1 Primzahltest

1.1 Motivation

Primzahlen spielen bei zahlreichen Algorithmen, die Methoden aus der Zahlen-Theorie 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 kennen lernen.

Zunächst noch ein paar Worte zu den deterministischen Verfahren: Eine nahe liegende 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 Binär-System, dann beträgt die Eingabe-Länge $l = \log_2 n$. Als Laufzeit erhalten wir $\mathcal{O}(\sqrt{2^l}) = \mathcal{O}(2^{l/2})$, das Verfahren benötigt also exponentiellen Aufwand in der Eingabe-Länge 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 Zahlen-Theorie, wie sie z. B. in der Vorlesung Diskrete Strukturen I behandelt werden. 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 Eratosthenes, das dem einen oder anderen vielleicht aus der Schule bekannt ist.

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

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

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

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

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 Zahlen-Theorie 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 Rückgabe-werte 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:*

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

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 dieses 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 Test-Bedingung erfüllt sein kann. Umgekehrt gilt: Wenn die Antwort Composite lautet, so ist n sicherlich zusammengesetzt. Also:

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

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 Fehler-Wahrscheinlichkeit 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 Binär-Darstellung von k wiederholt Quadrieren und unmittelbar danach den Modulus berechnen. Wir zeigen dies an einem Beispiel: $a = 43$, $k = 50$ und $n = 67$. Im Binär-System 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 Kryptographie: Das RSA-Verfahren

2.1 Einführung

Das RSA-Verfahren ist ein klassischer Vertreter der *asymmetrischen* Verschlüsselungsverfahren. Bei diesen Verfahren werden zum Ver- und Entschlüsseln unterschiedliche Schlüssel verwendet. Der grundsätzliche Ablauf einer sicheren Kommunikation verläuft dabei nach folgendem Schema:

Der Sender einer Nachricht besorgt sich von einer zentralen Stelle den *public key* des Empfängers und verschlüsselt mit diesem seine Nachricht. Der Empfänger besitzt einen *private key*, den er geheim hält und der deshalb (hoffentlich) nur ihm bekannt ist. Mit Hilfe dieses private key können Nachrichten entschlüsselt werden, die mit dem zugehörigen public key verschlüsselt wurden. Der Vorteil solcher Verfahren besteht darin, daß vor der Kommunikation keine Schlüssel auf sicheren Kanälen ausgetauscht werden müssen, da auch ein potentieller Angreifer den public key kennen darf. Sie werden deshalb häufig dazu verwendet, geheime Schlüssel für symmetrische Verfahren auszutauschen, die dann für die nachfolgende Kommunikation verwendet werden.

Oftmals können asymmetrische Verfahren auch in der Gegenrichtung verwendet werden. Das bedeutet, daß der Sender einer Nachricht diese mit seinem private key verschlüsselt und der Empfänger sie mit dem public key entschlüsselt. Selbstverständlich dient dieses Vorgehen nicht der Geheimhaltung der Nachricht (da der public key allgemein zugänglich ist), sondern zur Realisierung einer *digitalen Signatur*. Wenn durch das Entschlüsseln mit dem public key ein sinnvoller Klartext zum Vorschein kommt, so kann der Empfänger mit hoher Wahrscheinlichkeit davon ausgehen, daß die Nachricht auch wirklich vom Sender mit dem zugehörigen private key stammt. Es ist äußert schwierig, ohne Kenntnis des private key eine Nachricht zu erzeugen, die nach Entschlüsselung mit dem public key irgendeinen Sinn ergibt.

2.2 Etwas Zahlentheorie

Die Gruppe $\mathbb{Z}_n^* := \{x \in \mathbb{Z}_n \setminus \{0\} \mid \mathrm{ggT}(x, n) = 1\}$ spielt beim RSA-Verfahren eine wichtige Rolle. Die Anzahl der Elemente in $\mathbb{Z}_n^*$ wird angegeben durch folgende Funktion:

Definition 7 *Die Eulersche Φ-Funktion ist definiert durch $\Phi(n) := |\mathbb{Z}_n^*|$.*

Wir benötigen die Werte von Φ nur für einen einfachen Spezialfall:

Lemma 8 *Für $n = pq$ mit p, q prim gilt $\Phi(n) = (p-1)(q-1)$.*

Ferner verwendet das RSA-Verfahren einen Algorithmus zur Berechnung der multiplikativen Inversen in $\mathbb{Z}_n^*$.

Satz 9 *Für $n > 0$ kann das multiplikative Inverse zu $z \in \mathbb{Z}_n^*$ in polynomieller Zeit berechnet werden.*

Beweis: Wir berechnen mit Hilfe des erweiterten Euklidischen Algorithmus x, y so, daß

$$1 = \mathrm{ggT}(n, z) = xn + yz$$

Offensichtlich folgt daraus $1 = yz \bmod n$ und y ist somit das multiplikative Inverse zu z. □

Das folgende Theorem ist eine Verallgemeinerung des Satzes von Fermat (da $\Phi(p) = p - 1$ falls p prim).

Satz 10 (Satz von Euler) *Für $n > 1$ und $x \in \mathbb{Z}_n^*$ gilt*

$$x^{\Phi(n)} = 1 \bmod n$$

2.3 Funktionsweise von RSA

2.3.1 Initialisierungsphase

Wenn Alice mit RSA verschlüsselte Nachrichten empfangen will, so muß sie zunächst zwei Primzahlen p und q wählen. Diese Primzahlen sollten beide eine recht große Bitlänge besitzen, da davon die Sicherheit der Kommunikation abhängt. Allerdings steigt mit der Größe der Zahlen auch der Aufwand zum Ver- und Entschlüsseln.

Alice berechnet nun $n = pq$, sowie $\Phi(n) = (p-1)(q-1)$. Ferner wählt sie zufällig ein Element $e \in \mathbb{Z}^*_{\Phi(n)}$ mit $e > 1$. Mit Hilfe des Euklidischen Algorithmus kann sie effizient das multiplikative Inverse $d := e^{-1} \bmod \Phi(n)$ zu e berechnen. Da $ed = 1 \bmod \Phi(n)$ können wir ansetzen, daß $ed = j\Phi(n) + 1$ für ein geeignetes j.

Damit ist die Vorbereitung abgeschlossen. Das Paar (n, e) wird von Alice veröffentlicht und bildet den public key, während d den private key darstellt. Die Primzahlen p und q werden für die eigentliche Kommunikation nicht mehr benötigt, müssen aber geheimgehalten werden, denn mittels p und q kann $\Phi(n)$ und damit d effizient berechnet werden. Wenn ein Angreifer nur n kennt, so nimmt man an, daß die Faktorisierung $n = pq$ nicht effizient berechnet werden kann. Es konnte leider bislang noch nicht bewiesen werden, daß es keinen effizienten Faktorisierungsalgorithmus gibt, aber es wird angenommen, daß dieses Problem nicht in polynomieller Zeit gelöst werden kann.

2.3.2 Kommunikationsphase

Bob schickt nun an Alice eine Nachricht m. Wir nehmen an, daß die Nachricht als Zahl $m < n$ gegeben ist. Dazu unterteilt man beispielsweise einen Text in hinreichend kleine Blöcke (z. B. zu k Bits, wenn 2^k die größte Zweierpotenz kleiner als n ist) und interpretiert die Binär-Darstellung der Nachricht als Zahl).

Die Kommunikation verläuft wie folgt:

1. Bob berechnet die verschlüsselte Nachricht c durch $c := m^e \bmod n$ und sendet c an Alice.

2. Alice erhält den Klartext, indem sie $c^d \bmod n$ berechnet, denn

$$c^d = (m^e)^d = m^{ed} = m^{j\Phi(n)+1} = (m^{\Phi(n)})^j \cdot m = 1^j \cdot m = m \bmod n$$

   wegen Satz 10.

2.3.3 Digitale Signatur

Das digitale Signieren einer Nachricht funktioniert analog, wobei d und e die Rollen tauschen:

1. Alice berechnet die signierte Nachricht s durch $s := m^d \bmod n$ und sendet s an Bob.

2. Bob erhält den Klartext, indem er $s^e \bmod n$ berechnet, denn

$$s^e = (m^d)^e = m \bmod n$$

2.4 Generierung zufälliger Primzahlen

Wie bereits in Abschnitt 2.3.1 beschrieben, werden für den RSA-Algorithmus zufällig generierte Primzahlen benötigt. Der Algorithmus hierzu ist sehr einfach zu realisieren, wenn man über einen Primzahltest verfügt. Man generiert hierzu eine Zufallszahl der gewünschten Größe und testet, ob sie prim ist. Wenn nein, so wiederholt man das Verfahren, bis man Erfolg hat.

Man kann zeigen, daß es zwischen 1 und n ungefähr $n/\log(n)$ Primzahlen gibt. Also hat man bei jedem Versuch eine Erfolgswahrscheinlichkeit von etwa $1/\log(n)$, wenn man jede Zahl zwischen 1 und n gleichwahrscheinlich wählt, und ist somit im Mittel nach $\log(n)$ fertig. In der Praxis wird man allerdings bei der Wahl von p und q die Länge der Zahlen in Bits festlegen und keine kleineren Zahlen generieren, da, wie oben erwähnt, die Sicherheit von RSA unmittelbar davon abhängt. Außerdem macht es natürlich keinen Sinn, gerade Zahlen zu generieren.