to implement the base-m algorithm for determining the polynomial $f(x)$. The program accepts any positive integer d, from this $m = \lfloor n^{1/d} \rfloor$ is calculated and the coefficients of $f(x)$ are derived from the base-m expansion of n.

The rational factor base is determined using a straightforward implementation of the sieve of Eratosthenes. A description of this algorithm can be found on Wikipedia [era13].

To determine the algebraic factor base, we go through the p values found for the rational factor base and try to find the r values by finding all roots of $f(x)$ modulo p. This is presently accomplished by two algorithms: For small p (less than 7) we naïvely evaluate $f(r)$ for all $r = 0, 1, \ldots, p-1$. For larger p we use a more efficient algorithm that finds all roots of $f(x)$ modulo p without evaluating $f(r)$ for every $r = 0, 1, \ldots, p-1$. This algorithm is described in Section 5.5.

Since neither C nor GMP have support for polynomials, we implemented basic subroutines for doing arithmetic on polynomials modulo p, including routines for multiplication, division (including remainder), reduction modulo a polynomial $f(x)$, greatest common divisor and fast exponentiation.

Our implementation does not attempt to factor $f(x)$ into non-trivial factors if the degree of $f(x)$ is larger than 3. If the degree is at most 3, the algorithm described in Section 5.4 is used. If $f(x)$ is reducible of degree at most 3, the program will output two non-trivial factors and terminate.

6.2 The sieve

We have implemented the line sieve. For each $b = 1, 2, 3, \ldots$ in turn, assume that b is fixed and do the following until we have enough pairs a, b (at least as many as there

[1] `http://gmplib.org/`

are elements in the factor bases). Initialize an array with one element for each a such that $-M \leq a \leq M$ for some M. The element corresponding to (a, b) is initialized with $\lfloor \lg(a-bm) \rfloor + \lfloor \lg N(a-b\alpha) \rfloor$. We chose to use logarithms here to avoid an excessive amount of division operations. The approximation of the base 2 logarithm of a given number x can be calculated efficiently by counting the number of bits in the binary representation of x. The GMP library has a built-in function that performs this on large integers.

For each rational prime p, we find the smallest $a \geq -M$ such that $a - bm \equiv 0 \pmod{p}$. Then, for each integer $k \geq 0$ such that $a + kp \leq M$, we subtract $\lfloor \lg p \rfloor$ from the corresponding array element.

For each algebraic prime (p, q), we find the smallest $a \geq -M$ such that $a - br \equiv 0 \pmod{p}$. Then, we do as above: for each integer $k \geq 0$ such that $a + kp \leq M$, we subtract $\lfloor \lg p \rfloor$ from the corresponding array element.

Please note that we have used approximations to the logarithms (rounded to an integer), and we have also ignored powers of primes. In order to detect candidates for smooth numbers, we will pick the pairs (a, b) where the corresponding array element has a value below some threshold T.

For each candidate (a, b) below the threshold we perform trial division on $a - bm$ with primes from the rational factor base, and we also do trial division on $N(a - b\alpha)$ with primes from the algebraic factor base. This gives us the correct factorization, including prime powers. Whenever we find pairs that fully divide under these trial divisions, we have found a pair that is smooth. All the exponents of the primes modulo 2 are stored in the exponent vector, as well as calculating and storing the Legendre symbols $\left(\frac{a-bs}{q}\right)$ for each quadratic character (q, s). Then we tuck away the exponent vector in the matrix is to be used in the linear algebra step. In addition, we store the full factorization of each smooth pair so that we can reconstruct the rational square and square root in a later step.

The threshold T must be found via experimentation. We don't want to set it too low, or we lose smooth numbers divisible by primes with large powers. We don't want to set it too high either, or we end up doing expensive trial division on many non-smooth numbers.

6.3 The linear algebra

The current implementation uses a specialized Gauss-Jordan algorithm tailored for $GF(2)$, where each bit of an unsigned 32-bit integer holds one cell of the matrix. This reduces the runtime by a factor of approximately 32 compared to a hypothetical implementation that doesn't process multiple bits at once.

The system of linear equations $\mathbf{x}^\intercal A \equiv \mathbf{0} \pmod{2}$ has more unknowns than equations, so there will be at least one free variable. If we have k free variables we can obtain k essentially different linear combinations (a subset $\mathcal{S}$ of smooth pairs) of exponent vectors that represent rational and algebraic squares.

6.4 Square roots and factorization

The rational square root is computed directly from the set of smooth pairs $\mathcal{S}$ and the factorization of $a - bm$ for each $(a, b) \in \mathcal{S}$.

The algebraic square root is computed with an implementation of the algorithm by Couveignes described in Section 4.5.2. This algorithm depends in turn on subroutines for calculating the norm of a general element $a_0 + a_1\alpha + \cdots + a_{d-1}\alpha^{d-1}$ in a number ring $\mathbb{Z}[\alpha]$ (see Section 5.2), calculating square roots modulo p (Tonelli-Shanks algorithm, see Section 5.3) and in a finite field F_{p^n} (see Section 5.3.1).

6.5 Verifying the implementation

Case [cas03] gives a complete and detailed example of a small factorization with NFS, with $n = 45113, m = 31$ and $f(x) = x^3 + 15x^2 + 29x + 8$ found using the base-m algorithm. This example also includes factor bases and quadratic characters. After taking care of the fact that our implementation uses $a - bm$ and $a - b\alpha$ and that this article uses $a + bm$ and $a + b\alpha$, our implementation find the same factor bases, quadratic characters, smooth pairs (a, b) given by Case [cas03] are found by our program, and the example exponent vector matches the one our program finds.

6.6 Example 1: $n = 4486873$

6.6.1 Finding the polynomial and checking for irreducibility

We show in detail how the algorithm works for the input $n = 4486873 = 1193 \cdot 3761$. We chose to use a polynomial of degree 3. By taking $m = \lfloor n^{1/3} \rfloor$, we find $m = 164$. Using the base-164 expansion of n we find

$$f(x) = x^3 + 2x^2 + 134x + 161.$$

We need to check whether $f(x)$ is irreducible over $\mathbb{Z}[x]$. Since $f(x)$ is of degree 3, it must have a linear factor (or equivalently, a root in $\mathbb{Z}$) if it is reducible. By Theorem 5.4.1 a root, if it exists, must divide 161, which leaves us with the candiates $\pm 1, \pm 7, \pm 23$ and ± 161. The root cannot be positive (as $f(x) > 0$ whenever $x > 0$), and by evaluating $f(x)$ for the remaining candidates we find that $f(-1), f(-7), f(-23)$ and $f(-161)$ are all nonzero. Hence, $f(x)$ has no roots in $\mathbb{Z}$, and $f(x)$ is irreducible over $\mathbb{Z}[x]$ and we can carry on with the factorization.

6.6.2 Determining the factor bases

The factor base consists of 3 parts:

- The rational factor base with primes in $\mathbb{Z}$. This also includes the unit -1.
- The algebraic factor base with first degree prime ideals in $\mathcal{O}_{\mathbb{Q}(\alpha)}$ of the form $\langle p, \alpha - r \rangle$.
- The quadratic character factor base.

We set the upper bound for both the rational and algebraic factor bases to $B = 140$, and use 6 quadratic characters.

Using the sieve of Eratosthenes, we find the 34 rational primes as shown in Table 6.2.

(2,1)	(5,2)	(7,0)	(7,6)	(11,7)
(13,4)	(19,3)	(23,0)	(31,16)	(31,22)
(37,10)	(37,29)	(37,33)	(43,30)	(59,30)
(61,4)	(61,17)	(61,38)	(73,10)	(73,66)
(73,68)	(83,69)	(89,2)	(89,27)	(89,58)
(107,105)	(109,52)	(113,66)	(127,48)	(131,54)
(137,48)	(137,109)	(137,115)	(139,93)	

Table 6.1: Algebraic factor base for $n = 4486873$, upper bound $B = 140$. Each pair (p, r) corresponds to a prime ideal $\langle p, \alpha - r\rangle$.

2,	3,	5,	7,	11,	13,	17,	19,	23,	29,	31,	37,
41,	43,	47,	53,	59,	61,	67,	71,	73,	79,	83,	89,
97,	101,	103,	107,	109,	113,	127,	131,	137,	139		

Table 6.2: Rational factor base for $n = 4486873$, upper bound $B = 140$.

For the algebraic factor base, we take each rational prime p and attempt to find r such that $f(r) \equiv 0 \pmod{p}$. This can be accomplished by the root finding algorithm described in Section 5.5.

The 34 elements in the algebraic factor base are shown in Table 6.1.

For this n we used 6 quadratic characters. Each of these is a pair (q_i, s_i) such that q_i is a prime larger than the largest prime in the algebraic factor base; that is, $q_i > 140$ and s_i satisfies $f(s_i) \equiv 0 \pmod{q_i}$ and $f'(s_i) \not\equiv 0 \pmod{q_i}$.

The following pairs (q_i, s_i) were found:

$$(149, 1) \ \ (151, 75) \ \ (157, 91) \ \ (173, 108) \ \ (179, 6) \ \ (193, 36).$$

The sieve phase was run with $|a| \leq 10000$, $b \geq 1$ and a threshold of $T = 20$ of accepting a smooth integer. The pairs (a, b) found such that $a - bm$ and $a - b\alpha$ were smooth over their respective factor bases are shown in Table 6.3. Having a rational factor base of 34 primes, an algebraic factor base of 34 prime ideals, 6 quadratic characters and sign of $a - bm$ results in an exponent vector of 75 elements. Hence, we need 76 to be guaranteed to find a non-zero linearly dependent subset of smooth pairs. For this example we ended the sieve phase after finding 78 pairs which gives us a few different subsets to try, in case some of them result in a trivial factorization.

Let us take a closer look at the pair $(19, 2)$ and derive the exponent vector. The pair $(19, 2)$ results in the rational integer

$$19 - 2 \cdot 164 = -309$$

and the algebraic integer

$$19 - 2\alpha.$$

The factorization of the rational integer into units and primes is

$$-309 = (-1) \cdot 3 \cdot 103,$$

and the factorization of the ideal $\langle 19 - 2\alpha\rangle$ into prime ideals is

$$\langle 19 - 2\alpha\rangle = \langle 5, \alpha - 2\rangle^2 \langle 7, \alpha - 6\rangle \langle 113, \alpha - 66\rangle.$$

(-301, 1)	(-263, 1)	(-253, 1)	(-226, 1)	(-206, 1)	(-92, 1)	(-78, 1)
(-57, 1)	(-46, 1)	(-28, 1)	(-23, 1)	(-22, 1)	(-14, 1)	(-13, 1)
(-8, 1)	(-7, 1)	(-5, 1)	(-4, 1)	(-2, 1)	(-1, 1)	(2, 1)
(3, 1)	(4, 1)	(10, 1)	(17, 1)	(22, 1)	(27, 1)	(29, 1)
(30, 1)	(35, 1)	(47, 1)	(48, 1)	(66, 1)	(69, 1)	(83, 1)
(84, 1)	(115, 1)	(139, 1)	(147, 1)	(161, 1)	(212, 1)	(322, 1)
(325, 1)	(383, 1)	(650, 1)	(810, 1)	(-53, 2)	(-41, 2)	(-35, 2)
(-23, 2)	(-5, 2)	(1, 2)	(19, 2)	(63, 2)	(69, 2)	(93, 2)
(103, 2)	(119, 2)	(205, 2)	(355, 2)	(-727, 3)	(-542, 3)	(-364, 3)
(-28, 3)	(-23, 3)	(-14, 3)	(-8, 3)	(-4, 3)	(-1, 3)	(4, 3)
(7, 3)	(17, 3)	(41, 3)	(47, 3)	(85, 3)	(208, 3)	(541, 3)
(-439,4)						

Table 6.3: Smooth pairs a, b found in the sieve phase

We double-check the last factorization by taking the norms of the ideals, using Theorem 3.2.5 to let us take the norm of an ideal with one generator, as well as using Theorem 3.2.11 to deal with the ideals with two generators:

$$\begin{aligned} N(\langle 19-2\alpha\rangle) &= N(\langle 5,\alpha-2\rangle)^2 \cdot N(\langle 7,\alpha-6\rangle) \cdot N(\langle 113,\alpha-66\rangle) \\ 19775 &= 5^2 \cdot 7 \cdot 113 \end{aligned}$$

Lastly, we calculate the quadratic character $\left(\frac{19-2s}{q}\right)$ for each element (q, s) in the quadratic character factor base:

$$\left(\frac{19-2\cdot 1}{149}\right) = 1$$
$$\left(\frac{19-2\cdot 75}{151}\right) = 1$$
$$\left(\frac{19-2\cdot 91}{157}\right) = -1$$
$$\left(\frac{19-2\cdot 108}{173}\right) = 1$$
$$\left(\frac{19-2\cdot 6}{179}\right) = -1$$
$$\left(\frac{19-2\cdot 36}{193}\right) = -1$$

An exponent vector has 75 elements in this example. The elements have the following meanings:

- 1: The sign of $a - bm$
- 2-35: One element for each prime ideal in the algebraic factor base
- 36-69: One element for each rational prime in the rational factor base
- 70-75: One element for each quadratic character

(-92, 1)	(-57, 1)	(-23, 1)	(-8, 1)	(-7, 1)	(2, 1)
(10, 1)	(17, 1)	(29, 1)	(35, 1)	(84, 1)	(115, 1)
(139, 1)	(-5, 2)	(19, 2)	(69, 2)	(93, 2)	(119, 2)
(-542, 3)	(-28, 3)	(-23, 3)	(-8, 3)		

Table 6.4: Pairs (a, b) derived from the solution of $x^T A \equiv 0 \pmod 2$

Looking at the factorization of -309, we notice that the number is negative and the prime factors are 3 and 109, the second and 29th primes in the rational factor base, respectively. Hence, the elements 1 (the sign), 37 and 64 in the element vector will be set to 1, as all prime powers occur with a power of 1.

From the factorization of $\langle 19 - 2\alpha \rangle$ we see that the second prime in the factor base, $\langle 5, \alpha - 2 \rangle$ occurs with a power of 2, while the fourth and 28th primes ($\langle 7, \alpha - 6 \rangle$ and $\langle 113, \alpha - 66 \rangle$) occur once. Hence, element 3 in the exponent vector is 2 and elements 5 and 29 become 1.

We see from above that the third, fifth and sixth Legendre symbols are all -1, the elements in the exponent vector corresponding to these should be set to 1. Hence, elements 72, 74 and 75 are set to 1.

All other elements in the exponent vector are set to 0 as they represent prime factors not occurring in the factorizations, or they represent Legendre symbols equalling 1.

The resulting exponent vector is

$$\overbrace{1}^{\text{sign}}\ \overbrace{0201000000000000000000000001000000}^{\text{algebraic primes}}\ \overbrace{0100000000000000000000000000100000}^{\text{rational primes}}\ \overbrace{001011}^{\text{quad.char.}}\ .$$

The matrix consists of all the exponent vectors reduced modulo 2. In our implementation we store the matrix in transposed form, and Figure 6.1 shows the transposed matrix, which has size 75×78. Column i contains the exponent vector for the i-th smooth pair a, b. We notice that some of the rows have especially many 1's: Row 1, which corresponds to the sign of $a - bm$, the first few rows of each of the rational and algebraic factor bases (since small primes occur more often), and the last 6 rows containing quadratic characters (1 should appear with probability around 0.5).

The linear algebra step will transform the matrix to the reduced row echelon form, and the reduced matrix is shown in Figure 6.2. The solution vector is $\mathbf{x} = (x_1, x_2, \ldots, x_{78})$ (one element for each exponent vector), and $x_i = 1$ means that the i-th smooth pair is part of the product that forms a square. We obtain the solution by setting the free variables arbitrarily, and then by setting the rest of the variables using back-substitution. Since we started with a 75×78 matrix, we are already guaranteed 3 free variables. In addition, there are 9 null rows in the reduced matrix, so we have 12 free variables in total. We set the second free variable (x_{67}) to 1 and the remaining free variables to 0 and determine the rest of the solution vector using back-substitution. Table 6.4 shows the pairs (a, b) that ensures that we have rational and algebraic squares. Let $\mathcal{S}$ be the set of these (a, b) pairs.

Now we can take the rational square root. Our rational square is expressed as

$$u^2 = f'(m)^2 \prod_{(a,b) \in \mathcal{S}} (a - bm).$$

1111111111111111111111111111111111111111100000011111111111110111111111111111101	Sign of $a - bm$
0110000000100001100001001011000001000101001000000000000000000000010001011010000	Algebraic factor base
0110000000100110000010001010000000000000010000001000000000000000100101000000000	
1000000001000001000000000000000000000001101000000100001000000000110100000000000	
0010011100010010000100000010000101100100000001000010010001001010000000100000000	
1000010000000000001000000000100000000101000000010000000001000010000100000000000100	
0000000000010000000000101000100001000010100000000001000000000000000100100000001	
0100110000000000000000100010000000000000000100100000000001000000001000000000011000	
0010000001010000000000000000000000010010010100000001000010000000000100000000000000	
0100000010000000000000000000001000000000000000000000010100000001000010000001000000	
0001000000000000000000000010000000000110000000000000000000000000000000100000001	
0000000000000000000000010000001000010000000000000000000000000000000000000000000	
0000000000000001000000000000000000001000000000100010000000000000001100000000000000001	
0101001000000000001000000000000000000000000000001000000000001000000000000000000000	
0000000000000100000000000000010000000000000000000000000000001000100000000010001 0001	
0000100000000000000000000001000000000000000010000000001000001010001000000000000100	
1000000100000000000000010000000000000000000000000010000000100000000000000000000000	
0000000000000000000000001000000000000001000101010000000000000000000000000000000000	
0000100000010000000000000000000000000000000000000000000000000000000100000000010	
0000000000000000000000100000000000000000000000001000000001000000000000000000000010	
0001000000000000100000000000000000010000100100000000000000000000100000000000000000000	
0000001000000000010000000000000000000000000000000000000010000010000000000000000000	
0100000000000100000000000000000000001000000000001000000000000000000000000000000100000	
0000000000000000001000000000000000000000000000000000000010000000000000000000000	
0000000000000000000000000100000000000000000011000000000000000000110001000000000000	
0000000000000000000000000000000000000001000000000000000000000001000000000100000001000	
0000000000000000010000000000000000000000100000000000000001000000000000000000000100	
0000000100000000000000000000000000000000010000000000010000000000000000000000000010000	
0000000000000000000000000000000010000000000000000000001000000000001000000000001000	
0000100000000000000000000000000100000000000000100000000000000000010000000001000000	
0000000000000000000000000000000000000000000000000010000000000000000000000000000001	
0001000000000000000000000000000010000000001000001000000000000000000000000010000000	
0000000010000000000000000000000000000000010000000000000001000000000000000000000	
0000000000100000000000000000000000001000000000000000000001000000000000000000000100	
0000000010000000000000000000000000000000000000000000000000000000000000000000000000	
0001101010011000011010110100100010000000001001100000000000000000111010001000000000	Rational factor base
1011000011010100010100001001010000000000110011010110110000011000000000000000000001	
1001100010000000000100100001000001010000000000000000000001010000000110100010010001	
0100000001000000000100010100000000000000000000000100000000000100000000000000000000000	
0000000000100000000100010000000000000000000000000000000000001000100010000000101000	
0001000100000000000000000000001000000000000000000001000000000000010000000000000000	
0000000010010000000000000000000000000001000000100000000000000000000000100000000	
0000000000000001000000000000000000010000000000010000000000010000000000000001000000	
0000000000000000000010000000000000000000010000000000000000001000010000000000000	
0000000000000000000000000000000100000000000000000000000000000000000001000000000	
1000000000100000000000000000000000000000000000000000000000000000000010000000000	
0000100000000000000000000000000000000000000000000000100010000000000000000000001000	
0000000000000000000000000000000000000000000000001000000000010000000000000100000	
0000000000000010000000000000000100000000000000000000000000000000000000000000000000	
0000000000000000000000000000000000000000000000000000100000100000000000000000000	
0000000000000000000000000000000000000000000000000001000000100000000000000000000	
0000000000000100000000000000000000000000000000000000000000000000000000000000000	
0100000000000000000000000000000000000000000000000000000000000000000000100000000	
0000000000000000000000000010000000000000000000000000000000000000000000000000000	
0000000000000000000000001000000000000000000000000000000000000000000000000000000100	
0000000000000000000000000000000000000000100000000000000000000000000000000000000001	
0000000000000000000000000000000000000000100000000000000000000000000000000000000	
0000000000000000000100000000000000000000000000000000000000000000000000000000000	
0000000000001000000000000000000000000000000000000000000000000000000000000000010000	
0000000000000000000000000000000000000000000000000000000000000000000000010000000	
0000000000000000000000000000000000000000000000000000000000000000000000000000000	
0000000000000000000000000000000000000000000000000001000000000001000000000000000	
0000000000000000000000000000000000000000000000000000000000010000000000000000000	
0000000000000000000000000000000000000000000000000100000000000000000000000000000	
0000000000000000000000000000000000000000000000000000000000000000000000000000000	
0000000000000000000000000000000000000000001000000000000000000000000000000000000	
0000000000000000000000000000000000000000000000000000000000000000000000000000000	
0000000000000000000000000010000000000000000000000000000000000000000000000000000	
0010000000000000000000000000000000000000000000000000000000000000000000000000000	
0111111100011101001101100100010010010111110100110000000111011100001000011101110	Quadratic characters
1100011011110100100101001000110010000000010111011110001011010010110100000000110	
1000010011100101110101101101100101111001011111111001110011100000100001000101 01	
1001110110100001011001111101001011001011101100000111011101100000001010101111010	
0010000010101111000011001011101001011100011000101010100010011001000111011001 10	
0100001010010010111111011000001011101111100110110011100001110101011001010100 10	

Figure 6.1: The transposed matrix containing all exponent vectors as columns

```
100000000000000000000000000000000000000000000000000000000001000000011100011001
010000000000000000000000000000000000000000000000000000000000000000001000000000
001000000000000000000000000000000000000000000000000000000000000000000000000000
000100000000000000000000000000000000000000000000000000000001000000001000000100
000010000000000000000000000000000000000000000000000000000001000000001000111010
000001000000000000000000000000000000000000000000000000000000000000100100110101
000000100000000000000000000000000000000000000000000000000000000000011100010110
000000010000000000000000000000000000000000000000000000000001000000110001101110
000000001000000000000000000000000000000000000000000000000000000000000000000000
000000000100000000000000000000000000000000000000000000000001000000000000000001
000000000010000000000000000000000000000000000000000000000001000000101000111000
000000000001000000000000000000000000000000000000000000000001000000001100011001
000000000000100000000000000000000000000000000000000000000000000000000000010000
000000000000010000000000000000000000000000000000000000000000000000000000000000
000000000000001000000000000000000000000000000000000000000001000000101001111100
000000000000000100000000000000000000000000000000000000000000000000111000111111
000000000000000010000000000000000000000000000000000000000001000000011000110000
000000000000000001000000000000000000000000000000000000000000000000010001011111
000000000000000000100000000000000000000000000000000000000000000000000000000000
000000000000000000010000000000000000000000000000000000000000000000010000100110
000000000000000000001000000000000000000000000000000000000000000000100001011110
000000000000000000000100000000000000000000000000000000000001000000001101011001
000000000000000000000010000000000000000000000000000000000000000000011000001100
000000000000000000000001000000000000000000000000000000000000000000100001011100
000000000000000000000000100000000000000000000000000000000001000000110001110011
000000000000000000000000010000000000000000000000000000000000000000000000000100
000000000000000000000000001000000000000000000000000000000000000000000000000000
000000000000000000000000000100000000000000000000000000000001000000110101100101
000000000000000000000000000010000000000000000000000000000000000000000000000000
000000000000000000000000000001000000000000000000000000000001000000101001111100
000000000000000000000000000000100000000000000000000000000000000000000000011110
000000000000000000000000000000010000000000000000000000000000000000001000000000
000000000000000000000000000000001000000000000000000000000000000000000000100100
000000000000000000000000000000000100000000000000000000000000000000000101101111
000000000000000000000000000000000010000000000000000000000000000000011100000111
000000000000000000000000000000000001000000000000000000000000000000100001000010
000000000000000000000000000000000000100000000000000000000001000000101101000011
000000000000000000000000000000000000010000000000000000000000000000011000001000
000000000000000000000000000000000000001000000000000000000001000000010000101100
000000000000000000000000000000000000000100000000000000000000000000001100110111
000000000000000000000000000000000000000010000000000000000000000000000000110011
000000000000000000000000000000000000000001000000000000000000000000000000000000
000000000000000000000000000000000000000000100000000000000001000000001100100001
000000000000000000000000000000000000000000010000000000000000000000000000000001
000000000000000000000000000000000000000000001000000000000000000000000001011111
000000000000000000000000000000000000000000000100000000000001000000000001111010
000000000000000000000000000000000000000000000010000000000000000000000000000000
000000000000000000000000000000000000000000000001000000000001000000000000000100
000000000000000000000000000000000000000000000000100000000001000000000000110001
000000000000000000000000000000000000000000000000010000000000000000000000000001
000000000000000000000000000000000000000000000000001000000001000000111101001001
000000000000000000000000000000000000000000000000000100000000000000000000000000
000000000000000000000000000000000000000000000000000010000000000000111001011001
000000000000000000000000000000000000000000000000000001000000000000000100100110
000000000000000000000000000000000000000000000000000000100000000000110101111011
000000000000000000000000000000000000000000000000000000010000000000100001011110
000000000000000000000000000000000000000000000000000000001000000000000000110111
000000000000000000000000000000000000000000000000000000000101000000111101101010
000000000000000000000000000000000000000000000000000000000011000000000000100100
000000000000000000000000000000000000000000000000000000000000100000000100100110
000000000000000000000000000000000000000000000000000000000000010000100001011110
000000000000000000000000000000000000000000000000000000000000001000000000000000
000000000000000000000000000000000000000000000000000000000000000100111001110101
000000000000000000000000000000000000000000000000000000000000000010111001011001
000000000000000000000000000000000000000000000000000000000000000001000101011110
000000000000000000000000000000000000000000000000000000000000000000000010000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Figure 6.2: The matrix in reduced row echelon form

p	Square root of γ in $\mathbb{Z}_{p_i}/\langle f(x)\rangle$
2305843009213693951	$1681812579256330563\alpha^2 + 481917539782026790\alpha + 2053587909481111827$
2305843009213693967	$1681812579256330563\alpha^2 + 481917539782027030\alpha + 2053587909481112131$
2305843009213693973	$1681812579256330563\alpha^2 + 481917539782027120\alpha + 2053587909481112245$
2305843009213694381	$1681812579256330563\alpha^2 + 481917539782033240\alpha + 2053587909481119997$

Table 6.5: Square roots for each modulo p

From the known factorization of $a - bm$ for each $(a, b) \in \mathcal{S}$ we get

$$u^2 = 81478^2 \cdot 2^{22} \cdot 3^{14} \cdot 5^{10} \cdot 7^6 \cdot 11^4 \cdot 13^2 \cdot 17^2 \cdot 19^2 \cdot 37^2 \cdot 43^2 \cdot 47^2 \cdot 103^2.$$

Finding u is just a matter of halving each exponent:

$$u = 81478 \cdot 2^{11} \cdot 3^7 \cdot 5^5 \cdot 7^3 \cdot 11^2 \cdot 13 \cdot 17 \cdot 19 \cdot 37 \cdot 43 \cdot 47 \cdot 103$$
$$u = 1530734055289535882078092800000$$
$$u \equiv 3739412 \pmod{4486873}$$

Because of the obstructions mentioned in Section 4.4, we cannot use a method similar to the above even if we know the factorization of each ideal $\langle a - b\alpha\rangle$ into prime ideals. Instead, we run our implementation of Couveignes' algorithm which computes the square root directly.

Our algebraic square is expressed as

$$\gamma = f'(\alpha)^2 \prod_{(a,b)\in\mathcal{S}} (a - b\alpha).$$

Evaluating this in $\mathbb{Z}[\alpha]$ gives

$$\gamma = 88447792045738866941140181562395466286 3\alpha^2 + 185233142010457316153316446224448238014 83\alpha + 211241980498409503712100797930238920774 32$$

We will first calculate the square roots β_i of γ (with the coefficients reduced modulo p_i) in the finite field $\mathbb{Z}_{p_i}/\langle f(x)\rangle$ for multiple p_i and use the Chinese Remainder Theorem to obtain $\beta = \sqrt{\gamma}$. First, we need to ensure that $\prod_p$ is greater than the bound from (4.7). We evaluate the base-2 logarithm of the bound, which is 236.27. Hence we require $\lg \prod_p \geq 236.27$. We pick four 61-bit primes whose product is large enough.

$$q_1 = 2305843009213693951$$
$$q_2 = 2305843009213693967$$
$$q_3 = 2305843009213693973$$
$$q_4 = 2305843009213694381$$

The square roots of γ in each of the four finite fields are shown in Table 6.5

These are the "correct" square roots having $N(\beta_i) \equiv N(\sqrt{\gamma}) \pmod{p_i}$. $N(\sqrt{\gamma})$ is calculated from the known factorization into ideals of the product of all $\langle a - b\alpha\rangle$. We use the Chinese Remainder Theorem to obtain the square root in $\mathbb{Z}[\alpha]$:

$$\beta = 1681812579256330563\alpha^2 - 3410572759842338 2475\alpha - 417574292655790732 42$$

We apply the homomorphism given by (4.1) and get

$$\begin{aligned} v = \sigma(\beta) &= 16818125792563305 63 \cdot 164^2 - 341057275984233824 75 \cdot 164 \\ &\quad - 41757429265579073242 \text{ (modulo } 4486873) \\ &= 1941654 \end{aligned}$$

Finally we obtain a non-trivial factor

$$\gcd(n, u - v) = \gcd(4486873, 3739412 - 1941654) = 3761.$$

6.7 Example: $n = 1027465709$

We show in detail how to factor $n = 1027465709 = 1009 \cdot 1018301$. We choose a degree 3 polynomial using the base-m expansion. By taking $m = \lfloor n^{1/3} \rfloor$ we get $m = 1009$. This results in

$$f(x) = x^3 + 220x.$$

As $f(x)$ has no constant term, it is divisible by x and has 0 as a root. Hence, $f(x) = x(x^2 + 220)$ is a factorization of $f(x)$. Hence, the algorithm terminates early with the factorization

$$\begin{aligned} f(m) &= g(m)h(m) \\ f(1009) &= g(1009)h(1009) \\ 1024765709 &= 1009 \cdot (1009^2 + 220) \\ &= 1009 \cdot 1018301. \end{aligned}$$

Chapter 7

Experiments

In this chapter we use our implementation of the NFS to perform some experiments. All experiments are performed on a PC with an Intel i7-2600K CPU and 16 GB RAM running Windows 7 64-bit. The experiments involve changing important parameters in the algorithm and observing the effect this has on the sieving, as well as the total time the program needs in order to find a factor. A short discussion concludes each experiment.

Although our program is capable of factoring n with 50-60 digits (where n has two prime factors of similar sizes) within a few hours, we chose to perform the experiments with a smaller n to be able to perform many runs within a shorter time frame.

7.1 Changing the factor base size

In this experiment we factor the 35-digit integer $n = 78325683705012095897299536068804821$ using a degree 3 polynomial, sieve width $|a| \leq 500000$ and 15 quadratic characters. We change the bound B for the factor base and observe the effect this has on the amount of work done in the sieve phase and the total time needed to get a factor. For each different B our program was run once, and we recorded the number of smooth pairs $|\mathcal{T}|$ we needed to find, the largest b checked in the sieve (this is slightly higher than the total number of elements across all factor bases), the total number of sieve operations and the total time our program needed to find a factor. A *sieve operation* is defined as processing one array element in the sieve for one prime p. For a smooth number given by a, b, the array element for this a need to be processed once for each prime p that divides either $a - bm$ or the norm of $a - b\alpha$. Sieve operations for non-smooth numbers are naturally also counted, and operations on pairs a, b with $\gcd(a, b) > 1$ are also counted here since doing the sieve operation is cheaper than checking the gcd.

Table 7.1 shows the results from all runs. The table includes $B = 67337$ which is the bound recommended by our implementation if we don't specify a bound.

From the table we learn that the program is not at its fastest if we let the program choose the asymptotically optimal bound B. The shortest runtime we achieved was 165 seconds, which happened at both $B = 100000$ and $B = 110000$, slightly above the recommended bound $B = 67337$. We notice that the number of sieve operations decreases when B increases for the values we tested. However, increasing B also increases the size of the factor base, which in turn increases the size of the matrix used in the linear algebra step. Since the linear algebra step is a bottleneck in our implementation, this has a negative effect on the runtime of our program. This explains the increase in the runtime

Bound B	Number of smooth pairs $\|\mathcal{T}\|$ found	Largest value of b checked	Number of sieve operations (10^6)	Time (s)
30000	6530	1608	6673	1311
40000	8462	613	2579	502
50000	10310	345	1466	331
60000	12141	241	1032	296
67337	13429	196	844	194
70000	13901	182	785	183
80000	15741	147	638	178
90000	17498	127	554	171
100000	19301	112	490	165
110000	21003	102	448	165
120000	22686	94	415	362
130000	24370	88	389	230
140000	26102	83	368	229
150000	27748	79	352	276
175000	31891	72	322	237
200000	36032	67	302	369
225000	40194	63	285	387
250000	44256	61	277	398
300000	52149	58	265	548

Table 7.1: Results from running the NFS with different factor base bounds

when $B \geq 120000$ despite less sieve work. With a more efficient implementation of the linear algebra step and the square root step it is likely that the minimal runtime would be achieved for a significantly higher B.

We also notice some random-looking spikes in the runtimes, especially for $B = 120000$. The reason is that we don't necessarily find a non-trivial factor on the first linear combination of exponent vectors we try, and the square root procedure needs a couple of seconds per try. This particular run was unlucky, and many linear combinations had to be tested before a factor was found.

7.2 Changing the width of the line sieve

In this experiment we investigate the effect of changing the width of the line sieve. We use the same settings as in the previous experiment: $n = 78325683705012095897299536068804821$, a degree 3 polynomial, factor base bound $B = 67337$ and 15 quadratic characters. We try different sieve bounds M. For a given M, we sieve all a that satisfy $|a| \leq M$. For each different M we decided to test, we did a full run of our program and recoded the largest value of b checked in the sieve phase, as well as number of seconds the program needed in order to find a factor.

The results from our runs are shown in Table 7.2. The total number of sieve operations is not reported as it is perfectly proportional to the largest value of b checked. We notice that the maximal value of b (the "height" of our rectangular sieving region) decreases as we increase the range of allowed a values. An increase of M leads to faster runtimes up to

Sieve bound M (10^3)	Largest value of b checked	Time (s)
100	6436	1180
200	1409	494
400	313	256
500	196	194
1000	53	145
2000	18	118
5000	7	112
10000	4	119
20000	3	253

Table 7.2: Results from running the NFS with different sieve widths

a certain point. Increasing the width causes smooth pairs with higher absolute values of a to be used, which increases the potential size of the products of which we take square roots. This causes the algebraic square root algorithm to use a higher bound for the product of the moduli, which requires us to use more prime moduli in the Chinese Remainder Theorem portion. Hence, for large enough sieve widths, the square root algorithm becomes a bottleneck.

Chapter 8

Conclusion and future work

In this thesis we have studied the NFS algorithm and the mathemathics which was required in order to understand the algorithm. We dived deeply into algebraic number theory. In particular we studied the factorization of an ideal generated by an algebraic integer into prime ideals, and looked at how to calculate the norm of algebraic integers and ideals.

Based on these studies we took a thorough look at the NFS algorithm itself. We have described every aspect of the algorithm, and it should be possible for the readers of this thesis to implement the algorithm.

We implemented the algorithm and found it to be a rather large and complicated undertaking. We encountered practical difficulties that weren't mentioned in existing literature. These difficulties do not represent mathematical obstacles, but still they can still represent a challenge during implementation. Some of these problems include an efficient way of generating the algebraic factor base (which boils down to finding roots of a polynomial $f(x)$ modulo a prime p) and calculating the norm of a general algebraic integers $a_0 + a_1\alpha + \cdots + a_{d-1}\alpha^{d-1}$ (here, linear algebra came to the rescue). The most difficult part of the implementation was to take the square root of an algebraic integer (square). The difficulty of this step was somewhat expected, as this step is traditionally known to be the most difficult phase of the NFS algorithm.

The sieve phase was quite interesting to implement and tweak. While the theory [cra05] gives asymptotically optimal values for the bounds of the factor bases and the sieve widths, in practice many of these values can be tuned for better performance. In addition, many of the possible implementation tricks have ways to be tweaked (such as the threshold for regarding a pair (a, b) as smooth, based on approximate logarithm calculations). We did not exhaust all the tweaking possibilities in our experiments, but a logical conclusion to the experiments is that we recommend to spend a significant amount of time to tune the implementation before embarking on a huge factorization task. After all, the sieve phase is the most time-consuming phase of the NFS under the assumption that all phases are implemented efficiently.

8.1 Future work

In this section we identify areas of improvement, both in our studies and in our implementation.

8.1.1 The theory

There are several ways to improve the NFS algorithm, and before implementing these the theory needs to be studied. These ways mostly involve doing an entire stage with a totally different algorithm. Earlier in the thesis we mentioned briefly the existence of a more efficient sieving algorithm (the *lattice sieve*), faster ways of doing the linear algebra step (*Block Lanczos* and *Block Wiedemann*) as well as methods for computing algebraic square roots that are not limited to number rings of odd degrees. See the respective sections in Chapter 4 for references to these methods.

There are other aspects of the theory we didn't look into in this thesis, such as the analysis of the asymptotic number of operations needed in order to factor n as well as deriving asympotically optimal parameter values.

8.1.2 The implementation

There are multiple ways to improve our implementation which we didn't explore in this thesis. In this section we list some suggested improvements.

Algorithmic improvements

We consider both the linear algebra and the algebraic square root phases to be major bottlenecks in our implementation that keep us from factoring integers much larger than 60 digits. We implemented Gaussian elimination which has a runtime of N^3 for a matrix of size $N \times N$. In addition, our implementation of Couveignes' algorithm for taking algebraic square roots is not as fast as it could be. First, we chose to use the easier-to-implement weak bound for the size of the coefficients of the square root instead of a better, but harder bound to implement. This requires us to use more moduli in the Chinese Remainder Theorem processing. In addition, we didn't utilize a particular implementation trick mentioned by Couveignes [cou93] that would result in slightly smaller numbers in intermediate calculations. All the shortcomings mentioned here can be addressed by changing to the more efficient algorithms mentioned in Section 8.1.1.

Large prime variation

The line sieve can be improved by allowing an additional large prime factor q for each of $a - bm$ and $N(a - b\alpha)$ where q can be larger than the factor base bound. In order to use these new pairs a, b we need to find a subset of pairs a, b such that the product of the rational integers and algebraic integers only have even powers of these large primes.

Parallelism

Several phases of the NFS algorithm can be parallelized. In the line sieve, we fix b and sieve along a for a given interval. The processing for each b is totally independent, and is "embarassingly parallel", which means that we can simply run different threads doing line sieve for different values of b.

Taking the square root of an algebraic integer is also a computationally intensive operation. As part of this algorithm we take the square root of an element in a finite field for each modulo. These intermediate square roots are computed independently, and each of them can therefore be done in parallel.

Bibliography

[atk13] *Sieve of Atkin - Wikipedia*, `http://en.wikipedia.org/wiki/Sieve_of_atkin`, retrieved on 2013-02-05.

[bha94] P. B. Bhattacharya, S. K. Jain, S. R. Nagpaul. *Basic abstract algebra.* Cambridge University Press, 1994.

[bri98] Matthew E. Briggs. *An introduction to the general number field sieve.* 1998.

[cas03] Michael Case. *A beginner's guide to the general number field sieve*, 2003.

[coh93] Henri Cohen. *A course in computational algebraic number theory*, 1993.

[cop93] Don Coppersmith. *Solving linear equations over GF(2): Block Lanczos algorithm*, 1993.

[cou93] Jean-Marc Couveignes. *Computing a square root for the number field sieve*, The development of the number field sieve, 1993.

[cra05] Richard Crandall, Carl Pomerance. *Prime numbers: a computational perspective.* Springer Verlag, 2005.

[era13] *Sieve of Eratosthenes - Wikipedia*, `http://en.wikipedia.org/wiki/Sieve_of_eratosthenes`, retrieved on 2013-02-05.

[knu98] Donald E. Knuth. *The art of computer programming volume 2 - Seminumerical algorithms*, third edition, 1998.

[len82] A. K. Lenstra, H. W. Lenstra, Jr, L. Lovács. *Factoring polynomials with rational coefficients*, 1982.

[len91] A. K. Lenstra, H. W. Lenstra, Jr, M. S. Manasse, J. M. Pollard. *The factorization of the ninth Fermat number*, 1991.

[len93] A. K. Lenstra, H. W. Lenstra, *The development of the number field sieve*, 1993.

[mar77] Daniel A. Marcus. *Number fields.* Springer Verlag, 1977.

[men01] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press, 2001.

[mur99] Brian Antony Murphy. *Polynomial selection for the number field sieve integer factorisation algorithm*, PhD thesis, 1999.

[ngu98] Phong Nguyen. *A Montgomery-like square root for the number field sieve*, 1998.

[sma03] Nigel Smart. Cryptography: an introduction, third edition, 2003.

[ste02] Ian Stewart, David Tall. *Algebraic number theory and Fermat's last theorem.* 2002.

[wie86] Douglas H. Wiedemann. *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory, vol IT-32 no 1, pp 54-62, 1986.

Appendices

Appendix A

Program listings

This appendix contains a listing (A.1) of our C code implementing the Number Field Sieve. There is only one code file, `nfs.c` which requires one library, GMP. The code follows the C89 standard with the exception of using the `long long` data types (which means that `gcc` can compile the code in C89 mode). Upon running the program, it will read multiple input lines from `stdin` (the number n to factor, factor base bounds, sieve width and so on). See Listing A.2 for an example input file, which is the one used in Section 6.6. This input file example is well-documented, and should explain the input format sufficiently. Blank lines are ignored, and lines beginning with a semicolon are treated as comments (and are ignored).

Listing A.1: `nfs.c`

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <gmp.h>

typedef unsigned char uchar;
typedef unsigned long long ull;
typedef long long ll;
typedef unsigned int uint;

void error(char *s) {
    puts(s);
    exit(1);
}

/* base 2 logarithm */
double log2(double a) {
    const double z=1.44269504088896340736; /* 1/log(2) */
    return log(a)*z;
}

#define BIGDEG 10
#define MAXDEG 10
```

```c

/* input parameters */

mpz_t opt_n; /* number to factorize */
ull opt_Ba; /* bound for algebraic factor base */
ull opt_Br; /* bound for rational factor base */
int opt_Bq; /* number of quadratic characters */
int opt_deg; /* degree of polynomial (must be odd, >=3) */
mpz_t opt_m; /* m value for base-m algorithm */
ull opt_sievew; /* width of line sieve */
int opt_thr; /* threshold for accepting a number in the sieve */
int opt_skip; /* skip this amount of smallest primes in the sieve */
int opt_extra; /* number of extra relations wanted for linear algebra */
int opt_signb; /* -1: a-bm, 1: a+bm */

void getnextline(char *s) {
    int l;
loop:
    if(!fgets(s,1048570,stdin)) { s[0]=0; return; }
    if(s[0]=='\n' || s[0]=='\r') goto loop;
    if(s[0]==';' || s[0]=='%' || s[0]=='#') goto loop;
    l=strlen(s);
    while(l && (s[l]=='\n' || s[l]=='\r')) s[l--]=0;
}

gmp_randstate_t gmpseed;

/* get a d-digit random number */
void getmpzrandom(mpz_t r,int d) {
    static char t[1024];
    mpz_t a,b;
    int i;
    if(d>1022) error("too many digits");
    mpz_init(a); mpz_init(b);
    t[i]='1';
    for(i=1;i<=d;i++) t[i]='0';
    t[i]=0;
    mpz_set_str(a,t,10);
    t[i-1]=0;
    mpz_set_str(b,t,10);
    do mpz_urandomm(r,gmpseed,a); while(mpz_cmp(r,b)<0);
    mpz_clear(b); mpz_clear(a);
}

/* return a mod p where a is mpz and p is int */
int mpz_mod_int(mpz_t a,int p) {
    mpz_t b;
    int r;
    mpz_init(b);
    r=mpz_mod_ui(b,a,p);
```

```
    mpz_clear(b);
    return r;
}

void readoptions() {
    static char s[1048576],t[4096];
    int z,i;
    /* read n */
    mpz_init(opt_n);
    mpz_init(opt_m);
    getnextline(s);
    sscanf(s,"%4090s",t);
    if(t[0]=='c') {
        /* generate a z-digit composite number without small prime factors */
        z=strtol(t+1,NULL,10);
        do {
            getmpzrandom(opt_n,z);
            if(!mpz_mod_int(opt_n,2)) continue;
            if(mpz_probab_prime_p(opt_n,25)) continue;
            if(z>10) for(i=3;i<20000;i+=2) if(!mpz_mod_int(opt_n,i)) continue;
        } while(0);
    } else if(t[0]=='r') {
        /* generate RSA number: a z-digit number that is the product
           of two similarly sized primes */
        z=strtol(t+1,NULL,10);
        /* TODO, pick two primes of z/2 digits and multiply */
        error("not implemented yet");
    } else {
        /* take literal number */
        mpz_set_str(opt_n,t,10);
    }
    getnextline(s); sscanf(s,"%I64d",&opt_Ba);
    getnextline(s); sscanf(s,"%I64d",&opt_Br);
    getnextline(s); sscanf(s,"%d",&opt_Bq);
    getnextline(s); sscanf(s,"%d",&opt_deg);
    getnextline(s); mpz_set_str(opt_m,s,10);
    getnextline(s); sscanf(s,"%I64d",&opt_sievew);
    getnextline(s); sscanf(s,"%d",&opt_thr);
    /* TODO support percentage for skip (that is, skip x percent of the
       primes) */
    getnextline(s); sscanf(s,"%d",&opt_skip);
    getnextline(s); sscanf(s,"%d",&opt_extra);
    if(opt_deg>MAXDEG) error("too high degree");
    if(!(opt_deg&1)) error("degree must be odd");
    getnextline(s); sscanf(s,"%d",&opt_signb);
    if(opt_signb!=1 && opt_signb!=-1) error("wrong sign");
}

/* all polynomials have the following format:
   coefficients in f[i], f[0]=a_0, f[1]=a_1, ..., f[i]=a_i
```

```
   size of f[] is MAXDEG+1 */

/* auxilliary routines */

/* need these since gmp doesn't support long long */
ull mpz_get_ull(mpz_t a) {
    static char s[1048576];
    ull ret;
    mpz_get_str(s,10,a);
    sscanf(s,"%I64d",&ret);
    return ret;
}

void mpz_set_ull(mpz_t b,ull a) {
  mpz_import(b, 1, 1, sizeof(a), 0, 0, &a);
}

/* return a mod p where a is mpz */
ull mpz_mod_ull(mpz_t a,ull p) {
    ull r;
    mpz_t b;
    mpz_init(b);
    mpz_set_ull(b,p);
    mpz_mod(b,a,b);
    r=mpz_get_ull(b);
    mpz_clear(b);
    return r;
}

ull gcd(ull a,ull b) {
    return b?gcd(b,a%b):a;
}

/* nfs init stage: create polynomials, determine bounds, create factor base */
/* includes many subroutines for polynomials */

/* calculate asymptotically optimal d (degree of polynomial)
   warning, doesn't work for n with more than 307 digits or so
   (n must fit in double) */
int findhighestdegree(mpz_t n) {
    double N=mpz_get_d(n);
    return pow(3*log(N)/log(log(N)),1./3);
}

/* calculate upper bound for factor base
   warning, doesn't work for n with more than 307 digits or so
   (n must fit in double) */
ull findB(mpz_t n) {
    double N=mpz_get_d(n),z=1./3;
    return exp(pow(8./9,z)*pow(log(N),z)*pow(log(log(N)),z+z));
```

```
}

/* calculate number of quadratic characters to obtain
   warning, doesn't work for n with more than 307 digits or so
   (n must fit in double) */
ull findK(mpz_t n) {
    double N=mpz_get_d(n);
    return 3*log(N)/log(2.728182818);
}

/* given n, m and d, return polynomial of degree d which is the
   base-m expansion of n. return 0 if something went wrong (degree doesn't
   match expansion, polynomial isn't monic etc) */
/* assume that *f is allocated with (d+1) uninitialized elements.
   f[0]=a_0, f[1]=a_1, ..., f[d]=1,
   polynomial is f(x)=a_d x^d + ... + a_1 x + a_0 */
int getpolynomial(mpz_t n,mpz_t m,int d,mpz_t *f) {
    mpz_t N;
    int i,r=1;
    mpz_init_set(N,n);
    for(i=0;i<=d;i++) {
        mpz_init(f[i]);
        mpz_fdiv_qr(N,f[i],N,m);
    }
    /* error if base-m expansion of n requires degree!=d or f isn't monic */
    if(mpz_cmp_si(N,0) || mpz_cmp_si(f[d],1)) r=0;
    mpz_clear(N);
    return r;
}

void printmpzpoly(mpz_t *f,int d) {
    for(;d>1;d--) gmp_printf("%Zd x^%d + ",f[d],d);
    gmp_printf("%Zd x + %Zd\n",f[1],f[0]);
}

void printullpoly(ull *f,int d) {
    printf("(%d) ",d);
    for(;d>-1;d--) printf("%I64u ",f[d]);
}

/* calculate the norm of a-b*alpha without using division.
   needs the minimal polynomial f (must be monic!) and its
   degree d (f[] has d+1 elements, where f[0]=a_0, f[i]=a_i and f[d]=1).
   put the answer in r */
void calcnorm(mpz_t r,mpz_t a,mpz_t b,mpz_t *f,int d) {
    static mpz_t x[MAXDEG+1];
    mpz_t y,temp;
    int i;
    mpz_init(temp);
    for(i=0;i<=d;i++) mpz_init(x[i]);
```

```
    mpz_set_si(x[0],1);
    mpz_set(x[1],a);
    mpz_init_set_si(y,1);
    mpz_set_si(r,0);
    for(i=2;i<=d;i++) mpz_mul(x[i],x[i-1],a);
    for(i=d;i>=0;i--) {
        mpz_mul(temp,y,x[i]);
        mpz_addmul(r,temp,f[i]);
        mpz_mul(y,y,b);
    }
    mpz_clear(temp);
    mpz_clear(y);
    for(i=0;i<=d;i++) mpz_clear(x[i]);
}

/* factor base routines */
uchar *sieve;
#define SETBIT(p) sieve[(p)>>3]|=1<<((p)&7);
#define CLEARBIT(p) sieve[(p)>>3]&=~(1<<((p)&7));
#define CHECKBIT(p) (sieve[(p)>>3]&(1<<((p)&7)))

/* allocate and generate bit-packed sieve up to (not including) N */
void createsieve(ull N) {
    ull i,j;
    sieve=malloc((N+7)>>3);
    memset(sieve,0xaa,(N+7)>>3);
    sieve[0]=172;
    for(i=2;i*i<N;i++) if(CHECKBIT(i)) for(j=i*i;j<N;j+=i) CLEARBIT(j);
}

/* algebraic factor base */
ull *p1,*r1;
ull bn1;
/* rational factor base */
ull *p2;
ull bn2;
/* quadratic characters */
ull *p3,*r3;
ull bn3;

/* evaluate f(x), assume f monic */
void evalpoly(mpz_t *f,int deg,mpz_t x,mpz_t ret) {
    int i;
    mpz_set(ret,f[deg]);
    for(i=deg-1;i>=0;i--) {
        mpz_mul(ret,ret,x);
        mpz_add(ret,ret,f[i]);
    }
}
```

```
/* warning, requires 64-bit compiler, i think */
typedef __uint128_t ulll;
ull ullmulmod2(ull a,ull b,ull mod) { return (ulll)a*b%mod; }

/* evaluate f(x)%p, assume f monic. requires p<2^63 */
ull evalpolymod(ull *f,int df,ull x,ull p) {
	ull r=f[df];
	int i;
	for(i=df-1;i>=0;i--) r=(ullmulmod2(r,x,p)+f[i])%p;
	return r;
}

/* start of routine that finds all roots (aka linear factors) of a polynomial
   modulo a prime */
/* begins with various routines for doing polynomial arithmetic over Z_p */
/* in general, all routines that do stuff modulo m should be fed numbers in
   0, 1, ..., m-1 */

/* calculate inverse of a mod m (m can be composite, but 0 will be
   returned if an inverse doesn't exist). warning, don't use if m>=2^63 */
ll inverse(ll a,ll m) {
	ll b=m,x=0,y=1,t,q,lastx=1,lasty=0;
	while(b) {
		q=a/b;
		t=a,a=b,b=t%b;
		t=x,x=lastx-q*x,lastx=t;
		t=y,y=lasty-q*y,lasty=t;
	}
	return a==1?(lastx%m+m)%m:0;
}

/* modular square root! */

/* calculate the jacobi symbol, returns 0, 1 or -1 */
/* 1: a is quadratic residue mod m, -1: a is not, 0: a mod m=0 */
/* based on algorithm 2.3.5 in "prime numbers" (crandall, pomerance) */
/* WARNING, m must be an odd positive number */
int jacobi(ll a,ll m) {
	int t=1;
	ll z;
	a%=m;
	while(a) {
		while(!(a&1)) {
			a>>=1;
			if((m&7)==3 || (m&7)==5) t=-t;
		}
		z=a,a=m,m=z;
		if((a&3)==3 && (m&3)==3) t=-t;
		a%=m;
	}
```

```
	if(m==1) return t;
	return 0;
}

ull ullpowmod(ull n,ull k,ull mod) {
	int i,j;
	ull v=n,ans=1;
	if(!k) return 1;
	/* find topmost set bit */
	for(i=63;!(k&(1ULL<<i));i--);
	for(j=0;j<=i;j++) {
		if(k&(1ULL<<j)) ans=ullmulmod2(ans,v,mod);
		v=ullmulmod2(v,v,mod);
	}
	return ans;
}

/* calculate legendre symbol, returns 0, 1 or -1 */
/* 1: a is quadratic residue mod p, -1: a is not, 0: a mod p=0 */
/* WARNING, p must be an odd prime */
int legendre(ll a,ll p) {
	a%=p;
	if(a<0) a+=p;
	int z=ullpowmod(a,(p-1)>>1,p);
	return z==p-1?-1:z;
}

ull rand64() {
	return (rand()&32767) +
			((rand()&32767)<<15) +
			((rand()&32767ULL)<<30) +
			((rand()&32767ULL)<<45) +
			((rand()&15ULL)<<60);
}

/* find square root of a modulo p (p prime) using tonelli-shanks */
/* runtime O(ln^4 p) */
/* mod 3,5,7: algorithm 2.3.8 from "prime numbers" (crandall, pomerance) */
/* mod 1: from http://www.mast.queensu.ca/~math418/m418oh/m418oh11.pdf */
ull sqrtmod(ull a,ull p) {
	int p8,alpha,i;
	ull x,c,s,n,b,J,r2a,r;
	if(p==2) return a&1;
	a%=p;
	if(legendre(a,p)!=1) return 0; /* no square root */
	p8=p&7;
	if(p8==3 || p8==5 || p8==7) {
		if((p8&3)==3) return ullpowmod(a,(p+1)/4,p);
		x=ullpowmod(a,(p+3)/8,p);
		c=ullmulmod2(x,x,p);
```

```
        return c==a?x:ullmulmod2(x,ullpowmod(2,(p-1)/4,p),p);
    }
    alpha=0;
    s=p-1;
    while(!(s&1)) s>>=1,alpha++;
    r=ullpowmod(a,(s+1)/2,p);
    r2a=ullmulmod2(r,ullpowmod(a,(s+1)/2-1,p),p);
    do n=rand64()%(p-2)+2; while(legendre(n,p)!=-1);
    b=ullpowmod(n,s,p);
    J=0;
    for(i=0;i<alpha-1;i++) {
        c=ullpowmod(b,2*J,p);
        c=ullmulmod2(r2a,c,p);
        c=ullpowmod(c,1ULL<<(alpha-i-2),p);
        if(c==p-1) J+=1ULL<<i;
    }
    return ullmulmod2(r,ullpowmod(b,J,p),p);
}

/* set b(x)=a(x) */
void polyset(ull *a,int da,ull *b,int *db) {
    int i;
    for(*db=da,i=0;i<=*db;i++) b[i]=a[i];
}

/* set c(x)=a(x)+b(x) */
void polyaddmod(ull *a,int da,ull *b,int db,ull *c,int *dc,ull p) {
    static ull r[MAXDEG+1];
    int i,dr;
    dr=da>db?da:db;
    for(i=da+1;i<=dr;i++) r[i]=0;
    for(i=0;i<=da;i++) r[i]=a[i];
    for(i=0;i<=db;i++) {
        r[i]+=b[i];
        if(r[i]>=p) r[i]-=p;
    }
    while(dr>-1 && !r[dr]) dr--;
    *dc=dr;
    for(i=0;i<=dr;i++) c[i]=r[i];
}

/* negates a (modifies a) */
void polynegmod(ull *a,int da,ull p) {
    for(;da>-1;da--) a[da]=((ll)p-(ll)a[da])%p;
}

/* given polynomials a(x) and b(x), calculate quotient and
   remainder of a(x)/b(x) (mod p)
   dega, degb are the degrees of a and b, respectively. assume that *c, *d
   has enough pre-allocated memory to hold the results. don't assume that
```

```
   any of a,b,c,d are non-overlapping memory areas.
   if c is non-NULL, return quotient.
   if d is non-NULL, return remainder. remainder==0 has degree -1.
*/
void polydivmod(ull *a,int dega,ull *b,int degb,ull *c,int *degc,ull *d,int *degd,ull p) {
    static ull u[MAXDEG+1],q[MAXDEG+1];
    ull inv=inverse(b[degb],p);
    int k,j;
    for(k=0;k<=dega;k++) u[k]=a[k];
    for(k=dega-degb;k>-1;k--) {
        q[k]=ullmulmod2(u[degb+k],inv,p);
        for(j=degb+k-1;j>=k;j--) {
            u[j]=u[j]-ullmulmod2(q[k],b[j-k],p);
            if(u[j]>=p) u[j]+=p;
        }
    }
    if(c) for(*degc=dega-degb,k=*degc;k>-1;k--) c[k]=q[k];
    if(d) for(*degd=-1,k=0;k<degb && k<=dega;k++) if((d[k]=u[k])) *degd=k;
}

/* make polynomial monic, destroy input polynomial */
void polymonic(ull *a,int da,ull p) {
    ull z;
    int i;
    if(da<0 || a[da]==1) return;
    z=inverse(a[da],p);
    for(i=0;i<da;i++) a[i]=ullmulmod2(a[i],z,p);
    a[da]=1;
}

/* return a(x)*b(x) over Z_p */
void polymulmod(ull *a,int dega,ull *b,int degb,ull *c,int *degc,ull p) {
    static ull r[2*MAXDEG+1];
    int i,j;
    *degc=dega+degb;
    for(i=0;i<=*degc;i++) r[i]=0;
    for(i=0;i<=dega;i++) {
        for(j=0;j<=degb;j++) {
            r[i+j]=r[i+j]+ullmulmod2(a[i],b[j],p);
            if(r[i+j]>=p) r[i+j]-=p;
        }
    }
    for(i=0;i<=*degc;i++) c[i]=r[i];
    while(*degc>-1 && !c[*degc]) (*degc)--;
}

/* reduce a(x) mod v(x) over Z_p */
/* runtime: O(degree^2) */
void polyreduce(ull *a,int da,ull *v,int dv,ull *c,int *dc,ull p) {
    static ull w[2*MAXDEG+1];
```

```
    ull t;
    int i,j,z;
    for(i=0;i<=da;i++) w[i]=a[i];
    for(i=da+1;i<=dv;i++) w[i]=0;
    /* for each i=da, da-1, ..., dv, subtract a(i)*v(x)*x^(i-dv) */
    for(i=da;i>=dv;i--) for(j=0;j<=dv;j++) {
        z=i-dv; t=w[i];
        w[z+j]=(w[z+j]+p-ullmulmod2(t,v[j],p))%p;
    }
    /* tighten dc */
    for(*dc=-1,i=0;i<dv;i++) if((c[i]=w[i])) *dc=i;
}

/* given f, return g=f' (mod p) */
void polyderivemod(ull *f,int df,ull *g,int *dg,ull p) {
    int i;
    *dg=df-1;
    for(i=1;i<=df;i++) g[i-1]=ullmulmod2(f[i],i,p);
    while(*dg>-1 && !g[*dg]) (*dg)--;
}

/* return a(x)*b(x) mod v(x) over Z_p */
/* this can probably also be used to multiply two elements
   in the quotient ring Z_p/<v(x)> */
/* WARNING, not efficient. integrate mulmod and reduce more tightly */
void polymulmodmod(ull *a,int da,ull *b,int db,ull *v,int dv,ull *c,int *dc,ull p) {
    static ull d[2*MAXDEG+1];
    int dd;
    polymulmod(a,da,b,db,d,&dd,p);
    polyreduce(d,dd,v,dv,c,dc,p);
}

/* return a(x)^n mod v(x) over Z_p, put result in c */
/* warning, not very efficient, really, but care about that later */
void polypowmodmod(ull *a,int da,ull n,ull *v,int dv,ull *c,int *dc,ull p) {
    ull z[MAXDEG+1],y[MAXDEG+1]={1};
    int dz,dy=0,i;
    polyset(a,da,z,&dz);
    while(n) {
        if(n&1) {
            n>>=1;
            polymulmodmod(y,dy,z,dz,v,dv,y,&dy,p);
            if(!n) break;
        } else n>>=1;
        polymulmodmod(z,dz,z,dz,v,dv,z,&dz,p);
    }
    for(*dc=dy,i=0;i<=*dc;i++) c[i]=y[i];
}

/* return a(x)^n mod v(x) over Z_p, put result in c, exponent is mpz */
```

```
/* warning, not very efficient, really, but care about that later */
void polypowmodmodmpz(ull *a,int da,mpz_t N,ull *v,int dv,ull *c,int *dc,ull p) {
    ull z[MAXDEG+1],y[MAXDEG+1]={1};
    int dz=da,dy=0,i;
    mpz_t t,n;
    mpz_init(t);
    mpz_init_set(n,N);
    for(i=0;i<=dz;i++) z[i]=a[i];
    while(mpz_cmp_si(n,0)>0) {
        if(mpz_mod_ui(t,n,2)) {
            mpz_fdiv_q_2exp(n,n,1);
            polymulmodmod(y,dy,z,dz,v,dv,y,&dy,p);
            if(!mpz_cmp_si(n,0)) break;
        } else mpz_fdiv_q_2exp(n,n,1);
        polymulmodmod(z,dz,z,dz,v,dv,z,&dz,p);
    }
    for(*dc=dy,i=0;i<=*dc;i++) c[i]=y[i];
    mpz_clear(n);
    mpz_clear(t);
}

/* return a(x)^n over Z_p */
void polypowmod(ull *a,int da,ull n,ull *c,int *dc,ull p) {
    ull z[MAXDEG+1],y[MAXDEG+1]={1};
    int dz=da,dy=0,i;
    for(i=0;i<=dz;i++) z[i]=a[i];
    while(n) {
        if(n&1) {
            n>>=1;
            polymulmod(y,dy,z,dz,y,&dy,p);
            if(!n) break;
        } else n>>=1;
        polymulmod(z,dz,z,dz,z,&dz,p);
    }
    for(*dc=dy,i=0;i<=*dc;i++) c[i]=y[i];
}

/* given polynomials a(x), b(x), calculate g(x)=gcd(a(x),b(x)) mod p. */
void polygcdmod(ull *a,int da,ull *b,int db,ull *g,int *dg,ull p) {
    static ull c[MAXDEG+1],d[MAXDEG+1],e[MAXDEG+1];
    int dc,dd,de,i;
    polyset(a,da,c,&dc);
    polyset(b,db,d,&dd);
    /* sanity check: a==0 */
    if(da<0) {
        for(*dg=dd,i=0;i<=*dg;i++) g[i]=d[i];
        goto end;
    }
    while(dd>-1) {
        polydivmod(c,dc,d,dd,NULL,NULL,e,&de,p);
```

```
			polyset(d,dd,c,&dc);
			polyset(e,de,d,&dd);
		}
		polyset(c,dc,g,dg);
end:
		/* make output monic */
		polymonic(g,*dg,p);
}

/* calculate inverse of a mod m (m can be composite, but 0 will be
   returned if an inverse doesn't exist). warning, don't use if m>=2^63 */
ll inversemal(ll a,ll m) {
		ll b=m,x=0,y=1,t,q,lastx=1,lasty=0;
		while(b) {
			q=a/b;
			t=a,a=b,b=t%b;
			t=x,x=lastx-q*x,lastx=t;
			t=y,y=lasty-q*y,lasty=t;
		}
		return a==1?(lastx%m+m)%m:0;
}

/* find the inverse g(x)=a^1(x) of a(x) mod f(x) mod p using
   the extended euclid algorithm */
/* f(x) is assumed to be monic. if an inverse doesn't exist, return g=0 */
void polyinversemodmod(ull *in,int din,ull *f,int df,ull *g,int *dg,ull p) {
		ull b[MAXDEG+1],x[MAXDEG+1],y[MAXDEG+1],lastx[MAXDEG+1],lasty[MAXDEG+1];
		ull t[MAXDEG+1],q[MAXDEG+1],a[MAXDEG+1],z[MAXDEG+1],v;
		int db,dx,dy,lastdx,lastdy,dt,dq,da,dz,i;
		if(din<0) { *dg=-1; return; }
		polyset(f,df,b,&db);
		polyset(in,din,a,&da);
		dx=-1; y[0]=1; dy=0;
		lastx[0]=1; lastdx=0; lastdy=-1;
		while(db>-1) {
			/* set a=b, b=a%b, q=a/b */
			polyset(a,da,t,&dt);
			polyset(b,db,a,&da);
			polydivmod(t,dt,a,da,q,&dq,b,&db,p);
			/* set x=lastx-q*x, lastx=x */
			polyset(x,dx,t,&dt);
			polymulmod(q,dq,x,dx,z,&dz,p);
			polyset(lastx,lastdx,x,&dx);
			polynegmod(z,dz,p);
			polyaddmod(x,dx,z,dz,x,&dx,p);
			polyset(t,dt,lastx,&lastdx);
			/* set y=lasty-q*y, lasty=y */
			polyset(y,dy,t,&dt);
			polymulmod(q,dq,y,dy,z,&dz,p);
			polyset(lasty,lastdy,y,&dy);
```

```
            polynegmod(z,dz,p);
            polyaddmod(y,dy,z,dz,y,&dy,p);
            polyset(t,dt,lasty,&lastdy);
        }
        /* now a is gcd(a,f). if !=1 return failure */
        if(da>0) { *dg=-1; return; }
        /* lastx is inverse, multiply with inverse of a[0] */
        if(a[0]!=1) {
            v=inverse(a[0],p);
            for(i=0;i<=lastdx;i++) lastx[i]=ullmulmod2(lastx[i],v,p);
        }
        for(*dg=lastdx,i=0;i<=lastdx;i++) g[i]=lastx[i];
}

/* return 1 if u(x) is squarefree. u is squarefree iff gcd(u,u')==1.
   u(x) must be monic. unpredictable results if deg u <= p */
int ispolymodsquarefree(ull *u,int du,ull p) {
        static ull ud[MAXDEG+1],g[MAXDEG+1];
        int dud,dg,i;
        for(dud=du-1,i=0;i<du;i++) ud[i]=ullmulmod2(u[i+1],i+1,p);
        while(dud>-1 && !ud[dud]) dud--;
        polygcdmod(u,du,ud,dud,g,&dg,p);
        return dg==0;
}

/* find all roots by naive method (evaluate in(x) for all x), inefficient */
void polylinmodnaive(ull *in,int dv,ull p,ull *f,int *fn) {
        ll x;
        for(*fn=x=0;x<p;x++) if(!evalpolymod(in,dv,x,p)) f[(*fn)++]=x;
}

/* find roots of u(x) mod p, p must be an odd prime larger than
   the degree of u(x) */
/* based on algorithm 1.6.1 in cohen */
void polyfindrootmod(ull *z,int dz,ull p,ull *f,int *fn) {
        /* cast out gcd(f',f) */
        static ull g[MAXDEG+1],ud[MAXDEG+1],u[MAXDEG+1],m1[MAXDEG+1];
        static ull q[MAXDEG+1][MAXDEG+1];
        ull d,e;
        int du,dg,dud,qn=1,done,i,dm1;
        static int dq[MAXDEG+1];
        *fn=0;
        polyset(z,dz,u,&du);
        /* force u monic */
        polymonic(u,du,p);
        polyderivemod(u,du,ud,&dud,p);
        polygcdmod(u,du,ud,dud,g,&dg,p);
        /* force gcd monic */
        polymonic(g,dg,p);
        /* divide out squares */
```

```
    polydivmod(u,du,g,dg,u,&du,NULL,NULL,p);
    /* cast out 0-factor */
    if(!u[0]) {
        g[0]=0; g[1]=1; dg=1;
        polydivmod(u,du,g,dg,u,&du,NULL,NULL,p);
        f[(*fn)++]=0;
    }
    /* m1(x)=-1 (p-1) */
    m1[0]=p-1; dm1=0;
    /* take gcd(x^(p-1)-1, u(x)) and isolate roots */
    /* first take d=x^(p-1) mod u, then take gcd(d-1,u) */
    g[0]=0; g[1]=1; dg=1;
    polypowmodmod(g,dg,p-1,u,du,g,&dg,p);
    polyaddmod(g,dg,m1,dm1,g,&dg,p);
    polygcdmod(g,dg,u,du,q[0],&dq[0],p);
    do {
        done=1;
        /* if deg>2, try to split polynomial. benchmarking shows it's faster
           to split down to deg 2 rather than deg 1. */
        for(i=0;i<qn;i++) if(dq[i]>2) {
            do {
                g[0]=rand64()%p; g[1]=1; dg=1;
                polypowmodmod(g,dg,p>>1,q[i],dq[i],g,&dg,p);
                polyaddmod(g,dg,m1,dm1,g,&dg,p);
                polygcdmod(g,dg,q[i],dq[i],g,&dg,p);
            } while(!dg || dg==dq[i]);
            polydivmod(q[i],dq[i],g,dg,q[i],&dq[i],NULL,NULL,p);
            polyset(g,dg,q[qn],&dq[qn]);
            qn++;
            done=0;
        }
    } while(!done);
    /* go through each item in the list, and output roots */
    for(i=0;i<qn;i++) {
        if(dq[i]==1) {
            if(q[i][1]==1) f[(*fn)++]=(p-q[i][0])%p;
            else f[(*fn)++]=ullmulmod2((p-q[i][0])%p,inverse(q[i][1],p),p);
        } else if(dq[i]==2) {
            d=ullmulmod2(q[i][1],q[i][1],p);
            e=ullmulmod2(q[i][0],q[i][2],p);
            e=sqrtmod((d+p-ullmulmod2(e,4,p))%p,p);
            d=ullmulmod2(inverse(2,p),q[i][2],p);
            f[(*fn)++]=ullmulmod2((p+e-q[i][1])%p,d,p);
            f[(*fn)++]=ullmulmod2((p+p-e-q[i][1])%p,d,p);
        }
    }
}

/* entry point for new routine */
void findideals2(ull *u,int du,ull p,ull *f,int *fn) {
```

```
    /* naive algorithm for small enough p: evaluate f(r) for all 0<=r<p */
    if(p<200 || p<=du) return polylinmodnaive(u,du,p,f,fn);
    polyfindrootmod(u,du,p,f,fn);
}

/* determinant using stupid and slow O(n!) algorith, but n will never
   be huge (say, never larger than 6 and in practice it will always be 3).
   generate permutations using fancy loop-free algorithm by knuth
   where successively generated permutations have alternating parity
   [an easy O(n^3) algorithm: gauss-jordan and return product of diagonal
   times the numbers we divided the rows with] */
ull calcdet(ull A[MAXDEG+1][MAXDEG+1],int n,ull p) {
    ull res=0,r;
    int o[100],c[100],j,s,q,a[100],sign=1;
    char t;
    for(j=0;j<n;j++) c[j]=0,o[j]=1,a[j]=j;
p2:
    /* visit permutation */
    r=sign?1:p-1;
    for(j=0;j<n;j++) r=ullmulmod2(r,A[j][a[j]],p);
    res+=r;
    if(res>=p) res-=p;
    sign^=1;
    /* end visit */
    j=n; s=0;
p4:
    q=c[j-1]+o[j-1];
    if(q<0) goto p7;
    if(q==j) goto p6;
    t=a[j-c[j-1]+s-1]; a[j-c[j-1]+s-1]=a[j-q+s-1]; a[j-q+s-1]=t;
    c[j-1]=q;
    goto p2;
p6:
    if(j==1) return res;
    s++;
p7:
    o[j-1]=-o[j-1]; j--;
    goto p4;
}

/* calculate norm mod p of general element a(x) in field with minimal
   polynomial f(x). uses determinant method */
/* tested against calcnorm() with tens of millions of numbers of the form
   a+b*alpha with degrees 3-6, with a,b huge modulo a huge prime */
ull calcnormmod(ull *a,int da,ull *f,int df,ull p) {
    static ull A[MAXDEG+1][MAXDEG+1];
    ull b[MAXDEG+1]={0,1},c[MAXDEG+1];
    int i,j,db=1,dc;
    polyset(a,da,c,&dc);
    for(i=0;i<=dc;i++) A[i][0]=a[i];
```

```
    for(;i<df;i++) A[i][0]=0;
    for(j=1;j<df;j++) {
        polymulmodmod(c,dc,b,db,f,df,c,&dc,p);
        for(i=0;i<=dc;i++) A[i][j]=c[i];
        for(;i<df;i++) A[i][j]=0;
    }
    return calcdet(A,df,p);
}

/* B1 and B2 are upper bound for primes (algebraic and rational)
   f is polynomial, deg is degree
   p1,r1 is algebraic factor base, bn1 is number of primes
   p2 is rational factor base, bn2 is number of primes */
void createfactorbases(ull B1,ull B2,ull Bk,mpz_t *f,int deg,ull **_p1,ull **_r1,ull *bn1,ull **_p2,ull *bn2,
                       ull **_p3,ull **_r3,ull *bn3) {
    static ull b[MAXDEG+1];
    static ull root[MAXDEG+1];
    ull B=B1>B2?B1:B2,i,j,q;
    ull *p1,*r1,*p2,*p3,*r3;
    int fn;
    int db,k;
    char *sieve=malloc(B+1);
    memset(sieve,1,B+1);
    for(i=2;i*i<=B;i++) if(sieve[i]) for(j=i*i;j<=B;j+=i) sieve[j]=0;
    /* generate rational factor base */
    for(*bn2=0,i=2;i<=B2;i++) if(sieve[i]) (*bn2)++;
    if(!(p2=malloc(*bn2*sizeof(ull)))) error("couldn't allocate rational factor base");
    for(*bn2=0,i=2;i<=B2;i++) if(sieve[i]) p2[(*bn2)++]=i;

    /* generate algebraic factor base */
    for(*bn1=0,i=2;i<=B1;i++) if(sieve[i]) {
        /* find all eligible r: r such that f(r)=0 (mod p) using factorization */
        db=deg;
        for(k=0;k<=db;k++) b[k]=mpz_mod_ull(f[k],i);
        findideals2(b,db,i,root,&fn);
        *bn1+=fn;
    }
    if(!(p1=malloc(*bn1*sizeof(ull)))) error("couldn't allocate algebraic factor base");
    if(!(r1=malloc(*bn1*sizeof(ull)))) error("couldn't allocate algebraic factor base");
    for(*bn1=0,i=2;i<=B1;i++) if(sieve[i]) {
        /* find all roots again. we happily waste some computing resources since
           the sieve stage will dominate the runtime anyway */
        /* slow method again TODO replace with factorization */
        db=deg;
        for(k=0;k<=db;k++) b[k]=mpz_mod_ull(f[k],i);
        findideals2(b,db,i,root,&fn);
        for(j=0;j<fn;j++) p1[*bn1]=i,r1[(*bn1)++]=root[j];
    }

    /* generate quadratic characters */
```

```
    *bn3=Bk;
    if(!(p3=malloc(*bn3*sizeof(ull)))) error("couldn't allocate quadratic characters");
    if(!(r3=malloc(*bn3*sizeof(ull)))) error("couldn't allocate quadratic characters");
    for(i=0,q=B1+1;i<*bn3;q++) {
        /* check if q is prime */
        for(j=0;j<*bn2 && p2[j]*p2[j]<=q;j++) if(q%p2[j]==0) goto noprime;
        db=deg;
        for(k=0;k<=db;k++) b[k]=mpz_mod_ull(f[k],q);
        findideals2(b,db,q,root,&fn);
        if(!fn) continue;
        /* find value from root such that f'(value)!=0 mod q */
        polyderivemod(b,db,b,&db,q);
        for(k=0;k<fn;k++) if(evalpolymod(b,db,root[k],q)) {
            p3[i]=q;
            r3[i]=root[k];
            i++;
            break;
        }
    noprime:;
    }

    free(sieve);
    *_p1=p1; *_r1=r1; *_p2=p2; *_p3=p3; *_r3=r3;
}

/* return index of v in p, or -1 if it doesn't exist */
ull bs(ull *p,ull bn,ull v) {
    ull lo=0,hi=bn,mid;
    while(lo<hi) {
        mid=(lo+hi)>>1;
        if(v>p[mid]) lo=mid+1;
        else hi=mid;
    }
    return lo<bn && p[lo]==v?lo:-1;
}

/* matrix (global) */
uint **M;
int notsmooth,missed,smooth;

/* gaussian elimination mod 2 on bitmasks, A is n*m, b is n*o */
/* a is a malloced array of pointers, each a[i] is of size
   sizeof(uint)*(m+o+31)/32 */
/* return 0: no solutions, 1: one solution, 2: free variables */
#define ISSET(a,row,col) (a[(row)][(col)>>5]&(1U<<((col)&31)))
#define MSETBIT(a,row,col) a[(row)][(col)>>5]|=(1U<<((col)&31))
#define MTOGGLEBIT(a,row,col) a[(row)][(col)>>5]^=(1U<<((col)&31))
int bitgauss32(uint **a,int n,int m,int o) {
    int i,j,k,z=m+o,c=0,fri=0,bz=(z+31)>>5;
    uint t;
```

```
	/* process each column */
	for(i=0;i<m;i++) {
		/* TODO check words instead of bits */
		for(j=c;j<n;j++) if(ISSET(a,j,i)) break;
		if(j==n) { fri=1; continue; }
		/* swap? */
		if(j>c) for(k=0;k<bz;k++) {
			t=a[j][k],a[j][k]=a[c][k],a[c][k]=t;
		}
		/* subtract multiples of this row */
		for(j=0;j<n;j++) if(j!=c && ISSET(a,j,i)) {
			for(k=0;k<bz;k++) a[j][k]^=a[c][k];
		}
		c++;
	}
	/* detect no solution: rows with 0=b */
	for(i=0;i<n;i++) {
		/* TODO make bit-efficient solution later */
		for(j=0;j<m;j++) if(ISSET(a,i,j)) goto ok;
		for(;j<z;j++) if(ISSET(a,i,j)) return 0;
	ok:;
	}
	return 1+fri;
}

/* find all free variables: variable i is free if there is no row having its first
   1-element in column i */
int findfreevars(uint **a,int rows,int cols,uchar *freevar) {
	int i,j,r=cols;
	memset(freevar,1,cols);
	for(i=0;i<rows;i++) {
		for(j=0;j<cols;j++) if(ISSET(a,i,j)) {
			freevar[j]=0;
			r--;
			break;
		}
	}
	return r;
}

/* find exponents of square. id is the index of the free variable we want to
   use
   rows: factor base
   cols: relations */
void getsquare(uint **a,int rows,int cols,uchar *freevar,int id,uchar *v) {
	int i,j,k;
	memset(v,0,cols);
	/* set id-th free variable */
	for(j=i=0;i<cols;i++) if(freevar[i]) {
		if(id==j) { v[i]=1; break; }
```

```
            j++;
        }
        /* get solution vector by back substitution! set the first 1-element to the
           xor of the others. */
        for(i=rows-1;i>=0;i--) {
            for(j=0;j<cols;j++) if(ISSET(a,i,j)) goto ok;
            continue;
        ok:
            for(k=j++;j<cols;j++) if(ISSET(a,i,j) && v[j]) v[k]^=1;
        }
}

/* store rational factors for pairs (a,b) */
ull **faclist;
int *facn;
/* store algebraic factors for pairs (a,b) */
ull **alglist;
int *algn;

/* get rational square root! */
void getratroot(mpz_t n,uchar *v,int cols,mpz_t *f,int df,mpz_t m,mpz_t root,int *aval,int *bval) {
        mpz_t t;
        static mpz_t fd[MAXDEG+1];
        static int *ev;
        int dfd;
        mpz_init(t);
        mpz_set_si(root,1);
        int i,j;
        ev=calloc(bn2,sizeof(int));
        if(!ev) error("out of memory");
        for(i=0;i<cols;i++) if(v[i]) for(j=0;j<facn[i];j++) ev[faclist[i][j]]++;
        /* sanity */
        for(i=0;i<bn2;i++) if(ev[i]&1) error("odd exponent in rat");
        for(i=0;i<bn2;i++) if(ev[i]) {
            mpz_set_ull(t,p2[i]);
            for(j=0;j+j<ev[i];j++) mpz_mul(root,root,t);
            mpz_mod(root,root,n);
        }
        /* multiply value with f'(m)^2 */
        dfd=df-1;
        for(i=0;i<=dfd;i++) {
            mpz_init_set(fd[i],f[i+1]);
            mpz_mul_ui(fd[i],fd[i],i+1);
        }
        evalpoly(fd,dfd,m,t);
        mpz_mod(t,t,n); /* t = f'(m) mod n */
        mpz_mul(root,root,t); /* multiply in f'(m) */
        mpz_mod(root,root,n); /* and reduce mod n */
        mpz_mul(t,root,root);
        mpz_mod(t,t,n);
```

```c
    gmp_printf("rational root: %Zd, square %Zd\n",root,t);
    for(i=0;i<=dfd;i++) mpz_clear(fd[i]);
    free(ev);
    mpz_clear(t);
}

/* start of routines for algebraic square root */

/* return 1 if f(x) is irreducible mod p */
int polyirredmod(mpz_t *in,int df,ull p) {
    /* check if gcd(x^(p^d)-x,f) is a non-constant
       polynomial for 1<=d<=df/2 */
    static ull g[MAXDEG+1],h[MAXDEG+1],f[MAXDEG+1];
    int dg,i,j,dh;
    for(i=0;i<=df;i++) f[i]=mpz_mod_ull(in[i],p);
    for(i=1;i+i<=df;i++) {
        /* form x^p^i - x */
        /* use that x^p^i = ((x^p)^p) ... ^p (i times) */
        g[0]=0; g[1]=1; dg=1;
        for(j=0;j<i;j++) polypowmodmod(g,dg,p,f,df,g,&dg,p);
        h[0]=0; h[1]=p-1; dh=1;
        polyaddmod(g,dg,h,dh,g,&dg,p);
        polygcdmod(g,dg,f,df,g,&dg,p);
        if(dg>0) return 0;
    }
    return 1;
}

/* calculate the legendre symbol of the element a (in polynomial format)
   in the field F_p^df:
   1 if element is a quadratic residue, -1 if not.
   p must be an odd prime! */
int polylegendre(ull *a,int da,ull *f,int df,ull p) {
    ull b[MAXDEG+1];
    mpz_t n,P;
    int db,i;
    for(i=0;i<=da;i++) if(a[i]) goto notzero;
    return 0;
notzero:
    mpz_init(n);
    mpz_init(P);
    mpz_set_ull(P,p);
    mpz_pow_ui(n,P,df);
    mpz_sub_ui(n,n,1);
    mpz_divexact_ui(n,n,2);
    polypowmodmodmpz(a,da,n,f,df,b,&db,p);
    mpz_clear(n);
    mpz_clear(P);
    if(b[0]==p-1) return -1;
    if(b[0]==1) return 1;
```

```
    error("error in polylegendre, res not 1 or -1");
    return 0;
}

int findexpdiv2(mpz_t P,int df) {
    mpz_t s;
    int r=0;
    mpz_init(s);
    mpz_pow_ui(s,P,df);
    mpz_sub_ui(s,s,1);
    while(!mpz_tstbit(s,0)) {
        r++;
        mpz_fdiv_q_2exp(s,s,1);
    }
    mpz_clear(s);
    return r;
}

/* given a, find b such that b^2=a in the field F_{p^df} given by the
   minimal polynomial f with degree df */
/* based on description in briggs */
/* algorithm is pretty much tonelli-shanks, adapted to F_{p^df} */
/* warning, i took a dubious short cut when implementing. p^df-1 should
   not have a divisor 2^s for a large s. this was circumvented by avoiding
   finite fields with this property */
void polysqrtmod(ull *a,int da,ull *f,int df,ull *b,int *db,ull p) {
    mpz_t s,z;
    ull j,c[MAXDEG+1],d[MAXDEG+1],e[MAXDEG+1];
    int r=0,dc,i,dd,t,de;
    /* does the square root exist? */
    if(1!=polylegendre(a,da,f,df,p)) { *db=-1; printf("not a square\n"); return; }
    mpz_init(s);
    mpz_init(z);
    /* write p^df-1 as 2^r * s for s odd */
    mpz_set_ull(s,p);
    mpz_pow_ui(s,s,df);
    mpz_sub_ui(s,s,1);
    while(!mpz_tstbit(s,0)) {
        r++;
        mpz_fdiv_q_2exp(s,s,1);
    }
    if(r>10) error("error, unsuitable r");
    /* find an element in F_{p^df} which is a non-residue */
    for(j=1;;j++) {
        for(dc=df-1,i=0;i<=dc;i++) c[i]=j;
        if(-1==polylegendre(c,dc,f,df,p)) break;
    }
    /* d=a^s */
    polypowmodmodmpz(a,da,s,f,df,d,&dd,p);
    /* find t such that c^2st = d. guaranteed to be <2^r */
```

```
    for(t=0;t<(1<<r);t++) {
        mpz_mul_ui(z,s,2*t);
        polypowmodmodmpz(c,dc,z,f,df,e,&de,p);
        /* c^2st == d? */
        if(de==dd) {
            for(i=0;i<=de;i++) if(e[i]!=d[i]) goto noteq;
            goto eq;
        }
    noteq:;
    }
    error("didn't find t in sqrt");
eq:;
    mpz_mul_ui(z,s,t);
    polypowmodmodmpz(c,dc,z,f,df,e,&de,p);
    /* calculate the inverse of e */
    polyinversemodmod(e,de,f,df,e,&de,p);
    /* the root is a^(s+1)/2 * e^-1 */
    mpz_add_ui(s,s,1);
    mpz_fdiv_q_2exp(s,s,1);
    polypowmodmodmpz(a,da,s,f,df,c,&dc,p);
    polymulmodmod(c,dc,e,de,f,df,b,db,p);
    mpz_clear(z);
    mpz_clear(s);
}

/* here follows some subroutines for polynomial arithmetic over Z */

/* multiply two polynomials, c(x)=a(x)*b(x) */
void polymulmpz(mpz_t *a,int da,mpz_t *b,int db,mpz_t *c,int *dc) {
    static mpz_t r[2*BIGDEG+2];
    int i,j;
    for(i=0;i<=da+db;i++) mpz_init_set_ui(r[i],0);
    for(i=0;i<=da;i++) for(j=0;j<=db;j++) mpz_addmul(r[i+j],a[i],b[j]);
    for(*dc=da+db,i=0;i<=*dc;i++) mpz_set(c[i],r[i]);
    for(i=0;i<=da+db;i++) mpz_clear(r[i]);
}

/* reduce a(x) mod f(x), return result in b(x) */
void polyreducempz(mpz_t *a,int da,mpz_t *f,int df,mpz_t *b,int *db) {
    mpz_t w[2*BIGDEG+2],t;
    int i,j,z;
    mpz_init(t);
    for(i=0;i<=da;i++) mpz_init_set(w[i],a[i]);
    for(;i<=df;i++) mpz_init_set_ui(w[i],0);
    /* for each i=da, da-1, ..., dv, subtract a(i)*v(x)*x^(i-dv) */
    for(i=da;i>=df;i--) for(j=0;j<=df;j++) {
        z=i-df;
        mpz_set(t,w[i]);
        mpz_submul(w[z+j],t,f[j]);
    }
```

```
    for(i=0;i<df;i++) mpz_set(b[i],w[i]);
    *db=df-1;
    /* tighten db */
    while(*db>-1 && !mpz_cmp_si(b[*db],0)) (*db)--;
    for(i=0;i<=da;i++) mpz_clear(w[i]);
    for(;i<=df;i++) mpz_clear(w[i]);
    mpz_clear(t);
}

/* given f, return g=f' */
void polyderivempz(mpz_t *f,int df,mpz_t *g,int *dg) {
    int i;
    *dg=df-1;
    for(i=1;i<=df;i++) mpz_mul_si(g[i-1],f[i],i);
    while(*dg>-1 && !mpz_cmp_si(g[*dg],0)) (*dg)--;
}

/* calculate the algebraic number and display it */
void printalgnum(mpz_t n,uchar *v,int cols,mpz_t *f,int df,mpz_t m,int *aval,int *bval) {
    mpz_t a[2*BIGDEG+2],b[BIGDEG+1];
    int da,db,i;
    for(i=0;i<2*BIGDEG+2;i++) mpz_init_set_ui(a[i],i==0);
    for(i=0;i<BIGDEG+1;i++) mpz_init(b[i]);
    da=0;
    /* multiply with f'(alpha)^2 */
    polyderivempz(f,df,b,&db);
    polymulmpz(a,da,b,db,a,&da);
    polyreducempz(a,da,f,df,a,&da);
    polymulmpz(a,da,b,db,a,&da);
    polyreducempz(a,da,f,df,a,&da);
    for(i=0;i<cols;i++) if(v[i]) {
        mpz_set_si(b[0],aval[i]);
        mpz_set_si(b[1],-bval[i]);
        db=1;
        polymulmpz(a,da,b,db,a,&da);
        polyreducempz(a,da,f,df,a,&da);
    }
    printf("algebraic square:\n");
    printmpzpoly(a,da);
    for(i=0;i<BIGDEG+1;i++) mpz_clear(b[i]);
    for(i=0;i<2*BIGDEG+2;i++) mpz_clear(a[i]);
}

/* get algebraic square root! v is the subset of (a,b) pairs */
/* use couveignes' algorithm */
int getalgroot(mpz_t n,uchar *v,int cols,mpz_t *in,int df,mpz_t m,mpz_t root,int *aval,int *bval) {
    double logest=0,b;
    mpz_t P,M,temp,ans;
    ull *q,pp,*ai,f[MAXDEG+1],fd[MAXDEG+1],g[MAXDEG+1],h[MAXDEG+1];
    ull n1,n2,xi;
```

```
    const ull MAX=(1ULL<<61)-1; /* start here to check for primes */
    int i,s,maxu,qn,dfd,j,dg,dh,k,ret=0;
    double zp;
    static int *ev;
    /* populate exponent vector */
    ev=calloc(bn1,sizeof(int));
    if(!ev) error("out of memory");
    for(i=0;i<cols;i++) if(v[i]) for(j=0;j<algn[i];j++) {
        if(alglist[i][j]<0 || alglist[i][j]>=bn1) error("error");
        ev[alglist[i][j]]++;
    }
    mpz_init(P);
    mpz_init_set_si(M,1);
    mpz_init(temp);
    mpz_init(ans);
    mpz_set_ui(ans,0);
    /* rough estimate:
       d^(d+5)/2 * n * (2*u*sqrt(d)*m)^(s/2)
       calculate log2 of this since it's huge */
    /* if this turns out to be bad, check the paper of couveignes for a
       tighter bound using complex roots and direct evaluation of stuff */
    logest=log2(df)*(df+5)*.5;
    logest+=mpz_sizeinbase(n,2);
    /* get u and s */
    maxu=0;
    for(i=0;i<cols;i++) {
        if(maxu<-aval[i]) maxu=-aval[i];
        if(maxu<aval[i]) maxu=aval[i];
        if(maxu<-bval[i]) maxu=-bval[i];
        if(maxu<bval[i]) maxu=bval[i];
    }
    for(s=i=0;i<cols;i++) s+=v[i];
    b=2*maxu*sqrt(df)*mpz_get_d(m);
    logest+=s*.5*log2(b);
    printf("estimate: %f bits\n",logest);
    /* find multiple q such that their product has >= logest digits */
    qn=(int)(1+logest/log2(MAX));
    q=malloc(qn*sizeof(ull));
    if(!q) error("out of memory in algroot");
    ai=malloc(qn*sizeof(ull));
    if(!ai) error("out of memory in algroot");
    /* don't be super duper tight and take primes just below 2^63.
       it seems there are overflow issues in some of the subroutines,
       the suspects are polyderivemod and polymulmodmod (and their callees) */
    for(pp=MAX,i=0;i<qn;pp+=2) {
        mpz_set_ull(P,pp);
        /* P must be prime and f(x) mod P must be irreducible */
        if(!mpz_probab_prime_p(P,30)) continue;
        if(!polyirredmod(in,df,pp)) continue;
        /* we also want to avoid P such that 2^r for large r divides P^df-1 */
```

```
        if(findexpdiv2(P,df)>5) continue;
        q[i++]=pp;
        mpz_mul(M,M,P);
    }
    /* for each i, compute a_i */
    for(zp=i=0;i<qn;i++) {
        mpz_set_ull(P,q[i]);
        mpz_fdiv_q(temp,M,P);
        pp=mpz_mod_ull(temp,q[i]);
        ai[i]=inverse(pp,q[i]);
    }
    /* for each q_i, calculate f'^2 * prod(a-bx) mod f, mod q_i
       and calculate its square root in Z_p/<f> */
    dfd=df-1;
    for(i=0;i<qn;i++) {
        for(j=0;j<=df;j++) f[j]=mpz_mod_ull(in[j],q[i]);
        polyderivemod(f,df,fd,&dfd,q[i]);
        /* form f'^2 * prod_{(a,b)} (a-b*alpha) mod q[i] */
        polymulmodmod(fd,dfd,fd,dfd,f,df,g,&dg,q[i]);
        for(j=0;j<cols;j++) if(v[j]) {
            h[0]=(aval[j]%(ll)q[i]+(ll)q[i])%(ll)q[i];
            h[1]=((-(ll)bval[j])%(ll)q[i]+(ll)q[i])%(ll)q[i];
            dh=1;
            polymulmodmod(g,dg,h,dh,f,df,g,&dg,q[i]);
        }
        /* take square root of g */
        polysqrtmod(g,dg,f,df,g,&dg,q[i]);
        /* sanity, not a square */
        if(dg<0) {
            printf("failed in %d of %d\n",i+1,qn);
            puts("error!");
            printf("p %I64d, f(x) mod p = ",q[i]);
            printullpoly(f,df);printf("\n");
            printf("g(x) mod p is not square: ");
            printullpoly(g,dg);printf("\n");
            goto quit;
        }
        /* norm of root (in g,dg) */
        n1=calcnormmod(g,dg,f,df,q[i]);
        /* norm of f'(alpha) */
        n2=calcnormmod(fd,dfd,f,df,q[i]);
        /* norm of square root of all prime ideals */
        for(j=0;j<bn1;j++) if(ev[j]) {
            /* norm of prime factor represented by the pair (p,r) is p */
            for(k=0;k+k<ev[j];k++) n2=ullmulmod2(n2,p1[j],q[i]);
        }
        /* if the norms are different, negate the root */
        if(n1!=n2) for(j=0;j<=dg;j++) g[j]=(q[i]-g[j])%q[i];
        n1=calcnormmod(g,dg,f,df,q[i]);
        if(n1!=n2) { printf("error %d/%d, norms are not equal!\n",i+1,qn); goto quit; }
```

```
            /* calculate a_i*x_i*P_i mod n and add it to result */
            mpz_set_ull(P,q[i]);
            mpz_fdiv_q(temp,M,P);
            mpz_set_ull(P,ai[i]);
            mpz_mul(temp,temp,P);
            xi=evalpolymod(g,dg,mpz_mod_ull(m,q[i]),q[i]);
            mpz_set_ull(P,xi);
            mpz_mul(temp,temp,P);
            mpz_add(ans,ans,temp);
            mpz_fdiv_r(ans,ans,M);
        }
        ret=1;
        mpz_set(root,ans);
        mpz_fdiv_r(root,root,n);
        mpz_mul(temp,root,root);
        mpz_fdiv_r(temp,temp,n);
        gmp_printf("root %Zd root^2 %Zd\n",root,temp);
quit:
        free(q);
        mpz_clear(ans);
        mpz_clear(temp);
        mpz_clear(M);
        mpz_clear(P);
        free(ev);
        return ret;
}

/* use trial division to check that a-bm (rational) and a-b*alpha (algebraic)
   are smooth with regard to our factor base. return 1 if smooth and also
   return the indexes of the factors in *f1,*f2,*f3. also set f0 to 1 if
   a-bm is negative. f3 will contain list of indexes where legendre
   symbol=-1. */
int trialsmooth(mpz_t a,mpz_t b,mpz_t *f,int deg,mpz_t m,int *f0,ull *f1,int *fn1,
                ull *f2,int *fn2,ull *f3,int *fn3) {
        mpz_t rat,alg,t,u,div;
        ull i,j,r,A,B;
        int ret=0;
        mpz_init(t);
        mpz_init(u);
        mpz_set(t,a);
        mpz_set(u,b);
        mpz_abs(t,t);
        mpz_abs(u,u);
        mpz_gcd(t,t,u);
        if(mpz_cmp_si(t,1)) goto cleanupgcd;
        mpz_init(div);
        /* rat = a-bm */
        mpz_init(rat);
        mpz_mul(rat,b,m);
        mpz_sub(rat,a,rat);
```

```
	/* check for negative a-bm */
	if(mpz_cmp_si(rat,0)<0) *f0=1,mpz_abs(rat,rat);
	else *f0=0;
	*fn2=0;
	/* trial division on a-bm */
	for(i=0;i<bn2;i++) {
		/* break if p2[i]^2 > rat */
		mpz_set_ull(div,p2[i]);
		mpz_mul(t,div,div);
		if(mpz_cmp(t,rat)>0) break;
		/* factor out div from rat and keep count */
		mpz_fdiv_qr(t,u,rat,div);
		if(mpz_cmp_si(u,0)) continue;
		mpz_set(rat,t);
		f2[(*fn2)++]=i;
		while(1) {
			mpz_fdiv_qr(t,u,rat,div);
			if(mpz_cmp_si(u,0)) break;
			mpz_set(rat,t);
			f2[(*fn2)++]=i;
		}
	}
	/* if remainder of rat > largest prime in factor base, number isn't smooth */
	mpz_set_ull(div,p2[bn2-1]);
	if(mpz_cmp(div,rat)<0) goto cleanuprat;
	if(mpz_cmp_si(rat,1)>0) {
		/* add remainder to primes */
		f2[(*fn2)++]=bs(p2,bn2,mpz_get_ull(rat));
		if(mpz_get_ull(rat)!=p2[bs(p2,bn2,mpz_get_ull(rat))])
			error("sanity test failed, rational remainder is not equal to prime found");
	}
	/* alg = norm(a-b*alpha) */
	mpz_init(alg);
	calcnorm(alg,a,b,f,deg);
	mpz_abs(alg,alg);
	*fn1=0;
	/* trial division on norm(a-b*alpha) */
	for(i=0;i<bn1;i++) {
		/* break if p1[i]^2 > alg */
		mpz_set_ull(div,p1[i]);
		mpz_mul(t,div,div);
		if(mpz_cmp(t,alg)>0) break;
		/* check if p1[i] divides alg */
		mpz_fdiv_r(t,alg,div);
		if(mpz_cmp_si(t,0)) continue;
		/* if a-br=0 mod p this is the prime we want */
		mpz_set_ull(t,r1[i]);
		mpz_mul(t,b,t);
		mpz_sub(t,a,t);
		mpz_fdiv_r(t,t,div);
```

```
            if(mpz_cmp_si(t,0)) continue;
            mpz_fdiv_q(alg,alg,div);
            /* factor out div from alg and keep count */
            f1[(*fn1)++]=i;
            while(1) {
                mpz_fdiv_qr(t,u,alg,div);
                if(mpz_cmp_si(u,0)) break;
                mpz_set(alg,t);
                f1[(*fn1)++]=i;
            }
        }
        /* check if alg>largest prime in factor base */
        mpz_set_ull(div,p1[bn1-1]);
        if(mpz_cmp(div,alg)<0) goto cleanupalg;
        if(mpz_cmp_si(alg,1)>0) {
            /* add reminder to primes */
            /* find index of first eligible pair (p,r) */
            i=bs(p1,bn1,mpz_get_ull(alg));
            if(i==-1) {
                gmp_printf("a = %Zd, b = %Zd\n",a,b);
                printf("tried to find %I64d, not in factor base\n",mpz_get_ull(alg));
                r=mpz_get_ull(alg);
                for(i=0;i<bn1;i++) if(p1[i]>r-1000 && p1[i]<r+1000)
                    printf("[%I64d %I64d] ",p1[i],r1[i]);
                error("\n");
            }
            /* find r such that a-br=0 (mod p) which is a*inverse(b) mod p */
            mpz_fdiv_r(u,a,alg);
            A=mpz_get_ull(u);
            mpz_fdiv_r(u,b,alg);
            B=mpz_get_ull(u);
            r=ullmulmod2(inverse(B,p1[i]),A,p1[i]);
            for(j=i;j<bn1;j++) {
                if(p1[j]!=p1[i]) break;
                if(r1[j]==r) goto ok;
            }
            error("(p,r) not found, shouldn't happen!");
        ok:
            f1[(*fn1)++]=j;
        }
        /* we won, (a,b) is smooth. now get the quadratic characters */
        *fn3=0;
        for(i=0;i<bn3;i++) {
            /* if legendre(a-br/p)==-1, then add this (p,r) */
            mpz_set_ull(t,p3[i]);
            mpz_set_ull(u,r3[i]);
            mpz_mul(u,b,u);
            mpz_sub(u,a,u);
            if(mpz_legendre(u,t)<0) f3[(*fn3)++]=i;
        }
```

```
	ret=1;
cleanupalg:
	mpz_clear(alg);
cleanuprat:
	mpz_clear(rat);
	mpz_clear(div);
cleanupgcd:
	mpz_clear(u);
	mpz_clear(t);
	return ret;
}

/* sieve from a1,b to a2,b, inclusive. restriction: a1 and a2 are int */
/* return 1 whenever enough relations are found */
int linesieve(int a1,int a2,int b,mpz_t n,mpz_t *f,int fn,mpz_t m,int extra,int *aval,int *bval) {
	int *sieve;
	mpz_t rat,norm,A,B,t,u;
	double invl2=1./log(2);
	ull j,z;
	int size=a2-a1+1,i,a,v,lgp,ret=0;
	int flog[MAXDEG+1];
	int blog[MAXDEG+1];
	double temp;
	if(!(sieve=malloc(size*sizeof(int)))) error("out of memory in line sieve");
	mpz_init(rat);
	mpz_init(norm);
	mpz_init(t);
	mpz_init(u);
	mpz_init(A);
	mpz_init(B);
	/* initialize rat=a-bm */
	mpz_set_si(B,b);
	mpz_mul(t,B,m);
	mpz_set_si(A,a1);
	mpz_sub(rat,A,t);
	mpz_set(t,rat);
	/* precalculate values for fast log_2(norm) */
	for(i=0;i<=fn;i++) flog[i]=mpz_sizeinbase(f[i],2);
	for(temp=0,i=0;i<=fn;i++,temp+=log(temp)*invl2) blog[i]=(int)(0.5+temp);
	for(a=a1,i=0;i<size;i++) {
		calcnorm(norm,A,B,f,fn);
		/* store lg norm + lg rat in sieve */
		v=mpz_sizeinbase(t,2)+mpz_sizeinbase(norm,2);
		/* fast version! approximate log2(norm) faster than calculating
		   the full norm every time */
		/* TODO */
		/* v+=mpz_sizeinbase(t,2); */
		sieve[i]=v;
		mpz_add_ui(t,t,1);
		mpz_add_ui(A,A,1);
```

```
    }
    /* process each rational prime */
    for(j=opt_skip;j<bn2;j++) {
        lgp=.5+log(p2[j])*invl2;
        /* find starting point: first smallest i>=0 such that a+i-bm=0 mod p */
        z=mpz_mod_ull(rat,p2[j]);
        /* subtract lg(prime) for each eligible element in sieve */
        for(i=z?p2[j]-z:0;i<size;i+=p2[j]) sieve[i]-=lgp;
    }
    /* process each algebraic prime */
    mpz_set_si(A,a1);
    for(j=opt_skip;j<bn1;j++) {
        lgp=.5+log(p1[j])*invl2;
        /* find starting point: find smallest i>=0 such that a+i-br=0 mod p */
        mpz_set_ull(t,r1[j]);
        mpz_mul(t,t,B);
        mpz_sub(t,A,t);
        z=mpz_mod_ull(t,p1[j]);
        for(i=z?p1[j]-z:0;i<size;i+=p1[j]) sieve[i]-=lgp;
    }
    /* find candidates for smooth numbers by taking the ones with small
       remaining log values. only taking 0-values is too strict, since
       sieve doesn't subtract powers of primes, and all logs are rounded
       to int */
    for(i=0;i<size;i++) {
        /* WARNING, magic constants */
        static ull f1[100000],f2[100000],f3[100000];
        int fn1=0,fn2=0,fn3=0,f0;
        a=a1+i;
        if(a==0) continue;
        if(gcd(a>0?a:-a,b>0?b:-b)>1) continue;
        if(sieve[i]<=opt_thr) {
            mpz_add_ui(t,A,i);
            if(trialsmooth(t,B,f,fn,m,&f0,f1,&fn1,f2,&fn2,f3,&fn3)) {
                /* insert in transposed matrix:
                   column i is the ith relation we find
                   row corresponds to -1, prime or quadratic character */
                if(f0) MSETBIT(M,0,smooth);
                for(j=0;j<fn1;j++) MTOGGLEBIT(M,1+f1[j],smooth);
                for(j=0;j<fn2;j++) MTOGGLEBIT(M,1+bn1+f2[j],smooth);
                for(j=0;j<fn3;j++) MSETBIT(M,1+bn1+bn2+f3[j],smooth);
                /* store the rational divisors */
                faclist[smooth]=malloc(fn2*sizeof(ull));
                if(!faclist[smooth]) error("out of memory trialsmooth");
                alglist[smooth]=malloc(fn1*sizeof(ull));
                if(!alglist[smooth]) error("out of memory trialsmooth");
                memcpy(faclist[smooth],f2,sizeof(ull)*fn2);
                facn[smooth]=fn2;
                memcpy(alglist[smooth],f1,sizeof(ull)*fn1);
                algn[smooth]=fn1;
```

```
                    /* store the actual a,b pair */
                    aval[smooth]=a1+i;
                    bval[smooth]=b;
                    smooth++;
                    if(smooth%100==0) {
                        printf("%d/%I64d found: (%d, %d) is smooth, log %d\n",
                                smooth,extra+1+bn1+bn2+bn3,a1+i,b,sieve[i]);
                    }
                    if(smooth==extra+1+bn1+bn2+bn3) {
                        puts("==> enough relations gathered!");
                        ret=1;
                        goto end;
                    }
                } else notsmooth++;
            } else {
                /* remove the continue if you want to benchmark
                   smooth numbers not found by the sieving */
                continue;
                mpz_add_ui(t,A,i);
                if(trialsmooth(t,B,f,fn,m,&f0,f1,&fn1,f2,&fn2,f3,&fn3)) {
                    printf("%d - %d*alpha is smooth, log %d MISSED\n",a1+i,b,sieve[i]);
                    missed++;
                }
            }
        }
end:
    mpz_clear(B);
    mpz_clear(A);
    mpz_clear(u);
    mpz_clear(t);
    mpz_clear(norm);
    mpz_clear(rat);
    free(sieve);
    return ret;
}

void testsieve(mpz_t n,mpz_t *f,int fn,mpz_t m,int extra,int *aval,int *bval) {
    int B;
    /* factor lists */
    puts("start sieve");
    notsmooth=missed=smooth=0;
    faclist=malloc((1+bn1+bn2+bn3+extra)*sizeof(ull*));
    if(!faclist) error("out of memory");
    facn=malloc((1+bn1+bn2+bn3+extra)*sizeof(int));
    if(!facn) error("out of memory");
    alglist=malloc((1+bn1+bn2+bn3+extra)*sizeof(ull*));
    if(!alglist) error("out of memory");
    algn=malloc((1+bn1+bn2+bn3+extra)*sizeof(int));
    if(!algn) error("out of memory");
    for(B=1;;B++) if(linesieve(-opt_sievew,opt_sievew,-1*opt_signb*B,n,f,fn,m,extra,aval,bval)) break;
```

```
    printf("smooth numbers found: %d\n",smooth);
    printf("nonsmooth numbers trial-divided: %d\n",notsmooth);
    printf("smooth numbers missed: %d\n",missed);
    puts("end sievetest");
}

void takegcd(mpz_t ans,mpz_t alg,mpz_t rat,mpz_t n) {
    mpz_t sub;
    mpz_init(sub);
    mpz_sub(sub,alg,rat);
    mpz_gcd(ans,n,sub);
    mpz_clear(sub);
}

/* takes a number n and returns a factor p, if found
   return values:
   1: factor found
   0: factor not found
   -1: n is even
   -2: n is a perfect power
   -3: n is probably prime
   -4: mysterious error */
int donfs(mpz_t n) {
    mpz_t m,f[MAXDEG+1],r,temp;
    mpz_t ratrot,algrot;
    ull Br=opt_Br,Ba=opt_Ba,rows,k;
    int *aval,*bval;
    int deg=opt_deg;
    int err,retval=0,i,Bk=opt_Bq,j;
    int extra=opt_extra,zero;
    uchar *v;
    uchar *freevar;
    mpz_init(r); mpz_init(m); mpz_init(temp);
    mpz_init(ratrot); mpz_init(algrot);
    for(i=0;i<=MAXDEG;i++) mpz_init(f[i]);
    /* check prerequisites: n cannot be even, prime or perfect power */
    /* (if n is a perfect power, try running nfs again on the root */
    mpz_fdiv_r_ui(r,n,2);
    if(!mpz_cmp_si(r,0)) { retval=-1; goto end; }
    if(mpz_perfect_power_p(n)) { retval=-2; goto end; }
    /* we want to be really, REALLY sure that n is composite */
    if(mpz_probab_prime_p(n,100)) { retval=-3; goto end; }
    mpz_set(m,opt_m);
    if(!mpz_cmp_si(m,0)) mpz_root(m,n,deg); /* deg-th root of n, get our base m */
    gmp_printf("m = %Zd\n",m);
    err=getpolynomial(n,m,deg,f);
    if(!err) error("polynomial isn't monic or is otherwise wrong");
    printmpzpoly(f,deg);
    /* for now, only try to find linear factors when
       the a_0 coefficient is small enough */
```

```
	/* TODO replace with better way to find all linear factors.
	   fully factorize f[0] (possibly by pollard rho or even qs) and
	   generate all divisors by generating all exponent tuples. in this way,
	   the program will have full degree 3 support */
	/* TODO move this to a function */
	if(mpz_cmp_si(f[0],2000000000)<0) {
		if(!mpz_cmp_si(f[0],0)) {
			printmpzpoly(f,deg);
			gmp_printf("f(x) factored, found factor %Zd\n",m);
			retval=1;
			goto end;
		}
		j=mpz_get_si(f[0]);
		for(i=1;i*i<=j;i++) if(j%i==0) {
			if(i>1) {
				mpz_set_si(r,-i);
				evalpoly(f,deg,r,temp);
				if(!mpz_cmp_si(temp,0)) {
					printmpzpoly(f,deg);
					gmp_printf("f(x) factored, found factor %d\n",i);
					retval=1;
					goto end;
				}
			}
			mpz_set_si(r,-j/i);
			evalpoly(f,deg,r,temp);
			if(!mpz_cmp_si(temp,0)) {
				printmpzpoly(f,deg);
				gmp_printf("f(x) factored, found factor %d\n",i);
				retval=1;
				goto end;
			}
		}
	}
	/* TODO try to factorize polynomial properly and terminate early */

	/* factor base */
	if(!Ba) Ba=findB(n)*1;
	if(!Br) Br=findB(n)*1;
	if(!Bk) Bk=findK(n)*0.25; /* number of quadratic characters */
	puts("factor base info:");
	printf(" bound %I64d\n",Ba);
	createfactorbases(Ba,Br,Bk,f,deg,&p1,&r1,&bn1,&p2,&bn2,&p3,&r3,&bn3);
	printf(" %I64d rational primes\n",bn2);
	printf(" %I64d algebraic primes\n",bn1);
	printf(" %d quadratic characters\n",Bk);
	printf(" total size %I64d\n",bn1+bn2+Bk);
	if(!extra) extra=3+(bn1+bn2+bn3+1)/1000;

	puts("continue with factorization!");
```

```
    /* allocate memory for matrix, uncompressed */
    rows=1+bn1+bn2+bn3;
    M=malloc(sizeof(uint *)*rows);
    for(i=0;i<rows;i++) {
        M[i]=calloc(((rows+31+extra)/32),sizeof(uint));
        if(!M[i]) error("out of memory while allocating matrix");
    }
    aval=malloc(sizeof(int)*(rows+extra));
    if(!aval) error("out of memory");
    bval=malloc(sizeof(int)*(rows+extra));
    if(!bval) error("out of memory");
    testsieve(n,f,deg,m,extra,aval,bval);
    puts("start gauss");
    bitgauss32(M,rows,rows+extra,0);
    v=malloc(rows+extra);
    if(!v) error("out of memory");
    freevar=malloc(rows+extra);
    if(!freevar) error("out of memory");
    zero=findfreevars(M,rows,rows+extra,freevar);
    printf("gauss done, %d free variables found\n",zero);
    for(k=0;k<zero;k++) {
        puts("------------------------------------");
        getsquare(M,rows,rows+extra,freevar,k,v);
        if(!getalgroot(n,v,rows+extra,f,deg,m,algrot,aval,bval)) continue;
        getratroot(n,v,rows+extra,f,deg,m,ratrot,aval,bval);
        gmp_printf("algroot %Zd ratroot %Zd\n",algrot,ratrot);
        takegcd(temp,algrot,ratrot,n);
        /* trivial result, try next linear combination */
        if(!mpz_cmp_si(temp,1) || !mpz_cmp(temp,n)) continue;
        gmp_printf("found factor %Zd after %d tries\n",temp,k+1);
        retval=1;
        break;
    }
    if(!retval) puts("no factor found");
    free(v);
end:
    for(i=0;i<=MAXDEG;i++) mpz_clear(f[i]);
    mpz_clear(ratrot); mpz_clear(algrot);
    mpz_clear(m); mpz_clear(r); mpz_clear(temp);
    return retval;
}

int main() {
    gmp_randinit_mt(gmpseed);
    gmp_randseed_ui(gmpseed,time(0));
    readoptions();
    gmp_printf("try to factor %Zd\n",opt_n);
    printf("return %d\n",donfs(opt_n));
    return 0;
}
```

Listing A.2: Sample input.

```
; input file for nfs
;
; first line: number to factor.
; - can be a literal number n
; - can be c[m], make a random composite number of m digits
; - can be r[m], make a random composite number of m digits that is the
; product of two similarly sized primes
; example from "cryptography, an introduction": n=45113 m=31 deg=3
;45113

; my example 1
4486873
; my example 2
;1027465709

; r80
;39436474109097683634320295131655814958311666003281971576453608419180282406191557
; r70
;4493658538520740276161242376826080121055754889927558057399451364896803
; r60
;160967735740568108627966290684899321608893044314961348169843
; r50
;32160137412888834732051225949878741400809992284289
; r40
;3565260354721980199129400248402571306803
; r39
;208105107011856763735887399456439331987
; r38
;25348924873403921164412907702279733193
; r37
;7511663247147032357037656316584448877
; r36
;228264844518616987380835399399539853
; r35
;78325683705012095897299536068804821
; r34
;1564875138070655023123959837084599
; r33
;523221436353855391814506581063557
; r32
;74520163184103070906530082210517
; r30
;189029013605764030727921585951
; r19
;7122214749230196817

; bounds:
; - algebraic factor base
; - rational factor base
```

```
; - number of quadratic characters
; enter 0 to let the program determine the values
140
140
6

; degree of polynomial
3
; m value (set to 0 to let program determine)
; warning, only choose m such that f(x) is monic of specified degree
0

; sieve width a (-a to a)
10000

; threshold for accepting numbers in the sieve (log in base 2)
20

; skip this number of smallest primes on each side
0

; number of extra relations wanted for linear algebra
3

; sign of b
-1
```