

Faktorisierungsalgorithmen

Moritz Kerger

05.12.2023

Faktorisierung

Anwendungsfälle

Komplexität/Laufzeit

Probleme

Algorithmen

Probedivision

Fermat Faktorisierung

Faktorisierung nach Lehmann

Pollard-Rho

Unter Faktorisierung versteht man allgemein das Zerlegen einer Zahl in nichttriviale Teiler. Jede natürliche Zahl kann als Produkt von Primzahlen dargestellt werden. Gibt es nur einen Primfaktor, so muss die Zahl selbst eine Primzahl sein.

- ▶ Faktorisierung algebraischer Terme.
- ▶ Linearfaktorzerlegung von Polynomen.
- ▶ Faktorisierung von Matrizen zur Lösung von LGS.
- ▶ Faktorisierungsalgorithmen können verwendet werden, um die Kryptografische Sicherheit des RSA-Verfahrens zu brechen.

PublicKey(e, N), PrivateKey(d)

Ein Außenstehender kennt die zusammengesetzte Zahl N und e . Angenommen es existiert ein Faktorisierungsalgorithmus, der N in kurzer Zeit faktorisieren kann. Dann kann mit den Faktoren p, q folgendermaßen auf den PrivateKey geschlossen werden:

- ▶ Berechne $\Phi(N) = (p - 1) \cdot (q - 1)$
- ▶ d ist das multiplikative Inverse von $e \bmod \Phi(N)$.
- ▶ Suche ein $e \cdot d \equiv 1 \bmod \Phi(N)$

Bei der Fermat Faktorisierung wird das nochmal an einem Beispiel demonstriert.

Das Faktorisierungsproblem kann nach heutigem Wissensstand nur in exponentieller Laufzeit gelöst werden*. Für Zahlen mit vielen Stellen faktorisiert das Zahlkörpersieb am schnellsten.

$$O\left(\sqrt[3]{\frac{64}{9}} \sqrt[3]{\log N} \sqrt[3]{(\log \log N)^2}\right)$$

* Ausnahme bildet Shor's Algorithmus

Große Primzahlen berechnen und multiplizieren ist einfacher, als die Faktoren eines großen Primzahlprodukts zu finden. Diese Komplexitätsdifferenz macht sich das RSA Verfahren zunutze.

Rechenaufwand zur Faktorisierung bestimmter RSA-Zahlen:

- ▶ RSA-330 - 1991 - Mehrere Tage Rechenaufwand
- ▶ RSA-640 - 2005 - 5 Monate auf 80 AMD Opteron CPUs
- ▶ RSA-829 - 2020 - 2700 CPU-Jahre auf Intel Xeon CPUs
- ▶ RSA-2048 - ?

RSA-2048 ist mit bekannten Algorithmen auf einer gewöhnlichen Rechnerarchitektur nicht faktorisierbar.

Die Zahl N soll in ein Produkt von Primzahlen zerteilt werden.

$$N = \prod_{p \in \mathbb{P}} p^{n_i} \quad n_i \in \mathbb{N}, N \in \mathbb{N}$$

- ▶ Generiere Liste von Primzahlen.
- ▶ Teile N so lange durch eine Primzahl, bis diese N nicht mehr teilt.
- ▶ Fahre mit der nächsten Primzahl fort, bis $p_i > \sqrt{N}$.

Faktoriere 1980

Faktoriere 1980

► $N := 1980/2 = 990$

Faktoriere 1980

► $N := 1980/2 = 990$

► $N := 990/2 = 495$

Faktorisiere 1980

- ▶ $N := 1980/2 = 990$
- ▶ $N := 990/2 = 495$
- ▶ $N := 495/3 = 165$

Faktorisiere 1980

- ▶ $N := 1980/2 = 990$
- ▶ $N := 990/2 = 495$
- ▶ $N := 495/3 = 165$
- ▶ $N := 165/3 = 55$

Faktorisiere 1980

- ▶ $N := 1980/2 = 990$
- ▶ $N := 990/2 = 495$
- ▶ $N := 495/3 = 165$
- ▶ $N := 165/3 = 55$
- ▶ $N := 55/5 = 11$

Faktorisiere 1980

- ▶ $N := 1980/2 = 990$
- ▶ $N := 990/2 = 495$
- ▶ $N := 495/3 = 165$
- ▶ $N := 165/3 = 55$
- ▶ $N := 55/5 = 11$
- ▶ $5 > \sqrt{11}$, also brich ab.

Die Primfaktoren sind $\{2, 2, 3, 3, 5, 11\}$, also $2^2 \cdot 3^2 \cdot 5 \cdot 11 = N$

Faktorisiere 999999866000004473

▶ $N := 999999866000004473/2$

▶ ...

Faktorisiere 999999866000004473

- ▶ $N := 999999866000004473/2$
- ▶ ...
- ▶ $N := 999999866000004473/999999929 = 999999937$
- ▶ $999999929 > \sqrt{999999937}$, also brich ab.

Die Primfaktoren sind $\{999999929, 999999937\}$, also
 $999999929 \cdot 999999937 = N$

Für Eingaben mit großen Primfaktoren dauert das Verfahren lange. Im zweiten Beispiel sind zwei große Primfaktoren enthalten. Dafür braucht der Computer mehrere Sekunden (≈ 12). Typische RSA Schlüssellängen sind heutzutage 2048-4096 bits.

Laufzeit: $O(\sqrt{N})$

Laufzeit mit Primzahl-LUT $O\left(\frac{\sqrt{N}}{\log(N)}\right)$

Die Methode von Fermat nutzt die dritte Binomische Formel.
Ziel ist es, a und b zu finden, für die gilt:

$$N = a^2 - b^2$$

Dadurch ergeben sich dann die Faktoren

$$N = (a + b) \cdot (a - b) = p \cdot q$$

$$p = (a + b)$$

$$q = (a - b)$$

Wähle a ungefähr in der "Mitte" von N und schau dann, ob es ein b gibt, das den gleichen Abstand zu beiden Faktoren hat. Ist b keine solche Zahl, dann vergrößern wir a . Da der Algorithmus am Anfang von einem kleinen b und einem $a \approx \sqrt{N}$ ausgeht, funktioniert dieser Algorithmus besonders gut für ähnlich große Primfaktoren.

Wir berechnen p und q folgendermaßen:

```
fermat_factor(N):  
    a ←  $\left\lceil \sqrt{N} \right\rceil + 1$   
    while True:  
        b ←  $\sqrt{a^2 - N}$   
        if isInteger(b):  
            break  
        a ← a + 1  
    return (a+b), (a-b)
```

$$N = 999896292007358227$$

► Schritt 1:

► $a = 999948145$

► $b = \sqrt{a^2 - N} = 26107.140747312795$

$$N = 999896292007358227$$

► Schritt 1:

► $a = 999948145$

► $b = \sqrt{a^2 - N} = 26107.140747312795$

► Schritt 2:

► $a = 999948146$

► $b = 51783$

► b ist ganzzahlig, also brich ab.

$$N = 999896292007358227$$

► Schritt 1:

► $a = 999948145$

► $b = \sqrt{a^2 - N} = 26107.140747312795$

► Schritt 2:

► $a = 999948146$

► $b = 51783$

► b ist ganzzahlig, also brich ab.

$$p = (999948146 - 51783)$$

$$q = (999948146 + 51783)$$

$$\text{Test: } 999948146^2 - 51783^2 = 999896292007358227 \checkmark$$

Die Methode von Lehmann nutzt unter anderem die Probedivision.

- ▶ Führe Probedivision bis $\sqrt[3]{N}$ durch.
- ▶ Wenn keine Faktoren gefunden wurden, besteht die Zahl aus zwei Primfaktoren, oder ist selbst Primzahl.
- ▶ Führe den nachfolgenden Algorithmus aus.

```
lehman_factor(N):  
  for k  $\leftarrow$  1.. $\left\lceil \sqrt[3]{N} \right\rceil$  :  
    for a  $\leftarrow$   $\left\lceil 2\sqrt{kN} \right\rceil$ .. $\left\lfloor 2\sqrt{kN} + \frac{\sqrt[6]{N}}{4\sqrt{k}} \right\rfloor$  :  
      b  $\leftarrow \sqrt{a^2 - 4kN}$   
      if isInteger(b):  
        return ggT(a + b, N)
```

$$N = 999896292007358227$$

► Schritt 1:

► $k = 1$

► $x = 1999896290$

► $y = \sqrt{x^2 - 4kn} = 52214,2$

$N = 999896292007358227$

► Schritt 1:

► $k = 1$

► $x = 1999896290$

► $y = \sqrt{x^2 - 4kn} = 52214,2$

► Schritt 2:

► $x = 1999896291$

► $y = 82012,9$

$$N = 999896292007358227$$

► Schritt 1:

► $k = 1$

► $x = 1999896290$

► $y = \sqrt{x^2 - 4kn} = 52214,2$

► Schritt 2:

► $x = 1999896291$

► $y = 82012,9$

► Schritt 3:

► $x = 1999896292$

► $y = 10725916356$

► $y = 103566$

► $\gcd(1999896292 + 103566, N) = 999999929$

Der Algorithmus terminiert für ungerade Zahlen.

Die Laufzeit für den ersten Schritt beträgt $\sqrt[3]{N}$. Werden in diesem keine Faktoren gefunden, beträgt die Laufzeit $\sqrt[3]{N} \cdot \log \log n$.

Für Beispiel 2 aus der Probedivision kann die Rechenzeit von 12 Sekunden auf 0.0002 Sekunden reduziert werden!

Definiere Funktion $g(x) := x^2 - c \bmod k$.

Für den Anfang ist $c = 1$ und $k = N$.

Idee: Wende die Funktion $g(x)$ immer wieder auf das Ergebnis an.

$$x_i = g^i(x_0)$$

Irgendwann wird wegen $\bmod k$ ein Zyklus entstehen.

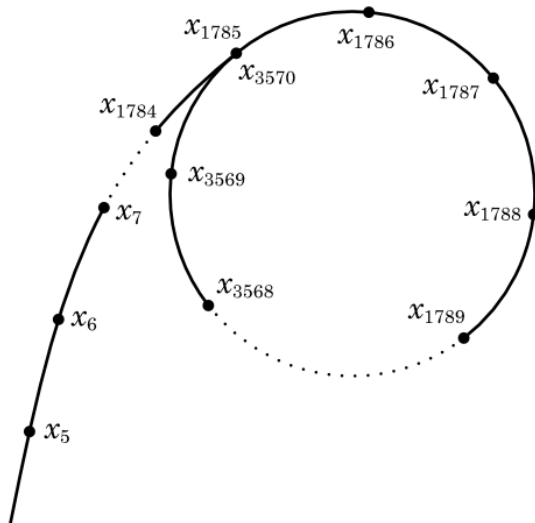


Figure: Charakteristischer Zyklus

```
pollard_rho(N):  
    t,h ← 2, 2  
    while True:  
        t = g(t)  
        h = g(g(h))  
        d = gcd(|t-h|, N)  
        if d == N:  
            return False  
        if d > 1:  
            return False  
    if d:  
        return int(d)
```

```
pollard_rho(N):  
    t,h ← 2, 2  
    while True:  
        c ← 1  
        while True:  
            t = g(t)  
            h = g(g(h))  
            d = gcd(|t-h|, N)  
            if d == N:  
                d ← False  
            if d > 1:  
                break  
        if d:  
            return int(d)  
    c ← c + 1
```

$$N = 12999999077$$

► Schritt 1:

- $t = 2^2 - 1 = 3$
- $h = (2^2 - 1)^2 - 1 = 8$
- $d = \gcd(5, N) = 1$

► Schritt 2:

- $t = 3^2 - 1 = 8$
- $h = (8^2 - 1)^2 - 1 = 3968$
- $d = \gcd(3960, N) = 1$

► Schritt 3:

- $t = 8^2 - 1 = 63$
- $h = (3968^2 - 1)^2 - 1 = 8766871215$
- $d = \gcd(8766871152, N) = 13$
- $d > 1$, also ist d Faktor von N
- $p = 13, q = N/13 = 999999929$

$$\text{Test: } 13 \cdot 999999929 = 12999999077 \checkmark$$

- ▶ Für Primzahlen kein Abbruch

$$\gcd(|t-h|, N) = 1 \text{ für } N \in \mathbb{P}$$






- ▶ $c = 1$ und $k < N$ liefert das eventuell keine Faktoren.
- ▶ Für mehr als 2 Faktoren rekursive Anwendung auf das Ergebnis nötig.
- ▶ Bei 2 Primzahlen gleicher Länge terminiert der Algorithmus im Mittel deutlich später.

Erwartungswert der Zyklenlänge bei Zahl N:

$$\sqrt{k}$$

Da die Laufzeit von der Zyklenlänge und damit k abhängt, ist dieser Algorithmus besonders effizient für Zahlen, bei denen ein Faktor sehr klein ist. Handelt es sich beispielsweise um eine zusammengesetzte Zahl mit zwei ähnlich großen Primfaktoren, so terminiert der Algorithmus in der Hälfte aller Fälle nach

$$O\left(N^{\frac{1}{4}} \cdot \text{Li}(N)\right)$$

-  *Recognizing Primes and Composites*, pages 117–171.
Springer New York, New York, NY, 2005.
-  W. Y. Feng.
How to quickly factor a number: Pollard's rho algorithm.
-  B. R. S. Lehman.
Factoring large integers.
Mathematics of Computation, 28:637–646, 1974.
-  C. Pomerance and P. Erdős.
A tale of two sieves.
1998.
-  M. Pound and S. Riley.
Breaking rsa.
- [1] [2] [3] [4] [5]