Aufgabe: Gegeben sei die Faktorisierung von $n-1$, wobei n Primzahl ist. (n sei eine m-Bitzahl.) Welche Komplexität hat ein Algorithmus, der schließlich eine Primitivwurzel a modulo n auffindet? (Ein solcher Algorithmus wird im Zuge der Initialisierung eines Krypto-Verfahrens benötigt, das auf dem diskreten Logarithmus beruht.)

Antwort: Um eine Zahl $a \in \mathbb{Z}_n^*$ auf Primitivwurzel zu testen, muss gelten $\underbrace{a^{(n-1)/q} \not\equiv 1 \pmod n}_{\text{ModExp}(a,(n-1)/q,n) \neq 1}$ für alle Primfaktoren q von $n-1$.

Die hierbei notwendige Exponentiation hat Komplexität $O(m^3)$.

Die Anzahl der verschiedenen Primfaktoren q in $n-1$ kann (grob) mit $\log_2(n-1) \leq m$ abgeschätzt werden. Damit sind wir schon bei $O(m^4)$.

Man testet solange zufällige Kandidaten $a \in \mathbb{Z}_n^*$, bis man eine Primitivwurzel findet. Man sollte also

wissen, wieviele Primitivwurzeln in $\mathbb{Z}_n^* = \{1, 2, \ldots, n-1\}$ vorhanden sind. Wir wissen, dies sind $\varphi(\varphi(n)) = \varphi(n-1)$ viele. Ferner gilt:

$$\varphi(n-1) \geq \frac{n-1}{6 \cdot \ln(\ln(n-1))}$$

Die Wahrscheinlichkeit für eine Primitivwurzel ist somit mindestens $\frac{n-1}{6 \cdot \ln(\ln(n-1))} / (n-1)$

$$\approx \frac{1}{6 \ln(\ln(n))} = \frac{1}{O(\log(m))}$$

Die erwartete Anzahl von Versuchen, bis Primitivwurzel gefunden, ist daher $O(\log m)$.

Die Komplexität insgesamt also: $O(m^4 \cdot \log m)$.

Einige Fakten aus den letzten Vorlesungen, speziell kombiniert:

Sei n eine ungerade Primzahl, $\varphi(n) = n-1$ ist gerade Zahl.

Wegen Euler/Fermat ist für $a \neq 0$:

$$a^{\varphi(n)} = a^{n-1} \equiv 1 \pmod{n}$$

Da $n-1$ gerade Zahl, ist $a^{n-1} = 1$ eine Quadratzahl mit $a^{(n-1)/2}$ eine Quadratwurzel der 1.

Modulo einer Primzahl n sind die einzigen Quadratwurzeln von 1 die Zahlen 1 und -1 $(= n-1)$.

Somit $a^{(n-1)/2} \in \{1, -1\}$.

- Wenn $a^{(n-1)/2} = 1$ so ist a keine Primitivwurzel.
- Wenn $a^{(n-1)/2} = -1$ so könnte a eine Primitivwurzel sein (das hängt von den weiteren $a^{(n-1)/q}$ ab; q Primteiler von $n-1$).

Eine weitere Anwendung (kommt vor bei den verschiedenen Primzahltests):

Wenn n eine Zahl ist, von der unbekannt ist, ob sie eine Primzahl ist oder nicht:

- Falls $a^{n-1} \not\equiv 1 \pmod{n} \Longrightarrow n$ keine Primzahl!
- Falls für einen geraden Exponenten k gilt

$$a^k \equiv 1 \pmod{n}$$

und $a^{k/2} \not\equiv \pm 1 \pmod{n} \Longrightarrow n$ keine Primzahl!

Lemma von Bezout:

Gegeben: a, b und $d = ggT(a,b)$

Dann gibt es $x, y \in \mathbb{Z}$ so dass

$$d = a \cdot x + b \cdot y$$

<u>Beweis:</u> Sei $a = n_1$, $b = n_2$: Betrachte Abfolge beim Euklid-Algorithmus:

$$n_1 = q_1 \cdot n_2 + n_3$$

$$n_2 = q_2 \cdot n_3 + n_4$$

$$\vdots$$

$$n_{t-2} = q_{t-2} \cdot n_{t-1} + n_t$$

$$n_{t-1} = q_{t-1} \cdot n_t + 0$$

$$n_t = d = ggT(a,b)$$

erste Zeile: $n_3 = n_1 - q_1 \cdot n_2 = x \cdot n_1 + y \cdot n_2$ mit $x = 1$, $y = -q_1$

zweite Zeile: $n_4 = n_2 - q_2 \cdot n_3 = n_2 - q_2 \cdot (1 \cdot n_1 - q_1 n_2)$

$$= x' \cdot n_1 + y' \cdot n_2$$

usw. fortfahren bis zur $(t-2)$-ten Zeile

Ergibt eine Linearkombination für n_t der gesuchten Art. □

Methode:
Euklid-Algorithmus so erweitern, dass außer dem $ggT(a,b)$ auch noch Zahlen x, y berechnet werden, so dass $ggT(a,b) = ax + by$.

```
proc ExtEuklid (a,b)
// liefert (d,x,y) wobei d=ggT(a,b)
// und  d = a·x + b·y

if  b=0  then return (a, 1, 0)
else  (d,x,y) := ExtEuklid (b, a mod b);
      return (d, y, x-(a div b)·y)
```

Laufzeitanalyse: ebenfalls $O(m^2)$.

Hauptanwendung von ExtEuklid: Multiplikatives Inverses von $a \in \mathbb{Z}_n^*$ bestimmen:

Berechne $ExtEuklid(a,n) = (d, x, y)$. Dann ist $d=1$, da $a \in \mathbb{Z}_n^*$. Es gilt $d = 1 = a \cdot x + n \cdot y$

$$\equiv a \cdot x \pmod{n}$$

Also ist x (bzw. $x \bmod n$) multipl. Inverses von a.

Zur Korrektheit von ExtEuklid:

Falls $b=0$ so ist $ggT(a,b)=a$.

In diesem Fall ist die Ausgabe $(a,1,0)$ korrekt, denn $a = 1 \cdot a + 0 \cdot 0$

Sei nun $b>0$. Dann wird ExtEuklid rekursiv aufgerufen mit den Parametern $(b, a \bmod b)$ und liefert (d,x,y) als Ergebnis. Dies ist nach Ind.vor. korrekt, d.h. es gilt: $d = ggT(b, a \bmod b) = x \cdot b + y \cdot (a \bmod b)$

Mit Hilfe von $a \bmod b = a - (a \operatorname{div} b) \cdot b$ kann man dies umschreiben zu: $d = x \cdot b + y \cdot (a - (a \operatorname{div} b) \cdot b)$

$= a \cdot y + b \cdot (x - (a \operatorname{div} b) \cdot y)$

Dies sind dann also die Ergebnisparameter, die die Prozedur zurückliefern muss. □

Beispiel: $a = 21$, $b = 30$

a	b	d	x	y
21	30	3	3	-2
30	21	3	-2	3
21	9	3	1	-2
9	3	3	0	1
3	0	3	1	0

rekursiver Abstieg ↓

↑ Rückgabe-Parameter

$3 = 21 \cdot 3 + 30 \cdot (-2)$

Beim Pohlig-Hellman Kryptosystem wählt man eine große Primzahl n. ~~[illegible]~~. Zum Verschlüsseln von $x \in \mathbb{Z}_n^*$ wählt man einen Exponenten $e \in \mathbb{Z}_{n-1}^*$ zufällig und berechnet dazu $d \in \mathbb{Z}_{n-1}^*$ mit

$$e \cdot d \equiv 1 \pmod{n-1}$$

Um d zu finden, verwendet man Ext-Euklid.

Nun gilt: $(\underbrace{x^e \bmod n}_{=c})^d \bmod = x^{ed} \bmod n$

$$= x^{1+k(n-1)} \bmod n = x \cdot \underbrace{(x^{n-1})^k}_{=1} \bmod n$$

$$= x \bmod n = x.$$

Pohlig-Hellman ist ein klassisches Verfahren zur blockweisen Verschlüsselung. Der sowohl A wie B gemeinsam bekannte Schlüssel ist e (aus dem sich d effizient berechnen lässt). Diese beide müssen geheim bleiben.

Für ein echtes Public-Key Krypto-System

würde man ein System benötigen, das

ebenfalls einen Schlüssel e zum Verschlüsseln verwendet und einen Schlüssel d zum Entschlüsseln, wobei e öffentlich ist, aber d geheim bleiben muss.

(Bei Pohlig-Hellman ist das nicht der Fall.)

Aufgabe:
Gibt es eine Möglichkeit, das Pohlig-Hellman System so abzuändern, dass e öffentlich sein kann und trotzdem d geheim bleibt?

Die Ver-/Ent-Schlüsselung funktioniert nach wie vor wie gehabt:

$$\underbrace{(x^e \bmod n)}_{=\text{Chiffre } c}{}^d \bmod n = x.$$

Rivest, Shamir, Adleman (RSA) fanden 1978 einen Weg, wie das funktionieren kann, wofür sie 2002 den Turing-Award erhielten.
(Diffie + Hellman erhielten den Turing Award 2015.)

Die Lösung besteht darin, n nicht als Primzahl zu wählen, sondern $n = p \cdot q$ für 2 große zufällige Primzahlen p und q, die geheim bleiben. Dann ist

$$\varphi(n) = (p-1) \cdot (q-1)$$

Die Berechnung von $\varphi(n)$ ist (bei unbekannten p, q) genauso schwierig wie n zu faktorisieren.

Ähnlich wie bei Pohlig-Hellman wählt man $e \in \mathbb{Z}^*_{\varphi(n)} = \mathbb{Z}^*_{(p-1)(q-1)}$ zufällig und berechnet $d \in \mathbb{Z}^*_{(p-1)(q-1)}$ mit $e \cdot d \equiv 1 \pmod{\varphi(n)}$ mittels ExtEuklid. Tatsächlich gilt nun für alle $x \in \mathbb{Z}_n^*$ genauso, dass

$$\underbrace{\left((x^e \bmod n)^d\right) \bmod n}_{= \text{Chiffre } c} = x$$

Für die äußerst selten vorkommenden

$x \notin \mathbb{Z}_n^*$ (weil $x=0$ oder $ggT(x,n)=p$

oder $ggT(x,n)=q$)

gilt obige Gleichung aber auch (ohne Beweis).

Wenn p und q ca. 500-Bitzahlen sind, so ist die Wahrscheinlichkeit für $x \notin \mathbb{Z}_n^*$ etwa

$2 \cdot 2^{-500} = 2^{-499}$

Allgemein:

Aufbau eines Public Key Systems:

Jeder Teilnehmer X erzeugt durch einen probabilistischen Algorithmus ein Schlüsselpaar (k_X, k_X')

k_X: öffentlich — k_X': geheim.

$k_X \rightarrow k_X'$: unmöglich

mit den entsprechenden Ver- und Entschlüsselungsalgorithmen E und D gilt dann:

$$D(k_X', \underbrace{E(k_X, m)}_{= \text{Chiffre } c}) = m \quad \forall m$$

Konkret bei RSA:

Wählt große Primzahlen p, q (geheim)

Berechnet $n = p \cdot q$ (n ist öffentlich)

Wählt $e \in \mathbb{Z}^*_{\varphi(n)}$ zufällig, wobei $\varphi(n) = (p-1)(q-1)$

(e ist öffentlich). Berechnet d mit

$d \cdot e \equiv 1 \pmod{\varphi(n)}$ mittels Ext Euklid.

(d ist geheim).

Protokoll, so dass A an B eine geheime Nachricht m schicken kann:

A	B
Besorgt sich B's öffentlichen Schlüssel k_B (wie aus einem öffentlich bekannten „Telefonbuch")	
Berechnet $c = E(k_B, m)$	
$\xrightarrow{c}$	
	Berechnet $D(k'_B, c) = m$.

Konkret, bei RSA:

A	B

Besorgt sich den öffentlichen Schlüssel n_B, e_B von B.

Berechnet:

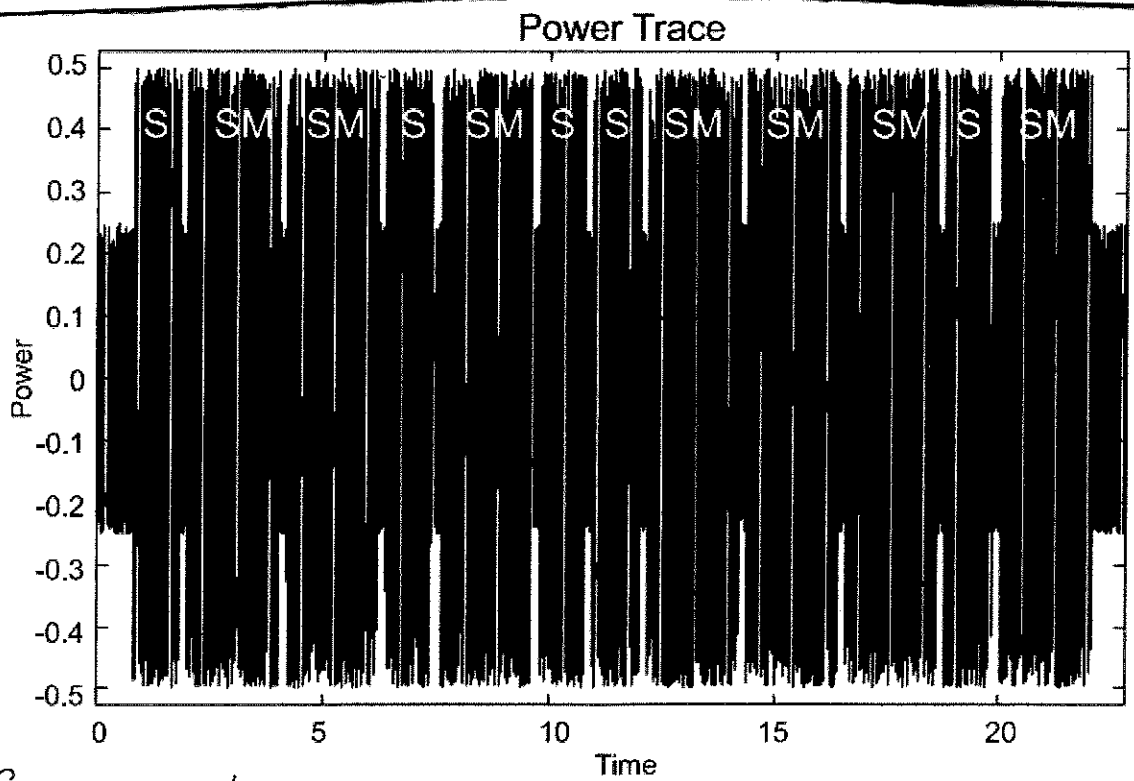
$$c = \underbrace{m^{e_B} \bmod n_B}_{= \text{modexp}(m, e_B, n_B)}$$

(hierbei muss $m < n_B$ sein. Die Nachricht ggf. in Blöcke aufteilen.)

$\xrightarrow{c}$

Berechnet:

$c^{d_B} \bmod n_B = m$



Spannungsschwankungen beim Durchführen einer modularen Exponentiation (z.B. bei RSA). S = square, M = multiply.

Brechen des RSA-Systems bedeutet:

Neben den öffentlich bekannten Daten n, e (von Teilnehmer B) fängt der Kryptoanalytiker eine chiffrierte Nachricht c ab. (an Teilnehmer B)

Aufgabe: m zu berechnen.

Offensichtlich gilt:

- Brechen von RSA $\leq_{\text{eff}}$ n faktorisieren

  (denn wenn man p, q kennt, kann man $\varphi(n) = (p-1)(q-1)$ berechnen und dann das Inverse von e modulo $\varphi(n)$ berechnen).

Es ist unbekannt, ob die Umkehrung $\geq_{\text{eff}}$ gilt.

Wenn man n faktorisieren kann, kann man RSA auch im starken Sinne brechen:

Finde d so dass $d \cdot e \equiv 1 \pmod{\varphi(n)}$

(Tatsächlich ist

RSA im starken Sinne brechen $\equiv_{\text{prob-eff}}$ Faktorisieren)