"PRIMES is in P"

Holger Szillat

szillat@informatik.uni-tuebingen.de

21. Juni 2005

Zusammenfassung

Im Jahr 2002 erschreckte die "Entdeckung" von Manindra Agrawal, Neeraj Kayal und Nitin Saxena die Welt der Theoretischen Informatik: Die Entscheidung, ob eine Zahl eine Primzahl ist oder nicht, ist in polynomieller Zeit zu finden. Bisher war man davon ausgegangen, dass ein Algorithmus die Entscheidung zwar in polynomieller Zeit treffen kann, aber u.U. sehr lange dafür braucht. Randomisierte Algorithmen sind zwar schneller, haben aber eine gewisse Fehlerquote beim Ergebnis. Der deterministische Algorithmus von Agrawal, Kayal und Saxena kann die Lösung in polynomieller Zeit finden, ohne auf bisher unbewiesene mathematische Theoreme zurückgreifen zu müssen.

Inhaltsverzeichnis

1 Einführung

Es ist ein elementares Problem der Zahlentheorie: Ist eine gegebene ganze Zahl n prim oder nicht?

Primzahlen haben die Menschen schon in der Antike beschäftigt. Waren sie lange Zeit "nur" Spielzeuge der Mathematik, so haben sie spätestens mit der Einführung des Computers als Kommunikationsmedium an Bedeutung gewonnen, da sie eine der Grundlagen für viele Verschlüsselungsalgorithmen bilden.

Eines der ersten Verfahren zum Primzahl-Test geht auf die "alten Griechen" zurück, das wohlbekannte "Sieb des Eratosthenes" (ca. 240 vor Christus): wird n von einer Zahl $m \leq \sqrt{n}$ geteilt, dann muss n zusammengesetzt sein, ansonsten prim. Dieses Verfahren ist sowohl einfach, als auch sehr langsam, denn es sind mindestens $\sqrt{n}$ Schritte notwendig um sicher sagen zu können, ob n eine Primzahl ist oder nicht. Für große Zahlen n mit mehreren Hundert oder Tausend Ziffern ist dieses Verfahren nicht praktikabel.

Ein brauchbares Verfahren sollte nur polynomiell viele Schritte, in Bezug auf die Länge der Zahl, brauchen. Solche Verfahren existieren schon lange, wie z.B. der Miller/Rabin-Test oder der Solovay/Strassen-Algorithmus. Diese Verfahren sind allerdings randomisiert, d.h. das Ergebnis des Algorithmus kann(!) falsch sein.

Das Problem des Primzahl-problems ist in der sog. Komplexitätsklasse *co-NP*, d.h. liefert ein Algorithmus "n nicht prim" zurück, so ist dies dadurch verifizierbar, dass man n durch einen seiner nicht-trivialen Teiler teilt. Das Problem dabei ist aber, dass man einen solchen Teiler ersteinmal finden muß, was gerade dem Primzahl-Test selbst entspricht. Damit ist das Primzahl-problem ist der Komplexitätsklasse $NP \cap$ *co-NP*, d.h. es könnte keinen deterministischen, unbedingten Algorithmus geben, der das Problem in polynomieller Zeit löst.

Deterministische Algorithmen, die das Problem in polynomieller Zeit lösen können sind bekannt, z.B. das Verfahren von Miller. Jedoch basiert dieser Algorithmus auf der (noch) unbewiesenen "Erweiterten Rhiemann Hypothese".

Daher war das Ziel vieler Forschungen gewesen, einen (nicht-randomisierten) Algorithmus zu finden, der in polynomieller Zeit entscheiden kann, ob n eine Primzahl ist, oder nicht. Dieses Ziel ist mit dem *Agrawal-Kayal-Saxena*-Algorithmus definitiv erreicht. In der Praxis hat diese Erkenntnis aber keine Auswirkung, da die absolute Laufzeit (viel) größer ist, als die der randomisierten Algorithmen.

2 Notationen und Definitionen

Vorweg einige Notationen und Definitionen.

2.1 Allgemeine Notationen

Notation. Sei $\mathbb{P}$ die Menge aller Primzahlen. PRIMES bezeichne das Entscheidungsproblem, ob für eine gegebene (ungerade) Zahl $n \in \mathbb{N}$ gilt: $n \in \mathbb{P}$.

2.2 Wichtige Definitionen

Definition 1. Zwei Zahlen a und b heißen **relativ prim** wenn gilt: $\gcd(a, b) = 1$.

Einige Eigenschaften von "relativ prim":

- 1 ist relativ prim zu jeder ganzen Zahl.
- 0 ist nur zu 1 und -1 relativ prim.
- Eine effiziente Berechnung ist mit dem euklidischen Algorithmus möglich.

Korollar 1.1. Wenn a und b relativ prim zueinander und $br \equiv bs \pmod{a}, b, s \in \mathbb{N}$, dann $r \equiv s \pmod{a}$.

Korollar 1.2. Wenn a und b_1 relativ prim, und a und b_2 relativ prim, dann auch a und $b_1 b_2$.

Korollar 1.3. Wenn a und b relativ prim und $a|bc$, dann auch $a|c$.

Definition 2 (Carmichael-Zahl). Eine (zusammengesetzte) Zahl $n \in \mathbb{N}$ heißt *Carmichael-Zahl*, gdw. für alle $a \in \mathbb{Z}_n^*$ gilt:

$$a^{n-1} \equiv 1 \pmod{n} \tag{1}$$

Das kleinste Beispiel einer Carmichael-Zahl ist 561, welche in $3 \times 11 \times 17$ faktorisiert werden kann.

Für den *Agrawal-Kayal-Saxena*-Algorithmus ist die Definition der folgenden beiden Funktionen nötig:

Definition 3. Seien $r \in Nat$, $a \in \mathbb{Z}$, relativ prim ($\gcd(a, r) = 1$), dann ist die "Ordnung von a modulo r" (i.Z. $o_r(a)$) definiert als die kleinste Zahl k mit $a^k \equiv 1 \pmod{n}$.

Definition 4 (Euler's ϕ-Funktion). Sei $r \in \mathbb{N}$, dann ist $\phi(r)$ die Anzahl der Zahlen kleiner als r, die relativ prim zu r sind.

Ein paar Eigenschaften von Euler's ϕ-Funktion:

Korollar 4.1. Ist p prim, dann gilt: $\phi(p) = p - 1$

Korollar 4.2. Für alle $a, r \in \mathbb{N}$ mit $\gcd(a, r) = 1$ gilt: $o_r(a)|\phi(r)$.

Theorem 1. *Sei $n \geq 2, n \in \mathbb{N}$ mit der Primfaktorisierung:*

$$n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$$

Dann gilt:

$$\phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \dots (1 - \frac{1}{p_r}) \tag{2}$$

2.3 Etwas Zahlentheorie

Definition 5. Man sagt: "a teilt b exakt" (i.Z. $a \| b$), gdw. $a|b$ und $\gcd(a, \frac{b}{a}) = 1$.

Korollar 5.1. Seien $q, k, n \in \mathbb{N}$, $k \geq 1$ und gilt $q^k \| n$, dann: $q^{k+1} \not| n$.

Für die Betrachtung der randomisierten Algorithmen zur Primzahlbestimmung sind die folgenden Definitionen hilfreich:

Definition 6 (Jacobi Symbol). Sei n eine ungerade Zahl mit der Primfaktorisierung $p_1^{k_1} p_2^{k_2} \dots p_t^{k_t}$, dann ist für alle a mit $\gcd(a, n) = 1$ das *Jacobi Symbol* definiert als:

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{t} \left(\frac{a}{p_i}\right)^{k_i}$$

2.4 Ringe, Polynome und Polynomringe

Für das Verständnis des *Agrawal-Kayal-Saxena*-Algorithmus ist es sinnvoll, sich die Definitionen von Ringen und Polynomen kurz ins Gedächtnis zu rufen:

Definition 7 (Ring).

1. Eine Menge $R \neq \emptyset$ heißt *(kommutativer) Ring* falls gilt: In R gibt es zwei Verknüpfungen $+$ und $\times$ mit den Eigenschaften:
   (a) $(R, +)$ ist eine abelsche Gruppe mit Neutralelement 0.
   (b) $(R, \times)$ ist eine Halbgruppe.
   (c) $\forall a, b, c \in R : (a + b) \times c = a \times c + b \times c$ (Distributivgesetz)
2. Ein Ring R heißt "Ring mit Eins", falls es ein Element $1 \in R, 1 \neq 0$ gibt, so dass: $\forall a \in R : a \times 1 = 1 \times a = a$
3. Ist $R \backslash 0$ eine Gruppe bezüglich $\times$ mit neutralem Element 1, dann nennt man R einen Körper.

Definition 8 (Polynom). Ein *Polynom* über (einem Ring) R ist definiert als:

$$u(x) = \sum_{i=0}^{n} u_i \times x^i$$

Definition 9 (Polynomring). Sei R ein Ring mit Eins, dann ist ein *Polynomring über* R definiert als:

$$R[X] = \{(a_0, a_1, \dots) : a_i \in R, \exists k \in \mathbb{N} : \forall n > k : a_n = 0\}$$

mit den Verknüpfungen:

- komponentenweise Addition;
- Multiplikation: $(a_0, a_1, \dots) \times (b_0, b_1, \dots) = (c_0, c_1, \dots)$ mit $c_n := \sum_{i=0}^{n} a_i \times b_{n-i}$. (D.h.: $c_0 = a_0 \times b_0$, $c_1 = a_0 \times b_1 + a_1 \times b_0$, $c_2 = a_0 \times b_2 + a_1 \times b_1 + a_2 \times b_0$, $\dots$

Bemerkung. Der $\gcd(f, g)$ für $f, g \in R[X]$ ist definiert als normiertes Polynom, welches f und g teilt (Berechnung mittels "Euklidischem Algorithmus").

3 Komplexität und Turing-Maschine

3.1 Komplexität eines Algorithmus

Um die Effizienz eines Algorithmus zu analysieren oder zu vergleichen, hat sich die sogenannte "O-Notation" eingebürgert. Es beschreibt eine asymptotische obere Grenze für das Wachstum einer Funktion.

Ein Beispiel soll dies verdeutlichen: In der Funktion $T(n) = 4n^2-2n+2$ überwiegt das Wachstum des Terms n^2 für wachsendes $n \to \infty$ die anderen Terme. Repräsentiert die Funktion die Anzahl der Schritte für einen Algorithmus **A**, so wird der Term n^2 der dominante Faktor, da die anderen Terme, abhängig von Hardware und Implementierung des Algorithmus, vernachlässigbar werden.

Um diesen Zusammenhang zu verdeutlichen schreibt man: $T(n) \in \mathrm{O}(n^2)$.

Definition 10 (O-Notation). Sei $f : \mathbb{N} \to \mathbb{N}$, dann ist $f(x) = \mathrm{O}(g(x))$, gdw. $\exists k \in \mathbb{N} : \exists x_0 \in \mathbb{N} : \forall x \geq x_0 : f(x) \leq kg(x)$.

Bemerkung. Die Schreibweise $f = \mathrm{O}(g(n))$ ist daher eigentlich falsch. Korrekt wäre: $f(x) \in \mathrm{O}(g(x))$, denn: $f(x) = \mathrm{O}(g(x)) \Leftrightarrow \mathrm{O}(g(x)) = f(x)$, was aber nicht zur obigen Definition passt.

Korollar 10.1. Gilt $\forall x : g(x) \neq 0$, so ist: $\limsup_{x\to\infty} |\frac{f(x)}{g(x)}| < \infty$

Analog zur O-Notation für eine "asymptotische obere Grenze", die ein Maß für die maximale Laufzeit eines Algorithmus ist, gibt es die Ω-Notation für eine "asymptotische untere Grenze", als Maß für die minimale Laufzeit eines Algorithmus.

Definition 11 (Ω-Notation). Sei $f : \mathbb{N} \to \mathbb{N}$, dann ist $f(x) = \Omega(g(x))$, gdw. $\exists k \in \mathbb{N} : \exists x_0 \in \mathbb{N} : \forall x \geq x_0 : f(x) \geq kg(x)$.

Korollar 11.1. Ebenso: $\forall x : g(x) \neq 0$, so ist: $\limsup_{x\to\infty} |\frac{f(x)}{g(x)}| > 0$

Damit verfügt man über Aussagen über die minimale und die maximale Laufzeit eines Algorithmus **A**, welche aber noch über "viel Spielraum" verfügen können. Daher ist es für manche Algorithmen möglich, "asympototisch enge Grenzen" anzugeben. Dann gilt:

Definition 12 (Θ-Notation). Sei $f : \mathbb{N} \to \mathbb{N}$, dann ist $f(x) \in \Theta(g(n)) :\equiv f \in \mathrm{O}(g)$ und $g \in \mathrm{O}(f)$.

3.2 Abhängigkeit von Hardware und Software

Um zwei Algorithmen **A** und **B** vergleichen zu können, muss man etwas über die Eingabe x, den gewählten Rechner **R**, die Programmiersprache S und die Implementierung I_a bzw. I_b der Algorithmen aussagen. Das Problem dabei ist aber, dass der Rechner, die Programmiersprache und die Implementierung schon nach kurzer Zeit veraltet sein können. Aus diesem Grund wird der Begriff der "Rechenzeit" vergröbert, so dass er nur noch von der Eingabe x und dem jeweiligen Algorithmus abhängt.

Als Rechenmodell ("Rechner") hat sich die sog. "Turing-Maschine" eingebürgert, die auf den englischen Logiker und Mathematiker Alan Turing zurückgeht, der sie 1936 eingeführt hat, um "eine mathematisch präzise Definition eines Algorithmus' " zu bekommen. Eine Turing-Maschine ist ein sehr einfaches, fast primitives Modell, welches aber jedes real existierende Rechnermodell mit nur geringen Rechenzeitverlusten simulieren kann. D.h. für jeden Algorithmus **A**, der auf einer Turing-Maschine in t Rechenschritten ausgeführt werden kann, existiert ein Polynom p, so dass dieser Algorithmus auf einer real existierenden Maschine in höchstens $p(t)$ Rechenschritten ausgeführt werden kann.

3.3 Die Turing-Maschine

Eine Turing-Maschine besteht aus:

1. einem Band, welches in Zellen unterteilt ist. Dies ist quasi der Speicher der Turing-Maschine. Der Einfachheit halber wird angenommen, dass dieser unendlich groß ist und nur Symbole eines endlichen Alphabets enthält.

2. einem Schreib-/Lesekopf, der Symbole vom Band liest oder auf das Band schreibt. Er kann immer nur ein Symbol entweder lesen oder schreiben und sich nur um eine Zelle auf dem Band nach links oder rechts bewegen.

3. einem Zustands-Register, welches den aktuellen Zustand der Turing-Maschine speichert. Es gibt nur endlich viele Zustände und das Register wird zum Start der Turing-Maschine mit einem Startzustand initialisiert.

4. einer Aktionstabelle oder Übergangsfunktion. Diese Tabelle schreibt vor, bei welchem aktuellen Zustand und dem gerade gelesenem Symbol, welches Symbol geschrieben, in welche Richtung der Kopf bewegt und in welchen neuen Zustand die Maschine wechseln soll.

Formal besteht eine Turing-Maschine daher aus einem 5-Tupel $M = (Q, \Gamma, s, F, \delta)$, wobei gilt:

1. Q ist die Menge der Zustände

2. Γ die endliche Menge des Bandalphabets

3. $s \in Q$ der Startzustand

4. $F \subseteq Q$ die Menge der finalen Zustände

5. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ die Übergangsfunktion (quasi das Programm der Turing-Maschine). Sie definiert $(q, X) \mapsto (p, Y, D)$ mit $p, q \in Q$, $X, Y \in \Gamma$ und $D \in \{L, R\}$, d.h. wird in einem Zustand q das Symbol X gelesen, dann versetze die Turing-Maschine in den Zustand p, schreibe das Symbol Y und bewege den Kopf nach *L*inks oder *R*echts.

Es ist offensichtlich, dass Turing-Maschinen primitiv anmuten, dennoch sind sie sehr mächtig. Da man aber meistens bei Turing-Maschinen nur daran interessiert ist, in welcher Komplexitätsklasse ein Algorithmus liegt, reduziert man Probleme für Turing-Maschinen häufig auf die Akzeptanz (oder Nicht-Akzeptanz) einer Sprache L, sogenannte "Entscheidungsprobleme". Man definiert dafür eine "Sprache" L (eine Menge), die korrekte Lösungen für das Problem beschreibt, und eine Turing-Maschine T, welche L "akzeptiert". Für eine Eingabe x gilt dann "T akzeptiert die Eingabe x", gdw. $x \in L$.

Beispiel. Gegeben sei eine Sprache $A = \{0^{2^n} | n \geq 0\}$, also die Sprache, die aus dem Zeichen 0 besteht und deren Zeichenketten die Länge 2^n haben, z.B. $A = \{0, 00, 0000, \dots\}$.

Die Turing-Maschine, welche die Sprache A akzeptiert, ist gegeben durch:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$
- $\Sigma = \{0\}$
- $\Delta = \{0, x, \sqcup\}$
- δ sei beschrieben durch Abbildung 1
- $F = \{q_{\text{accept}}, q_{\text{reject}}\}$
- Der Startzustand sei q_1

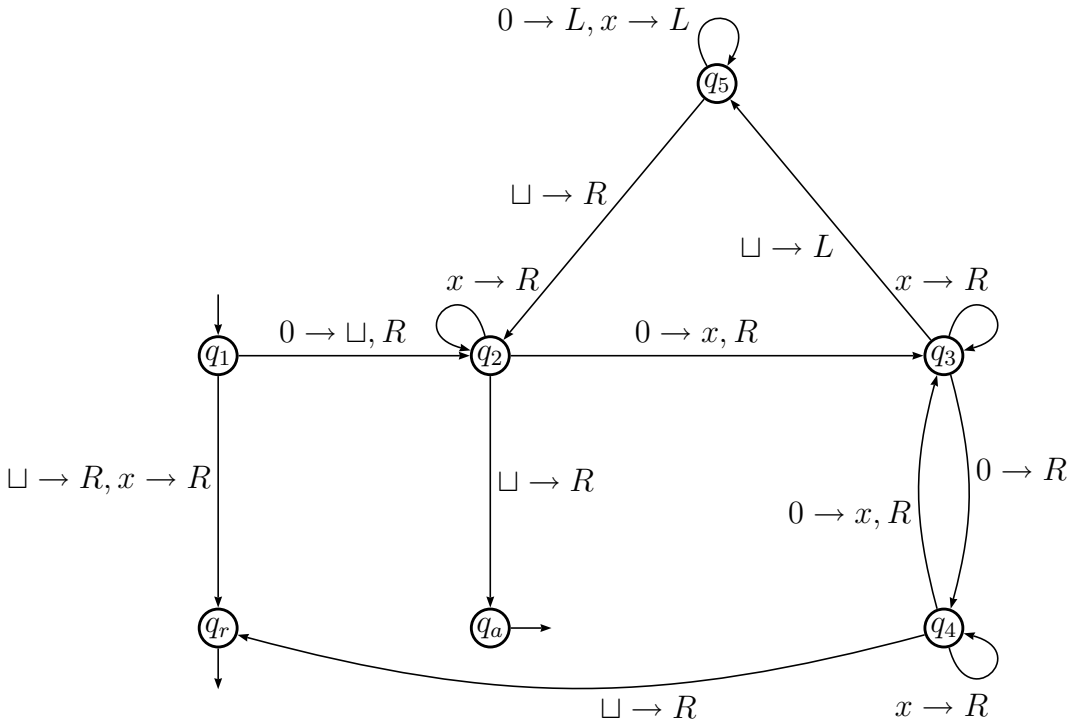
Informell lässt sich der Algorithmus so beschreiben:

1. Bewege den Lesekopf von links nach rechts über das Band, lösche dabei jede zweite 0 und speichere die Anzahl der gelöschten 0.
2. Wenn nur eine 0 gelöscht wurde: "accept".
3. Wenn mehr als eine 0 gelöscht wurde und die Anzahl der gelöschten 0 ungerade ist: "reject".
4. Bewege den Lesekopf wieder zurück an den Anfang des Bandes.
5. Gehe in Zustand 1.

Bemerkung. Die hier beschriebenen Turing-Maschinen sind alle *deterministisch*, d.h. jeder Zustand q ist nur von der Eingabe x (und dem Programm P) abhängig und bei jedem Lauf mit der (gleichen) Eingabe x identisch.

"Nicht-deterministische" Turing-Maschinen unterliegen nicht dieser Einschränkung, sie können (dürfen) in einem Zustand q zwischen mehreren Folge-Zuständen wählen und der gewählte Zustand kann sich bei zwei Läufen mit gleicher Eingabe x deutlich unterscheiden. Man kann daher Nicht-deterministische Turing-Maschinen mit randomisierten Algorithmen vergleichen.

Abbildung 1: Übergangsfunktion für A



3.4 Komplexitätsklassen

Man kann nun jedes (algorithmische) Problem in eine Klasse einsortieren, wobei die wichtigsten sind:

- *P* ist die Klasse der Sprachen, die von einer (deterministischen) Turing-Maschine in polynomieller Zeit akzeptiert werden.
- *NP* ist die Klasse der Sprachen, die von einer nicht-deterministischen Turing-Maschine in polynomieller Zeit akzeptiert werden. (Äquivalent: Eine Lösung zu einem Problem in *NP* kann in polynomieller Zeit verifiziert werden.)

  Beispiel (Teilsummen Problem). Gegeben: Eine Menge $M = \{a_1, a_2, \ldots, a_n\}$ mit $a_i \in \mathbb{N}$ und $s \in \mathbb{N}$. Frage: Gibt es eine Teilmenge $T \subseteq M$ mit $s = \sum T$? Es ist klar, dass eine gegebene Lösung in polynomieller Zeit verifiziert werden kann. Es ist aber kein Algorithmus bekannt, der eine solche Lösung in polynomieller Zeit liefern kann.
- *ZPP* (Zero-error probabilistic polynomial time) ist die Klasse der Sprachen, die von einem randomisierten Algorithmus in polynomieller maximaler Rechenzeit akzeptiert werden oder der mit einer gewissen Wahrscheinlichkeit versagt.

  D.h. entweder liefert der Algorithmus ein korrektes Ergebnis ("Ja"/"Nein") oder er gibt mit einer Wahrscheinlichkeit ϵ ein "Unbekannt" aus.
- *BPP* (Bounded-error probabilistic polynomial time) ist die Klasse der Sprachen, die ein randomisierter Algorithmus in polynomieller maximaler Rechenzeit und einer gewissen Fehlerwahrscheinlichkeit beim Ergebnis akzeptiert.

  D.h. der Algorithmus liefert ein Ergebnis, das mit einer gewissen Wahrscheinlichkeit *epsilon* falsch ist. Diese Algorithmen nennt man auch "Monte-Carlo-Algorithmen".
- *RP* (Random polynomial time) ist die Klasse der Sprachen, für die es einen randomisierten Algorithmus mit polynomieller maximaler Rechenzeit gibt, welcher jede nicht zu akzeptierende Eingabe ablehnt und für jede akzeptierte Eingabe eine beschränkte Fehlerwahrscheinlichkeit hat. (Als *co-RP* bezeichnet man die Klasse der Sprachen, deren Komplement in *RP* ist.)

  D.h. der Algorithmus liefert entweder ein deutliches "Nein" oder mit einer Wahrscheinlichkeit *epsilon* ein "Ja".

Es gilt dann: $P \subseteq ZPP \subseteq RP \subseteq NP$ sowie $P \subseteq ZPP \subseteq co\text{-}RP \subseteq BPP$.

Das grosse, offene Problem in der Komplexitätstheorie ist die Frage, ob $P = NP$ gilt?

3.5 Der Algorithmus von Solovay/Strassen

Gegeben sei eine natürliche Zahl n. Um herauszufinden, ob n prim ist oder nicht, kann man z.B. einfach "Probe-Dividieren", d.h. prüfe: $\forall a \leq n : a|n$. Dieser Algorithmus ist zwar sehr einfach, aber auch sehr langsam: $\Omega(\sqrt{n})$ Schritte sind notwendig.

Momentan basieren die meisten implementierten Verfahren auf "Randomisierten Algorithmen". Einer der bekannteren ist der Algorithmus von *Solovay/Strassen*.

Die Grundidee des Algorithmus von Solovay/Strassen ist es, für eine gegebene (ungerade, natürliche) Zahl n eine Menge W von "Zeugen" für die Annahme "n ist zusammengesetzt" zu bestimmen. Findet man dann ein beliebiges $w \in W$, dann ist $W \neq \emptyset$ und n kann daher nicht prim sein.

Um einen solchen Zeugen zu bestimmen, wählt man ein Element $a \in \mathbb{Z}_n^*$ und prüft, ob $a \in W$. Die Wahrscheinlichkeit ein solches a zu finden, wenn n nicht prim ist, ist $\Pr[.] \geq \frac{1}{2}$. Wiederholt man diesen Prozess k-mal, so verbessert sich die Wahrscheinlichkeit auf $\Pr[.] \geq 1 - (\frac{1}{2})^k$. Findet man kein a, dann ist n prim.

Dem Verfahren liegt die folgende Beobachtung zugrunde:

Beobachtung 1. $n \in \mathbb{P} \Leftrightarrow \left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$

Umgekehrt heißt das, dass wenn n nicht prim ist, es eine große Menge an $a \in \mathbb{Z}_n^*$ gibt, so dass $\left(\frac{a}{n}\right) \not\equiv a^{\frac{n-1}{2}} \pmod{n}$. Irgendein Element dieser Menge zu raten sollte dann möglich sein.

Diese Beobachtung führt zum Algorithmus A.2 von Solovay/Strassen.

Der Algorithmus ist immer korrekt wenn er "COMPOSITE" liefert, da dann ein $a \in \mathbb{Z}_n$ existiert, so dass entweder $\gcd(a, n) \neq 1$ oder $\left(\frac{a}{n}\right) \not\equiv a^{n-\frac{1}{2}} \pmod{n}$ gilt. In beiden Fällen kann n nicht prim sein.

Die Wahrscheinlichkeit, dass der Algorithmus "PRIME" zurückgibt, obwohl n zusammengesetzt ist, ist höchstens $\frac{1}{2}$. Ein Beweis dafür findet sich in [MR00].

4 Der *Agrawal-Kayal-Saxena*-Algorithmus

Eine grundlegende Idee zum Algorithmus von *Agrawal-Kayal-Saxena* ist eine Beobachtung von Fermats kleinem Theorem:

Theorem 2 (Fermats kleines Theorem). *Ist n prim, so gilt für alle $a \in \mathbb{Z}$ mit* $\gcd(a, n) = 1$*:*

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

Für gegebenes a und n könnte man effizient prüfen, ob $a^{n-1} \equiv 1 \pmod{n}$ gilt. Unglücklicherweise erfüllen sowohl viele zusammengesetzte Zahlen als auch die *Carmichael-Zahlen* (siehe die Definition 1) diese Bedingung.

Eine Verallgemeinerung dieses Theorems führt zu:

Lemma 4.1. *Sei $a \in \mathbb{Z}$, $n \in \mathbb{N}$, $n \geq 2$ und* $\gcd(a, n) = 1$*. Dann ist n prim, gdw.:*

$$(X + a)^n \equiv X^n + a \pmod{n} \tag{3}$$

Beweis. Nach dem Binomial-Satz gilt: $(X + a)^n = \sum_{i=0}^{n} \binom{n}{i} X^{n-i} a^i$

Angenommen n ist prim, dann ist $\binom{n}{i} \equiv 0 \pmod{n}$ (siehe B.1) und damit $a^n \equiv a \pmod{n}$.

Damit und B.2 gilt:

$$(X - a)^n = (-a)^n + X^n = X^n - a^n = X^n - a$$

Sei andererseits angenommen, dass n nicht prim ist, dann existiert eine Primzahl q, welche ein Faktor von n ist und damit auch ein $k \geq 1$, so dass gilt: $q^k \| n$.

Da $q|n$ und $\gcd(a, n) = 1$, gilt auch $\gcd(a, q) = 1$ und damit folgt:

$$\gcd(a^{n-q}, q^k) = 1 \tag{4}$$

Es gilt dann:

$$q^k \not| \binom{n}{q} \tag{5}$$

Beweis siehe B.3.

Der Koeffizient x^q in $(X-a)^n$ ist $\binom{n}{q}(-1)^{n-q}a^{n-q}$. Angenommen dieser Koeffizient ist teilbar durch n, dann gilt:

$$\binom{n}{q} a^{n-q} = \alpha n \text{ mit } \alpha \in \mathbb{Z}$$

Damit:

$$\frac{\binom{n}{q} a^{n-q}}{q^k} = \alpha n / q^k$$

Da die rechte Seite der Gleichung eine ganze Zahl ist, muss die linke Seite auch eine ganze Zahl sein. Damit würde Gleichung 4 aber implizieren, dass $\binom{n}{q}$ durch q^k

teilbar ist und damit im Widerspruch zu Gleichung 5 stehen würde. D.h. dass der Koeffizient von x^q in $(X-a)^n \not\equiv 0 \pmod{n}$ und damit $(X-a)^n \neq X^n - a$ im Ring $\mathbb{Z}_n[X]$ ist. □

Dieses Lemma führt dann zum folgenden Algorithmus von Agrawal/Kayal/Saxena.

Algorithm 1 *AKS:* Algorithmus von Agrawal/Kayal/Saxena

Require: $n \in \mathbb{N}, n > 1$
Ensure: PRIME, COMPOSITE
1: **if** $n = a^b, a \in \mathbb{N}, b > 1$ **then**
2: COMPOSITE
3: **end if**
4: Finde kleinstes r mit: $o_r(n) > 4\log^2(n)$
5: **if** $1 < \gcd(a, n) < n$ für ein $a \leq r$ **then**
6: COMPOSITE
7: **end if**
8: **if** $n \leq r$ **then**
9: PRIME
10: **end if**
11: **for** $a \leftarrow 1 \ldots \lfloor 2\sqrt{\phi(r)\log(n)} \rfloor$ **do**
12: **if** $(X+a)^n \neq X^n + a \pmod{X^r - 1, n}$ **then**
13: COMPOSITE
14: **end if**
15: **end for**
16: PRIME

Theorem 3. *Der Algorithmus gibt **PRIME** zurück, gdw. n ist prim.*

Der Beweis der einen Richtung des Theorems ist einfach:

Lemma 4.2. *Ist n prim, dann gibt der Algorithmus **PRIME** zurück.*

Beweis. Falls n prim ist, können Schritte 1 und 5 nicht "**COMPOSITE**" zurückgeben. Wegen Lemma 4.1 kann die `for`-Schleife ebenfalls nicht "**COMPOSITE**" zurückgeben. Daher wird der Algorithmus n als Primzahl entweder in Schritt 8 oder in Schritt 16 identifiziert. □

Die umgekehrte Richtung des Lemmas 4.2 ist leider nicht ganz so einfach: Sollte der Algorithmus "**PRIME**" in Schritt 4 zurückgeben, dann muss n prim, andernfalls hätte der Algorithmus in Schritt 3 einen nicht-trivialen Faktor von n gefunden. Daher muß der Algorithmus von Schritt 6 "**PRIME**" zurückgegeben haben. Der Beweis dafür wird im Vortrag von Wolfgang Keller gegeben.

5 Aufwandsanalyse

Die Rechenzeit des *Agrawal-Kayal-Saxena*-Algorithmus nachzuweisen ist einfach. Die Rechenoperationen von m-Bit langen Zahlen brauchen lediglich $\mathrm{O}(\log_2(m))$ Schritte; siehe z.B. [Knu98b]. Die gleichen Rechenoperationen auf Polynomen vom Grad d mit m-Bit Koeffizient brauchen ebenfalls $\mathrm{O}(\log_2(dm))$ Schritte.

Theorem 4. *Die asymptotische Zeitkomplexität von Algorithmus A.3 ist:*

$$O(\log_{19}(n))$$

Beweis. Sämtliche Operationen vor der `for`-Schleife in Schritt 17 lassen sich mit geringerem Aufwand erzielen:

Die Entscheidung in Schritt 1, ob n eine sog. "perfekte Potenz" ist, läßt sich in $\mathrm{O}(\log_4(n)\log\log(n))$ entscheiden (siehe [Smi03] für eine detailierte Analyse der benötigten Algorithmen).

Die `while`-Schleife braucht den Aufwand: $\mathrm{O}(\log_9(n)(\log\log(n))^2)$.

Für r gilt: $r = \mathrm{O}(\log_6(n))$.

Proposition 5.1. *Es existieren Konstanten $c_2 > c_1 > 0$, so dass für genügend großes n, eine Primzahl r existiert, mit:*

- $c_1 \log_6(n) < r \leq c_2 \log_6(n)$
- $r-1$ *hat einen Primfaktor q mit* $q \geq 2\sqrt{r}\log(n) + 2$
- $n^{(r-1)/q} \not\equiv 1 \pmod{r}$

Beweis. Siehe [Smi03]. □

Die ϕ-Funktion benutzt die gcd-Funktion, welche sich in $\mathrm{O}(\log_3(n))$ Schritten berechnen lässt.

Die Bestimmung, ob r eine Primzahl ist, braucht $\mathrm{O}(\sqrt{r}\log_2(r))$ Schritte.

Die Berechnung des größten Primfaktors q von $r-1$, lässt sich in $\mathrm{O}(\sqrt{r}\log_2(r))$ durchführen.

Die Berechnung von $n^{(r-1)/q} \pmod{r}$ braucht den Aufwand: $\mathrm{O}(\log_2(n)+\log_3(r))$.

Eine Iteration der Schleife braucht daher $\mathrm{O}(\log_3(n) + \sqrt{r}\log_2(r))$ Schritte. Da die Schleife $\mathrm{O}(\log_6(n))$-mal durchlaufen wird, folgt: $\mathrm{O}(\log_9(n) + \sqrt{r}\log_2(r)\log_6(n))$. Da $r = \mathrm{O}(\log_6(n))$ ist, ergibt sich also: $\mathrm{O}(\log_9(n)(\log\log(n))^2)$.

Bleibt die `for`-Schleife in Schritt 17: Eine Iteration der Schleife benötigt $\mathrm{O}(r^2\log_3(n))$ Zeit, um die Koeffizienten des Polynoms $(X-a)^n \pmod{X^r-1}$ in $\mathbb{Z}_n[X]$ zu berechnen. Das Polynom $(X^n - a) \pmod{X^r-1}$ kann mit demselben Aufwand berechnet werden. Damit braucht die Schleife für einen Durchlauf $\mathrm{O}(r^2\sqrt{r}\log_4(n))$. Da $r = \mathrm{O}(\log_6(n))$, ergibt sich $\mathrm{O}(\log_{19}(n))$. □

6 Schlussbetrachtung

Der “Bedarf” an Primzahlen ist ziemlich groß, da alle Public-Key-Cryptoverfahren große Primzahlen benötigen. Public-Key-Cryptoverfahren sind im Zeitalter des Internets sowohl zum geheimen Austausch von Nachrichten, als auch zum sicheren Übertragen von persönlichen Daten, wie z.B. Kreditkarten-Daten, notwendig.

Das Verlangen nach einem effizienten Algorithmus zur Bestimmung von (großen) Primzahlen ist also vorhanden.

Dennoch ist das Ergebnis von M. Agrawal, N. Kayal und N. Saxena in der Praxis nur bedingt anwendbar, da der Algorithmus selbst zwar in polynomieller Zeit läuft, aber diese Zeit (absolut) dennoch viel größer ist, als die Zeit, die z.B. der Solovay/Strassen-Algorithmus benötigt.

A Algorithmen

A.1 Euklidischer Algorithmus.

Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler zweier ganzer Zahlen. Die Laufzeitkomplexität ist $O(n)$, welcher im schlimmsten Fall eintritt, wenn die Eingabe aus zwei aufeinander folgenden Fibonacci-Zahlen besteht.

Algorithm 2 *Euklid:* Euklidischer Algorithmus

Require: $a, b \in \mathbb{Z}$
Ensure: $\gcd(a, b)$
1: **while** $b \neq 0$ **do**
2: $temp \leftarrow a \mod b$
3: $a \leftarrow b$
4: $b \leftarrow temp$
5: **end while**
6: return a

A.2 Algorithmus von Solovay/Strassen

Die Grundidee des Algorithmus von Solovay/Strassen ist es, für eine gegebene (ungerade, natürliche) Zahl n eine Menge W von "Zeugen" für die Annahme "n ist zusammengesetzt" zu bestimmen. Findet man dann ein bel. $w \in W$, dann ist $W \neq \emptyset$ und n ist nicht prim.

Um einen solchen Zeugen zu bestimmen, wählt man ein Element $a \in \mathbb{Z}_n^*$ und prüft, ob $a \in W$. Die Wahrscheinlichkeit ein solches a zu finden, wenn n nicht prim ist, ist $>= 1/2$. Wiederholt man diesen Prozess k-mal, so verbessert sich die Wahrscheinlichkeit auf $>= 1 - 1/2^k$. Findet man kein a, dann ist n prim.

Algorithm 3 *RandPrim1:* Algorithmus von Solovay/Strassen

Require: $n \in \mathbb{N}, n$ ungerade
Ensure: PRIME, COMPOSITE
1: $a \leftarrow$ Gleichförmig verteilte Zufallszahl aus $\mathbb{Z}_n \backslash 0$
2: $g \leftarrow \gcd(a, n)$
3: **if** $g \neq 1$ **then**
4: COMPOSITE
5: **else**
6: $j \leftarrow \left(\frac{a}{n}\right)$
7: $m \leftarrow a^{n-\frac{1}{2}} \pmod n$
8: **if** $j \equiv m$ **then**
9: PRIME
10: **else**
11: COMPOSITE
12: **end if**
13: **end if**

A.3 Der Algorithmus von Agrawal, Kayal und Saxena

Der folgende Algorithmus ist aus [AKS04]:

Algorithm 4 *AKS:* Algorithmus von Agrawal/Kayal/Saxena

Require: $n \in \mathbb{N}, n > 1$
Ensure: PRIME, COMPOSITE
1: **if** $(n = a^b, a \in \mathbb{N}, b > 1)$ **then**
2: COMPOSITE
3: **end if**
4: Finde kleinstes r mit: $o_r(n) > 4\log^2(n)$
5: **if** $(1 < \gcd(a, n) < n$ für $a \leq r)$ **then**
6: COMPOSITE
7: **end if**
8: **if** $(n \leq r)$ **then**
9: PRIME
10: **end if**
11: **for** $a \leftarrow 1 \ldots \lfloor \sqrt{\phi(r)} \log(n) \rfloor$ **do**
12: **if** $((X + a)^n \neq (X^n + a) \pmod{X^r - 1, n})$ **then**
13: COMPOSITE
14: **end if**
15: **end for**
16: PRIME

B Lemmata und Beweise

Lemma B.1. *Sei p eine Primzahl und $1 \leq i \leq p-1$, dann ist:*

$$\binom{n}{i} \equiv 0 \pmod{n}$$

Beweis. Da $\binom{p}{i}$ eine ganze Zahl ist,

$$\binom{n}{i} = \frac{n(n-1)(n-2)\dots(n-i+1)}{i!} \equiv 0 \pmod{n}$$

gilt und da $\gcd(n, i!) = 1$, folgt:

$$\frac{(n-1)(n-2)\dots(n-i+1)}{i!} \equiv 0 \pmod{n}$$

ist eine ganze Zahl. □

Lemma B.2. *Sei p eine Primzahl, dann gilt: $a^p \equiv a \pmod{p}$ für alle ganze Zahlen a.*

Beweis. Da $(-a)^p \equiv -a^p \pmod{p}$ für alle ganze Zahlen a, reicht es den nicht-negativen Fall zu betrachten. Der Beweis funktioniert per Induktion über a:

$a = 0$ Klar.

$a \geq 0$ Gelte die Behauptung für $a^p \equiv a \pmod{p}$. Durch Newtons Binomial Theorem gilt:

$$(a+1)^p = \sum_{i=0}^{p} \binom{p}{i} a^i$$

Durch Arithmetik in $\mathbb{Z}_p$ und Lemma B.1 gilt:

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

□

Proposition B.3. *Seien $q, k, n \in \mathbb{N}$, $k \geq 1$, n nicht prim und $q \| n$, dann gilt:*

$$q^k \not| \binom{n}{q}$$

Beweis. (Beweis durch Wiederspruch) Angenommen $q^k | \binom{n}{q}$. Dann $\binom{n}{q} = \alpha q^k$ für $\alpha \in \mathbb{N}$, d.h.

$$\frac{n(n-1)(n-2)\dots(n-q+1)}{q!} = \alpha q^k$$

Umformen führt zu:

$$n = \frac{\alpha(q-1)!q^{k+1}}{(n-1)(n-2)\dots(n-q+1)}$$

(Die rechte Seite ist eine ganze Zahl!) Sei $1 \leq j \leq q-1$ und sei angenommen, dass $q|(n-j)$, dann $(n-j) \equiv 0 \pmod{q}$. Da $n \equiv 0 \pmod{q}$, folgt $j \equiv 0 \pmod{q}$, was aber nicht sein kann. Damit gilt: $q \not| (n-j)$ für $1 \leq j \leq q-1$ und da q eine Primzahl ist, ist

$$\frac{\alpha(q-1)!}{(n-1)(n-2)\dots(n-q+1)}$$

eine ganze Zahl. Das impliziert aber $q^{k+1}|n$, was zum Wiederspruch führt. □

C Literatur und Quellen

Literatur

[AKS02] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. `http://www.cse.iitk.ac.in/users/manindra/index.html`, 2002.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[Buc00] Johannes Buchmann. *Einführung in die Kryptographie*. Springer Verlag, 2000.

[Cal] Chris Caldwell. The Prime Pages. `http://www.utm.edu/research/primes/prove/prove4_3.html`.

[DK02] Hans Delfs and Helmut Knebl. *Introduction to Cryptography*. Springer Verlag, 2002.

[EMU01] John E.Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theroy, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.

[Knu98a] Donald E. Knuth. *The Art of Computer Programming, Fundamental Algorithms*, volume 1. Addison Wesley, 2nd edition, 1998.

[Knu98b] Donald E. Knuth. *The Art of Computer Programming, Seminumerical Algorithms*, volume 2, chapter 4.3 Multiple-Precision Arithmetic, pages 265–318. Addison Wesley, 3rd edition, 1998.

[Knu98c] Donald E. Knuth. *The Art of Computer Programming, Seminumerical Algorithms*, volume 2. Addison Wesley, 3rd edition, 1998.

[KS98] H. Kurzweil and B. Stellmacher. *Theorie der endlichen Gruppen*. Springer Verlag, 1998.

[MR00] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*, chapter 14.6 Primality Testing. Cambridge University Press, 2000.

[Sip01] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 2001.

[Smi03] Michiel Smid. Primality testing in polynomial time. `Unknown`, May 2003.

[Weg03] Ingo Wegener. *Komplexitätstheorie*. Springer Verlag, 2003.