Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Angewandte Informatik

Bachelor-Arbeit
zur Erlangung des akademischen Grades
Bachelor of Science – B.Sc.

Vom Kleinen Satz von Fermat zum Lucas-Primzahltest

vorgelegt von Marcell Dietl

am 20.06.2014

Referent: Prof. Dr. Geib
Korreferent: Prof. Dr. Reith

Erklärung gem. ABPO, Ziff. 6.4.3

Ich versichere, dass ich die Bachelor-Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 20.06.2014 Marcell Dietl

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Arbeit:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger		×
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	×	
Veröffentlichung des Titels der Arbeit im Internet	×	
Veröffentlichung der Arbeit im Internet	×	

Wiesbaden, 20.06.2014 Marcell Dietl

Für meine Eltern

Inhaltsverzeichnis

1. Einleitung und Überblick **1**

2. Mathematische Grundlagen **3**
- 2.1. Hauptsatz der elementaren Zahlentheorie 3
- 2.2. Die Unendlichkeit der Primzahlen 4
- 2.3. Primzahlzählfunktion, -satz und -dichte 4

3. Naive Primzahltests **7**
- 3.1. Probedivision 7
  - 3.1.1. Pseudocode 9
  - 3.1.2. Laufzeituntersuchung 9
- 3.2. Sieb des Eratosthenes 10
  - 3.2.1. Pseudocode 12
  - 3.2.2. Laufzeituntersuchung 13

4. Fermatscher Primzahltest **14**
- 4.1. Satz von Euler 14
- 4.2. Kleiner Satz von Fermat 16
  - 4.2.1. Pseudocode 17
  - 4.2.2. Laufzeituntersuchung 19
- 4.3. Fermatsche Pseudoprimzahlen 19
  - 4.3.1. Anzahl und Häufigkeit 22
- 4.4. Carmichael-Zahlen 23
  - 4.4.1. Anzahl und Häufigkeit 24
  - 4.4.2. Korselts Kriterium 25
- 4.5. Primzahlen finden 26
  - 4.5.1. Pseudocode 27
  - 4.5.2. Laufzeituntersuchung 28

5. Lucas-Primzahltest **29**
- 5.1. Primitivwurzeln und primitive Restklassen 29
- 5.2. Variante von 1876: Vorläufer des Lucas-Primzahltests 32
  - 5.2.1. Pseudocode 33
  - 5.2.2. Laufzeituntersuchung 33
- 5.3. Variante von 1891: Lucas-Primzahltest 34
  - 5.3.1. Pseudocode 35
  - 5.3.2. Laufzeituntersuchung 36

5.4. Variante von 1953: Verbesserter Lucas-Primzahltest 37
5.4.1. Pseudocode 39
5.4.2. Laufzeituntersuchung 39
5.4.3. Implementierung ohne Sage 40
5.4.3.1. Pseudocode 40
5.4.3.2. Laufzeituntersuchung 41

6. Pépin-Primzahltest 43
6.1. Quadratische Reste, Nichtreste und das Legendre-Symbol 44
6.1.1. Das Quadratische Reziprozitätsgesetz und das Euler-Kriterium . . 44
6.1.2. Einige Eigenschaften von Fermatschen Primzahlen 46
6.2. Beweis des Pépin-Primzahltests 47
6.3. Pseudocode 48
6.4. Laufzeituntersuchung 49

7. Implementierung 50
7.1. Das Computeralgebrasystem Sage 50
7.2. Die Programmiersprache Python 51
7.2.1. Exponentiation durch wiederholtes Quadrieren 51

8. Fazit und Ausblick 55

A. Notationen 56
A.1. Komplexitätstheorie 56
A.2. Gruppentheorie 57
A.3. Zahlentheorie 57

B. Zahlenfolgen 59

C. Quellcode 60
C.1. Probedivision als Primzahltest 60
C.2. Sieb des Eratosthenes 61
C.3. Fermatscher Primzahltest 61
C.4. Primzahlen finden 62
C.5. Vorläufer des Lucas-Primzahltests 62
C.6. Lucas-Primzahltest 63
C.7. Verbesserter Lucas-Primzahltest 63
C.7.1. Implementierung ohne Sage 64
C.8. Pépin-Primzahltest 65
C.9. Modulare Exponentiation 65

D. Zusammenfassung 66

Literaturverzeichnis 68

Tabellenverzeichnis

2.1. Werte von $\pi(x)$ und $x/\ln x$ bzw. $\lfloor x/\ln x \rfloor$ im Vergleich. 5

3.1. Laufzeituntersuchung der Probedivision als Primzahltest (Listing C.1). . . 9
3.2. Laufzeituntersuchung des Siebs des Eratosthenes (Listing C.2). 13

4.1. Laufzeituntersuchung des Fermatschen Primzahltests (Listing C.3). 19
4.2. Werte von $\pi(x)$ und $P\pi(x)$ im Vergleich. 22
4.3. Werte von $\pi(x)$, $P\pi(x)$ und $C(x)$ im Vergleich. 24
4.4. Laufzeituntersuchung von Listing C.4 (Primzahlen finden). 28

5.1. Laufzeituntersuchung des Vorläufers des Lucas-Primzahltests (Listing C.5). 33
5.2. Laufzeituntersuchung des Lucas-Primzahltests (Listing C.6). 36
5.3. Laufzeituntersuchung des verbesserten Lucas-Primzahltests (Listing C.9). 41

6.1. Laufzeituntersuchung des Pépin-Primzahltests (Listing C.10). 49

7.1. Laufzeituntersuchung von Listing C.11 (Modulare Exponentiation). 54

Abbildungsverzeichnis

2.1. Werte von $\pi(x)$ und $x/\ln x$ im Vergleich. 6

5.1. Die kleinsten positiven Primitivwurzeln g_p mod der Primzahlen $p < 10^4$. . 31

A.1. Die ersten hundert bzw. tausend Werte der Eulerschen Phi-Funktion. . . . 58

Algorithmenverzeichnis

1. Probedivision als Primzahltest . . . 9
2. Sieb des Eratosthenes . . . 12
3. Fermatscher Primzahltest . . . 17
4. Primzahlen finden . . . 27
5. Vorläufer des Lucas-Primzahltests . . . 33
6. Lucas-Primzahltest . . . 35
7. Verbesserter Lucas-Primzahltest . . . 39
8. Probedivision als Faktorisierungsverfahren . . . 41
9. Pépin-Primzahltest . . . 48
10. Modulare Exponentiation . . . 52

1. Einleitung und Überblick

Seit Jahrtausenden versuchen Mathematiker wie Nicht-Mathematiker den Primzahlen

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

ihre Geheimnisse zu entlocken. Und oft genug waren diese Versuche nicht von Erfolg gekrönt. Einige bedeutende Resultate konnten im Laufe der Zeit jedoch erzielt werden:

Bspw. wusste schon Euklid (um 300 v. Chr.), dass es unendlich viele Primzahlen geben muss. Der Beweis, dass sich jede natürliche Zahl größer 1 – eindeutig – aus ihnen zusammensetzt, wurde schließlich von Gauß erbracht – mehr als 2000 Jahre nach Euklid, im Jahre 1801. In Kapitel 2 werden diese und weitere Erkenntnisse formal eingeführt, bewiesen bzw. auf Beweise verwiesen und mit Zahlenbeispielen untermauert.

Aufbauend auf diesen mathematischen Grundlagen, die die Basis der gesamten Arbeit bilden, werden in Kapitel 3 die ersten – naiven – Primzahltests präsentiert: die Probedivision und das Sieb des Eratosthenes. Beides Verfahren, die mit Sicherheit sagen können, ob eine Zahl prim oder zusammengesetzt ist. Beides Verfahren, die im Rahmen komplexerer Primzahltests noch immer nützlich sein können. Aber auch beides Verfahren, deren Laufzeit schon bei „kleinen“ Eingabewerten zu wünschen übrig lässt.

Um doch noch zu einem Primzahltest zu gelangen, der auch mit „großen“ Eingabewerten umgehen kann, wird in Kapitel 4 ein gänzlich anderer Ansatz gewählt: Kongruenzen als Grundlage von Primzahltests. Auf diesem Wege und mit Hilfe des Kleinen Satzes von Fermat wird es erstmals möglich, eine Entscheidung bzgl. der Primalität von Zahlen mit Hunderten Dezimalstellen zu fällen – und das im Bruchteil einer Sekunde. Leider muss man für diese Laufzeit einen Preis zahlen: Es gibt unendlich viele Zahlen, die zusammengesetzt sind, aber als Primzahl akzeptiert werden: Fermatsche Pseudoprimzahlen.

Um diesem Defizit Rechnung zu tragen, werden in Kapitel 5 drei Primzahltests dargelegt, die als Umkehrung des Kleinen Satzes von Fermat nicht auf zusammengesetzte Zahlen hereinfallen: der Vorläufer des Lucas-Primzahltests, der Lucas-Primzahltest und der verbesserte Lucas-Primzahltest. Alle drei Varianten fügen dem Kleinen Satz von Fermat eine weitere, jeweils eigene Bedingung hinzu, sodass sie mit Sicherheit sagen können, ob eine Zahl N prim oder zusammengesetzt ist. Diese Bedingung, die bei den beiden letztgenannten Varianten die vollständige Faktorisierung von $N-1$ voraussetzt, geht jedoch wieder auf Kosten der Laufzeit. In einem speziellen Fall allerdings kaum:

Wenn die Faktorisierung von $N-1$ bekannt oder einfach zu ermitteln ist, erweist sich der (verbesserte) Lucas-Primzahltest als effizientes Verfahren, um eine Entscheidung bzgl. der Primalität von N zu treffen. Diese Eigenart nutzt der letzte im Rahmen der vorliegenden Arbeit untersuche Primzahltest: der Pépin-Primzahltest aus Kapitel 6. Ein Primzahltest, der speziell auf Fermat-Zahlen zugeschnitten ist. Jene Zahlen, die sich durch ihr rasantes Wachstum hervortun und sich wie ein roter Faden durch die Geschichte der algorithmischen Zahlentheorie, der Primzahltests und Faktorisierungsverfahren ziehen.

Obwohl – oder gerade weil – alle Kapitel bis einschließlich Kapitel 6 ganz im Zeichen der Mathematik stehen, liegt der Schwerpunkt von Kapitel 7 auf der Implementierung bzw. den dahinterstehenden Technologien, also vor allem dem Computeralgebrasystem Sage und der Programmiersprache Python. Ganz ohne Mathematik geht es dann aber doch nicht: Im letzten Abschnitt von Kapitel 7 wird die Frage beantwortet, weshalb die modulare Exponentiation, also die Berechnung von $b = a^e \bmod N$, die in fast allen nachfolgend präsentierten Algorithmen Verwendung findet, auch bei Zahlen mit Hunderten Dezimalstellen nur den Bruchteil einer Sekunde benötigt, um zum Ende zu gelangen.

Im Anschluss an das Fazit und einen kurzen Ausblick in Kapitel 8 folgt in Anhang A eine Zusammenfassung der wichtigsten im Rahmen dieser Arbeit verwendeten mathematischen Notationen. In Anhang B sind die Nummern und Bezeichnungen all jener Zahlenfolgen von http://oeis.org/ (aufgerufen am 04.04.2014) aufgelistet, die sich im Laufe dieser Arbeit als hilfreich erwiesen haben, z. B. die zuvor abgedruckte Folge der Primzahlen, die nicht größer als 100 sind. Eine Beschreibung, wie die Laufzeit der Primzahltests untersucht wurde und die zugehörigen Quellcodes sind in Anhang C hinterlegt.

2. Mathematische Grundlagen

Bevor mit der Untersuchung von Primzahltests begonnen wird, sollen in den nächsten beiden Abschnitten zwei sehr grundlegende und wichtige Erkenntnisse über Primzahlen dargelegt werden. Einerseits, dass sie die Grundbausteine natürlicher Zahlen darstellen – sich diese aus ihnen zusammensetzen. Andererseits, dass es unendlich viele gibt.

Ersteres konnte Gauß im Jahre 1801 beweisen, siehe hierzu u. a. [Weisstein, E. W., 2007].

Letzteres hatte Euklid bereits um 300 v. Chr. bewiesen, siehe hierzu u. a. [Weisstein, E. W., 2014a] und [ProofWiki, 2012]. Sein antiker Beweis ist Grundlage von Abschnitt 2.2.

In Abschnitt 2.3 wird abschließend noch die Verteilung der Primzahlen betrachtet.

2.1. Hauptsatz der elementaren Zahlentheorie

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

$$N = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k} = \prod_{i=1}^{k} p_i^{e_i},$$

wobei p_i Primzahlen und die Exponenten e_i natürliche Zahlen sind.

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 13]. □

Korollar 2.1.1. *Jede natürliche Zahl $N > 1$ ist durch min. eine Primzahl teilbar.*

Obiges Theorem wird häufig auch *Fundamentalsatz der elementaren Zahlentheorie* bzw. *Fundamentalsatz der Arithmetik* genannt. Die Primzahlen, die Teiler von N sind, nennt man *Primfaktoren* bzw. *Primteiler* von N.

Beispiel. Die Zahl $N = 24$ ist eindeutig durch das Produkt

$$N = 2 \cdot 2 \cdot 2 \cdot 3 = 2^3 \cdot 3$$

bestimmt. Es gibt, bis auf Umordnung der Faktoren, keine andere als die angegebene Folge von Primzahlen, die miteinander multipliziert 24 ergeben. Aufgrund dieser Tatsache bezeichnet man Primzahlen auch als *Atome* der natürlichen Zahlen.

2.2. Die Unendlichkeit der Primzahlen

Wie zuvor bereits erwähnt, bewies Euklid bereits um 300 v. Chr., dass es unendlich viele Primzahlen gibt. Der nachfolgende Beweis zeigt die sehr elementare und dadurch zugleich bestechende Argumentation, die – ganz ähnlich – schon Euklid verwendete.

Lemma 2.2.1. *Wenn* $p \mid a$ *und* $p \mid b \Rightarrow p \mid a - b$, *mit* $a > b$, $p \in \mathbb{P}$ *und* $a, b \in \mathbb{N}$.

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

Die in Lemma 2.2.1 gemachten Einschränkungen bzgl. p, a und b sind für das nachfolgende Theorem ausreichend. Der Beweis ließe sich auch auf $p, a, b \in \mathbb{Z}$ verallgemeinern.

Theorem 2.2.1. *Es gibt unendlich viele Primzahlen.*

Beweis. Angenommen die Menge der Primzahlen $\mathbb{P} = \{2, 3, 5, 7, \ldots, p\}$ sei lediglich endlich und somit p die größte existierende Primzahl. Nun können diese endlich vielen Primzahlen miteinander multipliziert und das Ergebnis anschließend um 1 erhöht werden:

$$N = (2 \cdot 3 \cdot 5 \cdot 7 \cdot \ldots \cdot p) + 1$$

Ist N eine Primzahl, so fehlt sie in der Menge $\mathbb{P}$, da sie größer als die größte Primzahl p ist (Widerspruch!). Wenn N keine Primzahl, sondern zusammengesetzt ist, dann muss es gem. Korollar 2.1.1 eine Primzahl q geben, die N teilt. Diese Primzahl teilt auch $N - 1 = (2 \cdot 3 \cdot 5 \cdot 7 \cdot \ldots \cdot p)$. Gem. Lemma 2.2.1 teilt eine Primzahl, die zwei verschiedene natürliche Zahlen teilt, auch deren Differenz. Folglich muss q auch $N - (N - 1) = 1$ teilen. Da es keine Primzahl gibt, die 1 teilt, ist $\mathbb{P}$ nicht endlich (Widerspruch!).

Schlussfolgernd muss es unendlich viele Primzahlen geben. □

2.3. Primzahlzählfunktion, -satz und -dichte

Aufgrund der in Abschnitt 2.1 und Abschnitt 2.2 gewonnenen Erkenntnisse über Primzahlen, stellt sich unweigerlich die Frage nach deren Verteilung. Konkret: Wie viele Primzahlen gibt es, die kleiner oder gleich einer beliebigen natürlichen Zahl sind? Zur Untersuchung dieser Fragestellung hat sich folgende Notation etabliert:

Definition 2.3.1. Die Primzahlzählfunktion

$$\pi(x) = \#\{p \leq x \mid p \in \mathbb{P}\}$$

gibt an, wie viele Primzahlen existieren, die nicht größer als $x \in \mathbb{N}$ sind.

Beispiel. Unter den ersten zehn natürlichen Zahlen finden sich genau vier Primzahlen: 2, 3, 5 und 7. Folglich ist $\pi(10) = 4$. Da 2 per definitionem die kleinste Primzahl ist, folgt ferner, dass $\pi(0) = \pi(1) = 0$.

Sofort drängt sich die Frage auf, wie man den exakten Wert von $\pi(x)$ ermittelt. Die offensichtlichste Möglichkeit besteht darin, alle Primzahlen, die nicht größer als x sind, zu ermitteln, z. B. mit Hilfe der Probedivision, siehe Abschnitt 3.1, oder dem Sieb des Eratosthenes, siehe Abschnitt 3.2. Bei größer werdendem x wird dieser Ansatz jedoch immer ineffizienter werden.

Manchmal ist es jedoch gar nicht notwendig, den exakten Wert von $\pi(x)$ zu kennen. Eine „nicht allzu schlechte“ Schätzung wäre ausreichend, um die Verteilung der Primzahlen zu untersuchen. Die Funktion $f(x) = x/\ln x$ erfüllt genau diese Bedingung:

Theorem 2.3.1. *$\pi(x)$ und $x/\ln x$, mit $x \in \mathbb{N}$, verhalten sich asymptotisch gleich:*

$$\pi(x) \sim x/\ln x$$

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 365–377]. □

Obiges Theorem bezeichnet man als *Primzahlsatz*. Die darin eingeführte Notation besagt, dass $\pi(x)$ geteilt durch $x/\ln x$ bei größer werdendem x gegen 1 strebt:

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1$$

Vereinfacht ausgedrückt bedeutet Theorem 2.3.1, dass $f(x) = x/\ln x$ eine „nicht allzu schlechte“ Schätzung für die Primzahlzählfunktion darstellt – zumindest für großes x und obwohl die Differenz beider Funktionen durchaus unendlich groß werden kann.

Mit Hilfe einer Tabelle lässt sich diese Aussage leicht verdeutlichen:

x	$\pi(x)$	$\lfloor x/\ln x \rfloor$	$\pi(x) - \lfloor x/\ln x \rfloor$	$\frac{\pi(x)}{x/\ln x}$
10^1	4	4	0	0,921
10^2	25	21	4	1,151
10^3	168	144	24	1,161
10^4	1 229	1 085	144	1,132
10^5	9 592	8 685	907	1,104
10^6	78 498	72 382	6 116	1,084
10^7	664 579	620 420	44 159	1,071
10^8	5 761 455	5 428 681	332 774	1,061
10^9	50 847 534	48 254 942	2 592 592	1,054

Tabelle 2.1.: Werte von $\pi(x)$ und $x/\ln x$ bzw. $\lfloor x/\ln x \rfloor$ im Vergleich.

In Tabelle 2.1 ist gut erkennbar, dass der Quotient aus Primzahlzählfunktion und $x/\ln x$ gegen 1 strebt, genau so, wie es der Primzahlsatz, siehe Theorem 2.3.1, voraussagt.

Die Werte wurden mit Hilfe von Sage, siehe Abschnitt 7.1, berechnet. Für x bis einschließlich $\pi(10^{23})$ siehe u. a. [Ribenboim, P., 2011, S. 181] und [Weisstein, E. W., 2014c].

Grafisch besonders ansprechend ist die Betrachtung der Primzahlzählfunktion in einem kleinen Intervall, nachfolgend z. B. zwischen 100 und 500, da hierbei gut erkennbar ist, dass $\pi(x)$ sprunghaft wächst. Jeder Anstieg entspricht einer weiteren Primzahl. Verläuft der Graph horizontal, entspricht dies einer Folge zusammengesetzter Zahlen. Daher gilt:

$$\pi(x+1) = \pi(x) + 1 \Leftrightarrow (x+1) \in \mathbb{P}$$

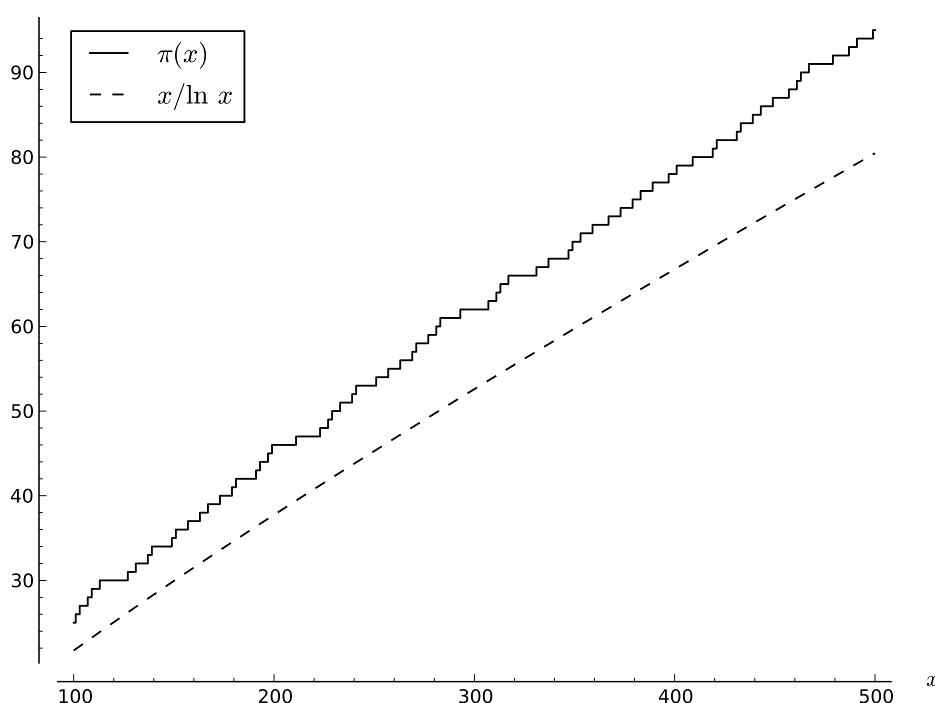


Abbildung 2.1.: Werte von $\pi(x)$ und $x/\ln x$ im Vergleich.

Ein letztes Detail, das in [Crandall, R.; Pomerance, C., 2005, S. 8] gleich zu Anfang erwähnt wird, soll hier nicht unerwähnt bleiben: Primzahlen sind selten und werden immer seltener. Formal heißt dies, dass $\pi(x) = o(x)$, was folgendem Grenzwert entspricht:

$$\lim_{x\to\infty} \frac{\pi(x)}{x} = 0$$

Die Größe $\pi(x)/x$ bezeichnet man üblicherweise als *Primzahldichte*.

3. Naive Primzahltests

Gem. Theorem 2.1.1 ist jede natürliche Zahl $N > 1$ ein Produkt von Primzahlen. Ein intuitiver Ansatz, um zu erkennen, ob es sich bei einer gegebenen Zahl $N > 1$ um eine Primzahl handelt, besteht nun darin, N in seine Primfaktoren zu zerlegen – zu *faktorisieren* – und folgende Fallunterscheidung vorzunehmen:

Besteht N nur aus einem einzigen Primfaktor p, der nicht mehrfach vorkommt, so ist $N = p$ eine Primzahl. Andernfalls ist N zusammengesetzt.

Dieser Ansatz wird in Abschnitt 3.1 zu einem ersten Primzahltest weiterentwickelt. In Abschnitt 3.2 wird ein antikes Verfahren, das *Sieb des Eratosthenes*, präsentiert, mit dem sich alle Primzahlen unterhalb einer frei wählbaren Schranke ermitteln lassen.

3.1. Probedivision

Eines gleich vorweg: Die Probedivision wird häufig nicht als Primzahltest, sondern als Faktorisierungsverfahren verwendet. Bei Letzterem handelt es sich um eine Abfolge von Rechenschritten, die zu einer natürlichen Zahl $N > 1$ die Folge der Primzahlen ermittelt, die miteinander multipliziert N ergeben, siehe Abschnitt 2.1.

Nachfolgend soll die Probedivision daher zuerst als Faktorisierungsverfahren eingeführt werden, da sich dann die Verwendung als Primzahltest leichter nachvollziehen lässt – vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 118–120].

Die grundlegende Idee der Probedivision ist einfach: Überprüfe, ob und wie oft eine Zahl N durch 2 teilbar ist. Vermerke das Ergebnis und fahre anschließend mit der nächsten Primzahl fort. Theorem 2.1.1 garantiert, dass dieses deterministische Verfahren stets terminieren und N vollständig faktorisieren muss. Das Problem besteht jedoch darin, dass alle Primzahlen, durch die N geteilt werden soll, bekannt sein müssen. Entweder indem diese fortlaufend mit Hilfe eines Primzahltests ermittelt werden oder aber als zuvor angelegte Primzahltabelle zur Verfügung stehen. Bei größer werdendem N wird dieser Ansatz immer mehr Rechenzeit bzw. Speicherplatz verbrauchen.

Zudem wird diese Variante offensichtlich nicht als eigenständiger Primzahltest funktionieren können, verlangt sie doch das Vorhandensein eines ebensolchen. Verändert man jedoch ein Detail – die in Erwägung gezogenen Teiler – kann die Probedivision zu einem eigenständigen Primzahltest, siehe Algorithmus 1 auf Seite 9, weiterentwickelt werden:

Statt ausschließlich Primzahlen als Teiler von N zu berücksichtigen, werden die Zahl 2 und alle ungeraden Zahlen als Teiler überprüft. Diese Variante wird keine Primzahl übersehen, da alle Primzahlen außer 2 ungerade sind. Und obgleich auch zusammengesetzte ungerade Zahlen als Primteiler von N überprüft werden, können diese niemals fälschlicherweise als solche erkannt werden. Denn wann immer als Teiler von N eine zusammengesetzte ungerade Zahl d getestet wird, sind deren Primteiler, die kleiner als d und ungerade sind, bereits überprüft und der Algorithmus bei Bedarf beendet worden.

Ein weiteres Detail, das vor allem die Laufzeit betrifft und unnötige Operationen einspart, besteht in einer Abbruchbedingung, die ebenfalls in Algorithmus 1 auf Seite 9 verwendet wird: Sobald als Teiler von N eine Zahl d getestet werden soll, für die $d^2 > N$ bzw. $d > \sqrt{N}$ gilt, ist N mit Gewissheit eine Primzahl. Dies erklärt sich wie folgt:

Theorem 3.1.1. *Sei $N = a \cdot b$ eine zusammengesetzte natürliche Zahl, $a, b \in \mathbb{N}$ seien echte Teiler von N, dann ist min. einer der beiden Teiler von N nicht größer als $\sqrt{N}$.*

Beweis. Angenommen beide Teiler von $N = a \cdot b$ seien größer als $\sqrt{N}$, also $a > \sqrt{N}$ und $b > \sqrt{N}$. Dann gilt aber auch, dass $a \cdot b > N$ (Widerspruch!). □

Obiges Theorem lässt sich auch derart interpretieren, dass zu jedem Teiler von N, der größer als $\sqrt{N}$ ist, ein korrespondierender Teiler existiert, der kleiner als $\sqrt{N}$ ist.

Beispiel. Die kleinste zusammengesetzte Zahl $4 = 2 \cdot 2$ ist eine Quadratzahl. Es gilt $2^2 \leq 4$ wie gefordert. Die kleinste ungerade Zahl, die keine Quadrat- und keine Primzahl ist, lautet 15. Sie besteht aus den Teilern 3 und 5, wobei $5^2 > 15$, aber dafür $3^2 \leq 15$.

Korollar 3.1.1. *Zu jedem N gem. Theorem 3.1.1 existiert min. ein Primteiler von N, der nicht größer als $\sqrt{N}$ ist.*

Ein ebenso direkter wie eleganter Beweis für Korollar 3.1.1 ist in [Scheid, H.; Frommer, A., 2013, S. 7] zu finden. Zur Verdeutlichung sei erwähnt, dass es sich bei den Teilern in Theorem 3.1.1 um Primzahlen handeln kann, aber nicht muss. Wie bereits bewiesen, ist min. einer nicht größer als $\sqrt{N}$. Handelt es sich bei diesem um keine Primzahl, sondern um eine zusammengesetzte Zahl, so besteht diese jedoch aus Primteilern, die alle kleiner als die Zahl selbst sind. Zu guter Letzt ergibt sich noch folgende Erkenntnis, die als Umkehrung von Korollar 3.1.1 die Basis der Probedivision als Primzahltest darstellt:

Korollar 3.1.2. *Existiert zu einer natürlichen Zahl $N > 1$ kein Primteiler von N, der nicht größer als $\sqrt{N}$ ist, dann ist N prim.*

Beispiel. Bei 97 handelt es sich um eine Primzahl, da $2 \nmid 97$, $3 \nmid 97$, $5 \nmid 97$, $7 \nmid 97$ und $11 > \sqrt{97}$ bzw. $11^2 > 97$. Es existiert somit kein Primteiler, der nicht größer als $\sqrt{97}$ ist und demnach ist 97 prim gem. Korollar 3.1.2. Wie bereits erwähnt, würde Algorithmus 1 auf Seite 9 zusätzlich noch überprüfen, ob 9 ein Primteiler von 97 ist, jedoch wegen $9 = 3 \cdot 3$ und $3 \nmid 97$ zu dem Schluss kommen, dass auch $9 \nmid 97$ gilt.

3.1.1. Pseudocode

Algorithmus 1: Probedivision als Primzahltest

Input: Eine natürliche Zahl $N > 2$

1 **if** $2 \mid N$ **then**
2 | **return** „N is definitely composite“
3 **end if**
4 $d \leftarrow 3$
5 **while** $d^2 \leq N$ **do**
6 | **if** $d \mid N$ **then**
7 | | **return** „N is definitely composite“
8 | **end if**
9 | $d \leftarrow d + 2$
10 **end while**
11 **return** „N is definitely prime“

Obiger Pseudocode basiert auf Algorithmus 3.1.1 aus [Crandall, R.; Pomerance, C., 2005, S. 118–119]. Für eine Implementierung siehe Listing C.1 auf Seite 60.

3.1.2. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt.

Primzahl (N)	gefunden mit[1]	Zeit in s
1 000 000 000 000 037	`next_prime(10^15)`	2,057
10 000 000 000 000 061	`next_prime(10^16)`	6,464
100 000 000 000 000 003	`next_prime(10^17)`	20,429
1 000 000 000 000 000 003	`next_prime(10^18)`	64,208
10 000 000 000 000 000 051	`next_prime(10^19)`	509,368
100 000 000 000 000 000 039	`next_prime(10^20)`	1 802,671

Tabelle 3.1.: Laufzeituntersuchung der Probedivision als Primzahltest (Listing C.1).

Zur Untersuchung der Laufzeit der Probedivision als Primzahltest wurden bewusst nur Primzahlen als Eingabewerte herangezogen. Das liegt einerseits daran, dass diese den schlechtestmöglichen Fall der Probedivision darstellen und die meisten – ca. $\sqrt{N}/2$ –

[1] Siehe hierzu u. a.: http://www.sagemath.org/doc/reference/misc/sage/rings/arith.html#sage.rings.arith.next_prime (aufgerufen am 06.05.2014).

Divisionen benötigen, vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 120]. Andererseits ist es aber oft gerade dieser Fall, der min. einmal durchlaufen werden muss, z. .B. bei der Suche nach Primzahlen zur weiteren Verwendung in kryptographischen Verfahren. Dass die Probedivision vergleichsweise große zusammengesetzte Zahlen mit kleinen Primfaktoren schnell als solche erkennen kann, ist immer dann von Vorteil, wenn die Probedivision im Rahmen eines komplexeren Verfahrens nur unvollständig verwendet werden soll, z. B. um eine erste grobe Vorauswahl zu treffen.

Tabelle 3.1 zeigt deutlich, dass die Probedivision als eigenständiger Primzahltest schnell ineffizient wird. Besonders auffällig ist der sprunghafte Anstieg der Laufzeit in der vorletzten Zeile. Dies liegt mit an Sicherheit grenzender Wahrscheinlichkeit an der Art und Weise, wie Python numerische Datentypen intern verwaltet. Zum Verständnis:

Python unterscheidet zwischen zwei Ganzzahl-Datentypen: *plain integers*, die unter Verwendung des C-Datentyps `long` implementiert sind und folglich eine begrenzte Länge haben, sowie *long integers*, die theoretisch unbegrenzt lang sein können, aber grundsätzlich weniger effizient zu handhaben sind – vgl. hierzu u. a. die Links in Fußnote 1 und Abschnitt 8.3.1 in [Ernesti, J.; Kaiser, P., 2008].

Mittels `sys.maxint`[1] lässt sich zudem leicht die größtmögliche natürliche Zahl in Erfahrung bringen, die Python als *plain integer*, also effizient verwalten kann. Auf dem zur Laufzeituntersuchung verwendeten 64-Bit-System ergab die Abfrage, dass

$$\texttt{sys.maxint} = 2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807,$$

was zu folgenden Ungleichungen führt:

$$\underbrace{1\,000\,000\,000\,000\,000\,003}_{=\,\texttt{next_prime(10\^{}18)}} < 9\,223\,372\,036\,854\,775\,807 < \underbrace{10\,000\,000\,000\,000\,000\,051}_{=\,\texttt{next_prime(10\^{}19)}}$$

Es zeigt sich also, dass die in der vorletzten Zeile von Tabelle 3.1 überprüfte Primzahl von Python nicht mehr als *plain integer* verwaltet werden kann, sondern als *long integer* verwaltet werden muss – was die Laufzeit, deutlich sichtbar, negativ beeinflusst.

3.2. Sieb des Eratosthenes

Einen anderen Weg, um Primzahlen zu erkennen, siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 7], bietet das als *Sieb des Eratosthenes* bezeichnete antike Verfahren, mit dem sich alle Primzahlen unterhalb einer frei wählbaren Schranke ermitteln lassen.

Es eignet sich daher vor allem zum Erstellen halbwegs umfangreicher Primzahltabellen, z. B. zur Verwendung in anderen Algorithmen oder zur Analyse arithmetischer Progressionen von Primzahlen, vgl. hierzu u. a. [Weisstein, E. W., 2014b], aber auch zum Ermitteln der exakten Werte der in Abschnitt 2.3 eingeführten Primzahlzählfunktion.

[1] Siehe hierzu u. a.: https://docs.python.org/2/library/sys.html#sys.maxint und https://docs.python.org/2/library/stdtypes.html#typesnumeric (aufgerufen am 07.04.2014).

Nachfolgend wird das Siebverfahren, gem. der Beschreibung in [Scheid, H.; Frommer, A., 2013, S. 8], zunächst theoretisch dargelegt und anschließend beispielhaft demonstriert:

Sieb des Eratosthenes:

1. Man schreibe alle natürlichen Zahlen von 2 bis N auf.
2. Man markiere die Zahl 2 und streiche dann jede zweite Zahl.
3. Ist m die erste nicht-gestrichene und nicht-markierte Zahl, so markiere man m und streiche dann jede m-te Zahl aus $\{m+1, m+2, \ldots, N\}$.
4. Man führe Schritt 3 für alle m mit $m^2 \leq N$ aus. Ist $m^2 > N$, so stoppe man.
5. Alle markierten bzw. nicht-gestrichenen Zahlen sind Primzahlen. Und zwar sind dies alle Primzahlen, die nicht größer als N sind.

Beispiel. Mit Hilfe des Siebs des Eratosthenes soll der exakte Wert von $\pi(15)$ ermittelt werden. Gem. Schritt 1 werden alle natürlichen Zahlen von 2 bis 15 aufgeschrieben:

$$2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15$$

Anschließend wird gem. Schritt 2 die Zahl 2 markiert und jede zweite Zahl gestrichen:

$$\underline{2} \quad 3 \quad \cancel{4} \quad 5 \quad \cancel{6} \quad 7 \quad \cancel{8} \quad 9 \quad \cancel{10} \quad 11 \quad \cancel{12} \quad 13 \quad \cancel{14} \quad 15$$

Die erste nicht-gestrichene und nicht-markierte Zahl lautet 3 und folglich wird gem. Schritt 3 ebendiese markiert und jede dritte Zahl gestrichen:

$$\underline{2} \quad \underline{3} \quad \cancel{4} \quad 5 \quad \cancel{6} \quad 7 \quad \cancel{8} \quad \cancel{9} \quad \cancel{10} \quad 11 \quad \cancel{12} \quad 13 \quad \cancel{14} \quad \cancel{15}$$

Da $5^2 > 15$ ist, wird das Siebverfahren gem. Schritt 4 bereits beendet. Die Primzahlen, die nicht größer als 15 sind, lauten somit: 2, 3, 5, 7, 11 und 13. Demnach ist $\pi(15) = 6$.

Die in Schritt 4 geforderte Abbruchbedingung ähnelt nicht zufällig der in der Probedivision, siehe Abschnitt 3.1, eingebauten. Es handelt sich erneut um eine Anwendung von Korollar 3.1.2. Sobald im Siebverfahren eine Primzahl p erreicht wird, deren Quadrat größer ist als die Schranke N, ist N entweder prim oder teilbar durch einen Primteiler kleiner p, dessen Vielfache – somit auch N selbst – jedoch bereits gestrichen worden sein müssen. Da diese Argumentation auch auf alle Zahlen, die kleiner als N sind, anwendbar ist, denn offensichtlich ist $p^2 > N > N - 1 > \ldots$, kann das Sieben beendet werden.

Eine weitere Detailverbesserung, die in Algorithmus 2 auf Seite 12 verwendet wird, besteht in einer Weiterentwicklung von Schritt 3. Statt mit dem Streichen jeder m-ten Zahl bei $m+1$ zu beginnen, reicht es, wenn dieser Vorgang bei m^2 beginnt, da die Vielfachen $k \cdot m$ von m, für die $1 < k < m$ gilt, bereits als Vielfache von k gestrichen worden sind.

3.2.1. Pseudocode

Algorithmus 2: Sieb des Eratosthenes

Input: Eine natürliche Zahl $N > 1$

```
1  P ← [ ]                                  ▷ Leere Liste
2  A ← [N + 1]                              ▷ Liste mit N + 1 Einträgen
3  m ← 2
4  while m ≤ N do
5      A[m] ← True                          ▷ Liste A initialisieren
6      m ← m + 1
7  end while
8  m ← 2
9  while m² ≤ N do
10     if A[m] = True then
11         P ← P ∪ m                        ▷ Erste nicht-gestrichene Zahl ist prim
12         j ← m²
13         while j ≤ N do
14             A[j] ← False                 ▷ Vielfache von m streichen
15             j ← j + m
16         end while
17     end if
18     m ← m + 1
19 end while
20 while m ≤ N do
21     if A[m] = True then
22         P ← P ∪ m                        ▷ Verbliebene Primzahlen an P anhängen
23     end if
24     m ← m + 1
25 end while
26 return P
```

Obiger Pseudocode basiert auf dem unter [Wikipedia, 2014d] angegebenen Pseudocode, der i. W. das in Abschnitt 3.2 beschriebene Siebverfahren wiedergibt bzw. geringfügig weiterentwickelt. Bspw. wird die Liste `A` nicht mit den natürlichen Zahlen 2 bis N befüllt, stattdessen dienen die Indizes der Listenelemente als Repräsentanten ebendieser Zahlen.

Für eine Implementierung siehe Listing C.2 auf Seite 61.

3.2.2. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt. Der Speicherverbrauch wurde mit Hilfe des Kommandozeilen-Programms *htop* kurz vor der `return`-Operation, siehe Algorithmus 2 von Seite 12, abgelesen und entspricht dem dort angegebenen *RES*-Wert, vgl. hierzu u. a. das Manual von *htop*. Die Werte, die in Tabelle 3.2 unterstrichen dargestellt sind, entsprechen den originären Angaben von *htop*. Bis ca. 100 MB wurden diese in KB, anschließend nur noch in MB angegeben.

Schranke (N)	**Zeit in s**	**Speicherverbrauch in KB**	**und in MB**
$100\,000 = 10^5$	0,084	<u>6 476</u>	6
$1\,000\,000 = 10^6$	0,430	<u>15 556</u>	15
$10\,000\,000 = 10^7$	4,354	104 448	<u>102</u>
$100\,000\,000 = 10^8$	45,995	969 728	<u>947</u>

Tabelle 3.2.: Laufzeituntersuchung des Siebs des Eratosthenes (Listing C.2).

Tabelle 3.2 zeigt deutlich, dass der zeitliche Aufwand des Siebverfahrens kontinuierlich steigt. Da dieser Schritt jedoch nur einmalig durchgeführt werden muss und anschließend das Abfragen, ob es sich bei einer gegebenen Zahl um eine Primzahl handelt, deutlich schneller vonstattengeht, ist dieser Einmalaufwand durchaus akzeptabel.

Als wesentlich problematischer erweist sich der zum Ablegen der Listen benötigte Speicherplatz, der schon bei einer vergleichsweise kleinen Schranke von 10^9 den auf dem zur Laufzeituntersuchung eingerichteten System zur Verfügung stehenden Arbeitsspeicher in Höhe von ca. 2 GB deutlich überschritt.

Zur Erinnerung: Die kleinste bei der Laufzeituntersuchung der Probedivision, siehe Abschnitt 3.1.2, untersuchte Primzahl lautete $10^{15} + 37$. Eine solche Schranke wäre mehr als undenkbar, weswegen das Sieb des Eratosthenes als eigenständiger Primzahltest als praktisch ungeeignet angesehen werden kann.

Beim Erstellen halbwegs umfangreicher Primzahltabellen erweist sich das Siebverfahren wiederum als durchaus nützlich, da es alle Primzahlen unterhalb einer Schranke wesentlich schneller findet als die Probedivision. In einem direkten Vergleich zwischen Probedivision, siehe Listing C.1 auf Seite 60, und Sieb des Eratosthenes, siehe Listing C.2 auf Seite 61, zum Auffinden aller Primzahlen, die kleiner als 10^6 sind, terminierte das Sieb des Eratosthenes etwa zehnmal schneller als die Probedivision.

Solche Primzahltabellen können, wie in Abschnitt 3.2 erwähnt, z. B. zur Analyse arithmetischer Progressionen von Primzahlen, zur Verwendung in anderen Algorithmen oder zum Ermitteln der exakten Werte der Primzahlzählfunktion verwendet werden.

4. Fermatscher Primzahltest

Die in Kapitel 3 eingeführten Primzahltests, die z. T. schon in der Antike bekannt waren, haben gezeigt, dass Primzahltests keineswegs eine Erfindung des Computerzeitalters sind und zudem mit relativ geringem Programmieraufwand realisiert werden können. Es hat sich jedoch auch gezeigt, dass weder die Probedivision, siehe Abschnitt 3.1, noch das Sieb des Eratosthenes, siehe Abschnitt 3.2, dazu geeignet ist, Primzahlen zu erkennen bzw. zu finden, die groß genug sind, um sie bspw. in kryptographischen Verfahren einzusetzen.

Nebenbei bemerkt: Unter *groß genug* versteht man – unter der Annahme, dass ein RSA-Schlüssel mit 2048 Bits derzeit noch als sicher angesehen werden kann, vgl. hierzu u. a. [BSI, 2014, S. 28] – Primzahlen in einer Größenordnung von 2^{1024}, was ca. 300 Dezimalstellen entspricht. Die Laufzeituntersuchungen haben mehr als deutlich gezeigt, dass eine solche Größenordnung mit den beschriebenen Verfahren nicht zu erreichen ist.

Um zu einem Primzahltest zu gelangen, der auch mit Zahlen mit Hunderten Dezimalstellen umgehen kann, braucht es daher einen gänzlich anderen Ansatz – einen der weder zu viel Rechenzeit noch zu viel Speicherplatz benötigt, um zu einer Entscheidung zu gelangen. Einen ebensolchen bieten Primzahltests auf der Grundlage von Kongruenzen.

In den folgenden Abschnitten sollen mit Mitteln der Gruppentheorie der Satz von Euler und der Kleine Satz von Fermat bewiesen werden, um Letzteren zu einem Primzahltest weiterzuentwickeln, der mit fast beliebig großen Zahlen als Eingabewert umgehen kann.

4.1. Satz von Euler

Die im Folgenden eingeführten Lemmata und Theoreme sowie deren Beweise basieren auf der Argumentationskette des zweiten unter [ProofWiki, 2013a] dargelegten Beweises.

Eine alternative Argumentationskette ist bspw. in [Scheid, H.; Frommer, A., 2013, S. 125–132] zu finden – oder aber im ersten bzw. dritten Beweis von [ProofWiki, 2013a].

Lemma 4.1.1. *In einer endlichen Gruppe teilt die Ordnung eines Elements der Gruppe,* Elementordnung *genannt, die Ordnung seiner Gruppe,* Gruppenordnung *genannt.*

Beweis. Siehe hierzu u. a. [ProofWiki, 2013b]. □

In den nachfolgenden Ausführungen wird der Einfachheit halber nur noch von einer *Gruppe* statt einer *endlichen Gruppe* gesprochen.

Lemma 4.1.2. *Sei $|G|$ die Ordnung einer Gruppe G, so gilt $\forall g \in G$, dass $g^{|G|} = e$, wobei e das neutrale Element, auch* Identität *genannt, von G ist.*

Beweis. Sei $|g|$ die Ordnung eines Elements von G, so gilt gem. Lemma 4.1.1, dass $|g|$ die Ordnung von G teilt, woraus folgt, dass $\exists k \in \mathbb{Z}$, mit $|G| = k \cdot |g|$ und schließlich ist

$$g^{|G|} = g^{k \cdot |g|} = (g^{|g|})^k = (e)^k = e,$$

was zu beweisen war. □

Dass die in Lemma 4.1.2 durchgeführten Umformungen erlaubt und korrekt sind, soll hier und im Weiteren als korrekt angenommen und nicht rigoros bewiesen werden.

Theorem 4.1.1. *Seien a und m teilerfremde ganze Zahlen, also $ggT(a, m) = 1$, $m > 1$ und $\varphi(m)$ die Eulersche Phi-Funktion, so gilt:*

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

Beweis. Sei $[a]_m = \{x \mid x \equiv a \pmod{m}\}$ die Restklasse all jener ganzen Zahlen x, die bei Division durch m den gleichen Rest wie a aufweisen. Aus $ggT(a, m) = 1$ folgt, dass $[a]_m$ Element der primen Restklassengruppe $\mathbb{Z}_m^*$ ist, der all jene Restklassen angehören, deren Elemente, auch *Repräsentanten* genannt, teilerfremd zu m sind.

Gem. der Definition der Eulerschen Phi-Funktion ist $|\mathbb{Z}_m^*| = \varphi(m)$. Gem. Lemma 4.1.2 gilt ferner, dass die Restklasse $[a]_m$ potenziert mit der Ordnung der primen Restklassengruppe $\mathbb{Z}_m^*$, der sie angehört, dem neutralen Element von $\mathbb{Z}_m^*$ entspricht, sodass

$$[a]_m^{\varphi(m)} = [a^{\varphi(m)}]_m = [1]_m \Leftrightarrow a^{\varphi(m)} \equiv 1 \pmod{m},$$

was zu beweisen war. □

Dass die Umformung $[a]_m^{\varphi(m)} = [a^{\varphi(m)}]_m$ in Theorem 4.1.1 erlaubt ist, folgt aus der Definition der Multiplikation von Restklassen, wonach $[a]_m \cdot [a]_m = [a \cdot a]_m$, vgl. hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 120].

Obiges Theorem wird häufig als *Satz von Euler* bezeichnet und soll zum Verständnis beispielhaft demonstriert werden:

Beispiel. Sei $\mathbb{Z}_5^* = \mathbb{Z}_5 \backslash \{[0]\} = \{[1], [2], [3], [4]\}$ die prime Restklassengruppe mod 5.

Da 5 eine Primzahl ist, gilt augenscheinlich, dass $ggT(1, 5) = ggT(2, 5) = ggT(3, 5) = ggT(4, 5) = 1$. Da $ggT(0, 5) = 5$, ist $[0]$ kein Element von $\mathbb{Z}_5^*$. Die Ordnung von $\mathbb{Z}_5^*$, also die Anzahl der Elemente von $\mathbb{Z}_5^*$, ist gem. der Definition der Eulerschen Phi-Funktion gleich $\varphi(5) = 4$. Es lässt sich nun leicht nachvollziehen, dass das, was Theorem 4.1.1 behauptet, stimmt:

$$1^4 \equiv 2^4 \equiv 3^4 \equiv 4^4 \equiv 1 \pmod{5}$$

4.2. Kleiner Satz von Fermat

Dank der in Abschnitt 4.1 gewonnenen Erkenntnisse handelt es sich beim nun folgenden Kleinen Satz von Fermat i. W. um ein Korollar von Theorem 4.1.1, der – um seine Wichtigkeit zu unterstreichen – jedoch nicht als solches bezeichnet wird.

Lemma 4.2.1. *Sei p eine Primzahl und $a \in \mathbb{N}$, so gilt, dass $p \nmid a \Rightarrow ggT(a, p) = 1$.*

Beweis. Sei $ggT(a, p) = d$. Da d per definitionem sowohl a als auch p teilt, aber p prim ist, folgt, dass $d = 1$ oder $d = p$ sein muss. Wenn aber $d = p$ wäre, würde auch $p \mid a$ gelten, was zuvor ausgeschlossen wurde. Folglich bleibt nur noch $d = 1$ übrig. □

Obiges Lemma lässt sich dergestalt interpretieren, dass eine Primzahl p teilerfremd zu allen Zahlen ist, ausgenommen der 0, sich selbst und den eigenen Vielfachen. Es gilt also ferner, dass eine natürliche Zahl x, mit $0 < x < p$, immer teilerfremd zu p sein muss.

Theorem 4.2.1. *Sei p eine Primzahl und $a \in \mathbb{N}$, mit $p \nmid a$, so gilt:*

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

Beweis. Gem. Lemma 4.2.1 sind p und a wegen $p \nmid a$ teilerfremd, also $ggT(a, p) = 1$, was gem. dem Satz von Euler, siehe Theorem 4.1.1, bedeutet, dass

$$a^{\varphi(p)} \equiv 1 \pmod{p}.$$

Da p prim ist und somit alle natürlichen Zahlen x, für die $0 < x < p$ gilt, teilerfremd zu p sind, gilt folglich, dass $\varphi(p) = p - 1$ und

$$a^{\varphi(p)} \equiv a^{p-1} \equiv 1 \pmod{p},$$

was zu beweisen war. □

Korollar 4.2.1. *Sei p eine Primzahl und $a \in \mathbb{N}$, mit $0 < a < p$, so gilt:*

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

Bei obigem Theorem handelt es sich um den *Kleinen Satz von Fermat*. Alternativ kann auch folgendes Korollar als ebendieser Satz bezeichnet werden:

Korollar 4.2.2. *Sei p eine Primzahl und $a \in \mathbb{N}$, so gilt, dass $a^p \equiv a \pmod{p}$.*

Die zuvor getroffene Einschränkung, dass es sich bei der Basis a um eine natürliche statt einer ganzen Zahl handeln muss, ist lediglich deshalb getroffen worden, weil der noch folgende Fermatsche Primzahltest, der auf Korollar 4.2.1 beruht, die Basis so wählt, dass $0 < 2 \leq a \leq N - 2 < N$ und somit die Teilerfremdheit von a und N gewährleistet ist – jedenfalls dann, wenn es sich bei der Eingabe N um eine Primzahl handelt.

4.2.1. Pseudocode

Algorithmus 3: Fermatscher Primzahltest

Input: Eine ungerade natürliche Zahl $N > 3$, eine natürliche Zahl a, mit $2 \leq a \leq N-2$

```
1 b ← a^(N−1) mod N
2 if b = 1 then
3 |  return „N is probably prime“
4 end if
5 return „N is definitely composite“
```

Obiger Pseudocode basiert auf Algorithmus 3.4.3 aus [Crandall, R.; Pomerance, C., 2005, S. 132]. Für eine Implementierung siehe Listing C.3 auf Seite 61.

Bei der Wahl von Basis a wurden zwei Einschränkungen getroffen, die auf folgenden Überlegungen beruhen – vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 132]:

Proposition 4.2.1. *Sei* $N \in \mathbb{N}\backslash\{0,1\}$, *so gilt stets* $1^{N-1} \equiv 1 \pmod{N}$.

Proposition 4.2.2. *Sei* $N \in \mathbb{N}\backslash\{0,1\}$ *ungerade, so gilt stets* $(N-1)^{N-1} \equiv 1 \pmod{N}$.

Proposition 4.2.1 sagt aus, dass die Wahl von 1 als Basis keine Aussage über $N > 1$ wird treffen können und daher nicht von Interesse ist. Viel spannender ist jedoch die Einschränkung, die Proposition 4.2.2 bzgl. der Basis a trifft: Erlaubt man als Eingabe nur ungerades $N > 1$, so führt die Wahl von $N-1$ als Basis zu keiner Aussage über N.

Dies erklärt sich wie folgt: Ist N ungerade, so ist $N-1 = 2 \cdot m$, $m \in \mathbb{N}$, gerade. Das heißt aber auch, dass die Basis geradzahlig oft vorkommt, also ist

$$(N-1)^{2 \cdot m} \equiv ((N-1) \bmod N)^{2 \cdot m} \equiv (-1)^{2 \cdot m} \equiv ((-1)^2)^m \equiv (1)^m \equiv 1 \pmod{N}.$$

Dass das voraussetzt, gerade Zahlen als Eingabe auszuschließen, stellt keine wirkliche Einschränkung dar, da es keinen geschickten Primzahltest braucht, um zu erkennen, ob eine Zahl $N > 2$ gerade und somit zusammengesetzt ist: Gerade Zahlen enden auf 0, 2, 4, 6 oder 8 und ihr LSB (engl. *least significant bit*) ist niemals 1, sondern stets 0.[1]

Ein letztes Detail soll nicht unerwähnt bleiben: In manchen Beschreibungen des Fermatschen Primzahltests, bspw. der unter [Wikipedia, 2013a], ist zu lesen, dass man, bevor man $a^{N-1} \bmod N$ berechnet, überprüfen sollte, ob $ggT(a,N) = 1$. Ist nämlich $ggT(a,N) > 1$, für $1 < a < N$ und $N > 2$, kann der Algorithmus beendet werden, da N definitiv zusammengesetzt ist, da es einen nichttrivialen Teiler besitzt. Wenngleich diese Abbruchbedingung absolut korrekt ist, ist sie doch auch vollkommen unnötig.

Dies erklärt sich wie folgt – vgl. hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 125]:

[1] Siehe hierzu u. a.: http://rosettacode.org/wiki/Even_or_odd (aufgerufen am 16.04.2014).

Theorem 4.2.2. *Seien a und N von Null verschiedene ganze Zahlen sowie $N > 1$, so gilt, dass $[a]_N$ bzgl. der Multiplikation invertierbar ist $\Leftrightarrow ggT(a, N) = 1$.*

Beweis. (Fall: $\Rightarrow$) Aus der Invertierbarkeit von $[a]_N$ bzgl. der Multiplikation folgt, dass $\exists x \in \mathbb{Z}$, sodass $a \cdot x \equiv 1 \pmod{N}$. Sei nun $ggT(a, N) = d$ und folglich $d \mid a$ sowie $d \mid N$. Da $N \mid a \cdot x - 1$, gilt also auch $d \mid a \cdot x - 1$. Wegen $d \mid a$ gilt aber auch $d \mid a \cdot x$. Die einzige natürliche Zahl jedoch, die eine Zahl und ihren direkten Vorgänger teilt, ist 1 und schlussfolgernd gilt $d = 1$ bzw. $ggT(a, N) = 1$.

(Fall: $\Leftarrow$) Aus $ggT(a, N) = 1$ folgt, dass es ganze Zahlen u, v gibt, sodass $a \cdot u + N \cdot v = 1$. Somit gilt aber auch, dass $a \cdot u + N \cdot v \equiv a \cdot u \equiv 1 \pmod{N}$. Folglich ist u das multiplikativ Inverse von a mod N. □

Der Beweis von Theorem 4.2.2 setzt Kenntnisse über grundlegende Teilbarkeitseigenschaften voraus, die bisher und im Folgenden nicht explizit eingeführt wurden bzw. werden, vgl hierzu. u. a. [Scheid, H.; Frommer, A., 2013, S. 5–6]. Die Schreibweise des größten gemeinsamen Teilers zweier ganzer Zahlen als Linearkombination, *Vielfachsummendarstellung* genannt, wird u. a. in [Scheid, H.; Frommer, A., 2013, S. 27–32] behandelt.

Beispiel. Da 2 und 5 augenscheinlich teilerfremd sind, also $ggT(2, 5) = 1$, ist $[2]_5$ gem. Theorem 4.2.2 bzgl. der Multiplikation invertierbar. Die Kongruenz

$$2 \cdot x \equiv 1 \pmod{5}$$

ist folglich eindeutig lösbar, wobei $x \in \mathbb{Z}$ das multiplikativ Inverse von 2 mod 5 ist. Konkret:

$$2 \cdot 3 \equiv 6 \equiv 1 \pmod{5}$$

Wie bereits erwähnt, gilt dies auch umgekehrt: Die Lösbarkeit der Kongruenz sagt aus, dass 2 und 5 teilerfremd sein müssen. Hieraus ergeben sich noch folgende Korollare:

Korollar 4.2.3. *Sei N eine natürliche Zahl größer 2 und a eine natürliche Zahl, für die $1 < a < N$ gilt, so folgt aus $a^{N-1} \equiv 1 \pmod{N}$ stets, dass $ggT(a, N) = 1$.*

Korollar 4.2.4. *Sei N eine natürliche Zahl größer 2 und a eine natürliche Zahl, für die $1 < a < N$ gilt, so folgt aus $ggT(a, N) > 1$ stets, dass $a^{N-1} \not\equiv 1 \pmod{N}$.*

Die Erklärung für obige Korollare ist relativ einfach und basiert auf einer simplen Umformung, die dank $N > 2$ bzw. $N - 2 > 0$ durchgeführt werden kann:

$$a^{N-1} \equiv a \cdot a^{N-2} \equiv 1 \pmod{N}$$

Obige Kongruenz impliziert, dass $[a]_N$ bzgl. der Multiplikation invertierbar ist, wobei a^{N-2} das multiplikativ Inverse von a mod N ist. Und genau diese Berechnung ist es, die im Fermatschen Primzahltest, siehe Algorithmus 3 von Seite 17, durchgeführt wird.

Anders ausgedrückt: Wann immer die Variable `b` im Algorithmus den Wert 1 annimmt, bedeutet dies, dass $ggT(a, N) = 1$. Ferner wird sie den Wert 1 nicht annehmen, wenn

$ggT(a, N) > 1$. Zusammenfassend zeigt sich also, dass die vorherige Überprüfung, ob Basis und zu überprüfende Zahl teilerfremd sind, nicht notwendig ist.

Der einzige Vorteil besteht vermutlich darin, dass – im Falle von $ggT(a, N) > 1$ – auf diese Weise ein nichttrivialer Teiler von N gefunden wird. Für die Entscheidung, ob N prim oder zusammengesetzt ist, braucht es diese Zusatzinformation allerdings nicht.

4.2.2. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt.

Primzahl (N)	**gefunden mit**[1]	$2^{N-1} \bmod N$	**Zeit in s**
$10^{100} + 267$	`next_prime(10^100)`	1	0,034
$10^{200} + 357$	`next_prime(10^200)`	1	0,044
$10^{400} + 69$	`next_prime(10^400)`	1	0,054
$10^{800} + 1537$	`next_prime(10^800)`	1	0,094

Tabelle 4.1.: Laufzeituntersuchung des Fermatschen Primzahltests (Listing C.3).

Tabelle 4.1 macht deutlich, wie überaus schnell der Fermatsche Primzahltest entscheiden kann, ob eine Zahl *vermutlich* prim oder *definitiv* zusammengesetzt ist. Er wäre damit also prinzipiell dazu geeignet, Primzahlen zu finden, die gem. [BSI, 2014] als groß genug gelten, um in kryptographischen Verfahren weiterverwendet zu werden.

Zur Erinnerung: Mit Hilfe der Probedivision konnte die Zahl $10^{20} + 39$ binnen 30 Minuten als Primzahl erkannt werden. Der Fermatsche Primzahltest beurteilt eine Zahl mit Hunderten Dezimalstellen in weniger als einer Zehntelsekunde. Entscheidend für eine solche Laufzeit ist allerdings, wie die modulare Exponentiation in Algorithmus 3 von Seite 17 implementiert wird:

Ein naiver Aufruf wie `(2**(N-1))%N` führte schnell zu einem Voll- bzw. Überlaufen des gesamten zur Verfügung stehenden Speichers – auf dem zur Laufzeituntersuchung eingerichteten System immerhin etwa 4 GB. Ein Aufruf von `pow(2, N-1, N)` brachte schließlich den gewünschten Erfolg. Für weitere Details siehe Abschnitt 7.2.1.

4.3. Fermatsche Pseudoprimzahlen

Die Zeit, die der Fermatsche Primzahltest benötigt, um eine Entscheidung bzgl. der Primalität einer Zahl zu treffen, ist beeindruckend, siehe Abschnitt 4.2.2. Es gibt jedoch einen entscheidenden Nachteil, auf den bislang noch gar nicht eingegangen wurde, der aber erklärt, warum im Erfolgsfall „nur“ von einer *vermutlichen Primzahl* die Rede ist:

[1] Bzgl. der Sage-Methode `next_prime()` siehe die Laufzeituntersuchung in Abschnitt 3.1.2.

Die Umkehrung des Kleinen Satzes von Fermat gilt nicht.

Anders ausgedrückt: Für jede Primzahl p und jede dazu teilerfremde Basis a ist die Kongruenz $a^{p-1} \equiv 1 \pmod{p}$ erfüllt. Das bedeutet aber nicht, dass p eine Primzahl ist, nur weil die Kongruenz erfüllt ist.

Benutzt man bspw. den Fermatschen Primzahltest, um alle Primzahlen, die nicht größer als 1000 sind, zu ermitteln, vgl. auch Abschnitt 2.3, lautet die falsche Antwort: 171. Tatsächlich ist aber $\pi(1000) = 168$. Folgender Python-Code listet die drei Zahlen auf, die fälschlicherweise als Primzahlen mitgezählt wurden:

```
1  #!/usr/bin/python2.7
2
3  from fermat_primality_test import *
4  from sieve_of_eratosthenes import *
5
6  P = [2, 3]
7  for odd in xrange(5, 1000, 2):
8      if fermat_primality_test(odd, 2) == str(odd) + " is probably prime":
9          P.append(odd)
10
11 print [p for p in P if p not in sieve_of_eratosthenes(1000)]
```

Das Ergebnis: Der Fermatsche Primzahltest kann die zusammengesetzten Zahlen

$$\begin{aligned} 341 &= 11 \cdot 31, \\ 561 &= 3 \cdot 11 \cdot 17, \\ 645 &= 3 \cdot 5 \cdot 43 \end{aligned}$$

nicht als solche erkennen – jedenfalls nicht zur Basis 2. Eine solche Zahl nennt man *Fermatsche Pseudoprimzahl zur Basis 2* oder kurz *psp(2)* (engl. *pseudoprime to base 2*). Allgemein gilt:

Definition 4.3.1. Eine natürliche Zahl N ist eine Fermatsche Pseudoprimzahl zur Basis a, kurz psp(a), wenn die Kongruenz

$$a^{N-1} \equiv 1 \pmod{N}$$

erfüllt ist, obwohl N zusammengesetzt ist. Sich also N bzgl. a wie eine Primzahl verhält.

Wie bereits in Abschnitt 4.2.1 erwähnt, ist jede zusammengesetzte natürliche Zahl N gem. Proposition 4.2.1 eine psp(1). Gem Proposition 4.2.2 ist ferner noch jede zusammengesetzte ungerade natürliche Zahl N eine psp($N-1$).

Um dem Problem der Fermatschen Pseudoprimzahlen zur Basis 2 zu begegnen, könnte man geneigt sein, eine andere Basis zu wählen, um zu entscheiden, ob es sich bei einer eingegebenen Zahl um eine Primzahl handelt. Folgender Python-Code, der als Basis 3

statt 2 wählt, lässt erahnen, dass dieser Ansatz nicht funktionieren wird:

```
1  #!/usr/bin/python2.7
2
3  from fermat_primality_test import *
4  from sieve_of_eratosthenes import *
5
6  P = [2, 3]
7  for odd in xrange(5, 1000, 2):
8      if fermat_primality_test(odd, 3) == str(odd) + " is probably prime":
9          P.append(odd)
10
11 print [p for p in P if p not in sieve_of_eratosthenes(1000)]
```

Das Ergebnis: Der Fermatsche Primzahltest kann die zusammengesetzten Zahlen

$$
\begin{aligned}
91 &= 7 \cdot 13, & 703 &= 19 \cdot 37,\\
121 &= 11 \cdot 11, & 949 &= 13 \cdot 73\\
671 &= 11 \cdot 61,
\end{aligned}
$$

zur Basis 3 nicht als solche erkennen. Die Zahl $286 = 2 \cdot 11 \cdot 13$ ist ebenfalls eine psp(3), wurde aber von obigem Code nicht mitgezählt, da sie gerade ist. Dennoch: Die Wahl einer anderen Basis hat das Problem nur zu anderen Zahlen verschoben, nicht aber gelöst.

Es lässt sich sogar zeigen, dass keine Basis fehlerfrei verwendet werden kann:

Theorem 4.3.1. *Für jede natürliche Zahl $a \geq 2$ gilt: Es gibt unendlich viele psp(a).*

Beweis. Siehe hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 133]. □

Demgegenüber gilt aber auch, dass Fermatsche Pseudoprimzahlen selten sind. 1981 verbesserte Pomerance die in [Erdős, P., 1956] angegebene obere Schranke für die Anzahl Fermatscher Pseudoprimzahlen zur Basis 2, die nicht größer als $x \in \mathbb{N}$, und bewies:

Theorem 4.3.2. *Sei $P\pi(x)$ die Anzahl Fermatscher Pseudoprimzahlen zur Basis 2, die nicht größer als $x \in \mathbb{N}$ sind, so gilt für alle großen x, dass*

$$P\pi(x) \leq \frac{x}{\sqrt{L(x)}},$$

mit

$$L(x) = e^{\ln x \ln\ln\ln x / \ln\ln x}.$$

Beweis. Siehe hierzu u. a. [Pomerance, C., 1981]. □

Im Beweis zu Theorem 4.3.2 wird darauf hingewiesen, dass das Ergebnis hinsichtlich beliebiger Basen $a \geq 2$ verallgemeinert werden kann. Die Schreibweise $P\pi(x)$, die nicht

zufällig an die der Primzahlzählfunktion $\pi(x)$ erinnert, ist aus [Ribenboim, P., 2011, S. 226–227] entnommen. Verallgemeinert bezeichnet

$$P\pi_a(x) = \#\{1 \leq N \leq x \mid N \textit{ ist eine psp}(a)\}$$

die Anzahl Fermatscher Pseudoprimzahlen zur Basis a, die nicht größer als x sind. Folglich ist $P\pi(x) = P\pi_2(x)$.

In [Crandall, R.; Pomerance, C., 2005, S. 132] heißt es ferner noch, dass die Anzahl Fermatscher Pseudoprimzahlen zu einer Basis $a \geq 2$ gleich $o(\pi(x))$ ist. Dass also der Grenzwert

$$\lim_{x \to \infty} \frac{P\pi_a(x)}{\pi(x)} = 0$$

existiert und Fermatsche Pseudoprimzahlen folglich weitaus seltener als Primzahlen sind.

4.3.1. Anzahl und Häufigkeit

Mit Hilfe einer Tabelle lassen sich die Resultate anschaulich numerisch untermauern:

x	$\pi(x)$	$P\pi(x)$	$P\pi(x)/\pi(x)$
10^3	168	3	0,018
10^4	1 229	22	0,018
10^5	9 592	78	0,008
10^6	78 498	245	0,003
10^7	664 579	750	0,001
10^8	5 761 455	2 057	$<$ 0,001
10^9	50 847 534	5 597	$<$ 0,001
$10 \cdot 10^9$	455 052 511	14 884	$<$ 0,001
$25 \cdot 10^9$	1 091 987 405	21 853	$<$ 0,001

Tabelle 4.2.: Werte von $\pi(x)$ und $P\pi(x)$ im Vergleich.

Da 341 die kleinste $psp(2)$ ist, fängt obige Tabelle erst bei $x = 10^3$ an. Die Werte der $P\pi(x)$-Spalte sind [Pomerance, C.; Selfridge, J. L.; Wagstaff, Jr., S. S., 1980] entnommen.

Alle anderen Spalten wurden mit Hilfe von Sage, siehe Abschnitt 7.1, berechnet. Vor allem die Spalte, die den Quotienten aus $P\pi(x)$ und Primzahlzählfunktion angibt, macht deutlich, wie überaus selten Fermatsche Pseudoprimzahlen zur Basis 2 sind. Für größere Werte von $P\pi(x)$ siehe u. a. Tabelle 18 in [Ribenboim, P., 2011, S. 229].

4.4. Carmichael-Zahlen

Nachdem Abschnitt 4.3 deutlich gemacht hat, dass es keine Basis gibt, die sich bei der Beurteilung natürlicher Zahlen N nicht unendlich oft irren wird, siehe Theorem 4.3.1, stellt sich unweigerlich die Frage, wie man diese Fälle zumindest minimieren kann.

Der intuitivste Ansatz besteht schlicht darin, mehrere Basen zu kombinieren, den Fermatschen Primzahltests also mehrmals hintereinander auszuführen. Verletzt auch nur eine einzige der gewählten Basen a, mit $1 < a < N$, die Kongruenz

$$a^{N-1} \equiv 1 \pmod{N},$$

so ist N definitiv zusammengesetzt.

Beispiel. $341 = 11 \cdot 31$ ist zwar eine Fermatsche Pseudoprimzahl zur Basis 2, da

$$2^{340} \equiv 1 \pmod{341}.$$

Sie ist aber keine Fermatsche Pseudoprimzahl zur Basis 3, da

$$3^{340} \not\equiv 1 \pmod{341}.$$

Kombiniert man die Basen 2 und 3, um zu beurteilen, ob N prim ist, wird sich der Fermatsche Primzahltest bei den ersten 1000 Zahlen nicht mehr irren. Bei $1105 = 5 \cdot 13 \cdot 17$ jedoch schon, denn 1105 ist zwar augenscheinlich zusammengesetzt, dennoch gilt

$$2^{1104} \equiv 3^{1104} \equiv 1 \pmod{1105},$$

womit 1105 die kleinste natürliche Zahl ist, die sowohl zur Basis 2 als auch 3 lügt. Unter den ersten 10 000 Zahlen gibt es noch sechs weitere derartige Zahlen.

Die Zahl 1105 ist jedoch aus einem anderen Grund noch viel interessanter: Es ist die zweitkleinste Zahl, die bzgl. *aller* zu ihr teilerfremden Basen lügt – die kleinste ist 561.

Definition 4.4.1. Wenn eine zusammengesetzte natürliche Zahl N die Kongruenz

$$a^{N-1} \equiv 1 \pmod{N}$$

für alle zu ihr teilerfremden Basen a erfüllt, nennt man N eine *Carmichael-Zahl*.

Gem. Korollar 4.2.4 muss eine Carmichael-Zahl von den nicht-teilerfremden Basen entlarvt werden, da $ggT(a, N) > 1$ impliziert, dass $a^{N-1} \not\equiv 1 \pmod{N}$, siehe Abschnitt 4.2.1.

Um die Auswirkungen der Carmichael-Zahlen auf den Fermatschen Primzahltest beurteilen zu können, müssen folgende Fragen geklärt werden: Wie viele Carmichael-Zahlen gibt es? Und wie häufig treten diese auf?

Die erste Frage konnte 1994 von Alford, Granville und Pomerance beantwortet werden:

Theorem 4.4.1. *Es gibt unendlich viele Carmichael-Zahlen.*

Beweis. Siehe hierzu u. a. [Alford, W. R.; Granville, A.; Pomerance, C., 1994b]. □

4.4.1. Anzahl und Häufigkeit

Um ein Gefühl dafür zu bekommen, wie häufig Carmichael-Zahlen auftreten, soll Tabelle 4.2 angepasst werden und statt dem Quotienten aus $P\pi(x)$ und Primzahlzählfunktion die Anzahl der Carmichael-Zahlen, die nicht größer als $x \in \mathbb{N}$ sind, kurz $C(x)$, angeben:

x	$\pi(x)$	$P\pi(x)$	$C(x)$
10^3	168	3	1
10^4	1 229	22	7
10^5	9 592	78	16
10^6	78 498	245	43
10^7	664 579	750	105
10^8	5 761 455	2 057	255
10^9	50 847 534	5 597	646
$10 \cdot 10^9$	455 052 511	14 884	1 547
$25 \cdot 10^9$	1 091 987 405	21 853	2 163

Tabelle 4.3.: Werte von $\pi(x)$, $P\pi(x)$ und $C(x)$ im Vergleich.

Die Schreibweise $C(x)$ und die Werte der gleichnamigen Spalte sind [Pomerance, C.; Selfridge, J. L.; Wagstaff, Jr., S. S., 1980] entnommen. Für größere Werte von $P\pi(x)$ und $C(x)$ siehe u. a. Tabelle 18 und 19 in [Ribenboim, P., 2011, S. 229–230]. Letztere entspricht i. W. den Ergebnissen, die in [Pinch, R. G. E., 2007] zu finden sind.

Über Carmichael-Zahlen lässt sich dank Tabelle 4.3 sagen, dass diese zwar selten sind, aber nicht so selten, wie man es sich für einen nahezu fehlerfreien Primzahltest wünschen würde. Im Beweis zur Unendlichkeit der Carmichael-Zahlen, siehe [Alford, W. R.; Granville, A.; Pomerance, C., 1994b], wird gezeigt, dass für genügend großes x

$$C(x) > x^{2/7}$$

gilt. In [Crandall, R.; Pomerance, C., 2005, S. 134] wird ergänzt, dass die 96. Carmichael-Zahl – 8 719 309 – *genügend groß* zu sein scheint. Außerdem wird darauf hingewiesen, dass Erdős vermutete, dass für jedes $\varepsilon > 0$ ein $x_0(\varepsilon)$ existiert, sodass für alle $x \geq x_0(\varepsilon)$

$$C(x) > x^{1-\varepsilon}$$

gilt. Es drängt sich nun unweigerlich die Frage auf, wie gut die Chancen stehen, dass der Fermatsche Primzahltest eine Carmichael-Zahl als zusammengesetzt erkennt, wenn der seltene Fall eingetreten ist, dass eine solche als Primzahl in Betracht gezogen wird.

Wie bereits in Abschnitt 4.4 dargelegt, werden Carmichael-Zahlen ausschließlich von nicht-teilerfremden Basen entlarvt. Da zu einer Carmichael-Zahl N genau $\varphi(N)$ teilerfremde Basen existieren, die nicht größer als N sind, gibt es folglich $(N-1) - \varphi(N)$

nicht-teilerfremde Basen a, mit $2 \leq a \leq N-2$. Laxer formuliert: Je größer $\varphi(N)$ im Vergleich zu N ist, desto schlechter stehen die Chancen. Für eine grafische Darstellung des sprunghaften Wachstums der Eulerschen Phi-Funktion siehe Abb. A.1 auf Seite 58.

Beispiel. $561 = 3 \cdot 11 \cdot 17$ wird von $\varphi(561) = 320$ Basen fälschlicherweise als Primzahl akzeptiert und von $560 - 320 = 240$ Basen als zusammengesetzt erkannt. Was immerhin etwa jeder zweiten Basis entspricht. Demgegenüber wird jedoch $252\,601 = 41 \cdot 61 \cdot 101$ von lediglich $252\,600 - \varphi(252\,601) = 252\,600 - 240\,000 = 12\,600$ Basen abgelehnt.

Obiges Beispiel macht deutlich, dass die Größe der Primfaktoren ausschlaggebend für die Anzahl nicht-teilerfremder Basen ist, denn

$$(N-1) - \varphi(N) \approx \frac{N}{p_1} + \frac{N}{p_2} + \frac{N}{p_3},$$

mit $N = p_1 \cdot p_2 \cdot p_3$. Anders ausgedrückt: Jedes Vielfache eines Primfaktors p_i ist eine nicht-teilerfremde Basis und folglich resultieren aus möglichst kleinen Primfaktoren besonders viele Basen, die N als zusammengesetzt erkennen.

In [Alford, W. R.; Granville, A.; Pomerance, C., 1994a] wurde jedoch gezeigt, dass es unendlich viele Carmichael-Zahlen gibt, die keine „kleinen“ Primfaktoren enthalten. Um diesem Problem zu begegnen, müsste man den Fermatschen Primzahltest so anpassen, dass er für praktische Zwecke irrelevant wird.

Bspw. könnte man ihn deterministisch gestalten und nacheinander alle Basen $a \geq 2$ überprüfen, bis sichergestellt ist, dass N garantiert prim bzw. zusammengesetzt ist. Ein solches Vorgehen würde den Fermatschen Primzahltest jedoch zu einer Art – besonders rechenintensiven – Probedivision, siehe Abschnitt 3.1, verkümmern lassen.

4.4.2. Korselts Kriterium

Eine einfache Möglichkeit, um zu erkennen, ob eine Zahl N eine Carmichael-Zahl ist, besteht immer dann, wenn man die vollständige Primfaktorzerlegung von N kennt.

Theorem 4.4.2. *Eine ganze Zahl N ist eine Carmichael-Zahl gdw. N positiv, zusammengesetzt sowie quadratfrei ist und für jeden Primteiler p von N gilt: $p - 1 \mid N - 1$.*

Beweis. Siehe hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 134]. □

Korollar 4.4.1. *Alle Carmichael-Zahlen sind ungerade.*

Obiger Korollar erklärt sich wie folgt: Angenommen eine Carmichael-Zahl N sei gerade. Da sie quadratfrei ist, kommt jeder ihrer Primteiler genau einmal vor. Da sie zusammengesetzt ist, muss es neben der 2 noch min. einen weiteren Primteiler p geben. Dieser ist zwingend ungerade. Da N gerade ist, gilt ferner, dass $N-1$ ungerade ist. Da p ungerade ist, gilt ferner, dass $p-1$ gerade ist. Es gilt somit aber auch, dass $p - 1 \mid N - 1$ bedeutet, dass eine gerade Zahl eine ungerade teilt (Widerspruch!).

Beispiel. $561 = 3 \cdot 11 \cdot 17$ ist die kleinste Carmichael-Zahl. Sie ist augenscheinlich positiv, zusammengesetzt und quadratfrei. Es gilt ferner, dass

$$\begin{aligned} 3 - 1 &\mid 561 - 1, \text{ da } 560 = 280 \cdot 2, \\ 11 - 1 &\mid 561 - 1, \text{ da } 560 = 56 \cdot 10, \\ 17 - 1 &\mid 561 - 1, \text{ da } 560 = 35 \cdot 16. \end{aligned}$$

Korselts Kriterium stellt zwar scheinbar eine sehr direkte Möglichkeit dar, um den Fermatschen Primzahltest vor Carmichael-Zahlen zu schützen, ist aber letztlich ebenfalls unbrauchbar. Denn hat man erst einmal eine Zahl vollständig faktorisiert, braucht es keinen Primzahltest mehr, um zu entscheiden, ob sie prim oder zusammengesetzt ist.

Zusammenfassend bleibt vorerst nur zu sagen, dass Carmichael-Zahlen einer der Gründe sind, warum ein anderer – probabilistischer – Primzahltest den Fermatschen Primzahltest verdrängt hat, siehe Kapitel 8 sowie die Empfehlung in [BSI, 2014, S. 69–70].

4.5. Primzahlen finden

Im Folgenden soll mit Hilfe des Fermatschen Primzahltests beispielhaft demonstriert werden, wie eine n-Bit-Primzahl gefunden, gewissermaßen „generiert“, werden kann.

Aufgrund der Unendlichkeit der Fermatschen Pseudoprimzahlen und Carmichael-Zahlen, siehe Abschnitt 4.3 sowie Abschnitt 4.4, besteht stets die Gefahr, dass es sich nicht um eine Primzahl, sondern um eine zusammengesetzte Zahl handelt. Wie gering dieses Risiko, gerade bei großen Zahlen ist, wird u. a. in [Kim, S. H.; Pomerance, C., 1989] untersucht.

In [Alford, W. R.; Granville, A.; Pomerance, C., 1994a] wird zudem erwähnt, dass, wenn N weder prim noch eine Carmichael-Zahl ist, mehr als die Hälfte der Basen in $[1, N-1]$ dies korrekt erkennen und N ablehnen werden. Keine schlechte Ausgangssituation.

Primzahlen finden:

1. Man wähle eine zufällige ungerade Zahl N im Intervall $\left(2^{n-1}, 2^n\right)$, mit $n \geq 3$.
2. Man wähle eine zufällige Basis a, mit $2 \leq a \leq N-2$, und teste mit Hilfe von Algorithmus 3 von Seite 17, ob N zusammengesetzt oder vermutlich prim ist.
3. Man führe Schritt 2 bis zu T-mal aus, mit $T \geq 1$. Wird N als zusammengesetzt entlarvt, beginne man wieder bei Schritt 1. Andernfalls akzeptiere man N.

Obige Beschreibung basiert i. W. auf Algorithmus 3.5.7 aus [Crandall, R.; Pomerance, C., 2005, S. 138] – dieser nutzt in Schritt 2 jedoch nicht den Fermatschen Primzahltest.

Beispiel. Im Folgenden soll eine 4-Bit-Primzahl gefunden werden, also gilt $n = 4$. Die Anzahl der Iterationen wird auf 3 festgelegt, also gilt $T = 3$. Gem. Schritt 1 wird zuerst eine zufällige ungerade Zahl N im Intervall $(8, 16)$ gewählt, etwa $N = 9$. Anschließend

wird gem. Schritt 2 eine zufällige Basis a, mit $2 \leq a \leq 7$, gewählt, etwa $a = 2$. Es stellt sich heraus, dass 9 keine Primzahl ist, da

$$2^8 \not\equiv 1 \pmod 9$$

und folglich wird gem. Schritt 3 wieder bei Schritt 1 begonnen. Dieses Mal wird die ungerade Zahl $N = 11$ gewählt. Gem. Schritt 2 und 3 werden nacheinander 3 zufällige Basen a gewählt, für die $2 \leq a \leq 9$ gilt, etwa 2, 5 und 6. Es stellt sich heraus, dass

$$2^{10} \equiv 5^{10} \equiv 6^{10} \equiv 1 \pmod{11}$$

und folglich wird $11 = (1011)_2$ gem. Schritt 3 als 4-Bit-Primzahl akzeptiert.

4.5.1. Pseudocode

Algorithmus 4: Primzahlen finden

Input: Eine natürliche Zahl $n \geq 3$, eine natürliche Zahl $T \geq 1$

```
1  while True do
2  |  Wähle ein zufälliges N im Intervall (2^{n-1}, 2^n)
3  |  if 2 ∤ N then
4  |  |  i ← 1
5  |  |  while i ≤ T do
6  |  |  |  Wähle eine zufällige Basis a, mit 2 ≤ a ≤ N − 2
7  |  |  |  if N ist gem. Algorithmus 3 von Seite 17 zusammengesetzt then
8  |  |  |  |  break
9  |  |  |  end if
10 |  |  |  i ← i + 1
11 |  |  end while
12 |  |  if i > T then
13 |  |  |  return N                    ▷ N nach T Iterationen akzeptieren
14 |  |  end if
15 |  end if
16 end while
```

Obiger Pseudocode basiert auf der in Abschnitt 4.5 angegebenen Beschreibung zum Finden von Primzahlen, die i. W. Algorithmus 3.5.7 aus [Crandall, R.; Pomerance, C., 2005, S. 138] entspricht. Für eine Implementierung siehe Listing C.4 auf Seite 62.

Wegen der äußeren while-Schleife von Algorithmus 4 stellt sich die Frage, was passiert, wenn im Intervall $\left(2^{n-1}, 2^n\right)$ keine einzige Primzahl existiert bzw. warum dieser Fall nicht berücksichtigt wird. Denn offensichtlich scheint das Programm dann in einer Endlosschleife zu verharren. Tatsächlich ist es aber so, dass dieser Fall niemals eintreten wird,

denn laut Bertrands Postulat, siehe hierzu u. a. [Ribenboim, P., 2011, S. 188], existiert zwischen $N > 1$ und $2 \cdot N$ min. eine Primzahl p, für die $N < p < 2 \cdot N$ gilt.

Zwischen $N = 2^{n-1}$ und $2 \cdot N = 2 \cdot 2^{n-1} = 2^n$ existiert für $n \geq 3$ also stets eine Primzahl.

Was bei Algorithmus 4 durchaus passieren kann, ist, dass in der inneren while-Schleife eine Basis mehr als einmal gewählt wird, was einem kleineren T-Wert gleichzusetzen ist und somit zu einer höheren Unsicherheit führt. Zudem kann es in der äußeren while-Schleife vorkommen, dass Zahlen, die bereits als zusammengesetzt erkannt wurden, erneut gewählt und dann vielleicht nicht entlarvt werden. Beide Risiken sind bei großen n-Werten verschwindend gering und werden daher zugunsten der Rechenzeit, des Speicherplatzbedarfs sowie der Lesbarkeit des Pseudocodes als vernachlässigbar angesehen.

4.5.2. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt.

Bits (n)	**Dezimalstellen**	**Iterationen (T)**	**Zeit in s**
512	155	100	0,332
1024	309	100	2,039
2048	617	100	5,363

Tabelle 4.4.: Laufzeituntersuchung von Listing C.4 (Primzahlen finden).

Die zu obigen Zeitangaben gehörenden Zahlen werden aus Platzgründen nicht abgedruckt. Dass es sich um Primzahlen handelte, wurde mit Hilfe von Sage verifiziert. Konkret wurde getestet, ob `next_prime(N - 1) == N` gilt[1]. Die Ausgabe lautete stets `True`.

Es soll zudem nicht unerwähnt bleiben, dass die Laufzeit von Listing C.4 stark schwankt. Bspw. terminierte das Programm für $n = 1024$ mehrmals in weniger als 1 Sekunde, brauchte aber auch oftmals mehr als 6 Sekunden. Dies liegt u. a. an der eher vorsichtigen Wahl des Parameters T, der vergleichsweise groß gewählt wurde, vgl. hierzu u. a. die Bemerkung in [BSI, 2014, S. 69–70].

[1] Bzgl. der Sage-Methode `next_prime()` siehe die Laufzeituntersuchung in Abschnitt 3.1.2.

5. Lucas-Primzahltest

Der in Kapitel 4 eingeführte Fermatsche Primzahltest und der darauf aufbauende Algorithmus zum Finden von Primzahlen, siehe Abschnitt 4.5, haben gezeigt, dass wenige Zeilen Code ausreichen, um beliebig lange Primzahlen zu „generieren“. Selbst eine vergleichsweise große 2048-Bit-Primzahl konnte in wenigen Sekunden gefunden werden.

Das Problem gegenüber den naiven Primzahltests, die in Kapitel 3 vorgestellt wurden, besteht jedoch darin, dass es sich nicht mit Sicherheit um Primzahlen handelt, sondern nur mit hoher Wahrscheinlichkeit.

Um dies zu verdeutlichen, sprechen Crandall und Pomerance in [Crandall, R.; Pomerance, C., 2005, S. 132] nicht vom Fermatschen Primzahltest, sondern von einem *probable prime test*. Die Zahlen, die sich durch (wiederholte) Ausführung eines derartigen Tests finden lassen, seien zudem weniger Primzahlen als vielmehr *industrial-grade primes*, siehe hierzu [Crandall, R.; Pomerance, C., 2005, S. 138]. Für praktische Zwecke vollkommen ausreichend, aber eben nicht garantiert prim.

Um diesem Defizit Rechnung zu tragen, werden in den folgenden Abschnitten verschiedene Varianten des Lucas-Primzahltests präsentiert, die – als Umkehrung des Kleinen Satzes von Fermat – allesamt dazu geeignet sind, zu *beweisen*, dass eine Zahl prim ist.

Deren Jahreszahlen und Bezeichnungen sind [Wikipedia, 2013b] entnommen – vgl. auch [Ribenboim, P., 2011, S. 40] und [Crandall, R.; Pomerance, C., 2005, S. 173–174].

5.1. Primitivwurzeln und primitive Restklassen

Um den Lucas-Primzahltest verstehen zu können, werden in diesem Abschnitt weitere wichtige Begriffe und Resultate der Zahlentheorie eingeführt und beispielhaft erklärt.

Definition 5.1.1. Sei N eine natürliche Zahl größer 1, dann gilt: Eine ganze Zahl a heißt genau dann *Primitivwurzel* mod N, wenn die Elementordnung von a gleich der Gruppenordnung der primen Restklassengruppe $\mathbb{Z}_N^*$, $\varphi(N)$, ist. In diesem Fall wird

$$\mathbb{Z}_N^* = \{[a]^1, [a]^2, \ldots, [a]^{\varphi(N)}\},$$

mit $[a]^{\varphi(N)} = [1]$, von den Potenzen der *primitiven Restklasse* $[a]$ erzeugt, vgl. hierzu u. a. [Wikipedia, 2014c] und [Scheid, H.; Frommer, A., 2013, S. 132]. Eine solche Gruppe, die von den Potenzen eines einzigen Elements erzeugt wird, nennt man *zyklisch*, vgl. hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 136].

Beispiel. Aus $2^4 \equiv 1 \pmod 5$ und

$$2^1 \not\equiv 1 \pmod 5,$$
$$2^2 \not\equiv 1 \pmod 5,$$
$$2^3 \not\equiv 1 \pmod 5$$

folgt, dass die Elementordnung von 2 gleich $\varphi(5) = 4$ ist. Es handelt sich folglich um eine Primitivwurzel mod 5, sodass die prime Restklassengruppe

$$\mathbb{Z}_5^* = \{[2]^1, [2]^2, [2]^3, [2]^{\varphi(5)}\} = \{[2], [4], [3], [1]\}$$

von der primitiven Restklasse [2] erzeugt wird. Demzufolge ist $\mathbb{Z}_5^*$ eine zyklische Gruppe.

Theorem 5.1.1. *Eine primitive Restklasse mod $N > 1$ existiert gdw. $N \in \{2, 4, p^k, 2p^k\}$, wobei p eine ungerade Primzahl und k eine natürliche Zahl größer 0 ist.*

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 134–135]. □

Zur Verdeutlichung: Die Begriffe *primitive Restklasse* und *Primitivwurzel* werden i. W. synonym verwendet, da eine primitive Restklasse aus Primitivwurzeln besteht. Die Existenz einer primitiven Restklasse impliziert folglich die Existenz von Primitivwurzeln.

Lemma 5.1.1. *Sei $[a]$, $a \in \mathbb{Z}$, eine prime Restklasse mod $N > 1$, $N \in \mathbb{N}$, dann gilt:*

$$ord_N([a]^h) = k \Leftrightarrow ggT(h, k) = 1,$$

wobei h eine natürliche Zahl größer 0 und $k = ord_N([a])$ die Ordnung von $[a]$ mod N ist.

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 131]. □

Theorem 5.1.2. *Sei $[a]$, $a \in \mathbb{Z}$, eine primitive Restklasse mod $N > 1$, $N \in \mathbb{N}$, dann ist*

$$\{[a]^i \mid 1 \leq i \leq \varphi(N) \wedge ggT(i, \varphi(N)) = 1\}$$

die Menge aller primitiven Restklassen mod N.

Beweis. Da die prime Restklassengruppe

$$\mathbb{Z}_N^* = \{[a]^1, [a]^2, \ldots, [a]^{\varphi(N)}\}$$

von den Potenzen der primitiven Restklasse $[a]$ erzeugt wird, haben alle primen Restklassen die Form $[a]^i$, mit $1 \leq i \leq \varphi(N)$. $[a]^i$ ist genau dann primitiv, wenn $ord_N([a]^i) = ord_N([a]) = \varphi(N)$, was gem. Lemma 5.1.1 genau dann gilt, wenn $ggT(i, \varphi(N)) = 1$. Da jede primitive auch eine prime Restklasse ist, sind dies alle primitiven Restklassen. □

Aus Theorem 5.1.1 und Theorem 5.1.2 folgt, dass, falls N eine Primzahl ist, $\varphi(\varphi(N))$ primitive Restklassen mod N existieren, vgl. hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 136]. In [Crandall, R.; Pomerance, C., 2005, S. 174] fügen Crandall und Pomerance ergänzend hinzu, dass für $N > 200\,560\,490\,131$ mehr als

$$N/(2 \cdot \ln \ln N)$$

Primitivwurzeln in $\{1, 2, \ldots, N-1\}$ existieren. Da der Ausdruck im Nenner vergleichsweise langsam wächst, existieren bei großem N derart viele Primitivwurzeln, dass man, um eine zu finden, einfach ein zufälliges a aus $\{1, 2, \ldots, N-1\}$ wählen kann. Nach etwa $2 \cdot \ln \ln N$ Versuchen sollte man auf eine Primitivwurzel mod N gestoßen sein.

Beispiel. Um zu einer 1024-Bit-Primzahl mit ca. 300 Dezimalstellen eine Primitivwurzel zu finden, wird man voraussichtlich etwa

$$\lfloor 2 \cdot \ln \ln 10^{300} \rfloor = 13$$

Versuche benötigen. Eine durchaus überschaubare Anzahl.

Diese Strategie lässt sich sogar noch optimieren, wenn man sich die folgende Frage stellt: Wenn N eine Primzahl ist, wie klein ist die kleinste positive Primitivwurzel mod N?

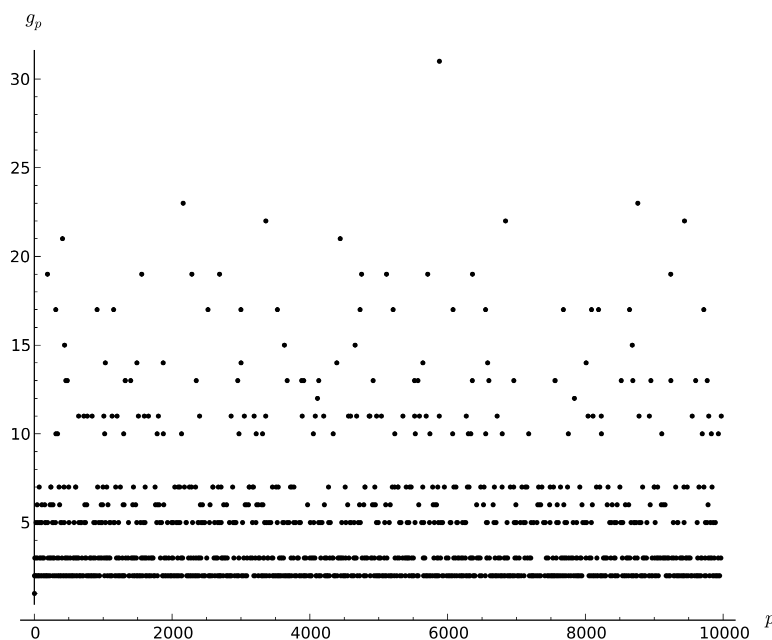


Abbildung 5.1.: Die kleinsten positiven Primitivwurzeln g_p mod der Primzahlen $p < 10^4$.

Obige Abbildung, mit der deutlich sichtbaren Häufung im unteren Bereich, legt den Schluss nahe, dass Primitivwurzeln mod einer Primzahl p tendenziell klein sind. Von

den Primzahlen, die nicht größer als 10 000 sind, 1229 an der Zahl, besitzen immerhin 470 die Zahl 2, 292 die Zahl 3 und 171 die Zahl 5 als kleinste positive Primitivwurzel g_p.

Zusammengenommen machen diese drei Zahlen bereits mehr als 75 % aller Primitivwurzeln in Abb. 5.1 von Seite 31 aus. Ob sie Primitivwurzeln mod unendlich vieler Primzahlen sind, ist laut [Scheid, H.; Frommer, A., 2013, S. 138] noch immer unklar.

Klar hingegen ist, dass g_p nicht allzu schnell wächst: Burgess konnte 1962 beweisen, dass für jedes $\varepsilon > 0$ eine Konstante $C > 0$ existiert, die lediglich von ε abhängt, sodass

$$g_p \leq C \cdot p^{1/4+\varepsilon}$$

für genügend großes p gilt, siehe hierzu u. a. [Burgess, D. A., 1962] und vgl. auch [Ribenboim, P., 2011, S. 19–20]. Knapp 20 Jahre später bewies Grosswald, dass für $p > e^{e^{24}}$

$$g_p < p^{0{,}499}$$

gilt, siehe hierzu u. a. [Grosswald, E., 1981]. Bezogen auf die Suche nach Primitivwurzeln: Bevor man zufällige Werte aus $\{1, 2, \ldots, N-1\}$ als Primitivwurzeln in Betracht zieht, sollte man sein Glück zuvor mit einigen besonders kleinen Werten versucht haben.

5.2. Variante von 1876: Vorläufer des Lucas-Primzahltests

Im Jahre 1876 gelang Lucas folgende Umkehrung des Kleinen Satzes von Fermat – vgl. hierzu u. a. [Wikipedia, 2013b] und [Ribenboim, P., 2011, S. 40]:

Theorem 5.2.1. *Eine natürliche Zahl $N > 2$ ist genau dann eine Primzahl, wenn es eine natürliche Zahl a, mit $1 < a < N$, gibt, für die sowohl*

$$a^{N-1} \equiv 1 \pmod{N}$$

als auch

$$a^m \not\equiv 1 \pmod{N}$$

für alle natürlichen Zahlen $0 < m < N-1$ gilt.

Beweis. Siehe den Beweis des verbesserten Lucas-Primzahltests in Abschnitt 5.4. □

Zur Erinnerung: Die erste Bedingung in Theorem 5.2.1 ist die des Fermatschen Primzahltests aus Kapitel 4. Wird sie verletzt, ist N definitiv zusammengesetzt. Andernfalls ist N entweder eine Primzahl oder eine Fermatsche Pseudoprimzahl zur Basis a.

Fügt man noch die zweite Bedingung hinzu und wird sie nicht verletzt, dann bedeutet dies erst einmal nur, dass a die Ordnung $N-1$ hat. Daraus folgt jedoch unweigerlich, dass $\varphi(N) = N-1$ ist, dass a eine Primitivwurzel mod N ist und letztlich auch, dass N eine Primzahl ist. Warum die Existenz eines Elements der Ordnung $N-1$ all diese Schlussfolgerungen nach sich zieht, wird ausführlich in Abschnitt 5.4 thematisiert.

5.2.1. Pseudocode

Algorithmus 5: Vorläufer des Lucas-Primzahltests

Input: Eine natürliche Zahl $N > 2$, eine natürliche Zahl a, mit $1 < a < N$

```
1  b ← a^(N−1) mod N
2  if b = 1 then
3  |  m ← 1
4  |  while m < N − 1 do
5  |  |  b ← a^m mod N
6  |  |  if b = 1 then
7  |  |  |  return               ▷ Die Ordnung von a mod N ist kleiner als N - 1
8  |  |  end if
9  |  |  m ← m + 1
10 |  end while
11 |  return „N is definitely prime“
12 end if
13 return „N is definitely composite“
```

Obiger Pseudocode basiert auf Theorem 5.2.1, das i. W. dem unter [Wikipedia, 2013b] angegebenen *Vorläufer des Lucas-Tests* entspricht.

Für eine Implementierung siehe Listing C.5 auf Seite 62.

5.2.2. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt.

Primzahl (N)	**gefunden mit**[1]	**Primitivwurzel (a)**	**Zeit in s**
100 003	`next_prime(10^5)`	2	2,168
1 000 003	`next_prime(10^6)`	2	12,570
10 000 019	`next_prime(10^7)`	6	121,908

Tabelle 5.1.: Laufzeituntersuchung des Vorläufers des Lucas-Primzahltests (Listing C.5).

Die Primitivwurzeln, die Listing C.5 auf Seite 62 als Parameter a übergeben wurden, sind vorab mit Hilfe der Methode `primitive_root()`[2] bestimmt worden. So war einerseits

[1] Bzgl. der Sage-Methode `next_prime()` siehe die Laufzeituntersuchung in Abschnitt 3.1.2.
[2] Siehe hierzu u. a.: http://www.sagemath.org/doc/reference/misc/sage/rings/arith.html#sage.rings.arith.primitive_root (aufgerufen am 06.05.2014).

sichergestellt, dass das Programm die Zahl N als Primzahl erkennen wird, andererseits war dies mit der größtmöglichen Anzahl an Schleifendurchläufen verbunden, da

$$a^m \not\equiv 1 \pmod{N}$$

für alle natürlichen Zahlen $0 < m < N - 1$ überprüft werden musste. Und genau diese Vielzahl an Überprüfungen ist es, die Algorithmus 5 praktisch unbrauchbar werden lässt.

Man muss zwar fairerweise dazusagen, dass man bei der Implementierung auf den Einsatz von Sage hätte verzichten können. Dies wäre jedoch zulasten der Vergleichbarkeit mit den anderen Varianten des Lucas-Primzahltests gegangen. Zudem ist der Zugewinn an Geschwindigkeit irrelevant: Am Ergebnis ändert er letztlich nichts, da sich die Grenze, ab der das Programm praktisch unbrauchbar wird, nur unwesentlich verschiebt.

5.3. Variante von 1891: Lucas-Primzahltest

Im Jahre 1891 verbesserte Lucas Theorem 5.2.1, was zu dem nach ihm benannten Primzahltest führte – vgl. hierzu u. a. [Wikipedia, 2013b] und [Ribenboim, P., 2011, S. 40]:

Theorem 5.3.1. *Eine natürliche Zahl $N > 2$ ist genau dann eine Primzahl, wenn es eine natürliche Zahl a, mit $1 < a < N$, gibt, für die sowohl*

$$a^{N-1} \equiv 1 \pmod{N}$$

als auch

$$a^m \not\equiv 1 \pmod{N}$$

für alle echten Teiler $m \in \mathbb{N}$ von $N - 1$.

Beweis. Siehe den Beweis des verbesserten Lucas-Primzahltests in Abschnitt 5.4. □

Obiges Theorem unterscheidet sich vom Vorläufer des Lucas-Primzahltests dadurch, dass die zweite Bedingung für weit weniger m geprüft werden muss. Und dennoch ist sichergestellt, dass die Elementordnung von a gleich $N - 1$ ist. Dies erklärt sich wie folgt:

Theorem 5.3.2. *Sei $[a]$, $a \in \mathbb{Z}$, eine prime Restklasse mod $N > 1$, $N \in \mathbb{N}$, dann gilt:*

$$[a]^h = [1] \Leftrightarrow k \mid h,$$

wobei h eine natürliche Zahl größer 0 und $k = ord_N([a])$ die Ordnung von $[a]$ mod N ist.

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 131]. □

Zur Erinnerung: Gem. Korollar 4.2.3, siehe Abschnitt 4.2.1, folgt aus

$$a^{N-1} \equiv 1 \pmod{N} \Leftrightarrow [a]^{N-1} = [1],$$

mit a und N gem. Theorem 5.3.1, dass $ggT(a, N) = 1$. Also ist $[a]$ eine prime Restklasse. Gem. Theorem 5.3.2 folgt daraus, dass $N - 1$ von der Elementordnung von a geteilt wird

bzw. dass $N-1$ ein Vielfaches der Elementordnung von a ist. Demnach ist es ausreichend, alle *echten* Teiler von $N-1$ zu überprüfen, um zu zeigen, dass $ord_N([a]) = N-1$.

Zum Verständnis: Der triviale Teiler 1 muss nicht überprüft werden, da für eine Basis a, mit $1 < a < N$ und $N > 2$, stets $[a]^1 \neq [1]$ gilt. Den trivialen Teiler $N-1$ darf man nicht (erneut) überprüfen, da sonst keine Primzahl als solche erkannt werden könnte, denn es wird augenscheinlich nicht gleichzeitig sowohl $[a]^{N-1} = [1]$ als auch $[a]^{N-1} \neq [1]$ gelten.

Beispiel. Mit $a = 3$ soll überprüft werden, ob $N = 7$ eine Primzahl ist. Aus

$$3^6 \equiv 1 \pmod 7$$

folgt, dass $ord_7([3]) \mid 6$. Da 2 und 3 die einzigen echten Teiler von 6 sind, folgt aus

$$3^2 \not\equiv 1 \pmod 7,$$
$$3^3 \not\equiv 1 \pmod 7,$$

dass $ord_7([3]) = 6 = N-1$ und gem. Theorem 5.3.1, dass 7 eine Primzahl ist.

5.3.1. Pseudocode

Algorithmus 6: Lucas-Primzahltest

Input: Eine natürliche Zahl $N > 2$, eine natürliche Zahl a, mit $1 < a < N$

```
1  b ← a^(N−1) mod N
2  if b = 1 then
3      foreach m ∈ [Echte Teiler von N − 1] do
4          b ← a^m mod N
5          if b = 1 then
6              return                ▷ Die Ordnung von a mod N ist kleiner als N - 1
7          end if
8      end foreach
9      return „N is definitely prime“
10 end if
11 return „N is definitely composite“
```

Obiger Pseudocode basiert auf Theorem 5.3.1, das i. W. dem unter [Wikipedia, 2013b] angegebenen *Lucas-Test* entspricht. Zeile 3 wurde mit Hilfe der Sage-Methode `divisors()`[1] realisiert. Für eine Implementierung siehe Listing C.6 auf Seite 63.

[1] Siehe hierzu u. a.: http://www.sagemath.org/doc/reference/misc/sage/rings/arith.html#sage.rings.arith.divisors (aufgerufen am 06.05.2014).

5.3.2. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt.

Primzahl (N)	gefunden mit[1]	Primitivwurzel (a)	Zeit in s
$10^{50} + 151$	`next_prime(10^50)`	11	1,165
$10^{60} + 7$	`next_prime(10^60)`	5	1,384
$10^{70} + 33$	`next_prime(10^70)`	23	1,773
$10^{80} + 129$	`next_prime(10^80)`	–	–
$10^{90} + 289$	`next_prime(10^90)`	3	17,740
$10^{100} + 267$	`next_prime(10^100)`	–	–
$10^{110} + 7$	`next_prime(10^110)`	5	1,237
$10^{120} + 79$	`next_prime(10^120)`	–	–
$10^{130} + 1113$	`next_prime(10^130)`	–	–
$10^{140} + 13$	`next_prime(10^140)`	2	1,477
$10^{150} + 67$	`next_prime(10^150)`	–	–

Tabelle 5.2.: Laufzeituntersuchung des Lucas-Primzahltests (Listing C.6).

Obige Tabelle stellt die bislang ausführlichste, aber auch lückenhafteste Laufzeituntersuchung dar – und das ist Absicht: Versteht man Tabelle 5.2, hat man auch die Vor- und Nachteile des Lucas-Primzahltests verstanden. Doch eins nach dem anderen.

Ein Vorteil ist unübersehbar: Der Lucas-Primzahltest hat Zahlen mit weit über hundert Dezimalstellen als Primzahlen identifiziert. Damit ist er dem eingangs vorgestellten Vorläufer des Lucas-Primzahltests, siehe Abschnitt 5.2, signifikant überlegen.

Die Primitivwurzeln wurden zwar wieder mit Hilfe Methode `primitive_root()`[2] in Erfahrung gebracht, doch auch ohne diese Vorarbeit wäre die Laufzeit nur unwesentlich schlechter ausgefallen. Bspw. hätte man zur Identifizierung der Primzahl $10^{50} + 151$ ca. 11 Sekunden gebraucht, wenn man alle Zahlen $2 \leq a \leq 11$ durchgegangen wäre.

Zur Identifizierung der Primzahl $10^{90} + 289$ hätte man ca. 35 Sekunden gebraucht, wenn man zuvor noch die Zahl 2 als Primitivwurzel getestet hätte. Das ist zwar immer noch akzeptabel, aber deutlich langsamer als die anderen in Tabelle 5.2 gemessenen Werte.

Der Grund für diese Auffälligkeit – und auch die Lücken in Tabelle 5.2 – ist in der dritten Zeile von Algorithmus 6 von Seite 35 zu finden: Das Bestimmen der echten Teiler der

[1] Bzgl. der Sage-Methode `next_prime()` siehe die Laufzeituntersuchung in Abschnitt 3.1.2.
[2] Bzgl. der Sage-Methode `primitive_root()` siehe die Laufzeituntersuchung in Abschnitt 5.2.2.

Zahl $N-1$ ist i. A. keineswegs trivial. Es ist vielmehr so, dass das Bestimmen der Teilermenge mit der Faktorisierung der Zahl $N-1$ praktisch gleichzusetzen ist. Da für dieses Problem derzeit kein effizienter Algorithmus bekannt ist, vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005], hängt die Laufzeit des Lucas-Primzahltests nur geringfügig vom Finden einer Primitivwurzel oder der Berechnung von Kongruenzen ab.

Das wesentliche Hindernis stellt die vollständige Faktorisierung der Zahl $N-1$ dar.

Daher auch die Lücken in Tabelle 5.2: Wann immer Sage nicht innerhalb von etwa einer halben Minute die Zahl $N-1$ faktorisieren bzw. eine Primitivwurzel mod N finden konnte, wurde auf die Laufzeituntersuchung postwendend verzichtet. Das ist zwar eine durchaus strenge Zeitvorgabe, macht aber ein weiteres Problem deutlich:

Es lässt sich nicht generell vorhersagen, wie lange der Lucas-Primzahltest benötigen wird, um die echten Teiler von $N-1$ zu bestimmen. Obwohl er an der Zahl

$$(10^{80}+129)-1$$

„scheiterte", gelang die Faktorisierung der wesentlich größeren Zahl

$$(10^{140}+13)-1$$

bzw. das Bestimmen ihrer echten Teiler binnen einer Sekunde. Im Normalfall gilt jedoch: Je größer N, desto schwieriger ist die vollständige Faktorisierung von $N-1$.

Trotz der genannten – teils eklatanten – Nachteile bietet der Lucas-Primzahltest einen Vorteil, der leicht vergessen werden kann: Wann immer die Faktorisierung von $N-1$ bekannt oder einfach zu ermitteln ist, ist er überaus effizient. Doch dazu später mehr.

5.4. Variante von 1953: Verbesserter Lucas-Primzahltest

Im Jahre 1953 gelang Lehmer eine Verbesserung von Theorem 5.3.1 – vgl. hierzu u. a. [Wikipedia, 2013b] und [Crandall, R.; Pomerance, C., 2005, S. 173–174]:

Theorem 5.4.1. *Eine natürliche Zahl $N>2$ ist genau dann eine Primzahl, wenn es eine natürliche Zahl a, mit $1<a<N$, gibt, für die sowohl*

$$a^{N-1} \equiv 1 \pmod N$$

als auch

$$a^{(N-1)/q} \not\equiv 1 \pmod N$$

für alle Primteiler q von $N-1$.

Beweis. Gem. Korollar 4.2.3 folgt aus der ersten Bedingung in Theorem 5.4.1, dass a und N teilerfremd sind, also $ggT(a, N) = 1$ ist. Dies ist gleichbedeutend damit, dass $[a]$ eine prime Restklasse bzw. Element der primen Restklassengruppe $\mathbb{Z}_N^*$ ist. Aus

$$a^{N-1} \equiv 1 \pmod N \Leftrightarrow [a]^{N-1} = [1]$$

folgt gem. Theorem 5.3.2, dass die Ordnung von $[a]$ mod N ein Teiler von $N-1$ ist, also

$$ord_N([a]) \mid N-1,$$

somit $\exists k \in \mathbb{N}$, mit $k \cdot ord_N([a]) = N-1$. Nun gilt es herauszufinden, ob $k = 1$ oder $k > 1$. Angenommen $k > 1$: Dann wird k gem. Korollar 2.1.1 von min. einer Primzahl p geteilt. Diese würde aber auch $N-1$ teilen und folglich wäre

$$a^{(N-1)/p} \equiv 1 \pmod N.$$

Hat man jedoch alle Primteiler q von $N-1$ überprüft und galt jedes Mal

$$a^{(N-1)/q} \not\equiv 1 \pmod N,$$

so muss $k = 1$ und folglich $ord_N([a]) = N-1$ sein. Da – lax formuliert – die Elementordnung die Gruppenordnung teilt, gilt gem. Lemma 4.1.1, dass

$$ord_N([a]) \mid \varphi(N) \Rightarrow N-1 \mid \varphi(N) \Rightarrow N-1 \leq \varphi(N),$$

denn $[a] \in \mathbb{Z}_N^*$ und $|\mathbb{Z}_N^*| = \varphi(N)$. Sollte N zusammengesetzt sein, dann gibt es in der Menge $\{1, 2, \ldots, N\}$ neben N noch min. eine weitere Zahl größer 1, die N teilt, die also nicht-teilerfremd zu N ist und folglich ist $\varphi(N) \leq N-2$. Da dies einen Widerspruch zur obigen Ungleichung $N-1 \leq \varphi(N)$ bedeuten würde, kann N nicht zusammengesetzt sein.

N ist folglich eine Primzahl und $\varphi(N) = N-1$.

Ist andererseits $N > 2$ eine (ungerade) Primzahl, so existiert gem. Theorem 5.1.1 eine primitive Restklasse $[a]$ mod N, deren Ordnung per definitionem gleich $\varphi(N) = N-1$ ist, die also insbesondere beide Bedingungen in Theorem 5.4.1 erfüllt. □

Obiger Beweis basiert auf den Beweisen des verbesserten Lucas-Primzahltests, die in [Crandall, R.; Pomerance, C., 2005, S. 173] und unter [MathPages, 2014] zu finden sind:

Der Beweis unter [MathPages, 2014] setzt für $N > 1$ voraus, dass $\varphi(N) \leq N-1$ und $\varphi(N) = N-1 \Leftrightarrow N \in \mathbb{P}$, womit der erste Teil des obigen Beweises bereits an dem Punkt beendet wäre, an dem $N-1 \leq \varphi(N)$ gezeigt wurde. Der Beweis in [Crandall, R.; Pomerance, C., 2005, S. 173] setzt eben dies nicht voraus, übergeht aber jegliche Details, warum aus der zweiten Bedingung von Theorem 5.4.1 folgt, dass $ord_N([a]) = N-1$.

Um dies nochmals zu betonen: Die wesentliche Idee des Lucas-Primzahltests, in allen vorgestellten Varianten, ist das Finden eines Elements der Ordnung $N-1$, da daraus

folgt, dass N eine Primzahl und $\varphi(N) = N - 1$ ist. Man könnte alternativ auch sagen, dass es um das Finden einer Primitivwurzel mit der größtmöglichen Ordnung $N-1$ geht.

Der Vorläufer des Lucas-Primzahltests, siehe Abschnitt 5.2, der Lucas-Primzahltest, siehe Abschnitt 5.3, und der verbesserte Lucas-Primzahltest unterscheiden sich i. W. dadurch, wie geschickt bzw. ungeschickt sie bei dieser Suche vorgehen.

5.4.1. Pseudocode

Algorithmus 7: Verbesserter Lucas-Primzahltest

Input: Eine natürliche Zahl $N > 2$, eine natürliche Zahl a, mit $1 < a < N$

```
1  b ← a^(N-1) mod N
2  if b = 1 then
3      foreach q ∈ [Primteiler von N − 1] do
4          b ← a^((N-1)/q) mod N
5          if b = 1 then
6              return                ▷ Die Ordnung von a mod N ist kleiner als N - 1
7          end if
8      end foreach
9      return „N is definitely prime“
10 end if
11 return „N is definitely composite“
```

Obiger Pseudocode basiert auf Theorem 5.4.1, das i. W. dem unter [Wikipedia, 2013b] angegebenen *verbesserten Lucas-Test* entspricht. Zeile 3 wurde mit Hilfe der Sage-Methode `factor()`[1] realisiert. Für eine Implementierung siehe Listing C.7 auf Seite 63.

5.4.2. Laufzeituntersuchung

Auf eine separate Laufzeituntersuchung des verbesserten Lucas-Primzahltests wird an dieser Stelle verzichtet, da das Laufzeitverhalten zum größten Teil dem des Lucas-Primzahltests entspricht, siehe Abschnitt 5.3 bzw. die Laufzeituntersuchung in Abschnitt 5.3.2. Es ist zwar i. A. so, dass der verbesserte Lucas-Primzahltest weit weniger Kongruenzen überprüfen muss, aber auch er muss zuerst die Zahl $N - 1$ faktorisieren.

Viel spannender und aussagekräftiger ist die Laufzeituntersuchung des verbesserten Lucas-Primzahltests, wenn man diesen ganz ohne Sage realisiert, siehe Abschnitt 5.4.3.

[1] Siehe hierzu u. a.: http://www.sagemath.org/doc/reference/misc/sage/rings/arith.html#sage.rings.arith.factor (aufgerufen am 06.05.2014).

5.4.3. Implementierung ohne Sage

Um die Vergleichbarkeit untereinander zu gewährleisten, wurden alle drei vorgestellten Varianten des Lucas-Primzahltests in Sage bzw. mit Hilfe von Sage-Methoden implementiert. Auf diesem Wege konnte zwar gezeigt werden, dass die Varianten, welche die Faktorisierung von $N-1$ benötigen, dem Vorläufer des Lucas-Primzahltests als auch den naiven Primzahltests, also insbesondere der Probedivision, deutlich überlegen sind.

Aber gerade der Vergleich mit der Probedivision ist holprig: Auf der einen Seite steht eine möglichst einfache Python-Implementierung, die ganz ohne Hilfsmittel auskommen muss. Auf der anderen Seite ein komplexes Computeralgebrasystem, das auf moderne Algorithmen zurückgreifen kann. Es überrascht kaum, dass Letzteres schneller ist.

Um diesem Defizit Rechnung zu tragen, soll der verbesserte Lucas-Primzahltest ein weiteres Mal, ganz ohne Methoden von Sage, implementiert und anschließend mit der Probedivision aus Abschnitt 3.1 verglichen werden. Es geht also i. W. darum, die Faktorisierungs*methode* `factor()`[1] durch eine eigene, möglichst einfache Python-Implementierung zu ersetzen. Als Faktorisierungs*verfahren* soll die Probedivision verwendet werden.

Zur Erinnerung, siehe Abschnitt 3.1: Die Probedivision wird häufig nicht als Primzahltest, sondern als Faktorisierungsverfahren verwendet. Die Unterschiede sind marginal: Statt bei einem gefundenen Primfaktor zu verkünden, dass eine Zahl zusammengesetzt ist, wird dieser vermerkt und die Zahl durch den gefundenen Primfaktor geteilt. Sollte am Ende ein Rest größer 1 übrig bleiben, so wird dieser als letzter Primfaktor vermerkt.

Die Probedivision als Faktorisierungsverfahren, als Ersatz für die Methode `factor()` in der Implementierung des verbesserten Lucas-Primzahltests, bietet neben ihrer Einfachheit einen weiteren, wichtigen Vorteil: Die Vergleichbarkeit mit der Probedivision als Primzahltest ist gegeben, da beide Verfahren gleichermaßen naiv vorgehen.

5.4.3.1. Pseudocode

Nachfolgender Pseudocode basiert auf Algorithmus 3.1.1 aus [Crandall, R.; Pomerance, C., 2005, S. 118–119]. Für eine Implementierung siehe Listing C.8 auf Seite 64.

Für eine Implementierung des verbesserten Lucas-Primzahltests, unter Verwendung des nachfolgenden Algorithmus als Faktorisierungsverfahren, siehe Listing C.9 auf Seite 64.

[1] Bzgl. der Sage-Methode `factor()` siehe die Anmerkungen zum Pseudocode in Abschnitt 5.4.1.

Algorithmus 8: Probedivision als Faktorisierungsverfahren

Input: Eine natürliche Zahl $N > 1$

```
1  F ← [ ]                                  ▷ Leere Liste
2  while 2 | N do
3  │  N ← N/2
4  │  F ← F ∪ 2                             ▷ 2 als Primfaktor vermerken
5  end while
6  d ← 3
7  while d^2 ≤ N do
8  │  while d | N do
9  │  │  N ← N/d
10 │  │  F ← F ∪ d                          ▷ d als Primfaktor vermerken
11 │  end while
12 │  d ← d + 2
13 end while
14 if N ≠ 1 then
15 │  F ← F ∪ N                             ▷ N als letzten Primfaktor vermerken
16 end if
17 return F
```

5.4.3.2. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt.

Primzahl (N)	gefunden mit[1]	Primitivwurzel (a)	Zeit in s
$10^{20} + 39$	`next_prime(10^20)`	3	6,252
$10^{22} + 9$	`next_prime(10^22)`	7	0,467
$10^{24} + 7$	`next_prime(10^24)`	–	–
$10^{26} + 67$	`next_prime(10^26)`	2	0,082
$10^{28} + 331$	`next_prime(10^28)`	–	–
$10^{30} + 57$	`next_prime(10^30)`	5	159,213

Tabelle 5.3.: Laufzeituntersuchung des verbesserten Lucas-Primzahltests (Listing C.9).

[1] Bzgl. der Sage-Methode `next_prime()` siehe die Laufzeituntersuchung in Abschnitt 3.1.2.

Zum Verständnis: Die Primitivwurzeln wurden dieses Mal *nicht* mit Hilfe der Sage-Methode `primitive_root()`[1] in Erfahrung gebracht. Stattdessen wurden – gem. den Überlegungen in Abschnitt 5.1 – nacheinander alle natürlichen Zahlen $a > 1$ ausprobiert.

Bspw. musste Listing C.9 auf Seite 64 für $N = 10^{30} + 57$ ganze viermal ausgeführt werden. Erst $a = 5$ hatte die gewünschte größtmögliche Ordnung $N - 1$ und diente damit als Beweis, dass N eine Primzahl ist. Jeder einzelne Durchgang dauerte ca. 40 Sekunden. Der Wert in der letzten Spalte entspricht also stets der insgesamt verstrichenen Zeit.

Offensichtlich kann man diesen Wert bei Bedarf schnell und unkompliziert reduzieren: Statt die Faktorisierung als zeitintensivsten Schritt immer wieder aufs Neue durchzuführen, könnte man Listing C.9 auf Seite 64 so umschreiben, dass a als Schwellwert interpretiert wird, bis zu dem nach Primitivwurzeln gesucht werden soll. Und dennoch:

Trotz dieser recht naiven Implementierung und mehrmaligen Faktorisierung von $N - 1$ scheint der verbesserte Lucas-Primzahltest der Probedivision als Primzahltest überlegen zu sein. Für die kleinste Primzahl, $10^{20} + 39$, in Tabelle 5.3 benötigte diese bereits mehr als 30 Minuten, der verbesserte Lucas-Primzahltest nur etwas mehr als sechs Sekunden.

An zwei Zahlen in der Tabelle „scheiterte“ die Implementierung jedoch – selbst nach mehr als 5 Minuten lag noch keine vollständige Faktorisierung von $N - 1$ vor. Ein Muster, das bereits in der Laufzeituntersuchung in Abschnitt 5.3.2 zu erkennen war. Führt man die Faktorisierung mit Hilfe von Sage durch, erkennt man das Problem augenblicklich:

$$(10^{24} + 7) - 1 = 2 \cdot 7 \cdot 29 \cdot 2\,463\,054\,187\,192\,118\,226\,601$$
$$(10^{28} + 331) - 1 = 2 \cdot 5 \cdot 523 \cdot 1\,912\,045\,889\,101\,338\,432\,122\,371$$

Beide Zahlen bestehen aus einem vergleichsweise großen letzten Primfaktor, mit 22 resp. 25 Dezimalstellen. Zum Vergleich: Die Primzahl $10^{20} + 39$, für die die Probedivision als Primzahltest mehr als 30 Minuten benötigte, hatte „nur“ 21 Dezimalstellen. Da es sich bei der Probedivision als Faktorisierungsverfahren i. W. um den gleichen Algorithmus handelt, sind derart riesige Primfaktoren eine schier unüberwindbare Hürde.

Zusammenfassend zeigt sich somit erneut, dass der (verbesserte) Lucas-Primzahltest genau dann überaus effizient ist, wenn die Faktorisierung von $N - 1$ bekannt oder einfach zu ermitteln ist. Eine Eigenart, die in Kapitel 6 zu einem Primzahltest weiterentwickelt wird, der – bestimmte – Zahlen mit vielen tausend Dezimalstellen identifizieren kann.

[1] Bzgl. der Sage-Methode `primitive_root()` siehe die Laufzeituntersuchung in Abschnitt 5.2.2.

6. Pépin-Primzahltest

In Kapitel 3 wurde mit Hilfe der Probedivision bewiesen, dass $10^{20} + 39$, eine Zahl mit 21 Dezimalstellen, eine Primzahl ist. Der verbesserte Lucas-Primzahltest aus Kapitel 5, ob mit oder ohne Sage implementiert, konnte diesen „Rekord" problemlos knacken und u. a. beweisen, dass $10^{30} + 57$, eine Zahl mit 31 Dezimalstellen, eine Primzahl ist.

Das Laufzeitverhalten des Lucas-Primzahltests ist jedoch wesentlich sprunghafter als das der Probedivision: Manche Zahlen konnten im Bruchteil einer Sekunde als Primzahl identifiziert werden, bei anderen Zahlen war auch nach mehreren Minuten noch keine Antwort absehbar. Schuld ist die Herangehensweise des Lucas-Primzahltests:

Um zu entscheiden, ob N eine Primzahl ist, versucht dieser nicht etwa N, sondern stattdessen $N - 1$ zu faktorisieren. Um dies zu betonen, sprechen Crandall und Pomerance in [Crandall, R.; Pomerance, C., 2005, S. 173] auch nicht vom verbesserten Lucas-Primzahltest, sondern stattdessen vom *Lucas theorem* resp. dem *$n - 1$ test*.

Da derzeit kein effizienter Algorithmus bekannt ist, um große natürliche Zahlen in ihre Primfaktoren zu zerlegen, vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005], hängt die Laufzeit des Lucas-Primzahltests maßgeblich vom Aufbau von $N - 1$ ab. Bspw. waren „große" Primfaktoren für Listing C.9 auf Seite 64 eine schier unüberwindbare Hürde.

Doch was, wenn man den Spieß umdreht und nur bestimmte Zahlen betrachtet? Nämlich solche, bei denen die Faktorisierung von $N - 1$ kein Problem darstellt. Die Fermat-Zahlen

$$3, 5, 17, 257, 65\,537, 4\,294\,967\,297, 18\,446\,744\,073\,709\,551\,617, \ldots$$

wären ideale Kandidaten, da sie stets gleich aufgebaut sind. Die k-te Fermat-Zahl

$$F_k = 2^{2^k} + 1, \text{ für } k \geq 0,$$

und folglich besitzt $F_k - 1$ nur einen einzigen Primfaktor: die Zahl 2. In den folgenden Abschnitten soll diese Idee zum Pépin-Primzahltest weiterentwickelt werden. Einem Primzahltest, mit dem gezeigt werden konnte, dass

$$F_{24} = 2^{2^{24}} + 1,$$

eine Zahl mit unglaublichen 5 050 446 Dezimalstellen, aller Voraussicht nach *keine* Primzahl ist, vgl. hierzu u. a. [Crandall, R. E.; Mayer, E. W.; Papadopoulos, J. S., 2003].

6.1. Quadratische Reste, Nichtreste und das Legendre-Symbol

Um den Pépin-Primzahltest, genauer gesagt den Beweis des Pépin-Primzahltests verstehen zu können, werden in den folgenden Abschnitten weitere wichtige Begriffe und Resultate der Zahlentheorie eingeführt und beispielhaft erklärt.

Definition 6.1.1. Seien a und m teilerfremde ganze Zahlen, also $ggT(a, m) = 1$, und $m > 1$, so gilt: a heißt genau dann *quadratischer Rest* mod m, wenn die Kongruenz

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

für eine ganze Zahl x lösbar ist. Andernfalls nennt man a einen *quadratischen Nichtrest* mod m – vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 96].

Beispiel. Die Zahl 2 ist zwar ein quadratischer Rest mod 7, da

$$(\pm 3)^2 \equiv (\pm 4)^2 \equiv 2 \pmod{7},$$

aber ein quadratischer Nichtrest mod 5, da für alle ganzen Zahlen x gilt, dass

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

Für den Fall, dass m eine ungerade Primzahl p ist, hat sich folgende Notation etabliert:

Definition 6.1.2. Für eine ganze Zahl a und eine ungerade Primzahl p gilt:

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

Obige Kurzschreibweise, alternativ auch (a/p), wird als *Legendre-Symbol* bezeichnet.

Beispiel. Anknüpfend an das vorherige Beispiel ist $(2/7) = 1$, aber $(2/5) = -1$.

Folgende Eigenschaft des Legendre-Symbols ist bereits ohne Beweis klar ersichtlich – siehe obige Definitionen und vgl. auch [Scheid, H.; Frommer, A., 2013, S. 239]:

Proposition 6.1.1. *Sei a eine ganze Zahl und p eine ungerade Primzahl, so gilt:*

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

6.1.1. Das Quadratische Reziprozitätsgesetz und das Euler-Kriterium

Im Folgenden werden mehrere Theoreme eingeführt, mit denen sich das Legendre-Symbol effizient berechnen lässt, vgl. hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 238–246].

Theorem 6.1.1. *Seien p und q ungerade Primzahlen, mit $p \neq q$, so gilt:*

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4}$$

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 241–243]. □

Daraus folgt laut [Scheid, H.; Frommer, A., 2013, S. 241] vor allem, dass

$$\left(\frac{p}{q}\right) = -\left(\frac{q}{p}\right),$$

wenn $p \equiv q \equiv 3 \pmod 4$. Vereinfacht ausgedrückt: Wenn man „Zähler“ und „Nenner“ des Legendre-Symbols vertauscht, muss man möglicherweise auch das Vorzeichen wechseln.

Theorem 6.1.2. *Sei p eine ungerade Primzahl, so gilt:*

$$\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}$$

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 241]. □

Beispiel. Anknüpfend an das Beispiel aus Abschnitt 6.1 ist

$$\left(\frac{2}{7}\right) = (-1)^{(7^2-1)/8} = (-1)^{(49-1)/8} = (-1)^{48/8} = (-1)^6 = ((-1)^2)^3 = (1)^3 = 1$$

und folglich ist 2 ein quadratischer Rest mod 7. Demgegenüber ist

$$\left(\frac{2}{5}\right) = (-1)^{(5^2-1)/8} = (-1)^{(25-1)/8} = (-1)^{24/8} = (-1)^3 = -1$$

und folglich ist 2 ein quadratischer Nichtrest mod 5.

Bei Theorem 6.1.2 handelt es sich um den sogenannten *2. Ergänzungssatz* des Quadratischen Reziprozitätsgesetzes. Der 1. Ergänzungssatz wird im Folgenden nicht benötigt und wurde daher ausgespart – vgl. hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 243].

Theorem 6.1.3. *Sei a eine ganze Zahl und p eine ungerade Primzahl, so gilt:*

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

Beweis. Siehe hierzu u. a. [Scheid, H.; Frommer, A., 2013, S. 239–240]. □

Theorem 6.1.3 wird häufig als *Euler-Kriterium* bezeichnet. Im Beweis des Pépin-Primzahltests wird es eine wichtige Rolle spielen. Doch dazu mehr in Abschnitt 6.2.

Ein Algorithmus, der auf den genannten Theoremen aufbaut, um das Legendre-Symbol zu berechnen, ist bspw. in [Crandall, R.; Pomerance, C., 2005, S. 98] zu finden. In Sage steht u. a. die Methode `legendre_symbol()`[1] zur Verfügung.

[1] Siehe hierzu u. a.: http://www.sagemath.org/doc/reference/misc/sage/rings/arith.html#sage.rings.arith.legendre_symbol (aufgerufen am 12.05.2014).

6.1.2. Einige Eigenschaften von Fermatschen Primzahlen

Um den Beweis des Pépin-Primzahltests in Abschnitt 6.2 zu vereinfachen, wird in diesem Abschnitt Vorarbeitet geleistet, indem gezeigt wird, dass

$$\left(\frac{3}{F_k}\right) = -1,$$

wenn F_k eine Primzahl ist. Dass also 3 ein quadratischer Nichtrest mod F_k ist.

Definition 6.1.3. Sei k eine natürliche Zahl, so nennt man

$$F_k = 2^{2^k} + 1$$

eine *Fermat-Zahl*. Falls F_k eine Primzahl ist, nennt man F_k eine *Fermatsche Primzahl* – nicht zu verwechseln mit den Fermatschen *Pseudo*primzahlen aus Kapitel 4.

Derzeit sind lediglich fünf Fermatsche Primzahlen bekannt:

$$\begin{aligned} F_0 &= 2^{2^0} + 1 = 3 \\ F_1 &= 2^{2^1} + 1 = 5 \\ F_2 &= 2^{2^2} + 1 = 17 \\ F_3 &= 2^{2^3} + 1 = 257 \\ F_4 &= 2^{2^4} + 1 = 65\,537 \end{aligned}$$

Fermat vermutete einst, dass alle Fermat-Zahlen auch Primzahlen sind, vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 174], womit er sich jedoch massiv irrte: Euler fand mit F_5 das erste Gegenbeispiel, indem er zeigte, dass 641 ein Teiler von F_5 ist, vgl. hierzu u. a. [Ribenboim, P., 2011, S. 71]. Seitdem erwies sich auch jede andere Fermat-Zahl, von der entschieden werden konnte, ob sie prim oder zusammengesetzt ist, als zusammengesetzt, vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 174].

Da die Grundlagen nun hinreichend geklärt sind, kann der Beweis, dass 3 ein quadratischer Nichtrest mod einer Fermatschen Primzahl ist, endlich angegangen werden:

Lemma 6.1.1. *Sei F_k die k-te Fermat-Zahl und $k \in \mathbb{N}$ größer 0, so gilt:*

$$F_k \equiv 1 \pmod 4$$

Beweis. Da 2^k gerade ist, also $2^k = 2 \cdot m$, mit $m \in \mathbb{N}\backslash\{0\}$, gilt folglich, dass

$$(2)^{2^k} = (2)^{2 \cdot m} = (2^2)^m = (4)^m$$

und somit ist

$$F_k \equiv (2)^{2^k} + 1 \equiv (4)^m + 1 \equiv ((4) \bmod 4)^m + 1 \equiv (0)^m + 1 \equiv 1 \pmod 4,$$

was zu beweisen war. □

Lemma 6.1.2. *Sei F_k die k-te Fermat-Zahl und $k \in \mathbb{N}$ größer 0, so gilt:*

$$F_k \equiv 2 \pmod 3$$

Beweis. Da 2^k gerade ist, also $2^k = 2 \cdot m$, mit $m \in \mathbb{N}\backslash\{0\}$, gilt folglich, dass

$$(2)^{2^k} = (2)^{2\cdot m} = (2^2)^m = (4)^m = (3+1)^m$$

und somit ist

$$F_k \equiv (2)^{2^k} + 1 \equiv (3+1)^m + 1 \equiv ((3+1) \bmod 3)^m + 1 \equiv (1)^m + 1 \equiv 2 \pmod 3,$$

was zu beweisen war. □

Anmerkung: Lemma 6.1.1 und Lemma 6.1.2 gelten augenscheinlich auch für Fermatsche Primzahlen, da jede Fermatsche Primzahl auch eine Fermat-Zahl ist.

Lemma 6.1.3. *Sei F_k eine Fermatsche Primzahl und $k \in \mathbb{N}$ größer 0, so gilt:*

$$\left(\frac{3}{F_k}\right) = -1$$

Beweis. Um $(3/F_k)$ zu berechnen, werden zuerst „Zähler“ und „Nenner“ vertauscht:

$$\left(\frac{3}{F_k}\right) = \left(\frac{F_k}{3}\right)$$

Dies ist gem. dem Quadratischen Reziprozitätsgesetz, siehe Theorem 6.1.1, erlaubt, da sowohl 3 als auch $F_k \geq 5$ ungerade Primzahlen sind und $3 \neq F_k$ ist. Es kommt zu keinem Vorzeichenwechsel, da $F_k \not\equiv 3 \pmod 4$ gem. Lemma 6.1.1. Ferner ist

$$\left(\frac{F_k}{3}\right) = \left(\frac{F_k \bmod 3}{3}\right) = \left(\frac{2}{3}\right)$$

gem. Proposition 6.1.1 und Lemma 6.1.2. Dank Theorem 6.1.2 gilt schlussfolgernd, dass

$$\left(\frac{2}{3}\right) = (-1)^{(3^2-1)/8} = (-1)^{(9-1)/8} = (-1)^{8/8} = -1$$

und daher ist 3 ein quadratischer Nichtrest mod einer Fermatschen Primzahl. □

6.2. Beweis des Pépin-Primzahltests

Aufbauend auf den Lemmata von Abschnitt 6.1.2 und dem verbesserten Lucas-Primzahltest aus Abschnitt 5.4 kann nun der Pépin-Primzahltest bewiesen werden – vgl. hierzu u. a. die Beweisführung in [Crandall, R.; Pomerance, C., 2005, S. 174]:

Theorem 6.2.1. *Sei $F_k = 2^{2^k} + 1$ die k-te Fermat-Zahl und $k \in \mathbb{N}$ größer 0, so gilt:*

$$3^{(F_k-1)/2} \equiv -1 \pmod{F_k} \Leftrightarrow F_k \in \mathbb{P}$$

Beweis. (Fall: ⇒) Da 2 der einzige Primteiler von $F_k - 1$ ist, gilt gem. dem verbesserten Lucas-Primzahltest, siehe Theorem 5.4.1, dass F_k eine Primzahl ist, da

$$3^{(F_k-1)/2} \not\equiv 1 \pmod{F_k}$$

und

$$(3^{(F_k-1)/2}) \cdot (3^{(F_k-1)/2}) \equiv 3^{F_k-1} \equiv (-1) \cdot (-1) \equiv 1 \pmod{F_k}.$$

(Fall: ⇐) Da F_k prim ist, gilt gem. Lemma 6.1.3, dass

$$\left(\frac{3}{F_k}\right) = -1$$

und folglich ist

$$\left(\frac{3}{F_k}\right) \equiv 3^{(F_k-1)/2} \equiv -1 \pmod{F_k}$$

gem. dem Euler-Kriterium, siehe Theorem 6.1.3. □

In [Crandall, R.; Pomerance, C., 2005, S. 174] fügen Crandall und Pomerance ergänzend hinzu, dass Pépin beim Beweis des nach ihm benannten Primzahltests die Basis 5 statt der Basis 3 nutzte und dementsprechend $k > 1$ voraussetzte. Die Beweisführung ist der soeben präsentierten jedoch sehr ähnlich: Es ist lediglich noch zu zeigen, dass auch

$$F_k \equiv 2 \pmod{5}$$

gilt – bspw. mittels vollständiger Induktion und indem man sich bewusst macht, dass

$$\left(2^{2^k}\right)^2 = 2^{2^k} \cdot 2^{2^k} = 2^{2^k+2^k} = 2^{2 \cdot 2^k} = 2^{2^{k+1}}.$$

6.3. Pseudocode

Algorithmus 9: Pépin-Primzahltest

Input: Eine natürliche Zahl $k > 0$

```
1 F_k ← 2^(2^k) + 1
2 b ← 3^((F_k−1)/2) mod F_k
3 if b = F_k − 1 then
4 |  return „F_k is definitely prime“
5 end if
6 return „F_k is definitely composite“
```

Obiger Pseudocode basiert auf Theorem 6.2.1.

Für eine Implementierung siehe Listing C.10 auf Seite 65.

6.4. Laufzeituntersuchung

Die Zeitmessung wurde gem. der Beschreibung in Anhang C durchgeführt.

Exponent (k)	Dezimalstellen	Primzahl?	Zeit in s
10	309	nein	0,039
11	617	nein	0,074
12	1 234	nein	0,204
13	2 467	nein	1,302
14	4 933	nein	9,159
15	9 865	nein	67,188
16	19 729	nein	534,833

Tabelle 6.1.: Laufzeituntersuchung des Pépin-Primzahltests (Listing C.10).

Die kleineren Exponenten wurden in Tabelle 6.1 bewusst ausgespart, da die Zeit vernachlässigbar, quasi konstant war. Das Ergebnis entsprach jedoch stets der Erwartung: Für $1 \leq k \leq 4$ bestätigte Listing C.10 auf Seite 65, dass es sich um Primzahlen handelt. Bei F_0, einer Primzahl, irrte sich der Algorithmus, weswegen zuvor $k > 0$ vorausgesetzt wurde. Alle anderen Fermat-Zahlen wurden problemlos als zusammengesetzt erkannt.

Obige Tabelle ist zudem ein sehr schönes Beispiel für die immense Leistungssteigerung, die Computer in den letzten Jahrzehnten erfahren haben: Der (maschinelle) Beweis, dass F_{14} zusammengesetzt ist, stellte 1964 noch eine echte Herausforderung dar, siehe hierzu u. a. [Hurwitz, A.; Selfridge, J. L., 1964]. Heutzutage braucht es nicht einmal 10 Sekunden, um das damalige Ergebnis verifizieren zu können.

Doch schon bei F_{16} wird deutlich, was Fermat-Zahlen zur Herausforderung werden lässt: Sie wachsen unglaublich rasant. Mit jeder Inkrementierung des Exponenten verdoppelt sich die Anzahl der Dezimalstellen. So verwundert es kaum, dass der Pépin-Primzahltest nur bei wenigen Fermat-Zahlen zum Einsatz kam, vgl. hierzu u. a. [Wikipedia, 2014b].

Viel häufiger wird die Zusammengesetztheit einer Fermat-Zahl F_k dadurch bewiesen, dass ein „kleiner“ Faktor der Form

$$a \cdot 2^{k+1} + 1$$

gefunden wird. Euler, der als Erster beweisen konnte, dass *jeder* Faktor dieser Form entsprechen muss, nutzte dieses Wissen seinerzeit, um $641 = 10 \cdot 2^6 + 1$ als Teiler von F_5 zu finden und damit Fermats ursprüngliche Vermutung, dass alle F_k Primzahlen sind, zu widerlegen – vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 28]. Eine Übersicht zu vergangenen und aktuellen Rekorden rund um Fermat-Zahlen bietet [Keller, W., 2013].

7. Implementierung

Nachdem es in Kapitel 6 mit Hilfe des Pépin-Primzahltests in weniger als 10 Minuten gelungen ist, die Primalität einer Zahl mit fast 20 000 Dezimalstellen zweifelsfrei zu beurteilen, und damit alle im Rahmen dieser Arbeit aufgestellten „Rekorde“ in den Schatten zu stellen, ist es nun an der Zeit, einen Blick hinter die Kulissen zu werfen:

In Abschnitt 7.1 erfolgt eine Zusammenstellung all jener Methoden des Computeralgebrasystems Sage, die sich bei der Untersuchung und teilweise auch der Implementierung der zuvor vorgestellten Primzahltests als besonders nützlich erwiesen haben.

Im Anschluss werden die Details eines Phänomens beleuchtet, das erstmals in Abschnitt 4.2.2 zum Tragen kam und bei der Implementierung der meisten Algorithmen eine wesentliche Rolle spielte: Modulare Exponentiation, also das Berechnen von

$$b = a^e \bmod N$$

– in Python: `b = pow(a, e, N)` – ist auch bei vergleichsweise großem e überaus effizient.

7.1. Das Computeralgebrasystem Sage

Es gibt viele Gründe, warum man Sage als ernsthafte Alternative zu gängigen – meist kommerziellen – Computeralgebrasystemen in Betracht ziehen sollten. Für diese Arbeit waren i. W. folgende Aspekte ausschlaggebend, vgl. hierzu u. a. [Stein, W. A. et al., 2014]:

- Sage versucht nicht, das Rad neu zu erfinden. Viele Aufgaben werden an bereits bestehende, teils hochspezialisierte Software-Komponenten weitergereicht.
- Diesem Ansatz folgend hat Sage auch keine eigene Programmiersprache erfunden, sondern setzt stattdessen auf Python, genauer gesagt Python 2.7.
- Neben Python gibt es noch zwei weitere, sehr nützliche Möglichkeiten, Sage anzusteuern: über eine Weboberfläche im Browser oder eine interaktive Kommandozeile.

Und zu guter Letzt: Sage ist nicht nur frei erhältlich, sondern vor allem auch quelloffen. Dank dieser Offenheit war es stets ein Leichtes, die Funktionsweise der aufgerufenen Methoden bzw. der dahinterstehenden Algorithmen zu analysieren und zu verinnerlichen.

Folgende Methoden erwiesen sich im Rahmen dieser Arbeit als besonders nützlich:

- `divisors(N)`: Liste aller natürlichen Teiler (Teilermenge) von $N \neq 0$.

- `euler_phi(N)`: Eulersche Phi-Funktion. Für $N \leq 0$ ist `euler_phi(N)` gleich 0.
- `factor(N)`: Faktorisierung von $N \neq 0$. Für $N < 0$ gilt `-1 * factor(abs(N))`.
- `is_prime(N)`: `True`, falls N eine Primzahl ist, andernfalls `False`.
- `legendre_symbol(a, p)`: Das Legendre-Symbol (a/p), mit $a \in \mathbb{Z}$ und $p \in \mathbb{P}\backslash\{2\}$.
- `next_prime(N)`: Die nächste Primzahl, die größer als N ist.
- `list(primes(start, stop))`: Liste aller Primzahlen $start \leq p \leq stop - 1$.
- `prime_pi(N)`: Anzahl der Primzahlen, die nicht größer als N sind: $\pi(N)$.
- `primitive_root(p)`: Die kleinste positive Primitivwurzel mod einer Primzahl p.
- `xgcd(a, b)`: Tripel (g, s, t), sodass $ggT(a, b) = g = a \cdot s + b \cdot t$, mit $a, b, g, s, t \in \mathbb{Z}$.

Weiterführende Informationen und Beispiele sind der sehr ausführlichen Online-Dokumentation von Sage zu entnehmen: http://www.sagemath.org/doc/reference/misc/sage/rings/arith.html (aufgerufen am 18.05.2014). Der Primzahlzählfunktion $\pi(N)$ ist sogar eine eigene kleine Seite gewidmet worden: http://www.sagemath.org/doc/reference/functions/sage/functions/prime_pi.html (aufgerufen am 18.05.2014).

7.2. Die Programmiersprache Python

Aufgrund der Entscheidung, Sage als Computeralgebrasystem zu nutzen, war es naheliegend, Python auch für jene Programme zu nutzen, die ohne Sage-Methoden auskommen:

So war einerseits sichergestellt, dass alle Quellcodes einander gleichen und leicht miteinander verglichen werden können. Andererseits ist Python an und für sich eine gute Wahl, wenn die Lesbarkeit von Quellcode wichtiger als eine möglichst schnelle Programmausführung ist. Aus demselben Grund wurde auch auf viele der Möglichkeiten, die die Python-Syntax bietet, um kompakten Quellcode zu schreiben, bewusst verzichtet:

Vorrangiges Ziel waren lesbare, den Pseudocodes ähnelnde Quellcodes. Die mathematischen Ideen, die ihnen zugrunde lagen, sollten noch immer klar erkennbar sein.

Weiterführende Informationen und Beispiele sind der sehr ausführlichen Online-Dokumentation von Python zu entnehmen – siehe hierzu u. a. https://www.python.org/download/releases/2.7/ und https://docs.python.org/2.7/ (aufgerufen am 18.05.2014).

7.2.1. Exponentiation durch wiederholtes Quadrieren

Zur Erinnerung: Um zu entscheiden, ob eine ungerade natürliche Zahl $N > 3$ vermutlich prim oder definitiv zusammengesetzt ist, berechnet der Fermatsche Primzahltest

$$b = a^{N-1} \bmod N,$$

mit $2 \leq a \leq N-2$. Falls b gleich 1 ist, so ist N vermutlich prim, andernfalls definitiv zusammengesetzt. Die Laufzeit des Fermatschen Primzahltests hängt also ganz entscheidend von der Effizienz dieser Berechnung, *modulare Exponentiation* genannt, ab.

Die Laufzeituntersuchung in Abschnitt 4.2.2 hat gezeigt, dass der Aufruf `(a**(N-1))%N` alles andere als effizient ablief. Dies liegt vermutlich daran, dass dieser zuerst

$$a^{N-1} = a \cdot a \cdot \ldots \cdot a$$

berechnet und erst am Schluss den Rest mod N bestimmt. Da N zumeist vergleichsweise groß ist, würde dies auch das Voll- bzw. Überlaufen des Speichers erklären. Einen wesentlich geschickteren Algorithmus erhält man, wenn man die n-Bit-Zahl

$$N - 1 = n_0 \cdot 2^0 + n_1 \cdot 2^1 + \ldots + n_{n-1} \cdot 2^{n-1} = \sum_{i=0}^{n-1} n_i \cdot 2^i$$

im Binärsystem aufschreibt, n_i ist also entweder 0 oder 1, denn daraus folgt, dass

$$a^{N-1} = a^{n_0 \cdot 2^0 + n_1 \cdot 2^1 + \ldots + n_{n-1} \cdot 2^{n-1}} = a^{n_0 \cdot 2^0} \cdot a^{n_1 \cdot 2^1} \cdot \ldots \cdot a^{n_{n-1} \cdot 2^{n-1}} = \prod_{i=0}^{n-1} a^{n_i \cdot 2^i}.$$

Augenscheinlich kann man all jene Multiplikationen auslassen, bei denen $n_i = 0$ ist, denn dann ist $a^{0 \cdot 2^i} = 1$, was am Ergebnis nichts ändert. Wenn man sich nun noch in Erinnerung ruft, siehe Abschnitt 6.2, dass

$$\left(a^{2^i}\right)^2 = a^{2^i} \cdot a^{2^i} = a^{2^i + 2^i} = a^{2 \cdot 2^i} = a^{2^{i+1}},$$

ist man schon fast am Ziel: Nachfolgendem Algorithmus, der $b = a^{N-1} \bmod N$ durch wiederholtes Quadrieren und stetiges Reduzieren überaus effizient berechnet.

Algorithmus 10: Modulare Exponentiation

Input: Eine ungerade natürliche Zahl $N > 3$, eine natürliche Zahl a, mit $2 \leq a \leq N-2$

```
1  b ← 1
2  a ← a mod N
3  e ← N − 1
4  while e > 0 do
5      if e ist ungerade then
6          b ← (b · a) mod N          ▷ Das niederwertigste Bit von e ist 1
7      end if
8      e ← ⌊e/2⌋                      ▷ Bits um eine Stelle nach rechts verschieben
9      a ← (a · a) mod N
10 end while
11 return b                           ▷ b entspricht nun (a**(N-1))%N
```

Algorithmus 10 von Seite 52 basiert i. W. auf dem Pseudocode der *Right-to-left binary method*, die unter [Wikipedia, 2014a] zu finden ist. Die Schreibweise der Zeilen 5 und 8 ist von Algorithmus 14.76 aus [Menezes, A. J.; van Oorschot, P. C.; Vanstone, S. A., 1996, S. 614] inspiriert worden. Für eine Implementierung siehe Listing C.11 auf Seite 65.

Beispiel. Mit der soeben vorgestellten Methode soll $2^{12} \bmod 13$ berechnet werden: Gem. den ersten drei Zeilen des Pseudocodes ist

$$\begin{aligned} b &= 1, \\ a &= 2 \bmod 13 = 2, \\ e &= 13 - 1 = 12. \end{aligned}$$

Augenscheinlich ist 12 größer 0, aber nicht ungerade. Folglich wird e halbiert und a quadriert, sodass nun $e = 6$ resp. $a = 2 \cdot 2 \bmod 13 = 4$ gilt. Da 6 ebenfalls größer 0, aber nicht ungerade ist, wird e erneut halbiert und a ein weiteres Mal quadriert, sodass nun $e = 3$ resp. $a = 4 \cdot 4 \bmod 13 = 3$ gilt. Erstmals ist e nicht nur größer 0, sondern auch ungerade. Folglich wird b mit a multipliziert, $b = 1 \cdot 3 \bmod 13 = 3$, anschließend e halbiert sowie a quadriert, sodass nun $e = \lfloor 3/2 \rfloor = 1$ resp. $a = 3 \cdot 3 \bmod 13 = 9$ gilt.

Erneut ist e nicht nur größer 0, sondern auch ungerade. Folglich wird b nochmals mit a multipliziert, $b = 3 \cdot 9 \bmod 13 = 1$, e ein letztes Mal halbiert und a ein letztes Mal quadriert, sodass nun $e = \lfloor 1/2 \rfloor = 0$ resp. $a = 9 \cdot 9 \bmod 13 = 3$ gilt. Da e den Wert 0 angenommen hat, wird die Schleife verlassen und das Ergebnis verkündet:

$$b = 2^{12} \bmod 13 = 1$$

Um obiges Zahlenbeispiel – und dessen Korrektheit – intuitiv nachvollziehen zu können, kann es von Vorteil sein, sich $N - 1$ im Binärsystem aufzuschreiben und vorzustellen:

$$N - 1 = 12 = (1100)_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 2^3 + 2^2 + 0 + 0$$

Daraus folgt, dass

$$a^{N-1} = a^{12} = a^{2^3+2^2+0+0} = a^{2^3} \cdot a^{2^2} \cdot a^0 \cdot a^0 = a^{2^3} \cdot a^{2^2} \cdot 1 \cdot 1.$$

Setzt man nun $a = 2$, so erhält man $2^{2^3} = 2^8 = 256$ und $2^{2^2} = 2^4 = 16$ sowie ferner

$$\begin{aligned} 256 \bmod 13 &= 9, \\ 16 \bmod 13 &= 3, \end{aligned}$$

also genau die Werte, die bereits zuvor miteinander multipliziert werden mussten und zur Erkenntnis führten, dass

$$b = 2^{12} \bmod 13 = 9 \cdot 3 \bmod 13 = 1,$$

dass also 13 gem. dem Fermatschen Primzahltest vermutlich prim ist.

Nachfolgende Laufzeituntersuchung, deren Zeitmessung gem. der Beschreibung in Anhang C durchgeführt wurde, untermauert die Effizienz des obigen Verfahrens:

N	**Zeit in s:** `(2**(N-1))%N`	`modular_exponentiation(2, N)`
10^6	0,046	0,038
10^7	0,098	0,038
10^8	0,765	0,038
10^9	9,620	0,038
$10 \cdot 10^9$	–	0,038

Tabelle 7.1.: Laufzeituntersuchung von Listing C.11 (Modulare Exponentiation).

Die Lücke in Tabelle 7.1 ist der Tatsache geschuldet, dass der gesamte zur Verfügung stehende Speicher vollgelaufen war und das Betriebssystem den Prozess notgedrungen beendete. Und das bei einem vergleichsweise kleinen N, mit „nur“ 11 Dezimalstellen.

Im Gegensatz dazu blieb die Zeit, die Listing C.11 auf Seite 65 benötigte, praktisch konstant. Das Ergebnis schwankte nur minimal – in der dritten Nachkommastelle.

Ein Detail zum Schluss: Algorithmus 10 von Seite 52 ist, umgeben von Fehlerbehandlung und mit einer weniger strengen Vorgabe der Parameter, Grundlage der Python-Methode `int_pow()`[1], welche für die Exponentiation von *plain integers*, siehe Abschnitt 3.1.2, zuständig ist. Für *long integers* wird ein etwas komplexeres, ausgefeilteres Verfahren verwendet, das dem soeben vorgestellten jedoch nicht ganz unähnlich ist[2].

1 Siehe hierzu u. a.: http://svn.python.org/view/python/branches/release27-maint/Objects/intobject.c?view=markup (aufgerufen am 20.05.2014).

2 Siehe hierzu u. a.: http://svn.python.org/view/python/branches/release27-maint/Objects/longobject.c?view=markup (aufgerufen am 21.05.2014).

8. Fazit und Ausblick

Ausgangspunkt der vorliegenden Arbeit war eine – scheinbar einfache – Frage: Kann man den Kleinen Satz von Fermat, siehe Abschnitt 4.2, als Primzahltest verwenden?

Am Ende ist klar: man kann – jedenfalls dann, wenn man in Kauf nimmt, dass sich der daraus ergebende Primzahltest, siehe Kapitel 4, in einigen wenigen Fällen irren wird. Schuld sind die Fermatschen Pseudoprimzahlen, zusammengesetzte Zahlen, die sich dem Kleinen Satz von Fermat gegenüber wie Primzahlen verhalten. Sie sind zwar selten, seltener noch als Primzahlen, aber es gibt unendlich viele von ihnen. Und nicht nur das:

Es gibt sogar Zahlen, die zu *jeder* teilerfremden Basis pseudoprim sind: Carmichael-Zahlen, siehe Abschnitt 4.4. Diese sind zwar noch seltener, aber es gibt unendlich viele von ihnen. Zudem gibt es unendlich viele Carmichael-Zahlen, die keine „kleinen" Primfaktoren enthalten und nur mit viel Glück als zusammengesetzt entlarvt werden können.

Trotz dieser Nachteile kann man die Bedeutung des Kleinen Satzes von Fermat kaum überschätzen. Wie ein roter Faden zieht er sich durch die Geschichte der Primzahltests:

Der Lucas-Primzahltest, siehe Kapitel 5, ist i. W. eine Umkehrung des Kleinen Satzes von Fermat, mit der sich effizient beweisen lässt, dass eine Zahl N prim ist, wenn die Faktorisierung von $N - 1$ leicht zu ermitteln ist. Eine Eigenart, die sich der Pépin-Primzahltest, siehe Kapitel 6, zunutze macht, um die Primalität von Fermat-Zahlen zu beurteilen. Jene Zahlen, die schnell tausende oder gar Millionen Dezimalstellen lang sind.

Und selbst der Miller-Rabin-Primzahltest, jener Test, der in [BSI, 2014, S. 69–70] als probabilistischer Primzahltest empfohlen wird, beruht auf dem Kleinen Satz von Fermat.

Er erweitert diesen jedoch so geschickt, dass sich die Probleme des Fermatschen Primzahltests weniger stark auswirken bzw. verschwinden: Zusammengesetzte Zahlen, die sich dem Miller-Rabin-Primzahltest gegenüber wie Primzahlen verhalten, *starke Pseudoprimzahlen*, kommen zwar vor, sind aber viel seltener als Fermatsche Pseudoprimzahlen. Und so etwas wie „*starke* Carmichael-Zahlen" kennt der Miller-Rabin-Primzahltest nicht.

Es ist somit nicht übertrieben zu sagen, dass er den Fermatschen Primzahltest abgelöst – quasi verdrängt – hat. Schließlich behält der Miller-Rabin-Primzahltest alle Vorteile des Fermatschen Primzahltests bei und beseitigt bzw. mildert einige Nachteile, fügt aber keine neuen hinzu, vgl. hierzu u. a. [Crandall, R.; Pomerance, C., 2005, S. 135–142].

A. Notationen

$\mathbb{N}$	Menge der natürlichen Zahlen, $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$
$\mathbb{Z}$	Menge der ganzen Zahlen, $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{Q}$	Menge der rationalen Zahlen, $\mathbb{Q} = \left\{\frac{v}{w} \mid v, w \in \mathbb{Z}, w \neq 0\right\}$
$\mathbb{I}$	Menge der irrationalen Zahlen, $\mathbb{I} = \left\{\sqrt{2}, e, \pi, \ldots\right\}$
$\mathbb{R}$	Menge der reellen Zahlen, $\mathbb{R} = \mathbb{Q} \cup \mathbb{I}$
$\mathbb{P}$	Menge der Primzahlen, $\mathbb{P} = \{2, 3, 5, 7, 11, 13, \ldots\}$
$d \mid N$	d teilt N, mit $d, N \in \mathbb{Z} \Leftrightarrow \exists k \in \mathbb{Z}$, mit $N = k \cdot d$
$d \nmid N$	d teilt N nicht, mit $d, N \in \mathbb{Z} \Leftrightarrow \forall k \in \mathbb{Z}$ gilt, dass $N \neq k \cdot d$
$ggT(a, b)$	größter gemeinsamer Teiler von $a \in \mathbb{Z}$ und $b \in \mathbb{Z}$, $ggT(a, b) = 1 \Leftrightarrow a$ und b sind teilerfremd
$\lfloor x \rfloor$	Abrundungsfunktion: Die größte ganze Zahl, die kleiner oder gleich $x \in \mathbb{R}$ ist

A.1. Komplexitätstheorie

Sei im Folgenden x eine natürliche Zahl.

$f(x) \sim g(x)$ — $f(x)$ und $g(x)$ verhalten sich asymptotisch gleich:

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 1$$

$f(x) = o(g(x))$ — $f(x)$ ist gegenüber $g(x)$ asymptotisch vernachlässigbar:

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

A.2. Gruppentheorie

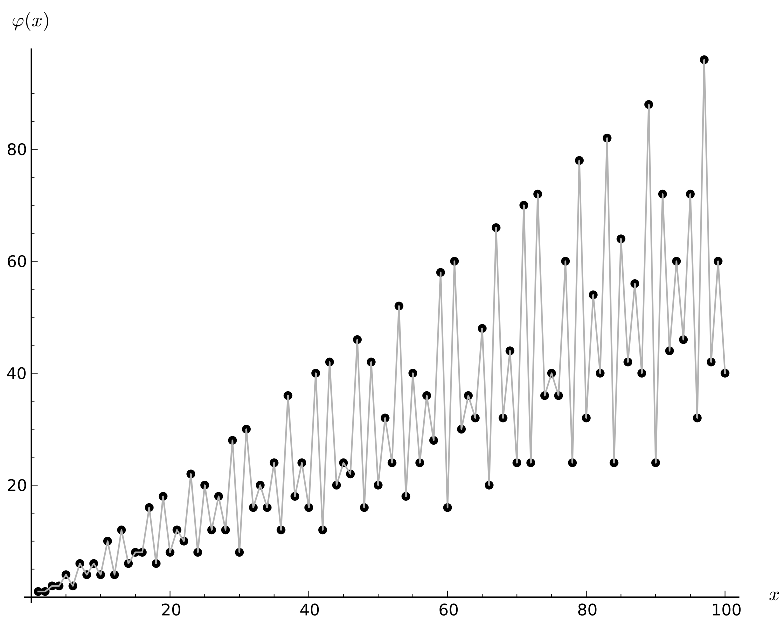
Sei im Folgenden G eine endliche Gruppe, $g \in G$ und e ihr neutrales Element. Ferner sei a eine ganze Zahl und N eine natürliche Zahl größer 0.

$\|G\|$	Ordnung von G: Anzahl der Elemente von G
$\|g\|$	Ordnung von g: Kleinste natürliche Zahl $k > 0$, sodass $g^k = e$, existiert kein solches k, hat g unendliche Ordnung
$ord_N([a])$	Ordnung von $[a]$ mod N: Kleinste natürliche Zahl $k > 0$, sodass $[a]^k = [1] \Leftrightarrow a^k \equiv 1 \pmod{N}$, $ord_N([a]) = ord_N(a)$
$\varphi(N)$	Eulersche Phi-Funktion: Anzahl der zu N teilerfremden natürlichen Zahlen $m > 0$, die nicht größer als N sind, $\varphi(1) = 1$, siehe auch Abb. A.1 auf Seite 58
$\mathbb{Z}_N^*$	Prime Restklassengruppe: Gruppe der Restklassen, deren Repräsentanten teilerfremd zu N sind, $\|\mathbb{Z}_N^*\| = \varphi(N)$

A.3. Zahlentheorie

Sei im Folgenden x eine natürliche Zahl.

$\pi(x)$	Primzahlzählfunktion: Anzahl der Primzahlen, die nicht größer als x sind, siehe Abschnitt 2.3 und Abb. 2.1 von Seite 6
$P\pi_a(x)$	Anzahl Fermatscher Pseudoprimzahlen zur Basis a, die nicht größer als x sind, $P\pi(x) = P\pi_2(x)$, siehe Abschnitt 4.3
$C(x)$	Anzahl der Carmichael-Zahlen, die nicht größer als x sind, siehe Abschnitt 4.4



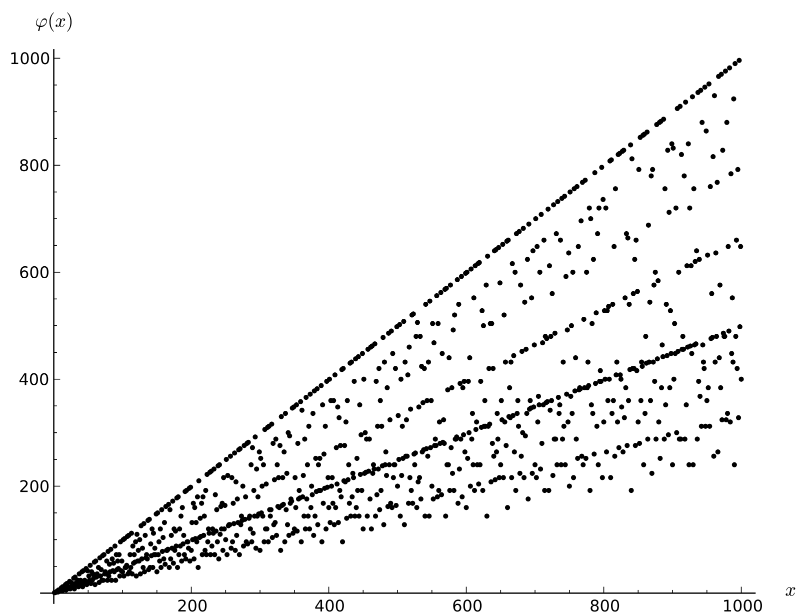


Abbildung A.1.: Die ersten hundert bzw. tausend Werte der Eulerschen Phi-Funktion.

B. Zahlenfolgen

Alle Zahlenfolgen sind unter http://oeis.org/ (aufgerufen am 04.04.2014) einzusehen – die englischen Bezeichnungen wurden i. W. unverändert übernommen:

A000010	Euler's totient function $\varphi(n)$.
A000040	The prime numbers.
A000215	Fermat numbers: $2^{2^k}+1$, $k \geq 0$.
A000720	$\pi(n)$, the number of primes $\leq n$.
A001567	Fermat pseudoprimes to base 2.
A001918	Least positive primitive root of n-th prime.
A002808	The composite numbers.
A002997	Carmichael numbers.
A005935	Fermat pseudoprimes to base 3.
A006880	Number of primes $< 10^n$.
A010554	$\varphi(\varphi(n))$, where $\varphi(n)$ is Euler's totient function.
A019434	Fermat primes: primes of form $2^{2^k}+1$, for some $k \geq 0$.
A033948	Numbers that have a primitive root.
A033949	Positive integers that do not have a primitive root.
A046144	Number of primitive roots modulo n.
A052155	Fermat pseudoprimes to both base 2 and base 3.
A055550	Number of Fermat pseudoprimes to base 2 less than 10^n.
A055553	Number of Carmichael numbers less than 10^n.
A057755	Number of digits in n-th Fermat number.
A114245	Number of Fermat pseudoprimes to base 3 less than 10^n.
A114246	Number of Fermat pseudoprimes to bases 2 and 3 less than 10^n.

C. Quellcode

Nachfolgende Programme bzw. Methoden wurden im Rahmen dieser Arbeit einer Laufzeituntersuchung unterzogen, um ihre Leistungsfähigkeit beurteilen bzw. miteinander vergleichen zu können. Um die äußeren Einflüsse zu minimieren, wurden alle Zeitmessungen auf einem eigens zur Laufzeituntersuchung eingerichteten System durchgeführt, das über eine CPU mit 3,90 GHz Taktfrequenz und ca. 2 GB Arbeitsspeicher verfügte.

Die Parameter wurden den Methoden immer direkt, nie über die Kommandozeile übergeben. Beim Sieb des Eratosthenes, siehe Listing C.2 auf Seite 61, wurde auf eine Ausgabe via `print()` verzichtet. Stattdessen wurde der Rückgabewert der Methode, ein vergleichsweise großes Array, einer Variablen zugewiesen, vgl. hierzu auch Abschnitt 3.2.2.

Die Zeitmessung selbst erfolgte mit Hilfe des Kommandozeilen-Programms *time*. Der Wert *Zeit in s*, der bei jeder vorgenommenen Laufzeituntersuchung vermerkt wurde, entspricht der *real* verstrichenen Zeit, vgl. hierzu u. a. das Manual von *time*.

Beispiel. Um zu erfahren, wie lange der Fermatsche Primzahltest benötigt, um $10^{400} + 69$ zur Basis 2 zu beurteilen, wurde Listing C.3 auf Seite 61 um folgende Zeile ergänzt:

```
1 print fermat_primality_test(1000000000000000000000000000000000000...69, 2)
```

Anschließend konnte die *real* verstrichene Zeit mit Hilfe von *time* ermittelt werden:

```
1 $ time python fermat_primality_test.py
2 1000000000000000000000000000000000000...69 is probably prime
3
4 real    0m0.054s
```

C.1. Probedivision als Primzahltest

Listing C.1: trial_division.py

```
1 #!/usr/bin/python2.7
2
3 def trial_division(N):
4
5     if N%2 == 0:
6         return str(N) + " is definitely composite"
7
8     d = 3
```

```
 9        while d*d <= N:
10            if N%d == 0:
11                return str(N) + " is definitely composite"
12            d = d + 2
13
14        return str(N) + " is definitely prime"
```

Listing C.1 implementiert Algorithmus 1 von Seite 9 für $N \in \mathbb{N} \backslash \{0, 1, 2\}$.

C.2. Sieb des Eratosthenes

Listing C.2: sieve_of_eratosthenes.py

```
 1  #!/usr/bin/python2.7
 2
 3  def sieve_of_eratosthenes(N):
 4
 5      P = []
 6      A = [None]*(N + 1)
 7
 8      m = 2
 9      while m <= N:
10          A[m] = True
11          m = m + 1
12
13      m = 2
14      while m*m <= N:
15          if A[m] == True:
16              P.append(m)
17              j = m*m
18              while j <= N:
19                  A[j] = False
20                  j = j + m
21          m = m + 1
22
23      while m <= N:
24          if A[m] == True:
25              P.append(m)
26          m = m + 1
27
28      return P
```

Listing C.2 implementiert Algorithmus 2 von Seite 12 für $N \in \mathbb{N} \backslash \{0, 1\}$.

C.3. Fermatscher Primzahltest

Listing C.3: fermat_primality_test.py

```
 1  #!/usr/bin/python2.7
 2
```

```
3  def fermat_primality_test(N, a):
4
5      b = pow(a, N-1, N)
6
7      if b == 1:
8          return str(N) + " is probably prime"
9
10     return str(N) + " is definitely composite"
```

Listing C.3 implementiert Algorithmus 3 von Seite 17 für eine ungerade natürliche Zahl $N > 3$ und eine natürliche Zahl a, mit $2 \leq a \leq N - 2$.

C.4. Primzahlen finden

Listing C.4: find_prime.py

```
1  #!/usr/bin/python2.7
2
3  from random import randint
4  from fermat_primality_test import *
5
6  def find_prime(n, T):
7
8      while True:
9          N = randint(pow(2, n-1) + 1, pow(2, n) - 1)
10         if N%2 != 0:
11             i = 1
12             while i <= T:
13                 a = randint(2, N - 2)
14                 if fermat_primality_test(N, a) == str(N) + " is definitely
                       composite":
15                     break
16                 i = i + 1
17             if i > T:
18                 return N
```

Listing C.4 implementiert Algorithmus 4 von Seite 27 für eine natürliche Zahl $n \geq 3$ und eine natürliche Zahl $T \geq 1$.

C.5. Vorläufer des Lucas-Primzahltests

Listing C.5: lucas_primality_test_1.sage

```
1  #!/usr/bin/env sage
2
3  def lucas_primality_test_1(N, a):
4
5      b = pow(a, N-1, N)
6
7      if b == 1:
```

```
 8          m = 1
 9          while m < N-1:
10              b = pow(a, m, N)
11              if b == 1:
12                  return
13              m = m + 1
14          return str(N) + " is definitely prime"
15
16      return str(N) + " is definitely composite"
```

Listing C.5 implementiert Algorithmus 5 von Seite 33 für eine natürliche Zahl $N > 2$ und eine natürliche Zahl a, mit $1 < a < N$.

C.6. Lucas-Primzahltest

Listing C.6: lucas_primality_test_2.sage

```
 1  #!/usr/bin/env sage
 2
 3  def lucas_primality_test_2(N, a):
 4
 5      b = pow(a, N-1, N)
 6
 7      if b == 1:
 8          for m in divisors(N-1)[1:-1]:
 9              b = pow(a, m, N)
10              if b == 1:
11                  return
12          return str(N) + " is definitely prime"
13
14      return str(N) + " is definitely composite"
```

Listing C.6 implementiert Algorithmus 6 von Seite 35 für eine natürliche Zahl $N > 2$ und eine natürliche Zahl a, mit $1 < a < N$.

C.7. Verbesserter Lucas-Primzahltest

Listing C.7: lucas_primality_test_3.sage

```
 1  #!/usr/bin/env sage
 2
 3  def lucas_primality_test_3(N, a):
 4
 5      b = pow(a, N-1, N)
 6
 7      if b == 1:
 8          for q in factor(N-1):
 9              b = pow(a, (N-1)//q[0], N)
10              if b == 1:
11                  return
```

```
12          return str(N) + " is definitely prime"
13
14      return str(N) + " is definitely composite"
```

Listing C.7 implementiert Algorithmus 7 von Seite 39 für eine natürliche Zahl $N > 2$ und eine natürliche Zahl a, mit $1 < a < N$.

C.7.1. Implementierung ohne Sage

Listing C.8: factor.py

```
1   #!/usr/bin/python2.7
2
3   def factor(N):
4
5       F = []
6
7       while N%2 == 0:
8           N = N/2
9           F.append(2)
10
11      d = 3
12      while d*d <= N:
13          while N%d == 0:
14              N = N/d
15              F.append(d)
16          d = d + 2
17
18      if N != 1:
19          F.append(N)
20
21      return F
```

Listing C.8 implementiert Algorithmus 8 von Seite 41 für $N \in \mathbb{N} \setminus \{0, 1\}$.

Listing C.9: lucas_primality_test.py

```
1   #!/usr/bin/python2.7
2
3   from factor import *
4
5   def lucas_primality_test(N, a):
6
7       b = pow(a, N-1, N)
8
9       if b == 1:
10          for q in set(factor(N-1)):
11              b = pow(a, (N-1)//q, N)
12              if b == 1:
13                  return
14          return str(N) + " is definitely prime"
15
```

```
16      return str(N) + " is definitely composite"
```

Listing C.9 implementiert Algorithmus 7 von Seite 39 für eine natürliche Zahl $N > 2$ und eine natürliche Zahl a, mit $1 < a < N$.

C.8. Pépin-Primzahltest

Listing C.10: pepin_primality_test.py

```
1  #!/usr/bin/python2.7
2
3  def pepin_primality_test(k):
4
5      Fk = pow(2, pow(2, k)) + 1
6      b = pow(3, (Fk - 1)//2, Fk)
7
8      if b == Fk - 1:
9          return "F_" + str(k) + " is definitely prime"
10
11     return "F_" + str(k) + " is definitely composite"
```

Listing C.10 implementiert Algorithmus 9 von Seite 48 für eine natürliche Zahl $k > 0$.

C.9. Modulare Exponentiation

Listing C.11: modular_exponentiation.py

```
1  #!/usr/bin/python2.7
2
3  def modular_exponentiation(a, N):
4
5      b = 1
6      a = a%N
7      e = N-1
8
9      while e > 0:
10         if e & 1:
11             b = (b*a)%N
12         e = e//2
13         a = (a*a)%N
14
15     return b
```

Listing C.11 implementiert Algorithmus 10 von Seite 52 für eine ungerade natürliche Zahl $N > 3$ und eine natürliche Zahl a, mit $2 \leq a \leq N - 2$.

D. Zusammenfassung

Der Kleine Satz von Fermat besagt, dass für jede Primzahl N und jede dazu teilerfremde natürliche Zahl a stets

$$a^{N-1} \equiv 1 \pmod{N}$$

gilt. Die Umkehrung gilt jedoch nicht. Es gibt unendlich viele Zahlen N, welche die Kongruenz erfüllen, obwohl sie zusammengesetzt sind: Fermatsche Pseudoprimzahlen (zur Basis a). Da diese selten sind, seltener noch als Primzahlen, und weil die modulare Exponentiation, also die Berechnung von $b = a^{N-1} \bmod N$, auch bei vergleichsweise großen Zahlen effizient ist, ist der Kleine Satz von Fermat prinzipiell als Primzahltest geeignet:

Um zu entscheiden, ob eine Zahl N prim ist, berechnet man für eine Reihe zufällig gewählter Basen $1 < a < N$, ob $b = a^{N-1} \bmod N$ gleich oder ungleich 1 ist. Ist b jedes Mal gleich 1, so ist N vermutlich prim. Andernfalls ist N definitiv zusammengesetzt.

Je mehr Basen man testet, desto wahrscheinlicher ist, dass N tatsächlich prim ist. Problematisch wird es immer dann, wenn N eine Carmichael-Zahl ist, also eine zusammengesetzte Zahl, die zu *jeder* teilerfremden Basis pseudoprim ist, und N keine „kleinen" Primfaktoren enthält. Solche Zahlen sind zwar überaus selten, aber es gibt unendlich viele von ihnen und sie werden von obigem Verfahren nur mit viel Glück entlarvt.

Eine Möglichkeit, um diesem Problem zu begegnen, bietet der Lucas-Primzahltest, der den Kleinen Satz von Fermat umkehrt, indem er ihm eine weitere Bedingung hinzufügt, dank der sich dann zweifelsfrei feststellen lässt, ob N prim ist. Da diese Bedingung jedoch die Faktorisierung von $N - 1$ voraussetzt, ist der Lucas-Primzahltest nur dann wirklich effizient, wenn die Faktorisierung von $N-1$ bekannt oder leicht zu ermitteln ist.

Eine Eigenart, die im Pépin-Primzahltest Verwendung findet. Jenem Primzahltest, der speziell auf Fermat-Zahlen

$$F_k = 2^{2^k} + 1,$$

mit $k \in \mathbb{N}$, zugeschnitten ist und sich zunutze macht, dass $F_k - 1$ nur einen einzigen Primfaktor enthält: die Zahl 2. Für beliebige Zahlen ist dieser Primzahltest zwar augenscheinlich ungeeignet. Die Leistungsfähigkeit in Bezug auf die rasant wachsenden Fermat-Zahlen, die schnell tausende Dezimalstellen lang sind, ist jedoch beeindruckend.

Ziel der vorliegenden Arbeit ist es, diese und weitere Primzahltests theoretisch wie praktisch zu analysieren, um ihre jeweiligen Vor- bzw. Nachteile herausarbeiten und miteinander vergleichen zu können. Um dies zu erreichen, liegt der Schwerpunkt der Analysen

auf den mathematischen Ideen hinter den Primzahltests. Erst wenn diese vollständig verstanden und rigoros bewiesen wurden, folgen Zahlenbeispiele, Pseudocode und eine beispielhafte Implementierung in Python bzw. im Falle des Lucas-Primzahltests in Sage.

Dank dieser kann die Leistungsfähigkeit der Primzahltests im Rahmen einer Laufzeituntersuchung betrachtet und die Frage beantwortet werden, ob der jeweilige Primzahltest dazu geeignet ist, Primzahlen zu finden, die groß genug sind, um bspw. in kryptographischen Verfahren weiterverwendet zu werden. Darüber hinaus bietet diese praktische Komponente die Chance, die Theorie zu reflektieren und leichter nachprüfbar zu machen.

Literaturverzeichnis

[Alford, W. R.; Granville, A.; Pomerance, C., 1994a] Alford, W. R.; Granville, A.; Pomerance, C. (1994a). On the difficulty of finding reliable witnesses. In *Algorithmic Number Theory: First International Symposium, ANTS-I, Ithaca, NY, USA*, volume 877, pages 1–16. Springer-Verlag. ISBN: 978-3-540-58691-3.

[Alford, W. R.; Granville, A.; Pomerance, C., 1994b] Alford, W. R.; Granville, A.; Pomerance, C. (1994b). There are infinitely many Carmichael numbers. *Annals of Mathematics*, 139(3):703–722.

[BSI, 2014] BSI (2014). *Kryptographische Verfahren: Empfehlungen und Schlüssellängen (Version 2014-01)*. http://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm.html (aufgerufen am 13.04.2014).

[Burgess, D. A., 1962] Burgess, D. A. (1962). On Character Sums and Primitive Roots. *Proceedings of the London Mathematical Society*, 12(3):179–192.

[Crandall, R. E.; Mayer, E. W.; Papadopoulos, J. S., 2003] Crandall, R. E.; Mayer, E. W.; Papadopoulos, J. S. (2003). The twenty-fourth Fermat number is composite. *Mathematics of Computation*, 72(243):1555–1572.

[Crandall, R.; Pomerance, C., 2005] Crandall, R.; Pomerance, C. (2005). *Prime Numbers: A Computational Perspective*. Springer Science+Business Media, Inc., 2nd edition. ISBN: 978-0-387-25282-7.

[Erdős, P., 1956] Erdős, P. (1956). On pseudoprimes and Carmichael numbers. *Publicationes Mathematicae Debrecen*, 4:201–206.

[Ernesti, J.; Kaiser, P., 2008] Ernesti, J.; Kaiser, P. (2008). *Python: Das umfassende Handbuch*. Galileo Press. http://openbook.galileocomputing.de/python/ (aufgerufen am 12.05.2014). ISBN: 978-3-836-21110-9.

[Grosswald, E., 1981] Grosswald, E. (1981). On Burgess' Bound for Primitive Roots Modulo Primes and an Application to $\Gamma(p)$. *American Journal of Mathematics*, 103(6):1171–1183.

[Hurwitz, A.; Selfridge, J. L., 1964] Hurwitz, A.; Selfridge, J. L. (1964). Fermat Numbers and Mersenne Numbers. *Mathematics of Computation*, 18(85):146–148.

[Keller, W., 2013] Keller, W. (2013). Prime factors $k \cdot 2^n + 1$ of Fermat numbers F_m and complete factoring status. http://www.prothsearch.net/fermat.html (aufgerufen am 15.05.2014).

[Kim, S. H.; Pomerance, C., 1989] Kim, S. H.; Pomerance, C. (1989). The Probability that a Random Probable Prime is Composite. *Mathematics of Computation*, 53(188):721–741.

[MathPages, 2014] MathPages (2014). Lucas's Primality Test With Factored $N-1$. http://www.mathpages.com/home/kmath473.htm (aufgerufen am 08.05.2014).

[Menezes, A. J.; van Oorschot, P. C.; Vanstone, S. A., 1996] Menezes, A. J.; van Oorschot, P. C.; Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press. http://cacr.uwaterloo.ca/hac/ (aufgerufen am 20.05.2014). ISBN: 978-0-849-38523-0.

[Pinch, R. G. E., 2007] Pinch, R. G. E. (2007). The Carmichael numbers up to 10^{21}. In *Proceedings of Conference on Algorithmic Number Theory 2007*, number 46, pages 129–131. Turku Centre for Computer Science.

[Pomerance, C., 1981] Pomerance, C. (1981). On the Distribution of Pseudoprimes. *Mathematics of Computation*, 37(156):587–593.

[Pomerance, C.; Selfridge, J. L.; Wagstaff, Jr., S. S., 1980] Pomerance, C.; Selfridge, J. L.; Wagstaff, Jr., S. S. (1980). The Pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026.

[ProofWiki, 2012] ProofWiki (2012). Euclid: The Elements. http://www.proofwiki.org/w/index.php?title=ProofWiki:Books/Euclid/The_Elements&oldid=85434 (aufgerufen am 14.04.2014).

[ProofWiki, 2013a] ProofWiki (2013a). Fermat's Little Theorem. http://www.proofwiki.org/w/index.php?title=Fermat's_Little_Theorem&oldid=169786 (aufgerufen am 13.04.2014).

[ProofWiki, 2013b] ProofWiki (2013b). Order of Element Divides Order of Finite Group. http://www.proofwiki.org/w/index.php?title=Order_of_Element_Divides_Order_of_Finite_Group&oldid=169750 (aufgerufen am 13.04.2014).

[Ribenboim, P., 2011] Ribenboim, P. (2011). *Die Welt der Primzahlen*. Springer-Verlag, 2. Auflage. ISBN: 978-3-642-18078-1.

[Scheid, H.; Frommer, A., 2013] Scheid, H.; Frommer, A. (2013). *Zahlentheorie*. Springer-Verlag, 4. Auflage. ISBN: 978-3-642-36835-6.

[Stein, W. A. et al., 2014] Stein, W. A. et al. (2014). *Sage Mathematics Software (Version 6.1.1)*. The Sage Development Team. http://www.sagemath.org/ (aufgerufen am 14.04.2014).

[Weisstein, E. W., 2014a] Weisstein, E. W. (2014a). "Euclid's Theorems." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/EuclidsTheorems.html (aufgerufen am 14.04.2014).

[Weisstein, E. W., 2014b] Weisstein, E. W. (2014b). "Prime Arithmetic Progression." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/PrimeArithmeticProgression.html (aufgerufen am 11.04.2014).

[Weisstein, E. W., 2014c] Weisstein, E. W. (2014c). "Prime Counting Function." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/PrimeCountingFunction.html (aufgerufen am 04.04.2014).

[Weisstein, E. W., 2007] Weisstein, E. W. (2007). "Gauss, Karl Friedrich (1777-1855)." Eric Weisstein's World of Scientific Biography. http://scienceworld.wolfram.com/biography/Gauss.html (aufgerufen am 24.05.2014).

[Wikipedia, 2013a] Wikipedia (2013a). Fermatscher Primzahltest. http://de.wikipedia.org/w/index.php?title=Fermatscher_Primzahltest&oldid=115960768 (aufgerufen am 14.04.2014).

[Wikipedia, 2013b] Wikipedia (2013b). Lucas-Test (Mathematik). http://de.wikipedia.org/w/index.php?title=Lucas-Test_(Mathematik)&oldid=116893243 (aufgerufen am 27.04.2014).

[Wikipedia, 2014a] Wikipedia (2014a). Modular exponentiation. http://en.wikipedia.org/w/index.php?title=Modular_exponentiation&oldid=597521911 (aufgerufen am 20.05.2014).

[Wikipedia, 2014b] Wikipedia (2014b). Pépin's test. http://en.wikipedia.org/w/index.php?title=Pépin's_test&oldid=600219097 (aufgerufen am 15.05.2014).

[Wikipedia, 2014c] Wikipedia (2014c). Primitivwurzel. http://de.wikipedia.org/w/index.php?title=Primitivwurzel&oldid=128491054 (aufgerufen am 30.04.2014).

[Wikipedia, 2014d] Wikipedia (2014d). Sieb des Eratosthenes. http://de.wikipedia.org/w/index.php?title=Sieb_des_Eratosthenes&oldid=128248293 (aufgerufen am 10.04.2014).