Kapitel 4
Kürzen und Kryptographie

Mathematik schafft Klarheit durch Aufdecken von Strukturen. Nutzt man die strukturellen Eigenschaften eines Problems, so erhält man effiziente Algorithmen. Das wollen wir anhand der Berechnung des größten gemeinsamen Teilers illustrieren. Dafür machen wir einen kurzen Ausflug in die Zahlentheorie, der uns unvermutet in die Kryptographie führt. Tatsächlich sind bestimmte, weltweit verbreitete Verschlüsselungsverfahren nur so lange sicher, wie niemand einen effizienten Algorithmus zur Primfaktorzerlegung gefunden hat. Vielleicht gibt es den ja auch gar nicht...

4.1 Kürzen von Brüchen

Die effiziente Berechnung des größten gemeinsamen Teilers $\mathrm{ggT}(a,b)$ zweier gegebener ganzer Zahlen a und b ist alles andere als ein Spielproblem. Anwendungen gibt es in der Computeralgebra (Addition und Kürzen von Brüchen, ...) aber auch in völlig anderen Zusammenhängen wie beispielsweise der Kryptographie. Bevor wir zur Berechnung von $\mathrm{ggT}(a,b)$ kommen, wollen wir ein paar Begriffe klären.

Definition 4.1 (Teiler, Primzahlen und größter gemeinsamer Teiler ggT). Eine ganze Zahl $i \neq 0$ *teilt* eine ganze Zahl a oder, gleichbedeutend, heisst *Teiler* von a, falls der Quotient a/i wieder eine ganze Zahl ist. In diesem Fall schreiben wir $i|a$.

Eine Zahl p heisst *Primzahl*, wenn $i = 1,\ p$ die einzigen natürliche Zahlen mit der Eigenschaft $i|p$ sind.

Eine Zahl $i \neq 0$ mit der Eigenschaft $i|a$ und $i|b$ heißt *gemeinsamer Teiler* von a und b. Den *größten gemeinsamen Teiler* von a und b bezeichnen wir mit $\mathrm{ggT}(a,b)$. Zwei ganze Zahlen a, b heissen *teilerfremd*, wenn $\mathrm{ggT}(a,b) = 1$ ist.

Beispielsweise gilt $2|4$, $3|45651$ und $i|0$ für alle $i \in \mathbb{Z} \setminus \{0\}$ sowie $\mathrm{ggT}(21,15) = 3$, $\mathrm{ggT}(29,59) = 1$ und $\mathrm{ggT}(a,0) = a$ für alle $a \in \mathbb{Z} \setminus \{0\}$. Der Ausdruck $\mathrm{ggT}(0,0)$ ist sinnlos.

Ob $i|a$ vorliegt kann man mit der Modulo-Funktion feststellen.

Definition 4.2 (Gauß–Klammer und Modulo–Funktion). Es seien a und $i \neq 0$ zwei ganze Zahlen. Dann bezeichnet die *Gauß-Klammer* $\lfloor a/i \rfloor$ die größte ganze Zahl z mit der Eigenschaft $z \leq a/i$.

Die *Modulo-Funktion* $\mathrm{mod} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ ist dann definiert durch

$$\mathrm{mod}\,(a,i) = a - \lfloor a/i \rfloor i\,. \tag{4.1}$$

Die Auswertung von $\mathrm{mod}\,(a,i)$ liefert gerade den nichtnegativen Rest bei Division von a durch $i \neq 0$. Beispielsweise ist $\mathrm{mod}\,(17,3) = 2$ und aus $-2/5 = -1 + 3/5$ folgt $\mathrm{mod}\,(-2,5) = 3$. Offenbar gilt

$$i|a \quad \Longleftrightarrow \quad \mathrm{mod}\,(a,i) = 0\,.$$

Nun wollen wir uns mit der Berechnung von $\mathrm{ggT}(a,b)$ beschäftigen. Da das Vorzeichen für die Teilbarkeit keine Rolle spielt und wir den sinnlosen Fall $a = b = 0$ ausschließen müssen, nehmen wir im Folgenden an, daß

$$0 \leq b \leq a \quad \text{und} \quad 0 < ab$$

vorliegt. Im Falle $b = 0$ ist bekanntlich $\mathrm{ggT}(a,b) = a$. Sonst gilt

$$1 \leq \mathrm{ggT}(a,b) \leq b\ .$$

Wir können daher $\mathrm{ggT}(a,b)$ durch einfaches Ausprobieren berechnen: Im Falle $b > 0$ werden alle Zahlen $i = 1,\ldots,b$ daraufhin untersucht, ob sie a und b teilen. Die größte dieser Zahlen ist dann unser gesuchtes Ergebnis.

Algorithmus 4.3 (TumbGGT).

```
function ggT=tumbggt(a,b)

if b==0 ggT=a;

else

     for i = 1:b

          if mod(a,i)==0 & mod(b,i)==0
             ggT = i;
          end
     end
end
```

Wir wollen den Aufwand dieses Verfahrens untersuchen. Im Hinblick auf Definition 2.4 müssen wir dazu als erstes eine Referenzoperation wählen. Jede Auswertung der Modolo-Funktion mod erfordert eine Division mit Rest. Gegenüber den anderen erforderlichen Operationen dominiert dies den Rechenaufwand. Daher ist unsere

$$\text{Referenzoperation}: \quad \text{Auswertung von mod}\ .$$

In der Zahlentheorie ist es üblich, den Aufwand eines Algorithmus' auf die Anzahl der *Stellen* des Arguments zu beziehen. Also wählen wir die

$$\text{Eingabelänge } n\text{:} \quad \text{Anzahl der Stellen von } b\ .$$

Die Anzahl der Dezimalstellen von $b \geq 1$ ist gerade $n = 1 + \lfloor \log(b) \rfloor$ wobei $\log(b)$ den Logarithmus zur Basis 10 bezeichnet. Im Falle $b = 0$ setzen wir $n = 0$. Als Spezialfall der allgemeinen Definition 2.4 vereinbaren wir nun den folgenden Aufwandsbegriff.

Definition 4.4 (Aufwand). Der Aufwand $T_A(n)$ eines Algorithmus' A zur Berechnung von $\mathrm{ggT}(a,b)$ ist das Maximum der benötigten mod–Auswertungen über alle $b \geq 0$ mit n Dezimalstellen und beliebige $a \geq b$ mit $ab > 0$.

Bei Algorithmus 4.3 werden in jedem Schleifendurchlauf zwei Auswertungen von mod durchgeführt. Die Schleife wird genau b mal durchlaufen. Insgesamt sind also $2b$ Auswertungen von mod erforderlich. Aus den Potenzrechenregeln erhält man

$$\tfrac{1}{5} \cdot 10^n = 2 \cdot 10^{1+\lfloor \log(b) \rfloor - 1} \leq 2b \leq 2 \cdot 10^{1+\lfloor \log(b) \rfloor} = 2 \cdot 10^n$$

und damit den folgenden Satz.

Satz 4.5. *Für den Aufwand von TumbGGT gilt* $T_{\mathrm{TumbGGT}}(0) = 0$ *und*

$$\tfrac{1}{5} \cdot 10^n \leq T_{\mathrm{TumbGGT}}(n) \leq 2 \cdot 10^n\ , \qquad n = 1,2\ldots\ .$$

Der Aufwand von TumbGGT wächst also exponentiell.

Algorithmus 4.3 findet übrigens alle Teiler von a und b, obwohl doch nur der größte Teiler $\mathrm{ggT}(a,b)$ berechnet werden soll. Es ist daher klüger, die Schleife mit $i = b$ beginnend rückwärts zu durchlaufen und gleich beim ersten (also dem größten) gefundenen gemeinsamen Teiler abzubrechen. Das verbessert den Aufwand (im Sinne unseres Aufwandsbegriffs) allerdings nicht (siehe Aufgabe 4.5).

4.2 Kongruenzen und Restklassen

Die geraden Zahlen sind dadurch charakterisiert, daß sie sich ohne Rest durch 2 teilen lassen, während bei ungeraden Zahlen der Rest 1 übrig bleibt. Die bei Division durch 2 auftretenden Reste 0 und 1 kann man

also zu einer Einteilung der ganzen Zahlen in die zwei Pakete mit der Aufschrift *gerade* und *ungerade* nutzen. Wieviele solcher Pakete erhält man aus den Resten bei Division durch 3 oder gar durch M und was kann man damit anfangen? Das wollen wir genauer untersuchen, zunächst einmal aus lauter Freude an der Abstraktion. Wie so oft beginnen wir mit einer grundlegenden Definition.

Definition 4.6 (Kongruenzen[1]). Sei M eine positive ganze Zahl, die wir *Modul* nennen. Zwei ganze Zahlen a, b heissen dann *kongruent modulo* M, falls $M|(a-b)$ vorliegt. Gleichbedeutend schreiben wir die *Kongruenz*

$$a \equiv b \bmod M\,.$$

Anschaulich bedeutet $a \equiv b \bmod M$, daß a und b bei Division durch M denselben Rest ergeben. Ist beispielsweise $M = 2$, so gilt

$$a \text{ gerade}: \quad a \equiv 0 \bmod 2, \qquad a \text{ ungerade}: \quad a \equiv 1 \bmod 2\,.$$

Kongruenzen spielen eine ähnliche Rolle wie Gleichungen. Durch

$$x \cong a \quad \Longleftrightarrow \quad x \equiv a \bmod M\,.$$

wird nämlich eine Äquivalenzrelation definiert. Die entsprechenden Äquivalenzklassen verdienen einen eigenen Namen.

Definition 4.7 (Restklassen). Für jedes $a \in \mathbb{Z}$ heisst die Menge

$$[a]_M = \{x \in \mathbb{Z} \mid x \equiv a \bmod M\}$$

die von a erzeugte *Restklasse modulo* M. Jedes Element $a' \in [a]_M$ heisst Vertreter von $[a]_M$.

Die Restklasse $[a]_M$ enthält also alle ganzen Zahlen, die bei Division durch M denselben Rest ergeben. Im Falle $M = 2$ gibt es an nichtnegativen Resten nur 0 oder 1. Also gibt es nur zwei verschiedene Restklassen modulo 2, nämlich

$$\text{gerade Zahlen:} \quad [0]_2, \qquad \text{ungerade Zahlen:} \quad [1]_2\,.$$

Im allgemeinen sind bei der Division durch M die nichtnegativen Reste $0, 1, 2, \ldots, M-1$ möglich. Man nennt sie die *kanonischen Vertreter* der M Restklassen modulo M

$$[0]_M, [1]_M, \ldots, [M-1]_M\,.$$

Nach Definition 4.7 gilt

$$a' \in [a]_M \quad \Longleftrightarrow \quad a' \equiv a \bmod M\,.$$

Man kann sich also aussuchen, welche der beiden Schreibweisen man bevorzugt, Restklassen oder Kongruenzen. In der Zahlentheorie haben sich die Kongruenzen durchgesetzt. Wir werden daher auch von nun an nur noch diese Schreibweise verwenden.

Für die Addition und Multiplikation gelten wegen Definition 4.6 die folgenden Rechenregeln: Aus $a' \equiv a \bmod M$ und $b' \equiv b \bmod M$ folgt

$$a' + b' \equiv a + b \bmod M, \qquad a'b' \equiv ab \bmod M\,. \tag{4.2}$$

Im Falle $M = 2$ bedeutet das beispielsweise

$$1 + 1 \equiv 2 \equiv 0 \bmod M\,.$$

Für das Modul $M = 12$ können wir die Addition von Kongruenzen anhand des Ziffernblatts einer Uhr veranschaulichen: Wandert der kleine Zeiger von 11 Uhr zwei Stunden weiter, so ist es 1 Uhr. Dementsprechend ist $11 + 2 \equiv 1 \bmod 12$. Wegen (4.2) reicht es auch, nur die Addition und Multiplikation der kanonischen Vertreter $0, 1, \ldots, M-1$ kongruenter Zahlen (Restklassen) zu betrachten. Für den Fall $M = 4$ sind die zugehörigen Additions- und Multiplikationstabellen in Tabelle 4.1 aufgeführt.

[1] Unabhängig voneinander kamen der Chinese Ch'in Chiu Shao (1247) und Goldbach (1730) auf diese Idee. Erst Gauß (1801) erkannte aber die fundamentale Bedeutung von Kongruenzen für die Algebra und Zahlentheorie [13].

+	0	1	2	3
0	0	1	2	3
1	0	2	3	0
2	2	3	0	1
3	0	0	1	2

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Tabelle 4.1 Additions- und Multiplikationstabelle für Restklassen modulo 4

Über die Beziehungen (4.2) vererben sich die üblichen Rechenregeln wie Assoziativität, Kommutativität und Distributivitätsgesetz von den ganzen Zahlen auf Kongruenzen. Überaschende Dinge passieren allerdings bei der Division. Dividiert man beide Seiten einer Gleichung durch eine von Null verschiedene ganze Zahl, so bleibt die Gleichung erhalten. Die entsprechende Äquivalenz für Kongruenzen

$$ab \equiv ac \bmod M \quad \Longleftrightarrow \quad b \equiv c \bmod M \qquad \forall a \neq 0 \tag{4.3}$$

ist im allgemeinen falsch! Beispielsweise ist nämlich

$$2 \cdot 2 \equiv 4 \equiv 2 \cdot 0 \bmod 4$$

und aus (4.3) würde der Widerspruch $2 \equiv 0 \bmod 4$ folgen. Aber die Sache lässt sich klären.

Satz 4.8 (Kürzungsregel). *Die Kürzungsregel* (4.3) *gilt genau dann, wenn a und M teilerfremd sind.*

Beweis. Es seien a und M teilerfremd sowie $ab \equiv ac \bmod M$. Nach Definition 4.6 gibt es dann ein $z \in \mathbb{Z}$, so daß

$$a(b-c) = ab - ac = zM$$

vorliegt. Da a und M teilerfremd sind, muss also $a|z$ gelten. Division durch a liefert

$$b - c = z'M, \qquad z' = z/a \in \mathbb{Z}$$

und damit $b \equiv c \bmod M$. Die umgekehrte Implikation in (4.3) ist wegen (4.2) immer richtig.

Nun setzen wir voraus, daß (4.3) vorliegt, insbesondere für eine ganze Zahl $a > 1$. Im Widerspruch zur Behauptung nehmen wir an, daß es eine ganze Zahl $b < M$ mit der Eigenschaft $ab = M$ gibt. Dann ist $ab \equiv 0 \bmod M$ und aus (4.3) folgt $b \equiv 0 \bmod M$ im Widerspruch zu $b < M$. □

Wegen (4.2) hat die Kongruenz

$$a + x \equiv 0 \bmod M \qquad a \in \mathbb{Z}$$

die eindeutig bestimmte Lösung $x \equiv -a \bmod M$. Die Frage nach der Existenz von Lösungen der Kongruenz

$$ax \equiv 1 \bmod M \tag{4.4}$$

ist etwas schwieriger zu beantworten. Entsprechende Aufgaben für ganze Zahlen sind im allgemeinen nicht lösbar, sondern führen zur Konstruktion von rationalen Zahlen. Wie sich herausstellen wird, liegen die Dinge bei Kongruenzen anders.

Zunächst notieren wir das folgende klassische Resultat, das erst im nächsten Abschnitt bewiesen wird.

Lemma 4.9 (Lemma von Bachet, 1624). *Zu gegebenen Zahlen $a, b \in \mathbb{Z}$ gibt es Zahlen $x, y \in \mathbb{Z}$ mit der Eigenschaft*

$$ax + by = \mathrm{ggT}(a,b)\ .$$

Nun sind wir soweit.

Satz 4.10. *Sind a und M teilerfremd, so hat die Aufgabe* (4.4) *eine bis auf Kongruenz eindeutig bestimmte Lösung.*

Beweis. Wegen $\mathrm{ggT}(a,M) = 1$ gibt es nach Lemma 4.9 ganze Zahlen x, y mit der Eigenschaft $ax + My = 1$. Nun folgt unmittelbar

$$1 \equiv ax + My \equiv ax \bmod M\ .$$

Wir nehmen an, daß es zwei Lösungen x, x' gibt. Dann muss $ax' \equiv ax \bmod M$ gelten. Auf diese Kongruenz können wir wegen $\mathrm{ggT}(a,M) = 1$ die Kürzungsregel (4.3) anwenden (siehe Satz 4.8) und erhalten $x' \equiv x \bmod M$. □

Ist $M = p$ eine Primzahl, so gibt es nach Satz 4.10 für *jedes* $a \neq 0$ eine Lösung von (4.4), also eine multiplikative Inverse a^{-1} von a. Nun ist der Weg frei in die wunderbare Welt der algebraischen Zahlentheorie [5, 27], den wir hier aber nicht weiter gehen werden. Zum Abschluß dieses Abschnitts notieren wir nur noch einen weiteren Klassiker, der uns später noch gute Dienste tun wird.

Satz 4.11 (Kleiner Satz von Fermat). *Ist p eine Primzahl und kein Teiler von $a \in \mathbb{Z}$, so gilt*

$$a^{p-1} \equiv 1 \bmod p\ .$$

Beweis. Wir betrachten die $p-1$ verschiedenen Zahlen $a, 2a, \ldots, (p-1)a$. Als erstes zeigen wir, daß es zu jedem $j \in \{1,\ldots,p-1\}$ genau ein $k \in \{1,\ldots,p-1\}$ gibt, so daß

$$ja \equiv k \bmod p \tag{4.5}$$

vorliegt. Im Widerspruch zur Behauptung nehmen wir an, daß wir $j \neq j'$ und $k \in \{1,\ldots,p-1\}$ finden können, so daß

$$j \cdot a \equiv k \equiv j' \cdot a \bmod p \tag{4.6}$$

gilt. Da p eine Primzahl und kein Teiler von a ist, muss $\mathrm{ggT}(a,p) = 1$ sein. Deshalb können wir nach Satz 4.8 die Kürzungsregel (4.3) anwenden und erhalten

$$j \equiv j' \bmod p\ .$$

Wegen $j, j' \in \{1,\ldots,p-1\}$ ist daher $j = j'$ im Widerspruch zu $j \neq j'$. Unsere Behauptung (4.5) ist also richtig.

In Anwendung der Multiplikationsregel (4.2) können wir jetzt die jeweils zusammengehörenden Kongruenzen (4.5) für $j = 1,\ldots,p-1$ multiplizieren und erhalten

$$1 \cdot 2 \cdots (p-1) \equiv a \cdot 2a \cdots (p-1)a \equiv a^{p-1} 1 \cdot 2 \cdots (p-1) \bmod p\ .$$

Da $1 \cdot 2 \cdots (p-1) = (p-1)!$ und die Primzahl p teilerfremd sind, können wir diese Kongruenz nach Satz 4.8 durch $(p-1)!$ dividieren und erhalten die Behauptung. □

4.3 Der Euklidische Algorithmus

Der Euklidische Algorithmus beruht auf einem überraschenden Zusammenhang zwischen dem größten gemeinsamen Teiler und der Modulo-Funktion.

Lemma 4.12. *Für alle positiven, ganzen Zahlen a, $b \neq 0$ gilt*

$$\mathrm{ggT}(a,b) = \mathrm{ggT}(b, \bmod(a,b))\ .$$

Beweis. Durch elementare Umformung der Definition $\bmod(a,b) = a - \lfloor a/b \rfloor b$ der Modulo-Funktion ergibt sich

$$a = \lfloor a/b \rfloor b + \bmod(a,b)\ .$$

Unter Verwendung dieser Gleichung erhält man

$$d|a \text{ und } d|b \quad \Longleftrightarrow \quad d|b \text{ und } d|\bmod(a,b)\ .$$

Die Mengen aller gemeinsamen Teiler von a, b und b, $\bmod(a,b)$ stimmen also überein und damit auch deren größtes Element. □

Die Bedeutung des Lemmas 4.12 liegt darin, daß wir mit seiner Hilfe unser Ausgangsproblem äquivalent in ein kleineres Problem umformen können: Wegen $b \leq a$ und $\bmod(a,b) < b$ haben wir es bei der

Berechnung von $\mathrm{ggT}(b, \mathrm{mod}\,(a,b))$ mit kleineren Zahlen zu tun als vorher. Und falls $\mathrm{mod}\,(a,b) \neq 0$ ist, gibt es keinen Grund, schon nach dem ersten Reduktionsschritt aufzuhören: Durch rekursive Anwendung von Lemma 4.13 erhält man den Euklidischen Algorithmus.

Algorithmus 4.13 (Euklidischer Algorithmus).

```
function ggt=euklid(a,b)

while b > 0
    r = mod(a,b);
    a = b;
    b = r;
end

ggt=a;
```

Wir illustrieren den Ablauf anhand eines Beispiels:

Eingabe:	$a = 24128,$	$b = 3219$
1. Schritt:	$a = 3219,$	$b = 1595 = \mathrm{mod}\,(24128, 3219)$
2. Schritt:	$a = 1595,$	$b = 29 = \mathrm{mod}\,(3219, 1595)$
3. Schritt:	$a = 29,$	$b = 0 = \mathrm{mod}\,(1595, 29)$
Ergebnis:	$\mathrm{ggT} = 29$	

Nach diesem ermutigenden Auftakt wollen wir untersuchen, ob der Euklidische Algorithmus für alle zulässigen Eingaben $0 \leq b \leq a$ und $0 < ab$ korrekt arbeitet. Im Falle $b = 0$ erhält man unmittelbar das richtige Ergebnis $\mathrm{ggT}(a,0) = a$. Andernfalls wird b wegen $0 \leq \mathrm{mod}\,(a,b) < b$ bei jedem Reduktionsschritt, d.h. bei jedem Durchlauf der `while`-Schleife, mindestens um eins reduziert. Nach Lemma 4.12 ändert sich der größte gemeinsame Teiler dabei nicht. Nach höchstens b Schritten ist daher $b = 0$ erreicht und der Euklidische Algorithmus terminiert mit dem richtigen Ergebnis.

Nun wollen wir den Aufwand T_E des Euklidischen Algorithmus abschätzen. Da in jedem Schritt eine Division mit Rest ausgeführt wird, haben wir dazu die maximal benötigte Anzahl der Reduktionsschritte zu bestimmen. Die exponentielle Schranke $T_E(n) \leq 10^n$ haben wir schon nebenbei aus unseren obigen Überlegungen zur Termination des Euklidischen Algorithmus erhalten. Wir wollen versuchen, diese Schranke zu verbessern.

Dabei spielen überraschenderweise die sogenannten *Fibonacci-Zahlen* F_k, $k = 0, 1, \ldots$, eine zentrale Rolle. Sie sind durch die Drei-Term-Rekursion

$$\begin{aligned} F_0 = 0, \qquad F_1 = 1, \\ F_{k+1} - F_{k-1} - F_k = 0, \qquad k = 1, 2, 3, \ldots \end{aligned} \tag{4.7}$$

charakterisiert (siehe auch Abschnitt A.9). Offenbar gilt $F_{k+1} > F_k > 0$ für alle $k = 2, 3, \ldots$. Den entscheidenden Zusammmenhang mit dem Euklidischen Algorithmus beinhaltet folgendes Lemma.

Lemma 4.14. *Es sei $a \geq b \geq 0$ und $ab > 0$. Terminiert der Euklidische Algorithmus nach genau $k \geq 0$ Schritten, so gilt $a \geq F_{k+1}$ und $b \geq F_k$.*

Beweis. Wir führen einen Induktionsbeweis über k und beginnen mit dem Induktionsanfang. Terminiert der Euklidische Algorithmus nach $k = 0$ Schritten, so muss $b = 0 = F_0$ gelten. Nach Voraussetzung ist dann $a > 0$ also $a \geq 1 = F_1$.

Nun gehen wir davon aus, daß die Behauptung für ein $k - 1 \geq 0$ richtig ist und zeigen, daß sie dann auch für k gelten muss. Im ersten Schritt werden $m = b$ und $n = \mathrm{mod}\,(a,b)$ berechnet. Nach Voraussetzung sind zur Bestimmung von $\mathrm{ggT}(m,n)$ dann noch weitere $k-1$ Schritte nötig. Nach Induktionsvoraussetzung gilt deshalb

$$b = m \geq F_k, \qquad \mathrm{mod}\,(a,b) = n \geq F_{k-1}\,. \tag{4.8}$$

Wir haben also nur noch $a \geq F_{k+1}$ zu zeigen. Aus $a \geq b > 0$ folgt $\lfloor a/b \rfloor \geq 1$ und damit

$$b + \mathrm{mod}\,(a,b) = b + (a - \lfloor a/b \rfloor b) \leq a.$$

Zusammen mit der zweiten Abschätzung in (4.8) und der Drei-Term-Rekursion (4.7) erhalten wir

$$a \geq b + \bmod(a,b) \geq F_k + F_{k-1} = F_{k+1}.$$

und damit die Behauptung. □

Nun können wir die Anzahl der Schritte durch Einordnung von b in die Folge der Fibonacci-Zahlen F_k abschätzen.

Satz 4.15 (Lamé, 1848). *Es sei $a \geq b \geq 0$ und $ab > 0$. Ist $b < F_{k+1}$, so terminiert der Euklidische Algorithmus nach höchstens k Schritten.*

Beweis. Würden im Widerspruch zur Behauptung $j \geq k+1$ Schritte gebraucht, so müsste nach Lemma 4.14 $b \geq F_j \geq F_{k+1}$ sein. Das ist ein Widerspruch zur Voraussetzung $b < F_{k+1}$. □

Nach dem Satz von Lamé haben wir zu einem gegebenen b die kleinste Finonacci-Zahl F_{k+1} mit der Eigenschaft $F_{k+1} > b$ zu bestimmen, um die obere Schranke k für den Aufwand des Euklidischen Algorithmus zur Berechnung von $\mathrm{ggT}(a,b)$ zu erhalten. Dazu verwenden wir die *Formel von Moivre-Binet* (siehe Satz A.41)

$$F_k = \frac{1}{\sqrt{5}}\left(\phi^k - (1-\phi)^k\right), \qquad \phi = \frac{1+\sqrt{5}}{2}\,. \tag{4.9}$$

Nun können wir die gesuchte Aufwandsschranke angeben (siehe Definition 4.4).

Satz 4.16. *Der Aufwand des Euklidische Algorithmus' 4.13 genügt der Abschätzung*

$$T_E(n) \leq 1 + \log(\phi)^{-1} n\,, \qquad n = 0,1,\ldots,\,. \tag{4.10}$$

Beweis. Offenbar ist $T_E(0) = 0 < 1$. Es sei also $b \geq 1$ eine n–stellige ganze Zahl und $a \geq b$. Wir wählen k so, daß $F_k \leq b < F_{k+1}$ vorliegt. Nach dem Satz von Lamé terminiert der Euklidische Algorithmus 4.13 dann nach höchstens k Schritten. Es reicht also zu zeigen, daß $k \leq 1 + \log(\phi)^{-1}\, n$ gilt.

Durch geschicktes Umformen erhalten wir

$$\begin{aligned} F_k &= \frac{1}{\sqrt{5}}\left(\phi^k - (1-\phi)^k\right) = \frac{\phi^{k-1}}{\sqrt{5}}\left(\phi^2 - \left(\frac{1-\phi}{\phi}\right)^{k-1}(1-\phi)^2\right) \\ &\geq \frac{\phi^{k-1}}{\sqrt{5}}\left(\phi^2 - \left|\frac{1-\phi}{\phi}\right|^{k-1}(1-\phi)^2\right) \geq \frac{\phi^{k-1}}{\sqrt{5}}\left(\phi^2 - (1-\phi)^2\right) \\ &= \frac{\phi^{k-1}}{\sqrt{5}}\,(2\phi - 1) = \phi^{k-1}\,. \end{aligned}$$

Dabei haben wir die Ungleichungen $x - y \geq x - |y|\,,\ \forall x,y \in \mathbb{R}$ und $|\frac{1-\phi}{\phi}| < 1$ verwendet. Der Logarithmus ist eine monoton wachsende Funktion. Also gilt

$$\begin{aligned} k = \log_\phi(\phi^k) &= \log_\phi(\phi \cdot \phi^{k-1}) = \log_\phi(\phi) + \log_\phi(\phi^{k-1}) \\ &\leq 1 + \log_\phi(F_k) \leq 1 + \log_\phi(b)\,. \end{aligned}$$

Schließlich folgt aus den Potenzrechenregeln

$$k \leq 1 + \log_\phi(b) = 1 + \log(\phi)^{-1}\log(b) \leq 1 + \log(\phi)^{-1}(1 + \lfloor \log(b) \rfloor)\,.$$

und damit die Behauptung.

□

Der Euklidische Algorithmus hat also *linearen Aufwand* $1 + \log(\phi)^{-1} n = \mathcal{O}(n)$. Das ist eine dramatische Verbesserung gegenüber Algorithmus 4.3.

Der Euklidische Algorithmus ist exponentiell schneller als Ausprobieren.

n	b	mod –Aufrufe	obere Schranke
2	95	1	10
4	4853	11	20
6	800280	12	29
8	14188633	9	39
10	4217612826	16	48
12	915735525189	26	58
14	79220732955955	24	67
16	9594924263929030	29	77

Tabelle 4.2 Vergleich der tatsächlichen Aufrufe von mod mit der theoretischen oberen Schranke $1+\lfloor \log(\phi)^{-1} n \rfloor$.

Zum Vergleich der *tatsächlich benötigten Aufrufe* von mod mit der *theoretischen oberen Schranke* $1+\lfloor \log(\phi)^{-1} n \rfloor$ aus Satz 4.16 setzen wir

$$a = 12345678901234567890$$

und berechnen $\mathrm{ggT}(a,b)$ für wachsende, zufällig gewählte Werte von b. Die Ergebnisse sind in Tabelle 4.2 dargestellt. Die benötigten Aufrufe von mod bleiben jeweils deutlich unter der oberen Schranke. Diese lässt sich trotzdem nicht wesentlich verbessern. Im allgemeinen gilt nämlich (siehe Aufgabe 4.8)

$$\log(\phi)^{-1}(n-1) - 1 \le T_E(n) \; .$$

Übrigens erhält man $\mathrm{ggT}(a,b) = 2$ im letzten Beispiel von Tabelle 4.2 ($n = 16$). Sowohl TumbGGT als auch die verbesserte Variante hätten also etwa $2 \cdot 10^{16}$ Aufrufe von mod gebraucht, nicht nur 29.

Zum Abschluß wollen wir noch eine Erweiterung des Euklidischen Algorithmus vorstellen, die einen konstruktiven Beweis des Lemmas 4.9 von Bachet liefert. Ausgangspunkt ist folgende Beobachtung.

Lemma 4.17. *Aus $n = xa + yb$ und $m = x'a + y'b$, folgt*

$$\mathrm{mod}\,(n,m) = (x - qx')a + (y - qy')b, \quad q = (n - \mathrm{mod}\,(n,m))/m \; . \tag{4.11}$$

Beweis. Einsetzen und Ausklammern liefert

$$\mathrm{mod}\,(n,m) = n - qm = xa + yb - q(x'a + y'b) = (x - qx')a + (y - qy')b \; .$$

□

Die Vorschrift (4.11) beschreibt, wie man vorhandene Darstellungen von $n = xa + yb$ und $m = x'a + y'b$ in jedem Reduktionsschritt des Euklidischen Algorithmus vererbt. Ausgehend von den Anfangsdarstellungen $a = 1 \cdot a + 0 \cdot b$ und $b = 0 \cdot a + 1 \cdot b$ erhält man so induktiv entsprechende Darstellungen für jedes Zwischenergebnis und dann im letzten Schritt die gewünschte Darstellung von $\mathrm{ggT}(a,b)$. Damit ist Lemma 4.9 bewiesen.

Algorithmus 4.18 (Erweiterter Euklidischer Algorithmus).

```
function [ggt,x,y]=extendedeuklid(a,b)

x=1;  y=0;  x'=0;  y'=1;

while b > 0

    r = mod(a,b);  rx = x;          ry = y;   q = (a - r)/b;
    a = b;         x  = x';         y  = y';
    b = r;         x' = rx - q*x';  y' = ry - q*y';

end

ggt=a;
```

Wie wir im Beweis zu Satz 4.10 gesehen haben, kann im Falle teilerfremder ganzer Zahlen a und M die Lösung x der Kongruenz $ax \equiv 1 \bmod M$ berechnet werden, indem man eine Darstellung der Form

$1 = \mathrm{ggT}(a,M) = xa + yM$ findet. Das wollen wir jetzt für $a = a_0 = 121$ und $M = b_0 = 111$ mit dem erweiterten Euklidischen Algorithmus tun.

$$\begin{array}{lrcrcrcrcr}
\text{Initialisierung:} & a &=& 121 &=& a_0, & b &=& 111 &=& b_0 \\
\text{1. Schritt:} & a &=& 111 &=& b_0, & b &=& 10 &=& a_0 - b_0 \\
\text{2. Schritt:} & a &=& 10 &=& a_0 - 1b_0, & b &=& 1 &=& -11a_0 + 12b_0 \\
\text{3. Schritt:} & a &=& 1 &=& -11a_0 + 12b_0, & b &=& 0 &=& 111a_0 - 121b_0 \\
\text{Ergebnis:} & ggT &=& 1 &=& -11a_0 + 12b_0, & & & & &
\end{array}$$

Die berechnete Lösung $x = -11$ ist nur bis auf Kongruenz modulo 111 eindeutig bestimmt. Wer positive Lösungen vorzieht, kann daher auch den Repräsentanten $100 \equiv -11 \bmod 111$ wählen.

Durch die zusätzlich Buchführung über x und y ändert sich die Anzahl der Reduktionsschritte und damit der Aufwand (im Sinne unseres Aufwandbegriffs) nicht.

4.4 Komplexität und Kryptographie

Der erste Schritt zur Verschlüsselung von Nachrichten besteht darin, jedem Buchstaben des Alphabets eine Zahl zuzuordnen. Am einfachsten ist es, die Buchstaben einfach durchzunummerieren, also

$$\mathrm{A} \to 00, \quad \mathrm{B} \to 01, \quad \mathrm{C} \to 02, \ldots, \mathrm{X} \to 23, \quad \mathrm{Y} \to 24, \quad \mathrm{Z} \to 25\,. \tag{4.12}$$

Aus dem Wort CAESAR[2] wird auf diese Weise die natürliche Zahl

$$x = 02\ 00\ 04\ 18\ 00\ 17\,.$$

Da dieser Code leicht zu knacken ist, müssen die so entstandenen Nachrichten $x \in \mathbb{N}$ weiter verschlüsselt werden. Wir beschreiben dazu das sogenannte RSA-Verfahren, welches auf **R**ivest, **S**hamir und **A**dleman [31] zurückgeht und letztlich darauf beruht, daß gewisse Probleme (noch?) *nicht* effizient genug gelöst werden können.

Öffentlicher Schlüssel. Wir wählen zwei natürliche Zahlen, einen Exponenten e und ein Modul M, welche die Bedingungen

$$\begin{gathered} M = pq, \quad p,q \text{ Primzahlen}, \\ 1 < e < N = (p-1)(q-1) \quad \text{und} \quad \mathrm{ggT}(N,e) = 1 \end{gathered} \tag{4.13}$$

erfüllen. Außerdem soll M größer sein als die größte zulässige Nachricht x. Beschränkt man sich beispielsweise auf Wörter mit 5 Buchstaben, die durch die Vorschrift (4.12) codiert sind, so bedeutet das $M > 2525252525$. Die beiden Zahlen e und M brauchen nicht geheimgehalten werden. und heissen daher *öffentliche Schlüssel*. Die beiden Primzahlen p und q darf allerdings niemand erfahren.

Verschlüsseln. Die verschlüsselte Nachricht y ist

$$y \equiv \bmod (x^e, M)\,.$$

Natürlich ist y öffentlich.

Entschlüsseln. Aus den beiden geheimen Primzahlen p und q erhält man $N = (p-1)(q-1)$ und daraus durch Lösen der Kongruenz

$$e \cdot d \equiv 1 \bmod N \tag{4.14}$$

den *geheimen Schlüssel* d. Wegen Voraussetzung (4.13) und Satz 4.10 ist d bis auf Kongruenz eindeutig bestimmt. Mit Hilfe von d gewinnt man aus y die ursprüngliche Nachricht

$$x = \bmod (y^d, M) \tag{4.15}$$

[2] Julius Caesar benutzte diesen Code, um mit seinen Offizieren zu kommunizieren.

zurück. Wir zeigen nun, daß (4.15) richtig ist, daß also der geheime Schlüssel d auch wirklich passt.

Satz 4.19. *Unter den Voraussetzungen* (4.13) *und* (4.14) *gilt*

$$y \equiv x^e \bmod M \quad \Longleftrightarrow \quad x \equiv y^d \bmod M\,.$$

Beweis. Zunächst sei bemerkt, daß wegen (4.14) ein $m \in \mathbb{Z}$ existiert, so daß

$$ed = mN + 1 = m(p-1)(q-1) + 1\,.$$

Wir zeigen nun $y \equiv x^e \bmod M \Longrightarrow x \equiv y^d \bmod M$. Die Umkehrung folgt analog. Aus $y \equiv x^e \bmod M$ erhält man $y^d \equiv x^{ed} \bmod M$. Es reicht also

$$x^{ed} \equiv x \bmod M \tag{4.16}$$

nachzuweisen. Dazu zeigen wir zunächst

$$x^{ed} \equiv x \bmod p, \qquad x^{ed} \equiv x \bmod q \tag{4.17}$$

und konzentrieren uns dabei auf die erste der beiden Kongruenzen, denn die zweite folgt analog. Da p eine Primzahl ist, muss entweder $\mathrm{ggT}(x,p) = 1$ oder p ein Teiler von x und daher auch von x^{ed} sein. Im zweiten Fall ist $x^{ed} \equiv x \equiv 0 \bmod p$, die Behauptung also erfüllt. Sei also $\mathrm{ggT}(x,p) = 1$. Auf Grund von

$$x^{ed} \equiv x^{m(p-1)(q-1)} \cdot x \bmod p$$

reicht es dann aus

$$\left(x^{m(p-1)(q-1)}\right) \equiv 1 \bmod p$$

nachzuweisen. Wegen $\mathrm{ggT}(x^{m(q-1)},p) = \mathrm{ggT}(x,p) = 1$ folgt diese Kongruenz aber mit $a = x^{m(q-1)}$ aus dem kleinen Satz von Fermat 4.11.

Aus den beiden Kongruenzen (4.17) ergibt sich $p|(x^{ed}-x)$ und $q|(x^{ed}-x)$. Daher ist auch $M = pq$ ein Teiler von $x^{ed}-x$, also $x^{ed} \equiv x \bmod M$. □

Als Beispiel wählen wir den Exponenten $e = 7$ und das Modul $M = 22$. Unsere geheime Nachricht ist der Buchstabe C also $x = 2$. Verschlüsseln liefert

$$y = \bmod(2^7, 22) = \bmod(128, 22) = 18\,.$$

Es ist $M = 2 \cdot 11$ und daher $N = 1 \cdot 10 = 10$. Den geheimen Schlüssel berechnet man also aus

$$7d \equiv 1 \bmod 10\,.$$

Durch Ausprobieren finden wir schnell die Lösung $d = 3$. Nun haben wir den Code geknackt und können die geheime Nachricht entschlüsseln:

$$x = \bmod(18^3, 22) = \bmod(5832, 22) = 2\,.$$

Was soll an einer solchen Verschlüsselung sicher sein?

Um diese Frage zu beantworten bemerken wir zunächst, daß aktuelle minimale Sicherheitsstandards eine Schlüssellänge von 1024 Bits vorschreiben, daß also $M \approx 2^{1024}$ sein soll. Das ist einen Zahl mit über 300 Dezimalstellen. Auch Nachrichten x und Exponenten e sind in der Praxis deutlich größer als in unserem Beispiel. Damit die RSA-Verschlüsselung für solche Größenordnungen überhaupt praktikabel ist, müssen die nötigen Rechnungen bei bekannter Primfaktorisierung $M = pq$ also effizient durchführbar sein.

Wir wollen sehen, ob das der Fall ist und beginnen mit der Verschlüsselung. Wegen (4.2) gilt

$$\bmod(x^d, M) = \bmod(x^{d-1} \bmod (x, M), M)\,.$$

Induktive Anwendung dieser Identität liefert das folgenden Exponentiations-Verfahren.

Algorithmus 4.20 (Langsame Exponentiation).

```
function y=exponentiation(x,e,M)
```

```
y = x;
for i=2:e   y = mod(x*y,M)  end
```

Beispielsweise berechnet dieser Algorithmus im Falle $x = 100$, $M = 187$ und $e = 4$

$$\begin{aligned}
\text{Initialisierung:} \quad & y = 100 \\
\text{i=2:} \quad & y = \mod(100 * 100, 187) = 89 \\
\text{i=3:} \quad & y = \mod(100 * 89, 187) = 11 \\
\text{i=4:} \quad & y = \mod(100 * 11, 187) = 67 \\
\text{Ergebnis:} \quad & 67 = \mod(100^4, 187)
\end{aligned}$$

Natürlich könnte man auch erst $x^4 = 100000000$ und dann $100000000 \equiv 67 \bmod 187$ berechnen. Da auf diese Weise aber unnötig große Zahlen und damit die Gefahr von Overflow entsteht, ist das im allgemeinen keine gute Idee.

Der Aufwand von Algorithmus 4.20 ist offenbar $\mathcal{O}(e)$. Dieses naive Verfahren kann allerdings noch exponentiell beschleunigt werden, indem man den Exponenten e zunächst in das Dualsystem umrechnet, also Koeffizienten $e_i = 0, 1$ mit

$$e = \sum_{i=0}^{k} e_i 2^i \tag{4.18}$$

bestimmt. Anstelle der naiven Zerlegung $x^e = x \cdots x$ in e Faktoren erhält man so die Faktorisierung

$$x^e = x^{\sum_{i=0}^{k} e_i 2^i} = x^{e_k 2^k} \cdot x^{e_{k-1} 2^{k-1}} \cdots x^{e_1 2} \cdot x^{e_0} \tag{4.19}$$

mit $k \leq 1 + \log_2(e)$ Faktoren. Da jeder Faktor einer mod-Auswertung entspricht und auch die Koeffizienten mit maximal k Divisionen mit Rest berechnet werden können, kommt man insgesamt auf einen Aufwand von $\mathcal{O}(1 + \log(e))$ (siehe Aufgabe 4.11). Die resultierende *schnelle Exponentiation* wird bei der Entschlüsselung (4.15) in der Praxis verwendet.

Auch der fehlende geheime Schlüssel d kann bei bekannter Primfaktorzerlegung $M = pq$ und damit bekanntem $N = (p-1)(q-1)$, mit dem erweiterten Euklidischen Algorithmus 4.13 mit linearem Aufwand $\mathcal{O}(1 + \log(e))$ berechnet werden. Die RSA-Verschlüsselung ist also praktikabel.

Aber warum ist sie denn nun sicher? Weil Primzahlen p und q (derzeit) exponentiell schneller gefunden werden können als es umgekehrt möglich ist, die einzelnen Faktoren p und q aus dem Produkt $M = pq$ zurückzugewinnen! Man kann nämlich mit *polynomialem Aufwand* $\mathcal{O}(1 + (\log p)^6)$ feststellen, ob p eine Primzahl ist [1, 26]. Damit ist es möglich, Primzahlen p, q mit hunderten von Dezimalstellen bereit zu stellen. Umgekehrt erfordert die Faktorisierung von $M = pq$ *exponentiellen Aufwand*!

Die RSA-Verschlüsselung ist sicher, solange es keinen polynomialen Algorithmus zur Primzahl-Faktorisierung gibt.

Was exponentieller Aufwand praktisch bedeutet, soll das folgende Beispiel illustrieren. Im Jahre 2010 gelang es, mit geballter Kompetenz und Rechnerleistung (äquivalent zu mehr als 1500 Jahren Rechenzeit eines 2,2 GHz–Prozessors), die 232–stellige Zahl

$$\begin{aligned}
M = \; & 12301866845301177551304949583849627207728535695953347921973224521517264005072636575187452021997864693899 \\
& 56474942774063845925192557326303453731548268507917026122142913461670429214311602221240479274737794080665 \\
& 351419597459856902143413 \,,
\end{aligned}$$

in Fachkreisen bekannt als RSA-768, in ihre Faktoren

$$p = 33478071698956898786044169848212690817704794983713768568912431388982883793878002287614711652531743087737814467999489$$

und

$$q = 36746043666799590428244633799627952632279158164343087642676032283815739666511279233373417143396810270092798736308917$$

zu zerlegen. Die Rechnungen dauerte immerhin 20 Monate!

Wer öffentliche Schlüssel M mit 1024 Bits oder gut 300 Dezimalstellen, verwendet und diese auch noch jährlich wechselt, kann sich derzeit also einigermaßen sicher fühlen. Ob es überhaupt möglich ist, einen polynomialen Algorithmus zur Primfaktorisierung zu finden, ob dieses Problem also polynomiale oder exponentielle Komplexität hat, ist derzeit eine offene Frage. Weitere Einzelheiten über Primzahltests oder Primfaktorisierungen würden an dieser Stelle zu weit führen. Wir verweisen dazu auf die einschlägigen Lehrbücher von Ribbenboim [30], Rempe und Waldecker [29] sowie auf die dort zitierten Literatur.

4.5 Aufgaben

Aufgabe 4.1 *Zeigen Sie, daß durch*

$$x \cong a \quad \Longleftrightarrow \quad x \equiv a \bmod M$$

eine Äquivalenzrelation (reflexiv, transitiv, symmetrisch) definiert wird.

Aufgabe 4.2 *Zeigen Sie*

$$[a]_M = [\bmod(a,M)]_M \qquad \forall a \in \mathbb{Z}\,.$$

Aufgabe 4.3 *Zeigen Sie, daß die Additions- und Multiplikationsregeln* (4.2) *für Kongruenzen modulo M äquivalent zu den Additions- und Multiplikationsregeln*

$$[a]_M + [b]_M = [a+b]_M, \qquad [a]_M \cdot [b]_M = [a \cdot b]_M, \qquad a,b \in \mathbb{Z}\,,$$

für Restklassen sind.

Aufgabe 4.4 *Zeigen Sie, daß das Rechnen im N-Bit-Zweierkomplement dem Rechnen mit den Repräsentanten $-2^{N-1}, \cdots -1, 0, 1, \ldots, 2^{N-1}-1$ der 2^N Restklassen modulo 2^N entspricht. Wenn Rechnungen aus der Menge dieser Repräsentanten herausführen, tritt Overflow oder Underflow auf.*

Aufgabe 4.5 *Wir betrachten eine Variante von Algorithmus 4.3, bei der die Schleife mit $i = b$ beginnend rückwärts durchlaufen und gleich beim ersten (also dem größten) gefundenen gemeinsamen Teiler abgebrochen wird. (a) Zeigen Sie, daß dafür $2(1+b-\mathrm{ggT}(a,b))$ Aufrufe von* mod *nötig sind. (b) Zeigen Sie, daß der Aufwand dieser Variante exponentiell wächst.*

Aufgabe 4.6 *In Euklids Werk* Die Elemente *findet sich die Rechenvorschrift*

> *Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.*

(a) Was wird hier berechnet? Formulieren Sie diese Rechenvorschrift als MATLAB*-Programm.*
(b) Worin bestehen die Vor- und Nachteile dieses Verfahrens gegenüber dem Euklidischen Algorithmus 4.13?

Aufgabe 4.7 *Zeigen Sie, daß aufeinander folgende Fibonacci-Zahlen F_{k+1} und F_k teilerfremd sind, daß also*

$$\mathrm{ggT}(F_{k+1}, F_k) = 1 \qquad \forall k = 0, 1, 2, \ldots$$

gilt.

Aufgabe 4.8 *a) Zeigen Sie, daß der Euklidische Algorithmus zur Berechnung von* $\mathrm{ggT}(F_{k+1},F_k)$ *im Falle* $k \geq 2$ *nach* $k-1$ *Schritten terminiert.*
b) Zeigen Sie die untere Abschätzung

$$\log(\phi)^{-1}(n-1)-1 \leq T_E(n)\ , \quad n = 0,1,2\ldots\ .$$

c) Das bedeutet beispielsweise $T_E(14) \geq 61$ *oder* $T_E(16) \geq 71$. *Wie sind diese Abschätzung mit den Ergebnissen aus Tabelle 4.2 vereinbar?*

Aufgabe 4.9 *Zeigen Sie mit vollständiger Induktion, daß der erweiterte Euklidische Algorithmus 4.18 angewandt auf* $a > b \geq 0$ *eine Darstellung der Form* $1 = xa + yb$ *liefert.*

Aufgabe 4.10 *Geben Sie eine obere Schranke für die Komplexität des Problems ggT - finden Sie den größten gemeinsamen Teiler zweier natürlicher Zahlen an.*

Aufgabe 4.11 *Für ganze Zahlen x, e und M soll die Exponentiation* $y = \mathrm{mod}\,(x^e, M)$ *ausgewertet werden.*
(a) Entwickeln und implementieren sie einen auf der Dualdarstellung (4.18) *und der resultierenden Faktorisierung* (4.19) *beruhenden Algorithmus zur schnellen Exponentiation.*
(b) Vergleichen Sie numerisch den Aufwand der schnellen Exponentiation und des naiven Algorithmus 4.20 durch Berechnung von $\mathrm{mod}\,(x^e, M)$ *für* $x = 999$, $M = 1000$ *und* $e = 10^r$, $r = 1,\ldots 100$.
(c) Bestätigen Sie, daß der Gesamtaufwand T_{SXP} *dieses Verfahrens (inklusive der Umrechenung von e ins Dualsystem) von der Ordnung* $T_{SXP}(n) = \mathcal{O}(n)$ *für alle Eingabedaten e mit n Dezimalstellen und beliebigem x und M ist.*
(d) Gibt es auch eine untere Schranke $T_{SXP}(n) \geq C\,n$? *Begründen Sie Ihre Antwort.*

Aufgabe 4.12 *Durch naives Ausprobieren erhält man einen Algorithmus mit Aufwand von* $\mathcal{O}(M^{1/2})$ *zur Primfaktorisierung von M. Implementieren Sie dieses Verfahren und schätzen Sie ab, wie lange dieser Algorithmus braucht oder brauchen würde, um die Zahl M =*
114381625757888867669235779976146612010218296721242362562 5
618429357069352457338978305971235639587050589890751475992900268795 4
3541 = 3490529510847650949147849619903898133417764638493387843990820
*577 * 3276913299326670954996198819083446141317764296799294253979828*
8533 zu faktorisieren.