# 1 Primzahltest

## 1.1 Motivation

Primzahlen spielen bei zahlreichen Algorithmen, die Methoden aus der Zahlentheorie verwenden, eine zentrale Rolle. Hierzu zählen viele Anwendungen im Bereich der Kryptographie , wie z. B. das bekannte RSA-Verfahren, das unter anderem im Netscape Communicator oder im Programmpaket PGP (pretty good privacy) benutzt wird. Aber auch in anderen Bereichen werden Primzahlen benötigt, beispielsweise bei der Erzeugung von Hash-Funktionen für schnelle Datenstrukturen.

Deshalb interessiert man sich für einen Algorithmus, der folgende Frage beantwortet:

> Gegeben sei eine Zahl $n$. Ist $n$ eine Primzahl?

Leider existiert bis heute kein effizientes deterministisches Verfahren zur Lösung dieses Problems. In der Praxis kommen daher randomisierte Verfahren zum Einsatz. Eines davon werden wir im Folgenden kennenlernen.

Zunächst noch ein paar Worte zu den deterministischen Verfahren: Eine naheliegende Lösung besteht darin, für alle Zahlen $t \in \{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ zu testen, ob $t \mid n$ (sprich: $t$ teilt $n$). Dieser Algorithmus besitzt die Laufzeit $\mathcal{O}(\sqrt{n})$. Nicht schlecht, oder? Das Problem hierbei liegt darin, daß die Zahl $n$ im allgemeinen sehr groß ist. Codiert man sie, wie auf Rechnern üblich, im Binärsystem, dann beträgt die Eingabelänge $l = \log_2 n$. Als Laufzeit erhalten wir $\mathcal{O}(\sqrt{2^l}) = \mathcal{O}(2^{l/2})$, das Verfahren benötigt also exponentiellen Aufwand in der Eingabelänge und ist deshalb für große Zahlen $n$ absolut unbrauchbar[1].

Für das RSA-Verfahren wird gegenwärtig empfohlen, Schlüssel mit mindestens 768 Bits zu verwenden. Dazu sind Primzahlen mit etwa 384 Bits nötig. Grundsätzlich gilt: Je länger die Primzahlen, desto größer wird der Aufwand zum Brechen des Codes. Für solche Anwendungen benötigen wir unbedingt ein Verfahren, das polynomiell in $l = \log n$ ist.

## 1.2 Zahlentheoretische Grundlagen

Der vorgestellte Primzahltest verwendet grundlegende Aussagen und Terminologie aus der Zahlentheorie, wie sie z. B. in der Vorlesung Diskrete Strukturen I behandelt wird. Wir zitieren deshalb nur kurz die nötigen Theoreme und verweisen für eine genauere Behandlung auf DS I.

**Definition 1** $\langle G, \circ, e\rangle$ *heißt* Gruppe*, falls gilt*

- $\circ$ *ist eine Abbildung* $G \times G \to G$
- $\circ$ *ist assoziativ, d.h.*

$$(a \circ b) \circ c = a \circ (b \circ c) \qquad \forall a, b, c \in G$$

[1] Dies gilt ebenfalls für das Sieb des Erathosthenes, das dem einen oder anderen vielleicht aus der Schule bekannt ist.

- *e ist ein (linksseitiges) Einselement, d.h.*

$$e \circ a = a \qquad \forall a \in G$$

- *zu jedem Element a von G existiert ein Inverses, d.h.*

$$\forall a \in G \quad \exists a^{-1} \in G \quad mit \quad a \circ a^{-1} = e$$

Wir werden im folgenden die Gruppen $(\mathbb{Z}_n, +)$ und $(\mathbb{Z}_n^*, \cdot)$ betrachten, wobei Addition bzw. Multiplikation modulo $n$ durchgeführt werden. Zur Erinnerung: $\mathbb{Z}_n := \{0, 1, 2, \ldots, n-1\}$ und $\mathbb{Z}_n^* := \{x \in \mathbb{Z}_n \setminus \{0\} \mid \text{ggT}(x, n) = 1\}$ (Man kann hierbei nicht alle Elemente aus $\mathbb{Z}_n \setminus \{0\}$ verwenden, da für Elemente $x$ mit $\text{ggT}(x, n) \neq 1$ kein multiplikatives Inverses existiert).

In der Zahlentheorie spielt der *größte gemeinsame Teiler* (greatest common divisor, ggT) eine wichtige Rolle. Bei der Berechnung des ggT hilft folgendes

**Lemma 2** *Für ganze Zahlen $a \geq 0$ und $b > 0$ gilt*

$$ggT(a, b) = ggT(b, a \bmod b)$$

Darauf baut Algorithmus 1.1 auf, der schon in der Antike von Euklid formuliert wurde. Dieser Algorithmus berechnet $\text{ggT}(a, b)$ und dürfte vom Schulunterricht bekannt sein.

**Algorithmus 1.1** Euklidischer Algorithmus

**func** euclid($a$, $b$)

```
if (b == 0) then
    return a;
else
    return euclid(b, a mod b);
fi
```

Man kann den Euklidischen Algorithmus noch ein wenig erweitern und folgende Aussage zeigen:

**Satz 3** (Erweiterter Euklidischer Algorithmus) *Zu gegebenen $a, b \in \mathbb{N}$ können $x, y \in \mathbb{Z}$ bestimmt werden mit $ax + by = ggT(a, b) =: d$. Die Laufzeit des Algorithmus ist polynomiell in $\log(a + b)$.*

Algorithmus 1.2 zeigt das Vorgehen im Detail. Die Rückgabewerte der Funktion extendedEuclid($a$, $b$) entsprechen $(d, x, y)$ aus Satz 3. Die Korrektheit und die Laufzeit der Algorithmen 1.1 und 1.2 zeigt man jeweils durch Induktion über die Rekursionstiefe.

Diese Funktion kann man zur Berechnung des Inversen einer Zahl $a$ modulo $b$ verwenden. Dazu ruft man extendedEuclid($a$, $b$) auf. Das Inverse existiert für $\text{ggT}(a, b) = 1$. Somit gilt $1 = ax + by$ und deshalb $ax = 1 \bmod b$. $x$ ist also das multiplikative Inverse zu $a$ modulo $b$.

**Algorithmus 1.2** Erweiterter Euklidischer Algorithmus

**func** extendedEuclid($a$, $b$)

```
if (b == 0) then
    return (a, 1, 0);
else
    (d', x', y') := extendedEuclid(b, a mod b);
    (d, x, y) := (d', y', x' − ⌊a/b⌋ · y');
    return (d, x, y);
fi
```

## 1.3 Formulierung des Algorithmus (Idee der witnesses)

Das folgende Theorem ist das Fundament, auf das wir unseren Primzahltest aufbauen werden.

**Satz 4** (Satz von Fermat) *Für alle $n \in \mathbb{N}$ gilt:*

$$n \text{ Primzahl} \quad \Longleftrightarrow \quad a^{n-1} \equiv 1 \bmod n \quad \forall a \in \mathbb{Z}_n \setminus \{0\}.$$

Satz 4 besagt, daß es ein $a \in \mathbb{Z}_n \setminus \{0\}$ gibt mit $a^{n-1} \not\equiv 1 \bmod n$, gdw. $n$ eine zusammengesetzte Zahl ist. Wir könnten also durch Überprüfung dieser Eigenschaft für alle $a \in \mathbb{Z}_n \setminus \{0\}$ einen (deterministischen) Primzahltest konstruieren. Leider ist dieses Verfahren alles andere als effizient.

Die entscheidende Idee zur Lösung des Problems besteht nun darin, nicht *alle* $a \in \mathbb{Z}_n \setminus \{0\}$ zu überprüfen, sondern nur *wenige zufällig ausgewählte*. Ein $a$ mit $a^{n-1} \not\equiv 1 \bmod n$ nennen wir Zeuge (engl.: witness) für die Eigenschaft „$n$ ist zusammengesetzt“. Die Hoffnung besteht nun darin, daß es für zusammengesetzte $n$ ausreichend viele witnesses gibt, so daß wir mit genügend hoher Wahrscheinlichkeit bei der zufälligen Auswahl auf ein geeignetes $a$ stoßen.

Algorithmus 1.3 zeigt das Vorgehen im Detail.

- Test 1 erledigt triviale Randfälle.
- Test 2 setzt genau die Idee der witnesses um (nach dem Satz von Fermat).
- Test 3 dient zum Aufspüren von *Carmichael-Zahlen*. Diese haben die unangenehme Eigenschaft, daß sie sich bezüglich des Fermatschen Satzes (fast) wie Primzahlen verhalten. Es gilt

$$n \text{ Carmichael-Zahl} \implies a^{n-1} \equiv 1 \bmod n \quad \text{für alle } a \in \mathbb{Z}_n^*$$

Die Korrektheit des Algorithmus sieht man relativ leicht ein: Wenn $n$ prim ist, so liefert Algorithmus 1.3 das Resultat PRIME zurück, da keine Testbedingung erfüllt sein kann. Umgekehrt gilt: Wenn die Antwort COMPOSITE lautet, so ist $n$ sicherlich zusammengesetzt. Also:

$$\Pr[\text{Alg. liefert PRIME} \mid n \text{ ist prim}] = 1$$

Das Ergebnis PRIME kann man jedoch vorerst nur als „$n$ könnte prim sein“ interpretieren. Es läßt sich aber nachweisen, daß man sogar “$n$ ist wahrscheinlich prim“ folgern kann. Wir zitieren dies ohne Beweis:

**Lemma 5**

$$\Pr[\textit{Alg. liefert } \textsc{Prime} \mid n \textit{ ist zusammengesetzt}] \leq \frac{1}{2}$$

Man sagt, der Algorithmus macht einen einseitigen Fehler mit Wahrscheinlichkeit $\frac{1}{2}$. Solche Algorithmen, die mit gewisser Wahrscheinlichkeit Fehler machen, nennt man *Monte-Carlo-Algorithmen*. Durch $t$-malige Wiederholung des Tests können wir die Fehlerwahrscheinlichkeit auf $\frac{1}{2^t}$ reduzieren, da wir nur dann ein falsches Ergebnis akzeptieren, wenn *alle* (unabhängigen) Durchläufe die falsche Antwort liefern.

Alle Teilschritte von Algorithmus 1.3 und damit die gesamte Laufzeit sind polynomiell in $\log n$: Für die Berechnung von $\mathrm{ggT}(x)$ verwenden wir den Euklidischen Algorithmus. Der einzige verbleibende Teilschritt, dessen Laufzeit problematisch sein könnte ist die Berechnung von $a^{n-1} \bmod n$ oder allgemein $a^k \bmod n$. Diese läßt sich effizient realisieren, indem wir ausgehend von der Binärdarstellung von $k$ wiederholt Quadrieren und unmittelbar danach den Modulus berechnen. Wir zeigen dies an einem Beispiel: $a = 43$, $k = 50$ und $n = 67$. Im Binärsystem gilt $k = (110010)_2$. Wir berechnen $43^{50} \bmod 67$ durch

$$\begin{aligned} 43^{50} &\equiv 43^{2^5} \cdot 43^{2^4} \cdot 1^{2^3} \cdot 1^{2^2} \cdot 43^{2^1} \cdot 1^{2^0} \\ &\equiv (((((43^2) \cdot 43)^2 \cdot 1)^2 \cdot 1)^2 \cdot 43)^2 \cdot 1 \mod 67 \end{aligned}$$

Bei einer effizienten Implementierung dieses Rechenschemas führt man nach jeder Operation (Quadrieren bzw. Multiplikation) eine Berechnung des Modulus aus. Dies ändert nichts am Ergebnis, da wir ohnehin in $\mathbb{Z}_n$ rechnen, und wir können so garantieren, daß die Zahlen die Länge $\mathcal{O}(\log n)$ nicht überschreiten. Beispielsweise wird $10^7 \bmod 3$ durch

$$10^7 \bmod 3 = ((10^2 \bmod 3) \cdot 10 \bmod 3)^2 \bmod 3 \cdot 10 \bmod 3$$

berechnet. Insgesamt sind für die *Exponentiation durch sukzessives Quadrieren* nur $\mathcal{O}(\log k)$ Multiplikationen und Modulo-Berechnungen nötig mit Zahlen, die höchstens $\mathcal{O}(\log n)$ Bits umfassen. Damit erhalten wir die gewünschte Laufzeit.

**Lemma 6** *Die Laufzeit von Algorithmus „primalityTest“ ist polynomiell in* $\log n$.

## 1.4 Implementierung mit LEDA

In LEDA sind ganze Zahlen beliebiger Länge als Typ `integer` verfügbar. Für diese Zahlen werden auch Ein-/Ausgabe-Operationen (über LEDA-Streams) und grundlegende arithmetische Operationen angeboten.

**Algorithmus 1.3** Ein randomisierter Primzahltest

**func** primalityTest($n$)

```
Wähle a ∈ {2,...,n−1} zufällig;
// Test 1
if ggT(a,n) ≠ 1 ∨ n gerade then
    // trivialer Fall: Es gibt einen gemeinsamen Teiler von a und n oder n gerade
    return COMPOSITE
// Test 2
else if a^{n−1} ≢ 1 mod n then
    // a ist ein Zeuge für einen „Verstoß“ gegen den Satz von Fermat
    return COMPOSITE
else
    // Test 3
    // Zusatzbehandlung für die Carmichael-Zahlen:
    for all d ≥ 1 mit 2^d | n − 1 do
        // Betrachte geeignete zusätzliche Zeugen a^{(n−1)/2^d}
        if 1 < ggT(a^{(n−1)/2^d} − 1, n) < n then
            // Nicht-trivialer Teiler von n gefunden
            return COMPOSITE
        fi
    od
    // Kein Zeuge gefunden: Na gut, wir glauben, daß n prim ist.
    return PRIME
fi
```

# 2 Pattern Matching mit Fingerprints

## 2.1 Problemstellung und Lösungsidee

Ein sehr grundlegendes Problem der Informatik besteht im Suchen einer Zeichenkette $P = p_1 p_2 \dots p_m$ (Pattern) in einem langen Text $T = t_1 t_2 \dots t_n$. Wir stellen einen Lösungsansatz hierfür vor, der randomisierte, d. h. zufallsgesteuerte, Techniken einsetzt und Primzahlen verwendet. Im folgenden betrachten wir der Einfachheit halber Zeichenketten über dem Alphabet $\{0, 1\}$. Die Überlegungen lassen sich aber direkt auf allgemeine Alphabete verallgemeinern.

Eine Zeichenkette $X = x_1 x_2 \dots x_k$ aus $\{0, 1\}^k$ kann auf natürliche Weise mit einer ganzen Zahl identifiziert werden:

$$N(X) := \sum_{i=1}^{k} x_i \cdot 2^{i-1}.$$

Wir ordnen also beispielsweise der Zeichenkette 110 die Zahl 3 zu. Da die so definierte Umwandlung einer Zeichenkette in eine Zahl eindeutig ist, sind zwei Zeichenketten $X$ und $Y$ genau dann gleich, wenn $N(X) = N(Y)$.

Um das Problem des Pattern Matching zu lösen, kann man somit zu jeder Stelle $r$ des Textes die entsprechende Zahl $N_r(T) := N(t_r t_{r+1} \dots t_{r+m-1})$ berechnen und mit $N(P)$ vergleichen. Allerdings ist dieser Algorithmus nicht besonders effizient, da die verwendeten Zahlen sehr groß werden. Deshalb muß man bei der Analyse der Laufzeit davon ausgehen, daß arithmetische Operationen auf diesen Zahlen einen zur Anzahl der Ziffern proportionalen Zeitbedarf besitzen. Da alle $n$ Stellen des Textes untersucht werden müssen, erhalten wir somit Laufzeit $\Omega(nm)$ und sind damit höchstens so gut wie der triviale Algorithmus, der das Pattern nach und nach mit allen Textstellen vergleicht. Da man Pattern Matching bekanntlich in Zeit $\mathcal{O}(n+m)$ lösen kann (z. B. durch die Algorithmen von Boyer-Moore oder Knuth-Morris-Pratt), können wir uns damit nicht zufriedengeben.

Die entscheidende Idee besteht nun darin, die Berechnungen modulo einer geeigneten (nicht zu großen) Primzahl $p$ durchzuführen. Wenn an Stelle $r$ ein Match vorliegt und somit $N_r(T) = N(P)$ gilt, so folgt auch $N_r(T) \bmod p = N(P) \bmod p$. Matches werden also durch dieses Verfahren sicher erkannt.

Allerdings kann $N_r(T) \bmod p = N(P) \bmod p$ auch dann gelten, wenn $N_r(T) \neq N(P)$. In diesem Fall spricht man von einem *false match*, d. h. der Algorithmus erkennt fälschlicherweise ein Match obwohl ein Mismatch vorliegt. Wenn man jedoch die Primzahl $p$ aus einem Intervall $[2, \dots, \tau]$ wählt für $\tau$ hinreichend groß, so kann man zeigen, daß die Wahrscheinlichkeit für ein false match sehr klein wird.

## 2.2 Wahrscheinlichkeit eines false match

Zur Analyse der Fehlerwahrscheinlichkeit benötigen wir folgendes Lemma.

**Lemma 7** *Die Anzahl verschiedener Primfaktoren einer Zahl $x \leq 2^k$ ist kleiner gleich $k$.*

**Beweis:** Alle Primzahlen sind $\geq 2$. Eine Zahl mit mehr als $k$ Primfaktoren ist somit $> 2^k$. □

Sei $\pi(\tau)$ die Anzahl von Primzahlen $\leq \tau$. $\pi(\tau)$ ist aus der Zahlentheorie recht genau bekannt. Wir werden verwenden, daß $\pi(\tau) = \Theta(\tau / \ln \tau)$.

Wir schätzen nun die Wahrscheinlichkeit ab, daß an Stelle $r$ ein false Match auftritt. In diesem Fall gilt $N_r(T) \bmod p = N(P) \bmod p$ bzw. $\delta := N_r(T) - N(P) \equiv 0 \bmod p$. Da $\delta$ höchstens $m$ Stellen hat, besitzt $\delta$ maximal $m$ Primfaktoren. Daraus folgt

$$\Pr[\text{false match an Stelle } r] \leq \frac{m}{\pi(\tau)} = \mathcal{O}\left(\frac{m \log \tau}{\tau}\right).$$

Setzt man $\tau = n^2 m \log(n^2 m)$ so gilt

$$\Pr[\text{false match an Stelle } r] = \mathcal{O}\left(\frac{1}{n^2}\right).$$

Daraus folgt sofort, daß

$$\Pr[\text{false match (irgendwo)}] \leq n \cdot \Pr[\text{false match an Stelle } r] = \mathcal{O}\left(\frac{1}{n}\right).$$

Die Wahrscheinlichkeit für ein false match geht also gegen Null.

## 2.3 Laufzeit und Korrektheit

Zur Analyse der Laufzeit überlegt man sich leicht, daß es möglich ist, $N_{r+1}(T)$ aus $N_r(T)$ mittels $\mathcal{O}(1)$ arithmetischer Operationen zu berechnen. Wenn alle Rechenoperationen modulo $p$ durchgeführt werden, kann man erreichen, daß die Zwischenergebnisse höchstens $\log p$ Stellen besitzen. Unter der Annahme das arithmetische Operationen auf so kurzen Zahlen in Zeit $\mathcal{O}(1)$ durchgeführt werden können, ergibt sich somit eine Gesamtlaufzeit von $\mathcal{O}(n+m)$. Bei der Durchführung von Divisionen modulo 256 muß man wie oben erwähnt das multiplikative Inverse mit Hilfe des erweiterten Euklidischen Algorithmus berechnen.

False matches sind in der Praxis natürlich sehr lästig (auch wenn sie nur selten auftreten). Man kann sie ganz ausschalten, indem man nachträglich alle vermeintlichen Matches überprüft. Bei geschickter Implementierung verlängert sich dadurch die erwartete Gesamtlaufzeit asymptotisch nicht. Wir zeigen dies für den (relativ einfachen) Fall, daß nur höchstens ein Match auftreten kann. Wenn kein false match auftritt, benötigen wir Laufzeit $\mathcal{O}(n+m)$. Andernfalls schätzen wir die Laufzeit großzügig mit $\mathcal{O}(nm)$ ab. Im Erwartungswert erhalten wir somit eine Laufzeit von $\mathcal{O}((1-1/n)(n+m)+(1/n)nm) = \mathcal{O}(n+m)$.

## 2.4 Generierung zufälliger Primzahlen

Zur Generierung der zufälligen Primzahl $p \in [2,k]$ genügt folgende einfache Strategie: Man erzeugt hierzu eine Zufallszahl der gewünschten Größe und testet, ob sie prim ist. Wenn nein, so wiederholt man das Verfahren, bis man Erfolg hat.

Wie zuvor erwähnt, gibt es zwischen 1 und $k$ ungefähr $k/\log(k)$ Primzahlen. Also hat man bei jedem Versuch eine Erfolgswahrscheinlichkeit von etwa $1/\log(k)$, wenn man jede Zahl zwischen 1 und $k$ gleichwahrscheinlich wählt[2], und ist somit im Mittel nach $\log(k)$ Iterationen fertig.

[2] Allerdings ist es nicht besonders sinnvoll, gerade Zahlen zu erzeugen ...