UNIKASSEL
VERSITÄT

Bachelorarbeit

zum Thema

Algorithmen der Computeralgebra mit Maxima

Autor:

Christian Ewald

Student Bachelor Mathematik

Mnr.: 32210046

Betreuer:

Prof. Dr. Wolfram Koepf

Universität Kassel

FB-10 Mathematik
und Naturwissenschaften

Inhaltsverzeichnis

1 Einleitung

In dieser Bachelorarbeit soll das Computeralgebrasystem (*CAS*) *Maxima* anhand von Algorithmen aus der Computeralgebra vorgestellt werden. Dabei werden insgesamt zwei Schwerpunkte bedient. Einerseits die fachliche Komponente, die wichtige Definitionen und Sätze aus der Algebra und Computeralgebra widerspiegelt und deren Beweise oftmals bereits den Prototyp für viele der behandelten Algorithmen liefern. Andererseits die anwendungsbezogene Komponente, in der die Algorithmen in *Maxima* umgesetzt werden und damit gleichermaßen als Einführung in die Funktionalität und Syntax von *Maxima* dienen. Zusätzlich wird im Zuge dessen auch Codeanalyse betrieben und es werden Laufzeitvergleiche gemacht, um sukzessive effizientere Algorithmen zu generieren.
Da die Algebra hinter den zu Grunde liegenden Problemstellungen der Algorithmen oft sehr weitreichend ist und eine Reihe an Definitionen, Lemmata und Sätzen mit sich bringt, werden gewisse Kenntnisse über Begriffe und Definitionen als gegeben vorausgesetzt. Ferner werden viele der Lemmata und Sätze ohne Beweis als korrekt angenommen, um im Rahmen dieser Facharbeit möglichst viel Bezug auf die Arbeit mit *Maxima* nehmen zu können.
Weiterhin werden aufgrund des breit gefächerten Themengebiets einige Teilgebiete der Algebra stärker behandelt als andere. Insbesondere wird ein Fokus auf die Zahlentheorie und Kryptographie gelegt, wobei die Polynomarithmetik nur exemplarisch angerissen wird.

Zur Struktur:

- Enthält ein Abschnitt einen Block, der mit **Definition u. Satz** gekennzeichnet ist, so ist der „**Satz**“-Teil des Blocks jeweils in *Kursivschrift* geschrieben.

- Textabschnitte, die direkt auf Definition, Lemmata, etc. folgen, werden eingerückt, um eine deutlichere Abgrenzung zum Fließtext zu gestalten.

2 Über Maxima

Maxima ist ein System zur Manipulation von symbolischen und numerischen Ausdrücken. Diese beinhalten beispielsweise Ableiten, Integrieren, Differentialgleichungen, Listen, Matrizen, u.V.m. *Maxima* bietet extrem präzise numerische Resultate durch seine Implementierung der Ganz-/Gleitkommazahlarithmetik und kann sowohl 2-dimensionale als auch 3-dimensionale Grafiken erzeugen. Der Quellcode kann dabei auf vielen Betriebssystemen wie Windows, Linux und MacOS X kompiliert werden. Somit ist *Maxima* auf den gängigsten Betriebssystemen zugänglich. *Maxima* ist ein Nachfolger des Computeralgebrasystems *Macsyma*, welches in den späten 1960er Jahren am MIT entwickelt wurde. Es ist der einzige Nachfolger der noch frei zugänglich ist und dank seiner Open-Source-Philosophie eine rege Mitwirkerschaft besitzt. Der Ableger *Maxima* wurde von 1982 bis 2001 von William Schelter weiterentwickelt und gewartet, welcher sich in 1998 die Rechte gesichert hat, den Quellcode unter der GPL zu veröffentlichen. *Maxima* wird regelmäßig geupdatet, um Fehler zu warten und Quellcode sowie Dokumentation weiter zu verbessern.[1]
Maxima ist in *Common Lisp* programmiert, eine der ersten funktionalen Programmiersprachen. Dies spiegelt sich auch deutlich in der Syntax von *Maxima* wieder.
Maxima kann als Konsolenanwendung genutzt werden (der sogenannte *Text-Mode*), oder über das GUI-Tool *wxMaxima*, welches eine grafische, dokumentenbasierte Nutzeroberfläche zur Verfügung stellt (welches auch in dieser Facharbeit verwendet wird).

[1] Dieser Textabschnitt ist eine Zusammenfassung des *Maxima*-Abstracts, welcher auf der offiziellen *Maxima*-Webseite www.maxima.sourcefourge.net zu finden ist.

3 Einführung in Maxima

Um die Algorithmen später gut nachvollziehen zu können, soll zunächst eine kurze Einführung in die Syntax von *Maxima* gegeben werden. In den folgenden Unterabschnitten wird die Funktionalität von *Maxima* nur grob umrissen. Wichtige Befehle und Funktionen, welche für die Implementierung der Algorithmen benötigt werden, werden in den jeweiligen Abschnitten besprochen.

3.1 Grundbegriffe & Zuweisungen

Maxima kann in seiner Grundfunktion als CAS wie ein Taschenrechner benutzt werden:

```
(%i1) 17 + 4 / 2;
(%o1) 19
```

Ein Semikolon (`;`) signalisiert *Maxima* dabei das Ende einer Zeile, bzw. das Ende einer Anweisung. Jeder Eingabe/Ausgabe wird dabei automatisch ein Label vergeben, sodass beide in späteren Programmblöcken separat angesprochen werden können. Dies geschieht mit `''Labelname`:

```
(%i2) ''%o1^3;
(%o2) 6859
```

Das (`^`) dient dabei als Potenzoperator und potenziert in diesem Fall das Ergebnis (`19`) aus der ersten Rechnung mit `3`.
Zuletzt berechnete Ergebnisse müssen nicht über das Label angesprochen werden, sondern können direkt mit `%` abgerufen werden:

```
(%i3) sqrt(%), numer;
(%o3) 82.8190799272728
```

Mit `ANWEISUNG, numer;` erhält man das Ergebnis als Gleitkommazahl gerundet. Die Genauigkeit lässt sich dabei mit `fpprec: NACHKOMMASTELLEN` regulieren. Die Verwendung von `numer` ist oft wichtig zur Bewertung von Ergebnissen, da der Aufruf der Wurzelfunktion `sqrt(6859)` beispielsweise im Normalfall $19^{3/2}$ liefert.

Zuweisungen werden in *Maxima* mit einem Doppelpunkt (`:`) ausgeführt, da das Gleichheitszeichen (`=`) ausschließlich für Gleichungen reserviert ist:

```
(%i4) eq1 : x^2 + x - 2;
(%o4) x^2 + x - 2
```

Das Polynom $x^2 + x - 2$ ist nun in der Variable `eq1` gespeichert und kann in den folgenden Berechnungen verwendet werden.
(Achtung: Bei der Definition von Variablen dürfen keine bereits vorhandenen Konstanten überschrieben werden, die z.B. durch `%e` für die eulersche Zahl e oder `%pi` für die Kreiszahl π definiert sind.)
Mit:

```
(%i5) solve(eq1 = 0, x);
(%o5) [x = 1, x = -2]
```

lassen sich die Nullstellen der oben definierten Funktion berechnen. Der `solve`-Befehl kann auch mit Listen als Argumenten aufgerufen werden:

```
solve([GLEICHUNG_1, ..., GLEICHUNG_n], [VAR_1, ..., VAR_k])
```

Dieser Aufruf versucht ein Gleichungssystem mit n Gleichungen und k Variablen zu lösen. Listen können in *Maxima* eine beliebige Anzahl an verschiedenen Datentypen enthalten, die durch ein Komma (`,`) getrennt sind.
Dass das erste Ergebnis aus `%o5` tatsächlich stimmt, lässt sich in *Maxima* leicht verifizieren:

```
(%i6) eq1, ''%o5[1];
(%o6) 0
```

Da die Menge der Nullstellen eine Liste ist, kann mit `LISTE[k]` das k-te Ergebnis angesprochen werden. *Maxima* benutzt dann das Ergebnis `x = 1` als Ersetzungsregel für `eq1`.
(Achtung: Wie in vielen CAS beginnt *Maxima* bei mehrdimensionalen Objekten wie z.B. einer Liste bei 1 zu indizieren. Auch wenn diese Handhabung in der mathematischen Software soweit üblich ist, beginnen Programmiersprachen im Normalfall die Indizierung bei 0.)

3.2 Funktionsblöcke & Kontrollstrukturen

3.2.1 Funktionsblöcke

Um mehrzeilige Algorithmen mit Eingabeparametern in *Maxima* programmieren zu können, benötigen wir sogenannte *Funktionsblöcke*. Diese werden definiert durch:

```
FUNKTIONSNAME(EINGABEPARAMETER) := block([LOKALE VARIABLEN],
                ANWEISUNG_1, ANWEISUNG_2, ...)
```

Im folgenden, einfachen Beispiel wird eine Funktion definiert, die 2 Zahlen addiert und dann quadriert:

```
(%i7) add_n_square(x1, x2) := block([add, square],
         add : x1 + x2,
         print("Zahlen addiert!"),
         square : add^2,
         print("Ergebnis quadriert!"),
         square
      )$
```

Das Dollarzeichen ($) fungiert wie (;) und signalisiert das Ende einer Zeile/ Anweisung, unterdrückt dabei aber die Ausgabe. Dies ist sehr hilfreich bei Funktionsblöcken, da *Maxima* den Block sonst nochmal komplett in der Ausgabe wiedergibt. Ein Aufruf unserer Funktion liefert folgendes Ergebnis:

```
(%i8) add_n_square(4,5);
      Zahlen addiert!
      Ergebnis quadriert!
(%o8) 81
```

Auch wenn die Kommentierung der Zwischenschritte durch die `print(...)`-Befehle in diesem trivialen Beispiel nicht wirklich nötig ist, so kann die Ausgabe von Zwischenergebnissen/Ankerpunkten in komplexeren Algorithmen deutlich von Vorteil sein.
(Achtung: Die `print`-Methode gibt zwar den Text als Zwischenschritt aus, aber erst, nachdem die Berechnung **vollständig** durchgelaufen ist. Wir werden später noch eine Methode kennenlernen, um Zwischenschritte während der Berechnung auszugeben.)

3.2.2 Kontrollstrukturen

Kontrollstrukturen stellen wichtige Funktionen zur Flusskontrolle eines Programms dar, insbesondere `if/else`-Bedingungen und Schleifen.
Eine `if/else`-Bedingung:

```
if BEDINGUNG_1 then ANWEISUNG_1 elseif BEDINGUNG_2
   then ANWEISUNG_2 elseif ... else ANWEISUNG_0
```

evaluiert die erste `ANWEISUNG_k`, für die `BEDINGUNG_k` wahr ist. Sollte keine der Bedingungen wahr sein, wird die `else`-Anweisung ausgeführt (`ANWEISUNG_0`).
Eine einfache Anwendung bietet die Prüfung, ob eine Zahl gerade ist:

```
(%i9) is_even(n) := block([],
         if mod(n,2) = 0 then
             (print(n, "ist gerade"), NULL)
         else
             (print(n, "ist ungerade"), NULL)
      )$
(%i10) is_even(256);
       256 ist gerade
(%o10) NULL
```

Da wir die Ausgabe des Ergebnisses bereits durch einen `print`-Befehl handhaben, definieren wir die *Maxima*-Ausgabe der Funktion als `NULL`, was in unserem Beispiel eine erfolgreiche Terminierung des Programms bedeutet. Wie oben gut zu sehen ist, müssen Anweisungen, die selbst wieder mehrere Anweisungen enthalten, durch Kommata getrennt in eine Klammer eingeschlossen werden (`ANWEISUNG_1_1, ANWEISUNG_1_2, ...`).

Maxima bietet zusätzlich eine Vielzahl an Schleifen und Sprunganweisungen. Im nächsten Beispiel wird eine dieser Möglichkeiten vorgestellt, die `for-while`-Schleife.
Das Programm soll die Summe der Quadrate aller ungeraden Zahlen bis zu einer Grenze n bestimmen:

```
(%i11) odd_squares_sum(n) := block([sum : 0],
          for i : 1 step 2 while i <= n do
              sum : sum + i^ 2,
          sum
       )$
(%i12) odd_squares_sum(15);
(%o12) 680
```

Die Schleife beginnt bei $i = 1$ und läuft solange, bis $i > n$ ist. Dabei wird i in 2-er-Schritten erhöht.
Weitere Kontrollstrukturen werden, wie bereits erwähnt, bei Bedarf in späteren Abschnitten besprochen.

3.2.3 Rememberprogrammierung

Ein weiteres Feature findet sich in der *Rememberprogrammierung*. Wenn wir in *Maxima* einen Funktionsblock aufrufen, so werden stets alle Berechnungen neu ausgeführt. In manchen Situationen ist es aber hilfreich, wenn bereits berechnete Ergebnisse nicht nochmal berechnet werden müssen. Dies bringt natürlich einen gewissen Speicheraufwand mit sich, ist aber für manche Anwendungen deutlich effizienter. Ein Paradebeispiel hierfür liefert die rekursive Definition der Fibonacci-Zahlen:

```
(%i13) Fib[0] : 0$
(%i14) Fib[1] : 1$
(%i15) Fib[n] := Fib[n-1] + Fib[n-2]$
(%i16) Fib[10];
(%o16) 55
(%i17) Fib[15];
(%o17) 610
```

Beim Aufruf von `Fib[10]` wird die 10-te Fibonacci-Zahl berechnet. Gleichzeitig werden alle berechnen Fibonacci-Zahlen zwischen `Fib[2]` und `Fib[10]` gespeichert. Beim Aufruf von `Fib[15]` werden dann nur noch die Zahlen zwischen `Fib[11]` und `Fib[15]` neu berechnet, die anderen Ergebnisse/Berechnungen werden aus dem Speicher genommen. Der Unterschied in der Art der Programmierung liegt in der Verwendung von eckigen Klammern (`[]`) anstelle von runden Klammern (`()`) bei der Definition des Funktionsblocks.

4 Langzahlarithmetik

Die Langzahlarithmetik beschäftigt sich mit Operationen und Algorithmen für beliebig große ganze Zahlen ($\mathbb{Z}$). Wir wollen uns in diesem Abschnitt zwei Beispielen dieses Teilgebiets widmen, der *Primfaktorzerlegung* und dem *(erweiterten) Euklidischen Algorithmus*. Beide legen den Grundstein für spätere Algorithmen der *Modularen Arithmetik* und der *Kryptographie*.

4.1 Teilertheorie & Primfaktorzerlegung

Um einen Algorithmus für die Primfaktorzerlegung definieren zu können, muss die Existenz einer solchen zunächst gerechtfertigt werden. Beginnen wir mit folgender Definition:

Definition 4.1. Seien $x, y \in \mathbb{Z}$. Wir sagen „x **teilt** y“, „x ist **Teiler** von y“ oder „y ist ein **Vielfaches** von x“, falls ein $d \in \mathbb{Z}$ existiert, sodass $y = d \cdot x$ ist. In diesem Fall schreiben wir $x|y$. Falls $d, x \neq 1$ sind, nennen wir das Produkt $d \cdot x$ eine **Zerlegung** von y.

Mit der Definition eines Teilers können wir nun definieren, was es für eine Zahl heißt, *prim* zu sein:

Definition 4.2. Eine Zahl $p \in \mathbb{Z}$ heißt **prim**, falls aus $d \in \mathbb{Z}$ mit $d|p$ folgt: Entweder ist $d = 1$ oder $d = p$. Dann nennen wir p eine **Primzahl**.

Umgangssprachlich ist eine Primzahl also nur durch 1 und sich selber teilbar.
Wir bezeichnen von nun an die Menge aller Primzahlen mit $\mathbb{P}$.
Bevor wir uns der Primfaktorzerlegung zuwenden können, benötigen wir noch folgenden Hilfssatz:

Lemma 4.3. *Seien $p \in \mathbb{P}$ und $a, b \in \mathbb{Z}$. Dann gilt:*

$$p|a \cdot b \Rightarrow p|a \text{ oder } p|b$$

Mit allen Hilfsmitteln ausgestattet können wir uns nun einem der wichtigsten Sätze der Zahlentheorie annehmen:

Satz 4.4. ***(Fundamentalsatz der Zahlentheorie)*** *Jede Zahl $x \in \mathbb{N}_{\geq 2}$ besitzt eine* ***eindeutige*** *Zerlegung in Primzahlen (Primfaktorzerlegung).*

Beweis (PFZ := Primfaktorzerlegung/-en)
Existenz: Beweis durch vollständige Induktion:
IA Sei $x = 2$. Da 2 Primzahl ist, hält die Vermutung.
IV Jede Zahl $z \in \mathbb{N}_{\geq 2}$ mit $z < x$ besitzt eine PFZ.
IS Sei $x > 2$. Dann existieren für x zwei Möglichkeiten:
Möglichkeit 1: x ist prim. Dann stellt x sich selber dar und ist somit seine eigene PFZ.
Möglichkeit 2: x besitzt eine Zerlegung $x = a \cdot b\,(a, b \in \mathbb{N}_{\geq 2})$. Dann ist per Definition einer Zerlegung aber $a < x$ und $b < x$. Nach der IV existieren für a und b jeweils eine PFZ, also auch für $x = a \cdot b$ als Produkt der einzelnen PFZ.
Eindeutigkeit: Beweis durch vollständige Induktion:
IA Sei $x = 2$. Dann ist die PFZ trivialerweise eindeutig.
IV Für jede Zahl $z \in \mathbb{N}_{\geq 2}$ mit $z < x$ ist die PFZ eindeutig.
IS Sei $x > 2$. Seien $x = p_1 \cdots p_n = q_1 \cdots q_m$ mit $p_j \in \mathbb{P}\,(j = 1, ..., n)$ und $q_j \in \mathbb{P}\,(j = 1, ..., m)$. Da $p_1 | x = q_1 \cdots q_m$ folgt mit Lemma 4.3, dass $p_1 | q_j$ für ein j aus $1, ..., m$. Da q_j prim ist, folgt $p_1 = q_j$. Also ist $\frac{x}{p_1} \in \mathbb{N}$ mit:

$$x > \tfrac{x}{p_1} = p_2 \cdots p_n = q_1 \cdots q_{j-1} \cdot q_{j+1} \cdots q_m$$

Aus der IV folgt, dass die PFZ von $\frac{x}{p_1}$ gleich sein müssen, d.h. $p_2, ..., p_n = q_1, ..., q_{j-1}, q_{j+1}, ..., q_m$ bis auf Umordnung. Da aber $p_1 = q_j$ist, gilt auch $p_1, ..., p_n = q_1, ..., q_m$ bis auf Umordnung.

□

Maxima liefert mit `ifactors(n)` (kurz für: Integerfactors) eine effiziente Implementierung zur Bestimmung der PFZ von n:

```
(%i18) next_prime(10^20) · next_prime(10^21);
(%o18) 10000000000000000005070000000000000004563
(%i19) ifactors(%);
(%o19) [[10000000000000000039,1],[100000000000000000117,1]]
```

Die Funktion `next_prime(k)` liefert dabei die nächste Primzahl größer k. Dies gibt uns eine einfache Möglichkeit, die Funktion `ifactors` in ihrem

worst-case zu testen, nämlich genau dann, wenn die Zahl zwei große Primfaktoren besitzt. Dies zeigt auch folgende Analyse:

```
(%i20) time(%o19);
(%o20) [30.161]
```

Der Befehl `time(%o1, %o2, ...)` liefert die Zeit in Sekunden, welche die jeweiligen Ausgaben benötigt haben. In der Größenordnung einer 41-stelligen Zahl benötigt der Algorithmus im *worst-case*-Szenario bereits ca. 30 Sekunden. Von der Komplexität des Faktorisierens profitieren insbesondere Anwendungen der Kryptographie, wie wir in den späteren Abschnitten noch sehen werden.
(Achtung: Bei der Arbeit mit *wxMaxima* steht die Arbeit in Form eines kompletten Dokumentes im Vordergrund. Ändert man in diesem Prozess eine bereits ausgeführte Zeile ab oder schreibt diese neu, ändern sich auch `%i` & `%o`-Nummer. Dies kann schnell zu Fehlern/falschen Ergebnissen führen, insbesondere bei Befehlen, die bestimmte Ausgaben referenzieren.)

Wir werden uns dem Problem des Faktorisierens in einem späteren Kapital nochmal annehmen, nachdem wir etwas Vorarbeit in der *Modularen Arithmetik* geleistet haben. Bis dahin soll folgender Beispielalgorithmus gegeben werden, um die Primfaktorzerlegung händisch zu implementieren:

```
(%i21) factor_int(n) := block([div : 2, factors : [],
                               k : n, n_new],
          while k > 1 and not(primep(k)) do (
              while integerp(n_new : k / div) do (
                 factors : append(factors, [div]),
                 k : n_new
              ),
          div : next_prime(div)
          ),
          if primep(k) then factors : append(factors, [k]),
          factors
       )$
```

Dieser Algorithmus verwendet zwei interessante Schlüsselwörter, die wir etwas näher betrachten wollen:
`integerp(n)/primep(n)`: Test, ob $n \in \mathbb{Z}/n \in \mathbb{P}$ ist.

`append(liste_1, ..., liste_n)`: Fügt die Listen $1, \ldots, n$ zusammen.

Wie im Algorithmus gut zu sehen ist, kann die `while`-Schleife auch losgelöst von der `for`-Schleife implementiert werden. Mehrere Bedingungen können dabei mit `and/or` zusammengefügt werden. `not(BEDINGUNG)` negiert die jeweilige `BEDINGUNG`.
Zwar stellt die Abfrage `not(primep(k))` in Zeile 3 eine Optimierung zur klassischen *Probedivision*[2] dar, es zeigen sich aber doch deutliche Unterschiede zur effizienten Implementierung von *Maxima* :

```
(%i22) factor_int(next_prime(10^7) · next_prime(10^8));
(%o22) [10000019,100000007]
(%i23) time(%o22);
(%o23) [25.5]
```

Auch wenn unser Algorithmus Zahlen der Größenordnung 10^15 im *worst-case*-Szenario noch in angemessener Zeit faktorisieren kann, so gilt dies nicht für die Eingabe mit der Zahl aus unserem Initialbeispiel `(%i18)`:

```
(%i24) factor_int(''%o18); /· Sollte abgebrochen werden ·/
```

Schon bei Zahlen der Größenordnung 10^{20} beträgt die Laufzeit des Algorithmus weit über 5 Minuten.
Die Eingabe aus `%i24` sollte mit der Tastenkombination `Strg+g` abgebrochen werden. Wie oben zu sehen ist, lassen sich mit `/* Kommentar */` Kommentare einfügen. (*Maxima* macht dabei automatisch aus (`*`) ein (·))

Nach dem Fundamentalsatz der Zahlentheorie wollen wir uns einem weiteren, sehr wichtigen zahlentheoretischen Algorithmus widmen:

4.2 Der Euklidische Algorithmus

Wie in Definition 4.1 festgelegt, nennen wir eine Zahl x einen Teiler von y nur dann, falls ein d existiert, sodass $y = d \cdot x$ ($\in \mathbb{Z}$). Diese Definition ist sinnvoll, da jede Lockerung dieser Bedingung dazu führen würde, dass wir den Bereich der ganzen Zahlen ($\mathbb{Z}$) verlassen müssten. Trotzdem können wir auch in $\mathbb{Z}$ eine Division $\frac{y}{x}$ mit $x \nmid y$ durchführen, in dem wir uns trivialerweise der Grundschulmathematik bedienen:

[2] Die *Probedivision* beschreibt das Verfahren, in dem n sukzessive durch die Primzahlen $p \in \mathbb{P}$ beginnend mit $p = 2$ dividiert wird, bis $n = 1$ ist.

Definition u. Satz 4.5. (Division mit Rest) *Seien $y \in \mathbb{N}_{\geq 0}$ und $x \in \mathbb{N}$. Dann gibt es **genau** ein $(q, r) \in \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0}$ und $0 \leq r < x$ sodass:*

$$y = q \cdot x + r$$

Wir nennen $q = quotient(y, x)$ den **ganzzahligen Quotient/Anteil** von y und x, sowie $r = mod(y, x)$ den **(Divisions-)rest** der Division von y durch x. Falls $r = 0$ ist, so gilt nach Definition 4.1 $x|y$.

Satz 4.5 lässt sich dabei recht intuitiv auf den Bereich der ganzen Zahlen $(y, x \in \mathbb{Z}, x \neq 0)$ erweitern:

$\underline{y < 0 \ \& \ x > 0}$:	Setze $y = -y \rightarrow$ Berechne $y = q \cdot x + r \rightarrow$
	Setze $q = -q - 1$ und $r = -r + x$
$\underline{y > 0 \ \& \ x < 0}$:	Setze $x = -x \rightarrow$ Berechne $y = q \cdot x + r \rightarrow$
	Setze $q = -q$
$\underline{y < 0 \ \& \ x < 0}$:	Setze $y = -y$ und $x = -x \rightarrow$ Berechne $y = q \cdot x + r \rightarrow$
	Setze $q = -q - 1$ und $r = -r + (-x)$

Dabei wird stets die Bedingung $0 \leq r < x$ beibehalten.
In *Maxima* liefern folgende Eingaben den Quotienten resp. den Rest:

```
(%i25) floor(2300/7);
(%o25) 328
(%i26) mod(2300, 7);
(%o26) 4
```

Man beachte dabei, dass die `floor`-Funktion nicht explizit den ganzzahligen Quotienten der Division mit Rest berechnet, sondern die Rechnung in $\mathbb{Q}$ durchführt und das Ergebnis auf die nächste ganze Zahl abrundet. Dabei gilt aber stets *quotient(y,x)* = `floor(y/x)` $:= \lfloor \frac{y}{x} \rfloor$. Um den Euklidischen Algorithmus definieren zu können, müssen wir zunächst definieren, was sein Ziel ist:

Definition 4.6. Seien $a, b, c \in \mathbb{Z}$. Die Zahl c heißt **gemeinsamer Teiler** von a und b, falls $c|a$ und $c|b$. Falls für c nur $c \in \{-1, 1\}$ die Bedingung erfüllt, nennen wir a und b **teilerfremd**. Falls für alle anderen Teiler $d \in \mathbb{Z}$ mit $d|a$ und $d|b$ gilt $d|c$, heißt c der **größte gemeinsame Teiler** von a und b
$(c := gcd(a, b))$.

Die Definition des größten gemeinsamen Teilers führt uns zu folgendem Algorithmus:

Definition u. Satz 4.7. Seien $a, b \in \mathbb{Z}$. Das Iterationsschema:

$$x_{k-1} = q_k \cdot x_k + r_k$$

mit den Anfangsbedingungen ($x_0 = a, x_1 = b$) und den Iterationsvorschriften:

$$q_k = quotient(x_{k-1}, x_k) \qquad r_k = mod(x_{k-1}, x_k) \qquad x_{k+1} = r_k$$

heißt der **Euklidische Algorithmus**. Dabei gilt die Abbruchbedingung $r_k = 0$.
Der oben beschriebene Algorithmus ***terminiert für alle a, b*** *und liefert bei n Schritten*[3] $\boldsymbol{x_n = gcd(a, b)}$.

Um den Satz aus 4.7 zu beweisen, müssen wir davon ausgehen können, dass der größte gemeinsame Teiler von a und b unter dem Iterationsschema invariant bleibt. Diese Annahme führt zu folgendem Hilfssatz:

Lemma 4.8. *Seien $a, b \in \mathbb{Z}$ mit $a = q \cdot b + r$ durch Division mit Rest. Dann gilt:*

$$gcd(a, b) = gcd(b, r)$$

Beweis zu 4.7 Da der Euklidische Algorithmus mit der Division mit Rest arbeitet, und für jeden Rest r_k: $0 \leq r_k < x_k$ gilt, muss nach endlich vielen Schritten $r_k = 0$ erreicht werden. Sei n der letzte Iterationsschritt. Aus Lemma 4.8 folgt, dass $gcd(a, b) = gcd(x_n, r_n = 0)$ sein muss. Aus der Definition des größten gemeinsamen Teilers macht man sich schnell klar, dass für alle $x \in \mathbb{Z} : gcd(x, 0) = x$ gelten muss, da stets $x|0$ gilt und x somit die größte Zahl ist, die auch sich selber noch teilt. Daraus folgt:

$$gcd(a, b) = gcd(x_n, 0) = x_n$$

□

Die Konstruktion des Euklidischen Algorithmus liefert uns dabei zusätzlich eine Identität für den größten gemeinsamen Teiler:
Da $gcd(a, b) = x_n = r_{n-1}$ und $r_{n-1} = x_{n-2} - q_{n-1} \cdot x_{n-1}$ ist, gilt insbesondere $gcd(a, b) = x_{n-2} - q_{n-1} \cdot x_{n-1}$. Durch sukzessive Substitution erhält man so eine Darstellung $gcd(a, b) = s \cdot a + t \cdot b$ mit $s, t \in \mathbb{Z}$, die sogenannten *Bézoutkoeffizienten*.

[3] Die Anzahl der Schritte wird auch **euklidische Länge** von a und b genannt.

Der Euklidische Algorithmus samt Darstellung des Ergebnisses durch die Bézoutkoeffizienten wird auch der **erweiterte Euklidische Algorithmus** genannt. Diesen wollen wir in *Maxima* implementieren:

```
(%i27) ext_gcd(a,b) := block([q, r, s : [1,0], swapped : false,
                              t : [0,1], tmp],
          if a < b then (
             tmp : a, a : b, b : tmp, swapped : true
          ),
          if mod(a,b) = 0 then
             if swapped then
                return([b,[1,0]]) else
                return([b,[0,1]]),
          while not(mod(a,b) = 0) do (
             [q,r] : [floor(a/b),mod(a,b)],
             [a,b] : [b,r],
             s : [s[2],s[1] - q · s[2]],
             t : [t[2],t[1] - q · t[2]]
          ),
          if swapped then
             [b,[t[2],s[2]]] else [b,[s[2],t[2]]]
       )$
```

Der Algorithmus überprüft zunächst, ob $a < b$ gilt. Falls dies nicht der Fall ist, werden die Zahlen getauscht und der Algorithmus arbeitet normal weiter. Damit aber am Ende das richtige Ergebnis für die Zahlen a, b in Eingabereihenfolge ausgegeben werden kann, wird mit der Variable `swapped` festgehalten, ob getauscht wurde oder nicht. Falls $b|a$ gilt, haben wir den $gcd(a, b)$ bereits gefunden, nämlich b, mit $gcd(a, b) = 0 \cdot a + 1 \cdot b$. Hier wird zum ersten Mal das `return(...)`-Statement benutzt, welches das gesuchte Ergebnis zurückliefert und gleichzeitig den Funktionsblock an dieser Stelle beendet. (`return(...)` innerhalb einer Schleife beendet **nur** die Schleife!)
In der `while`-Schleife sieht man eine Technik, die auch im weiteren Verlauf dieser Arbeit noch öfter Anwendung finden wird:
Mit `[VAR_1, ..., VAR_n] : [VAL_1, ..., VAL_n]` lassen sich mehrere Zuweisungen gleichzeitig machen.

Eine weitere Eigenschaft von *Maxima* findet sich in der Arbeitsweise des Algorithmus. Betrachten wir folgenden Aufruf:

```
(%i28) [a,b] : [234,12]$
(%i29) ext_gcd(a,b);
(%o29) [6, [1, -19]]
(%i30) [a,b];
(%o30) [234,12]
```

In der Implementierung des Algorithmus ist zu sehen, dass die Werte a und b nicht in lokalen Variablen zwischengespeichert werden, sondern direkt im Algorithmus verwendet und **verändert** werden. Auf globaler Ebene bleiben die Variablen a und b dennoch unverändert. *Maxima* verwendet also entweder eine lokale Kopie der Eingabeparameter oder ändert die Werte am Ende des Funktionsaufrufs wieder auf ihr Original. Bei *Mathematica* beispielsweise ist das Verändern der Eingabeparameter eines Funktionsblocks innerhalb eines solchen nicht erlaubt!
Maxima selber bietet natürlich auch eine Implementierung beider Varianten des Euklidischen Algorithmus. Um die Suche nach den Funktionsnamen etwas einfacher zu gestalten, bietet *Maxima* folgende Hilfestellung:

```
(%i31) apropos("gcd");
(%o31) [ef_gcd,ef_gcdex,ezgcd,gcd,gcdex,gf_gcd,gf_gcdex,gcd]
```

`apropos("STICHWORT")` liefert alle Funktionsnamen **und** Flags (Einstellungen) die `STICHWORT` beinhalten[4]. Für eine genauere Beschreibung der Funktion kann man `?? FUNKTIONSNAME` oder `describe("FUNKTIONSNAME")` aufrufen. Folgender Aufruf ist somit äquivalent zu unserem Algorithmus:

```
(%i32) gcdex(a,b);
(%o32) [1,-19,6]
```

Hierbei sind die ersten beiden Argumente die Bézoutkoeffizienten und das dritte Argument ist der größte gemeinsame Teiler.

[4]Deswegen kommt `gcd` hier auch zwei mal vor. Die Funktion selber und die Flag welche steuert, ob der größte gemeinsame Teiler bei Umwandlungen mitgezogen werden soll. Das Setzen dieser Flag auf `false` kann gewisse Prozesse beschleunigen, in denen der *gcd* nicht weiter von Nöten ist.

Ein Laufzeitvergleich der beiden Algorithmen macht hier wenig Sinn:

```
(%i33) seed : make_random_state(3993)$
(%i34) set_random_state(seed)$
(%i35) showtime : true$
       Evaluation took 0.0000 seconds (0.0000 elapsed) using
       0 bytes.
(%i36) gcdex(random(10^500), random(10^500));
       Evaluation took 0.0270 seconds (0.0350 elapsed) using
       3.622 MB.
(%o36) [10561521409609790064146685743 8[440 Ziffern]
       189453297140473772094144467045,
       -360244168306652521232854578100[440 Ziffern]
       255727470295515905228882708693,1]
(%i37) set_random_state(seed)$
(%i38) ext_gcd(random(10^500), random(10^500));
       Evaluation took 0.1580 seconds (0.1600 elapsed) using
       10.985 MB.
(%o38) [1,[10561521409609790064146685743 8[440 Ziffern]
       189453297140473772094144467045,
       -360244168306652521232854578100[440 Ziffern]
       255727470295515905228882708693]]
```

Wie im Beispiel gut zu sehen ist, ist unser Algorithmus zwar etwas langsamer und speicheraufwendiger, evaluiert das Ergebnis aber trotzdem sehr schnell, zu mal es sich um Zahlen mit ca. 500 Stellen handelt.[5]
Die Funktionen `make_random_state/set_random_state` sorgen dafür, dass wir die von `random(GRÖßE)` erzeugten Zufallszahlen reproduzieren können. `make_random_state(ZAHL)` erstellt dabei einen Startpunkt für den Random-Number-Generator (RNG) von *Maxima* , welcher zu `ZAHL` eindeutig zugeordnet ist. `set_random_state(RANDOM_STATE)` setzt den RNG immer wieder auf den von uns gewählten Ausgangspunkt zurück.[6]

[5] Die `random`-Funktion schwankt dabei bei der Erstellung der Zahlen zwischen ∼498-500 Stellen.

[6] Wann der RNG voranschreitet, ist von Programm zu Programm unterschiedlich.

Die Flag `showtime` sorgt auf `true` dafür, dass jeweils Zeit-und Speicherverbrauch mit ausgegeben werden. Diese kann nach der Evaluation wieder auf `false` gesetzt werden:

```
(%i39) showtime : false$
```

5 Modulare Arithmetik

So simpel wie die Idee der Division mit Rest auch scheint, eröffnet sie uns doch ein komplett neues Teilgebiet der Algebra/Zahlentheorie, das Gebiet der *Modularen Arithmetik*. Damit wir uns den interessanten Aussagen und Algorithmen dieses Gebiets widmen können, ist aber erstmal etwas algebraische Vorarbeit nötig:

5.1 Gruppen, Ringe & Kongruenzen

Definition 5.1. Eine Menge G mit einer inneren, zweistelligen Verknüpfung $\star$ (Abbildung $\star : G \times G \to G, (a, b) \mapsto a \star b$) heißt **Gruppe**, falls die Verknüpfung folgende Eigenschaften erfüllt:

1. $(a \star b) \star c = a \star (b \star c)$ für alle $a, b, c \in G$ (Assoziativität)
2. $\exists e \in G$ mit $a \star e = e \star a = e$ für alle $a \in G$ (neutrales Element)
3. $\forall a \in G\ \exists a^{-1} \in G$ mit $a \star a^{-1} = a^{-1} \star a = e$ (inverses Element)

Gilt zusätzlich $a \star b = b \star a\ \forall a, b \in G$ nennen wir $(G, \star)$ **abelsche Gruppe**. Erfüllt $\star$ nur die erste Bedingung, nennen wir $(G, \star)$ **Halbgruppe**.

Dies führt uns zur Definition eines Rings:

Definition 5.2. Eine Menge R mit zwei inneren, zweistelligen Verknüpfungen $+, \cdot$ heißt **Ring**, falls folgende Eigenschaften erfüllt sind:

1. $(R, +)$ ist abelsche Gruppe
2. $(R, \cdot)$ ist Halbgruppe
3. Es gelten die Distributivgesetze:
   $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ und
   $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$ für alle $a, b, c \in R$

Besitzt $(R, \cdot)$ ein neutrales Element, so nennen wir $(R, +, \cdot)$ einen **unitären Ring (Ring mit Eins)**.
Ist $(R, \cdot)$ abelsch, nennen wir R einen **kommutativen Ring**.

Offensichtlich bietet unsere bisher betrachtete Menge $\mathbb{Z}$ mit $(\mathbb{Z}, +, \cdot)$ ein perfektes Beispiel für einen unitären, kommutativen Ring. Die Definition eines Rings ist essentiell für viele Teilgebiete der Algebra. Können wir zeigen, dass eine Struktur ein Ring ist, so können wir uns dessen Ringeigenschaften zu Nutze machen und darauf aufbauen.
Eine weitere Klassifizierung von wichtigen Elementen in Ringen liefert folgende Definition:

Definition u. Satz 5.3. Sei $(R, +, \cdot)$ ein unitärer, kommutativer Ring und 1_R das Einselement (neutrale Element bzgl. $\cdot$). Ein Element $a \in R$ heißt **Einheit**, falls ein $b \in R$ existiert mit:

$$a \cdot b = b \cdot a = 1_R$$

Die Menge der Einheiten $R^\times := \{x \in R \mid x \text{ ist Einheit}\}$ *bildet eine Gruppe.*

Die Menge $R^\times$ sind also alle Elemente, die ein Inverses bzgl. $\cdot$ besitzen. Für unseren Beispielring $(\mathbb{Z}, +, \cdot)$ gilt somit $\mathbb{Z}^\times = \{1, -1\}$.
Ausgehenden von unseren Definitionen wollen wir zunächst die Division mit Rest in eine algebraische Struktur überführen:

Definition u. Satz 5.4. Zwei Zahlen $a, b \in \mathbb{Z}$ heißen **kongruent modulo** $p \in \mathbb{N}_{\geq 2}$, falls sie bei der Division mit Rest durch p denselben Rest haben, d.h.:

$$mod(a, p) = mod(b, p)$$

Wir schreiben dann $\mathbf{a \equiv b \,(mod\, p)}$.
Die Kongruenz ($\equiv$) *definiert eine Äquivalenzrelation auf* $\mathbb{Z}$*, d.h. sie liefert eine Klasseneinteilung von* $\mathbb{Z}$ *in* p *Klassen* $([0]_p, \cdots, [p-1]_p)$*, wobei gilt:*

$$a \in \mathbb{Z}, r \in \{0, \cdots, p-1\} : a \in [r]_p \Leftrightarrow mod(a, p) = r$$

Zusätzlich ist sie verträglich mit der Addition und (Skalar-)Multiplikation.
Die Äquivalenzklassen $[a]_p$ werden **Restklassen** genannt, mit ihrem zugehörigen Vertreter $a \in \{0, \cdots, p-1\}$ und $\mathbb{Z}_p := \{[0]_p, \cdots, [p-1]_p\}$ als Menge aller Restklassen.

Die Verträglichkeitseigenschaften aus Satz 5.4 liefern uns dabei folgende, recht intuitive, Rechenregeln für Restklassen:

Definition u. Satz 5.5. Sei $p \in \mathbb{N}_{\geq 2}$ und φ Abbildung mit:

$$\begin{aligned} \varphi : \mathbb{Z} &\to \mathbb{Z}_p \\ a &\mapsto \varphi(a) := [a]_p \end{aligned}$$

Damit definieren wir:

$$\begin{aligned} [a]_p \oplus [b]_p &:= \varphi(a+b) = [a+b]_p \, und \\ [a]_p \otimes [b]_p &:= \varphi(a \cdot b) = [a \cdot b]_p \end{aligned}$$

als Addition bzw. Multiplikation in $\mathbb{Z}_p$. Außerdem gilt:
$(\mathbb{Z}_p, \oplus, \otimes)$ *ist unitärer, kommutativer Ring.*

Mit Satz 5.5 können wir uns die Ringeigenschaften zu Nutze machen und gleichzeitig eine interessante Möglichkeit zur Berechnung der Einheitengruppe $\mathbb{Z}_p^\times$ liefern:

Definition u. Satz 5.6. Ein Element $a \in \mathbb{Z}_p$ ist genau dann eine Einheit in $(\mathbb{Z}_p, \oplus, \otimes)$, falls $gcd(a, p) = 1$ ist.
Daraus folgt: Ist p prim, so ist $\mathbb{Z}_p$ *Körper.*
Das Inverse $b = a^{-1} \, (mod\, p)$ wird **modulares Inverses** von a genannt, falls $a \cdot b \equiv 1 \, (mod\, p)$ ist.

Das modulare Inverse lässt sich direkt über den Erweiterten Euklidischen Algorithmus von a und p bestimmen:

```
(%i40) [s,t,g] : gcdex(3,7);
(%o40) [-2,1,1];
(%i41) mod(s,7);
(%o41) 5
```

Das modulare Inverse zu 3 in $\mathbb{Z}_7$ ist demnach 5. Dies ist offensichtlich korrekt, wie die Rechnung $3 \cdot 5 = 15 \equiv 1 \, (mod\, 7)$ zeigt. *Maxima* liefert aber auch eine eigene Methode zum Bestimmen des modularen Inverses mit:

```
(%i42) inv_mod(3,7);
(%o42) 5
```

Die Methoden der modularen Arithmetik lassen uns interessante Problemstellungen lösen:

5.2 Der chinesische Restsatz

Wir betrachten folgendes Rätsel: Ein Gruppe von Menschen kann so in 5-er Reihen aufgestellt werden, dass eine 2-er Reihe übrigbleibt und gleichzeitig so in 7-er Reihen aufgestellt werden, dass eine 4-er Reihe übrigbleibt. Wie viele Personen befinden sich in der Gruppe?
Wie oft bei solchen Zahlenrätseln lassen sich die einzelnen Bedingungen in Gleichungen ausdrücken. Die modulare Arithmetik liefert uns das Handwerkszeug dafür:
Ist x die Anzahl der Personen in der Gruppe, so lassen sich die obigen Bedingungen wie folgt darstellen:

$$x \equiv 2 \,(mod\, 5) \text{ und}$$
$$x \equiv 4 \,(mod\, 7)$$

In unserem Beispiel lässt sich die Lösung recht einfach erraten. Besteht die Gruppe aus 32 Personen, so sind beide Bedingungen erfüllt und das Rätsel ist gelöst. Unsere Lösung ist aber keinesfalls eindeutig, da zum Beispiel eine Gruppe von 67 Personen ($67 \equiv 32 \,(mod\, 35)$) das Rätsel gleichermaßen löst. Diese Grundüberlegungen führen zu folgendem Satz:

Satz 5.7. ***(Chinesischer Restsatz)*** *Seien $p_1, \cdots, p_n \in \mathbb{Z}$ paarweise teilerfremd ($gcd(p_i, p_k) = 1$ für $i \neq k$) und $a_1, \cdots, a_n \in \mathbb{Z}$. Dann besitzt das Gleichungssystem:*

$$\begin{aligned} x &\equiv a_1 \,(mod\, p_1) \\ x &\equiv a_2 \,(mod\, p_2) \\ &\vdots \\ x &\equiv a_n \,(mod\, p_n) \end{aligned}$$

eine bis auf additive Vielfache von $p_1 \cdots p_n$ eindeutige Lösung.

Beweis
Wir konstruieren zunächst eine Lösung für die ersten beiden Gleichungen:

Da $gcd(p_1, p_2) = 1$ ist, existieren Bézoutkoeffizienten $s, t \in \mathbb{Z}$ mit:

$$s \cdot p_1 + t \cdot p_2 = 1$$

Definieren wir $l := a_2 \cdot s \cdot p_1 + a_1 \cdot t \cdot p_2$, so gilt:

$$l \equiv a_1 \cdot t \cdot p_2 \equiv a_1 \cdot (1 - s \cdot p_1) \equiv a_1 \,(mod\, p_1)$$
$$l \equiv a_2 \cdot s \cdot p_1 \equiv a_2 \cdot (1 - t \cdot p_2) \equiv a_2 \,(mod\, p_2)$$

Somit ist l Lösung für die ersten beiden Gleichungen. Da aber alle p_i paarweise teilerfremd sind, sind insbesondere alle p_j für $j = 3, \ldots, n$ teilerfremd zu $p_1 \cdot p_2$. Somit können die ersten beiden Gleichungen im Gleichungssystem durch:

$$x \equiv l \,(mod\, p_1 \cdot p_2)$$

ersetzt werden. Dadurch kann das Gleichungssystem sukzessive verringert werden, bis aus den verbleibenden 2 Gleichungen eine finale Lösung l^* mit $x \equiv l^* \,(mod\, p_1 \cdots p_n)$ gewonnen werden kann.
Eindeutigkeit: Seien l^* und $\hat{l^*}$ zwei finale Lösungen. Dann gilt:

$$l^* - \hat{l^*} \equiv 0 \,(mod\, p_1), \ldots, l^* - \hat{l^*} \equiv 0 \,(mod\, p_n)$$

also auch $l^* - \hat{l^*} \equiv 0 \,(mod\, p_1 \cdots p_n)$.

□

Anhand der Konstruktionsvorschrift aus dem Beweis wollen wir den chinesischen Restsatz in *Maxima* programmieren. Aufgrund der Art der Konstruktion bietet sich hier eine rekursive Funktion an:

```
(%i43) chinese_remainder(r,p) := block([g,s,t],
          [s,t,g] : gcdex(p[1],p[2]),
          if is(length(r) = 2) then
             mod(r[1]·p[2] + r[2]·s·p[1],p[1]·p[2]) else
             chinese_remainder(
               flatten([mod(r[1]·t·p[2] + r[2]·s·p[1],p[1]·p[2]),
                        lastn(r, length(r)-2)]),
               flatten([p[1]·p[2], lastn(p, length(p)-2)])
             )
       )$
```

In diesem Algorithmus finden sich 2 neue Funktionen. `flatten(AUSDRUCK)` fügt alle Teilausdrücke mit dem selben Operator wie `AUSDRUCK` unter einem Operator zusammen. In unserem Beispiel wird dadurch aus einer Liste mit Teillisten `[A, [B, ..., N]]` eine gemeinsame Liste `[A, B, ..., N]`.
Die Funktion `lastn(LISTE, n)` liefert die letzten n Elemente der `LISTE` zurück. In Kombination sorgen die beiden Funktionen in unserem Algorithmus dafür, dass die ersten beiden Elemente von Resten und Moduli jeweils durch die berechnete Teillösung ersetzt und dann als neue Liste in den rekursiven Funktionsaufruf eingesetzt werden können.
Dass der Algorithmus korrekt arbeitet, zeigt der Vergleich mit *Maxima's* eigener Implementierung:

```
(%i44) [a,b] : [[35,24,17], [89,27,23]]$
(%i45) chinese_remainder(a,b);
(%o45) 2616
(%i46) chinese(a,b);
(%o46) 2616
```

Falls der Vergleich mit *Maxima's* Implementierung nicht genug ist, können wir das Ergebnis auch für alle Gleichungen prüfen:

```
(%i47) map(lambda([i,j], is(mod(2616,j) = i)), a, b);
(%o47) [true,true,true]
```

Die obige Codezeile führt dabei zwei wichtige Konzepte der Programmierung mit *Maxima* ein:
Die Funktion `map(FUNKTION, AUSDRUCK_1, ..., AUSDRUCK_n)` behält dabei die Struktur der Ausdrücke bei, wendet aber `FUNKTION` auf jeden einzelnen `AUSDRUCK` an. Als Funktion haben wir einen sogenannten `lambda`-Ausdruck angegeben. `lambda`-Ausdrücke fungieren als namenlose (anonyme) Funktionen und simulieren demnach die Struktur einer Funktion mitsamt Ein-/Ausgabe. Der `lambda`-Ausdruck in Zeile 47 simuliert damit eine Funktion mit Eingabeparametern i und j, welche `true/false` zurückliefert, je nachdem, ob das Ergebnis aus Zeile 45/46 modulo j gleich i ist. Der Aufruf von `map` wendet diese Überprüfung dann auf die Listen a und b an, wobei i die Liste (a) der Reste durchläuft und j die Liste (b) der Moduli.

Die Erklärung von Zeile 47 mag etwas ausgiebig erscheinen, ist aber essentiell, denn `lambda`-Ausdrücke stellen ein Schlüsselkonzept in der funktionalen Programmierung dar.
Abschließend bleibt zu unserer Implementierung noch zu sagen, dass diese kein Error-Handling beinhaltet. Ein gutes Error-Handling ist in den meisten Programmiersprachen/CAS oftmals genau so wichtig wie die Funktionalität des Programms selbst. Besonders, wie in unserem Fall, rekursive Methoden, laufen bei schlechtem Error-Handling oft Gefahr, unendlich tief (so tief wie möglich) zu rekurrieren und somit potentiell nicht nur das Programm sondern den ganzen PC zum Absturz zu bringen[7].

5.3 Modulare Quadratwurzeln

Auch in der modularen Arithmetik können Operationen wie beispielsweise das Wurzelziehen definiert werden, auch wenn die Aussagen hier meistens nicht so trivial ausfallen wie in ihrem nicht-modularen Äquivalent:

Definition 5.8. Sei $N \in \mathbb{N}$ mit $N \geq 2$. Besitzt die Gleichung:

$$x^2 \equiv a \,(mod\, N)$$

eine Lösung, so nennen wir die Lösung $x := \sqrt{a} \,(mod\, N)$ **modulare Quadratwurzel** von a. Weiter nennen wir a in diesem Fall einen **quadratischen Rest** modulo N.

Dass nicht jede Zahl eine modulare Quadratwurzel besitzt, zeigt folgende Eingabe:

```
(%i48) map(lambda([x], mod(x^2,10)), makelist(n, n, 0, 9));
(%o48) [0,1,4,9,6,5,6,9,4,1]
```

Man sieht, dass nur die Zahlen 0,1,4,5,6,9 quadratische Reste in $\mathbb{Z}_{10}$ sind. Die Zahl 7 dahingegen lässt sich nicht durch a^2, $a \in \mathbb{Z}_{10}$ darstellen.
Die Funktion `makelist` erzeugt hier eine Liste der Zahlen von 0 bis 9.

[7] CAS wie *Maxima* und *Mathematica* setzen für ihre Berechnungen einen eigenen Kernel auf und verhindern dadurch automatisch solche PC-übergreifenden Abstürze. Dennoch ist beim iterativen/rekursiven programmieren Vorsicht geboten! Weiterhin bietet ein gutes Error-Handling eine wichtige Schnittstelle zum externen Benutzer.

Dabei ist das erste n der Ausdruck für jedes Listenelement (wir wollen lediglich die Zahl an sich) und das zweite n der Name der Variablen, die die Zahlen von 0 bis 9 durchläuft.
Aus Zeile 48 können wir uns somit auch einen relativ simplen Algorithmus zur Bestimmung der modularen Quadratwurzel herleiten:

```
(%i49) mod_root(x,p) := block([quad],
          quad : map(lambda([a],mod(a^2,p)),makelist(n,n,0,p)),
          sublist_indices(quad,lambda([y],x = y)),
          if %% = [] then (
             printf(true,
                    concat("~d besitzt keine modulare",
                           "Quadratwurzel modulo ~d")
                    x,p),
             return(-1)) else
             return(map(lambda([y], y-1),%%)))$
(%i50) map(lambda([x], mod(x^2,13)), makelist(n, n, 0, 12));
(%o50) [0,1,4,9,3,12,10,10,12,3,9,4,1]
(%i51) mod_root(11,13);
       11 besitzt keine modulare Quadratwurzel modulo 13
(%o51) -1
(%i52) mod_root(12,13);
(%o52) [5,8]
```

Unser Algorithmus berechnet zunächst Quadrate für alle $a \in \mathbb{Z}_p$. Die Funktion `sublist_indices(LISTE, WAHRHEITSFUNKTION)` liefert alle Indizes aus `LISTE` zurück, für welche die `WAHRHEITSFUNKTION true` zurückgeliefert hat (in unserem Fall der simple Vergleich mit x). Mit `%%` kann in Funktionsblöcken das Ergebnis der letzten Anweisung aufgerufen werden.
Ist also die Liste der Indizes leer, so erfüllt kein $a \in \mathbb{Z}_p$ die Bedingung und x besitzt keine modulare Quadratwurzel in $\mathbb{Z}_p$. Dies wird mit `printf` (*printing formatted*) ausgegeben. In der `printf`-Methode werden die benötigten Variablen und deren Typ mit $\sim$ direkt im Text „angekündigt“, sodass anhand des Typs bereits eine geeignete Formatierung gestaltet werden kann. In unserem Beispiel erwartet `printf` anhand der beiden `~d` im Text zwei Dezimalzahlen, x und p.

Die Methode `concat` fügt die beiden Teilsätze zu einem Satz zusammen[8]. Bevor wir uns einem weiteren Problem der modularen Arithmetik zuwenden, soll zunächst eine optimierte Methode zur Berechnung der modularen Quadratwurzeln vorgestellt werden. Wie man direkt sieht, ist die obige Variante für große Moduli deutlich zu aufwendig. Wir wollen nun einen effizienteren Algorithmus angeben, denn *Maxima* bietet hier ohne Zusatzpakete keine native Implementierung. Betrachten wir zunächst eine wichtige Definition aus der Zahlentheorie:

Definition 5.9. Sei p prim und $a \in \mathbb{Z}$ mit $gcd(a,p) = 1$. Dann ist das **Legendre-Symbol** definiert durch:

$$\left(\frac{a}{p}\right) = \begin{cases} +1, \text{ falls } a \text{ quadratischer Rest } mod\ p \text{ ist} \\ -1, \text{ falls } a \text{ kein quadratischer Rest } mod\ p \text{ ist.} \end{cases}$$

Bemerkung 5.10. Eine Erweiterung des Legendre-Symbols stellt das Jacobi-Symbol $\left(\frac{a}{m}\right)$ dar, welches auch für zusammengesetzte Zahlen m definiert ist. Das Jacobi-Symbol ist aber im Falle m prim äquivalent zum Legendre-Symbol. In den meisten CAS findet sich daher üblicherweise nur eine Funktion zum Bestimmen des Jacobi-Symbols.

Da der Algorithmus sehr umfangreich ist, soll hier nur kurz die Idee dahinter geschildert werden. Auf den nächsten Seiten wird der Algorithmus dann in seinem vollen Umfang dargestellt.
Konstruktion: Um effizient die Quadratwurzeln $r^2 \equiv a\,(mod\,p)$, p prim zu bestimmen, wird zunächst eine Fallunterscheidung für p getroffen:
1. Fall $(p \equiv 3\,(mod\,4))$: Dann gilt direkt $r \equiv \pm a^{\frac{p+1}{4}}\,(mod\,p)$.
2. Fall $(p \equiv 5\,(mod\,8))$: Dann ergibt sich durch $v \equiv (2a)^{\frac{p-5}{8}}\,(mod\,p)$ und $i \equiv 2av^2\,(mod\,p)$ die Lösung $r \equiv \pm av(i-1)\,(mod\,p)$.
3. Fall $(p \equiv 1\,(mod\,8))$: Dann können wir den Algorithmus von Tonelli-Shanks benutzen, um eine Lösung r zu bestimmen.

[8] `concat` kann beliebige Elemente zusammenfügen, nicht nur **Strings**. In unserem Beispiel ist die Verwendung von `concat` eigentlich nicht nötig, da der Satz auch in eine Zeile geschrieben werden kann. Um aber eine gute Anschaulichkeit und einen guten Codestil zu bewahren, wird der Satz hier in 2 Teilsätze aufgetrennt. `concat` bietet die einfachste Möglichkeit, zwei Sätze ohne unnötigen Zwischenraum zusammenzufügen.

Die formale Ablaufvorschrift für den Algorithmus aus Fall 3 wollen wir hier nicht ausführen.
Die Logik hinter der obigen Konstruktionsvorschrift stammt überwiegend direkt aus den Folgerungen und Sätzen rund um das Jacobi-Symbol.

```
(%i53) mod_sqrt(a,p) := block([d,e:0,i,k,q,r,r1,r2,v,w,x,y],
          if jacobi(a,p) # 1 then (
              printf(true,
                  "~d ist kein quadratischer Rest modulo ~d",
                  a,p),
              return(-1)
          )
          else if mod(p,4) = 3 then (
              r1 : power_mod(a,(p+1)/4, p),
              r2 : mod(-r1, p),
              return([r1,r2])
          )
          else if mod(p,8) = 5 then (
              v : power_mod(2·a, (p-5)/8, p),
              i : mod(2·a·v^2, p),
              r1 : mod(a·v·(i-1), p),
              r2 : mod(-r1,p),
              return([r1,r2])
          )
          else if mod(p,8) = 1 then (
              q : p-1,
              while integerp(q/2) do
                  [q,e] : [q/2,e+1],
              i : random(p)^q,
              while power_mod(i, 2^(e-1), p) = 1 do
                  i : random(p)^q,
              [y,r,x] : [i,e,power_mod(a, (q-1)/2, p)],
              v : mod(a·x,p),
              w : mod(v·x,p),
```

Wird auf der nächsten Seite fortgesetzt...

```
            while w # 1 do (
                k : 1,
                while power_mod(w, 2^k, p) # 1 do
                    k : k + 1,
                d : power_mod(y,2^(r-k-1),p),
                y : power_mod(d,2,p),
                [r,v,w] : [k,mod(d·v,p),mod(w·y,p)]
            ),
            [r1,r2] : [v, mod(-v,p)],
            return([r1,r2])
        )
    )$
```

Auch wenn der Algorithmus zunächst sehr komplex und aufwendig erscheint, so kann man doch recht schnell verifizieren, wie effizient er arbeitet:

```
(%i54) p : next_prime(10^1000)$
(%i55) jacobi(7,p);
(%o55) 1
(%i56) [a,b] : mod_sqrt(7,p)$
(%i57) mod(a^2,p);
(%o57) 7
(%i58) mod(b^2, p);
(%o58) 7
```

Dies zeigt zusätzlich sehr deutlich, wie effizient die Funktion `power_mod` die Potenzierung in $\mathbb{Z}_p$ durchführt. Die Anzahl der Ziffern von a und b beträgt jeweils 1000.

Zu obigem Algorithmus bleibt noch anzumerken, dass immer zwei Lösungen erwartet werden, falls eine Lösung existiert. Dies folgt aus der Tatsache, dass eine Zahl n modulo einer Primzahl p entweder keine modulare Quadratwurzel besitzt, oder **genau** zwei.

Anhand der Methode zur Berechnung von modularen Quadratwurzeln in $\mathbb{Z}_p$ soll noch eine Methode für $\mathbb{Z}_{p^2}$ vorgestellt werden, die in einem späteren Abschnitt über das Faktorisieren von Zahlen relevant wird:

Satz 5.11. *Sei $p > 2$ prim und a ein quadratischer Rest modulo p. Betrachte folgende Konstruktionsvorschrift:*

1. *Bestimme x_0 mit $x_0^2 \equiv a \,(mod\, p)$.*
2. *Berechne $x_i \equiv (2x_0)^{-1} \,(mod\, p^2)$.*
3. *Berechne $x = x_i \cdot (x_0^2 + a)$.*
4. *Berechne $r \equiv x \,(mod\, p^2)$.*

Dann gilt $r^2 \equiv a \,(mod\, p^2)$, d.h. r ist modulare Quadratwurzel von a modulo p^2.

Die 4 genannten Schritte lassen sich unter Verwendung unserer `mod_sqrt`-Funktion sehr einfach in *Maxima* überführen:

```
(%i59) mod_p2_sqrt(a,p) := block([x,x0,xi],
          x0 : mod_sqrt(a,p)[1],
          xi : inv_mod(2·x0, p^2),
          x : xi·(x0^2 + a),
          return(mod(x, p^2))
       )$
(%i60) mod_p2_sqrt(16,5);
(%o60) 21
(%i61) mod(21^2, 25);
(%o61) 16
```

5.4 Der diskrete Logarithmus

Wie wir gesehen haben, lassen sich viele algebraische (und auch analytische) Konzepte mit, wenn nötig, einigen Zusatzinformationen, in die modulare Arithmetik überführen. So auch der Logarithmus:

Definition 5.12. Seien $g, h \in \mathbf{F}_p^\times (:= \mathbb{Z}_p^\times, \, p \,\text{prim})$ und sei $L(h)$ die Lösung folgender Gleichung:

$$g^{L(h)} \equiv h \,(mod\, p)$$

Dann nennen wir $L(h)$ den **diskreten Logarithmus** von h modulo p bezüglich g.

Die oben definierte Menge $\mathbf{F}_p^\times$ erspart uns dabei etwas Schreibarbeit. Wir erinnern uns an die Besonderheit, dass $\mathbb{Z}_p$ ein Körper ist, falls p prim ist. Die Einheitengruppe $\mathbf{F}_p^\times$ enthält demnach alle Elemente $\{1, \ldots, p-1\}$, da in einem Körper alle Elemente außer die 0 invertierbar und demnach Einheiten sind. Dass diese Gleichung nicht immer eine Lösung besitzen muss, zeigt folgendes einfaches Beispiel:

```
(%i62) makelist(mod(2^n, 7), n, 1, 10);
(%o62) [2,4,1,2,4,1,2,4,1,2]
```

Hier sehen wir die ersten zehn 2-er Potenzen in $\mathbb{Z}_7$. Offensichtlich zeichnet sich hier ein 3-er Zyklus [2,4,1] ab, der sich mit wachsendem Expontenten immer wiederholt. Dass tatsächlich eine Periodizität vorliegt, zeigt folgende einfache Rechnung:

$$2^0 = 1; 2^1 = 2; 2^2 = 4; 2^3 = 1$$
$$n \in \mathbb{N} : 2^n = 2^{k \cdot 3} \cdot 2^l = (2^3)^k \cdot 2^l = 1^k \cdot 2^l = 2^l \text{ mit } l \in \{0, 1, 2\}$$
$$\Rightarrow 2^n \in \{2, 4, 1\}$$

Die oben bewiesene Eigenschaft der 2-er Potenzen impliziert damit aber auch folgende Aussage:

$$\text{Die Gleichung } 2^k \equiv 5 \,(mod\, 7) \text{ besitzt keine Lösung.}$$

Damit also die Gleichung $g^{L(h)} \equiv h \,(mod\, p)$ aus Definiton 5.12 eine Lösung besitzt, muss noch eine Zusatzbedingung an g gestellt werden:

Definition u. Satz 5.13. Sei $a \in \mathbb{Z}_n^\times$. Die **Ordnung** von a modulo n ist die kleinste Zahl $k > 0$, sodass $a^k \equiv 1 \,(mod\, n)$ ist (**ord(a)**).
Ist $a \in \mathbf{F}_p^\times$ so teilt $ord(a)$ stets $p - 1$.
Gilt für die Ordnung eines Elementes $ord(a) = p - 1$, so nennen wir a **Primitivwurzel (Erzeuger)** modulo p.

Ist ein $a \in \mathbf{F}_p^\times$ Primitivwurzel, so sieht man schnell, dass die Potenzen von a^q, $q \in \{1, \cdots, p-1\}$ jedes Element aus $\mathbf{F}_p^\times$ darstellen, da die kleinste Potenz die wieder zu 1 wird, $p - 1$ ist (daher auch der Name Erzeuger).
Wissen wir also, dass g ein Erzeuger der Gruppe $\mathbf{F}_p^\times$ ist, so können wir mit Sicherheit sagen, dass für alle $h \in \mathbf{F}_p^\times$ ein diskreter Logarithmus $L(h)$ modulo p bezüglich g existiert.

Maxima bietet eine eigene Funktion zur Berechnung des diskreten Logarithmus:

```
(%i63) p : 113$
(%i64) g : zn_primroot(p);
(%o64) 3
(%i65) zn_log(93,g,p);
(%o65) 51
(%i66) mod(3^51, 113);
(%o66) 93
```

Die Funktion `zn_primroot(p)` berechnet die kleinste Primitivwurzel in $\mathbf{F}_p^\times$. Der Aufruf von `zn_log(h,g,p)` löst dann die Gleichung $g^k \equiv h \,(mod\, p)$. Zeile 66 beweist, dass die Funktion korrekt gearbeitet hat.
Würde man einen Algorithmus zum Bestimmen des diskreten Logarithmus implementieren, so könnte man ganz naiv alle Potenzen $1, \ldots, p-1$ durchprobieren, bis man die Lösung gefunden hat. Diese „Brute-Force"-Methode büßt aber mit größer werdendem p sehr schnell an Effizienz ein. Im Abschnitt über Kryptographie werden wir uns deshalb noch einmal mit einem effizienteren Algorithmus zur Berechnung des diskreten Logarithmus beschäftigen.

6 Polynomarithmetik

Bevor wir uns im nächsten Abschnitt mit den direkten Folgen der modularen Arithmetik für die Kryptographie befassen, wollen wir zunächst einen Blick auf *Maxima's* Funktionen zur Polynomarithmetik werfen. Die Arbeit mit Polynomen stellt in vielen Teilgebieten der Algebra eine essentielle Komponente dar. Wir erweitern hier die Definition unseres Rings R um eine Veränderliche x und erhalten somit den Polynomring $R[x]$ der Polynome über R mit der Darstellung $\sum_{i=0}^{n} a_i x^i$, $a_i \in R$, $a_n \neq 0$.
Wir wollen zu Beginn ein paar elementare Funktionen für Polynome kennenlernen:

```
(%i67) f[x] := x^3 + 3·x^2 + 4·x + 5$
(%i68) f[4];
(%o68) 133;
(%i69) g[x] := (x+3)^5;
(%o69) g_x:=(x+3)^5
(%i70) expand(g[x]);
(%o70) x^5 + 15x^4 + 90x^3 + 270x^2 + 405x + 243
(%i71) t : x^2 - 3·x + 2$
(%i72) factor(t);
(%o72) (x-1)(x-2)
```

Mit `p[x] := POLYNOM` definiert man das Polynom als Funktion, sodass man direkt Werte für x einsetzen kann. Wie man in der Eingabezeile 71 sieht, kann *Maxima* aber auch mit Polynomen als einfache Zuweisung arbeiten. Die Funktion `expand(AUSDRUCK)` versucht so weit wie möglich, `AUSDRUCK` zu expandieren, in dem beispielsweise Produkte und Potenzen ausmultipliziert werden. Die Funktion `factor(AUSDRUCK)` arbeitet genau gegenläufig und versucht, `AUSDRUCK` in irreduzibele Elemente über den ganzen Zahlen zu zerlegen. Eine weitere wichtige Funktion bietet `coeff(P,TERM)`. Mit ihr erhält man den zu `TERM` zugehörigen Koeffizienten:

```
(%i73) coeff(t,x);
(%o73) -3
```

Will man eine Liste aller Koeffizienten haben, so muss unter Angabe des Grads des Polynoms eine eigene Funktion definieren:

```
(%i74) all_coef(P,n) := block([],
          map(lambda([t], coeff(P,x,t)), makelist(k,k,n,0,-1))
       )$
(%i75) all_coef(t,2);
(%o75) [1,-3,2]
```

Die Funktion `coeff(P,x,t)` liefert den Koeffizienten des Terms x^t. Die -1 als letztes Argument von `makelist` legt die Schrittweite bei der Erstellung der Liste fest. Hierdurch werden die Zahlen von n bis 0 rückwärts durchlaufen. Anhand der Definition des Polynomrings $R[x]$ lassen sich auch Eigenschaften von $\mathbb{Z}$ auf $R[x]$ übertragen, insbesondere auch die Division mit Rest:

Satz 6.1. *Seien $\mathbb{K}$ Körper und $a(x), b(x) \in \mathbb{K}[x]$. Dann existiert genau ein Paar $(q(x), r(x)) \in \mathbb{K}[x]$ mit:*

$$a(x) = q(x)b(x) + r(x) \text{ und } deg(r(x), x) < deg(b(x), x)$$

*$deg(p(x), x)$ beschreibt dabei den **Grad** von $p(x)$ in x, d.h.:*
Ist $p(x) = \sum_{i=0}^{n} a_i x^i$, so ist $deg(p(x), x) = max_i\{a_i \neq 0\}$, also die größte x-Potenz von $p(x)$.

In *Maxima* erhält man $q(x)$ und $r(x)$ mit den Funktionen `quotient` resp. `remainder`:

```
(%i76) p : x^6 + x^3 + 2$
(%i77) m : x^2 + 4·x + 3$
(%i78) quotient(p,m);
(%o78) x^4-4x^3+13x^2-39x+117
(%i79) remainder(p,m);
(%o79) -351x - 349
```

Die Division mit Rest impliziert auch einen größten gemeinsamen Teiler in $\mathbb{K}[x]$ sowie die Existenz von Bézoutkoeffizienten. Da wir uns hier in $\mathbb{K}[x]$ befinden, kann es passieren, dass der größte gemeinsame Teiler das konstante Polynom 1 ist. Dies ist der Fall, wenn die beiden Polynome keinen gemeinsamen Faktor besitzen oder eins der Polynome/beide Polynome nicht über $\mathbb{K}[x]$ faktorisierbar ist/sind:

```
(%i80) ezgcd(p,m);
(%o80) [1,x^6+x^3+2,x^2+4x+3]
```

Die Funktion `ezgcd` bestimmt zunächst den größten gemeinsamen Teiler d von $p(x)$ und $m(x)$ und teilt die Polynome dann durch d. Die Ergebnisse der Division sind der zweite und dritte Listeneintrag der Ergebnisliste.
Diese kurze Abschnitt sollte eine kleine Einführung in *Maxima's* Umgang mit Polynomen geben. Natürlich existiert noch ein enormes Spektrum an weiteren Funktionen, beispielsweise zum Vereinfachen von rationalen Polynomen, für die Polynomarithmetik an sich und die Polynomarithmetik in Restklassenringen, diese sollen aber hier nicht weiter behandelt werden.

7 Kryptographie

In diesem Abschnitt wollen wir uns mit den algebraischen Anwendungen in der Kryptographie beschäftigen. Dabei sollen verschiedene Kryptosysteme vorgestellt werden und ein besonderer Fokus darauf gelegt werden, wie wichtig alte und moderne Algorithmen der Computeralgebra sind, um die Sicherheit dieser Systeme einschätzen zu können.

7.1 Einführung und Private-Key-Verfahren

Wir beginnen zunächst mit der Definition einiger wichtiger Begriffe:

Definition 7.1. Sei $\mathcal{A}$ eine nicht-leere, endliche Menge und $(a_1, a_2, \ldots, a_n) \in \mathcal{A}^n =: \underbrace{\mathcal{A} \times \cdots \times \mathcal{A}}_{\text{n-mal}}$ ein n-Tupel aus dem n-fachen Produkt von $\mathcal{A}$. Dann nennen wir $\mathcal{A}$ ein **Alphabet** und $(a_1, a_2, \ldots, a_n)$ einen **Text der Länge n** über $\mathcal{A}$. Die **Wortmenge** $\mathcal{A}^*$ sind alle Texte beliebiger Länge über $\mathcal{A}$, d.h. $\mathcal{A}^* = \cup_{n\in\mathbb{N}}\mathcal{A}^n$.

Der Einfachheit halber werden wir in Zukunft die Menge $\mathcal{A} = \{A, B, \ldots, Z\}$ das **natürliche Alphabet** nennen. Wenn wir mit dem natürlichen Alphabet rechnen müssen, weisen wir jedem Buchstaben seinen zugehörigen Platz im Alphabet zu, **beginnend bei 0**, d.h. $\mathcal{A} = \{A, B, \ldots, Z\} = \{0, 1, \ldots, 25\}$.
Außerdem muss klar sein, was wir unter einer Verschlüsselung verstehen:

Definition 7.2. Seien $\mathcal{A}, \mathcal{B}$ zwei Alphabete und $\mathcal{A}^*, \mathcal{B}^*$ die zugehörigen Wortmengen. Ein **Private-Key-Verfahren** ist ein Tupel $(\mathcal{W}, \mathcal{C}, \mathcal{K}, E, D)$ mit folgenden Definitionen:

- $\mathcal{W}$ ist die Menge der zulässigen **Klartexte**, d.h. die Menge der erlaubten Texte, die verschlüsselt werden können (meistens $\mathcal{P} = \mathcal{A}^*$).
- $\mathcal{C}$ ist die Menge der **Chiffretexte**, d.h. die Menge der Texte, die durch die Verschlüsselung entstehen können (meistens $\mathcal{C} = \mathcal{B}^*$).
- $\mathcal{K}$ ist der **Schlüsselraum**, d.h. die Menge aller zulässigen Schlüssel.
- $E : \mathcal{W} \times \mathcal{K} \to \mathcal{C}$, $w \mapsto E(p) := c$ heißt **Verschlüsselungsfunktion**.
- $D : \mathcal{C} \times \mathcal{K} \to \mathcal{W}$, $c \mapsto D(c) := w$ heißt **Entschlüsselungsfunktion**.

Wird die Verschlüsselungsfunktion E (*encrypt*) auf einen Text $w \in \mathcal{P}$ angewandt, nennen wir dies die **Verschlüsselung** von w.
Wird die Entschlüsselungsfunktion D (*decrypt*) auf einen Text $c \in \mathcal{C}$ angewandt, nennen wir dies die **Entschlüsselung** von c.

Nachdem die wichtigsten Begriffe geklärt sind, wollen wir uns einigen historischen Beispielen samt *Maxima*-Implementierung widmen:

Die Vigenère-Chiffre: Seien $\mathcal{A}$ das natürliche Alphabet, $\mathcal{W} = \mathcal{C} = \mathcal{A}^*$ und $s \in \mathbb{N}$. Wir nennen die bijektive Abbildung $\sigma_k : \mathcal{A} \to \mathcal{A}$, $a \mapsto a + k \,(mod\, 26)$ **Links-Shift (s $\geq$ 0)** bzw. **Rechts-Shift (s $<$ 0)**. Als Schlüsselraum wählen wir $\mathcal{K} = \{0, \ldots, 25\}^s$. Für einen Schlüssel $k = (k_1, \ldots, k_s)$ und einen Text $w = w_1 \cdots w_n \in \mathcal{W}$ der Länge n definieren wir:

$$\begin{aligned} E(k, w) &= E(k, w_1) \cdots E(k, w_n) \\ &= \sigma_{k_1}(w_1) \cdots \sigma_{k_s}(w_s)\sigma_{k_1}(w_{s+1}) \cdots \sigma_{k_s}(w_{2s}) \cdots \\ &= \sigma_k(w) := c \end{aligned}$$

und für $c = c_1 \cdots c_n \in \mathcal{C}$:

$$\begin{aligned} D(k, x) &= D(k, c_1) \cdots D(k, c_n) \\ &= \sigma_{-k_1}(c_1) \cdots \sigma_{-k_s}(c_s)\sigma_{-k_1}(c_{s+1}) \cdots \sigma_{-k_s}(c_{2s}) \cdots \\ &= \sigma_{-k}(c) := w \end{aligned}$$

Wir ordnen also jedem Buchstaben aus w sein Chiffrat basierend auf dem Schlüssel k zu. Überschreitet die Länge von w dabei die Länge von k, wird der Schlüssel wieder von Vorne angewandt.
Folgendes Beispiel soll die Funktionsweise verdeutlichen ($k = (1, 3, 5)$):

Wort	KRYPTOGRAPHIE												
Zerlegung	K	R	Y	P	T	O	G	R	A	P	H	I	E
+ Schlüssel	1	3	5	1	3	5	1	3	5	1	3	5	1
Chiffrat	L	U	D	Q	W	T	H	U	F	Q	K	N	F

Um die Vigenère-Chiffre in *Maxima* zu implementieren benötigen wir zunächst Funktionen zur Umwandlung Buchstabe $\rightleftarrows$ Zahl:

```
(%i81) text_to_num(s) := block([],
          split(s,""),
          map(lambda([c], cint(c) - 65), %%)
       )$
(%i82) num_to_text(l) := block([],
          map(lambda([i], ascii(i + 65)), l),
          simplode(%%)
       )$
(%i83) text_to_num("KRYPTOGRAPHIE");
(%o83) [10,17,24,15,19,14,6,17,0,15,7,8,4]
(%i84) num_to_text(%);
(%o84) KRYPTOGRAPHIE
```

Die Funktion `split(s,"")` trennt das Wort s in einzelne Buchstaben und liefert das Ergebnis als Liste zurück[9]. `cint(CHAR)` liefert den zu `CHAR` gehörigen ASCII-Code. Da die Großbuchstaben in dem ASCII-Code, den *Maxima* verwendet, bei 65 beginnen, muss das Ergebnis noch verrechnet werden. Andersherum liefert die Funktion `ascii(CODE)` den zu `CODE` gehörigen Charakter. Die Funktion `simplode` fügt die Liste aus Buchstaben wieder zu einem Wort zusammen.
Dies führt uns zu folgendem Programm:

```
(%i85) encode_vigenere(k,w) := block([],
          w : text_to_num(w),
          k : flatten(makelist(k,
                               ceiling(length(w)/length(k)))),
          k : firstn(k, length(w)),
          num_to_text(mod(w+k, 26))
       )$
```

[9]Anstelle von `""` hätte man hier auch z.B ein Leerzeichen `" "` eingeben können, dann würde s nur bei allen Leerzeichen getrennt werden.

```
(%i86) decode_vigenere(k,c) := block([],
          c : text_to_num(c),
          k : flatten(makelist(-k,
                               ceiling(length(c)/length(k)))),
          k : firstn(k, length(c)),
          num_to_text(mod(c+k, 26))
       )$
(%i87) encode_vigenere([1,3,5], "KRYPTOGRAPHIE");
(%o87) LUDQWTHUFQKNF
(%i88) decode_vigenere([1,3,5], %);
(%o88) KRYPTOGRAPHIE
```

Die Caesar-Chiffre: Sie ist wohl eine der ältesten und bekanntesten Verschlüsselungsarten. Sie ist eine Abwandlung der Vigenère-Chiffre mit einer Schlüssellänge $s = 1$:

```
(%i89) encode_vigenere([3], "CAESARCODE");
(%o89) FDHVDUFRGH
```

Dass die Caesar-Chiffre in der modernen Kryptographie unbrauchbar ist lässt sich durch einen einfachen Aufruf schnell zeigen:

```
(%i90) map(lambda([i],
           num_to_text(mod(text_to_num(%) - i, 26))),
           makelist(n,n,0,25));
(%o90) [FDHVDUFRGH,ECGUCTEQFG,DBFTBSDPEF,CAESARCODE,
        BZDRZQBNCD,AYCQYPAMBC,...]
```

Einfaches durchprobieren der Möglichkeiten liefert uns direkt den unverschlüsselten Text.
Eine weitere Abwandlung der Vigenère-Chiffre liefert die Vernam-Chiffre. In diesem Fall ist die Schlüssellänge s gleich der Textlänge n. Diese Art der Verschlüsselungen nennt man **monoalphabetische Substitutionschiffren**.

7.2 Asymmetrische Verschlüsselungsverfahren

Auch wenn gute Private-Key-Verfahren eine der sichersten Varianten der Verschlüsselung darstellen, sind sie in der Praxis meistens eher unpraktikabel, da die Schlüssel „per Hand“ ausgetauscht werden müssen. Besonders im In-

ternet, wo die Kryptographie eine zentrale Rolle spielt, ist dies unmöglich. Um trotzdem ein gutes Maß an Sicherheit gewährleisten zu können, existieren die **Public-Key-Verfahren**, welche öffentliche und private Schlüssel miteinander verbinden.
Folgende Definition stellt eines der bekanntesten Public-Key-Verfahren vor:

Definition 7.3. (RSA-Verfahren) Eine Nachricht $N \in \mathbb{N}$ soll verschlüsselt werden. Das RSA-Verfahren operiert nach folgendem Protokoll:

(Empfänger)

- Wähle zwei zufällige, genügend große Primzahlen p und q.
- Bestimme $n = p \cdot q$.
- Bestimme $\varphi = (p-1) \cdot (q-1)$.
- Bestimme ein zufälliges $e \in \mathbb{N}$ mit $e < n$ und $gcd(e, \varphi) = 1$.
- Das Paar (e, n) wird öffentlich gemacht. (öffentlicher Schlüssel)
- Die Zahl $d = e^{-1} \, (mod\, n)$ wird geheim gehalten. (privater Schlüssel)
- Die Zahlen p, q und φ werden aus Sicherheitsgründen gelöscht.

(Absender)

- Die Nachricht N wird mit $C = E_{(e,n)}(N) = mod(N^e, n)$ verschlüsselt. (Falls $N \geq n$ ist, wird die Nachricht vorher in geeignete Blöcke zerlegt.)

(Empfänger)

- Die Nachricht kann mit $N = D_{(e,n)}(C) = mod(C^d, n)$ entschlüsselt werden.

Um das RSA-Verfahren in *Maxima* implementieren zu können, benötigen wir zunächst einige Hilfsfunktionen für die Umwandlung der Texte in die Zahl N:

```
(%i91) text_to_N(s) := block([len,N : 0],
          s : map(lambda([c], cint(c)), split(s,"")),
          len : length(s),
          for i : 1 thru len do (
              N : N + 1000^(i-1) · lastn(s,1),
              s : firstn(s, length(s)-1)
          ),
          N[1]
       )$
(%i92) text_to_N("Kryptographie");
(%o92) 751141211121161111031140971121041051 01
(%i93) N_to_text(N) := block([s : []],
          while N # 0 do (
              s : append(s, [mod(N, 1000)]),
              N : (N - mod(N, 1000)) / 1000
          ),
          map(lambda([i], ascii(i)), reverse(s)),
          simplode(%%)
       )$
(%i94) N_to_text(''%o92);
(%o94) Kryptographie
```

Die Schleifenbedingung `N # 0` ist dabei äquivalent zur Überprüfung $N \neq 0$. Die Funktion `reverse(LISTE)` erzeugt eine umgekehrte Version von `LISTE`.

Nun benötigen wir noch die Ver-/Entschlüsselungsfunktion:

```
(%i95) encrypt_rsa(s) := power_mod(text_to_N(s), e, n)$
(%i96) decrypt_rsa(C) := N_to_text(power_mod(C, d, n))$
```

Die Funktion `powermod(a,n,m)` liefert eine effiziente Implementierung für die Rechnung $a^n \,(mod\, m)$, dabei liefert der Aufruf `powermod(a,-1,m)` das modulare Inverse von a modulo m. (Die Funktion wurde bereits einmal im Kontext der modularen Quadratwurzeln verwendet)
Mit allen nötigen Hilfsfunktionen ausgestattet, können wir nun das RSA-Modul implementieren:

```
(%i97) rsa_module() := block([],
         p : next_prime(random(10^100)),
         q : next_prime(random(10^100)),
         n : p · q,
         e : phi : (p-1) · (q-1),
         while gcd(e, phi) # 1 do
            e : next_prime(random(10^50)),
         d : power_mod(e, -1, phi),
         kill(p, q, phi)
         return("RSA-Module initialized")
       )$
```

Die Funktion `kill(VAR_1, ..., VAR_n)` löscht die Variablen `VAR_1, ..., VAR_n` aus der aktuellen *Maxima*-Sitzung und gibt deren Speicher für zukünftige Rechnungen wieder frei[10].
Mit unserem RSA-Modul können wir nun beginnen, Nachrichten zu verschlüsseln:

```
(%i98) message : "Execute Order 66"$
(%i99) set_random_state(seed)$
(%i100) rsa_module();
(%o100) RSA-Module initialized;
(%i101) [e,n];
(%o101) [60037114172868838252528941048419617783718312434069,
        152992780035440866392276030606[139Ziffern]
        614543240523012234142022119287]
(%i102) encrypt_rsa(message);
(%o102) 42578904647177086809313337166 2[139 Ziffern]
       928934764214484623960358504322
(%i103) decrypt_rsa(%);
(%o103) Execute Order 66
```

Die Sicherheit des RSA-Verfahrens resultiert offensichtlich daraus, dass, wie bereits besprochen, die Faktorisierung für genügend große Zahlen n mit zwei

[10]Als Variablen können auch selbst erstellte Funktionen zählen. `kill` kann prinzipiell alles, was vom Nutzer definiert ist, löschen.

großen Primfaktoren beinahe unmöglich ist / eine absurde Laufzeit hat[11].
Trotz des hohen Sicherheitsstandards des RSA-Verfahrens werden auch die Algorithmen zur Faktorisierung immer besser und stellen somit mit steigender Rechenleistung ein immer höheres Sicherheitsrisiko dar. Wir wollen uns im Zuge dessen noch einmal mit der Faktorisierung beschäftigen:

7.2.1 Faktorisierung nach der Methode von Fermat

Wir haben bereits gezeigt, dass unser naiver Algorithmus der optimierten Probedivision zur Bestimmung der Primfaktorzerlegung mit immer größer werdenden Zahlen sehr schnell an seine Grenzen gerät. Diese Erkenntnis kommt nicht besonders überraschend, wir müssen nur einen Blick auf die Definition unseres RSA-Moduls werfen:
Die Zahl n im Modul ist bewusst so konstruiert, dass sie zwei große Primfaktoren p und q enthält. Um das n aus unserem RSA-Beispiel zu faktorisieren, müsste der Algorithmus im Fall $p < q$ die Zahl n durch alle Primzahlen bis zur Größenordnung 10^{100} dividieren, um p zu finden. Dies ist selbst auf den modernsten Computern eine, in einem akzeptablen Zeitraum, unlösbare Aufgabe.
Auch wenn das Problem der Faktorisierung im Sinne der Komplexitätstheorie möglicherweise nie einen „effizienten“ Algorithmus zur Lösung besitzen wird, so existieren doch Verfahren, welche deutlich effizienter arbeiten als unser Algorithmus, wie der Vergleich mit *Maxima's* `ifactors`-Funktion gezeigt hat[12].
Viele dieser Verfahren bedienen sich geschickt den Erkenntnissen der Zahlentheorie, um die Algorithmen zu optimieren. Eines dieser Verfahren ist das **Quadratische Sieb**, welches in den folgenden Seiten besprochen und implementiert werden soll.
Zunächst wenden wir uns aber einer etwas älteren Faktorisierungsmethode von Fermat zu, welche den Grundstein für das Quadratische Sieb legt:

[11] Auch wenn der Algorithmus die Aufgabe der Primfaktorzerlegung theoretisch immer lösen kann, so sind beispielsweise potentielle Laufzeiten von 100 Jahren+ eher ungefährlich für moderne Sicherheitssysteme.

[12] Der Verweis auf die Komplexitätstheorie spielt hier darauf an, dass zwar bekannt ist, dass das Problem der Faktorisierung in NP liegt, aber nicht, ob nicht doch ein Algorithmus zur Lösung mit polynomieller Laufzeit existiert.

Voraussetzung: Wir haben uns in Abschnitt 5 bereits mit modularen Quadratwurzeln befasst. Eine wichtige Erkenntnis war, dass diese, wenn sie überhaupt existieren, nicht eindeutig sein müssen. Im Falle einer zusammengesetzten Zahl N kann es sogar bis zu 4 modulare Quadratwurzeln $(\pm b_1, \pm b_2)$ als Lösung der Gleichung $b^2 \equiv a\,(mod\,N)$ geben. Wir nehmen an, dass $N = p{\cdot}q$ für zwei Primzahlen p, q ist. Der chinesische Restsatz verrät uns, dass für die Lösungen $\pm b_1 \in \mathbb{Z}_p$ und $\pm b_2 \in \mathbb{Z}_q$ gelten muss (oder umgekehrt)[13]. Hat man nun Kenntnis über zwei Zahlen x, y mit $x^2 \equiv y^2 \equiv a\,(mod\,N)$ und $x \not\equiv \pm y\,(mod\,N)$, kann man eine Zerlegung von N konstruieren, denn aus $x^2 \equiv y^2\,(mod\,N)$ folgt direkt $(x+y)(x-y) \equiv 0\,(mod\,N)$. Da N nach Voraussetzung keinen dieser Faktoren teilt, erhalten wir mit $gcd(x+y, N)$ einen nicht-trivialen Teiler von N.
Die Voraussetzungen implizieren eine bereits optimierte Version von folgendem Verfahren:

Definition 7.4. (Faktorisierungsverfahren von Fermat) Sei $N \in \mathbb{N}$ ungerade und zusammengesetzt. Konstruiere eine Faktorisierung von N nach folgender Vorschrift:

- Wähle $x_0 := \lceil \sqrt{N} \rceil$
- Bilde für $x = x_0, x_0+1, x_0+2, \ldots$ nacheinander die Differenzen $x^2 - N$, bis eine Quadratzahl entsteht.
- Dann ist $x^2 - N = y^2$ und es folgt daraus die Zerlegung:

$$N = x^2 - y^2 = (x+y)(x-y)$$

Wir wollen dieses Verfahren in *Maxima* implementieren:

```
(%i104) factor_fermat(N) := block([x : ceiling(sqrt(N))],
          while not integerp(sqrt(x^2 - N)) do
             x : x+1,
          return([x + (sqrt(x^2 - N)), x - (sqrt(x^2 - N))])
       )$
```

[13]Der chinesische Restsatz impliziert eine Isomorphie $\mathbb{Z}_N \cong \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_n}$, falls N die Primfaktorzerlegung $N = p_1 \cdots p_n$ besitzt.

Das Fermat-Verfahren ist für eine große Zahl N nur dann geeignet, wenn die Primzahlen, aus denen sie sich zusammensetzt, sehr nah beieinander liegen. Dann ist der Algorithmus aber erstaunlich effizient:

```
(%i105) [p,q] : [next_prime(10^20),next_prime(10^21)]$
(%i106) N : p · q;
(%o106) 100000000000000000507000000000000000004563
(%i107) showtime: true$
(%i108) ifactors(N);
        Evaluation took 6.5220 seconds (6.5940 elapsed)
        using 2761.449 MB.
(%o108) [[100000000000000000039,1],[100000000000000000117,1]]
(%i109) factor_fermat(N);
        (Muss abgebrochen werden)
(%i110) [p,q] : [next_prime(10^20),
                 next_prime(next_prime(10^20))]$
(%i111) N : p · q;
(%o111) 100000000000000000168000000000000000005031
(%i112) ifactors(N);
        Evaluation took 18.7360 seconds (18.8600 elapsed)
        using 8069.978 MB.
(%o112) [[100000000000000000039,1],[100000000000000000129,1]]
(%i113) factor_fermat(N);
       Evaluation took 1.4320 seconds (1.4490 elapsed)
       using 296.997 MB.
(%o113) [100000000000000000129,100000000000000000039]
(%i114) showtime: false$
```

Die Erweiterung des Fermat'schen Faktorisierungsverfahrens betrachtet eine Reihe von Kongruenzen $u_i^2 \equiv v_i \,(mod\, N)$. Bilden eine Auswahl $v_1 \cdots v_n := y^2$ ein Quadrat, so ergibt sich eine Kongruenz $u_1 \cdots u_n =: x^2 \equiv y^2 \,(mod\, N)$. Um die richtigen v_k zu finden, muss man nur die Primfaktorzerlegung der v_k betrachten und darauf achten, dass die Exponenten der Primfaktoren im Produkt immer gerade sind. Hat man die nötigen v_k gefunden, kann man x und y berechnen und mit $gcd(x + y, N)$ einen nicht-trivialen Teiler von N bestimmen. Diese Idee führt uns nun zum Quadratischen Sieb:

7.2.2 Faktorisierung mit dem Quadratischen Sieb

Um die Auswahl der v_k für die kommenden Algorithmen etwas konkreter zu gestalten, beginnen wir mit folgender Definition:

Definition 7.5. (Auswahl für Kongruenzen) Um genügend quadratische Reste $u_i^2 \equiv v_i \,(mod\, N)$ zu erhalten, die sich leicht faktorisieren lassen, legen wir den Kongruenzen eine **Faktorbasis** $\mathcal{F}$ zugrunde, mit

$$\mathcal{F} := \{-1, 2, p_2, \ldots, p_{n-1}\} \text{ und } 2 < p_i \leq L \in \mathbb{N} \text{ für } i = 2, \ldots, n-1$$

Bei der Auswahl der Kongruenzen betrachten wir nur die v_i, welche sich vollständig in Elemente aus $\mathcal{F}$ zerlegen lassen.

Angenommen, wir haben ein $v_i = \prod_{0 \leq \nu < n} p_\nu^{\alpha_{i,\nu}}$ mit $p_\nu \in \mathcal{F}$ gefunden. Da für die Produktbildung nur wichtig ist, ob die $\alpha_{i,\nu}$ gerade sind, bilden wir zu jedem der $\alpha_{i,\nu}$ ein $\beta_{i,\nu} := \alpha_{i,\nu} \,(mod\, 2) \in \mathbf{F}_2$ und speichern den Vektor:

$$\vec{\beta}_i = (\beta_{i,0}, \ldots, \beta_{i,n-1}) \in \mathbf{F}_2^n \tag{1}$$

Bemerkung 7.6. Maximal n der β_i können linear unabhängig sein. Spätestens beim Fund von v_{n+1} muss demnach $\vec{\beta}_{i,n+1}$ von vorher berechneten $\vec{\beta}_{i,j}$ linear abhängig sein, d.h. es existieren Indizes $i_1, \ldots, i_r$ mit:

$$\vec{\beta}_{i,1} + \cdots + \vec{\beta}_{i,r} = \vec{0} \in \mathbf{F}_2^n$$

Daraus folgt, dass $\prod_{k=1}^{r} v_{i,k} =: y^2$ eine Quadratzahl ist.

Das Auswahlkriterium für die Kongruenzen liefert uns zwar eine gute Möglichkeit zu entscheiden, welche v_i's für das Faktorisierungsverfahren günstig sind, rollt aber auch gleichzeitig ein altes Problem wieder auf. Um zu entscheiden, ob ein v_i das Auswahlkriterium erfüllt, müssten wir uns wieder der Probedivision bedienen, um zu sehen, ob v_i ausschließlich in Primfaktoren aus der Faktorbasis zerfällt. An dieser Stelle setzt das Quadratische Sieb ein:

Definition 7.7. (Quadratisches Sieb) Werden die quadratischen Reste von einem quadratischen Polynom $Q(x)$ erzeugt (Fermat: $Q(x) = x^2 - N$), so ist $Q(\xi) = \eta$ genau dann durch eine Primzahl p teilbar, falls $Q(\xi) \equiv 0 \,(mod\, p)$ ist. Diese Gleichung hat zwei Lösungen x_1 und x_2 modulo p, also sind alle $Q(x_1 + kp)$ und $Q(x_2 + kp)$ mit $k \in \mathbb{Z}$ durch p teilbar.
Dies spart die Division aller anderen Werte $Q(x)$ durch p.

Ein einzelnes Polynom zum Erzeugen der quadratischen Reste birgt dabei auch gewisse Risiken. Betrachten wir $Q(x)$ beispielsweise auf einem Intervall $|x - \sqrt{N}| \leq M$, so kann es passieren, dass zu wenige quadratische Reste komplett in Elemente aus der Faktorbasis zerfallen. Verändert man eine der 3 Komponenten, so wird entweder die Rechnung deutlich aufwendiger, oder die Wahrscheinlichkeit eines geeigneten quadratischen Restes noch geringer. Dies führt uns zur Konstruktion eines optimierten Quadratischen Siebs:

Definition 7.8. (Multipolynomiales Quadratisches Sieb (MPQS)) Sei $Q(x) = ax^2 + 2bx + c$; $a, b, c \in \mathbb{Z}$ mit der Diskriminante $\Delta := b^2 - ac = N$. Dann sind alle Werte:

$$aQ(x) = (ax + b)^2 - (b^2 - ac) = (ax + b)^2 - N$$

quadratische Reste modulo N. Wählen wir $a = q^2$ mit q prim und $q \nmid N$, so sind auch alle Werte $Q(x)$ quadratische Reste. Dies liefert eine zusätzliche Einschränkung für die Faktorbasis:
Gilt für eine Primzahl $p > 2$ dass $p|Q(x)$, so folgt:

$$(ax + b)^2 \equiv N \,(mod\, p) \Rightarrow \left(\frac{N}{p}\right) = +1$$

d.h. N ist quadratischer Rest modulo p. Wir konstruieren die zusätzlichen Elemente der Faktorbasis $\mathcal{F}$ also nur aus Primzahlen $p > 2$, welche die Bedingung $\left(\frac{N}{p}\right) = +1$ erfüllen.

Daraus können wir zunächst eine Funktion zum Erstellen der Faktorbasis gewinnen:

```
(%i115) gen_factorbase(N,len) := block([F : [-1,2],k,p : 2],
          for k : 3 thru len do (
             p : next_prime(p),
             while jacobi(N,p) #1 do
                p : next_prime(p),
             F : append(F, [p])
          ),
          return(F)
       )$
(%i116) Fb : gen_factorbase(400003, 10);
(%o116) [1,2,3,7,29,31,37,43,71,73]
```

Als Nächstes benötigen wir die modularen Quadratwurzeln von N modulo den Elementen aus $\mathcal{F}$. Hierzu greifen wir auf unsere effiziente Implementierung aus Abschnitt 5.3 zurück:

```
(%i117) N_roots_F(N,F) := block([k,roots : [1,1]],
          for k : 3 thru length(F) do
             roots : append(roots, [mod_sqrt(N,F[k])[1]]),
          return(roots)
       )$
(%i118) N_roots_F(400003, Fb);
(%o118) [1,1,1,4,8,14,25,24,29,67]
```

Konstruktion der Polynome: Um die Polynome zu konstruieren, müssen wir die Gleichung $b^2 - ac = N$ der Diskriminante lösen, dabei folgt daraus $b^2 \equiv N \,(mod\, a)$. Wählen wir wie definiert $a = q^2$ mit $p > 2$ prim, so muss $b^2 \equiv N \,(mod\, q^2)$ gelten, N muss also quadratischer Rest modulo q^2 sein. Wählen wir die Lösung $0 \leq b < a = q^2$, so folgt aus $b^2 \equiv N \,(mod\, a)$, dass $a|b^2 - N$ gilt und wir erhalten mit $c := \frac{b^2-N}{a}$ eine Lösung der Ausgangsgleichung.

```
(%i119) make_pol(N,q) := block([a,b,c,qq,tmp],
           if is(primep(q)) then
              qq : q else
              qq : next_prime(q),
           if is(qq = 2) then
              qq : next_prime(qq),
           while jacobi(N,qq) # 1 do
              qq : next_prime(qq),
           a : qq^2,
           b : mod_p2_sqrt(N,qq),
           c : floor((b^2-N) / a),
           return([qq,a,b,c])
       )$
(%i120) make_pol(400003, 4);
(%o120) [7,49,4,-8163]
(%i121) %[3]^2 - %[2]·%[4]
(%o121) 400003
```

Siebtechnik: Zunächst müssen wir ein Intervall wählen, über dem wir diejenigen x_i herausssieben, für die $Q(x_i)$ komplett in Elemente aus der Faktorbasis zerfällt.
Ist x_0 das Minimum von $Q(x)$, so wollen wir die Schranke für das Intervall $|x - x_0| \leq M$ so wählen, dass $M \approx |Q(x_0)|$. Betrachten wir zunächst die Gleichung $aQ(x) = (ax + b)^2 - N$. Ist ein $2 < p \in \mathcal{F}$, so gilt $p|Q(x)$ genau dann, wenn $(ax + b)^2 \equiv N \,(mod\, p) \Leftrightarrow ax + b \equiv \pm r \,(mod\, p)$ ist, wobei r die modulare Quadratwurzel von N modulo p ist. Seien nun ξ_1, ξ_2 mit $\xi_{1,2} \equiv a^{-1}(\pm r - b) \,(mod\, p)$, so ist $Q(x)$ für $x = \xi_1 + kp$ und $x = \xi_2 + kp$ mit $k \in \mathbb{Z}$ durch p teilbar. Anstatt nun für alle x den Wert $Q(x)$ durch p zu teilen, verfahren wir wie folgt: Nach der Initialisierung des Siebintervalls addiere für jedes $p \in \mathcal{F}$ an den Stellen $x = \xi_\nu^{(p)} + kp$ den Wert $log(p)$. Ist $Q(x)$ vollständig in Elemente aus $\mathcal{F}$ zerlegbar, so addieren sich die Logarithmen zu $log(|Q(x)|)$. Wählen wir aufgrund der Rechenungenauigkeit nun eine Toleranzgrenze
$t > 0$, so können wir als potentielle Werte $Q(x)$ diejenigen x in Betracht ziehen, für welche die Summe der Logarithmen den Wert $log(|Q(x)|)$ bis auf die Toleranz t erreicht. Da sich der Wert $log(|Q(x)|)$ nur sehr langsam mit x ändert, können wir ihn auch auf dem gesamten Siebintervall durch sein Maximum ersetzen, was zusätzliche Rechnungen einspart.
Bilden des Logarithmusvektors: Die Logarithmen müssen nicht exakt sein, daher können wir die Rundung von $log(p) \cdot scale$ auf eine Ganzzahl betrachten, wobei $scale$ ein Skalierungsfaktor ist, hier $scale = 128$. Da der Logarithmusvektor immer wieder benötigt wird und nicht ständig neu berechnet werden soll, benutzen wir zur Implementierung *Maxima's* Rememberprogrammierung:

```
(%i122) scale : 128$
(%i123) log_F[F] := block([k,lg_F : [0]],
           for k : 2 thru length(F) do
              lg_F : append(lg_F,
                            [round(ev(log(F[k]),numer)·scale)]),
           return(lg_F)
        )$
```

Wie im Beispiel für die rekursive Berechnung der Fibonacci-Zahlen verwen-

den wir eckige Klammern `log_F[F]` anstatt runder Klammern `log_F(F)`. Wird diese Funktion nun einmal auf eine Faktorbasis $\mathcal{F}$ angewandt, so wird das Ergebnis des Aufrufs `log_F[`$\mathcal{F}$`]` gespeichert und beim zweiten Aufruf von `log_F[`$\mathcal{F}$`]` abgerufen anstatt nochmal neu berechnet. Der Aufruf `ev(log(...),numer)` sorgt dafür, dass der Logarithmus als Gleitkommazahl dargestellt wird und somit gerundet werden kann.

Algorithmus: Wir wollen uns nun, nach der geleisteten Vorarbeit, dem eigentlichen Algorithmus zuwenden. Da dieser recht komplex ist und sich über mehrere Funktionen erstreckt, wird seine Beschreibung in Unterabschnitte eingeteilt, in denen die Funktionsweise der jeweiligen Funktionen detailliert beschrieben wird.

1. Initialisierung der Variablen: Zunächst werden alle Variablen definiert, die von mehreren Funktionen des Algorithmus benutzt werden müssen. Wie beim RSA-Modul werden diese Variablen global definiert und lassen sich somit ohne Einschränkung im aktuellen *Maxima*-Kernel abrufen:

```
(%i124) QS_init(N) := block([blen,i],
           Num : N,
           blen : bit_length(N),
           Flen : max(8, floor(blen^2 / 32)),
           Srange : blen·256,
           Startq : isqrt(floor(isqrt(2·N) / Srange)),
           Fbas : gen_factorbase(N,Flen),
           Rvec : N_roots_F(N,Fbas),
           Lvec : log_F[Fbas],
           Mat : ematrix(Flen+1,2·Flen+1,0,1,1),
           Pivot : [],
           for i : 1 thru Flen do
              Pivot : append(Pivot, [i]),
           CurRow : 1,
           Count1 : 0, Count2 : 0,
           Stack : [],
           Sieve : makelist(0,n,2·Srange),
           Vvec : makelist(0,n,Flen+1),
           Uvec : makelist(0,n,Flen+1),
           return(Flen)
       )$
```

Da diese Variablen immer wieder in den kommenden Funktionen auftauchen, sollen sie hier im Detail erklärt werden:

Num: Die zu faktorisierende Zahl N.

blen: Die Bit-Länge der Zahl N.

(Die Funktion `bit_length(n)` liefert die Anzahl der Bits zurück, die nötig sind, um n im Binärsystem zu repräsentieren (**Bit-Länge**).)

Flen: Die vorgegebene Anzahl der Elemente, die in der Faktorbasis enthalten sein sollen.

Srange: Die Länge der Hälfte des Siebintervalls (das M aus der Erklärung der Siebtechnik).

Startq: Der initiale Wert für die Funktion `make_pol(N,q)`, d.h. bei `Startq` beginnt die Funktion in ihrem ersten Durchlauf die Suche nach einer geeigneten Primzahl zum Erzeugen der Siebpolynome.

(Die Funktion `isqrt(n)` liefert den ganzzahligen Anteil von $\sqrt{n}$.)

Fbas: Die Faktorbasis zur Zahl N.
Rvec: Die modularen Quadratwurzeln von N bzgl. der Faktorbasis `Fbas`.
Lvec: Der skalierte Logarithmusvektor bzgl. der Faktorbasis `Fbas`.
Mat: Die Matrix die verwendet wird, um die $\vec{\beta_i}$ aus Gleichung (1) aus Definition 7.5. auf lineare Unabhängigkeit zu testen. Die Logik dahinter ist wie folgt: Angenommen die Vektoren $a_1, \ldots, a_m \in \mathbf{F}_2^n$ sollen auf lineare Unabhängigkeit gestestet werden. Wir bilden zunächst die $m \times n$-Matrix A mit den Zeilen a_i und erweitern diese mit der m-reihigen Einheitsmatrix E_m zu der $m \times (n+m)$-Matrix B. Durch elementare Zeilenumformungen, die simultan auf A und E_m angewandt werden, bringen wir A auf Zeilenstufen-Form und erhalten somit A' aus A und C aus E_m. Ist nun die k-te Zeile von A' gleich null, so gilt für die k-te Zeile c_k der Matrix C außerdem $c_k A = 0$.
Daraus folgt: Die Summe der Zeilen j von A, für die gilt $c_{k_j} = 1$, ist 0.
(Die Funktion `ematrix(m,n,x,i,j)` erzeugt eine $m \times n$-Matrix gefüllt mit 0-en, wobei das Element in Zeile i, Spalte j, den Wert x bekommt.)
Pivot: Diese Liste enthält die Zahlen von 1 bis `Flen` und wird für das oben beschriebene Gauß'sche Eliminationsverfahren benötigt.
CurRow: Enthält den Index der Zeile von `Mat`, auf der die aktuellen Operationen stattfinden.
Count1: Zählt, wie viele verschiedene quadratische Polynome zum Erzeugen der Kongruenzen benötigt werden.
Count2: Zählt, wie viele der gefunden quadratischen Reste komplett in Primfaktoren aus der Faktorbasis `Fbas` zerfallen.
Stack: Speichert vorübergehend die Paare (u, v), die durch das Sieb-Verfahren entstehen. Hier wird der abstrakte Datentyp **Stack**[14] durch eine Liste implementiert.
Sieve: Eine Liste, die das Siebintervall repräsentiert.
Uvec, Vvec: Listen, die jeweils die Werte der Paare (u, v) speichern, welche der Kongruenz $u^2 \equiv v \,(mod\, N)$ genügen.
Damit sind alle nötigen globalen Variablen initialisiert. Nun können die einzelnen Funktion besprochen werden, die später in der Hauptfunktion zu ei-

[14] In der Programmierung stellt ein abstrakter Datentyp (ADT) eine Struktur zur Datenspeicherung mit Ein-/Ausgabefunktion dar. Dadurch dass ein ADT nur eine Beschreibung der Funktionalität liefert, muss er in den Programmiersprachen separat implementiert werden.

nem Verfahren zusammengefügt werden.

2. Das Sieb-Verfahren:

```
(%i125) get_qres(N) := block([Q,UV],
           while is(length(Stack) = 0) do (
              Q : make_pol(N,Startq), Count1 : Count1 + 1,
              disp(concat("Polynom: ", Count1)),
              do_sieve(Q),
              sieve_results(Q),
              Startq : Q[1] + 2
           ),
           UV : pop(Stack),
           return(UV)
        )$
```

Die Funktion `get_qres(N)` ermittelt zunächst, ob noch Paare (u, v) im Stack vorhanden sind. Wenn ja, wird durch die Methode `pop(Stack)` das erste Element aus `Stack` ausgelesen, gelöscht und zurückgegeben. Wenn nicht, wird zunächst ein neues Polynom zum Erzeugen der Paare gebildet. Die Funktion `disp(...)` gibt den Inhalt `...` auf dem Bildschirm wieder. Der Unterschied zur `print`-Methode besteht darin, dass `disp` den Inhalt während der Berechnung ausgibt und nicht erst, wenn die Berechnung durchgelaufen ist. Hier wird damit die aktuelle Anzahl an erzeugten Polynomen ausgegeben.
Danach werden die Funktionen `do_sieve(Q)` und `sieve_results(Q)` mit dem gerade erzeugten Polynom aufgerufen, welche das eigentliche Sieb-Verfahren nach der zu Beginn besprochenen Siebtechnik durchführen.

```
(%i126) do_sieve(Q) := block([a1,b1,i,i0,k,p,r,r1,s,z],
           Sieve : makelist(0,n,2·Srange),
           for k : 3 thru Flen do (
               [p,z,r] : [Fbas[k],Lvec[k],Rvec[k]],
               s : mod(-Srange,p),
               if mod(Q[2],p) # 0 then (
                   a1 : inv_mod(Q[2],p),
                   b1 : mod(Q[3],p),
                   r1 : mod((r-b1)·a1,p),
                   if is(r1 >= s) then
                       i0 : r1-s
                   else
                       i0 : p+r1-s,
                   for i : (i0+1) thru 2·Srange step p do
                       Sieve[i] : Sieve[i] + z,
                   r2 : mod((p-r-b1)·a1,p),
                   if is(r2 >= s) then
                       i0 : r2-s
                   else
                       i0 : p+r2-s,
                   for i : (i0+1) thru 2·Srange step p do
                       Sieve[i] : Sieve[i] + z
               )
           )
       )$
```

Die Funktion `do_sieve` initialisiert zunächst das Siebintervall mit 0-en. Danach übernimmt sie den Teil der Siebtechnik, in dem für jedes $p \in \mathcal{F}$ an den Stellen $x = \xi_\nu^{(p)} + kp$ mit $k \in \mathbb{Z}$ des Siebintervalls, der Wert des Logarithmusvektors der Primzahl aus der Faktorbasis addiert wird, wobei $\xi_\nu^{(p)}$ wie oben beschrieben durch $\xi_\nu^{(p)} \equiv a^{-1}(\pm r - b)\,(mod\, p)$ definiert ist.
Den zweiten Teil der Siebtechnik übernimmt dann die Funktion `sieve_results`:

```
(%i127) sieve_results(Q) := block([k,qinv,target,u,v,x]
          target : round(ev(log(Num/Q[2])·scale,numer)
                         -Lvec[Flen]),
          qinv : inv_mod(Q[1],Num),
          for k : 1 thru 2·Srange do (
              if is(Sieve[k] >= target) then (
                  x : k-1-Srange,
                  u : Q[2]·x + Q[3],
                  v : (u + Q[3])·x + Q[4],
                  u : mod(qinv·u,Num),
                  Stack : push([u,v],Stack)
              )
          ),
          return(length(Stack))
        )$
```

Die Funktion legt zunächst die Toleranzgrenze `target` fest und läuft dann durch das von `do_sieve` befüllte Siebintervall. Dabei werden nur diejenigen Stellen berücksichtigt, die in der Summe über ihre logarithmierten Werte an die Toleranzgrenze herangekommen sind. Aus diesen Werten werden dann die Paare (u, v) konstruiert und im `Stack` abgelegt. Die Funktion `push([u,v], Stack)` fügt dabei das Paar (u, v) als Liste an vorderster Stelle im `Stack` ein und gibt diesen zurück.

3. Probedivision potentieller Werte für v: Nachdem wir die Anzahl der potentiellen Paare (u, v) deutlich eingeschränkt haben, können wir nun für die verbleibenden Kandidaten eine Probedivision durchführen, um zu sehen, ob v wirklich vollständig in Elemente aus der Faktorbasis zerfällt:

```
(%i128) qr_trialdiv(v) := block([i,found : false,m,p],
          Mat[CurRow] : makelist(0,n,length(Mat[CurRow])),
          if is(v < 0) then (
              v : -v,
              Mat[CurRow][1] : 1
          ),
```

Wird auf der nächsten Seite fortgesetzt...

```
        for i : 2 thru Flen do (
            p : Fbas[i],
            m : 0,
            while is(v > 1) and is(mod(v,p) = 0) do
                [v,m] : [floor(v/p),m+1],
            if is(oddp(m)) then
                Mat[CurRow][i] : 1,
            if is(v <= 1) then (
                found : true,
                Mat[CurRow][Flen+CurRow] : 1,
                Count2 : Count2 + 1,
                return(true)
            )
        ),
        if found then
            return(true)
        else
            return(false)
    )$
```

Die Funktion `qr_trialdiv(v)` führt besagte Probedivision für v durch. Dabei wird zunächst die Zeile der Matrix `Mat`, auf der wir aktuell operieren, mit 0-en initialisiert. Danach agiert die Methode wie folgt:

1. Ist v negativ, so wird v durch $-v$ ersetzt. Da demnach -1 mit einem ungeraden Exponenten in der Primfaktorzerlegung von v auftaucht, setzte den ersten Eintrag in der aktuellen Zeile der Matrix `Mat` auf 1. (Errinerung: -1 ist **immer** der erste Eintrag in der Faktorbasis.)

2. Teile nun v sukzessive durch die Elemente der Faktorbasis. Dabei wird so oft durch ein Element geteilt, bis das Ergebnis nicht mehr ganzzahlig ist. Kann durch ein Element eine ungerade Anzahl oft geteilt werden, setzte den zugehörigen Eintrag in der aktuellen Zeile der Matrix `Mat` auf 1.

3. Erreicht v während dieses Verfahrens die Bedingung $v \leq 1$, so lässt sich v komplett in Elemente der Faktorbasis zerlegen. In diesem Fall wird zusätzlich in der aktuellen Zeile der Matrix `Mat` das zugehörige Diagonalelement der Einheitsmatrix auf der rechten Seite auf 1 gesetzt. Außerdem wird das Verfahren abgebrochen und `true` zurückgegeben.

4. Erreicht v diese Bedingung nicht, so lässt sich v nicht komplett in Elemente aus der Faktorbasis zerlegen und es wird `false` zurückgegeben.

4. Test auf lineare Abhängigkeit: Hat ein v die Probedivision „bestanden“, so wird überprüft, ob der neu gewonnene Exponentenvektor der Zerlegung von v den Rang der Matrix `Mat` um 1 erweitert, oder durch eine Linearkombination der vorherigen Vektoren dargestellt werden kann.

```
(%i129) gauss_elim(k) := block([bool : true,i,j,j0],
           for i : 1 thru k-1 do (
               if is(Mat[k][Pivot[i]] = 1) then
                   Mat[k] : mod(Mat[k] + Mat[i],2)
           ),
           for i : k thru length(Pivot) do (
               j : Pivot[i],
               if is(Mat[k][j] = 1) then (
                   j0 : Pivot[k], Pivot[k] : j, Pivot[i] : j0,
                   bool : false,
                   return(0)
               )
           ),
           if bool then
               return(true)
           else
               return(false)
       )$
```

Die Funktion `gauss_elim(k)` geht davon aus, dass die Matrix `Mat` bereits bis zur $(k-1)$-ten Zeile auf Zeilenstufen-Form gebracht wurde und setzt diese Transformation bis zur k-ten Zeile fort. Hier wird zum ersten Mal die Variable `Pivot` benutzt. Die Interaktion zwischen `Mat` und `Pivot` ist dabei wie

folgt:
Der `Pivot`-Vektor ist eine Permutation der Zahlen von 1 bis `Flen`. Dabei gilt für alle $i < k$, dass die Komponente der i-ten Zeile der Matrix `Mat` mit dem Index `Pivot[i]` gleich 1 ist. Die Komponenten der i-ten Zeile der Matrix `Mat` mit Index `Pivot[j]` sind dabei für alle $j < i$ gleich 0.
Für die Transformation werden zunächst alle Spalten der k-ten Zeile der Matrix `Mat` mit einem Index `Pivot[i]` für $i < k$, die nicht 0 sind, durch Addition vorheriger Zeilen zu 0 gemacht (Erste `for`-Schleife). Danach wird ein Index $j :=$ `Pivot[k]` bestimmt, sodass die k-te Zeile in der j-ten Spalte den Eintrag 1 hat. Wird ein solcher Index gefunden, wird der `Pivot`-Vektor so angepasst, dass die Funktion im nächsten Durchlauf die Transformation fortsetzen kann. Kann dieser Index allerdings nicht gefunden werden, so wurde die k-te Zeile durch die vorangegangen Zeilenumformungen zu 0 gemacht, es liegt also eine lineare Abhängigkeit vor. In diesem Fall wird `true` zurückgegeben, andernfalls `false`.

5. Finden der Abhängigkeiten: Liefert die Funktion `gauss_elim` ein positives Ergebnis, so muss aus der Matrix `Mat` ausgelesen werden, welche Zeilen addiert wurden, um die Nullzeile zu erzeugen. Per Konstruktion und Verarbeitung der Matrix `Mat` sind das nun trivialerweise die Indizes `Flen`$+i$, $0 \leq i \leq k$, der k-ten Zeile von `Mat`, welche den Wert 1 haben.
Dieses Verfahren übernimmt die Funktion `get_rel(row)`:

```
(%i130) get_rel(row) := block([i, rel : []],
           for i : 1 thru row do (
               if is(Mat[row][Flen+i] = 1) then
                   rel : append(rel, [i])),
           return(rel)
        )$
```

6. Finden eines nicht-trivialen Teilers: Der aus Punkt 5. gefundene Relationsvektor kann nun verwendet werden, um den Versuch zu starten, einen nicht-trivialen Faktor von N zu finden. Die Funktion `find_factor` arbeitet dabei auf den Vektoren `Uvec` und `Vvec` nach der, am Ende des Fermat'schen Faktorisierungsverfahrens, besprochenen Methode:
Seien $k_1, \ldots, k_r$ die Einträge des Relationsvektors, sowie $u_i :=$ `Uvec`$[k_i]$ und $v_i :=$ `Vvec`$[k_i]$. Dann gilt $u_i^2 \equiv v_i \, (mod\, N)$ für alle $0 \leq i \leq r$. Weiterhin ist

$z := \prod_{i=1}^{r} v_i =: y^2$ eine Quadratzahl. Setzten wir nun $x := \prod_{i=1}^{r} u_i$, so gilt: $x^2 \equiv y^2 \,(mod\, N)$, d.h. $(x+y)(x-y) \equiv 0 \,(mod\, N)$. Ein möglicher Teiler von N kann also gesucht werden, in dem $gcd(x+y, N)$ berechnet wird.

```
(%i131) find_factor(N,relvec) := block([d,i,k,u,v,v1,x,y],
           k : relvec[1],
           [x,v,y] : [Uvec[k],Vvec[k],1],
           for i : 2 thru length(relvec) do (
              k : relvec[i],
              [u,v1] : [Uvec[k],Vvec[k]],
              x : mod(x·u,N),
              d : gcd(v,v1),
              v : floor(v/d)·floor(v1/d),
              y : mod(y·d,N)
           ),
           y : mod(y·isqrt(v),N),
           d : gcd(x+y,N),
           if is(d <= 1) or is(d = N) then
              d : 0,
           return(d)
        )$
```

Die Zwischenschritte der Funktion `find_factor(N,relvec)` resultieren dabei aus folgender Überlegung:
Wir können zwar x modulo N berechnen, jedoch nicht y^2 modulo N, da aus $y^2 \equiv \hat{y}^2 \,(mod\, N)$ nicht $y \equiv \hat{y} \,(mod\, N)$ folgt. Außerdem wird das Produkt über die v_i sehr schnell sehr groß. Um diesem Problem zu entgehen, agiert die Funktion nach folgender Logik:
Das Gesamtprodukt ist zwar ein Quadrat, die einzelnen Faktoren jedoch nicht. Man kann also bei der sukzessive Produktbildung erwarten, dass nichttriviale gemeinsame Teiler der Faktoren auftreten. Betrachten wir beispielsweise das Produkt zweier Zahlen A und B mit $d = gcd(A, B)$, also $A = ad$ und $B = bd$, so gilt $AB = ab \cdot d^2$. Man dividiert also die gemeinsamen Teiler jeweils heraus und baut somit bereits ein Teil von y auf.

7. Zusammenführen der Funktionen: Zum Schluss führen wir alle besprochenen Funktionen in einer Hauptfunktion zusammen, welche uns im Idealfall

einen nicht-trivialen Teiler der Zahl N liefert:

```
(%i132) QS_factorize(N) := block([d,relvec,result : 0,u,v],
          QS_init(N),
          disp(concat("Quadratisches Sieb der Länge: ",
                      2·Srange)),
          disp(concat("Faktorbasis 2 ...", Fbas[Flen],
                      " der Länge ", Flen-1)),
          nbfail : 0,
          while is(nbfail < 32) do (
              [u,v] : get_qres(N),
              if qr_trialdiv(v) then (
                  Uvec[CurRow] : u,
                  Vvec[CurRow] : v,
                  if not gauss_elim(CurRow) then (
                      CurRow : CurRow + 1
                  )
                  else (
                      relvec : get_rel(CurRow),
                      disp("Relation gefunden: Suche Faktor"),
                      d : find_factor(N,relvec),
                      if is(d > 0) then (
                          result : d,
                          return(0)
                      )
                      else
                          nbfail : nbfail + 1
                  )
              )
          ),
          printf(true,
                 concat("Resultat: ~d, ~%",
                        "Polynome: ~d, ~%",
                        "Komplett faktorisierte Reste: ~d"),
                 result, Count1, Count2),
```

Wird auf der nächsten Seite fortgesetzt...

```
            if is(result > 0) then
               return("Faktor gefunden")
            else
               return("Kein Faktor gefunden")
        )$
```

Die Funktion `QS_factorize(N)` arbeitet nun alle besprochen Schritte der Reihe nach ab und verarbeitet in einer Schleife dabei jeweils die positiven/ negativen (`true/false`) Rückgabewerte der einzelnen Methoden. Desweiteren wird eine Schranke `nbfail` angeführt. Wird mehr als 32 Mal ein Relationsvektor gefunden, aus dem kein nicht-trivialer Faktor von N resultiert, so bricht die Funktion ab. Außerdem findet sich als Neuerung in der `printf`-Methode die Anweisung `~%`, welche einen Zeilenumbruch darstellt.
Ein einzelner Aufruf der Funktion setzt nun den Algorithmus in Gang:

```
(%i133) QS_factorize(2^50+1);
        Quadratisches Sieb der Länge: 26112
        Faktorbasis 2 ... 1069 der Länge 80
        Polynom: 1
        Polynom: 2
        Relation gefunden: Suche Faktor
        Relation gefunden: Suche Faktor
        Resultat: 27118601,
        Polynome: 2,
        Komplett faktorisierte Reste: 71
(%o133) Faktor gefunden
```

Zum Schluss bleibt noch anzumerken, dass unser Algorithmus deutlich weniger optimiert ist als die Vorlage von *Otto Forster* aus seinem Buch *Algorithmische Zahlentheorie*, auf dem diese Implementierung basiert. Überprüft man die Laufzeiten der einzelnen Funktionen, so fallen besonders die längeren Laufzeiten der Funktionen `do_sieve`, `sieve_results` und `qr_trialdiv` ins Gewicht. Dies liegt vermutlich einerseits daran, dass *Forster* in seiner Implementierung die sehr effizienten Bit-Operationen verwendet, andererseits eventuell aber auch daran, dass die von *Forster* entwickelte Programmiersprache *Aribas* in *C* programmiert ist und *Maxima* in *Common Lisp*.

Eine weitere Möglichkeit hierfür liegt in der Diskrepanz zwischen den Stack-Implementierungen.

7.3 Kryptographie auf elliptischen Kurven

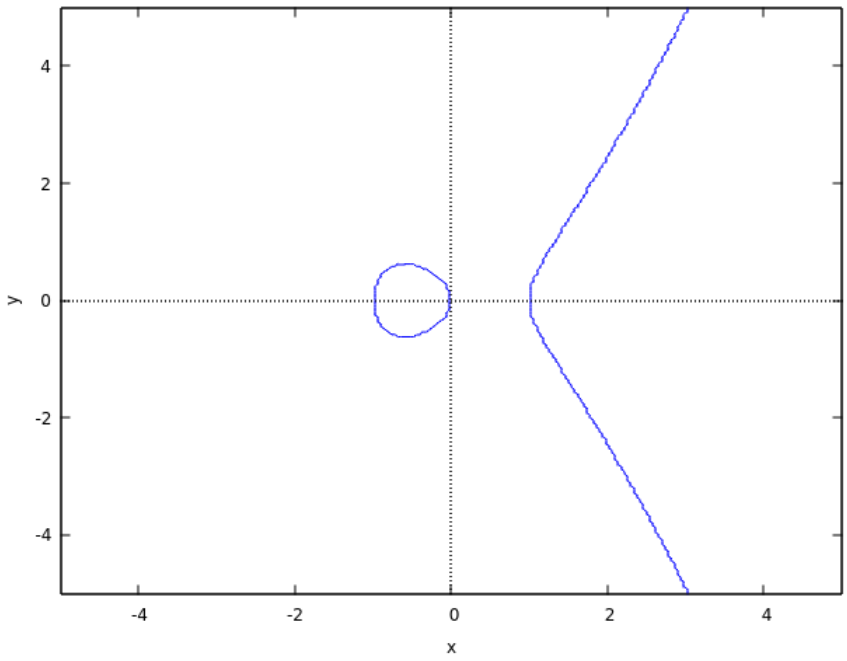
In diesem Abschnitt wollen wir eine kurze Einführung in die Kryptographie auf elliptischen Kurven geben, um anschließend nochmals einen Blick auf das diskrete Logarithmus-Problem zu werfen. Da das Gebiet um elliptische Kurven ein sehr umfangreiches Teilgebiet der Algebra darstellt, sollen hier nur die relevantesten Informationen geliefert werden.

Definition 7.9. Sei $y^2 = x^3 + Ax + b$ eine Gleichung über einer Menge K, nennen wir den Graph $E : y^2 = x^3 + Ax + b$ eine **elliptische Kurve**. Ist K ein Körper, so nennen wir $E(\mathbf{K})$ eine **elliptische Kurve über K**.

Wir werden elliptische Kurven in diesem Abschnitt immer über dem endlichen Körper $\mathbf{F}_p = \mathbb{Z}_p^\times$ betrachten, den wir in der modularen Arithmetik eingeführt haben. Betrachten wir als Beispiel den Graphen der Gleichung $y^2 = x^3 - x$:

```
(%i134) load(implicit_plot)$
(%i135) implicit_plot(y^2 = x^3 - x, [x,-5,5], [y,-5,5])$
```

Die Funktion `load(implicit_plot)` lädt das Modul *implicit_plot* in den aktuellen *Maxima*-Kernel, welches benötigt wird, um nicht-lineare Gleichungen zu zeichnen. Der Aufruf liefert folgende Grafik:



Die Definition einer elliptischen Kurve liefert uns folgende Konstruktion für Addition und Subtraktion:

Definition u. Satz 7.10. Sei $E : y^2 + Ax + B$ eine elliptische Kurve. Ferner seien $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$ zwei Punkte auf E mit $P_1, P_2 \neq \infty$. Definiere $P_1 + P_2 = P_3 = (x_3, y_3)$ wie folgt:

- Falls $x_1 \neq x_2$, dann ist:

$$x_3 = m^2 - x_1 - x_2 \text{ und } y_3 = m(x_1 - x_3) - y_1 \text{ mit } m = \frac{y_2 - y_1}{x_2 - x_1}.$$

- Falls $x_1 = x_2$ aber $y_1 \neq y_2$, dann ist $P_1 + P_2 = \infty$.

- Falls $P_1 = P_2$ und $y_1 \neq 0$, dann ist:

$$x_3 = m^2 - 2x_1 \text{ und } y_3 = m(x_1 - x_3) - y_1 \text{ mit } m = \frac{3x_1^2 + A}{2y_1}$$

- Falls $P_1 = P_2$ und $y_1 = 0$, dann ist $P_1 + P_2 = \infty$

Ferner sei $P + \infty = P$ für alle P auf E.
$(E, +)$ *ist eine abelsche Gruppe mit* ∞ *als neutralem Element.*

Die obige Konstruktion resultiert dabei aus dem Schnittpunkt zwischen der Gerade durch P_1 und P_2 und der elliptischen Kurve. Die skalare Multiplikation kP ist dann definiert durch die k-fache Addition des Punktes mit sich selber. Weiterhin gilt für die Subtraktion $Q - P = Q + (-P)$ mit $-P = (x_P, -y_P)$.
Die Definition einer elliptischen Kurve zzgl. einer Addition auf einer solchen liefert uns folgende kryptographische Anwendung:

Definition 7.11. (Diffie-Hellman-Schlüsselaustausch auf elliptischen Kurven) Angenommen zwei Personen (A,B) benutzen ein Private-Key-Verfahren, um Nachrichten zu verschlüsseln. Im folgenden Protokoll soll ein Algorithmus definiert werden, um den Schlüssel des Verfahrens in einem öffentlichen Kommunikationsraum sicher austauschen zu können:

- A und B einigen sich auf eine elliptische Kurve E über einem endlichen Körper $\mathbf{F}_p$, so dass das diskrete Logarithmus-Problem schwer über $E(\mathbf{F}_p)$ zu lösen ist (Diese Eigenschaft wird später noch genauer betrachtet).
- A wählt ein geheimes $a \in \mathbb{Z}$, berechnet $P_a = aP$ und sendet P_a an B.
- B wählt ein geheimes $b \in \mathbb{Z}$, berechnet $P_b = bP$ und sendet P_b an A.
- A berechnet $aP_b = abP$. B berechnet $bP_a = baP$.
- A und B einigen sich öffentlich auf eine Methode, aus abP einen Schlüssel zu generieren, beispielsweise die letzten k-Bits der x-Koordinate von abP.

Ein potentieller Angreifer (Mithörer) hat die Informationen über die Kurve E, den endlichen Körper F_p, die Punkte P, aP, bP und die Methode zur Schlüsselbestimmung. Um den Punkt abP berechnen zu können, muss aber zunächst a oder b berechnet werden. Sei beispielsweise $aP = (x_k, y_k)$. Dann nennen wir das Problem:

$$aP \equiv (x_k, x_y) \, (mod \, p)$$

das **diskrete Logarithmus-Problem** (DL-Problem) über der additiven Gruppe $E(\mathbf{F}_p)$. Da $\mathbf{F}_p$ extra so gewählt ist, dass dieses Problem „schwer“

zu lösen ist, hat der Angreifer nach heutigem Wissensstand keine effiziente Möglichkeit, dieses a zu berechnen.
Zur Entscheidung, ob ein DL-Problem über $\mathbf{F}_p$ schwer zu lösen ist, wollen wir im folgenden Abschnitt einen Algorithmus betrachten und implementieren, der sich dieses Problems annimmt:

7.3.1 Ein Algorithmus für das diskrete Logarithmus-Problem über additiven Gruppen

Um mit elliptische Kurven rechnen zu können, müssen wir die Addition (Multiplikation) zunächst in *Maxima* implementieren.
Das Einzige was hierbei zusätzlich zur oben angegebene Konstruktionsvorschrift für die Addition (Multiplikation) beachtet werden muss, ist, dass das Ergebnis noch in ein Element aus $\mathbf{F}_p$ überführt werden muss.

Der Algorithmus folgt auf der nächsten Seite.

```
(%i136) add_ell(E,L,p) := block([m,x1,x2,x3,y1,y2,y3],
           if is(L[1] = inf) or is(L[1] = 0) then
              return(L[2]),
           if is(L[2] = inf) or is(L[2] = 0) then
              return(L[1]),
           [x1,y1] : mod([L[1][1],L[1][2]],p),
           [x2,y2] : mod([L[2][1],L[2][2]],p),
           if x1 # x2 then (
              m : (y2-y1)/(x2-x1),
              x3 : m^2-x1-x2,
              y3 : m·(x1-x3)-y1
           )
           else if is([x1,y1] = [x2,y2]) and is(y1 # 0) then (
              m : (3·x1^2+coeff(rhs(E),x))/2·y1,
              x3 : m^2-2·x1,
              y3 : m·(x1-x3)-x1
           ) else return(inf),
           x3 : mod(num(x3)·inv_mod(denom(x3),p),p),
           y3 : mod(num(y3)·inv_mod(denom(y3),p),p),
           return([x3,y3])
        )$
(%i137) mult_ell(E,P,k,p) := block([i,Q : P],
           for i : 1 thru (k-1) do
              Q : add_ell(E,[Q,P],p),
           return(Q)
        )$
```

Die Funktion `rhs(E)` liefert die rechte Seite der Gleichung der elliptischen Kurve.
Die Funktionen `num` und `denom` liefern den Zähler resp. den Nenner einer Zahl.

Folgendes Beispiel demonstriert die Arbeitsweise der Algorithmen:

```
(%i138) E1 : y^2 = x^3 + 2·x + 1$
(%i139) P : [0,1]$
(%i140) Q : [30,40]$
(%i141) add_ell(E1,[P,P],41);
(%o141) [1,39]
(%i142) add_ell(E1,[P,Q],41);
(%o142) [9,16]
(%i143) mult_ell(E1,P,4,41);
(%o143) [38,38]
(%i144) mult_ell(E1,Q,8,41);
(%o144) [19,16]
```

Durch die implementierte Addition & Multiplikation können wir uns nun dem Algorithmus nähern, vorher aber noch ein wichtiger Satz aus der Algebra:

Satz 7.12. ***(Satz von Hasse)*** *Die Gruppenordnung (Menge der Punkte) einer elliptischen Kurve $E(\mathbf{F}_p)$ über dem endlichen Körper $\mathbf{F}_p$ kann abgeschätzt werden durch:*

$$|N - (p+1)| \leq 2\sqrt{p}$$

wobei $N := ord(E(\mathbf{F}_p)) =$ „Anzahl der Punkte“.

Dieser Satz ist essentiell für die Implementation des folgenden Algorithmus:

Definition 7.13. (Baby Step, Giant Step) Sei $E(\mathbf{F}_p)$ eine elliptische Kurve über dem endlichen Körper $\mathbf{F}_p$. Ferner seien $P, Q \in E(\mathbf{F}_p)$ und $N = ord(E(\mathbf{F}_p))$ die Gruppenordnung von $E(\mathbf{F}_p)$.
Wir wollen das Problem $Q \equiv kP \,(mod\, p)$ lösen und definieren dazu folgenden Algorithmus:

1. Wähle ein $m \geq \sqrt{N}$ und berechne mP.
2. Erstelle eine Liste der Werte iP für $0 \leq i < m$.
3. Berechne die Punkte $Q - jmP$ für $0 \leq j < m$, bis eines der Resultate in der Liste aus **2.** auftaucht.
4. Falls $iP = Q - jmP$ ist, dann ist $Q = kP$ mit $k \equiv i + jm \,(mod\, N)$.

```
(%i145) B_G_Step(E,P,Q,p) := block([i,j : 1,L : [],m,
                                    match : false,N,tmpP,tmpQ],
          N : ceiling(2·sqrt(p)+(p+1)),
          m : ceiling(sqrt(N)),
          for i : 1 thru (m-1) do
              L : append(L,[mult_ell(E,P,i,p)]),
          while not match do (
              tmpP : mult_ell(E,P,j·m,p),
              tmpP : [first(tmpP),-second(tmpP)],
              tmpQ : add_ell(E,[Q,tmpP],p),
              if member(tmpQ,L) then
                  match : true,
              j : j+1
          ),
          i : sublist_indices(L, lambda([x], x = tmpQ)),
          return(mod(i[1] + (j-1)·m,N))
       )$
```

Die Funktionen `first` und `second` liefern jeweils das Erste resp. das Zweite Element einer Liste zurück. Die Funktion `member(ELEMENT, LIST)` überprüft, ob `ELEMENT` in `LISTE` vorhanden ist.
Nun wollen wir das Problem $Q \equiv kP\,(mod\,41)$ für die Punkte P, Q und die elliptische Kurve E aus unserem Beispiel lösen:

```
(%i146) B_G_Step(E1,P,Q,41)
(%o146) 23
(%i147) mult_ell(E1,P,23,41);
(%o147) [30,40]
```

Der Algorithmus hat also richtig gearbeitet und als Lösung $k = 23$ ermittelt. Was heißt das für unsere kryptographische Anwendung? Viele der Algorithmen zur Lösung des direkten Logarithmus-Problems basieren auf der Gruppenordnung $E(\mathbf{F}_p)$. Für den Anfang ist es auf jeden schonmal wichtig, $\mathbf{F}_p$ so zu wählen, dass $E(\mathbf{F}_p)$ möglichst viele Punkte enthält. Natürlich existieren noch eine Reihe an weiteren Algorithmen, die nicht zwingend von einer großen Gruppenordnung behindert werden., deswegen müssen in der Praxis definitiv noch zusätzliche Sicherheitsmaßnahmen getroffen werden.

8 Abschlusswort

Auch wenn in dieser Facharbeit einige Konzepte von *Maxima* vorgestellt werden konnten, so konnte im Gesamtkontext CAS *Maxima* doch nur an der Oberfläche gekratzt werden. Die zahlreichen Funktionen, Zusatzpakete und Möglichkeiten der direkten Common-Lisp-Programmierung würden aber eine Facharbeit in diesem Ausmaß bei Weitem übersteigen. Abschließend bleibt dem Leser nur mitzugeben, wenn durch diese Facharbeit Interesse in der Programmierung mit *Maxima* geweckt wurde, sich selbst durch die offizielle *Maxima*-Dokumentation tiefere Einblicke zu verschaffen und bei Bedarf auch aktiv in der Entwicklergemeinschaft *Maxima* zu partizipieren.

Vielen Dank für das Lesen meiner Bachelorarbeit!

9 Literaturverzeichnis

- Forster, Otto: *Algorithmische Zahlentheorie*, 2. Auflage. Springer Spektrum, 2015. S. 127-130, 163-179.
- Koepf, Wolfram: *Computeralgebra*: Eine algorithmitisch orientierte Einführung. Springer-Verlag Berlin Heidelberg, 2006. S. 51-178.
- Washingto, Lawrence C.: *Elliptic Curves*: Number Theory and Cryptography, Second Edition. Taylor & Francis Ltd, 2008. S. 9-15, 143-147, 170-171.
- J. Buchmann: *Einführung in die Kryptographie*. Springer, 1999.
- Offizielle Maxima Dokumentation: http://maxima.sourceforge.net/docs/manual/maxima.html
- Maxima-Einführung: http://maxima.sourceforge.net/docs/tutorial/de/maxima-einfuehrung.pdf
- Modulare Quadratwurzeln und Algorithmen: https://www.rieselprime.de/ziki/Modular_square_root
- Erweiteter Euklidischer Algorithmus (Konstruktion): https://de.wikipedia.org/wiki/Erweiterter_euklidischer_Algorithmus
- Maxima-Logo: https://en.wikipedia.org/wiki/Maxima_(software)#/media/File:Maxima-new.svg