Wintersemester 2013/2014

Implementierung des Quadratischen Siebs in dem Computeralgebrasystem Sage

im Wahlprojekt *Grenzen der Algorithmik*

Daniel Christ Marcell Dietl Alexander Weinhandl

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Fachbereich Design Informatik Medien
Studiengang Angewandte Informatik (B.Sc.)

Betreuer: Steffen Reith
Version: 2014.1
Erstellt am: 31. Januar 2014

Zusammenfassung

Das Quadratische Sieb ist eines der ältesten und populärsten subexponentiellen Verfahren, mit dem sich eine zusammengesetzte ungerade natürliche Zahl, die keine perfekte Potenz ist, in ihre Primfaktoren zerlegen lässt. Ziel des vorliegenden Dokuments ist es, dem Leser sowohl die mathematischen als auch die technischen Hürden, welche zur Implementierung des genannten Faktorisierungsverfahrens gemeistert werden mussten, näherzubringen. Neben verschiedenen Beweisen aus der Zahlentheorie werden zusätzlich noch die Probedivision und Fermats Algorithmus als Faktorisierungsverfahren präsentiert, da sie dem Verständnis der eingesetzten Mathematik und der Idee hinter dem Quadratischen Sieb dienlich sind. Jedes Verfahren ist mit einer groben Laufzeitanalyse bzw. Laufzeituntersuchung, einem konkreten Zahlenbeispiel, Pseudocode und ausführbarem Quellcode untermauert. Letzterer ist in Python bzw. Cython, teilweise unter Zuhilfenahme des Computeralgebrasystems Sage, realisiert worden.

Inhaltsverzeichnis

1. Einleitung

Die Mathematik ist die Königin der Wissenschaften und
die Zahlentheorie ist die Königin der Mathematik.
Carl Friedrich Gauß

1.1. Überblick

Seit Menschengedenken sind Mathematiker wie Nicht-Mathematiker gleichermaßen von der Zahlentheorie fasziniert, einem Teilgebiet der Mathematik, das sich mit den Eigenschaften der ganzen Zahlen beschäftigt und dessen populärste Probleme sich oft dadurch auszeichnen, dass sie überraschend einfach zu verstehen, aber enorm schwer zu lösen sind.

Unter den ganzen Zahlen stechen vor allem die Primzahlen heraus, jene Zahlen größer 1, die nur durch 1 und sich selber ganzzahlig teilbar sind und deren unendliche Folge[1]

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, \ldots$$

seit jeher Anlass zu wilden Spekulationen gibt. Zwei Eigenschaften, die in Kapitel 2 gleich zu Anfang bewiesen werden, sind jedoch gewiss: Es gibt unendlich viele Primzahlen und jede natürliche Zahl größer 1 lässt sich, bis auf Umordnung der Faktoren, eindeutig als Produkt von Primzahlen darstellen.

Ausgehend von dieser wichtigen Erkenntnis werden in Kapitel 2 eine Reihe weiterer Beweise geführt, bei denen Primzahlen zumeist eine wesentliche Rolle spielen und die als Basis aller nachfolgenden Kapitel dienen.

In Kapitel 3 werden – aufbauend auf der Theorie des vorherigen Kapitels – drei Verfahren präsentiert, mit denen sich natürliche Zahlen größer 1 in ihre Primfaktoren zerlegen lassen: Die Probedivision, Fermats Algorithmus und das Quadratische Sieb. Zum Verständnis gibt es stets eine grobe Laufzeitanalyse bzw. Laufzeituntersuchung, ein konkretes Zahlenbeispiel, Pseudocode und eine Implementierung in Python[2] bzw. Cython[3], teilweise unter Zuhilfenahme des Computeralgebrasystems Sage [11].

Da sich die Implementierung des Quadratischen Siebs als durchaus komplex erweist, wird diese in Kapitel 4 eigenständig und detailliert behandelt. Einige der Tricks, die von der Theorie abweichen bzw. auf dieser aufbauen, werden kurz vorgestellt und mit beispielhaftem Python- bzw. Cython-Quellcode untermauert. Das Ergebnis ist eine in Anhang C hinterlegte Implementierung, deren Laufzeit in Anhang B anhand konkreter Faktorisierung von Zahlen mit bis zu 61 Dezimalstellen untersucht wird.

Eine Erklärung, der in dieser Arbeit verwendeten Notationen, ist in Anhang A zu finden.

[1] Siehe hierzu u. a.: http://oeis.org/A000040 (aufgerufen am 12.01.2014).
[2] Siehe hierzu u. a.: http://www.python.org/ (aufgerufen am 20.01.2014).
[3] Siehe hierzu u. a.: http://cython.org/ (aufgerufen am 07.01.2014).

1.2. Änderungen

Version	**Änderung(en)**
2014.1	Keine. Version, die zur Bewertung eingereicht wurde.

2. Mathematische Grundlagen

Mach' dir keine Sorgen wegen deiner Schwierigkeiten
mit der Mathematik. Ich kann dir versichern,
dass meine noch größer sind.
Albert Einstein

2.1. Primzahlen und der Fundamentalsatz der Arithmetik

Theorem 2.1.1. (Fundamentalsatz der Arithmetik[1]). *Jede natürliche Zahl $N > 1$ lässt sich, bis auf Umordnung der Faktoren, eindeutig als Produkt von Primzahlen darstellen:*

$$N = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_l^{e_l} = \prod_{k=1}^{l} p_k^{e_k}, \tag{2.1}$$

$p_i \in \mathbb{P}$ (p_i sind Primzahlen), $e_i \in \mathbb{N}$ (die Exponenten sind natürliche Zahlen), $1 \leq i \leq l$.

Beweis. Siehe hierzu u. a. [1, S. 23]. □

Beispiel. Die Zahl $N = 60$ ist eindeutig durch das Produkt

$$60 = 2 \cdot 2 \cdot 3 \cdot 5 = 2^2 \cdot 3 \cdot 5$$

definiert. In diesem konkreten Fall lauten die weiteren Werte von (2.1): $l = 3$, $p_1 = 2$, $e_1 = 2$, $p_2 = 3$, $e_2 = 1$, $p_3 = 5$ und $e_3 = 1$.

Wichtig ist jedoch nicht nur die Erkenntnis, dass sich jede natürliche Zahl größer 1 als Produkt von Primzahlen darstellen lässt und dass diese Darstellung, bis auf Umordnung der Faktoren, eindeutig ist, sondern vor allem, dass es unendlich viele Primzahlen gibt[2].

Lemma 2.1.1. *Wenn $p \mid a$ (p teilt a) und $p \mid b \Rightarrow p \mid a - b$, mit $a > b$ und $p, a, b \in \mathbb{N}$.*

Beweis. Aus $p \mid a$ folgt, dass $\exists k_1 \in \mathbb{N}$, mit $a = k_1 \cdot p$. Aus $p \mid b$ folgt, dass $\exists k_2 \in \mathbb{N}$, mit $b = k_2 \cdot p$. Demnach ist $a - b = (k_1 \cdot p) - (k_2 \cdot p) = (k_1 - k_2) \cdot p \Leftrightarrow p \mid a - b$. □

Theorem 2.1.2. *Es gibt unendlich viele Primzahlen.*

Beweis. Angenommen die Menge der Primzahlen $\mathbb{P}$ sei lediglich endlich, z.B. sei $\mathbb{P} = \{2, 3, 5, \ldots, p\}$. Nun können diese endlich vielen Primzahlen miteinander multipliziert und das Ergebnis anschließend um 1 erhöht werden:

$$N = (2 \cdot 3 \cdot 5 \cdot \ldots \cdot p) + 1$$

Ist N eine Primzahl, so fehlt sie in der Menge $\mathbb{P}$, da sie größer als alle bisherigen Elemente von $\mathbb{P}$ ist (Widerspruch!). Wenn N keine Primzahl, sondern zusammengesetzt ist, dann muss es eine Primzahl q geben, die N teilt. Diese Primzahl würde aber auch das Produkt $(2 \cdot 3 \cdot 5 \cdot \ldots \cdot p)$ teilen. Eine Zahl jedoch, die zwei verschiedene Zahlen teilt, muss gem. Lemma 2.1.1 auch deren Differenz teilen. Somit muss q auch $N - (2 \cdot 3 \cdot 5 \cdot \ldots \cdot p) = 1$ teilen. Da es keine Primzahl gibt, die 1 teilt, ist $\mathbb{P}$ nicht endlich (Widerspruch!).

Schlussfolgernd muss es unendlich viele Primzahlen geben. □

[1] Häufig auch als *Fundamental-* bzw. *Hauptsatz der elementaren Zahlentheorie* bezeichnet.
[2] Einen recht intuitiven Beweis dieser Behauptung lieferte schon Euklid [2] – etwa 300 v. Chr. [3].

2.2. Quadratische Reste und das Legendre-Symbol

Zum Verständnis des Legendre-Symbols, das ein Spezialfall des Jacobi-Symbols ist[1], sei die Kongruenz

$$x^2 \equiv a \pmod{m} \tag{2.2}$$

gegeben, wobei $x, a \in \mathbb{Z}$ (x und a sind ganze Zahlen), $m \in \mathbb{N}\backslash\{0\}$ (m ist eine natürliche Zahl größer 0) sowie $ggT(a, m) = 1$ (a und m sind teilerfremd). Die Frage, die sich nun stellt, ist, ob diese Kongruenz lösbar ist, also $\pm x$ existiert, das sie erfüllt.

Sollte die Kongruenz lösbar sein, nennt man a einen *quadratischen Rest* modulo m, andernfalls wird a als *quadratischer Nichtrest* modulo m bezeichnet.

Beispiel. Zwar ist 2 ein quadratischer Rest modulo 7, da

$$(\pm 3)^2 \equiv 2 \pmod{7},$$

aber ein quadratischer Nichtrest modulo 5, da für alle $x \in \mathbb{Z}$ gilt, dass

$$(\pm x)^2 \not\equiv 2 \pmod{5}.$$

Das Bestimmen der Lösungen von (2.2) kann mit Hilfe des Tonelli-Shanks-Algorithmus realisiert werden, vorausgesetzt m ist eine ungerade Primzahl p und a ein quadratischer Rest modulo p. Siehe hierzu Algorithmus 2.3.8 in [4, S. 100].

Die Frage, ob a ein quadratischer Rest oder Nichtrest modulo p ist, es also überhaupt Sinn macht, nach den Lösungen $\pm x$ der Kongruenz (2.2) zu suchen, wird mit Hilfe des Legendre-Symbols beantwortet.

Definition 2.2.1. Das Legendre-Symbol $\left(\frac{a}{p}\right)$ bzw. (a/p) ist für $a \in \mathbb{Z}$ und $p \in \mathbb{P}\backslash\{2\}$ (p ist eine ungerade Primzahl) definiert durch

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{wenn } a \text{ ein Vielfaches von } p \text{ ist,} \\ 1, & \text{wenn } a \text{ ein quadratischer Rest modulo } p \text{ ist,} \\ -1, & \text{wenn } a \text{ ein quadratischer Nichtrest modulo } p \text{ ist.} \end{cases}$$

Beispiel. Anknüpfend an die obigen Werte ist $\left(\frac{2}{7}\right) = 1$, $\left(\frac{2}{5}\right) = -1$ und $\left(\frac{9}{3}\right) = 0$.

[1] Für die Implementierung des Quadratischen Siebs ist nur dieser Spezialfall wichtig. Der Algorithmus 2.3.5 in [4, S. 98], welcher das Legendre-Symbol berechnet, ist jedoch so realisiert, dass er gleichermaßen für beide Symbole gilt.

2.2.1. Das Quadratische Reziprozitätsgesetz

Zum effizienten Berechnen des Legendre-Symbols steht das Quadratische Reziprozitätsgesetz zur Verfügung, das Euler 1783 erstmals vermutete und dessen Korrektheit von Gauß 1796 bewiesen wurde [5]. Dieses lässt sich, zusammen mit seinen Ergänzungssätzen, in verallgemeinerter Form auch auf das Jacobi-Symbol anwenden[1].

Theorem 2.2.1. (Quadratisches Reziprozitätsgesetz). *Seien $p, q \in \mathbb{P}\backslash\{2\}$, wobei $p \neq q$ (p und q sind zwei verschiedene ungerade Primzahlen), dann gilt*

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4} = \begin{cases} -1, & \text{wenn } p \equiv q \equiv 3 \pmod 4, \\ 1, & \text{sonst.} \end{cases} \tag{2.3}$$

Beweis. Siehe hierzu u. a. [1, S. 206–207]. □

Theorem 2.2.2. (2. Ergänzungssatz). *Sei $p \in \mathbb{P}\backslash\{2\}$, dann gilt*

$$\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8} = \begin{cases} -1, & \text{wenn } p \equiv 3 \pmod 8 \text{ oder } p \equiv 5 \pmod 8, \\ 1, & \text{sonst.} \end{cases} \tag{2.4}$$

Beweis. Siehe hierzu u. a. [1, S. 205–206]. □

Im Falle des Quadratischen Siebs sind neben Theorem 2.2.1 und Theorem 2.2.2 noch die folgenden drei Regeln – die sich ebenfalls im Algorithmus 2.3.5 zum Berechnen des Legendre-Symbols in [4, S. 98] wiederfinden – von Interesse ($a, b \in \mathbb{Z}$):[2]

$$\left(\frac{a}{p}\right) = \left(\frac{a \bmod p}{p}\right) \tag{2.5}$$

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right) \tag{2.6}$$

$$\left(\frac{1}{p}\right) = 1 \tag{2.7}$$

Beispiel. Vier ist ein quadratischer Rest modulo fünf, da

$$\left(\frac{4}{5}\right) \overset{(2.6)}{=} \left(\frac{2}{5}\right)\left(\frac{2}{5}\right) \overset{(2.4)}{=} (-1)^{(5^2-1)/8} \cdot (-1)^{(5^2-1)/8} = 1.$$

[1] Die Primzahlen p und q in (2.3), (2.4), (2.5), (2.6) und (2.7) können durch beliebige ungerade ganze Zahlen a und b größer 1 ersetzt werden, sofern diese teilerfremd sind, also $ggT(a, b) = 1$.

[2] Die drei Regeln (2.5), (2.6) und (2.7) können mit Hilfe des Euler-Kriteriums, siehe Abschnitt 2.2.2, auf einfache Weise bewiesen werden, vgl. hierzu u. a. [1, S. 204].

2.2.2. Das Euler-Kriterium

Einen direkten, technisch aber wenig effizienten Weg, um das Legendre-Symbol zu bestimmen, gibt das Euler-Kriterium an, dessen nachfolgender Beweis von einem deutlich kürzeren Beweis in [1, S. 204] inspiriert ist, der aber nur den Fall $p \nmid a$ betrachtet.

Lemma 2.2.1. *Ist p eine ungerade Primzahl, so gibt es eine primitive Restklasse mod p.*

Beweis. Siehe hierzu u. a. [1, S. 148–149]. □

Lemma 2.2.2. *Sei $p \in \mathbb{P}$, $a \in \mathbb{Z}$ und $p \nmid a$, dann gilt $a^{p-1} \equiv 1 \pmod{p}$.*

Beweis. Siehe hierzu u. a. [1, S. 142]. □

Lemma 2.2.3. *Wenn $p \mid a \Rightarrow p \mid a^b$, mit $p, a, b \in \mathbb{N}$ und $b > 0$.*

Beweis. Aus $p \mid a$ folgt, dass $\exists k \in \mathbb{N}$, mit $a = k \cdot p$. Potenzieren beider Seiten mit b ergibt schließlich $a^b = (k \cdot p)^b = k^b \cdot p^b = (k^b \cdot p^{b-1}) \cdot p \Leftrightarrow p \mid a^b$. □

Korollar 2.2.1. *Wenn $a \equiv 0 \pmod{p} \Rightarrow a^b \equiv 0 \pmod{p}$, mit $p, a, b \in \mathbb{N}$ und $b > 0$.*

Theorem 2.2.3. (Euler-Kriterium). *Sei $a \in \mathbb{Z}$ und $p \in \mathbb{P}\backslash\{2\}$, dann gilt*

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

Beweis. (Fall: $p \nmid a$) Sei $[\,g\,] = \{b \mid b \equiv g \pmod{p}\}$ eine primitive Restklasse mod p, mit $p \in \mathbb{P}\backslash\{2\}$, $b \in \mathbb{Z}$, so nennt man einen Repräsentanten $g \in \mathbb{Z}$ von $[\,g\,]$ die Primitivwurzel bzw. den Generator der primen Restklassengruppe $\mathbb{Z}_p^* = \{[\,g\,]^1, [\,g\,]^2, \ldots, [\,g\,]^{p-1}\}$. Die Existenz einer primitiven Restklasse $[\,g\,]$ mod p ist gem. Lemma 2.2.1 garantiert.

Zu jedem Repräsentanten $a \in \mathbb{Z}$ einer Restklasse $[\,a\,] \in \mathbb{Z}_p^*$ existiert folglich stets ein $\alpha \in \{1, 2, \ldots, p-1\}$, sodass $g^\alpha \equiv a \pmod{p}$.

Ist nun a ein quadratischer Rest modulo p, dann ist α gerade, etwa $\alpha = 2\beta$, denn daraus folgt, dass $(g^\beta)^2 \equiv a \pmod{p}$, was der Schreibweise von (2.2) entspricht. Ferner gilt, dass

$$a^{(p-1)/2} \equiv (g^{2\beta})^{(p-1)/2} \equiv (g^\beta)^{(p-1)} \equiv (g^{(p-1)})^\beta \equiv (1)^\beta \equiv 1 \pmod{p}. \tag{2.8}$$

Die Umformung im vorletzten Schritt erfolgt gem. Lemma 2.2.2.

Ist nun a ein quadratischer Nichtrest modulo p, dann ist α ungerade, etwa $\alpha = 2\beta + 1$, wodurch ferner folgt, dass

$$a^{(p-1)/2} \equiv (g^{(2\beta+1)})^{(p-1)/2} \equiv (g^{2\beta} \cdot g)^{(p-1)/2} \equiv (g^{2\beta})^{(p-1)/2} \cdot (g)^{(p-1)/2} \pmod{p}.$$

Der letzte Term kann dank (2.8) weiter vereinfacht werden, denn $(g^{2\beta})^{(p-1)/2} \equiv 1 \pmod{p}$, woraus letztlich folgt, dass

$$(g^{2\beta})^{(p-1)/2} \cdot (g)^{(p-1)/2} \equiv (g)^{(p-1)/2} \equiv (g^{(p-1)})^{1/2} \equiv (1)^{1/2} \equiv \sqrt{(1)} \equiv -1 \pmod{p}.$$

Im letzten Schritt kam als Ergebnis von $\sqrt{(1)} \equiv \pm 1 \pmod{p}$ nur -1 in Frage, da $(g)^{(p-1)/2} \equiv 1 \pmod{p}$ bedeuten würde, dass die Elementordnung von $g \leq (p-1)/2$ wäre. Sie ist aber gleich der Gruppenordnung der primen Restklassengruppe, also $p-1$.

(Fall: $p \mid a$) Da alle Primzahlen außer 2 ungerade sind, ist $p \in \mathbb{U}$ und folglich $(p-1)$ eine gerade natürliche Zahl, also $(p-1) \in \mathbb{N}\backslash\mathbb{U}$. Daraus folgt ferner, dass $(p-1)/2 \in \mathbb{N}$ eine natürliche Zahl größer 0 sein muss, da die kleinste ungerade Primzahl 3 ist und somit ist $(3-1)/2 = 1$. Aus $p \mid a$ folgt, dass $a \equiv 0 \pmod{p}$ und somit gilt gem. Korollar 2.2.1 auch $a^b \equiv 0 \pmod{p}$, für $b \in \mathbb{N}\backslash\{0\}$. Setzt man abschließend $b = (p-1)/2$ ist letztlich

$$a \equiv a^b \equiv a^{(p-1)/2} \equiv 0 \pmod{p}.$$

□

2.2.3. Pseudocode

Algorithm 1 Legendre-Symbol[1]

Input: $a \in \mathbb{Z}, p \in \mathbb{P}\backslash\{2\}$

```
1:  a ← a mod p
2:  t ← 1
3:
4:  while a ≠ 0 do
5:      while 2 | a do
6:          a ← a/2
7:          if p mod 8 ∈ {3, 5} then
8:              t ← −t
9:          end if
10:     end while
11:     a, p ← p, a
12:     if a ≡ p ≡ 3 (mod 4) then
13:         t ← −t
14:     end if
15:     a ← a mod p
16: end while
17:
18: if p = 1 then
19:     return t
20: end if
21: return 0
```

[1] Pseudocode auf Basis von Algorithmus 2.3.5 aus [4, S. 98]. Implementiert in Listing C.1.

2.3. Die Komplexitätsklassen PTIME und EXPTIME

Um verschiedene Algorithmen, unabhängig von technischen Faktoren, wie der konkreten Implementierung oder der verwendeten Hardware, vergleichen zu können, hat sich die Betrachtung von asymptotischen Laufzeitschranken[1] bewährt. Das bedeutet, dass man das Verhältnis zwischen der Eingabelänge – bei Faktorisierungsverfahren meist die Anzahl der Bits n – und der Laufzeit eines Algorithmus betrachtet. Eine Möglichkeit, einen Algorithmus grob in eine Kategorie bzgl. der Laufzeit einzuteilen, bietet die $\mathcal{O}$-Notation:

$$\mathcal{O}(f(n)) = \{g(n) \mid \exists c \in \mathbb{N}\backslash\{0\}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

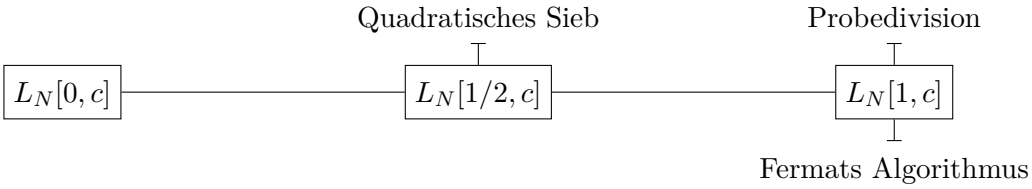
Sie besagt, dass eine Funktion $g(n) \in \mathcal{O}(f(n))$ ist, wenn sie ab einem Wert n_0 nicht schneller wächst als das c-fache von $f(n)$, also nicht wesentlich schneller als $f(n)$.

Die Komplexitätsklasse PTIME beinhaltet nun genau die Probleme, die von Algorithmen gelöst werden können, deren Laufzeit in $\mathcal{O}(n^k)$ liegt, $k \in \mathbb{N}$, k fest, die also in polynomieller Zeit terminieren. Diese Probleme werden auch *effizient lösbar* genannt. Im Gegensatz dazu steht die Komplexitätsklasse EXPTIME, die all jene Probleme beinhaltet, die durch Algorithmen gelöst werden können, deren Laufzeit in $\mathcal{O}(2^{p(n)})$ liegt, für ein beliebiges Polynom $p(n)$, die also in exponentieller Zeit terminieren.[2]

Beispielsweise existieren für das Problem der Faktorisierung großer natürlicher Zahlen derzeit keine bekannten effizienten Algorithmen, wenngleich seit den 1970er Jahren mehrere Algorithmen entwickelt wurden, die eine subexponentielle, jedoch noch immer superpolynomielle, Laufzeit aufweisen, vgl. hierzu u. a. [4, S. 225]. Um Laufzeitunterschiede zwischen derartigen Algorithmen herausstellen zu können, hat sich die L-Notation bewährt. Für $\alpha \in \mathbb{R}$, $0 \leq \alpha \leq 1$, $N \in \mathbb{N}$ und eine positive Konstante $c \in \mathbb{R}$ lautet sie:

$$L_N[\alpha, c] = e^{(c+o(1))(\ln N)^{\alpha}(\ln \ln N)^{1-\alpha}},\ \text{für } N \to \infty$$

Für $\alpha = 0$ ist $L_N[0, c] = e^{(c+o(1)) \ln \ln N} = (\ln N)^{c+o(1)}$ eine Funktion mit polynomiellem Wachstum in $\ln N$ und für $\alpha = 1$ ist $L_N[1, c] = e^{(c+o(1)) \ln N}$ eine Funktion mit exponentiellem Wachstum in $\ln N$. Da α angibt, wie nahe ein Algorithmus an PTIME herankommt, können Algorithmen auf einer Skala einsortiert werden, vgl. hierzu u. a. [4]:



[1] Angelehnt an den aus der Mathematik stammenden Begriff der *Asymptote*, nähert sich ein Algorithmus bei größer werdender Eingabelänge der *asymptotischen Laufzeitschranke* immer weiter an.

[2] Es ist mittlerweile bekannt, dass PTIME eine echte Teilmenge von bzw. ungleich EXPTIME ist.

3. Faktorisierungsverfahren

Manchmal muss man schwierige Dinge sagen, aber
man sollte sie so einfach sagen, wie man kann.
Godfrey Harold Hardy

3.1. Probedivision

Den intuitivsten Ansatz, um eine natürliche Zahl N größer 1 in ihre Primfaktoren zu zerlegen, verfolgt ein Faktorisierungsverfahren, das als *Probedivision* (engl. *trial division*) bekannt ist. Bei diesem Verfahren werden, grob gesprochen, nacheinander alle Primzahlen, beginnend bei 2, als Teiler von N ausprobiert und bei Erfolg vermerkt.

Der Vorteil der Probedivision liegt vor allem darin, dass sie bei gleicher Eingabe stets gleich ablaufen wird und irgendwann enden muss[1]. Denn aufgrund des Fundamentalsatzes der Arithmetik lässt sich N als Produkt von Primfaktoren gem. (2.1) schreiben.

3.1.1. Laufzeitanalyse

Entscheidend für die Anzahl der Divisionen, welche die Probedivision braucht, um N vollständig zu faktorisieren, ist vor allem die Auswahl der Teiler, durch die N dividiert werden soll. Im ersten Absatz ist der kürzeste Weg beschrieben: Lediglich Primzahlen müssen als Teiler von N ausprobiert werden. Das Problem besteht jedoch dann darin, dass zuvor alle in Frage kommenden Primzahlen ermittelt bzw. abgespeichert werden müssen, was z. B. die Implementierung eines Primzahltests erfordern kann und bei wachsendem N immer zeitintensiver bzw. speicherplatzfressender werden wird.

Eine Alternative, die auch in Algorithm 2 auf Seite 13 Anwendung findet, ist, die Zahl 2 und alle ungeraden Zahlen als Teiler von N in Betracht zu ziehen. Das funktioniert, da alle Primzahlen außer 2 ungerade sind. Und wenngleich auch viele zusammengesetzte ungerade Zahlen fälschlicherweise als Primteiler[2] in Betracht gezogen werden, kann es nicht passieren, dass sie letztlich als solche vermerkt werden[3].

Eine weitere Optimierung in Algorithm 2 besteht darin, eine Abbruchbedingung zu definieren. Sobald als Divisor von N' eine Zahl d getestet werden soll, für die $d^2 > N'$ bzw. $d > \sqrt{N'}$ gilt, wird N' als letzter Teiler der eingegebenen Zahl N vermerkt und die Faktorisierung ist vollbracht. Das folgt aus der Beobachtung, dass eine Zahl $N = p \cdot q$ nicht aus zwei Teilern p und q bestehen kann, die beide größer $\sqrt{N}$ sind, da sonst $p \cdot q > N$ wäre, was einen Widerspruch zur Anfangsbedingung $N = p \cdot q$ darstellt. Anders ausgedrückt: Zu jedem $p > \sqrt{N}$ existiert ein korrespondierendes $q < \sqrt{N}$, das durch den Algorithmus bereits überprüft wurde, da sich dieser von kleinen zu großen Zahlen vorarbeitet.

Eine weitere Erkenntnis ist folglich, dass $N \in \mathbb{P}$ den schlechtesten Fall der Probedivision darstellt und es ca. $\sqrt{N}/2$ Divisionen braucht, um zum Ende zu gelangen, vgl. hierzu u. a. [4, S. 120].

---

[1] Die Probedivision ist folglich ein deterministischer, terminierender Algorithmus für $N \in \mathbb{N}\backslash\{0, 1\}$.

[2] p ist Primteiler von N, wenn $p \mid N$ und $p \in \mathbb{P}$. Synonym zu Primfaktor verwendet.

[3] Wann immer als Divisor der Zahl N eine zusammengesetzte ungerade Zahl d in Betracht gezogen wird, sind deren Primteiler, die kleiner d sein müssen, bereits aus N herausgezogen worden, womit N nicht fälschlicherweise als teilbar durch d vermerkt werden wird.

Beispiel. Für $N = 492$ soll die Probedivision demonstriert werden:

$2 \mid 492$? Ja, denn $492 = 2 \cdot 246$.
$2 \mid 246$? Ja, denn $246 = 2 \cdot 123$.
$2 \mid 123$? Nein.

$3 \mid 123$? Ja, denn $123 = 3 \cdot 41$.
$3 \mid 41$? Nein.
$5 \mid 41$? Nein.
$7 \mid 41$? Abbruch, da $7^2 = 49 > 41$.

Folglich ist $N = 492 = 2^2 \cdot 3 \cdot 41$.

3.1.2. Pseudocode

Algorithm 2 Probedivision[1]

Input: $N \in \mathbb{N} \backslash \{0, 1\}$

```
1:  F ← [ ]
2:  N' ← N
3:
4:  while 2 | N' do
5:      N' ← N'/2
6:      F ← F ∪ 2
7:  end while
8:
9:  d ← 3
10: while d^2 ≤ N' do
11:     while d | N' do
12:         N' ← N'/d
13:         F ← F ∪ d
14:     end while
15:     d ← d + 2
16: end while
17:
18: if N' ≠ 1 then
19:     F ← F ∪ N'
20: end if
21:
22: return F
```

[1] Pseudocode auf Basis von Algorithmus 3.1.1 aus [4, S. 118–119]. Implementiert in Listing C.2.

3.2. Fermats Algorithmus

Eines der klassischen Faktorisierungsverfahren stammt von Fermat und wurde von diesem 1643 in einem Brief erwähnt [6]. Die Grundidee von Fermats Algorithmus besteht darin, eine zu zerlegende Zahl N als Differenz von Quadratzahlen darzustellen und auf diesem Wege eine Faktorisierung zu erreichen, da $N = x^2 - y^2 = (x + y) \cdot (x - y)$.

Theorem 3.2.1. *Sei $\mathbb{U}$ die Menge der ungeraden natürlichen Zahlen und sei $N = a \cdot b$, mit $N, a, b \in \mathbb{U}$, dann gilt: Es gibt Zahlen $x, y \in \mathbb{Z}$, $x \neq y$, sodass N als Differenz von deren Quadraten geschrieben werden kann: $N = x^2 - y^2 = (x + y) \cdot (x - y)$.*[1]

Beweis. Seien $N, a, b \in \mathbb{U}$ sowie $N = a \cdot b$. Sei $a = x + y$ und $b = x - y$, so folgt daraus, dass $a + b = (x + y) + (x - y) = 2x$ und $a - b = (x + y) - (x - y) = 2y$ bzw. ferner $x = (a + b)/2 \in \mathbb{Z}$ und $y = (a - b)/2 \in \mathbb{Z}$. Einsetzen und Umformen zeigt, dass

$$x^2 - y^2 = \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2 = \frac{\left(a^2 + 2ab + b^2\right) - \left(a^2 - 2ab + b^2\right)}{2^2} = \frac{4ab}{4} = a \cdot b = N,$$

was beweist, dass sich N stets als Differenz zweier Quadratzahlen darstellen lässt. □

Um aus dieser Existenzeigenschaft einen vollwertigen Algorithmus zur Faktorisierung einer Zahl $N = a \cdot b$, mit $N, a, b \in \mathbb{U}$ zu machen, wird wie folgt vorgegangen:

Zuerst wird $x = \lceil \sqrt{N} \rceil$ sowie $y^2 = x^2 - N$ gesetzt. Anschließend wird $\sqrt{y^2} = y$ berechnet und überprüft, ob $y \in \mathbb{Z}$ ist. Sollte dies nicht der Fall sein, wird x um 1 erhöht und der Test für $y^2 = (x + 1)^2 - N$ ausgeführt. Diese Inkrementierung von x und Überprüfung von y wird solange wiederholt, bis ein $y \in \mathbb{Z}$ gefunden wurde. Gelingt dies, können die Faktoren a und b von N wie folgt berechnet werden: $a = x + y$ und $b = x - y$.

3.2.1. Laufzeitanalyse

Fermats Algorithmus kann nur dann eine gute Laufzeit aufweisen, wenn sich eine zusammengesetzte ungerade natürliche Zahl N in zwei annähernd gleich große Faktoren zerlegen lässt. Andernfalls wird sich der Algorithmus recht weit von $\lceil \sqrt{N} \rceil$ entfernen müssen, um die gesuchten Quadratzahlen zu finden. Den schlechtesten Fall stellt es folglich dar, wenn N das dreifache einer Primzahl p, also $N = 3p$, ist. Denn das ist maximale Distanz, die die Faktoren einer ungeraden natürlichen Zahl N haben können.

In diesem Fall ist $x = (3 + p)/2$, $y = (3 - p)/2$ und folglich

$$N = \left(\frac{3+p}{2}\right)^2 - \left(\frac{3-p}{2}\right)^2.$$

[1] Vereinfacht ausgedrückt, besagt Theorem 3.2.1, dass sich jede zusammengesetzte ungerade natürliche Zahl als Differenz von Quadratzahlen darstellen lässt.

Beispiel. Für $N = 314731$, $\lceil \sqrt{n} \rceil = 562$, soll Fermats Algorithmus demonstriert werden:

k	x_k	$y_k^2 = x_k^2 - N$	y_k
0	562	1113	33,361...
1	563	2238	47,307...
2	564	3365	58,008...
3	565	4494	67,037...
4	566	5625	75

Bereits nach vier Schritten wird die Quadratzahl $5625 = 75^2$ gefunden, was zur Zerlegung $N = 566^2 - 75^2 = (566 + 75) \cdot (566 - 75) = 491 \cdot 641$ führt. Da es sich in diesem Fall bei beiden Faktoren um Primzahlen handelt, ist N vollständig faktorisiert worden.

Sollte ein Faktor keine Primzahl sein, so kann Fermats Algorithmus für diesen erneut ausgeführt werden. Dies lässt sich solange fortführen, bis es sich bei jedem gefundenen Faktor um eine Primzahl handelt und somit um die Primfaktorzerlegung gem. (2.1).

3.2.2. Pseudocode

Algorithm 3 Fermats Algorithmus[1]

Input: Zusammengesetztes $N = a \cdot b$, wobei $N, a, b \in \mathbb{U}$ sein müssen

1: $x \leftarrow \lceil \sqrt{N} \rceil$
2: $y^2 \leftarrow x \cdot x - N$
3:
4: **while** $\sqrt{y^2} \notin \mathbb{Z}$ **do**
5: $\quad y^2 \leftarrow y^2 + 2 \cdot x + 1$
6: $\quad x \leftarrow x + 1$
7: **end while**
8:
9: **return** $\left[x + \sqrt{y^2},\ x - \sqrt{y^2} \right]$

Um stetiges Quadrieren zu vermeiden, wird in Zeile 5 von Algorithm 3 auf die erste binomische Formel zurückgegriffen. Aufgrund der Tatsache, dass $(x + 1)^2 - N = (x^2 + 2x + 1) - N = (x^2 - N) + 2x + 1$ ist, lassen sich die y_k^2-Werte des obigen Beispiels effizient aus ihren jeweiligen Vorgängern plus $2x + 1$ berechnen.

[1] Implementiert in Listing C.3.

3.3. Quadratisches Sieb

Das Quadratische Sieb ist ein Algorithmus zur Faktorisierung großer natürlicher Zahlen mit subexponentieller Laufzeit, das von Pomerance Anfang der 1980er Jahre entwickelt und veröffentlicht wurde [7]. Es basiert auf Fermats Algorithmus, siehe Abschnitt 3.2, sowie dessen Erweiterung durch Kraitchik und ist auch von anderen zur damaligen Zeit gebräuchlichen Faktorisierungsverfahren inspiriert worden [8].

Für Zahlen mit bis zu 100 Dezimalstellen gilt das Quadratische Sieb als das schnellste allgemeine[1] Faktorisierungsverfahren, vgl. hierzu u. a. [9, S. 123] und [10, S. 492]. Zu klein sollte die Zahl allerdings auch nicht sein: Für Zahlen mit weniger als ca. 22 Dezimalstellen lieferte das Quadratische Sieb aus Abschnitt C.4 oft nur triviale Faktorisierungen.

3.3.1. Grundidee

Um eine ungerade natürliche Zahl N, die keine Primzahl und keine perfekte Potenz[2] ist, zu faktorisieren, werden zwei Zahlen $x, y \in \mathbb{Z}$, $x \neq y$, gesucht, sodass

$$x^2 \equiv y^2 \pmod{N} \Leftrightarrow x^2 - y^2 \equiv 0 \pmod{N} \tag{3.1}$$

sowie

$$x \not\equiv \pm y \pmod{N} \Leftrightarrow x \pm y \not\equiv 0 \pmod{N} \tag{3.2}$$

gilt. So ist sichergestellt, dass $N \mid (x^2 - y^2)$ bzw. $N \mid (x+y) \cdot (x-y)$, aber $N \nmid (x+y)$ und $N \nmid (x-y)$, womit $ggT(x+y, N)$ und $ggT(x-y, N)$ nichttriviale Teiler von N sind.

Der Vorteil gegenüber Fermats Algorithmus besteht nun darin, dass kein einzelnes x gefunden werden muss, für das $q(x) = x^2 - N$ eine Quadratzahl ist. Stattdessen genügt es schon durch Kombination mehrerer $q(x_i)$, $0 \leq i \leq k$, eine Quadratzahl

$$y^2 = q(x_0) \cdot q(x_1) \cdot \ldots \cdot q(x_k) = (x_0^2 - N) \cdot (x_1^2 - N) \cdot \ldots \cdot (x_k^2 - N)$$

zu erzielen, denn daraus folgt, dass

$$y^2 \equiv x_0^2 \cdot x_1^2 \cdot \ldots \cdot x_k^2 = (x_0 \cdot x_1 \cdot \ldots \cdot x_k)^2 = x^2 \pmod{N},$$

da $x_i^2 - N \equiv x_i^2 \pmod{N}$ gilt. Die gefundenen Quadratzahlen, x^2 und y^2, erfüllen bereits (3.1) und können somit zum Faktorisieren von N verwendet werden. Etwa jedes zweite Mal führt diese Zerlegung jedoch zu einer trivialen Faktorisierung, $N = N \cdot 1$ bzw. $N = 1 \cdot N$, nämlich immer dann, wenn (3.2) verletzt wird. In diesem Fall muss der Vorgang wiederholt und die Faktorisierung mit anderen Quadratzahlen versucht werden.

[1] *allgemein* bedeutet, dass die Laufzeit des Quadratischen Siebs nur von der Größe der zu faktorisierenden Zahl abhängt, nicht aber von speziellen Eigenschaften dieser Zahl.

[2] Siehe hierzu u. a.: http://mathworld.wolfram.com/PerfectPower.html (aufgerufen am 12.01.2014).

3.3.2. Initialisieren

Bevor der eigentliche Siebvorgang beginnen kann, bedarf es einer sogenannten Faktorbasis F_B, die alle Primzahlen kleiner gleich einer festgelegten Zahl $B \in \mathbb{N}$ enthält, zzgl. der Zahl -1, die zwar keine Primzahl ist, aber wie eine solche gehandhabt wird und stets in F_B liegt. Die Hinzunahme von -1 verhindert, dass die zu untersuchenden $q(x_i)$, $x_{min} \leq x_i \leq x_{max}$, siehe unten, zu groß werden, da auch nach negativen Werten gesiebt werden kann, bei denen die -1 als erster Faktor vermerkt wird. Zusammenfassend ist

$$F_B = \{p \in \mathbb{P} \mid p \leq B\} \cup \{-1\}.$$

Es sind im Folgenden nur die $|\, q(x_i) \,|$ interessant, deren Primfaktorzerlegung ausschließlich aus Primzahlen $\leq B$, besteht. Diese $q(x_i)$ heißen *B-glatt*.

Der Wert von B wird durch eine allgemeingültige Formel bestimmt, die sich in der Praxis bewährt und zu gleichmäßig guten Ergebnissen geführt hat:

$$B = \sqrt{e^{\sqrt{\ln N \cdot \ln \ln N}}}$$

Aus dem so ermittelten B werden üblicherweise noch die Grenzen des Siebintervalls abgeleitet, also der Startpunkt x_{min} und der Endpunkt x_{max}, zwischen denen die Werte $q(x_i)$ berechnet und auf B-Glattheit hin untersucht werden sollen. Die Länge des Intervalls wird häufig mit M bezeichnet und lässt sich bspw. wie folgt abschätzen:

$$M = B^2 = e^{\sqrt{\ln N \cdot \ln \ln N}}$$

Folglich ist das Siebintervall $I = \left[x_{min}, x_{max}\right] \cap \mathbb{N} = \left[\lceil \sqrt{N} \rceil - \frac{M}{2}, \lceil \sqrt{N} \rceil + \frac{M}{2}\right] \cap \mathbb{N}$.

Beide Werte, B und M, kann man bei Bedarf nach unten bzw. oben korrigieren, um u. U. bessere Ergebnisse, also eine schnellere Faktorisierung, zu erzielen, da es sich letztlich nur um Richtwerte handelt. Für bestimmte Zahlen mag dies zwar funktionieren, i. A. besteht jedoch die Gefahr, dass man damit die Faktorisierung eher gefährdet denn beschleunigt.

3.3.3. Sieben

Um unnötige Operationen einzusparen, kann die Faktorbasis F_B weiter eingeschränkt werden: Es werden alle Primzahlen $p \in F_B$ größer 2 gestrichen, für die N kein quadratischer Rest modulo p ist, da diese nie als Teiler von $q(x_i) = x_i^2 - N$, $x_i \in I$, auftreten. Dies liegt daran, dass $p \mid q(x_i)$ gdw. $x_i^2 - N \equiv 0 \pmod{p} \Leftrightarrow x_i^2 \equiv N \pmod{p}$, d. h. nur dann, wenn N ein quadratischer Rest modulo p ist und somit $(N/p) = 1$, siehe Abschnitt 2.2. Zusammenfassend ist die eingeschränkte Faktorbasis

$$F'_B = F_B \backslash \{p \in F_B \mid p > 2 \wedge (N/p) \neq 1\}.$$

Des Weiteren wird die Menge $S = \{q(x_i) \mid x_i \in I\}$ erzeugt, welche alle Zahlen enthält, auf denen der Siebvorgang durchgeführt werden soll, die also auf ihre B-Glattheit hin untersucht werden sollen. Um zu bestimmen, ob eine Zahl $q(x_i)$ B-glatt ist, könnte man diese durch Probedivision oder ein anderes Faktorisierungsverfahren vollständig in ihre Primfaktoren zerlegen. Allerdings wäre dies zeitlich äußerst ineffizient, weswegen man u. a. auf das nachfolgend beschriebene Siebverfahren zurückgreift:

Ist unter den ersten p Zahlen $q(x_i) = x_i^2 - N$, $x_{min} \leq x_i < x_{min+p}$, eine Zahl durch p teilbar, so ist, von dieser Zahl ausgehend, jede p-te Zahl ebenfalls durch p teilbar[1]. Für $k \in \mathbb{N}$ gilt nämlich, dass

$$\begin{aligned} q(x_i + kp) &= (x_i + kp)^2 - N \\ &= x_i^2 + 2x_i kp + (kp)^2 - N \\ &= (x_i^2 - N) + 2x_i kp + (kp)^2 \\ &= q(x_i) + 2x_i kp + (kp)^2 \\ &\equiv q(x_i) \pmod{p}. \end{aligned}$$

Die entsprechend gefundenen Zahlen, die alle durch p teilbar sind, werden anschließend sooft durch p geteilt, wie es möglich ist. Erfolglose Divisionen werden somit auf ein Minimum reduziert. Ein Spezialfall stellt die -1 dar, die keine Primzahl, aber Element der Faktorbasis F'_B ist. Um dieser gerecht zu werden, multipliziert man alle negativen Zahlen der Menge S zuvor mit -1 und kann auf den positiven Werten wie gewohnt verfahren.

Am Ende des Siebvorgangs, nachdem alle $p \in F'_B$ abgearbeitet worden sind, lassen sich die B-glatten Zahlen aus S direkt herauslesen. Bei allen Zahlen, die nach den durchgeführten Divisionen gleich 1 sind, handelt es sich um B-glatte Zahlen, da sich diese als Produkt von Primzahlen aus F'_B, zzgl. der Zahl -1, darstellen lassen.

3.3.4. Faktorisieren

Nachdem im Siebschritt alle B-glatten Zahlen der Menge S gefunden wurden, wird nun zu jeder B-glatten Zahl $q(x_i)$ ein Exponentenvektor $\vec{v}(q(x_i))$ aufgebaut. Dieser enthält jeweils die Exponenten der Primzahlen, zzgl. der Zahl -1, die miteinander multipliziert $q(x_i)$ ergeben[2]. Ist beispielsweise die Primfaktorzerlegung von

$$q(x_i) = x_i^2 - N = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_l^{e_l},$$

dann ist der dazu passende Exponentenvektor

$$\vec{v}(q(x_i)) = \vec{v}(x_i^2 - N) = (e_1, e_2, \ldots, e_l).$$

[1] Es genügt, unter den ersten p Zahlen nach höchstens zwei durch p teilbaren Zahlen zu suchen, da das Polynom $x_i^2 - N$ im Körper $\mathbb{Z}_p$ höchstens zwei Nullstellen hat, die binnen p Zahlen auftreten.

[2] Eine Information, die man im Siebschritt bereits erhalten hat, als getestet wurde, wie oft eine Zahl $q(x_i)$ durch eine Primzahl p teilbar ist. Die Anzahl möglicher Divisionen ist der Exponent von p. Ist eine Zahl nicht durch p teilbar, ist der Exponent entsprechend 0.

Um durch Kombinationen mehrerer $q(x_i)$, $0 \leq i \leq k$, letztendlich eine Quadratzahl zu erhalten, muss jeder Faktor der miteinander multiplizierten $q(x_i)$ geradzahlig oft bzw. gar nicht vorkommen, denn dann ist

$$q(x_0) \cdot q(x_1) \cdot \ldots \cdot q(x_k) = p_1^{2m \cdot e_1} \cdot p_2^{2n \cdot e_2} \cdot \ldots \cdot p_l^{2o \cdot e_l} = (p_1^{m \cdot e_1} \cdot p_2^{n \cdot e_2} \cdot \ldots \cdot p_l^{o \cdot e_l})^2 = y^2,$$

wobei $m, n, o \in \mathbb{N}$ sind und folglich $2m, 2n, 2o \in \mathbb{N} \backslash \mathbb{U}$. Um die Suche nach derartigen Kombinationen zu beschleunigen, wird aus den Exponentenvektoren $\vec{v}(q(x_i))$ eine Matrix aufgebaut, deren Zeilen die Exponentenvektoren, reduziert modulo 2, sind[1].

Mit Hilfe des Gaußschen Eliminationsverfahrens wird anschließend eine nichttriviale Linearkombinationen[2] von Zeilen der Matrix gesucht, die den Nullvektor ergeben. Um die Wahrscheinlichkeit zu erhöhen, dass eine solche Linearkombination auch tatsächlich gefunden wird, muss es – vereinfacht ausgedrückt – mehr Zeilen als Spalten geben.

Da die Anzahl der Spalten der Anzahl der Elemente in der Faktorbasis entspricht und die Zeilen den gefundenen B-glatten Zahlen, sollte es folglich mehr B-glatte Zahlen denn Elemente in der Faktorbasis geben. In der Praxis hat es sich bewährt, wenn es min. zehn B-glatte Zahlen mehr als Elemente in der Faktorbasis gibt, vgl. hierzu u. a. [4, S. 268].

Wurde schließlich eine nichttriviale Linearkombination gefunden, die den Nullvektor ergibt, führen die aufmultiplizierten $q(x_i)$ unweigerlich zu einer Quadratzahl y^2, die kongruent zu einer Quadratzahl x^2 modulo N ist, also $x^2 \equiv y^2 \pmod{N}$, womit bereits die Bedingung (3.1) erfüllt ist. Die Faktorisierung von N lautet folglich:

$$N = ggT(x - y, N) \cdot \frac{N}{ggT(x - y, N)},$$

was etwa jedes zweite Mal zu einer trivialen Faktorisierung $N = N \cdot 1$ bzw. $N = 1 \cdot N$ führt. In diesem Fall wiederholt man die Faktorisierung mit einer weiteren nichttrivialen Linearkombination, die den Nullvektor ergibt. Sollte keine der gefundenen nichttrivialen Linearkombinationen zu einer nichttrivialen Faktorisierung von N führen, muss im schlimmsten Fall von vorne begonnen werden, um mehr B-glatte Zahlen zu finden.

3.3.5. Laufzeitanalyse

In Abschnitt B.1 ist eine detaillierte Laufzeituntersuchung des in Listing C.4 implementierten Quadratischen Siebs angegeben.

[1] Es reicht aus, die Matrix über dem endlichen Körper GF(2) zu betrachten, da es lediglich um die Suche nach Kombinationen geht, die zu geradzahligen Exponenten führen. Ist ein Exponent bzw. die Summe mehrerer kongruent zu 0 modulo 2, so ist dieser geradzahlig und die ihm zugehörige Primzahl kommt folglich geradzahlig oft vor, andernfalls nicht.

[2] Eine triviale Linearkombination ist eine, bei der alle Koeffizienten gleich 0 sind, was der Kombination gar keiner Zahlen $q(x_i)$ entspricht und somit nicht zum Ziel führen kann.

Beispiel. Für $N = 87463$ soll das Quadratische Sieb demonstriert werden:

Im ersten Schritt muss eine Faktorbasis F_B angelegt werden, die alle Primzahlen kleiner gleich einer festgelegten Zahl $B \in \mathbb{N}$ enthält, zzgl. der Zahl -1. Da es sich bei N um eine vergleichsweise kleine Zahl handelt, führt die in Abschnitt 3.3.2 angegebene Formel zur Bestimmung von B zu einem zu geringen Wert, weswegen im folgenden

$$B = 19$$

gewählt wird. Folglich ist die Faktorbasis

$$F_B = \{-1, 2, 3, 5, 7, 11, 13, 17, 19\}.$$

Der Parameter $M = B^2 = 19^2 = 361$, aus dem üblicherweise die Grenzen des Siebintervalls abgeleitet werden, ist hingegen relativ groß, wenn man die in Abschnitt 3.3.2 angegebene Formel verwendet, weswegen er ebenfalls künstlich gewählt wird[1]:

$$M = 40$$

Daraus ergibt sich schließlich das Siebintervall

$$\begin{aligned} I &= \left[\lceil\sqrt{N}\,\rceil - \frac{M}{2}, \lceil\sqrt{N}\,\rceil + \frac{M}{2}\right] \cap \mathbb{N} \\ &= [296 - 20, 296 + 20] \cap \mathbb{N} \\ &= \{276, 277, 278, \ldots, 296, \ldots, 314, 315, 316\}. \end{aligned}$$

Im nächsten Schritt, siehe Abschnitt 3.3.3, wird die Faktorbasis weiter eingeschränkt, um unnötige Operationen einzusparen. Alle Primzahlen $p \in F_B$ größer 2, für die N kein quadratisches Rest modulo p ist, also $(N/p) \neq 1$, werden aus ihr entfernt:

$$F'_B = F_B \backslash \{5, 7, 11\} = \{-1, 2, 3, 13, 17, 19\}$$

Des Weiteren wird die Menge $S = \{q(x_i) = x_i^2 - N \mid x_i \in I\}$ erzeugt, deren Zahlen anschließend auf B-Glattheit hin untersucht werden sollen:

$$\begin{aligned} S &= \{q(276), q(277), q(278), \ldots, q(296), \ldots, q(314), q(315), q(316)\} \\ &= \{-11287, -10734, -10179, \ldots, 153, \ldots, 11133, 11762, 12393\} \end{aligned}$$

Um mit negativen Zahlen wie gewohnt verfahren zu können, werden diese noch mit -1 multipliziert, sodass

$$S = \{11287, 10734, 10179, \ldots, 153, \ldots, 11133, 11762, 12393\}.$$

Nun kann der eigentliche Siebvorgang durchgeführt werden, der hier nur beispielhaft demonstriert werden soll. Zuerst wird unter den ersten 2 Zahlen nach einer durch 2 teilbaren Zahl gesucht. Diese ist schnell gefunden, da $|\,q(277)\,| = 10734$ – genau einmal –

[1] Es wird nun deutlich, warum man das Quadratische Sieb für vergleichsweise kleine Zahlen in der Praxis nicht verwenden und stattdessen ein simpleres Faktorisierungsverfahren einsetzen sollte.

durch 2 teilbar ist. Folglich ist auch jede Zahl $|\, q(277 + k \cdot 2)\,|$, $k \in \mathbb{N}\backslash\{0\}$, – mindestens einmal – durch 2 teilbar.

Das gleiche Prozedere wird nun für die nächste Primzahl der Faktorbasis F'_B, die Zahl 3, wiederholt: Unter den ersten 3 Zahlen sind sowohl $|\, q(277)\,| = 10734$ als auch $|\, q(278)\,| = 10179$ durch 3 teilbar. Erstere nur einmal, letztere sogar dreimal. Folglich sind auch alle Zahlen $|\, q(277 + k \cdot 3)\,|$ sowie $|\, q(278 + k \cdot 3)\,|$ – mindestens einmal – durch 3 teilbar.

Hat man nach jeder erfolgreichen Division mit den durch p geteilten Zahlen $q(x_i)$ die alten Werte in der Menge S überschrieben, lassen sich die B-glatten Zahlen direkt aus

$$S = \{11287, 1789, 29, \ldots, 1, \ldots, 1237, 5881, 1\}$$

herauslesen, da es die Zahlen gleich 1 sind. Im obigen Beispiel können sowohl $q(296)$ als auch $q(316)$ als Produkt von Primzahlen der Faktorbasis F'_B geschrieben werden, denn

$$q(296) = 153 = 3 \cdot 3 \cdot 17 = 3^2 \cdot 17$$

und

$$q(316) = 12393 = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 17 = 3^6 \cdot 17.$$

Des Weiteren ist $q(299) = 1938 = 2 \cdot 3 \cdot 17 \cdot 19$ ebenfalls B-glatt. Wenngleich die Anzahl B-glatter Zahlen bedenklich niedrig ist, werden diese im weiteren Verlauf des Beispiels ausreichend sein, um N zu faktorisieren.

Als nächstes wird zu jeder gefundenen B-glatten Zahl $q(x_i)$ ein Exponentenvektor $\vec{v}(q(x_i))$ aufgebaut, siehe Abschnitt 3.3.4:

$$\begin{aligned}\vec{v}(q(296)) &= (0, 0, 2, 0, 1, 0)\\ \vec{v}(q(299)) &= (0, 1, 1, 0, 1, 1)\\ \vec{v}(q(316)) &= (0, 0, 6, 0, 1, 0)\end{aligned}$$

Diese bilden, reduziert modulo 2, die Zeilen einer Matrix

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Mit Hilfe des Gaußschen Eliminationsverfahrens wird anschließend eine nichttriviale Linearkombinationen von Zeilen der Matrix gesucht, die den Nullvektor ergeben. Anders ausgedrückt: Es wird ein Vektor $\vec{w}$ gesucht, sodass $\vec{w}M = \vec{0}$.

Für obige Matrix M findet sich $\vec{w} = (1, 0, 1)$, was gleichbedeutend mit der Aussage ist, dass die Multiplikation von $q(296)$ mit $q(316)$ zu einer Zahl führt, bei der alle Exponenten, folglich auch alle $p \in F'_B$, geradzahlig oft vorkommen.

Die Probe beweist, dass

$$q(296) \cdot q(316) = (153) \cdot (12393) = (3^2 \cdot 17) \cdot (3^6 \cdot 17) = (3^8 \cdot 17^2) = (3^4 \cdot 17)^2 = y^2$$

eine Quadratzahl ist. Sie ist kongruent zu der Quadratzahl $x^2 = (296 \cdot 316)^2$ modulo N:

$$y^2 = q(296) \cdot q(316) = (3^4 \cdot 17)^2 \equiv (296 \cdot 316)^2 = x^2 \pmod{87463}$$

Mit $x = \sqrt{x^2} = 296 \cdot 316$, $y = \sqrt{y^2} = 3^4 \cdot 17$ und $ggT(x - y, N) = 587$ kann die Faktorisierung von N endlich angegangen werden. Und tatsächlich führt

$$N = 87463 = ggT(x - y, N) \cdot \frac{N}{ggT(x - y, N)} = 587 \cdot \frac{87463}{587} = 587 \cdot 149$$

zu einer nichttrivialen Faktorisierung von N, die zudem aus zwei Primzahlen besteht.

3.3.6. Pseudocode

Algorithm 4 Quadratisches Sieb[1]

Input: N gem. den Bedingungen am Ende von Abschnitt C.4 auf Seite 41

1: $B \leftarrow \sqrt{e^{\sqrt{\ln N \cdot \ln \ln N}}}$ ▷ Siehe Abschnitt 3.3.2.
2: $F_B \leftarrow \{p \in \mathbb{P} \mid p \leq B\} \cup \{-1\}$
3: $M \leftarrow B^2$
4: $I \leftarrow \left[\lceil \sqrt{N} \rceil - \frac{M}{2}, \lceil \sqrt{N} \rceil + \frac{M}{2}\right] \cap \mathbb{N}$
5:
6: $F'_B \leftarrow F_B \backslash \{p \in F_B \mid p > 2 \wedge (N/p) \neq 1\}$ ▷ Siehe Abschnitt 3.3.3.
7: $S \leftarrow \{q(x_i) = x_i^2 - N \mid x_i \in I\}$
8: Multipliziere jede negative Zahl in S mit -1.
9: Siebe S bzgl. B-glatter Zahlen.
10:
11: Baue zu jeder B-glatten Zahl $q(x_i) = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_l^{e_l}$ einen Exponentenvektor $\vec{v}(q(x_i)) = (e_1, e_2, \ldots, e_l)$ auf. ▷ Siehe Abschnitt 3.3.4.
12: Bilde eine Matrix, deren Zeilen die Exponentenvektoren, reduziert modulo 2, sind.
13: Finde eine nichttriviale Linearkombination von Zeilen der Matrix, die den Nullvektor (mod 2) ergeben, bspw. sei $\vec{v}(q(x_0)) + \vec{v}(q(x_1)) + \ldots + \vec{v}(q(x_k)) = \vec{0}$.
14: $x \leftarrow x_0 \cdot x_1 \cdot \ldots \cdot x_k$
15: $y \leftarrow \sqrt{q(x_0) \cdot q(x_1) \cdot \ldots \cdot q(x_k)}$
16:
17: **return** $\left[\, ggT(x - y, N),\ \frac{N}{ggT(x-y,N)} \,\right]$

[1] Pseudocode inspiriert von Algorithmus 6.1.1 aus [4, S. 266]. Basis der Impl. von Listing C.4.

4. Implementierung

In der Informatik geht es genauso wenig um Computer
wie in der Astronomie um Teleskope.
Edsger Wybe Dijkstra

4.1. Das Computeralgebrasystem Sage

Die Entwickler des erstmals 2005 veröffentlichten Computeralgebrasystems Sage haben es sich zum Ziel gesetzt, eine vollwertige Open-Source-Alternative für gängige kommerzielle Mathematik-Software zu erschaffen [11]. Und wenngleich Sage zu großen Teilen in Python geschrieben und vorwiegend auch über Python angesteuert wird, bietet es zusätzlich noch Schnittstellen zu vielen anderen Computeralgebrasystemen.

Zudem kann Sage über eine interaktive Konsole in der Kommandozeile oder eine Weboberfläche im Browser bedient werden. Auch bei der Erzeugung von Grafiken erweist sich Sage als nützlich: Abbildung B.1 auf Seite 35 wurde mit Hilfe von Sage generiert.

In der Implementierung des Quadratischen Siebs, siehe Listing C.4 in Abschnitt C.4, spielt Sage an zwei entscheidenden Stellen eine Rolle:

Die Methode *___sieve_interval*(...) verwendet den in Sage eingebauten Algorithmus zum Finden der Lösungen $\pm x$ der Kongruenz $x^2 \equiv N \pmod{p}$, wobei N – im Quellcode *number* genannt – die zu faktorisierende Zahl ist und p – im Quellcode *prime* genannt – eine Primzahl der Faktorbasis, für die $(N/p) = 1$ gilt.

```
square_roots = mod(number, prime).sqrt(all = True)
```

Mit Hilfe der Lösungen $\pm x$ – im Quellcode *square_roots*[0] und *square_roots*[1] genannt – kann anschließend unter den ersten p Zahlen $q(x_i) = x_i^2 - N$, $x_{min} \leq x_i < x_{min+p}$, nach zwei durch p teilbaren Zahlen gesucht werden. Es sind gerade die Zahlen, für die $x_i \equiv \pm x \pmod{p}$ gilt. Ausgehend von diesen Zahlen ist auch jede p-te Zahl – mindestens einmal – durch p teilbar, siehe Abschnitt 3.3.3 und das Beispiel in Abschnitt 3.3.5.

Die Methode *generate_matrix*(...) verwendet den in Sage eingebauten Algorithmus zum Finden von nichttrivialen Linearkombinationen von Zeilen einer Matrix M über dem endlichen Körper GF(2), die den Nullvektor ergeben.

```
return matrix(GF(2), exponent_vectors).kernel().basis()
```

Das zweidimensionale Array *exponent_vectors* enthält die Exponentenvektoren $\vec{v}(q(x_i))$, siehe Abschnitt 3.3.4, die, reduziert modulo 2, das Python-Äquivalent der zuvor beschriebenen Matrix M darstellen. Anders ausgedrückt, liefert obige Zeile Quellcode eine Menge von Vektoren $\vec{w}$, für die $\vec{w}M = \vec{0}$ gilt. Siehe auch hierzu das Beispiel in Abschnitt 3.3.5.

4.2. Python mit Cython beschleunigen

Zwei der größten Herausforderungen, denen man bei der Entwicklung von Faktorisierungsverfahren und speziell beim Einsatz von Skriptsprachen wie Python stets begegnen wird, sind: zu niedrige Geschwindigkeit und zu hoher Speicherverbrauch[1].

Beiden Herausforderungen kann man in Python durch den Einsatz von Cython[2] entgegenwirken. Ein Compiler, dessen Entwickler es sich zum Ziel gesetzt haben, das Schreiben von C Extensions für Python so einfach wie Python selbst zu machen[3]. Und tatsächlich ist die Verwendung von Cython – zumindest unter Linux – überraschend einfach:

Mit Hilfe einer sogenannten Setup-Datei, siehe z. B. die in Listing C.5 angegebene, kompiliert man eine beliebige Python-Datei, der zuvor lediglich die Endung *pyx* gegeben werden muss, in eine Shared Object Library mit der Endung *so*. Am Beispiel des Quadratischen Siebs, siehe Listing C.4, sähe der Befehl wie folgt aus:

```
python setup.py build_ext --inplace
```

Das Ergebnis ist eine Datei namens *quadratic_sieve.so*, deren Methoden nun beispielsweise in einer Main-Datei, siehe z. B. die in Listing C.6 angegebene, importiert und ganz normal verwendet werden können. Dieser simple Prozess hat beim Quadratischen Sieb bereits eine Beschleunigung der Laufzeit von 25 % und mehr gebracht.

Doch Cython ist nicht nur ein Compiler, sondern versteht sich vielmehr als eigenständige Programmiersprache, die u. a. darauf optimiert wurde, auf C-Funktionen und -Typen in Python-Code zugreifen zu können und so noch effizienteren Code zu erreichen. In Listing C.4 ist dies bereits am Anfang des Quellcodes ersichtlich, beim Importieren zweier Funktionen der C-Standard-Bibliothek:

```
from libc.stdlib cimport free, malloc
```

Diese werden in der Methode *___sieve_interval(...)* verwendet, um ein vergleichsweise riesiges Array – im Quellcode *logarithms* genannt – möglichst platzsparend anzulegen und haben dadurch entscheidend dazu beigetragen, den Speicherverbrauch zu reduzieren und so indirekt zu einer Beschleunigung von mehreren Hundert Prozent geführt.

Was es mit diesem Array auf sich hat, wird in Abschnitt 4.3 erklärt.

[1] Mit *Speicher* ist sowohl der Arbeitsspeicher als auch der Swap-Bereich gemeint.
[2] Nicht zu verwechseln mit dem ursprünglichen, in C geschriebenen Python-Interpreter CPython.
[3] Siehe hierzu u. a.: http://cython.org/ (aufgerufen am 07.01.2014).

Ein weiteres Problem, das hier nicht verschwiegen werden soll, stellt das Iterieren über vergleichsweise große Bereiche dar, wie sie im Quadratischen Sieb immer wieder, vor allem beim Sieben, vorkommen. Da Sage derzeit intern Python 2.x voraussetzt, können die Python-Methoden $range(start, stop, step)$ sowie $xrange(start, stop, step)$ nicht verwendet werden, da ihre Argumente hinsichtlich der Größe beschränkt sind. Stattdessen wird folgender äquivalenter Code verwendet:[1]

```
islice(count(start, step), (stop - start + step - 1 + 2*(step < 0))//step)
```

Die Methoden $islice(\ldots)$ und $count(\ldots)$ müssen zuvor importiert worden sein:

```
from itertools import islice, count
```

4.3. Sieben mit 2er-Logarithmen

Statt der in Abschnitt 3.3.3 vorgestellten Variante des Siebvorgangs wird in der konkreten Implementierung in Listing C.4 eine technisch wesentlich effizientere Alternative genutzt. Das Hauptproblem der zuvor vorgestellten Variante besteht darin, dass diese das Erzeugen einer Menge $S = \{q(x_i) = x_i^2 - N \mid x_i \in I\}$ verlangt, die zu jeder Zahl x_i des Siebintervalls I die zugehörige Zahl $q(x_i)$ enthält.

Ein solcher Vorgang ist bei größer werdenden Zahlen – aufgrund der Quadrierung x_i^2 – nicht nur rechenintensiv, sondern vor allem verbraucht die Menge S leicht mehrere Dutzend Gigabyte Arbeitsspeicher.

Um dem Problem Herr zu werden, nutzt man eine Eigenschaft von Logarithmen aus, mit der sich technisch teure Multiplikation auf Additionen zurückführen lassen, denn

$$\log_2(x_i^2 - N) = \log_2(p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_l^{e_l}) = \log_2(p_1^{e_1}) + \log_2(p_2^{e_2}) + \ldots + \log_2(p_l^{e_l}),$$

wobei mit $p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_l^{e_l}$ die vollständige Faktorisierung von $x_i^2 - N$ gemeint ist.

Anstatt der Menge S wird in Listing C.4 das Array *logarithms* entsprechend der Länge des Siebintervalls – im Quellcode *sieving_array_length* genannt – angelegt, dessen Einträge jeweils nur 2 Byte groß und mit 0 initialisiert werden:

```
logarithms = <unsigned short *> malloc(sizeof(unsigned
    short)*sieving_array_length)
```

[1] Siehe hierzu u. a. den Hinweis zur Implementierung in CPython unter http://docs.python.org/2/library/functions.html#xrange (aufgerufen am 07.01.2014).

Das Siebverfahren beginnt, genau wie in Abschnitt 3.3.3 und 4.1 dargestellt, nun damit, unter den ersten p Zahlen $q(x_i) = x_i^2 - N$, $x_{min} \leq x_i < x_{min+p}$, nach der ersten von zwei durch p teilbaren Zahlen zu suchen. Befindet sich diese bspw. an der Stelle *index*, so wird der Wert des Arrays *logarithms* an ebendieser Stelle um den gerundeten Wert von $\log_2(p)$ erhöht. Ausgehend davon wird auch an jeder Stelle $index + k \cdot p$, $k \in \mathbb{N}\backslash\{0\}$, der Wert im Array *logarithms* um den gerundeten Wert von $\log_2(p)$ erhöht.

Anders ausgedrückt: Jeder zu Anfang noch mit 0 belegte Eintrag des Arrays *logarithms* entspricht stellvertretend einem x_i des Siebintervalls I bzw. einem $q(x_i)$ der Menge S.

Am Ende des Siebvorgangs beinhaltet das Array *logarithms* die ungefähren 2er-Logarithmen der Zahlen $q(x_i) = x_i^2 - N$, die auf B-Glattheit hin untersucht werden sollen. Es handelt sich nicht um die genauen 2er-Logarithmen, da höhere Potenzen von Primzahlen vernachlässigt werden und nach den kleinsten Primzahlen der Faktorbasis ebenfalls nicht gesiebt wird, vgl. hierzu u. a. [4, S. 267].

Um dieser Ungenauigkeit zu begegnen und herauszufinden, welche Zahlen wirklich B-glatt sind, wird zuerst ein sogenannter Schwellwert (engl. *threshold*) definiert. Dieser gibt an, ab welchem Wert eine Zahl als vermutlich B-glatt gilt. Diese Zahlen werden anschließend mit Hilfe einer abgewandelten Form der Probedivision, siehe Abschnitt 3.1, faktorisiert: sie werden nur durch Elemente der Faktorbasis dividiert. Bleibt nach den durchgeführten Divisionen lediglich 1 übrig, so ist eine untersuchte Zahl B-glatt, da sie sich als Produkt von Primzahlen der Faktorbasis, zzgl. der Zahl -1, darstellen lässt.

Für die Berechnung des Schwellwerts hat sich folgende Formel mehr als bewährt:

```
threshold = ceil(log((start_point + ((end_point - start_point)//5)*3)**2 -
    number, 2)) - 30
```

Mit *start_point* ist der kleinste Wert des Siebintervalls, zuvor x_{min} genannt, gemeint, mit *end_point* der entsprechend größte Wert, zuvor x_{max} genannt. Die Differenz beider Werte entspricht folglich der Länge des Siebintervalls. Dieser Wert wird durch 5 dividiert, mit 3 multipliziert und zum *start_point* hinzuaddiert. Vereinfacht gesagt, wird das x_i gewählt, das bei ca. 60 % des Siebintervalls zu finden ist. Für dieses x_i wird $q(x_i)$ berechnet und von diesem werden pauschal 30 subtrahiert, vgl. hierzu u. a. [4, S. 267].

Zusammenfassend entspricht obige Zeile Quellcode folgender Formel:

$$threshold = \left\lceil \log_2 \left(\left(x_{min} + \frac{3}{5} \cdot (x_{max} - x_{min}) \right)^2 - N \right) \right\rceil - 30,$$

wobei N im Quellcode *number* genannt wird.

4.4. Faktorisieren der siebten Fermat-Zahl

Am Beispiel der 39-stelligen siebten Fermat-Zahl[1]

$$F_7 = 2^{2^7} + 1 = 2^{128} + 1 = 340282366920938463463374607431768211457,$$

von der mittlerweile bekannt ist, dass sie aus genau zwei verschiedenen Primfaktoren besteht, soll gezeigt werden, wie das Quadratische Sieb aus Abschnitt C.4, zzgl. dem Legendre-Symbol aus Abschnitt C.1, verwendet werden muss, um F_7 zu faktorisieren.

Unter der Annahme, dass Sage, siehe Abschnitt 4.1, und Cython, siehe Abschnitt 4.2, korrekt installiert sind und die Dateien *quadratic_sieve.pyx*, *setup.py*, *main.py* und *legendre_symbol.py* alle im selben Ordner liegen, der das aktuelle Arbeitsverzeichnis darstellt, so kann in der Kommandozeile folgender Befehl eingegeben werden:

```
python setup.py build_ext --inplace
```

Die Ausgabe sollte in etwa wie folgt lauten:

```
Compiling quadratic_sieve.pyx because it changed.
Cythonizing quadratic_sieve.pyx
running build_ext
building 'quadratic_sieve' extension
creating build
creating build/temp.linux-x86_64-2.7
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall
    -Wstrict-prototypes -fPIC -I/usr/include/python2.7 -c quadratic_sieve.c
    -o build/temp.linux-x86_64-2.7/quadratic_sieve.o
gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions
    -Wl,-Bsymbolic-functions -Wl,-z,relro
    build/temp.linux-x86_64-2.7/quadratic_sieve.o -o
    /home/qs/beispiel/quadratic_sieve.so
```

Anschließend kann die Faktorisierung von F_7 gestartet werden:

```
sage main.py 340282366920938463463374607431768211457 1
```

Der zweite Parameter bestimmt, in wie viele Teilintervalle das Siebintervall zerlegt werden soll, siehe hierzu u. a. Abschnitt B.1. Die Ausgabe sollte wie folgt lauten:

```
[5704689200685129054721, 59649589127497217]
```

Die Probe beweist, dass $F_7 = 5704689200685129054721 \cdot 59649589127497217$.

[1] Siehe hierzu u. a.: http://mathworld.wolfram.com/FermatNumber.html (aufgerufen am 10.01.2014).

4.5. Mögliche Optimierungen und Ausblick

Wenngleich das in Listing C.4 implementierte Quadratische Sieb an vielen Stellen bereits optimiert wurde und fast alle der vorgeschlagenen Verbesserungen in [4, S. 266–268] umsetzt, so gibt es dennoch noch immer Möglichkeiten, die Geschwindigkeit weiter zu steigern bzw. den Speicherverbrauch weiter zu reduzieren.

Die vermutlich unkomplizierteste bestünde darin, die Anmerkung (5) in [4, S. 268] umzusetzen. Diese empfiehlt, mit dem Sammeln von B-glatten Zahlen aufzuhören, sobald man $|\,F'_B\,| + 10$ solcher Zahlen gefunden hat, wobei mit $|\,F'_B\,|$ die Mächtigkeit von F'_B, also die Anzahl der Elemente von F'_B, gemeint ist. Denn dann hat die in Abschnitt 3.3.4 eingeführte Matrix mehr Zeilen als Spalten – nämlich $|\,F'_B\,|+10$ Zeilen und $|\,F'_B\,|$ Spalten – und folglich gibt es mindestens 10 verschiedene nichttriviale Linearkombinationen von Zeilen dieser Matrix, die den Nullvektor ergeben.

Unter der Annahme, dass nur jede zweite solche Linearkombination zu einer nichttrivialen Faktorisierung von N führt, besteht eine Chance von über 99,9 %, dass die Faktorisierung gelingt. Denn nur in einem von 1024 Fällen werden 10 Faktorisierungsversuche zu einer trivialen Faktorisierung $N = N \cdot 1$ bzw. $N = 1 \cdot N$ führen.

Hat man diese Verbesserung erst einmal umgesetzt, empfiehlt sich die Implementierung eines weiteren Details, das jedoch erst dann von Bedeutung ist, wenn größere Zahlen faktorisiert werden sollen, bei denen das Siebintervall in mehrere Teilintervalle zerlegt werden muss, siehe hierzu u. a. Abschnitt B.1:

Angenommen das Siebintervall I wird in 11 Teilintervalle zerlegt, die mit $I_0, I_1, \ldots, I_{10}$ bezeichnet werden. Die Zahlen $|\,q(x)\,| = |\,x^2 - N\,|$ sind dann am kleinsten, wenn x in der Nähe von $\sqrt{N}$ liegt und folglich ist dort die Wahrscheinlichkeit am größten, auf B-glatte Zahlen zu stoßen. Anders ausgedrückt, ist die Wahrscheinlichkeit, auf B-glatte Zahlen zu stoßen, in der Mitte des Siebintervalls am größten und an den Rändern am geringsten.

Hat man gar nicht vor, das gesamte Siebintervall I zu durchlaufen, sondern stattdessen aufzuhören, wenn bspw. wie oben empfohlen $|\,F'_B\,| + 10$ B-glatte Zahlen gefunden wurden, dann empfiehlt sich in folgender Abfolge zu Sieben: $I_5, I_6, I_4, I_7, I_3, I_8, I_2, I_9, I_1, I_{10}$ und schließlich I_0. Man siebt gewissermaßen von der Mitte zu den Rändern.

Neben diesen recht simplen Optimierungen, gibt es noch weitere Ansätze, denen in [4] aufgrund ihrer Komplexität sogar eigene Abschnitte gewidmet wurden. Sie sind zumeist aus der Motivation heraus entstanden, große Zahlen – Zahlen mit 100 Dezimalstellen und mehr – noch schneller faktorisieren zu können. Zwei der wahrscheinlich populärsten Resultate dieser Bemühungen sind das Multipolynomiale Quadratische Sieb (engl. *Multiple Polynomial Quadratic Sieve*, MPQS) und das Selbstinitialisierende Quadratische Sieb (engl. *Self-Initializing Quadratic Sieve*, SIQS), vgl. hierzu u. a. [4, S. 273–276].

Beiden Verfahren ist gemein, dass Sie den zeitintensiven Siebschritt beschleunigen, indem Sie nicht mit einem einzigen Polynom $q(x) = x^2 - N$ nach B-glatten Zahlen sieben, da dessen absolute Werte relativ schnell wachsen, wenn sich x von $\sqrt{N}$ entfernt und folglich die Wahrscheinlichkeit, auf B-glatte Zahlen zu stoßen, relativ schnell sinkt.

Stattdessen verwenden sowohl das MPQS als auch das SIQS wechselnde Polynome $f(x) = ax^2 + 2bx + c$, mit $a, b, c \in \mathbb{Z}$ und $b^2 - ac = N$ sowie weiteren – hier nicht näher spezifizierten – Kriterien. Mit diesen kann die Suche nach B-glatten Zahlen

$$\begin{aligned} Q(x) = a \cdot f(x) &= a^2x^2 + 2abx + ac \\ &= a^2x^2 + 2abx + b^2 - b^2 + ac \\ &= (ax + b)^2 - (b^2 - ac) \\ &= (ax + b)^2 - N, \end{aligned}$$

deren Exponentenvektoren, reduziert modulo 2, die Zeilen der bereits bekannten Matrix ergeben, in mehreren – deutlich kleineren – Siebintervallen durchgeführt werden. Gelingt es schließlich durch Kombination mehrerer $Q(x_i)$, $0 \leq i \leq k$, eine Quadratzahl

$$\begin{aligned} y^2 &= Q(x_0) \cdot Q(x_1) \cdot \ldots \cdot Q(x_k) \\ &= ((ax_0 + b)^2 - N) \cdot ((ax_1 + b)^2 - N) \cdot \ldots \cdot ((ax_k + b)^2 - N) \end{aligned}$$

zu erzielen, so folgt wie bisher, siehe Abschnitt 3.3.1, dass

$$\begin{aligned} y^2 &\equiv (ax_0 + b)^2 \cdot (ax_1 + b)^2 \cdot \ldots \cdot (ax_k + b)^2 \\ &= ((ax_0 + b) \cdot (ax_1 + b) \cdot \ldots \cdot (ax_k + b))^2 \\ &= x^2 \pmod{N}, \end{aligned}$$

da $(ax_i + b)^2 - N \equiv (ax_i + b)^2 \pmod{N}$ gilt. Die gefundenen Quadratzahlen, x^2 und y^2, können anschließend – wie gewohnt – zur Faktorisierung von N herangezogen werden.

Der Unterschied zwischen dem MPQS und dem SIQS besteht im Wesentlichen in der Wahl der Polynome $f(x) = ax^2 + 2bx + c$ bzw. genauer gesagt in der Wahl der Koeffizienten a und b. Während das MPQS zu jedem a nur ein einziges b generiert, kann das SIQS durch geschicktere Wahl von a gleich eine ganze Reihe von Koeffizienten b generieren.

In Tests hat sich gezeigt, dass das SIQS etwa doppelt so schnell wie das MPQS ist, vgl. hierzu u. a. [4, S. 276], und dieses ist wiederum etwa 17-mal schneller als das Quadratische Sieb, vgl. hierzu u. a. [4, S. 274].

A. Notationen

$\mathbb{N}$	Menge der natürlichen Zahlen[1], $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$
$\mathbb{Z}$	Menge der ganzen Zahlen, $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{Q}$	Menge der rationalen Zahlen, $\mathbb{Q} = \left\{\frac{v}{w} \mid v, w \in \mathbb{Z}, w \neq 0\right\}$
$\mathbb{I}$	Menge der irrationalen Zahlen[2], $\mathbb{I} = \left\{\sqrt{2}, \pi, e, \ldots\right\}$
$\mathbb{R}$	Menge der reellen Zahlen, $\mathbb{R} = \mathbb{Q} \cup \mathbb{I}$
$\mathbb{P}$	Menge der Primzahlen, $\mathbb{P} = \{2, 3, 5, 7, 11, 13, \ldots\}$
$\mathbb{U}$	Menge der ungeraden natürlichen Zahlen, $\mathbb{U} = \{2u + 1 \mid u \in \mathbb{N}\}$
$p \mid N$	p teilt N, für $p \in \mathbb{P}$, $N \in \mathbb{N} \Leftrightarrow \exists k \in \mathbb{N}$, mit $N = k \cdot p$
$p \nmid N$	p teilt N nicht, für $p \in \mathbb{P}$, $N \in \mathbb{N} \Leftrightarrow \forall k \in \mathbb{N}$ gilt, dass $N \neq k \cdot p$
$ggT(a, m)$	größter gemeinsamer Teiler von $a \in \mathbb{Z}$ und $m \in \mathbb{Z}$, $ggT(a, m) = 1 \Leftrightarrow a$ und m sind teilerfremd
$\lceil x \rceil$	$min\{y \in \mathbb{Z} \mid y \geq x\}$, $x \in \mathbb{R}$, *Aufrundungsfunktion*, die kleinste ganze Zahl, die größer oder gleich x ist
$\left(\frac{a}{p}\right)$	Legendre-Symbol, siehe Abschnitt 2.2
$\mathcal{O}(f(n))$	$\mathcal{O}$-Notation, siehe Abschnitt 2.3
$L_N[\alpha, c]$	L-Notation, siehe Abschnitt 2.3

[1] $\mathbb{N}\backslash\{0\}$ zur Bezeichnung der Menge $\{1, 2, 3, 4, \ldots\}$ wird dem ebenfalls gebräuchlichen $\mathbb{N}^+$ vorgezogen.
[2] Können nicht als Bruch zweier ganzer Zahlen geschrieben werden und liegen somit nicht in $\mathbb{Q}$.

B. Faktorisierungen

70 Bit $109234354402315079139 7$
$= 32159416403 \cdot 33966522599$

80 Bit $1064616026292559296927751$
$= 973424573159 \cdot 1093681067489$

90 Bit $109280620359720027260858826 1$
$= 32046941312363 \cdot 34100171774447$

100 Bit $1038973066102443270031889058283$
$= 1112182146324839 \cdot 934175278335197$

110 Bit $1057368393145709609006000462102149$
$= 34257860832366203 \cdot 30864985946429183$

120 Bit $1080330230446510453959823665631751821$
$= 1045905287863818077 \cdot 1032914015238418673$

130 Bit $1079637023032048942303551249978418657051$
$= 31227030460885908653 \cdot 34573797351124104167$

140 Bit $1173323401440631813160813484326414060235097$
$= 1013634039263677081343 \cdot 1157541436052163463079$

150 Bit $1159768057578331066452367002954755784527754931$
$= 34528234757524866757163 \cdot 33588976260235212157337$

160 Bit $1044341195212758178340370130102256269119310650967$
$= 1131792207820030095999809 \cdot 92273227187549449600706 3$

170 Bit $1399159606087137191515478928115062264042721609866649$
$= 37715737564197988055795297 \cdot 37097500843130862162997817$

180 Bit $1419874410588526742774833603035744807851772612829340839$
$= 1149329362938318035129659937 \cdot 12353938360702338220626774 47$

190 Bit $1189865861132745902886240607095837205372038705589570926697$
$= 37318710200313916971719626499 \cdot 31883895631600285235017152803$

200 Bit $1092157627960920741470003802658274049265843660359829948673257$
$= 1089937882485683744951873593883 \cdot 1002036579800469640721914805579$

B.1. Laufzeituntersuchung

Die auf Seite 32 angegebenen Faktorisierungen wurden alle mit Hilfe des Quadratischen Siebs aus Abschnitt C.4 auf einer einzigen CPU mit 2,80 GHz Taktfrequenz durchgeführt. Der Computer selbst verfügte über Arbeitsspeicher in Höhe von ca. 24 GB und einem Swap-Bereich von ebenfalls ca. 24 GB.

Die nachfolgende Tabelle listet die jeweils zur Faktorisierung benötigte Zeit auf, wobei die Spalten *Zeit in min* und *Zeit in h* recht grob gerundete Werte enthalten, die sich aus der Spalte *Zeit in s* ableiten, die der offiziellen Messung in Sekunden entspricht.

Bits	**Dezimalstellen**	**Siebintervalle**[1]	**Zeit in s**	**Zeit in min**	**Zeit in h**
70	22	1	2	–	–
80	25	1	5	–	–
90	28	1	10	–	–
100	31	1	19	–	–
110	34	1	42	–	–
120	37	1	121	2	–
130	40	1	308	5	–
140	43	1	681	11	–
150	46	1	1442	24	–
160	49	1	3485	58	1
170	52	2	9064	151	2
180	55	4	24981	416	7
190	58	10	79859	1331	22
200	61	22	265219	4420	74

Anhand der obigen Tabelle und mit Hilfe des Computeralgebrasystems Sage, siehe Abschnitt 4.1, kann schließlich eine Laufzeituntersuchung des in Listing C.4 implementierten Quadratischen Siebs mit Hilfe der in Abschnitt 2.3 eingeführten L-Notation durchgeführt werden. Diese wird zusätzlich noch mit einem Vorfaktor $v \in \mathbb{R}$ multipliziert, um dem zur Faktorisierung eingesetzten Computer Rechnung zu tragen. Ferner kann α im Falle des Quadratischen Siebs auf $1/2$ festgelegt werden:

$$v \cdot L_N[1/2, c] = v \cdot e^{(c+o(1))(\ln N)^{1/2}(\ln \ln N)^{1/2}} \tag{B.1}$$

[1] Um das zeitintensive Auslagern von Daten in den Swap-Bereich zu verhindern, bietet das Quadratische Sieb aus Abschnitt C.4 die Möglichkeit, Arbeitsspeicher zu sparen, indem das Siebintervall, siehe Abschnitt 3.3.2, in beliebig kleine Teilintervalle zerlegt wird.

Statt der im Computeralgebrasystem Sage eingebauten Methode $find\_fit(data, model)$[1] die in (B.1) angegebene Funktion als Modell zu übergeben, sollte diese zuvor logarithmiert werden, um die Exponentialfunktion verschwinden zu lassen, denn

$$\begin{aligned}\ln\left(v \cdot e^{(c+o(1))(\ln N)^{1/2}(\ln\ln N)^{1/2}}\right) &= \ln v + \ln e^{(c+o(1))(\ln N)^{1/2}(\ln\ln N)^{1/2}} \\ &= \ln v + (c+o(1))(\ln N)^{1/2}(\ln\ln N)^{1/2}.\end{aligned}$$

Außerdem kann der $o(1)$-Term vernachlässigt werden, wenn $N \to \infty$, denn dann ist

$$\ln\left(v \cdot L_N[1/2, c]\right) \approx \ln v + c \cdot (\ln N)^{1/2} \cdot (\ln\ln N)^{1/2} = \ln v + c \cdot \sqrt{\ln N \cdot \ln\ln N}. \quad \text{(B.2)}$$

Um die besten Näherungen für c und v in (B.2) zu bestimmen, bedarf es nur noch eines zweidimensionalen Arrays mit Daten, welches $find\_fit(data, model)$ als ersten Parameter erwartet. Die Daten sind im Falle des Quadratischen Siebs der Wert $\ln N$ der zu faktorisierenden Zahl N und die ebenfalls logarithmierte Zeit in Minuten, die die Faktorisierung benötigte. Durch einen geschickten Basiswechsel kann $\ln N$ mit Hilfe der Bitlänge n der zu faktorisierenden Zahl N approximiert werden, denn

$$\ln N = \log_e N = \frac{\log_2 N}{\log_2 e} = \frac{1}{\log_2 e} \cdot \log_2 N = \log_e 2 \cdot \log_2 N = \ln 2 \cdot \log_2 N \approx \ln 2 \cdot n.$$

Vernachlässigt man Zahlen mit weniger als 120 Bits, da diese nicht einmal eine Minute zur Faktorisierung benötigten, ergibt sich folgender im Sage-Notebook ausführbarer Code:

```
1  data = [[ln(2)*120, ln(2)],
2          [ln(2)*130, ln(5)],
3          [ln(2)*140, ln(11)],
4          [ln(2)*150, ln(24)],
5          [ln(2)*160, ln(58)],
6          [ln(2)*170, ln(151)],
7          [ln(2)*180, ln(416)],
8          [ln(2)*190, ln(1331)],
9          [ln(2)*200, ln(4420)]]
10
11 var('c v')
12 model(x) = ln(v) + c*sqrt(x*ln(x))
13
14 print find_fit(data, model)
```

Das Ergebnis: $c \approx 1,08$ und $v \approx 1,44 \cdot 10^{-9}$. Verwendet man diese Werte, um einen Graph der Funktion

$$v \cdot L_N[1/2, c] \approx v \cdot e^{c \cdot (\ln N)^{1/2} \cdot (\ln\ln N)^{1/2}} = v \cdot e^{c \cdot \sqrt{\ln N \cdot \ln\ln N}}$$

zu zeichnen, führt dies zu einer plausiblen Grafik, die das subexponentielle Wachstum des Quadratischen Siebs anschaulich verdeutlicht – siehe Abbildung B.1 auf Seite 35.

[1] Siehe hierzu u. a.: http://www.sagemath.org/doc/reference/numerical/sage/numerical/optimize.html#sage.numerical.optimize.find_fit (aufgerufen am 15.12.2013).

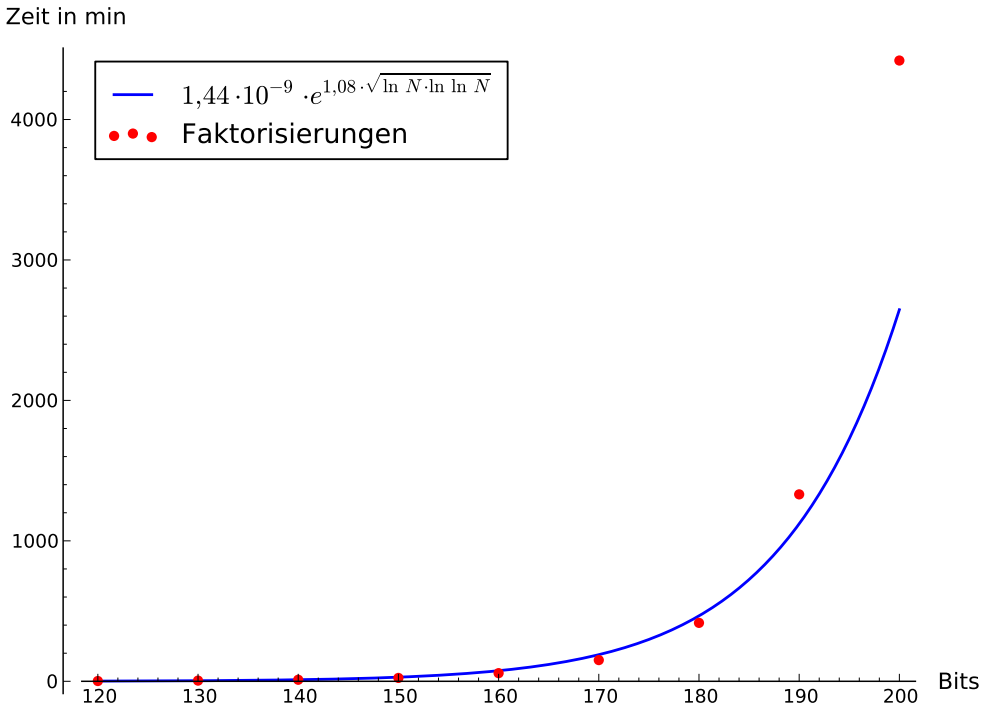


Abbildung B.1.: Laufzeituntersuchung des Quadratischen Siebs.

Was an obiger Abbildung sofort ins Auge fällt, ist die starke Abweichung des letzten Messwerts, welcher die Faktorisierung der auf Seite 32 aufgeführten 200-Bit-Zahl repräsentiert. Über 4000 Minuten dauerte diese, wenngleich die Näherung weniger als 3000 Minuten erwarten ließ. Eine deutliche Abweichung von mehr als 30 %, deren Ursache vermutlich nicht nur in einer Ungenauigkeit der Näherung zu finden ist. Vielmehr haben Tests mit kleinen wie großen Zahlen gezeigt, dass die Zunahme an Teilintervallen, also die Aufteilung des Siebintervalls in immer mehr bzw. kleinere Teilintervalle, um Arbeitsspeicher zu sparen, die Laufzeit spürbar beeinflussen kann.

Es lässt sich jedoch nicht pauschal sagen, wie viele Teilintervalle angemessen sind, da dies von verschiedenen Faktoren abhängt, z. B. der Größe der zu faktorisierenden Zahl, der Menge des zur Verfügung stehenden Arbeitsspeichers und der Größe bzw. Geschwindigkeit des Swap-Bereichs. Was sich jedoch sagen lässt, ist, dass es sich bei den durchgeführten Faktorisierungen als positiv erwies, den Arbeitsspeicher fast bis an seine Grenzen auszureizen, um das Auslagern von Daten in den Swap-Bereich zu vermeiden und möglichst große Bereiche nach B-glatten Zahlen durchsuchen zu können. Als Daumenregel:

Zerlege das Siebintervall in so viele Teilintervalle wie nötig, aber so wenige wie möglich.

C. Quellcode

Zum Testen komplexer Faktorisierungsverfahren, wie dem in Listing C.4 implementierten Quadratischen Sieb, sollten bevorzugt sogenannte RSA-Zahlen $N = p \cdot q$ verwendet werden, die das Produkt genau zweier Primzahlen $p, q \in \mathbb{P}\backslash\{2\}$, $p \neq q$, sind. Eine RSA-Zahl mit 100 Bit lässt sich auf der Kommandozeile beispielsweise wie folgt erzeugen:

```
openssl genrsa 100 | openssl rsa -modulus -noout
```

Die letzte Zeile der Ausgabe enthält die gewünschte RSA-Zahl – jedoch zur Basis 16. Mit Hilfe eines einfachen Tricks kann diese zur Basis 10 konvertiert werden:

```
echo 'ibase=16;B024C9F788A6F04F4D73067F3' | bc
```

C.1. Legendre-Symbol

Listing C.1: legendre_symbol.py

```
1  #!/usr/bin/env python
2
3  def legendre_symbol(a, p):
4
5      a = a%p
6      t = 1
7
8      while a != 0:
9          while a%2 == 0:
10             a = a/2
11             if p%8 in (3, 5): t = -t
12         a, p = p, a
13         if a%4 == p%4 == 3: t = -t
14         a = a%p
15
16     if p == 1:
17         return t
18     return 0
```

Listing C.1 implementiert Algorithmus 2.3.5 aus [4, S. 98] für $a \in \mathbb{Z}$ und $p \in \mathbb{P}\backslash\{2\}$ (Legendre-Symbol) bzw. $p \in \mathbb{U}\backslash\mathbb{P}$ (Jacobi-Symbol).

C.2. Probedivision

Listing C.2: trial_division.py

```
#!/usr/bin/env python

def trial_division(N):

    F = []

    while N%2 == 0:
        N = N/2
        F.append(2)

    d = 3
    while d*d <= N:
        while N%d == 0:
            N = N/d
            F.append(d)
        d = d+2

    if N != 1:
        F.append(N)

    return F
```

Listing C.2 implementiert Algorithmus 3.1.1 aus [4, S. 118–119] für $N \in \mathbb{N}\backslash\{0,1\}$.

C.3. Fermats Algorithmus

Listing C.3: fermats_method.py

```
#!/usr/bin/env sage

from sage.all import *

def fermats_method(N):

    x = ceil(sqrt(N))
    y_squared = x*x - N

    while not sqrt(y_squared) in ZZ:
        y_squared = y_squared + 2*x + 1
        x = x + 1

    return [x + sqrt(y_squared), x - sqrt(y_squared)]
```

Listing C.3 erwartet ein zusammengesetztes $N = a \cdot b$, wobei $N, a, b \in \mathbb{U}$ sein müssen.

C.4. Quadratisches Sieb

Listing C.4: quadratic_sieve.pyx

```
from sage.all import *
from itertools import islice, count
from libc.stdlib cimport free, malloc
from legendre_symbol import legendre_symbol

# Zu Anfang wird die Grenze B des Quadratischen Siebs berechnet,
# welche die Groesse der Faktorbasis und die Laenge M = B*B des
# Siebintervalls I bestimmt.
def calculate_boundary(number):

    return ceil(sqrt(e**(sqrt(ln(number)*ln(ln(number))))))

# Anschliessend wird die Faktorbasis erzeugt, welcher die Zahlen
# -1 und 2 stets angehoeren. Mit Hilfe des Legendre-Symbols werden
# zudem nur jene Primzahlen in die Faktorbasis aufgenommen, die
# im spaeteren Siebvorgang auch tatsaechlich als Teiler einer Zahl
# q(x) = x*x - number auftreten koennen.
def generate_factor_base(number, boundary):

    factor_base = [-1, 2]

    for prime in prime_range(3, boundary):
        if legendre_symbol(number, prime) == 1:
            factor_base.append(prime)

        # Statt des Legendre-Symbols kann auch das Euler-
        # Kriterium - number**((prime - 1)//2)%prime -
        # herangezogen werden. Die Berechnung wuerde dann
        # jedoch wesentlich laenger dauern.

    return factor_base

# Das Siebintervall I wird nun hinsichtlich B-glatter Zahlen
# q(x) gesiebt. Bei Bedarf kann das Siebintervall I in Teil-
# intervalle zerlegt werden, um den Speicherbedarf - bei ueber-
# schaubarem Leistungsverlust - zu reduzieren.
def sieve(number, boundary, factor_base, int number_of_intervals = 1):

    start_point = ceil(sqrt(number)) - (boundary**2)//2
    end_point = start_point + 2*(boundary**2)//2
    # Der Schwellwert legt fest, ab welchem Wert eine Zahl q(x),
    # deren ungefaehrer 2er-Logarithmus im Siebvorgang bestimmt
    # wird, als B-glatt vermutet und mit Probedivision ueberprueft
    # werden soll.
    threshold = ceil(log((start_point + ((end_point - start_point)//5)*3)**2
        - number, 2)) - 30

    sieved_array = []
    interval_length = (end_point - start_point)//number_of_intervals

```

```
50      end_point = start_point - 1
51      for interval in islice(count(0), number_of_intervals):
52          start_point = end_point + 1
53          end_point = start_point + interval_length
54          sieved_array.append(__sieve_interval(number, factor_base,
                start_point, end_point, threshold))
55
56      # Flachklopfen des aus - moeglicherweise mehreren - Teil-
57      # intervallen des Siebintervalls I bestehenden Arrays.
58      return reduce(lambda x, y: x + y, sieved_array)
59
60  def __sieve_interval(number, factor_base, start_point, end_point, threshold):
61
62      # Zur Reduzierung des Speicherbedarfs wird festgelegt,
63      # dass der ungefaehre 2er-Logarithmus einer Zahl q(x)
64      # max. 2 Byte verbrauchen darf.
65      cdef short logarithm
66
67      sieving_array_length = end_point - start_point + 1
68      logarithms = <unsigned short *> malloc(sizeof(unsigned
            short)*sieving_array_length)
69
70      # Um die Geschwindigkeit deutlich zu steigern, werden die
71      # kleinsten Primzahlen der Faktorbasis vernachlaessigt. Zum
72      # Ausgleich wurde der zuvor festgelegte Schwellwert reduziert.
73      for prime in factor_base[9:]:
74          # Bestimmen der Loesungen +x und -x der quadratischen
75          # Kongruenz x^2 = number (mod prime), um zu erfahren,
76          # welche Zahlen q(x) ein Vielfaches von prime sind.
77          square_roots = mod(number, prime).sqrt(all = True)
78
79          found = 0
80          logarithm = round(log(prime, 2))
81          # Statt den 2er-Logarithmus jeder einzelnen Zahl q(x) zu
82          # berechnen, werden nur die ersten 2 Zahlen q(x) gesucht,
83          # die ein Vielfaches von prime sind, denn dann sind auch
84          # alle Zahlen q(x + k*prime), k ist eine natuerliche Zahl
85          # groesser 0, ein Vielfaches von prime.
86          for index in islice(count(0), prime):
87
88              if (start_point + index)%prime == square_roots[0]:
89                  for multiple in islice(count(index, prime),
                        (sieving_array_length - index + prime - 1)//prime):
90                      logarithms[multiple] += logarithm
91                  found += 1
92
93              if (start_point + index)%prime == square_roots[1]:
94                  for multiple in islice(count(index, prime),
                        (sieving_array_length - index + prime - 1)//prime):
95                      logarithms[multiple] += logarithm
96                  found += 1
97
98              if found == 2: break
99
```

```
100     sieved_array = []
101     for index in islice(count(0), sieving_array_length):
102         # Anhand des Schwellwerts wird nun entschieden, welche
103         # Zahlen q(x) vermutlich B-glatt sind. Diese Vermutung
104         # wird anschliessend mittels Probedivision ueberprueft
105         # und die Faktorisierung bei Erfolg zurueckgegeben.
106         if logarithms[index] >= threshold:
107             composite = (start_point + index)**2 - number
108             factorization = []
109             if composite < 0:
110                 composite *= -1
111                 factorization.append(-1)
112             for prime in factor_base[1:]:
113                 while composite%prime == 0:
114                     composite /= prime
115                     factorization.append(prime)
116             if composite == 1:
117                 sieved_array.append([start_point + index, factorization])
118
119     free(logarithms)
120
121     return sieved_array
122
123 # Zu jeder B-glatten Zahl wird ein Exponentenvektor angelegt,
124 # der angibt, wie oft eine Primzahl der Faktorbasis in einer
125 # Zahl q(x) enthalten ist.
126 def generate_exponent_vectors(factor_base, sieved_array):
127
128     exponent_vectors = [[0 for index in islice(count(0), len(factor_base))]
            for index in islice(count(0), len(sieved_array))]
129
130     for index in islice(count(0), len(sieved_array)):
131         for factor in sieved_array[index][1]:
132             exponent_vectors[index][factor_base.index(factor)] += 1
133
134     return exponent_vectors
135
136 # Die Exponentenvektoren der B-glatten Zahlen werden mod 2
137 # reduziert und der Kernel der aus ihnen gebildeten Matrix
138 # bestimmt, also all jene Vektoren, die, multipliziert mit
139 # der Matrix, den Nullvektor ergeben. Anders ausgedrueckt:
140 # Es werden die Kombinationen von Zahlen q(x) gesucht, die
141 # miteinander multipliziert eine Quadratzahl sind.
142 def generate_matrix(exponent_vectors):
143
144     return matrix(GF(2), exponent_vectors).kernel().basis()
145
146 # Zum Schluss wird die Zahl number in zwei Faktoren zerlegt,
147 # die, sollten sie keine Primzahlen sein, wiederum in das
148 # Quadratische Sieb oder ein anderes Faktorisierungsver-
149 # fahren eingegeben werden koennen.
150 def factorize(number, sieved_array, matrix):
151
152     for row in matrix:
```

```
153             x = []
154             y = []
155             for index in islice(count(0), len(row)):
156                 if row[index] == 1:
157                     x.append(sieved_array[index][0])
158                     y.append(sieved_array[index][1])
159             # Da die Faktorisierung der miteinander multiplizierten
160             # Zahlen q(x) bekannt ist und jeder Faktor geradzahlig oft
161             # vorkommt, kann die Quadratwurzel effizient bestimmt werden:
162             # jeder zweite Faktor wird verworfen und die verbliebenen
163             # Faktoren miteinander multipliziert.
164             y = sorted(reduce(lambda x, y: x + y, y))[::2]
165             y = reduce(lambda i, j: i*j, y)
166             x = reduce(lambda i, j: i*j, x)
167             factor = gcd((x%number) - (y%number), number)
168             if (factor != 1) and (factor != number):
169                 return [factor, number/factor]
170 
171     return []
```

Listing C.5: setup.py

```
1  #!/usr/bin/env python
2
3  from distutils.core import setup
4  from Cython.Build import cythonize
5
6  setup(ext_modules = cythonize('quadratic_sieve.pyx'))
```

Listing C.6: main.py

```
1  #!/usr/bin/env sage
2
3  from quadratic_sieve import *
4
5  number = Integer(sys.argv[1])
6  boundary = calculate_boundary(number)
7  factor_base = generate_factor_base(number, boundary)
8  sieved_array = sieve(number, boundary, factor_base, int(sys.argv[2]))
9  exponent_vectors = generate_exponent_vectors(factor_base, sieved_array)
10 matrix = generate_matrix(exponent_vectors)
11
12 print factorize(number, sieved_array, matrix)
```

Listing C.6 erwartet ein zusammengesetztes $N = a \cdot b$, mit $N, a, b \in \mathbb{U}$. Ferner darf N keine perfekte Potenz sein, siehe Abschnitt 3.3.1, und keine Faktoren aus der Faktorbasis F'_B – im Quellcode *factor_base* genannt – enthalten, siehe Abschnitt 3.3.2 und 3.3.3.

Stabile Ergebnisse lieferten obige Listings bei RSA-Zahlen mit 22 bis 61 Dezimalstellen, siehe Anhang B. Die Verwendung der obigen Listings wird in Abschnitt 4.4 verdeutlicht.

Literaturverzeichnis

[1] Scheid, H. *Zahlentheorie*. BI-Wiss.-Verl., 2nd edition, 1994. ISBN: 3-411-14842-X.

[2] Weisstein, E. W. "Euclid's Theorems." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/EuclidsTheorems.html (aufgerufen am 19.11.2013).

[3] ProofWiki. http://www.proofwiki.org/wiki/ProofWiki:Books/Euclid/The_Elements (aufgerufen am 01.12.2013).

[4] Crandall, R.; Pomerance, C. *Prime Numbers: A Computational Perspective*. Springer Science+Business Media, Inc., 2nd edition, 2005. ISBN: 978-0387-25282-7.

[5] Weisstein, E. W. "Quadratic Reciprocity Theorem." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/QuadraticReciprocityTheorem.html (aufgerufen am 06.11.2013).

[6] Schönberger, K. „Faktorisierung großer Zahlen". Von Matroids Matheplanet. http://matheplanet.com/matheplanet/nuke/html/article.php?sid=1163 (aufgerufen am 25.11.2013).

[7] Pomerance, C. Analysis and comparison of some integer factoring algorithms. *Computational methods in number theory, Part I*, volume 154 of *Mathematical Centre Tracts*:89–139, 1982.

[8] Pomerance, C. A Tale of Two Sieves. *Notices of the AMS*, 43(12):1473–1485, 1996.

[9] *Cryptography and Computational Number Theory*. Birkhäuser Verlag, 2001. ISBN: 3-7643-6510-2, http://books.google.de/books?id=HNrQB4dvRvMC (aufgerufen am 17.12.2013).

[10] Cohen, H. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, 1993. ISBN: 3-540-55640-0, http://books.google.de/books?id=hXGr-9l1DXcC (aufgerufen am 17.12.2013).

[11] Stein, W. A. et al. *Sage Mathematics Software (Version 5.12)*. The Sage Development Team, 2013. http://www.sagemath.org/ (aufgerufen am 07.01.2014).