Primzahlerzeugung

Public – key Algorithmen benötigen Primzahlen.

Ein Netzwerk benötigt sehr viele Primzahlen.

1. Findet man genug Primzahlen?
   Ja! Es gibt ungefähr 10^{151} Primzahlen mit einer Länge von 512 Bits oder kürzer.
   Für Zahlen nahe n ist die Wahrscheinlichkeit für eine Zufallszahl prim zu sein ungefähr eins zu $\ln(n)$. Die Anzahl der Primzahlen kleiner n ist ungefähr $n/\ln(n)$.

2. Könnte es nicht sein, dass zwei verschiedene Personen zufälligerweise die gleiche Primzahl auswählen, obwohl sie dies nicht wollen?
   Nein! Wenn man unter 10^{151} Primzahlen wählen kann, wird dies nicht geschehen.

3. Könnte man nicht eine Datenbank mit allen in Frage kommenden Primzahlen aufbauen? Und diese dann dazu benutzen, public – key Algorithmen zu knacken?
   Nein! Es gibt einfach zu viele.

4. Man muss allerdings darauf achten, dass die Zufallsgeneratoren, mit denen man die Primzahlen auswählt, kryptographisch sicher, d. h. insbesondere unvorhersagbar sind!

Der Rabin – Miller – Test

- Es ist erstaunlich einfach zu prüfen, ob eine Zahl eine Primzahl ist. (Zumindest im Vergleich mit der Aufgabe, Primfaktoren zu finden)
- Der Rabin – Miller – Test ist probabilistisch, d. h. es gibt eine gewisse Wahrscheinlichkeit dafür, dass er eine falsche Antwort gibt.
- Durch wiederholtes Durchlaufen des Tests mit verschiedenen Parametern kann man die Irrtumswahrscheinlichkeit auf ein akzeptables Niveau bringen.

Der Rabin – Miller – Test ist ein modifizierter Fermat – Test: Er prüft, ob eine gegebene Zahl n prim ist. Man wählt eine Zufallszahl $a < n$ und prüft, ob eine gewisse Eigenschaft von a mod n erfüllt ist, die immer gilt, wenn n prim ist. Für zusammengesetzte Zahlen kann man hingegen zeigen, dass diese Eigenschaft für höchstens 25% der möglichen Werte von a gilt. Indem man den Test für verschiedene Zufallszahlen a wiederholt, kann man das Vertrauen in das Endergebnis steigern.

Ist n prim, so wird n auch immer als prim getestet. Ist n nicht prim, so zeigen 75% der möglichen Werte von a dies an und man kann die Wahrscheinlichkeit, dass dieses n mehrfache Tests besteht, so klein machen wie man will.

Im Algorithmus unten ist die Wahrscheinlichkeit für ein falsches Ergebnis 2^{-128}.

Der Algorithmus:

- Zunächst wird die Zahl $n-1$ geschrieben als $2^t s$, wobei s eine ungerade Zahl ist.

- Will man a^{n-1} berechnen, so kann man zunächst a^s berechnen und dann das Ergebnis t mal quadrieren um $a^{s\cdot 2^t} = a^{n-1}$ zu erhalten.

- Ist nun $a^s \equiv 1 \bmod n$, so ändert wiederholtes Quadrieren das Ergebnis nicht und man hat $a^{n-1} \equiv 1 \bmod n$.

- Ist $a^s \neq 1 \bmod n$, so sieht man sich die Zahlen $a^s, a^{s\cdot 2}, a^{s\cdot 2^2}, a^{s\cdot 2^3}, \ldots a^{s\cdot 2^t} \bmod n$ an.
  Ist n eine Primzahl, so weiß man, dass die letzte Zahl gleich 1 sein muss.
  Ist n eine Primzahl, so sind die Zahlen 1 und $n-1$ die einzigen Zahlen, die $x^2 \equiv 1 \bmod n$ erfüllen.
  Also muss für Primzahlen n eine der Zahlen in der Reihe oben gleich $n-1$ sein (, denn sonst könnte die letzte Zahl nie 1 sein).

- Genau das prüft der Rabin –Miller – Test. Zeigt irgendeine Wahl von a, dass n zusammengesetzt ist, so bricht der Test sofort ab.

- Besteht n den Test, so wird er solange mit verschiedenen Zufallswerten für a wiederholt, bis die Wahrscheinlichkeit, dass man irrtümlich eine zusammengesetzte Zahl als prim erklärt, genügend klein ist.

Rabin-Miller nach Ferguson / Schneier: Practical Cryptography

function Rabin-Miller
input: n An odd number ≥ 3.
output: b Boolean indicating whether *n* is prime or not.

```
assert n ≥ 3 ∧ n mod 2 = 1
First we compute (s, t) such that s is odd and 2^t s = n − 1.
(s, t) ← (n − 1, 0)
while s mod 2 = 0 do
    (s, t) ← (s/2, t + 1)
od
We keep track of the probability of a false result in k. The probability is at most 2^(−k).
We loop until the probability of a false result is small enough.
k ← 0
while k < 128 do
    Choose a random a such that 2 ≤ a ≤ n − 1.
    a ∈_R {2, 3, ... n − 1}
    The expensive operation: a modular exponentiation.
    v ← a^s mod n
    When v = 1, the number n passes the Test for basis a.
    if v ≠ 1 then
        The sequence v, v^2, ... v^(2^t) must finish on the value 1, and the last value
        not equal to 1 must be n − 1 if n is a prime.
        i ← 0
        while v ≠ n − 1 do
            if i = t − 1 then
                return false
            else
                (v, i) ← (v^2 mod n, i + 1)
            fi
        od
    fi
    When we get to this point, n has passed the primality test for the basis a. We
    have therefore reduced the probability of a false result by a factor of 2^2, so we
    can add 2 to k.
    k ← k + 2
od
return true
```