Aufgabe: Sei A ein probabilistischer Algorithmus so wie der Solovay-Strassen-Primzahltest:

Eingabe n prim $\longrightarrow$ "ja"

n nicht prim $\xrightarrow{1/2}$ "ja"

n nicht prim $\xrightarrow{1/2}$ "nein"

Wie kann man die „Fehlerwahrscheinlichkeit" von 1/2 auf 10^{-20} reduzieren?

Antwort: Man wiederhole den Algorithmus A bis zu t-mal. Wenn mindestens 1x die Antwort "nein" ist, so ist die endgültige Antwort "nein" und ist korrekt.

Wenn t-mal die Antwort "ja" ist, so besteht nur noch eine Fehlerwahrscheinlichkeit von 2^{-t}.

Wir setzen $2^{-t} = 10^{-20}$ und lösen nach t auf: $t \approx 66$. So oft muss der Algorithmus A wiederholt werden.

Wiederholung zum <u>Solovay-Strassen-Primzahltest</u>:

Wähle mehrere Zufallszahlen $a<n$, n ist Eingabe.

Falls
$$\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod n$$

so ist n sicher keine Primzahl. Wenn jedesmal

$\left(\frac{a}{n}\right) \equiv a^{(n-1)/2}$ gilt, so ist n sehr wahrscheinlich

Primzahl.

<u>Zahlenbeispiel:</u> $a=2$; $n=15$ (keine Primzahl)

$$\Rightarrow \left(\frac{2}{15}\right) = -1$$

Aber: $2^{(15-1)/2} \equiv 8 \pmod{15}$

da 15 keine Primzahl ist, braucht diese Berechnung auch nicht unbedingt ± 1 ergeben. Schon allein diese Tatsache (ohne mit dem Jacobi-Symbol $\left(\frac{2}{15}\right)$ vergleichen zu müssen) zeigt, dass 15 keine Primzahl ist.

Eine kryptographische Anwendung des Jacobi-Symbols: Sichere Verschlüsselung einzelner Bits (sog. <u>Blob</u>).

Falls $n = p \cdot q$ (p, q unbekannt)

und falls $\left(\frac{a}{n}\right) = 1 = \left(\frac{a}{p}\right) \cdot \left(\frac{a}{q}\right)$

so gibt es 2 Möglichkeiten:

$\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = 1$ ($\hat{=}$ geheimes Bit $= 1$)

oder $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = -1$ ($\hat{=}$ geheimes Bit $= 0$)

Um also ein Bit $b \in \{0,1\}$ zu verschlüsseln, wähle große Primzahlen p, q, berechne $n = p \cdot q$.

Wähle zufällig $a \in \mathbb{Z}_n^*$ bis:

$$\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = \begin{cases} 1 & \text{, falls } b = 1 \\ -1 & \text{, falls } b = 0 \end{cases}$$

Dies erfordert durchschnittlich 4 Versuche.

Veröffentliche a und n. (Dies kann Bestandteil eines kryptographischen Protokolls sein: Zuerst wird (a, n) übermittelt (commitment), in späterer Runde wird p, q bekannt gegeben.)

Das zugrunde liegende algorithmische Problem:

<u>gegeben</u>: n $(= p \cdot q$, geheim$)$
und $a \in \mathbb{Z}_n^*$ so dass $\left(\frac{a}{n}\right) = 1$.

<u>Stelle fest</u>, ob $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = 1$ (entspricht $b = 1$)
oder $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = -1$ (entspricht $b = 0$)

Dieses Problem heißt <u>quadratisches Rest-Problem</u>.

<u>Bemerkung</u> zu blinder Unterschrift:

Sofern sowieso mit Hashfunktion h unterschrieben werden soll (um die Länge der Unterschrift zu begrenzen), so sendet A an B den Hashwert $h(m)$ zur Unterschrift. Auf diese Weise kann B ebenfalls das ursprüngliche Dokument m nicht einsehen.
B unterschreibt dann $h(m)$, also $\hat{m} = D(k_B', h(m))$
Verifikation der Unterschrift: $E(k_B, \hat{m}) \stackrel{?}{=} h(m)$.

Idee von Commitment:

A hinterlegt bei B verschlüsselte Info $\tilde{x}$
Später schickt A an B den Schlüssel, um ~~ver~~ $\tilde{x}$ entschlüsseln zu können und x lesen zu können.
Keinesfalls darf es für A möglich sein, einen „falschen" Schlüssel zu schicken, der bei Entschlüsselung von $\tilde{x}$ zu falscher Botschaft x' führt.

Hashfunktion, um Commitment zu ermöglichen

A hinterlegt $\tilde{x} = E(k, x)$ bei B

Zusätzlich: $h(x) = y$

Später schickt A den Schlüssel k an B,
so dass B die Nachricht x entschlüsseln kann + überprüfen, dass $h(x) \stackrel{?}{=} y$.

Einige Faktorisierungsalgorithmen

(es geht also um Möglichkeiten der Kryptoanalyse, z.B. bei RSA).

Zu faktorisieren sei die Zahl n (bereits getestet, dass n keine Primzahl ist).
n sei eine Binärzahl mit m Bits.

Naiver Algorithmus:

Durchsuche alle Zahlen ab 2 (und ab 3 alle ungeraden), ob ein nicht-trivialer Teiler von n dabei ist. Es genügt, die Zahlen bis $\sqrt{n}$ zu durchsuchen.
Daher: Laufzeit entspricht Anzahl Schleifendurchläufe $= O(\sqrt{n}) = \underline{\underline{O(2^{m/2})}}$.

Pollard's ρ-Algorithmus beruht auf dem Geburtstagsproblem:

Man erzeugt Zufallszahlen

$$x_1, x_2, x_3, \ldots < n$$

Sei p der kleinste Primfaktor von n. Dann gilt: $p \leq \sqrt{n}$.

Betrachten wir diese Zufallszahlen modulo p (obwohl p im Moment noch unbekannt):

$$x_1', x_2', x_3', \ldots \qquad \text{wobei } x_i' = x_i \bmod p$$

Dann sind dies ebenfalls Zufallszahlen, aber im Zahleninterval bis $p-1$ (statt $n-1$).

Laut Geburtstagsproblem wird es bereits nach ca. $\sqrt{p} \leq \sqrt{\sqrt{n}} = n^{1/4}$ vielen Zahlen der Fall sein, dass für ein $i<j$ gilt: $x_i' = x_j'$.

Das bedeutet: $x_i \equiv x_j \pmod p$

(Gleichzeitig ist es sehr wahrscheinlich, dass $x_i \neq x_j$.)

Zahlenbeispiel: $n = 11 \cdot 13 = 143$, also $p = 11 \leq \sqrt{143}$

Zufallszahlen $x_1, x_2, x_3, \ldots = 135, 130, 21, 74, 59, 108, \ldots$

Dieselben Zahlen modulo 11 sind:

$$x_1', x_2', x_3', \ldots = 3, \underbrace{9, 10, 8, 4, 9}_{=}, \ldots$$

Also $x_i = 130$, $x_j = 108$ wobei $130 \equiv 108 \pmod{11}$

Nun gilt: $x_i - x_j \equiv 0 \pmod{p}$, also

$x_i - x_j = k \cdot p$ Also ergibt

$$\mathrm{ggT}(\underbrace{x_i - x_j}_{= k \cdot p}, \underbrace{n}_{= p \cdot q}) = p$$

Wir prüfen dies am Zahlenbeispiel nach:

$$\mathrm{ggT}(130-108, 143) = \mathrm{ggT}(143, 22) = 11$$

Algorithmisch heißt das:

Erzeuge Zufallszahlen $x_1, x_2, x_3, \ldots < n$

Finde „geeignete" i, j und

überprüfe, ob $\mathrm{ggT}(x_i - x_j, n) > 1$. Wenn ja, so ist dies ein Primfaktor von n.

Problem: Die Anzahl der Kandidaten (i,j) mit $i<j$ beträgt bei $\sqrt{p}$ Zufallszahlen $x_1, \dots, x_{\sqrt{p}}$ (diese Zahl sollte laut Geburtstagsproblem ausreichen) $\binom{\sqrt{p}}{2} = O(p) = O(\sqrt{n})$. Dies wäre nicht besser als der naive Algorithmus!

Lösung des Problems:

„Zufallszahlen" werden Computer-intern meist als Pseudozufallszahlen realisiert. Das heißt, es gibt eine Anfangszahl x_0 (den „seed") und dann berechnet man die weiteren Zahlen mit Hilfe einer einfachen Funktion $f: \mathbb{N} \to \mathbb{N}$, so dass

$$x_{i+1} = f(x_i) \bmod n$$

Bei einem linearen Kongruenzgenerator ist

$$f(x) = a \cdot x + b$$

In diesem Fall hat sich eine quadratische Funktion besser bewährt, z.B.

$$f(x) = x^2 + 1$$

Angenommen, für eine (i,j)-Kombination gilt:

$$x_i \equiv x_j \pmod p$$

$$\Downarrow$$

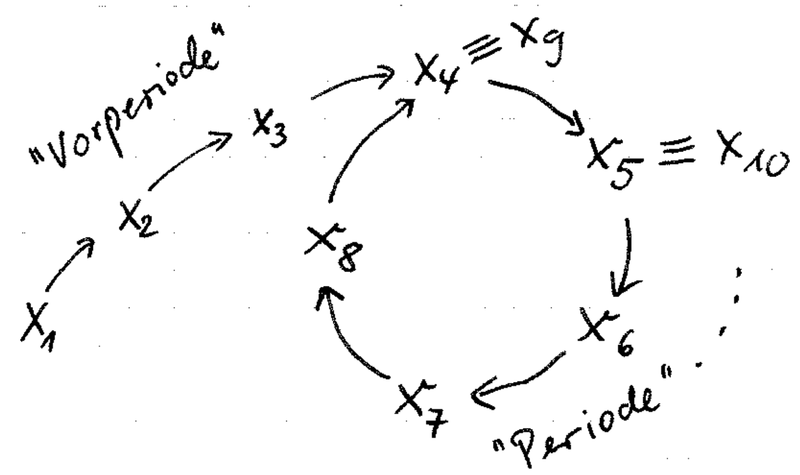
$$x_{i+1} = f(x_i) \equiv f(x_j) = x_{j+1} \pmod p$$

$$\Downarrow$$

$$x_{j+2} = f(x_{i+1}) \equiv f(x_{j+1}) = x_{j+2} \pmod p$$

$\Downarrow$ usw.

<u>Skizze</u> (mit $i=4$; $j=9$):



(Dieses Bild gibt dem Algorithmus den Namen)

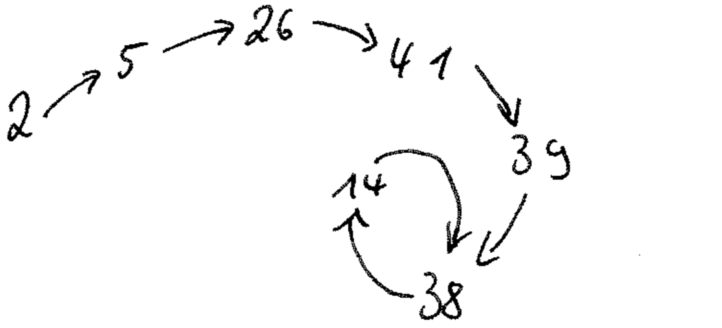
„<u>Floyd's Cycle Detection Trick</u>": Anstatt <u>alle</u> potenziellen (i,j)-Kombinationen mit $i<j$ zu testen, überprüfe: $(1,2)$, $(2,4)$, $(3,6)$, $(4,8)$, $\underline{(5,10)}$, $(6,12)$, usw.

Bei dem obigen Zahlenbeispiel würde man bei $(5,10)$ einen „Treffer" erzielen.

Zahlenbeispiel: $n = 53 \cdot 101 = 5353$; $x_0 = 1$

	x_i	x_{2i}	$ggT(x_i - x_{2i}, n)$	$x_i \bmod 53 = x_i'$
$i=1$	2	5	1	2
2	5	677	1	5
3	26	1681	1	26
4	677	1469	1	41
5	3325	1734	1	39
⇒ 6	1681	3907	53	38
7	4731	2953	1	14
8	1469	3165	53	38

$f(x) = x^2 + 1$



(mod 53)

Es gelte $x_i \equiv x_j \pmod{p}$ $(i<j)$. Wenn man die Elemente $(x_1, x_2), (x_2, x_4), \ldots, (x_k, x_{2k})$ analysiert, wann erzielt man spätestens einen „Treffer"? Frühestmöglicher Zeitpunkt, wenn $k=i$, dann sollte $2k=j$ sein. Worst case wäre, wenn $k=i$ aber auch und $2k = j+1$ + Vielfaches von $(j-i)$. Treffer wird erzielt, sobald $2k-k=k$ ein Vielfaches von $(j-i)$ ist. Der maximale k-Wert ist j

Man erhält folgenden Algorithmus

Eingabe: n

Wähle $x < n$ zufällig (entspricht x_0)

$y := x;$

<u>repeat</u>

$$\left.\begin{array}{l} x := f(x) \bmod n; \\ y := f(y) \bmod n; \\ y := f(y) \bmod n; \end{array}\right\} \text{ erzeugt } (x_i, x_{2i})$$

$g := ggT(x-y, n);$

<u>until</u> $(x \neq y)$ <u>and</u> $(g > 1)$

<u>Output</u> g

wobei $f(x) = x^2+1$

Die erwartete Laufzeit des ρ-Algorithmus ist

$$O(\sqrt{p}) = O(n^{1/4}) = O(2^{m/4})$$

m = Anzahl Bits der Eingabezahl n

<u>Aufgabe:</u> Der ρ-Algorithmus werde auf eine Zahl n als Eingabe angewandt. Man erhält eine Vorperiode der Länge 3 und die Periodenlänge 4, also $x'_3 = x'_{3+4}$.

Skizze:

Man erstelle eine Dreiecksmatrix für alle (i,j)-Kombinationen mit $i<j$ und trage die möglichen Positionen für einen Treffer ein, sowie die Kombinationen $(k,2k)$, die die Floyd-Strategie überprüft!

<u>Antwort:</u>