Ausgewählte Kapitel aus der Kryptographie

Wolfgang M. Ruppert

Sommersemester 2002

22. Juli 2002[1]

[1] Im Sommersemester 2002 am Mathematischen Institut der Universität Erlangen abgehaltene Vorlesung

Inhaltsverzeichnis

Vorbemerkungen

In der 2-stündig abgehaltenen Vorlesung 'Ausgewählte Kapitel aus der Kryptographie' ging es um den vor zwanzig Jahren von A. K. Lenstra, H. W. Lenstra und L. Lovász eingeführten LLL-Gitterbasen-reduktionsalgorithmus und einige Anwendungen. Die numerischen Experimente spielten eine wichtige Rolle für die Vorlesung um ein gewisses Gefühl für die Anwendbarkeit der vorgestellten Verfahren zu gewinnen. Daher finden sich im Anhang die zugehörigen Programme sowie verwendetes Beispielmaterial.

Die Programme benutzen Maple 6.01 oder das C++-Paket NTL-5.2 und liefen auf einem Pentium-II-PC mit 400 MHz und 128 MB Arbeitsspeicher unter Linux.

KAPITEL 1

Gitter und Gitterbasenreduktion

1. Definition

Wir legen im folgenden den reellen Vektorraum $\mathbf{R}^n$ zugrunde, wobei wir die Vektoren als Zeilenvektoren notieren. Außerdem verwenden wir das Standardskalarprodukt $v \cdot w$, das durch

$$(v_1, \ldots, v_n) \cdot (w_1, \ldots, w_n) = v_1 w_1 + \cdots + v_n w_n$$

gegeben ist. Dazu gehört eine Norm, d.h. eine Betragsfunktion, $\|v\| = \sqrt{v \cdot v}$, also

$$\|(v_1, \ldots, v_n)\| = \sqrt{v_1^2 + \cdots + v_n^2}.$$

DEFINITION. Eine Teilmenge $\Lambda \subseteq \mathbf{R}^n$ heißt Gitter, wenn es $\mathbf{R}$-linear unabhängige Vektoren $a_1, \ldots, a_m \in \mathbf{R}^n$ gibt mit

$$\Lambda = \mathbf{Z}a_1 + \mathbf{Z}a_2 + \cdots + \mathbf{Z}a_m = \{x_1 a_1 + x_2 a_2 + \cdots + x_m a_m : x_1, x_2, \ldots, x_m \in \mathbf{Z}\}.$$

Die Menge/Liste der Vektoren $a_1, \ldots, a_m$ heißt dann (eine) Gitterbasis von Λ.

Beispiele:

1. $\mathbf{Z}\sqrt{2} \subseteq \mathbf{R}$ ist ein Gitter.
2. $\mathbf{Z} + \mathbf{Z}\sqrt{2} \subseteq \mathbf{R}$ ist kein Gitter. (Warum?)
3. $\mathbf{Z}(1,0) + \mathbf{Z}(\frac{1}{2}, \sqrt{2}) \subseteq \mathbf{R}^2$ ist ein Gitter.
4. $\mathbf{Z}^n \subseteq \mathbf{R}^n$ ist ein Gitter.

Bemerkung: Ein Gitter $\Lambda \subseteq \mathbf{R}^n$ ist eine Untergruppe der additiven Gruppe des Vektorraums $\mathbf{R}^n$.

Ist $\Lambda = Za_1 + \cdots + \mathbf{Z}a_m \subseteq \mathbf{R}^n$ ein Gitter mit linear unabhängigen Vektoren $a_1, \ldots, a_m$, so ist der von Λ erzeugte $\mathbf{R}$-Untervektorraum

$$\mathbf{R}\Lambda = \mathbf{R}a_1 + \cdots + \mathbf{R}a_m,$$

also ist

$$m = \dim_{\mathbf{R}} \mathbf{R}\Lambda.$$

m wird der Rang des Gitters genannt. Insbesondere ist der Rang die Anzahl der Elemente jeder Gitterbasis von Λ.

Bemerkung: Ist $\Lambda \subseteq \mathbf{R}^n$ ein Gitter mit Gitterbasis $a_1, \ldots, a_m$, schreiben wir $a_i = (a_{i1}, \ldots, a_{in})$, so ist

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix}$$

eine Matrix vom Rang m, d.h. es gibt eine $m \times m$-Untermatrix mit Determinante $\neq 0$.

Version vom 17. Juli 2002

2. Wechsel der Gitterbasis

Wir untersuchen jetzt Inklusion und Gleichheit von Gittern. Zuvor eine Erinnerung an Matrizen mit ganzzahligen Einträgen.
$M_m(\mathbf{Z})$ sei die Menge der $m \times m$-Matrizen mit Einträgen aus $\mathbf{Z}$. Die Menge $M_m(\mathbf{Z})$ ist abgeschlossen gegenüber Matrizenaddition und Matrizenmultiplikation, die Menge $M_m(\mathbf{Z})$ bildet also einen Ring. Bezeichnet 1_m die $m \times m$-Einheitsmatrix, so ist

$$\mathrm{GL}_m(\mathbf{Z}) = \{M \in M_m(\mathbf{Z}) : \text{ es gibt } \widetilde{M} \in M_m(\mathbf{Z}) \text{ mit } M\widetilde{M} = \widetilde{M}M = 1_m\}$$

bezüglich Matrizenmultiplikation eine Gruppe.

Satz.

$$\mathrm{GL}_m(\mathbf{Z}) = \{M \in M_m(\mathbf{Z}) : \det M = \pm 1\}.$$

Beweis: Ist $M \in \mathrm{GL}_m(\mathbf{Z})$, so gibt es $\widetilde{M} \in M_m(\mathbf{Z})$ mit $M\widetilde{M} = 1_m$. Determinantenbildung liefert $\det(M)\det(\widetilde{M}) = 1$. Da M und $\widetilde{M}$ Einträge aus $\mathbf{Z}$ haben ist $\det(M), \det(\widetilde{M}) \in \mathbf{Z}$, was dann sofort $\det(M) = \pm 1$ impliziert.
Sei nun umgekehrt $M \in M_m(\mathbf{Z})$ mit $\det(M) = \pm 1$. Aus der Linearen Algebra kennt man die Formel

$$M \cdot M^{adj} = \det(M) 1_m,$$

wobei

$$M^{adj} = ((-1)^{i+j} \det M_{ji})_{ij},$$

wo M_{ij} aus M durch Streichen der Zeile i und Spalte j entsteht. Offensichtlich ist auch $M^{adj} \in M_n(\mathbf{Z})$. Mit $\det(M) = \pm 1$ folgt dann aus obiger Formel sofort $M \in \mathrm{GL}_m(\mathbf{Z})$. ■

Beispiel: Die Gruppe $\mathrm{GL}_2(\mathbf{Z})$ wird erzeugt von den beiden Matrizen

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

(Diese Aussage erfordert natürlich einen Beweis.)

Lemma. *Seien $\Lambda_a = \mathbf{Z}a_1 + \cdots + \mathbf{Z}a_m$ und $\Lambda_b = \mathbf{Z}b_1 + \cdots + \mathbf{Z}b_m$ Gitter vom Rang m in $\mathbf{R}^n$.*

1. *Genau dann gilt $\Lambda_b \subseteq \Lambda_a$, wenn eine Matrix $M \in M_m(\mathbf{Z})$ existiert mit*

$$\begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} = M \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}.$$

2. *Genau dann gilt $\Lambda_a = \Lambda_b$, wenn eine Matrix $M \in \mathrm{GL}_m(\mathbf{Z})$ existiert mit*

$$\begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} = M \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

Beweis:

1.

$$\begin{aligned} \Lambda_b \subseteq \Lambda_a &\iff b_1, \ldots, b_m \in \Lambda_a \\ &\iff \text{es gibt } m_{ij} \in \mathbf{Z} \text{ mit } b_i = \sum_{j=1}^{m} m_{ij} a_j \text{ für } i = 1, \ldots, m \\ &\iff \text{es gibt } m_{ij} \in \mathbf{Z} \text{ mit } \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} = \begin{pmatrix} m_{11} & \ldots & m_{1m} \\ \vdots & & \vdots \\ m_{m1} & \ldots & m_{mm} \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}, \end{aligned}$$

was die Behauptung zeigt.

2. Wir schreiben $a_i = (a_{i1} \dots a_{in})$ und $b_i = (b_{i1} \dots b_{in})$, bilden die Matrizen

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{m1} & \dots & b_{mn} \end{pmatrix}$$

und benutzen den bereits bewiesenen ersten Teil:

$$\begin{aligned} \Lambda_a = \Lambda_b \quad &\Longrightarrow \quad \Lambda_a \subseteq \Lambda_b \text{ und } \Lambda_b \subseteq \Lambda_a \\ &\Longrightarrow \quad \text{es gibt Matrizen } M, N \in M_m(\mathbf{Z}) \text{ mit } A = MB \text{ und } B = NA. \end{aligned}$$

Die letzte Aussage impliziert $A = MNA$. Da A Rang m hat, gibt es eine $m \times m$-Untermatrix $\widetilde{A}$ von A mit $\det \widetilde{A} \neq 0$. Dann folgt auch $\widetilde{A} = MN\widetilde{A}$ und damit $MN = 1_m$, also $m \in \mathrm{GL}_m(\mathbf{Z})$.
Ist umgekehrt $M \in \mathrm{GL}_m(\mathbf{Z})$ mit $A = MB$, so folgt $B = M^{-1}A$, und mit $M^{-1} \in M_m(\mathbf{Z})$ folgt $\Lambda_a \subseteq \Lambda_b$ und $\Lambda_b \subseteq \Lambda_a$, also $\Lambda_a = \Lambda_b$, was zu zeigen war. ■

Bemerkungen: Sei $\Lambda \subseteq \mathbf{R}^n$ ein Gitter vom Rang m und $a_1, \dots, a_m$ eine Gitterbasis.

1. Der letzte Satz charakterisiert, wie zwei Gitterbasen von Λ auseinanderhervorgehen.
2. Die Anordnung der Elemente $a_1, \dots, a_m$ spielt natürlich keine Rolle. Außerdem kann man a_i durch $a_i + ka_j$ ersetzen, wenn $i \neq j$ ist und $k \in \mathbf{Z}$. Die Formulierung mit Matrizen ist

$$\begin{pmatrix} a_1 \\ \vdots \\ a_j \\ \vdots \\ a_i + ka_j \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & \vdots & \ddots & & & \\ & & k & \dots & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_j \\ \vdots \\ a_i \\ \vdots \\ a_m \end{pmatrix}.$$

3. Im folgenden geht es darum, unter der Vielzahl von Gitterbasen eines Gitters geeignete auszuwählen.

Beispiel: Wir betrachten das Gitter

$$\Lambda = \mathbf{Z}(2,3) + \mathbf{Z}(4,5) \subseteq \mathbf{R}^2.$$

Durch obigen Ersetzungsprozess erhalten wir verschiedene Gitterbasen

$$\Lambda = \mathbf{Z}(2,3) + \mathbf{Z}(4,5) = \mathbf{Z}(2,3) + \mathbf{Z}(0,-1) = \mathbf{Z}(2,0) + \mathbf{Z}(0,1).$$

Die zugehörige Matrizengleichung lautet

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & 5 \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}.$$

3. Gittervolumen - Gitterdeterminante

Sei $\Lambda \subseteq \mathbf{R}^n$ ein Gitter und $a_1, \dots, a_m$ eine Gitterbasis. Zu der Gitterbasis bilden wir mit Hilfe des Skalarprodukts die Gramsche Matrix

$$(a_i \cdot a_j)_{1 \leq i,j \leq m} = \begin{pmatrix} a_1 \cdot a_1 & \dots & a_1 \cdot a_m \\ \vdots & & \vdots \\ a_m \cdot a_1 & \dots & a_m \cdot a_m \end{pmatrix}.$$

Da man das Skalarprodukt $a_i \cdot a_j$ auch durch Matrizenmultiplikation $a_i \cdot a_j = a_i a_j^t$ berechnen kann, kann man auch schreiben

$$(a_i \cdot a_j)_{1 \leq i,j \leq m} = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \begin{pmatrix} a_1^t & \dots & a_m^t \end{pmatrix} = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}^t.$$

Ist $b_1, \ldots, b_m$ eine andere Gitterbasis und $M \in \mathrm{GL}_m(\mathbf{Z})$ die zugehörige Transformationsmatrix, d.h.

$$\begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} = M \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix},$$

so ergibt sich für die Gramsche Matrix bezüglich $b_1, \ldots, b_m$

$$(b_i \cdot b_j)_{1 \le i,j \le m} = M (a_i \cdot a_j)_{1 \le i,j \le m} M^t.$$

Wegen $\det(M) = \pm 1$ liefert Determinantenbildung für die zugehörigen Gramschen Determinanten

$$\det((a_i \cdot a_j)_{1 \le i,j \le m}) = \det((b_i \cdot b_j)_{1 \le i,j \le m}),$$

d.h. die Gramsche Determinante hängt nur vom Gitter, nicht von der gewählten Gitterbasis ab. Wir werden später sehen, daß die Determinante eine positive reelle Zahl ist. Wir definieren nun das Gittervolumen bzw. die Gitterdeterminante durch

$$\mathrm{vol}\,(\Lambda) = \det(\Lambda) = \sqrt{\det((a_i \cdot a_j)_{1 \le i,j \le m})},$$

wobei die Wurzel ≥ 0 gewählt werden soll. Nach dem Gezeigten hängt die Gitterdeterminante bzw. das Gittervolumen nicht von das gewählten Basis ab.

Beispiele:

1. $\Lambda = \mathbf{Z}(1,2,3) \subseteq \mathbf{R}^3$.
$$\det \Lambda = \sqrt{14}.$$
2. $\Lambda = \mathbf{Z}(1,2,3) + \mathbf{Z}(4,5,6) \subseteq \mathbf{R}^3$
$$\det \Lambda = \sqrt{\det \begin{pmatrix} 14 & 32 \\ 32 & 77 \end{pmatrix}} = \sqrt{54}.$$
3. $\Lambda = Z(1,2,3) + \mathbf{Z}(4,5,6) + \mathbf{Z}(7,8,10) \subseteq \mathbf{R}^3$.
$$\det \Lambda = \sqrt{\det \begin{pmatrix} 14 & 32 & 53 \\ 32 & 77 & 128 \\ 53 & 128 & 213 \end{pmatrix}} = 3.$$

LEMMA. *Ist $\Lambda \subseteq \mathbf{R}^n$ ein Gitter vom Rang n und $a_1, \ldots, a_n$ eine Gitterbasis, so gilt*

$$\det \Lambda = |\det \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}|.$$

Beweis: Mit obiger Formel erhält man

$$\det(\Lambda)^2 = \det((a_i \cdot a_j)_{1 \le i,j \le m}) = \det(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}^t) = (\det \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix})^2,$$

was die Behauptung beweist. ■

Ist $\Lambda \subseteq \mathbf{R}^n$ ein Gitter und $a_1, \ldots, a_m$ eine Gitterbasis, so heißt

$$F = \{x_1 a_1 + \cdots + x_m a_m \in \mathbf{R}^n : 0 \le x_i < 1\}$$

ein Fundamentalgebiet - Fundamentalparallelotop - Grundmasche des Gitters (bezüglich der Gitterbasis $a_1, \ldots, a_m$).
Der folgende Satz spielt für die Volumenberechnung im $\mathbf{R}^n$ eine wichtige Rolle:

SATZ. *Ist F Fundamentalparallelotop eines Gitters Λ, so gilt*

$$\mathrm{vol}\,(F) = \det(\Lambda) = \mathrm{vol}\,(\Lambda).$$

4. Gitter als diskrete Untergruppen von Vektorräumen

Im folgenden soll eine alternative Charakterisierung von Gittern gegeben werden.

Eine Teilmenge $M \subseteq \mathbf{R}^n$ nennt man diskret, wenn jede beschränkte Menge, also z.B. jede Kugel $\{x \in \mathbf{R}^n : \|x\| \leq r\}$, nur endlich viele Elemente von M enthält. Dabei ist $\|(x_1, \ldots, x_n)\| = \sqrt{x_1^2 + \cdots + x_n^2}$.

LEMMA. *Sind $v_1, \ldots, v_m \in \mathbf{R}^n$ linear unabhängig, so gibt es eine Konstante $c > 0$ mit*

$$\|x_1 v_1 + \cdots + x_m v_m\| \geq c \max(|x_1|, \ldots, |x_m|) \text{ für alle } x_i \in \mathbf{R}.$$

Beweis: Da die Menge $Q = \{(x_1, \ldots, x_m) : \max(|x_1|, \ldots, |x_m|) = 1\}$ im $\mathbf{R}^m$ kompakt ist, nimmt die stetige Funktion $(x_1, \ldots, x_m) \mapsto \|x_1 v_1 + \cdots + x_m v_m\|$ dort ihr Minimum c an; es ist $c > 0$, da $v_1, \ldots, v_m$ linear unabhängig sind. Sei jetzt $(x_1, \ldots, x_m) \neq 0$ gegeben. Mit $t = \max(|x_1|, \ldots, |x_m|)$ ist $(\frac{x_1}{t}, \ldots, \frac{x_m}{t})$ ein Punkt von Q und somit folgt $\|\frac{x_1}{t} v_1 + \cdots + \frac{x_m}{t} v_m\| \geq c$, was durch Multiplikation mit t die Behauptung liefert. (Der Fall $t = 0$ ist trivial.) ■

LEMMA. *Ist $\Lambda \subseteq \mathbf{R}^n$ ein Gitter, so ist Λ eine diskrete Untergruppe von $\mathbf{R}^n$.*

Beweis: Sei $\Lambda = \mathbf{Z}a_1 + \cdots + \mathbf{Z}a_m$ mit linear unabhängigen $a_1, \ldots, a_m$ gegeben. Natürlich ist Λ eine Untergruppe von $\mathbf{R}^n$. Wir müssen zeigen, daß Λ diskret ist. Nach dem letzten Lemma gibt es ein $c > 0$ mit $\|x_1 a_1 + \cdots + x_m a_m\| \geq c \max(|x_1|, \ldots, |x_m|)$ für alle $x_i \in \mathbf{R}$. Dann ist aber für jedes $H > 0$ die Menge

$$\begin{aligned} \Lambda \cap \{x \in \mathbf{R}^n : \|x\| \leq H\} &= \{x_1 a_1 + \cdots + x_m a_m : x_i \in \mathbf{Z} \text{ und } \|x_1 a_1 + \cdots + x_m a_m\| \leq H\} \\ &\subseteq \{x_1 a_1 + \cdots + x_m a_m : x_i \in \mathbf{Z}, |x_i| \leq \frac{H}{c}\} \end{aligned}$$

endlich, was zeigt, daß Λ diskret ist. ■

LEMMA. *Ist $\Lambda \subseteq \mathbf{R}^n$ eine diskrete Untergruppe, so gibt es $\mathbf{R}$-linear unabhängige $a_1, \ldots, a_m \in \Lambda$ mit*

$$\Lambda = \mathbf{Z}a_1 + \mathbf{Z}a_2 + \cdots + \mathbf{Z}a_m.$$

Beweis: Sei V der von Λ aufgespannte $\mathbf{R}$-Untervektorraum in $\mathbf{R}^n$, d.h. der kleinste $\mathbf{R}$-Untervektorraum von $\mathbf{R}^n$, der Λ enthält. Sei $m = \dim_{\mathbf{R}} V$. Wir machen Induktion nach m. Für $m = 0$ ist nichts zu zeigen. Sei jetzt $m > 0$. Seien $a_1, a_2, \ldots, a_m \in \Lambda$ mit $V = \mathbf{R}a_1 + \mathbf{R}a_2 + \cdots + \mathbf{R}a_m$. Sei $V_0 = \mathbf{R}a_1 + \mathbf{R}a_2 + \cdots + \mathbf{R}a_{m-1}$ und $\Lambda_0 = \Lambda \cap V_0$. Natürlich ist auch Λ_0 eine diskrete Untergruppe von $\mathbf{R}^n$. Wegen $\Lambda_0 \subseteq V_0$ und $\dim_{\mathbf{R}} V_0 = m - 1$ können wir die Induktionsvoraussetzung auf Λ_0 anwenden, d.h. es gibt $\mathbf{R}$-linear unabhängige $b_1, b_2, \ldots, b_{m-1}$ mit

$$\Lambda_0 = \mathbf{Z}b_1 + \mathbf{Z}b_2 + \cdots + \mathbf{Z}b_{m-1}.$$

Wir betrachten jetzt

$$M = \{x_1 b_1 + x_2 b_2 + \cdots + x_{m-1} b_{m-1} + x_m a_m : 0 \leq x_1 < 1, 0 \leq x_2 < 1, \ldots 0 \leq x_{m-1} < 1, 0 < x_m \leq 1\}.$$

M ist eine beschränkte Teilmenge des $\mathbf{R}^n$, enthält also nur endlich viele Elemente von Λ. (Wegen $a_m \in M \cap \Lambda$ ist der Schnitt nicht leer.) Sei $b_m \in M \cap \Lambda$ mit minimalem Koeffizienten x_m bei a_m. Wir schreiben

$$b_m = z_1 b_1 + \cdots + z_{m-1} b_{m-1} + z_m a_m.$$

Wir wollen zeigen, daß $\Lambda = \mathbf{Z}b_1 + \cdots + \mathbf{Z}b_{m-1} + \mathbf{Z}b_m$ gilt, was dann die Behauptung beweist. Sei also $a \in \Lambda$ beliebig. Es gibt $x_1, \ldots, x_m \in \mathbf{R}$ mit

$$a = x_1 b_1 + \cdots + x_{m-1} b_{m-1} + x_m a_m.$$

Indem wir x_m durch z_m teilen erhalten wir eine Darstellung

$$x_m = k z_m + r \quad \text{mit} \quad k \in \mathbf{Z}, r \in \mathbf{R}, 0 \leq r < 1.$$

Es ist

$$a - k b_m = (x_1 - k z_1) b_1 + \cdots + (x_{m-1} - k z_{m-1}) b_{m-1} + r a_m \in \Lambda.$$

Wäre $r > 0$, so könnte man (nach Subtraktion einer ganzzahligen Linearkombination von $b_1, \ldots, b_{m-1}$) sofort ein Element in M finden, das wegen $r < z_m$ der Definition von b_m widerspricht. Daher muß $r = 0$ sein, d.h. $a - k b_m \in \Lambda_0$. Dann gibt es $k_1, \ldots, k_{m-1} \in \mathbf{Z}$ mit

$$a - k b_m = k_1 b_1 + \cdots + k_{m-1} b_{m-1},$$

was $a \in \mathbf{Z}b_1 + \cdots + \mathbf{Z}b_m$ und damit die Behauptung liefert. ■

5. Gram-Schmidt-Orthogonalisierung

Für linear unabhängige Vektoren $b_1, \ldots, b_m$ des $\mathbf{R}^n$ wird das Gram-Schmidt-Orthogonalisierungsverfahren rekursiv durch folgende Formeln definiert:

$$\begin{aligned} b_1^* &= b_1, \\ b_i^* &= b_i - \sum_{j=1}^{i-1} \mu_{ij} b_j^* \text{ für } i = 2, \ldots, m \text{ mit } \mu_{ij} = \frac{b_i \cdot b_j^*}{\|b_j^*\|^2}. \end{aligned}$$

Beispiel: Für $b_1 = (1,2,3)$, $b_2 = (4,5,6)$, $b_3 = (7,8,10)$ berechnet man

$$\mu_{21} = \frac{16}{7}, \quad \mu_{31} = \frac{53}{14}, \quad \mu_{32} = \frac{16}{9}, \quad b_1^* = (1,2,3), \quad b_2^* = (\frac{12}{7}, \frac{3}{7}, -\frac{6}{7}), \quad b_3^* = (\frac{1}{6}, -\frac{1}{3}, \frac{1}{6}).$$

Die wichtigsten Eigenschaften der Gram-Schmidt-Orthogonalisierung stellt folgender Satz zusammen:

SATZ. *Sei $b_1^*, \ldots, b_m^*$ die Gram-Schmidt-Orthogonalisierung der linear unabhängigen Vektoren $b_1, \ldots, b_m$ im $\mathbf{R}^n$ mit $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$. Dann gilt:*

1. *$b_i^* \cdot b_j^* = 0$ für $i \neq j$, d.h. $b_1^*, \ldots, b_m^*$ ist ein Orthogonalsystem.*
2. *$\mathbf{R}b_1 + \cdots + \mathbf{R}b_i = \mathbf{R}b_1^* + \cdots + \mathbf{R}b_i^*$ für $i = 1, \ldots, m$.*
3. *b_i^* ist die Projektion von b_i auf das orthogonale Komplement von $\mathbf{R}b_1 + \cdots + \mathbf{R}b_{i-1} = \mathbf{R}b_1^* + \cdots + \mathbf{R}b_{i-1}^*$.*
4. *Für die Transformationsmatrix gilt*

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ \mu_{21} & 1 & \ldots & 0 \\ \vdots & & \ddots & \vdots \\ \mu_{m1} & \mu_{m2} & \ldots & 1 \end{pmatrix} \begin{pmatrix} b_1^* \\ b_2^* \\ \vdots \\ b_m^* \end{pmatrix}.$$

5. *$\|b_i\|^2 = \|b_i^*\|^2 + \sum_{j<i} \mu_{ij}^2 \|b_j^*\|^2$, insbesondere $\|b_i^*\| \leq \|b_i\|$.*
6. *Für die Gramsche Determinanten gilt*

$$\det((b_i \cdot b_j)_{1 \leq i,j \leq m}) = \det((b_i^* \cdot b_j^*)_{1 \leq i,j \leq m}) = \|b_1^*\|^2 \ldots \|b_m^*\|^2 \leq \|b_1\|^2 \ldots \|b_m\|^2.$$

7. *Ist $\Lambda \subseteq \mathbf{R}^n$ das durch $b_1, \ldots, b_m$ definierte Gitter, so folgt*

$$\det \Lambda = \|b_1^*\| \ldots \|b_m^*\| \leq \|b_1\| \ldots \|b_m\|.$$

Beweis:

1. Sei o.E. $j < i$. Wir zeigen die Behauptung durch Induktion nach i. Für $i = 1$ ist nichts zu zeigen. Für $i = 2$ und $j = 1$ ergibt sich

$$b_2^* \cdot b_1^* = (b_2 - \mu_{21} b_1) \cdot b_1 = b_2 \cdot b_1 - \mu_{21} b_1 \cdot b_1 = b_2 \cdot b_1^* - \mu_{21} b_1^* \cdot b_1^* = 0.$$

   Nun folgt durch Induktion für $i > 2$:

$$b_i^* \cdot b_j^* = (b_i - \mu_{i1} b_1^* - \cdots - \mu_{i,i-1} b_{i-1}^*) \cdot b_j^* = b_i \cdot b_j^* - \mu_{ij} b_j^* \cdot b_j^* = 0.$$

2. Dies folgt unmittelbar aus der Konstruktion durch Induktion.
3. Wegen

$$b_i = b_i^* + (\sum_{j<i} \mu_{ij} b_j^*)$$

   folgt die Aussage sofort aus der Orthogonalität der b_i^*.
4. Dies ist die Matrixformulierung von $b_i = b_i^* + (\sum_{j<i} \mu_{ij} b_j^*)$.
5. Dies folgt aus $b_i = b_i^* + (\sum_{j<i} \mu_{ij} b_j^*)$ durch Quadrieren unter Benutzung der Orthogonalität der b_i^*'s.

6. Nach der Transformationsformel für die Gramsche Matrix und Determinante 1 der Transformationsmatrix folgt das erste Gleichheitszeichen. Das zweite Gleichheitszeichen folgt aus $b_i^* \cdot b_j^* = 0$ für $i \neq j$. Das letzte $\leq$-Zeichen folgt dann aus der zuvor bewiesenen Ungleichung $\|b_i^*\| \leq \|b_i\|$.
7. Dies folgt aus dem zuvor bewiesenen Punkt mit der Definition von $\det \Lambda$. ■

Bemerkung: Man sieht hier nochmals direkt, daß die Gramsche Determinante $\det((b_i \cdot b_j))$ eine positive reelle Zahl ist.

FOLGERUNG (Hadamardsche Determinantenabschätzung). *Für Vektoren $b_1, \ldots, b_n$ des $\mathbf{R}^n$ gilt*

$$|\det \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}| \leq \|b_1\| \cdots \|b_n\|.$$

Beweis: Sind die Vektoren $b_1, \ldots, b_n$ linear abhängig, so ist die linke Seite 0, die Behauptung also trivial. Im andern Fall folgt die Behauptung aus der Charakterisierung von $\det \Lambda$, wenn Λ das von $b_1, \ldots, b_n$ aufgespannte Gitter ist. ■

6. Wie findet man eine 'schöne' Gitterbasis?

Ist ein Gitter $\Lambda \subseteq \mathbf{R}^n$ durch eine Gitterbasis gegeben, wollen wir eine neue Gitterbasis mit 'schönen' Eigenschaften konstruieren. Allerdings ist nicht unmittelbar klar, was das bedeutet und wie man das praktisch machen kann.

Wir erinnern an ein Ergebnis des letzten Abschnitts: Ist $b_1, \ldots, b_m$ eine Gitterbasis von Λ, so gilt

$$\det \Lambda \leq \|b_1\| \ldots \|b_m\|.$$

Eine Existenzaussage macht hier ein Satz von Minkowski:

SATZ (Minkowski). *Sei $\Lambda \subseteq \mathbf{R}^n$ ein Gitter vom Rang m. Sei $a_1 \in \Lambda \setminus \{0\}$ ein Element minimaler Länge, sei $a_2 \in \Lambda \setminus \mathbf{R}a_1$ ein Element minimaler Länge, etc., sei schließlich $a_m \in \Lambda \setminus (\mathbf{R}a_1 + \mathbf{R}a_2 + \cdots + \mathbf{R}a_{m-1})$ ein Element minimaler Länge. Dann gilt:*

$$\|a_1\| \ldots \|a_m\| \leq (\frac{2}{\sqrt{\pi}})^m \Gamma(1 + \frac{m}{2}) \det \Lambda.$$

(Dabei bedeutet $\Gamma(1 + \frac{m}{2})$ die Γ-Funktion.)

Bemerkungen:

1. Leider ist nicht klar, wie man in angemessener Zeit Vektoren $a_1, \ldots, a_m$, wie im Satz beschrieben, konstruieren kann.
2. Vektoren $a_1, \ldots, a_m$, die die Eigenschaften des Satzes erfüllen, müssen keine Basis von Λ bilden, wie nachfolgendes Beispiel zeigt.

Beispiel: Wir betrachten das Gitter $\Lambda \subseteq \mathbf{R}^5$ vom Rang 5, das durch die Vektoren

$$b_1 = (1,0,0,0,0), \quad b_2 = (0,1,0,0,0), \quad b_3 = (0,0,1,0,0), \quad b_4 = (0,0,0,1,0), \quad b_5 = (\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2})$$

definiert wird. Dem Satz von Minkowski folgend wählen wir

$$a_1 = b_1, \quad a_2 = b_2, \quad a_3 = b_3, \quad a_4 = b_4, \quad a_5 = (0,0,0,0,1) = 2b_5 - b_1 - b_2 - b_3 - b_4,$$

wobei man sich überlegen muß, daß die entsprechenden Eigenschaften erfüllt sind. Allerdings ist $a_1, \ldots, a_5$ keine Gitterbasis, da die Übergangsmatrix von $b_1, \ldots, b_5$ zu $a_1, \ldots, a_5$ Determinante 2 hat, also kein Element von $\mathrm{GL}_5(\mathbf{Z})$ ist.

Auf L. Lovasz, H.W. Lenstra und A.K. Lenstra (1982) geht ein Algorithmus (LLL-Algorithmus) zurück, der praktikabel ist und heutzutage eine wichtige Rolle spielt. Um ihn zu erläutern wollen wir im folgenden

Gitterbasen $b_1, \ldots, b_m$ mit Hilfe der zugehörigen Gram-Schmidt-Orthogonalisierung $b_1^*, \ldots, b_m^*$ abändern. Will man kurze Basisvektoren erhalten, so liefert die Formel

$$\|b_i\|^2 = \|b_i^*\|^2 + \sum_{j=1}^{i-1} \mu_{ij}^2 \|b_j^*\|^2$$

einen wichtigen Wegweiser.

7. Basiswechsel unter Beibehaltung der Gram-Schmidt-Orthogonalisierung

LEMMA. *Sei $b_1, \ldots, b_m$ Basis eines Gitters $\Lambda \subseteq \mathbf{R}^n$ und $b_1^*, \ldots, b_m^*$ die zugehörige Gram-Schmidt-Orthogonalisierung. Ersetzt man für ein Indexpaar $i > j$ und $k \in \mathbf{Z}$ den Basisvektor b_i durch $b_i + kb_j$, so bleibt die Gram-Schmidt-Orthogonalisierung gleich.*

Beweis: b_i^* ist die Projektion von b_i auf das orthogonale Komplement von $U = \mathbf{R}b_1 + \cdots + \mathbf{R}b_{i-1} = \mathbf{R}b_1^* + \cdots + \mathbf{R}b_{i-1}^*$. Ändert man b_i um ein Element aus U ab, so ändert sich nichts an b_i^*. ■

Wir wollen nun sehen, was bei derartigen Basiswechseln mit den Größen μ_{ij} passiert.

LEMMA. *Sei $\Lambda \subseteq \mathbf{R}^n$ ein Gitter vom Rang m und $b_1, \ldots, b_m$ eine Gitterbasis, $b_1^*, \ldots, b_m^*$ die zugehörige Gram-Schmidt-Orthogonalisierung und $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$ für $i > j$. Seien $i_0 > j_0$ Indizes. Wir ersetzen b_{i_0} durch*

$$b'_{i_0} = b_{i_0} + kb_{j_0}$$

mit $k \in \mathbf{Z}$, die andern Basisvektoren sollen gleich bleiben. Für die neuen Größen μ'_{ij} gilt:

- *Für $i \neq i_0$ ist $\mu'_{ij} = \mu_{ij}$.*
- *Für $j < j_0$ ist $\mu'_{i_0 j} = \mu_{i_0 j} + k\mu_{j_0 j}$.*
- *Für $j = j_0$ ist $\mu'_{i_0 j_0} = \mu_{i_0 j_0} + k$.*
- *Für $j > j_0$ ist $\mu'_{i_0 j} = \mu_{i_0 j}$.*

Beweis: Bei dem angegebenen Basiswechsel bleibt die Gram-Schmidt-Orthogonalisierung $b_1^*, \ldots, b_m^*$ gleich. In die Formel $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$ für μ_{ij} geht nur b_i und b_j^* ein. Also ändert sich höchstens $\mu_{i_0 j}$ für $j < i_0$:

$$\mu'_{i_0 j} = \frac{(b_{i_0} + kb_{j_0}) \cdot b_j^*}{b_j^* \cdot b_j^*} = \mu_{i_0 j} + k \frac{b_{j_0} \cdot b_j^*}{b_j^* \cdot b_j^*}.$$

Wir unterscheiden drei Fälle:

- $j < j_0$. Hier gilt $\mu'_{i_0 j} = \mu_{i_0 j} + k\mu_{j_0 j}$.
- $j = j_0$. Aus $b_{j_0} \cdot b_{j_0}^* = b_{j_0}^* \cdot b_{j_0}^*$ folgt $\mu'_{i_0 j_0} = \mu_{i_0 j_0} + k$.
- $j > j_0$. Hier ist $\mu'_{i_0 j} = \mu_{i_0 j}$.

Damit folgt die Behauptung. ■

Bemerkung: Die Formel

$$\|b_i\|^2 = \|b_i^*\|^2 + \sum_{j=1}^{i-1} \mu_{ij}^2 \|b_j^*\|^2$$

legt es nahe, Basiswechsel vorzunehmen, bei denen $|\mu_{ij}|$ möglichst klein wird. Bei Basiswechsel

$$b'_{i_0} = b_{i_0} - kb_{j_0}$$

sieht man mit

$$\mu'_{i_0 j_0} = \mu_{i_0 j_0} - k,$$

daß bei Wahl von

$$k = \lfloor \mu_{i_0 j_0} \rceil \in \mathbf{Z},$$

wobei $\lfloor x \rceil$ (eine) x nächstliegende ganze Zahl bezeichnet, die Ungleichung

$$|\mu'_{i_0 j_0}| \leq \frac{1}{2}$$

erreicht werden kann. Allerdings ändern sich dann möglicherweise die Werte von $\mu_{i_0 j}$ für $j < j_0$.

Beispiel: Wir betrachten das Gitter $\Lambda = \mathbf{Z}a_1 + \mathbf{Z}a_2 + \mathbf{Z}a_3 + \mathbf{Z}a_4$ im $\mathbf{R}^4$ mit

$$a_1 = (4, 2, -3, 10), a_2 = (-9, 17, 9, -17), a_3 = (-14, -12, -19, -17), a_4 = (-9, -15, 3, -6)$$

und

$$\|a_1\|^2 = 129, \|a_2\|^2 = 740, \|a_3\|^2 = 990, \|a_4\|^2 = 351.$$

Die Gram-Schmidt-Orthogonalisierung ist

$$\begin{aligned} a_1^* &= (4, 2, -3, 10), \\ a_2^* &= (-\frac{365}{129}, \frac{2591}{129}, \frac{188}{43}, -\frac{203}{129}), \\ a_3^* &= (-\frac{541809}{55859}, \frac{164611}{55859}, -\frac{1166678}{55859}, -\frac{166202}{55859}), \\ a_4^* &= (-\frac{33148116}{5100385}, -\frac{5868396}{5100385}, \frac{11997288}{5100385}, \frac{18032112}{5100385}) \end{aligned}$$

mit

$$\|a_1^*\|^2 = 129, \|a_2^*\|^2 = 433.02, \|a_3^*\|^2 = 547.85, \|a_4^*\|^2 = 61.59.$$

Die zu $a_1, \ldots, a_4$ gehörigen μ-Werte sind

$$\mu_{21} = -1.542, \mu_{31} = -1.496, \mu_{32} = -0.595, \mu_{41} = -1.047, \mu_{42} = -0.585, \mu_{43} = -0.003.$$

Wegen $\mu_{21} = -1.542$ ersetzen wir a_2 durch $a_2 - (-2a_1)$ und erhalten

$$a_1 = (4, 2, -3, 10), a_2 = (-1, 21, 3, 3), a_3 = (-14, -12, -19, -17), a_4 = (-9, -15, 3, -6)$$

mit

$$\|a_1\|^2 = 129, \|a_2\|^2 = 460, \|a_3\|^2 = 990, \|a_4\|^2 = 351$$

und

$$\mu_{21} = 0.457, \mu_{31} = -1.496, \mu_{32} = -0.595, \mu_{41} = -1.047, \mu_{42} = -0.585, \mu_{43} = -0.003.$$

Wegen $\mu_{32} = -0.595$ ersetzen wir a_3 durch $a3 + a2$ und erhalten

$$a_1 = (4, 2, -3, 10), a_2 = (-1, 21, 3, 3), a_3 = (-15, 9, -16, -14), a_4 = (-9, -15, 3, -6)$$

mit

$$\|a_1\|^2 = 129, \|a_2\|^2 = 460, \|a_3\|^2 = 758, \|a_4\|^2 = 351$$

und

$$\mu_{21} = 0.457, \mu_{31} = -1.039, \mu_{32} = -0.405, \mu_{41} = -1.047, \mu_{42} = -0.585, \mu_{43} = -0.003.$$

Wegen $\mu_{31} = -1.039$ ersetzen wir a_3 durch $a3 + a1$ und erhalten

$$a_1 = (4, 2, -3, 10), a_2 = (-1, 21, 3, 3), a_3 = (-11, 11, -19, -4), a_4 = (-9, -15, 3, -6)$$

mit

$$\|a_1\|^2 = 129, \|a_2\|^2 = 460, \|a_3\|^2 = 619, \|a_4\|^2 = 351$$

und

$$\mu_{21} = 0.457, \mu_{31} = -0.039, \mu_{32} = -0.405, \mu_{41} = -1.047, \mu_{42} = -0.585, \mu_{43} = -0.003.$$

Wegen $\mu_{42} = -0.585$ ersetzen wir a_4 durch $a4 + a2$ und erhalten

$$a_1 = (4, 2, -3, 10), a_2 = (-1, 21, 3, 3), a_3 = (-11, 11, -19, -4), a_4 = (-10, 6, 6, -3)$$

mit

$$\|a_1\|^2 = 129, \|a_2\|^2 = 460, \|a_3\|^2 = 619, \|a_4\|^2 = 181$$

und

$$\mu_{21} = 0.457, \mu_{31} = -0.039, \mu_{32} = -0.405, \mu_{41} = -0.589, \mu_{42} = 0.415, \mu_{43} = -0.003.$$

Wegen $\mu_{41} = -0.589$ ersetzen wir a_4 durch $a4 + a1$ und erhalten

$$a_1 = (4, 2, -3, 10), a_2 = (-1, 21, 3, 3), a_3 = (-11, 11, -19, -4), a_4 = (-6, 8, 3, 7)$$

mit

$$\|a_1\|^2 = 129, \|a_2\|^2 = 460, \|a_3\|^2 = 619, \|a_4\|^2 = 158$$

und

$$\mu_{21} = 0.457, \mu_{31} = -0.039, \mu_{32} = -0.405, \mu_{41} = 0.411, \mu_{42} = 0.415, \mu_{43} = -0.003.$$

Jetzt gilt $|\mu_{ij}| \leq \frac{1}{2}$ für alle $i > j$.

Das angegebene Beispiel läßt sich sofort zu einem Verfahren verallgemeinern:

Algorithmus: Gegeben sei ein Gitter $\Lambda \subseteq \mathbf{R}^n$ durch eine Gitterbasis $b_1, \ldots, b_m$ mit Gram-Schmidt-Orthogonalisierung $b_1^*, \ldots, b_m^*$. Die Basis wird so abgewandelt, daß für die Größen $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$ gilt $|\mu_{ij}| \leq \frac{1}{2}$ für alle $i > j$.

- Für $i = 2, \ldots, m$ führen wir folgende Schritte durch:
- Für $j = i-1, i-2, \ldots, 2, 1$ berechnen wir $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$. Ist $|\mu_{ij}| > \frac{1}{2}$, ersetzen wir b_i durch $b_i - \lfloor \mu_{ij} \rceil b_j$.

(Die Reihenfolge, in der die Indizes i, j durchlaufen werden, stellt sicher, daß die erzeugten Ungleichungen $|\mu_{ij}| \leq \frac{1}{2}$ erhalten bleiben.)

8. Vertauschung zweier benachbarter Basisvektoren

Wir wollen sehen, was mit der Gram-Schmidt-Orthogonalisierung passiert, wenn wir zwei benachbarte Basisvektoren b_i und b_{i-1} vertauschen.

LEMMA. *Sei Λ ein Gitter vom Rang m in $\mathbf{R}^n$ und $b_1, \ldots, b_m$ eine Gitterbasis mit zugehöriger Gram-Schmidt-Orthogonalisierung $b_1^*, \ldots, b_m^*$ und Koeffizienten $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$. Sei $b_1', \ldots, b_m'$ die Basis, die durch Vertauschung der benachbarten Vektoren b_i und b_{i-1} entsteht, d.h.*

$$b_{i-1}' = b_i, \quad b_i' = b_{i-1} \quad \text{und} \quad b_j' = b_j \quad \text{für} \quad j \neq i-1, i.$$

Dann gilt:

$$\begin{aligned}
b_j'^* &= b_j^* \quad \text{für} \quad j \neq i-1, i, \\
b_{i-1}'^* &= \mu_{i,i-1} b_{i-1}^* + b_i^*, \\
b_i'^* &= \frac{1}{\|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2} (\|b_i^*\|^2 b_{i-1}^* - \mu_{i,i-1} \|b_{i-1}\|^2 b_i^*), \\
\|b_{i-1}'^*\|^2 &= \|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2, \\
\|b_i'^*\|^2 &= \frac{\|b_{i-1}^*\|^2 \|b_i^*\|^2}{\|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2}.
\end{aligned}$$

Beweis: Es ist klar, daß $b_1^*, \ldots, b_{i-2}^*$ unverändert bleiben. Nun ist

$$b_{i-1}' = b_i = b_i^* + \mu_{i,i-1} b_{i-1}^* + \mu_{i,i-2} b_{i-2}^* + \cdots + \mu_{i1} b_1^*.$$

Damit hat man sofort die gesuchte orthogonale Zerlegung von b_{i-1}' und es folgt

$$b_{i-1}'^* = b_i^* + \mu_{i,i-1} b_{i-1}^*.$$

Wir betrachten das in der Aussage definierte $b_i'^*$. Es ist orthogonal zu allen $b_j'^*$ für $j < i$. Außerdem gilt

$$b_i' \cdot b_i'^* = b_i'^* \cdot b_i'^*.$$

Daher ist das angegebene $b_i'^*$ das richtige. Daß für $j > i$ die Aussage $b_j'^* = b_j^*$ gilt, folgt aus der Charakterisierung der $b_j'^*$ durch die orthogonale Zerlegung der $b_j' = b_j$. ■

Bemerkung: Es gilt

$$\begin{aligned}
\|b_1\|^2 &= \|b_1^*\|^2, \\
\|b_2\|^2 &= \|b_2^*\|^2 + \mu_{21}^2 \|b_1^*\|^2, \\
\|b_3\|^2 &= \|b_3^*\|^2 + \mu_{31}^2 \|b_1^*\|^2 + \mu_{32}^2 \|b_2^*\|^2, \\
&\vdots \\
\|b_m\|^2 &= \|b_m^*\|^2 + \mu_{m1}^2 \|b_1^*\|^2 + \mu_{32}^2 \|b_2^*\|^2 + \cdots + \mu_{m,m-1}^2 \|b_{m-1}^*\|^2.
\end{aligned}$$

Es liegt daher nahe, zu untersuchen, ob eine Vertauschung von b_i und b_{i-1} zu einer Verkleinerung von $\|b_{i-1}^*\|$ führt. Allerdings zeigt die Gleichung

$$\det \Lambda = \|b_1^*\| \ldots \|b_{i-1}^*\| \|b_i^*\| \ldots \|b_m^*\|,$$

daß das Produkt $\|b_{i-1}^*\|\|b_i^*\|$ gleich bleibt. Wir schreiben im folgenden

$$c_{i,i-1} = \frac{\|b_{i-1}^{*\prime}\|^2}{\|b_{i-1}^*\|^2} = \frac{\|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2}{\|b_{i-1}^*\|^2} = \frac{\|b_i^*\|^2 + \mu_{i,i-1}^2 \|b_{i-1}^*\|^2}{\|b_{i-1}^*\|^2}.$$

Dabei gilt: Bei der Vertauschung von b_i und b_{i-1} ändert sich $\|b_{i-1}^*\|^2$ um den Faktor $c_{i,i-1}$.

Beispiel: Wir betrachten das Gitter $\Lambda \subseteq \mathbf{R}^4$ vom Rang 4, das durch folgende Basisvektoren gegeben wird:

$$(4,2,-3,10), (-9,17,9,-17), (-14,-12,-19,-17), (-9,-15,3,-6).$$

Normalisierung ($|\mu_{ij}| \leq \frac{1}{2}$) führt zu

$$(4,2,-3,10), (-1,21,3,3), (-11,11,-19,-4), (-6,8,3,7)$$

mit $c_{21} = 3.565891$, $c_{32} = 1.429062$, $c_{43} = .112440$. Vertauschung der Vektoren Nr. 3 und 4 und Normalisierung führt zu

$$(4,2,-3,10), (-1,21,3,3), (-6,8,3,7), (-11,11,-19,-4)$$

mit $c_{21} = 3.565891$, $c_{32} = .314596$, $c_{43} = 8.893606$. Vertauschung der Vektoren Nr. 2 und 3 und Normalisierung führt zu

$$(4,2,-3,10), (-6,8,3,7), (5,13,0,-4), (-5,3,-22,-11)$$

mit $c_{21} = 1.224806$, $c_{32} = 1.539521$, $c_{43} = 2.970995$.

Beispiel: Wir betrachten $b_1 = (1,0)$, $b_2 = (a,b)$. Dann ist $b_1^* = b_1 = (1,0)$ und

$$\mu_{21} = \frac{b_2 \cdot b_1^*}{b_1^* \cdot b_1^*} = a, \quad b_2^* = b_2 - \mu_{21} b_1^* = (a,b) - a(1,0) = (0,b).$$

Durch Normalisierung können wir $-\frac{1}{2} \leq a \leq \frac{1}{2}$ erreichen. Nun ist

$$\|b_1'^*\|^2 = \|b_2^* + \mu_{21} b_1^*\|^2 = \|(0,b) + a(1,0)\|^2 = a^2 + b^2 \quad \text{und} \quad c_{21} = \frac{\|b_1'^*\|^2}{\|b_1^*\|^2} = a^2 + b^2.$$

9. Die grundlegende Reduktionsidee

Gegeben sei ein Gitter Λ im $\mathbf{R}^n$ mit Gitterbasis $b_1, \ldots, b_m$. Wir wollen die Basis $b_1, \ldots, b_m$ in eine 'schönere' Basis abändern.

- Wir starten mit dem Fall $i = 2$.
- Für $j = i-1, i-2, \ldots, 2, 1$ führen wir folgende Schritte aus: Ist $|\mu_{ij}| > \frac{1}{2}$, ersetzen wir b_i durch $b_i - \lfloor \mu_{ij} \rceil b_j$, so daß nun $|\mu_{ij}| \leq \frac{1}{2}$ gilt.
- Wir testen nun, was Vertauschen von b_i und b_{i-1} bringen würde, d.h. wir vergleichen $\|b_{i-1}^*\|^2$ mit $\|b_{i-1}'^*\|^2$. Die naheliegendste Bedingung wäre, daß wir b_i mit b_{i-1} vertauschen, falls

  $$\|b_{i-1}'^*\|^2 < \|b_{i-1}^*\|^2$$

  gilt. Allerdings kann man in diesem Fall die Laufzeit nicht gut abschätzen. Wir wählen daher etwas allgemeiner eine Konstante $c \leq 1$ (für die Theorie dann $\frac{1}{4} < c < 1$) mit der Funktion: Ist

  $$\|b_{i-1}'^*\|^2 < c\|b_{i-1}^*\|^2,$$

  so vertauschen wir b_i und b_{i-1} und beginnen wieder von vorne mit $i-1$ statt i. Andernfalls sind wir zufrieden und beginnen von vorne mit $i+1$ statt i bzw. beenden das Verfahren im Fall $i = m$.

- Die obige Bedingung

$$\|{b'_{i-1}}^*\|^2 < c\|b^*_{i-1}\|^2$$

kann auch als

$$\|b^*_i + \mu_{i,i-1}b^*_{i-1}\|^2 < c\|b^*_{i-1}\|^2$$

bzw. als

$$\|b^*_i\|^2 < (c - \mu^2_{i,i-1})\|b^*_{i-1}\|^2$$

formuliert werden.

Beispiel: Sei Λ das durch die Vektoren

$$(20, 26, -13, 32), (-21, 6, -8, -3), (-29, -9, 35, -15), (-35, 47, 10, -11)$$

definierte Gitter im $\mathbf{R}^4$. Normalisierung führt zu

$$(20, 26, -13, 32), (-21, 6, -8, -3), (-9, 17, 22, 17), (16, 18, 4, -22)$$

mit $c_{21} = .242398$, $c_{32} = 1.964682$, $c_{43} = 1.015114$. Vertauschung der Vektoren Nr. 1 und 2 und Normalisierung führt zu

$$(-21, 6, -8, -3), (20, 26, -13, 32), (-9, 17, 22, 17), (16, 18, 4, -22)$$

mit $c_{21} = 4.125455$, $c_{32} = .528202$, $c_{43} = 1.015114$. Vertauschung der Vektoren Nr. 2 und 3 und Normalisierung führt zu

$$(-21, 6, -8, -3), (-9, 17, 22, 17), (20, 26, -13, 32), (16, 18, 4, -22)$$

mit $c_{21} = 2.078182$, $c_{32} = 1.893213$, $c_{43} = .532215$. Vertauschung der Vektoren Nr. 3 und 4 und Normalisierung führt zu

$$(-21, 6, -8, -3), (-9, 17, 22, 17), (16, 18, 4, -22), (20, 26, -13, 32)$$

mit $c_{21} = 2.078182$, $c_{32} = .890818$, $c_{43} = 1.878938$. Vertauschung der Vektoren Nr. 2 und 3 und Normalisierung führt zu

$$(-21, 6, -8, -3), (16, 18, 4, -22), (-9, 17, 22, 17), (20, 26, -13, 32)$$

mit $c_{21} = 1.963636$, $c_{32} = 1.122564$, $c_{43} = 1.907336$.

Beispiel: $\Lambda \subseteq \mathbf{R}^3$ sei durch die Basisvektoren

$$(2, 13, 17), (3, 11, 19), (5, 7, 29)$$

gegeben. Normalisierung führt zu

$$(2, 13, 17), (1, -2, 2), (-1, 2, 4)$$

mit $c_{21} = .019481$, $c_{32} = .305076$. Vertauschung der Vektoren Nr. 1 und 2 und Normalisierung führt zu

$$(1, -2, 2), (1, 15, 15), (-1, 2, 4)$$

mit $c_{21} = 50.111111$, $c_{32} = .044357$. Vertauschung der Vektoren Nr. 2 und 3 und Normalisierung führt zu

$$(1, -2, 2), (-1, 2, 4), (6, 5, 1)$$

mit $c_{21} = 2.333333$, $c_{32} = 3.077778$.

10. Der LLL-Algorithmus

Die obige Reduktionsidee liefert den auf Lovász, Lenstra, Lenstra zurückgehenden Algorithmus. Wir stellen hier eine erste Version vor.

LLL-Algorithmus: Gegeben sei ein Gitter Λ im $\mathbf{R}^n$ durch eine Gitterbasis $b_1, \ldots, b_m$. Wir wählen eine Konstante c (mit der Funktion, daß wir b_i mit b_{i-1} vertauschen, falls $c_{i,i-1} < c$ ist). Oft wird $c = \frac{3}{4}$ gewählt.

1. Setze $i := 1$.
2. Ist $i > m$ beende das Verfahren, ansonsten führe folgende Schritte aus:
   (a) Ist $i = 1$, setze $b_1^* = b_1$ und $i := 2$.
   (b) Setze $b_i^* := 0$.
   (c) Für $j = i-1, i-2, \ldots, 1$:
      (i) $\mu_{ij} := \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$.
      (ii) $k := \lfloor \mu_{ij} \rceil$.
      (iii) Ist $k \neq 0$, setze $b_i := b_i - k b_j$, $\mu_{ij} := \mu_{ij} - k$.
      (iv) $b_i^* := b_i^* - \mu_{ij} b_j^*$.
   (d) $b_i^* := b_i + b_i^*$.
   (e) $c_{i,i-1} := \frac{b_i^* \cdot b_i^*}{b_{i-1}^* \cdot b_{i-1}^*} + \mu_{i,i-1}^2$.
   (f) Ist $c_{i,i-1} < c$, vertausche b_i und b_{i-1}, setze $i := i - 1$. Ansonsten setze $i := i + 1$.

Bemerkungen:

1. Die Maple-Funktion 'lattice' führt obiges Verfahren aus. (Mit 'infolevel[lattice]:=3' kann man einzelne Schritte des Verfahrens sehen.)
2. Wir haben eine Maple-Funktion 'lll' geschrieben, die nach dem dargestellten Verfahren funktioniert.
3. In der C++-Bibliothek NTL gibt es eine Funktion 'LLL', die obige Gitterbasenreduktion durchführt.

Beispiel: Wir betrachten das Gitter $\Lambda \subseteq \mathbf{R}^3$, das durch die Basisvektoren $(1,2,3), (4,5,6), (7,8,10)$ gegeben wird.

1. Maple führt folgende Operationen aus:

```
                    a := [[1, 2, 3], [4, 5, 6], [7, 8, 10]]
> lattice(a);
lattice:   starting lattice reduction at time   .29e-1
lattice:   performing reductions using rational arithmetic
lattice/newmus:   Replacing v2   by   v2-2*v1
lattice/newmus:   v2 = [2, 1, 0]
lattice:   Interchanging v1 and v2
lattice/newmus:   Replacing v2   by   v2-v1
lattice/newmus:   v2 = [-1, 1, 3]
lattice/newmus:   Replacing v3   by   v3-3*v2
lattice/newmus:   v3 = [10, 5, 1]
lattice:   Interchanging v2 and v3
lattice/newmus:   Replacing v2   by   v2-5*v1
lattice/newmus:   v2 = [0, 0, 1]
lattice:   Interchanging v1 and v2
lattice:   Interchanging v2 and v3
lattice/newmus:   Replacing v2   by   v2-3*v1
lattice/newmus:   v2 = [-1, 1, 0]
lattice:   finishing lattice reduction at time   .50e-1
                        [[0, 0, 1], [-1, 1, 0], [2, 1, 0]]
```

2. Das gleiche Beispiel nun mit dem eigens geschriebenen Programm:

```
                    a := [[1, 2, 3], [4, 5, 6], [7, 8, 10]]

i=1: bb[1]=[1, 2, 3]
i=2:
mu_21=2.285714
b[2] wird ersetzt durch b[2]-(2)*b[1].
bb[2]=[12/7, 3/7, -6/7]
c_21=.357143
Tausche Vektoren 2 und 1
b[1]=[2, 1, 0] b[2]=[1, 2, 3]
i=1: bb[1]=[2, 1, 0]
i=2:
mu_21=.800000
b[2] wird ersetzt durch b[2]-(1)*b[1].
bb[2]=[-3/5, 6/5, 3]
c_21=2.200000
i=3:
mu_32=3.277778
b[3] wird ersetzt durch b[3]-(3)*b[2].
mu_31=5.000000
b[3] wird ersetzt durch b[3]-(5)*b[1].
bb[3]=[1/6, -1/3, 1/6]
c_32=.092593
Tausche Vektoren 3 und 2
b[2]=[0, 0, 1] b[3]=[-1, 1, 3]
i=2:
mu_21=0.000000
bb[2]=[0, 0, 1]
c_21=.200000
Tausche Vektoren 2 und 1
b[1]=[0, 0, 1] b[2]=[2, 1, 0]
i=1: bb[1]=[0, 0, 1]
i=2:
mu_21=0.000000
bb[2]=[2, 1, 0]
c_21=5.000000
i=3:
mu_32=-.200000
mu_31=3.000000
b[3] wird ersetzt durch b[3]-(3)*b[1].
bb[3]=[-3/5, 6/5, 0]
c_32=.400000
Tausche Vektoren 3 und 2
b[2]=[-1, 1, 0] b[3]=[2, 1, 0]
i=2:
mu_21=0.000000
bb[2]=[-1, 1, 0]
c_21=2.000000
i=3:
mu_32=-.500000
mu_31=0.000000
bb[3]=[3/2, 3/2, 0]
c_32=2.500000
                        [[0, 0, 1], [-1, 1, 0], [2, 1, 0]]
```

11. LLL-reduzierte Gitterbasen

Eine Gitterbasis wird nun LLL-reduziert genannt, wenn der oben dargestellte Reduktionsalgorithmus die Basis unverändert läßt. Formal läßt sich dies dann so ausdrücken:

DEFINITION. Sei $b_1, \ldots, b_m$ Basis eines Gitters Λ im $\mathbf{R}^n$, $b_1^*, \ldots, b_m^*$ die zugehörige Gram-Schmidt-Orthogonalisierung und $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$. Die Gitterbasis $b_1, \ldots, b_m$ heißt LLL-reduziert (bzgl. einer Konstanten c), wenn gilt

$$|\mu_{ij}| \leq \frac{1}{2} \text{ für alle } i > j \text{ und } \|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2 \geq c\|b_{i-1}^*\|^2,$$

wobei die zweite Bedingung auch als

$$\|b_i^*\|^2 \geq (c - \mu_{i,i-1}^2)\|b_{i-1}^*\|^2$$

formuliert werden kann und als Lovász-Bedingung bezeichnet wird. Wird über die Konstante c nichts gesagt, so meint man meist $c = \frac{3}{4}$.

Wir stellen nun einige wichtige Eigenschaften LLL-reduzierter Basen zusammen.

LEMMA. *$\Lambda \subseteq \mathbf{R}^n$ sei ein Gitter mit LLL-reduzierter Gitterbasis $b_1, \ldots, b_m$ und Gram-Schmidt-Orthogonalisierung $b_1^*, \ldots, b_m^*$. Dann gilt:*

$$\|b_j^*\| \leq 2^{\frac{i-j}{2}} \|b_i^*\| \quad \textit{für} \quad 1 \leq j \leq i \leq m$$

und

$$\|b_j\| \leq 2^{\frac{i-1}{2}} \|b_i^*\| \quad \textit{für} \quad 1 \leq j \leq i \leq m.$$

Beweis: Wegen $|\mu_{i,i-1}| \leq \frac{1}{2}$ folgt aus der Lovász-Bedingung

$$\|b_i^*\|^2 \geq (\frac{3}{4} - \mu_{i,i-1}^2)\|b_{i-1}^*\|^2 \geq \frac{1}{2}\|b_{i-1}^*\|^2, \quad \text{also} \quad \|b_{i-1}^*\|^2 \leq 2\|b_i^*\|,$$

was durch Induktion sofort

$$\|b_j^*\|^2 \leq 2^{i-j}\|b_i^*\|^2 \text{ für } j \leq i$$

und damit die erste Behauptung zeigt. Mit $|\mu_{ij}| \leq \frac{1}{2}$ ergibt sich dann damit

$$\begin{aligned}
\|b_i\|^2 &= \|b_i^*\|^2 + \mu_{i,i-1}^2\|b_{i-1}^*\|^2 + \cdots + \mu_{i,1}^2\|b_1^*\|^2 \leq \|b_i^*\|^2 + \frac{1}{4}\|b_{i-1}^*\|^2 + \cdots + \frac{1}{4}\|b_1^*\|^2 \leq \\
&\leq \left[1 + \frac{1}{4}(2 + 2^2 + \cdots + 2^{i-1})\right]\|b_i^*\|^2 = \left[1 + \frac{1}{2}(1 + 2 + 2^2 \cdots + 2^{i-2})\right]\|b_i^*\|^2 = \\
&= \left[1 + \frac{1}{2}(2^{i-1} - 1)\right]\|b_i^*\|^2 = \frac{2^{i-1}+1}{2}\|b_i^*\|^2 \leq 2^{i-1}\|b_i^*\|^2.
\end{aligned}$$

Für $j \leq i$ folgt

$$\|b_j\|^2 \leq 2^{j-1}\|b_j^*\|^2 \leq 2^{j-1} \cdot 2^{i-j}\|b_i^*\|^2 = 2^{i-1}\|b_i^*\|^2,$$

was auch die zweite Aussage beweist. ■

SATZ. *Sei $b_1, \ldots, b_m$ eine LLL-reduzierte Basis des Gitters Λ. Dann gilt:*

1.
$$\det(\Lambda) \leq \prod_{i=1}^{m} \|b_i\| \leq 2^{\frac{m(m-1)}{4}} \det(\Lambda).$$

2.
$$\|b_1\| \leq 2^{\frac{m-1}{4}} \det(\Lambda)^{\frac{1}{m}}.$$

3. *Sind $x_1, \ldots, x_t \in \Lambda$ linear unabhängig, so gilt*
$$\|b_1\|, \|b_2\|, \ldots, \|b_t\| \leq 2^{\frac{m-1}{2}} \max(\|x_1\|, \|x_2\|, \ldots, \|x_t\|).$$

4. *Für jedes* $x \in \Lambda \setminus \{0\}$ *gilt*

$$\|b_1\| \leq 2^{\frac{m-1}{2}} \|x\|.$$

Beweis:

1. Wir benutzen die Aussage $\|b_i\| \leq 2^{\frac{i-1}{2}} \|b_i^*\|$ des letzten Lemmas und erhalten mit den früher bewiesenen Aussagen über $\det \Lambda$

$$\det \Lambda = \prod_{i=1}^{m} \|b_i^*\| \leq \prod_{i=1}^{m} \|b_i\| \leq \prod_{i=1}^{m} 2^{\frac{i-1}{2}} \|b_i^*\| = 2^{\frac{m(m-1)}{4}} \prod_{i=1}^{m} \|b_i^*\| = 2^{\frac{m(m-1)}{4}} \det \Lambda.$$

2. Wir verwenden die Abschätzung $\|b_1\| \leq 2^{\frac{i-1}{2}} \|b_i^*\|$ des letzten Lemmas und erhalten

$$\|b_1\|^m \leq \prod_{i=1}^{m} 2^{\frac{i-1}{2}} \|b_i^*\| = 2^{\frac{m(m-1)}{4}} \prod_{i=1}^{m} \|b_i^*\| = 2^{\frac{m(m-1)}{4}} \det \Lambda,$$

   was durch Wurzelziehen die Behauptung zeigt.
3. Wir wählen k minimal mit $x_1, \ldots, x_t \in \mathbf{R}b_1 + \cdots + \mathbf{R}b_k$, was natürlich insbesondere $k \geq t$ impliziert. Also gibt es $r_{ij} \in \mathbf{Z}$ mit

$$x_i = \sum_{j=1}^{k} r_{ij} b_j.$$

   Mit $b_j = b_j^* + \mu_{j,j-1} b_{j-1}^* + \cdots + \mu_{j1} b_1^*$ erhalten wir eine Darstellung

$$x_i = \sum_{j=1}^{k} s_{ij} b_j^* \text{ mit } s_{ij} \in \mathbf{R} \text{ und } r_{ik} = s_{ik}.$$

   Wähle i mit $r_{ik} \neq 0$. Dann ist

$$\|x_i\|^2 = \sum_{j=1}^{k} s_{ij}^2 \|b_j^*\|^2 \geq s_{ik}^2 \|b_k^*\|^2 = r_{ik}^2 \|b_k^*\|^2 \geq \|b_k^*\|^2.$$

   Daher folgt für $j \leq k$ (und damit auch für $j \leq t$)

$$\|b_j\|^2 \leq 2^{k-1} \|b_k^*\|^2 \leq 2^{k-1} \|x_i\|^2 \leq 2^{m-1} \max(\|x_1\|^2, \ldots, \|x_t\|^2),$$

   die Behauptung beweist.
4. Dies ist ein Spezialfall der letzten Aussage. ■

Der folgende Satz soll zeigen, daß die Abschätzungen des letztes Satzes sich vom Charakter her nicht ändern, wenn man die Lovász-Bedingung $c = \frac{3}{4}$ durch $c = 1$ ersetzt.

Satz. *Sei* Λ *ein Gitter im* $\mathbf{R}^n$ *vom Rang* m *und* $b_1, \ldots, b_m$ *eine LLL-reduzierte Gitterbasis, die der Lovász-Bedingung mit* $c = 1$ *genügt. Dann gilt:*

$$\|b_1\| \leq (\frac{4}{3})^{\frac{m-1}{4}} (\det \Lambda)^{\frac{1}{m}}$$

Beweis: Die Lovász-Bedingung für $c = 1$ ist

$$\|b_i^*\|^2 \geq (1 - \mu_{i,i-1}^2) \|b_{i-1}^*\|^2.$$

Wegen $|\mu_{i,i-1}| \leq \frac{1}{2}$ liefert dies

$$\|b_i^*\|^2 \geq \frac{3}{4} \|b_{i-1}^*\|^2.$$

Durch Induktion ergibt sich mit $b_1 = b_1^*$

$$\|b_i^*\|^2 \geq (\frac{3}{4})^{i-1} \|b_1\|^2.$$

Es folgt

$$\begin{aligned}(\det\Lambda)^2 &= \prod_{i=1}^{m}\|b_i^*\|^2 \geq \prod_{i=1}^{m}[(\frac{3}{4})^{i-1}\|b_1\|^2] = (\frac{3}{4})^{\sum_{i=1}^{m}(i-1)}\|b_1\|^{2m} = (\frac{3}{4})^{\sum_{i=0}^{m-1} i}\|b_1\|^{2m} = \\ &= (\frac{3}{4})^{\frac{(m-1)m}{2}}\|b_1\|^{2m},\end{aligned}$$

was die Abschätzung

$$\|b_1\| \leq (\frac{4}{3})^{\frac{m-1}{4}}(\det\Lambda)^{\frac{1}{m}}$$

ergibt. ■

Das folgende Beispiel zeigt einen Fall, für den die Abschätzung des letztes Satzes bestmöglich ist.

Beispiel: Wir betrachten das Gitter $\Lambda = \mathbf{Z}b_1 + \mathbf{Z}b_2$ mit

$$b_1 = (1,0) \quad \text{und} \quad b_2 = (\frac{1}{2}, \frac{1}{2}\sqrt{3}).$$

Es ist

$$\det\Lambda = \frac{\sqrt{3}}{2} = (\frac{3}{4})^{\frac{1}{2}}$$

und

$$b_1^* = (1,0), \quad b_2^* = (0, \frac{1}{2}\sqrt{3}), \quad \mu_{21} = \frac{1}{2}, \quad \|b_1'^*\|^2 = \|b_2^* + \mu_{21}b_1^*\|^2 = 1 = \|b_1^*\|^2,$$

so daß die Lovaász-Bedingung mit $c = 1$ erfüllt ist. Die rechte Seite obiger Abschätzung ist

$$(\frac{4}{3})^{\frac{1}{4}}(\det\Lambda)^{\frac{1}{2}} = 1,$$

so daß wegen $\|b_1\| = 1$ für die Abschätzung des Satzes das Gleichheitszeichen gilt.

Der folgende Satz zeigt, wie man die Länge des 2. Vektors einer reduzierten Gitterbasis abschätzen kann, wenn man $\Lambda \subseteq \mathbf{Z}^n$ voraussetzt:

SATZ. *Sei $b_1, \ldots, b_m$ eine LLL-reduzierte Basis eines Gitters $\Lambda \subseteq \mathbf{R}^n$ und $b_i \in \mathbf{Z}^n$. Dann gilt*

$$||b_2|| \leq 2^{\frac{m}{4}}\det(\Lambda)^{\frac{1}{m-1}}.$$

Beweis: Wegen $b_i \in \mathbf{Z}^n$ folgt $b_i \cdot b_j \in \mathbf{Z}$ und damit $\det(\Lambda)^2 = \det((b_i \cdot b_j)) \in \mathbf{Z}$, also $\det(\Lambda) \geq 1$ und ebenso $\|b_1^*\| = \|b_1\| \geq 1$.
Aus der Abschätzung $\|b_i^*\| \geq \frac{1}{\sqrt{2}}\|b_{i-1}^*\|$ erhält man durch Induktion

$$\|b_i^*\| \geq (\frac{1}{\sqrt{2}})^{i-2}\|b_2^*\| \text{ für } i \geq 2.$$

Damit ergibt sich mit $\|b_1^*\| \geq 1$:

$$\det(\Lambda) = \prod_{i=1}^{m}\|b_i^*\| \geq \prod_{i=2}^{m}\|b_i^*\| \geq (\frac{1}{\sqrt{2}})^{\sum_{i=2}^{m}(i-2)}\|b_2^*\|^{m-1} = 2^{-\frac{(m-2)(m-1)}{4}}\|b_2^*\|^{m-1},$$

also

$$\|b_2^*\|^2 \leq 2^{\frac{m-2}{2}}\det(\Lambda)^{\frac{2}{m-1}}.$$

Mit $b_2 = b_2^* + \mu_{2,1}b_1^*$, $|\mu_{2,1}| \leq \frac{1}{2}$, $b_1^* = b_1$, $b_1^* \cdot b_2^* = 0$, $\|b_1\|^2 \leq 2^{\frac{m-1}{2}}\det(\Lambda)^{\frac{2}{m}}$ (allgemeine Abschätzung für $\|b_1\|$), $\det(\Lambda) \geq 1$ folgt nach Pythagoras:

$$\begin{aligned}\|b_2\|^2 &= \|b_2^*\|^2 + |\mu_{2,1}|^2\|b_1^*\|^2 \leq \|b_2^*\|^2 + \frac{1}{4}\|b_1\|^2 \leq \\ &\leq 2^{\frac{m-2}{2}}\det(\Lambda)^{\frac{2}{m-1}} + \frac{1}{4}\cdot 2^{\frac{m-1}{2}}\det(\Lambda)^{\frac{2}{m}} \leq (2^{\frac{m}{2}-1} + 2^{\frac{m}{2}-\frac{1}{2}-2})\det(\Lambda)^{\frac{2}{m-1}} \leq \\ &\leq (2^{\frac{m}{2}-1} + 2^{\frac{m}{2}-1})\det(\Lambda)^{\frac{2}{m-1}} = 2\cdot 2^{\frac{m}{2}-1}\det(\Lambda)^{\frac{2}{m-1}} = 2^{\frac{m}{2}}\det(\Lambda)^{\frac{2}{m-1}},\end{aligned}$$

was die Abschätzung

$$\|b_2\| \leq 2^{\frac{m}{4}}\det(\Lambda)^{\frac{1}{m-1}}$$

liefert. ■

12. Warum funktioniert der LLL-Algorithmus?

Gegeben sei ein Gitter $\Lambda \subseteq \mathbf{R}^n$ durch eine Gitterbasis $b_1, \ldots, b_m$. Dazu haben wir die Gram-Schmidt-Orthogonalisierung $b_1^*, \ldots, b_m^*$ und die Koeffizienten $\mu_{ij} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$.
Beim LLL-Algorithmus passiert folgendes:

1. Zu Beginn des Schrittes mit Index i ist $b_1, \ldots, b_{i-1}$ bereits LLL-reduziert, d.h.
$$|\mu_{21}| \leq \frac{1}{2}, \quad |\mu_{32}| \leq \frac{1}{2}, |\mu_{31}| \leq \frac{1}{2}, \quad \ldots, \quad |\mu_{i-1,i-2}| \leq \frac{1}{2}, \ldots, |\mu_{i-1,1}| \leq \frac{1}{2}$$
und
$$c_{21} \geq c, \quad c_{32} \geq c, \quad \ldots, \quad c_{i-1,i-2} \geq c,$$
mit
$$c_{j,j-1} = \frac{\|{b'_{j-1}}^*\|^2}{\|b_{j-1}^*\|^2} = \frac{\|b_j^* + \mu_{j,j-1} b_{j-1}^*\|^2}{\|b_{j-1}^*\|^2}.$$
(Zu Beginn ist $i = 2$ und natürlich ist b_1 LLL-reduziert.)
2. Zunächst wird nun b_i so abgeändert, daß
$$|\mu_{i,i-1}| \leq \frac{1}{2}, \quad |\mu_{i,i-2}| \leq \frac{1}{2}, \quad |\mu_{i,2}| \leq \frac{1}{2}, \quad |\mu_{i,1}| \leq \frac{1}{2}$$
gilt. (Die anderen μ_{jk} ändern sich dabei nicht.)
3. Nun berechnet man
$$c_{i,i-1} = \frac{\|{b'_{i-1}}^*\|^2}{\|b_{i-1}^*\|^2} = \frac{\|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2}{\|b_{i-1}^*\|^2}.$$
($c_{i,i-1}$ ist der Faktor, um den sich $\|b_{i-1}^*\|^2$ ändern würde bei Vertauschung von b_i und b_{i-1}.)
4. Ist $c_{i,i-1} \geq c$, so ist $b_1, \ldots, b_{i-1}, b_i$ LLL-reduziert und man fängt mit $i+1$ statt i von vorne an.
5. Ist $c_{i,i-1} < c$, so ist $b_1, \ldots, b_{i-1}, b_i$ nicht LLL-reduziert. Daher vertauscht man b_i und b_{i-1}. Man weiß nun nur noch, daß $b_1, \ldots, b_{i-2}$ LLL-reduziert ist. Daher fängt man mit $i-1$ statt i von vorne an.
6. Somit ist klar, daß der LLL-Algorithmus eine LLL-reduzierte Basis liefert, falls der Algorithmus nach endlich vielen Schritten fertig ist. Also bleibt die Frage, weshalb der Algorithmus nach endlich vielen Schritten zum Ende kommt.

Wir definieren
$$d_i = (\det(\mathbf{Z}b_1 + \mathbf{Z}b_2 + \cdots + \mathbf{Z}b_i))^2 = \det \begin{pmatrix} b_1 \cdot b_1 & \ldots & b_1 \cdot b_i \\ \vdots & & \vdots \\ b_i \cdot b_1 & \ldots & b_i \cdot b_i \end{pmatrix} \quad \text{für} \quad i = 1, \ldots, m.$$

Es gilt
$$d_i = \det \begin{pmatrix} b_1^* \cdot b_1^* & \ldots & b_1^* \cdot b_i^* \\ \vdots & & \vdots \\ b_i^* \cdot b_1^* & \ldots & b_i^* \cdot b_i^* \end{pmatrix} = \|b_1^*\|^2 \|b_2^*\|^2 \ldots \|b_i^*\|^2.$$

Was passiert mit den reellen Zahlen $d_1, d_2, \ldots, d_{m-1}, d_m$ während des LLL-Reduktionsprozesses?

- Wird b_i ersetzt durch $b_i - \lfloor \mu_{ij} \rceil b_j$ für ein Indexpaar $i > j$, so ändert sich die Gram-Schmidt-Orthogonalisierung $b_1^*, \ldots, b_m^*$ nicht, d.h. $d_1, d_2, \ldots, d_m$ bleiben gleich.
- Vertauscht man b_i und b_{i-1}, so bleiben bei der Gram-Schmidt-Orthogonalisierung die Ausdrücke
$$\|b_1^*\|, \quad \|b_2^*\|, \quad \ldots \|b_{i-2}\|, \quad \text{das Produkt } \|b_{i-1}^*\|\|b_i^*\|, \quad \|b_{i+1}^*\|, \quad \ldots, \|b_m^*\|$$
gleich, und somit verändern sich auch
$$d_1, \quad d_2, \ldots, d_{i-2}, \quad d_i, \quad d_{i+1}, \quad \ldots d_m$$
nicht. Es ändert sich nur d_{i-1} und zwar in
$$d'_{i-1} = \|b_1^*\|^2 \ldots \|b_{i-2}^*\|^2 \|{b_{i-1}^*}'\|^2 = \|b_1^*\|^2 \ldots \|b_{i-2}^*\|^2 \cdot c_{i,i-1} \|b_{i-1}^*\|^2 = c_{i,i-1} d_{i-1}.$$

Die Basisvektoren b_i und b_{i-1} werden nur vertauscht, wenn $c_{i,i-1} < c$ gilt, also gilt dann

$$d'_{i-1} < cd_{i-1}.$$

- Wegen $d_m = (\det \Lambda)^2$ bleibt natürlich d_m immer gleich.

Beispiel: Wir starten mit der Gitter basis $(1,2,3),(4,5,6),(7,8,10)$. Bei der LLL-Reduktion ergeben sich folgende Werte für (d_1,d_2,d_3):

$$(14,54,9),(5,54,9),(5,5,9),(1,5,9),(1,2,9).$$

Man erhält als LLL-reduzierte Basis $(0,0,1),(-1,1,0),(2,1,0)$.

Wir betrachten jetzt den Fall $\Lambda \subseteq \mathbf{Z}^n \subseteq \mathbf{R}^n$, d.h. die Basisvektoren $b_1,\ldots,b_m$ haben ganzzahlige Einträge.

- Dann sind die Skalarprodukte $b_i \cdot b_j \in \mathbf{Z}$, also ebenso $d_1,\ldots,d_m \in \mathbf{Z}$. Wegen $d_i > 0$ sind $d_1,\ldots,d_m$ also positive natürliche Zahlen.
- Wendet man den LLL-Algorithmus mit einer Konstanten $c \leq 1$ an, werden dabei die Vektoren b_i und b_{i-1} vertauscht, so folgt

  $$d'_{i-1} < cd_{i-1} \leq d_{i-1},$$

  also $d'_{i-1} \leq d_{i-1} - 1$, da wir es mit ganzen Zahlen zu tun haben. Wegen $d_1,\ldots,d_m \in \mathbf{Z}$ können also nur endlich viele Vertauschungen vorkommen. Daher hört der Algorithmus nach endlich vielen Schritten auf.
- Wir wollen jetzt d_i abschätzen: Sei

  $$H = \max(|b_{ij}| : 1 \leq i \leq m, 1 \leq j \leq n).$$

  Dann gilt für das Skalarprodukt

  $$|b_j \cdot b_k| = |\sum_{l=1}^{n} b_{jl} b_{kl}| \leq nH^2$$

  und für die Zeilennorm

  $$\|(b_j \cdot b_1, b_j \cdot b_2, \ldots, b_j \cdot b_i)\| = \sqrt{|b_j \cdot b_1|^2 + |b_j \cdot b_2|^2 + \ldots |b_j \cdot b_i|^2} \leq \sqrt{i \cdot n^2 H^4} = \sqrt{i} \cdot nH^2.$$

  Daher folgt mit der Hadamardschen Determinantenabschätzung

  $$d_i \leq \left(\sqrt{i} \cdot nH^2\right)^i = i^{i/2} \cdot n^i \cdot H^{2i}.$$

- Sei v_i die Anzahl der Vertauschungen der Basisvektoren Nr. i und $i+1$ während des Reduktionsprozesses. Bei jedem solchen Vertauschungsschritt gilt $d'_i < cd_i$. Ist d_i^e der letzte d_i-Wert (und d_i der erste), so gilt also

  $$d_i^e < c^{v_i} d_i.$$

  Mit $d_i^e \in \mathbf{N}$ folgt

  $$1 \leq c^{v_i} d_i.$$

  Um hier vernünftig weitermachen zu können, müssen wir $c < 1$ voraussetzen. Es gilt dann

  $$(\frac{1}{c})^{v_i} \leq d_i \leq i^{i/2} \cdot n^i \cdot H^{2i}.$$

  Logarithmenbildung liefert

  $$v_i \log \frac{1}{c} \leq \frac{i}{2} \log i + i \log n + 2i \log H.$$

- Wir schätzen damit ab, wieviele Vertauschungen insgesamt vorkommen können:

$$\begin{aligned}\sum_{i=1}^{m-1} v_i &\leq \frac{1}{\log\frac{1}{c}} \sum_{i=1}^{m-1} \left(\frac{i}{2}\log i + i\log n + 2i\log H\right) \leq \\ &\leq \frac{1}{\log\frac{1}{c}} \sum_{i=1}^{m-1} i\left(\frac{1}{2}\log(m-1) + \log n + 2\log H\right) \leq \\ &\leq \frac{1}{\log\frac{1}{c}} \frac{(m-1)m}{2}\left(\frac{1}{2}\log(m-1) + \log n + 2\log H\right) \leq \\ &\leq \frac{1}{\log\frac{1}{c}} m^2\,(\log n + \log H)\end{aligned}$$

  Für $c = \frac{3}{4}$ ergibt sich:

$$\sum_{i=1}^{m-1} v_i \leq 8.01 m^2(\log_{10} n + \log_{10} H).$$

- Da nur endlich viele Vertauschungen vorkommen, hört also der Algorithmus nach endlich vielen Schritten auf.

Beispiel: Wir starten mit der Gitterbasis

$$(-65, 122, 247), (-98, 180, 363), (-89, 159, 319).$$

Obige Abschätzung für die Anzahl der Vertauschungen liefert $v_1 \leq 44$, $v_2 \leq 92$. Tatsächlich ist $(v_1, v_2) = (7, 4)$, was zur reduzierten Basis $(-1, 0, 0), (0, 0, 1), (0, -1, 0)$ führt.

Ist das Gitter Λ nicht in $\mathbf{Z}^n$ enthalten, so kann man auf folgende Weise die Anzahl der Vertauschungen abschätzen:

- Sei ℓ_1 die Länge des kürzesten Vektors $\neq 0$ in Λ. Der Satz von Minkowski liefert für das Gitter $\Lambda_i = \mathbf{Z}b_1 + \mathbf{Z}b_2 + \cdots + \mathbf{Z}b_i$

$$\ell_1^i \leq \|a_1\|\|a_2\|\ldots\|a_i\| \leq (\frac{2}{\sqrt{\pi}})^i \Gamma(1 + \frac{i}{2}) \det \Lambda_i = (\frac{2}{\sqrt{\pi}})^i \Gamma(1 + \frac{i}{2})\sqrt{d_i}$$

  und somit

$$d_i \geq \frac{\pi^i \ell_1^{2i}}{4^i(\Gamma(1 + \frac{i}{2}))^2}.$$

- Wird während des LLL-Algorithmus der Basisvektor Nr. i mit dem Basisvektor Nr. $i+1$ v_i-mal vertauscht, so folgt wie oben

$$c^{v_i} d_i \geq \frac{\pi^i \ell_1^{2i}}{4^i(\Gamma(1 + \frac{i}{2}))^2},$$

  also

$$v_i \log\frac{1}{c} \leq \log d_i + i\log 4 + 2\log\Gamma(1 + \frac{i}{2}) - i\log\pi - 2i\log\ell_1.$$

  Auch hieraus ergibt sich, daß nur endlich viele Vertauschungen vorkommen. Also muß der LLL-Algorithmus nach endlich vielen Schritten aufhören. Leider kann man hier die Anzahl der Schritte nicht von vorne herein abschätzen, da ℓ_1 nicht bekannt ist.

Bemerkung: Nach Cohen läßt sich die Laufzeit des LLL-Algorithmus durch $O(n^6 \ln^3 B)$ abschätzen, wenn $\|b_i\|^2 \leq B$ für alle i gilt.

13. Laufzeit-Experimente

Die nachfolgenden experimentellen Daten sollen ein Gefühl für den Rechenaufwand einer LLL-Reduktion vermitteln, wobei die LLL-Reduktion zum Vergleich mit drei verschiedenen Funktionen durchgeführt wurde, nämlich mit der (von uns geschriebenen) Maple-Funktion 'lll', mit der Maple-Funktion 'lattice' und mit der NTL-Funktion 'LLL'.
Ein Tabelleneintrag (n h Rechenzeit) bedeutet dabei folgendes: Zufällig wurden n Vektoren $b_i \in \mathbf{Z}^n$ mit $|b_{ij}| \leq 10^h$ gewählt, $(H = \max(|b_{ij}| : 1 \leq i \leq m, 1 \leq j \leq n) \approx 10^h)$, dann das zugehörige Gitter in der angegebenen Rechenzeit (Stunden:Minuten:Sekunden) LLL-reduziert, wobei ein 400-MHz-Pentium-II-PC unter Linux benutzt wurde.

13.1. LLL-Reduktion mit der Maple-Funktion 'lll'.

n	h	Rechenzeit
10	10	0:00:09
10	10	0:00:11
10	10	0:00:07
15	10	0:00:40
15	10	0:01:21
15	10	0:01:51
20	10	0:06:13
20	10	0:05:38
20	10	0:17:56
25	10	0:08:53
25	10	0:18:48
25	10	0:19:59
30	10	0:55:40
30	10	0:56:48
30	10	0:57:11

n	h	Rechenzeit
10	50	0:02:30
10	50	0:02:51
10	50	0:03:10
15	50	0:18:01
15	50	0:42:02
15	50	0:16:34
20	50	2:19:52
20	50	1:32:56
20	50	1:51:34

n	h	Rechenzeit
10	100	0:13:57
10	100	0:09:21
10	100	0:11:15
15	100	0:53:23
15	100	1:43:16
15	100	0:44:37

13.2. LLL-Reduktion mit der Maple-Funktion 'lattice'.

n	h	Rechenzeit
10	10	0:00:01
10	10	0:00:01
10	10	0:00:01
15	10	0:00:06
15	10	0:00:08
15	10	0:00:09
20	10	0:00:27
20	10	0:00:25
20	10	0:00:40
25	10	0:01:03
25	10	0:01:14
25	10	0:01:15
30	10	0:02:43
30	10	0:02:48
30	10	0:02:53
35	10	0:05:05
35	10	0:06:00
35	10	0:05:29
40	10	0:10:01
40	10	0:10:50
40	10	0:10:27
45	10	0:19:50
45	10	0:18:06
45	10	0:18:07
50	10	0:29:38
50	10	0:29:41
50	10	0:31:38

n	h	Rechenzeit
10	50	0:00:18
10	50	0:00:18
10	50	0:00:17
15	50	0:01:59
15	50	0:02:36
15	50	0:01:56
20	50	0:07:45
20	50	0:07:37
20	50	0:11:25
25	50	0:19:15
25	50	0:21:48
25	50	0:19:37
30	50	0:47:06
30	50	0:53:53
30	50	0:48:26
35	50	1:55:16
35	50	1:42:31
35	50	1:53:19
40	50	3:09:20
40	50	3:03:06
40	50	3:05:12

n	h	Rechenzeit
10	100	00:01:23
10	100	00:00:51
10	100	00:01:17
15	100	00:07:22
15	100	00:08:23
15	100	00:09:33
20	100	00:24:29
20	100	00:28:17
20	100	00:25:53
25	100	01:15:14
25	100	01:12:01
25	100	01:34:57

13.3. LLL-Reduktion mit der NTL-Funktion 'LLL'.

n	h	Rechenzeit	n	h	Rechenzeit	n	h	Rechenzeit	n	h	Rechenzeit
10	10	00:00:00	10	50	00:00:00	10	100	00:00:00	10	200	00:00:00
10	10	00:00:00	10	50	00:00:00	10	100	00:00:00	10	200	00:00:00
10	10	00:00:00	10	50	00:00:00	10	100	00:00:00	10	200	00:00:00
20	10	00:00:00	20	50	00:00:01	20	100	00:00:03	20	200	00:00:12
20	10	00:00:00	20	50	00:00:01	20	100	00:00:03	20	200	00:00:05
20	10	00:00:00	20	50	00:00:01	20	100	00:00:02	20	200	00:00:04
30	10	00:00:00	30	50	00:00:03	30	100	00:00:10	30	200	00:00:32
30	10	00:00:00	30	50	00:00:03	30	100	00:00:09	30	200	00:00:21
30	10	00:00:00	30	50	00:00:03	30	100	00:00:11	30	200	00:00:33
40	10	00:00:01	40	50	00:00:12	40	100	00:00:25	40	200	00:01:23
40	10	00:00:01	40	50	00:00:10	40	100	00:00:29	40	200	00:01:16
40	10	00:00:01	40	50	00:00:09	40	100	00:00:26	40	200	00:01:18
50	10	00:00:02	50	50	00:00:24	50	100	00:01:10	50	200	00:03:45
50	10	00:00:02	50	50	00:00:23	50	100	00:01:21	50	200	00:03:16
50	10	00:00:02	50	50	00:00:25	50	100	00:01:23	50	200	00:03:55
60	10	00:00:04	60	50	00:00:53	60	100	00:02:53	60	200	00:07:39
60	10	00:00:05	60	50	00:00:56	60	100	00:02:32	60	200	00:08:07
60	10	00:00:05	60	50	00:00:53	60	100	00:03:09	60	200	00:07:54
70	10	00:00:09	70	50	00:01:39	70	100	00:05:00	70	200	00:14:43
70	10	00:00:09	70	50	00:01:41	70	100	00:05:21	70	200	00:16:04
70	10	00:00:09	70	50	00:01:42	70	100	00:05:19	70	200	00:14:51
80	10	00:00:15	80	50	00:03:14	80	100	00:09:18	80	200	00:26:55
80	10	00:00:15	80	50	00:03:01	80	100	00:09:13	80	200	00:28:07
80	10	00:00:16	80	50	00:03:01	80	100	00:09:33	80	200	00:26:31
90	10	00:00:26	90	50	00:05:17	90	100	00:15:24	90	200	00:45:51
90	10	00:00:26	90	50	00:05:05	90	100	00:16:00	90	200	00:45:16
90	10	00:00:27	90	50	00:05:43	90	100	00:15:13	90	200	00:47:10
100	10	00:00:42	100	50	00:08:26	100	100	00:25:13	100	200	01:11:21
100	10	00:00:42	100	50	00:08:42	100	100	00:24:22	100	200	01:11:37
100	10	00:00:41	100	50	00:08:29	100	100	00:25:32	100	200	01:13:03

Literatur: [**Co**], [**MeOoVa**].

KAPITEL 2

Rucksackalgorithmen – Die Merkle-Hellman-Rucksackverschlüsselung

1. Das allgemeine Rucksackproblem

Beim Rucksackproblem hat man natürliche Zahlen $a_1, a_2, \ldots, a_n$ und s. Man fragt, ob es eine Teilmenge von $\{a_1, \ldots, a_n\}$ gibt, sodaß die Summe der Elemente s ergibt. Mit anderen Worten: Gibt es Zahlen x_i mit

$$x_1a_1 + x_2a_2 + \cdots + x_na_n = s \quad \text{und} \quad x_i \in \{0, 1\}?$$

Im Englischen heißt das Rucksackproblem knapsack problem oder auch subset-sum problem.

Beispiel: Wir wählen (zufällig)

$$a_1 = 1009, a_2 = 1158, a_3 = 1202, a_4 = 1367, a_5 = 1405, a_6 = 1716, a_7 = 1894, a_8 = 1899, a_9 = 1914.$$

Zwischen 9100 und 9110 gibt es folgende Zahlen s, die sich als Teilsumme der a_i's schreiben lassen:

$$9101 = a_1 + a_2 + a_5 + a_6 + a_8 + a_9$$
$$9102 = a_1 + a_3 + a_4 + a_6 + a_7 + a_9$$
$$9107 = a_1 + a_3 + a_4 + a_6 + a_8 + a_9$$

Bemerkung: Hat man ein Verfahren, mit dem man entscheiden kann, ob ein Rucksackprobleme lösbar sind, so kann man auf folgende Weise auch explizit Lösungen konstruieren:
Gegeben seien natürliche Zahlen $a_1, \ldots, a_n$ und s. Wir wollen die Gleichung $\sum_{i=1}^{n} x_ia_i = s$ mit $x_i \in \{0, 1\}$ lösen.

1. Ist das Rucksackproblem mit den Parametern $a_1, \ldots, a_n$ und s nicht lösbar, so hat obige Gleichung keine Lösung und wir hören auf. Andernfalls gehen wir folgendermaßen vor:
2. Ist das Rucksackproblem $\sum_{i=1}^{n-1} x_ia_i = s - a_n$ mit $x_i \in \{0, 1\}$ lösbar, so setzen wir $x_n = 1$, andernfalls $x_n = 0$.
3. Wir haben jetzt x_n gewählt und wissen, daß das Rucksackproblem $\sum_{i=1}^{n-1} x_ia_i = s - x_na_n$ mit $x_1, \ldots, x_{n-1} \in \{0, 1\}$ lösbar ist. Nun beginnen wir mit diesem Problem von vorne.

Hätte man also ein Verfahren um die Lösbarkeit des Rucksackproblems schnell zu entscheiden, könnte man auch schnell eine explizite Lösung bestimmen.

Wie kann man das Rucksackproblem lösen?

Die naive Methode: Um $\sum_{i=1}^{n} x_ia_i = s$ zu untersuchen, berechnen wir für alle $(x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$ die Summe $\sum_{i=1}^{n} x_ia_i$ und testen, ob sie gleich s ist. Auf diese Weise kann man alle Lösungen des Rucksackproblems bestimmen. Allerdings beträgt die Anzahl der Schritte 2^n, d.h. die Schrittzahl wächst exponentiell mit der Anzahl der a_i's.

Bemerkung: Die naive Rucksackberechnungsmethode haben wir mit NTL programmiert ('ru_ntl.c') und damit die nachfolgenden Beispiele gerechnet.

Beispiele: Bei den nachfolgenden Beispielen wurden in Abhängigkeit von n zufällig $a_1, \ldots, a_n$ und $x_1, \ldots, x_n$ gewählt, damit $s = x_1a_1 + \ldots, x_na_n$ berechnet, sodann das zugehörige Rucksackproblem mit der naiven Methode angegangen. Die Rechenzeit ist in Stunden:Minuten:Sekunden angegeben.

Version vom 17. Juli 2002

n	$\min(a_i)$	$\max(a_i)$	$x_1 \dots x_n$	Zeit
18	5538838346	51426729846	0100100001100101 01	00:00:06
18	5538838346	51426729846	1101010111010001 10	00:00:03
18	5538838346	51426729846	0101001000000110 10	00:00:03
18	5538838346	51426729846	0101110011011101 00	00:00:02
18	5538838346	51426729846	0010000001010011 01	00:00:06
19	5876942502	200202218105	0100100001100101 011	00:00:15
19	5876942502	200202218105	1010101110100011 001	00:00:11
19	5876942502	200202218105	0100100000011010 010	00:00:05
19	5876942502	200202218105	1110011011101000 010	00:00:05
19	5876942502	200202218105	0000010100110110111	00:00:17
20	28145078618	676038033936	0100100001100101 0111	00:00:36
20	28145078618	676038033936	0101011101000110 0101	00:00:25
20	28145078618	676038033936	0010000001101001 0111	00:00:35
20	28145078618	676038033936	0011011101000010 0000	00:00:01
20	28145078618	676038033936	0101001101101111 0110	00:00:17
21	361335436848	42405447429 57	0100100001100101 01110	00:00:37
21	361335436848	42405447429 57	1010111010001100 10100	00:00:13
21	361335436848	42405447429 57	1000000110100101 11001	00:00:50
21	361335436848	42405447429 57	1011101000010000 00101	00:00:51
21	361335436848	42405447429 57	0011011011110110 11100	00:00:18
22	2186036449459	21521190275903	0100100001100101 01101	00:02:04
22	2186036449459	21521190275903	0101110100011001 01001 0	00:00:48
22	2186036449459	21521190275903	0000011010010111 00111 01	00:01:59
22	2186036449459	21521190275903	1101000010000001 01001 1	00:02:14
22	2186036449459	21521190275903	0110111101101110 01101 1	00:02:25
23	1486273674432	60349139834573	0100100001100101 01110 10	00:02:08
23	1486273674432	60349139834573	1011101000110010 10001 0	00:00:25
23	1486273674432	60349139834573	0001101001011100 11011 10	00:02:43
23	1486273674432	60349139834573	1000010000001010 01100 10	00:02:29
23	1486273674432	60349139834573	1111011011100110 11100 11	00:04:47
24	8049692606031	245022363996954	0100100001100101 01110 101	00:08:24
24	8049692606031	245022363996954	0111010001100101 00100 0000	00:00:12
24	8049692606031	245022363996954	0110100101110011 01110 0100	00:02:12
24	8049692606031	245022363996954	0010000001010011 01101 1111	00:11:57
24	8049692606031	245022363996954	0110111001101110 01110 0100	00:02:11
25	5623507954393	954531177072417	0100100001100101 01110 10101 0	00:08:39
25	5623507954393	954531177072417	1110100011001010 01000 0001	00:12:56
25	5623507954393	954531177072417	1010010111001101 11010 0001	00:13:23
25	5623507954393	954531177072417	0000001010011011 01111 0110	00:11:06
25	5623507954393	954531177072417	1110011011100111 01000 1100	00:04:51
26	61199209046087	4094486338897781	0100100001100101 01110 10101 01	00:35:41
26	61199209046087	4094486338897781	1101000110010100 10000 0110	00:19:54
26	61199209046087	4094486338897781	1001011100110111 01000 0100	00:03:21
26	61199209046087	4094486338897781	0001010011011011 11101 1110	00:24:40
26	61199209046087	4094486338897781	0110111001110100 01100 0101	00:33:35
27	137918113025299	16838811959505700	0100100001100101 01110 10101 011	01:32:36
27	137918113025299	16838811959505700	1010001100101001 00000 0110	00:37:38
27	137918113025299	16838811959505700	0101110011011101 00001 0000	00:00:50
27	137918113025299	16838811959505700	1010011011011110 11011 0110	00:44:20
27	137918113025299	16838811959505700	1110011101000110 00010 0011	01:28:42
28	1620483528711429	56162010095159827	0100100001100101 01110 10101 0111	03:32:08
28	1620483528711429	56162010095159827	0100011001010010 00000 1010	02:14:00
28	1620483528711429	56162010095159827	0111001101110100 00010 0000	02:23:19
28	1620483528711429	56162010095159827	0011011011110110 11001 1110	01:45:37
28	1620483528711429	56162010095159827	0111010001100001 01100 1110	01:09:21

Eine weitere allgemeine Methode für Rucksackprobleme: Gegeben sind $a_1, \ldots, a_n$ und s. Wir bilden zwei Listen/Mengen

$$S_1 = \{ \sum_{1 \le i \le \frac{n}{2}} x_i a_i : x_i \in \{0,1\}\} \quad \text{und} \quad S_2 = \{s - \sum_{\frac{n}{2} < i \le n} a_i x_i : x_i \in \{0,1\}\}$$

und testen, ob ein gemeinsames Element vorhanden ist. Wenn ja, haben wir eine Relation

$$\sum_{1 \le i \le \frac{n}{2}} x_i a_i = s - \sum_{\frac{n}{2} < i \le n} a_i x_i$$

und damit eine Lösung des Rucksackproblems. Praktisch kann man zuerst S_1 berechnen und sortieren (mit 'heapsort' oder 'quicksort'). Anschließend berechnet man Elemente aus S_2 und schaut (mit 'binary search') nach, ob sie in S_1 vorkommen. Das Sortieren und Vergleichen kann in $2^{n/2}\ln(2^{n/2})$ Schritten erfolgen, also kann man die Laufzeit durch $O(2^{\frac{n}{2}+\varepsilon})$ abschätzen. Im Unterschied zur naiven Methode muß man allerdings auch $2^{\frac{n}{2}}$ Speicherplätze haben, was problematisch sein kann.

Bemerkung: Die zuletzt vorgestellte Methode ist die beste (theoretische) Methode, die man für das allgemeine Rucksackproblem kennt. Rucksackprobleme scheinen also schwierig zu sein.

Verschlüsselung mit Rucksäcken: Die Schwierigkeit von Rucksackberechnungen kann man für die Kryptographie nutzen. Die grundlegende Idee ist ganz einfach. Man hat eine Folge von natürlichen Zahlen $a_1, \ldots, a_n$ gegeben. Einen Plaintext denkt man sich als Folge von Bits, bei denen man jeweils Blöcke der Länge n bildet. Der Plaintextblock $x_1 x_2 \ldots x_n$ (mit $x_i \in \{0,1\}$) wird dann zu

$$c = \sum_{i=1}^{n} x_i a_i$$

verschlüsselt. Da das Rucksackproblem schwierig ist, kann man auch bei Kenntnis von $a_1, \ldots, a_n$ und c nicht auf $x_1 \ldots x_n$ zurückschließen. Allerdings ist auch nicht klar, wie der Empfänger die Nachricht entschlüsseln kann, d.h. auf diese Weise erhält man noch kein sinnvolles Kryptosystem.

2. Rucksäcke mit der superincreasing-Eigenschaft

Während im allgemeinen Rucksackprobleme schwer zu lösen sind, gibt es Rucksäcke, die ganz einfach zu behandeln sind.

Beispiel: Ist $b_i = 2^{i-1}$ für $1 \le i \le n$, so sieht man aus

$$s = \sum_{i=1}^{n} x_i b_i = \sum_{j=0}^{n-1} x_{j+1} 2^j = (x_n x_{n-1} \ldots x_2 x_1)_2,$$

daß $x_n, \ldots, x_1$ die Ziffern der Binärentwicklung von s sind.

Ein Rucksack, der durch $b_1, \ldots, b_n$ gegeben ist, bzw. eine Folge $b_1, b_2, \ldots, b_n$ hat die superincreasing-Eigenschaft, wenn gilt

$$\sum_{i=1}^{j-1} b_i < b_j \quad \text{für} \quad 2 \le j \le n.$$

Beispiel: Die Folge

$$1, \quad 2, \quad 4, \quad 10, \quad 19, \quad 40$$

hat die superincreasing-Eigenschaft.

Rucksackprobleme mit der superincreasing-Eigenschaft lassen sich ganz einfach lösen, was auf folgender Beobachtung beruht: Sei

$$\sum_{i=1}^{n} x_i b_i = s \quad \text{mit} \quad x_i \in \{0,1\}.$$

Dann gilt:

$$x_n = 0 \quad \Longrightarrow \quad s = \sum_{i=1}^{n} x_i b_i \leq \sum_{i=1}^{n-1} b_i < b_n \qquad \text{und} \qquad x_n = 1 \quad \Longrightarrow \quad s = \sum_{i=1}^{n} x_i b_i \geq b_n.$$

Durch Vergleich von s und b_n sieht man also, ob $x_n = 1$ oder $x_n = 0$ gelten muß. Insbesondere ist also x_n eindeutig bestimmt. Man erhält dann durch Rekursion folgendes Verfahren:

Algorithmus zur Lösung des Rucksackproblems mit der superincreasing-Eigenschaft: $b_1, \ldots, b_n$ habe die superincreasing-Eigenschaft. Sei s ein natürliche Zahl. Wir wollen die Lösbarkeit der Gleichung $\sum_{i=1}^{n} x_i b_i = s$ untersuchen und eventuell eine Lösung bestimmen.

1. Setze $s_n = s$ und $j = n$.
2. Gilt $s_j \geq b_j$, setze $x_j = 1$, sonst $x_j = 0$.
3. Setze $s_{j-1} = s_j - x_j b_j$.
4. Gilt $j \geq 2$, setze $j := j - 1$ und gehe zu 2.
5. Gilt $s_0 = 0$ hat man eine Lösung des Rucksackproblems, sonst gibt es keine Lösung.

Obige Eindeutigkeitsaussage für x_n liefert durch Induktion sofort folgende Aussage:

LEMMA. *Die Folge $b_1, \ldots, b_n$ habe die superincreasing-Eigenschaft. Sind $x_i, y_i \in \{0, 1\}$ mit*

$$x_1 b_1 + \cdots + x_n b_n = y_1 b_1 + \cdots + y_n b_n,$$

so folgt $x_i = y_i$ für alle $i = 1, \ldots, n$.

Beispiel: Wir betrachten die Folge $b_1 = 129$, $b_2 = 187$, $b_3 = 413$, $b_4 = 901$, $b_5 = 1991$, die die superincreasing-Eigenschaft besitzt. Wir wollen das Rucksackproblem

$$129x_1 + 187x_2 + 413x_3 + 901x_4 + 1991x_5 = 3079$$

lösen. Wir setzen $s_5 = 3079$. Wegen $s_5 \geq b_5$ ist $x_5 = 1$. Dann ist $s_4 = 1088 \geq b_4 = 901$, also $x_4 = 1$. Mit $s_3 = 187 < b_3 = 413$ folgt $x_3 = 0$, mit $s_2 = 198 = b_2$ folgt schließlich $x_2 = 1$ und $x_1 = 0$. Das Rucksackproblem ist also lösbar und hat $(x_1 \ldots x_5) = (01011)$ als Lösung.

Da Rucksäcke mit der superincreasing-Eigenschaft von jedem leicht zu lösen sind, kann man sie zunächst nicht sinnvoll zur Verschlüsselung benutzen.

3. Die Merkle-Hellman-Rucksack-Verschlüsselung

Das Verfahren beruht auf einem Rucksack mit der superincreasing-Eigenschaft, wobei man aber diese Eigenschaft durch Permutation und (modulare) Multiplikation zu verbergen sucht.

Das Merkle-Hellman-Rucksack-Kryptosystem:

1. Die Schlüsselerzeugung erfolgt folgendermaßen:
   (a) Jeder Teilnehmer A wählt sich eine Folge $(b_1, b_2, \ldots, b_n)$ mit der superincreasing-Eigenschaft, eine Zahl M mit $M > b_1 + b_2 + \cdots + b_n$, eine (zufällige) Zahl W mit $1 \leq W \leq M - 1$ und $\mathrm{ggT}(W, M) = 1$ und berechnet U mit $1 \leq U \leq M - 1$ und $UW \equiv 1 \bmod M$. Außerdem wählt A eine (zufällige) Permutation π der Zahlen $1, 2, \ldots, n$.
   (b) A berechnet $a_i = W b_{\pi(i)} \bmod M$ für $1 \leq i \leq n$.
   (c) A's öffentlicher Schlüssel ist $(a_1, a_2, \ldots, a_n)$, sein privater Schlüssel ist $(U, M, (b_1, b_2, \ldots, b_n), \pi)$.
2. B will eine Nachricht m an A senden:
   (a) B besorgt sich den öffentlichen Schlüssel $(a_1, a_2, \ldots, a_n)$ von A.
   (b) Nach einem festgelegten Verfahren wird m in Blöcke $m_1 m_2 \ldots m_n$ mit je n Bits, d.h. $m_i \in \{0, 1\}$, zerlegt.
   (c) B berechnet $c = m_1 a_1 + \cdots + m_n a_n$ und schickt c als ciphertext an A.
3. Wie entschlüsselt A den ciphertext c?
   (a) A berechnet $d \equiv Uc \bmod M$ und bestimmt mit dem oben dargestellten Verfahren (für Folgen mit der superincreasing-Eigenschaft) $r_1, \ldots, r_n \in \{0, 1\}$ mit

   $$d = r_1 b_1 + r_2 b_2 + \cdots + r_n b_n.$$

   Dann liefert $m_i = r_{\pi(i)}$ den n-Bit-Block des Ausgangstexts m.

Beweis für die Richtigkeit der Entschlüsselung: Es gilt

$$\sum_i r_{\pi(i)} b_{\pi(i)} = \sum_i r_i b_i = d \equiv Uc = U \sum_i m_i a_i = \sum_i m_i U a_i \equiv \sum_i m_i b_{\pi(i)} \bmod M.$$

Nach Voraussetzung ist $M > b_1 + \cdots + b_n$, also folgt $0 \le \sum_i r_{\pi(i)} b_{\pi(i)}, \sum_i m_i b_{\pi(i)} \le M-1$. Die Kongruenz ist also sogar eine Gleichheit in $\mathbf{Z}$:

$$\sum_i r_{\pi(i)} b_{\pi(i)} = \sum_i m_i b_{\pi(i)}.$$

Die Eindeutigkeit der Koeffizienten bei einer Folge mit der superincreasing-Eigenschaft liefert nun $m_i = r_{\pi(i)}$, was gezeigt werden sollte. ■

Beispiel: Zur Erzeugung eines Schlüssels wählen wir $n = 5$ und $b_1 = 129$, $b_2 = 187$, $b_3 = 413$, $b_4 = 901$, $b_5 = 1991$. Dann $M = 4311$, $W = 271$ und $\pi(1) = 5$, $\pi(2) = 3$, $\pi(3) = 2$, $\pi(4) = 1$, $\pi(5) = 4$. Dann wird der öffentliche Schlüssel

$$a = (686, 4148, 3256, 471, 2755)$$

und der private Schlüssel

$$(U, M, b, \pi) = (3277, 4311, (129, 187, 413, 901, 1991), [5, 3, 2, 1, 4]).$$

Wir wollen die Bitfolge 10101 verschlüsseln: Man erhält

$$c = a_1 + a_3 + a_5 = 6697.$$

Wie entschlüsselt man dies? Wir erhalten $d = 3079 \equiv Uc \bmod M$ und müssen das Rucksackproblem $129r_1 + 187r_2 + 413r_3 + 901r_4 + 1991r_5 = 3079$ lösen. Das haben wir bereits erledigt mit dem Ergebnis $(r_1 \ldots r_5) = (01011)$, was nach Permutation $(m_1 \ldots m_5) = (10101)$ ergibt.

Bemerkungen:

1. Wir haben eine Maple-Funktion 'mehe_key' geschrieben zur 'zufälligen' Schlüsselerzeugung, wobei die Folgenlänge n angegeben werden muß. Dann wird mit Zufallszahlen $z_i \in \{1, \ldots, 2^n\}$ eine Folge b_i mit der superincreasing-Eigenschaft durch

   $$b_1 = 2^n + z_1, \quad \ldots, \quad b_i = b_1 + \cdots + b_{i-1} + z_i, \quad \ldots$$

   konstruiert. Weiter wählt man $M = b_1 + \cdots + b_n + z_{n+1}$ und $W \in \{1, \ldots, M-1\}$ zufällig mit $\mathrm{ggT}(W, M) = 1$. Nach Konstruktion einer zufälligen Permutation π der Zahlen $1, 2, \ldots, n$ kann man damit den zugehörigen privaten und öffentlichen Schlüssel angeben.
2. Verschlüsselung und Entschlüsselung passiert mit den Maple-Funktionen 'mehe_en' und 'mehe_de'. Bei der Verschlüsselung wird dabei eine Bytefolge eingegeben. Aus Bequemlichkeitsgründen wird dann die Folgenlänge als durch 8 teilbar vorausgesetzt.

Beispiel: Mit 'mehe_key' wurde folgendes Merkle-Hellman-Schlüsselpaar

$$\begin{aligned} U &= 386417, \quad M = 1064602, \\ b &= [1626, 2033, 4205, 8068, 16701, 33284, 66691, 133417, 266123, 532438], \\ \pi &= [9, 8, 6, 10, 5, 3, 1, 7, 2, 4], \\ a &= [730323, 268379, 889912, 1026556, 416915, 545703, 815090, 187617, 332949, 766758] \end{aligned}$$

erzeugt.

Bemerkung: Das Merkle-Hellman-Verschlüsselungssystem ist nicht sicher, ebenso wie einige andere kryptographische Verfahren, die auf Rucksackproblemen beruhen. Im folgenden sollen zwei Angriffe auf das Merkle-Hellman-System vorgestellt werden.

4. Ein Angriff auf das Merkle-Hellman-Verschlüsselungssystem

4.1. Der private Schlüssel ist nicht eindeutig bestimmt. Sei $(a_1, \ldots, a_n)$ ein öffentlicher Merkle-Hellman-Schlüssel mit zugehörigem privaten Schlüssel

$$U, \quad M, \quad (b_1, \ldots, b_n), \quad \pi.$$

Insbesondere ist

$$Ua_i \equiv b_{\pi(i)} \bmod M \quad \text{und} \quad M > b_1 + \cdots + b_n.$$

1. Sei $\widetilde{M} > \max(a_1, \ldots, a_n)$ und $\widetilde{U}$ mit $1 \leq \widetilde{U} \leq \widetilde{M} - 1$ und $\mathrm{ggT}(\widetilde{U}, \widetilde{M}) = 1$.
2. Indem man alle $\widetilde{U}a_i \bmod \widetilde{M}$ berechnet und sortiert, erhält man eine streng wachsende Folge

   $$\widetilde{b}_1 < \widetilde{b}_2 < \cdots < \widetilde{b}_n \quad \text{und eine Permutation } \widetilde{\pi}$$

   mit

   $$\widetilde{U}a_i \equiv \widetilde{b}_{\widetilde{\pi}(i)} \bmod \widetilde{M} \text{ für } i = 1, \ldots, n.$$

3. Ist $\widetilde{M} > \widetilde{b}_1 + \cdots + \widetilde{b}_n$ und hat $(\widetilde{b}_1, \ldots, \widetilde{b}_n)$ die superincreasing-Eigenschaft, so ist

   $$\widetilde{U}, \quad \widetilde{M}, \quad (\widetilde{b}_1, \ldots, \widetilde{b}_n), \quad \widetilde{\pi}$$

   ein privater Merkle-Hellman-Schlüssel mit $(a_1, \ldots, a_n)$ als zugehörigem öffentlichen Schlüssel. In diesem Fall kann man zum Entschlüsseln von mit $(a_1, \ldots, a_n)$ verschlüsselten Nachrichten auch den Schlüssel $(\widetilde{U}, \widetilde{M}, \widetilde{b}, \widetilde{\pi})$ verwenden.
4. Die Maple-Funktion 'aUM_test' berechnet bei Eingabe von $(a_1, \ldots, a_n)$, $\widetilde{U}$, $\widetilde{M}$ die Folge $(\widetilde{b}_1, \ldots, \widetilde{b}_n)$ und die Permutation $\widetilde{\pi}$ und testet, ob man einen privaten Merkle-Hellman-Schlüssel erhält.

Beispiel: Wir beginnen mit dem privaten Schlüssel

$$U = 521, \quad M = 1156, \quad b = (58, 81, 141, 292, 573), \quad \pi = [4, 5, 3, 2, 1].$$

Der öffentliche Schlüssel ist dann

$$a = [76, 933, 393, 29, 506].$$

Für $\widetilde{M} = 10000$ und alle $\widetilde{U}$ mit $1 \leq \widetilde{U} \leq \widetilde{M} - 1$ und $\mathrm{ggT}(\widetilde{U}, \widetilde{M}) = 1$ berechnen wir $\widetilde{b}$ und $\widetilde{\pi}$ mit $\widetilde{U}a_i \equiv b_{\pi(i)} \bmod \widetilde{M}$ und testen, ob ein privater Merkle-Hellman-Schlüssel gefunden wurde. Dies passiert in folgenden Fällen:

$\widetilde{U}$	$\widetilde{M}$	$[\widetilde{b}_1, \ldots, \widetilde{b}_n]$	$[\widetilde{\pi}(1), \ldots, \widetilde{\pi}(n)]$
397	10000	[172, 401, 882, 1513, 6021]	[1, 2, 5, 4, 3]
2787	10000	[222, 271, 823, 1812, 5291]	[4, 2, 5, 3, 1]
4491	10000	[103, 239, 1316, 2446, 4963]	[3, 1, 5, 2, 4]
7253	10000	[18, 337, 429, 1228, 7049]	[4, 5, 3, 2, 1]
7589	10000	[34, 81, 537, 2477, 6764]	[5, 3, 4, 2, 1]

Für $\widetilde{M} = 10007$ erhält man folgendes Ergebnis:

$\widetilde{U}$	$\widetilde{M}$	$[\widetilde{b}_1, \ldots, \widetilde{b}_n]$	$[\widetilde{\pi}(1), \ldots, \widetilde{\pi}(n)]$
397	10007	[142, 151, 742, 1506, 5916]	[2, 1, 5, 4, 3]
408	10007	[232, 398, 987, 1825, 6308]	[3, 2, 1, 4, 5]
2789	10007	[247, 317, 825, 1817, 5314]	[4, 2, 5, 3, 1]
4355	10007	[318, 373, 749, 2090, 6211]	[3, 2, 1, 5, 4]
4510	10007	[464, 699, 1191, 2522, 4890]	[4, 5, 3, 2, 1]
5202	10007	[71, 371, 753, 2958, 5079]	[5, 1, 4, 3, 2]
6586	10007	[185, 186, 440, 861, 6492]	[2, 3, 5, 4, 1]
8989	10007	[206, 499, 871, 2688, 5256]	[4, 3, 1, 2, 5]
9676	10007	[8, 408, 1394, 2633, 4865]	[5, 3, 1, 2, 4]

Alle angegebenen privaten Schlüssel liefern also die gleiche Entschlüsselungsfunktion.

Bemerkung: Zum Entschlüsseln muß man also nicht einen festen privaten Schlüssel finden, sondern irgendeinen, dessen zugehöriger privater Schlüssel $(a_1,\dots,a_n)$ ist. Natürlich ist die Frage, wie man zu einem geeigneten Paar $(\widetilde{U},\widetilde{M})$ kommt, wenn man nur $(a_1,\dots,a_n)$ kennt.

Beispiel: Wir wählen den privaten Schlüssel

$$U=521,\quad M=1156,\quad b=[58,81,141,292,573],\quad \pi=[4,5,3,2,1]$$

mit öffentlichem Schlüssel $a=[76,933,393,29,506]$. Wir probieren, ob sich bei Wahl von

$$\frac{\widetilde{U}}{\widetilde{M}}=\frac{U}{M}+\frac{(-1)^e}{2^m}$$

ein zu a gehöriger privater Schlüssel ergibt. Dies ist tatsächlich für folgende Werte von (m,e) der Fall:

$$(14,1),(15,1),(16,1),(17,1),(18,0),(18,1),(19,0),(19,1),(20,0),(20,1),(21,0),(21,1),\dots$$

Das Beispiel legt die Vermutung nahe: Ist $\frac{\widetilde{U}}{\widetilde{M}}$ hinreichend nah an $\frac{U}{M}$, so liefert $(\widetilde{U},\widetilde{M})$ eine privaten Schlüssel mit öffentlichem Schlüssel $(a_1,\dots,a_n)$.

4.2. Numerische Beobachtungen. Um eine Vorstellung von der Größenordnung der b_i's zu bekommen, geben wir das folgende Lemma an:

LEMMA. *Ist $b_1,\dots,b_n$ eine Folge mit der superincreasing-Eigenschaft, so gilt*

$$b_i\ge 2^{i-2}(b_1+1)\quad \textit{für}\quad i\ge 2.$$

Gilt $M>b_1+b_2+\cdots+b_n$, so folgt

$$M\ge 2^{n-1}(b_1+1).$$

Beweis: durch Induktion nach i. Für $i=2$ ist die Behauptung wegen $b_2>b_1$ klar. Sei nun $i>2$. Dann folgt mit Induktionsvoraussetzung

$$\begin{aligned} b_i &\ge b_{i-1}+b_{i-2}+\cdots+b_2+b_1+1\ge\\ &\ge 2^{i-3}(b_1+1)+2^{i-4}(b_1+1)+\cdots+2^0(b_1+1)+(b_1+1)=2^{i-2}(b_1+1).\end{aligned}$$

Damit ergibt sich auch

$$\begin{aligned} M &\ge b_n+b_{n-1}+\cdots+b_2+b_1+1\ge\\ &\ge 2^{n-2}(b_1+1)+2^{n-3}(b_1+1)+\cdots+(b_1+1)+(b_1+1)=2^{n-1}(b_1+1),\end{aligned}$$

was gezeigt werden sollte. ■

Wir hatten die Folge b_i mit

$$b_1=2^n+z_1,\quad b_i=b_1+\cdots+b_{i-1}+z_i,\quad M=b_1+\cdots+b_n+z_{n+1}$$

und 'zufälligen' $z_i\in\{1,\dots,2^n\}$ konstruiert.
Verwenden wir das $\approx$-Zeichen um nur die Größenordnung anzudeuten, so kann man annehmen

$$b_1,b_2,b_3,b_4,b_5\approx 2^n,\quad b_n\approx 2^{2n},\quad M\approx 2^{2n}.$$

Da W zufällig gewählt wird, wird man i.a. auch $a_i\approx 2^{2n}$ erhalten.

Wegen $Ua_i\equiv b_{\pi(i)} \bmod M$ gibt es ganze Zahlen $k_i\ge 0$ mit

$$Ua_i-Mk_i=b_{\pi(i)}.$$

Wegen $U<M$ und $b_{\pi(i)}>0$ folgt

$$0\le k_i<a_i.$$

Sei $i_j=\pi^{-1}(j)$, d.h. die Zahl mit $\pi(i_j)=j$. Insbesondere gilt dann

$$Ua_{i_j}-Mk_{i_j}=b_j\quad\text{und}\quad k_{i_j}=\frac{Ua_{i_j}-b_j}{M}.$$

Zunächst folgt

$$\frac{U}{M}-\frac{k_{i_j}}{a_{i_j}}=\frac{b_j}{Ma_{i_j}}$$

und damit

$$\frac{U}{M} \approx \frac{k_{i_j}}{a_{i_j}}.$$

Bemerkung: Kommt man also an eine Größe $\frac{k_{i_j}}{a_{i_j}}$, so hat man eine Approximation für $\frac{U}{M}$ und man kann probieren, ob man damit einen privaten Schlüssel $(\widetilde{U}, \widetilde{M}, \widetilde{b}, \widetilde{\pi})$ konstruieren kann. Diese Idee wird im folgenden durchgeführt.

Wir berechnen weiter:

$$\begin{aligned} a_{i_1}k_{i_j} - a_{i_j}k_{i_1} &= a_{i_1}\frac{Ua_{i_j} - b_j}{M} - a_{i_j}\frac{Ua_{i_1} - b_1}{M} = \frac{b_1 a_{i_j} - b_j a_{i_1}}{M}, \\ |a_{i_1}k_{i_j} - a_{i_j}k_{i_1}| &= |\frac{b_1 a_{i_j} - b_j a_{i_1}}{M}| \leq \max(\frac{b_1 a_{i_j}}{M}, \frac{b_j a_{i_1}}{M}) \leq \max(b_1, b_j). \end{aligned}$$

Nach unseren Annahmen ist größenordnungsmäßig $b_1, \ldots, b_5 \approx 2^n$. Damit folgt

$$|a_{i_1}k_{i_j} - a_{i_j}k_{i_1}| \lessapprox 2^n \text{ für } j = 1, \ldots, 5.$$

Dies ist sehr ungewöhnlich, da sowohl k_i als auch a_i in der Größenordnung von 2^{2n} sind. Wir werden später sehen, daß man wegen dieser Eigenschaft $k_{i_1}, \ldots, k_{i_5}$ meist berechnen kann.

Beispiel: Wir wählen $n = 100$ und konstruieren 'zufällig' ein Merkle-Hellman-Schlüsselpaar mit $\pi = 1$. Einige der b_i's:

i	b_i	$\log_2 b_i$
1	1840513607636247338295337088006	100.5379513
2	2781238898674724961292320952487	101.1335705
3	5406009535023128830493419483625	102.0924069
4	11286267099882531318408192195238	103.1543393
5	21491854839738487212225694400232	104.0835609
98	216824286109868597612520714669996391472208301480127392957255	197.1102840
99	433648572219737195225041429339982949465640594012585346853544	198.1102840
100	867297144439474390450082858679539873881060725285144091193217	199.1102839

$$\begin{aligned} M &= 173459428887894878090016571735871156804145119477735621882800 5 \\ W &= 817674882378214554446156710260975983334661574759526517901788 \\ U &= 772733168866902463167387397046452128429080557734636757290817 \end{aligned}$$

Für die zugehörigen a_i's gilt:

i	a_i	$\log_2 a_i$
1	142841176795215894723318610202125782422921901437807004452777 3	199.8300977
2	109296009887671502614382260707913005666580571664885926026832 6	199.4439264
3	101419340795899844408170412863012593112138989909437267023626 5	199.3360185
4	25412213989625284969615198420204608104438707050392272449865 9	197.3392796
5	76812075062476018264875265656835948362055625843221349986169 1	198.9350906
98	50899388166817118162727478766207104069388964773050684777631 5	198.3414059
99	15410811066481314790348221948600458868645142461322328673725 77	199.9396285
100	56835959469020580154279924870329665342464413667088705264144 1	198.5005616

Damit berechnet man die k_i's:

i	k_i	$\log_2 k_i$
1	6363339018081336080995725753845688620791609001516950914153 07	198.6635415
2	4868957116167620248119615424173958839397627320076897715243 71	198.2773704
3	4518064489204436477474736695065181180225134318098402084133 76	198.1694625
4	1132072252861969601053887083243555977616133415276264807990 33	196.1727236
5	3421851354568345484215462831159197020767664118302971305254 63	197.7685346

Man erhält, wie zuvor angedeutet:

$$\log_2 |k_1a_2 - k_2a_1| = 99.8, \quad \log_2 |k_1a_3 - k_3a_1| = 101.4,$$
$$\log_2 |k_1a_4 - k_4a_1| = 102.8, \quad \log_2 |k_1a_5 - k_5a_1| = 103.7$$

4.3. Ein Gitter zur Bestimmung von $k_{i_1}, \ldots, k_{i_5}$. Wir behalten die früheren Bezeichnungen bei. Sei $\mu = 2^n$ und Λ das durch die Zeilen folgender Matrix definierte Gitter im $\mathbf{Z}^5$:

$$\begin{pmatrix} 1 & -\mu a_{i_2} & -\mu a_{i_3} & -\mu a_{i_4} & -\mu a_{i_5} \\ 0 & \mu a_{i_1} & 0 & 0 & 0 \\ 0 & 0 & \mu a_{i_1} & 0 & 0 \\ 0 & 0 & 0 & \mu a_{i_1} & 0 \\ 0 & 0 & 0 & 0 & \mu a_{i_1} \end{pmatrix}$$

Die Gitterelemente haben also die Gestalt

$$(x_1, \mu(a_{i_1}x_2 - a_{i_2}x_1), \mu(a_{i_1}x_3 - a_{i_3}x_1), \mu(a_{i_1}x_4 - a_{i_4}x_1), \mu(a_{i_1}x_5 - a_{i_5}x_1))$$

mit $x_1, \ldots, x_5 \in \mathbf{Z}$.

Für $x_1 = a_{i_1}, \ldots, x_5 = a_{i_5}$ erhält man den Gittervektor

$$(a_{i_1}, 0, 0, 0, 0) \quad \text{mit Länge} \quad a_{i_1} \approx 2^{2n}.$$

Für $x_1 = k_{i_1}, \ldots, x_5 = k_{i_5}$ erhält man den Gittervektor

$$(k_{i_1}, \mu(a_{i_1}k_{i_2} - a_{i_2}k_{i_1}), \mu(a_{i_1}k_{i_3} - a_{i_3}k_{i_1}), \mu(a_{i_1}k_{i_4} - a_{i_4}k_{i_1}), \mu(a_{i_1}k_{i_5} - a_{i_5}k_{i_1}))$$

mit Länge

$$\approx \sqrt{5 \cdot 2^{4n}} \approx 2^{2n}.$$

Ist $v_1, \ldots, v_5$ eine LLL-reduzierte Gitterbasis von Λ, so erhält man

$$\|v_1\| \leq 2^{\frac{5-1}{4}} \det(\Lambda)^{\frac{1}{5}} = 2(\mu a_{i_1})^{\frac{4}{5}} \approx (2^n \cdot 2^{2n})^{\frac{4}{5}} = 2^{2.4n}.$$

Die oben explizit angegeben Vektoren sind also deutlich kürzer als die allgemeine Abschätzung für v_1 angibt. Daher kann man erwarten, daß die Vektoren doch recht speziell sind. Wir haben nun folgendes Ergebnis:

Experimentelles Resultat: In den meisten getesteten Fällen ist

$$\pm(a_{i_1}, 0, 0, 0, 0)$$

der 1. Vektor einer LLL-reduzierten Gitterbasis von Λ.
In den meisten Fällen ist

$$\pm(k_{i_1}, \mu(a_{i_1}k_{i_2} - a_{i_2}k_{i_1}), \mu(a_{i_1}k_{i_3} - a_{i_3}k_{i_1}), \mu(a_{i_1}k_{i_4} - a_{i_4}k_{i_1}), \mu(a_{i_1}k_{i_5} - a_{i_5}k_{i_1}))$$

oder

$$\pm(a_{i_1} - k_{i_1}, -\mu(a_{i_1}k_{i_2} - a_{i_2}k_{i_1}), -\mu(a_{i_1}k_{i_3} - a_{i_3}k_{i_1}), -\mu(a_{i_1}k_{i_4} - a_{i_4}k_{i_1}), -\mu(a_{i_1}k_{i_5} - a_{i_5}k_{i_1}))$$

der 2. Vektor einer LLL-reduzierten Gitterbasis.

Fazit: Hat man einen öffentlichen Schlüssel $(a_1, \ldots, a_n)$ gegeben und vermutungsweise $(i_1, i_2, i_3, i_4, i_5)$, so daß $\pi(i_1) = 1, \ldots, \pi(i_5) = 5$ gilt, so kann man durch LLL-Reduktion meist zwei Möglichkeiten für $k_{i_1}, \ldots, k_{i_5}$ bestimmen.

Beispiel: Wir wählen einen zufälligen Schlüssel mit $n = 25$:

$$a = [852100675921734, 873488284681723, 348717847215955, 758677205019896, \ldots],$$

wobei wir uns bei der Konstruktion

$$(i_1, \ldots, i_5) = (17, 2, 3, 25, 21)$$

gemerkt haben. LLL-Gitterbasenreduktion führt dann auf folgende Möglichkeiten für $(k_{i_1}, k_{i_2}, k_{i_3}, k_{i_4}, k_{i_5})$:

```
ka=[334937279012136, 308881399907424, 123312995388387, 194405754336291, 145213308827423]
kb=[612234643311318, 564606884774299, 225404851827568, 355355898318605, 265436617256308]
```

Tatsächlich liefert die erste Möglichkeit die richtige Lösung.

Bemerkung: Obiges Verfahren haben wir in Maple programmiert, wobei allerdings für die LLL-Reduktion NTL benutzt wird. Die Rechenzeit ist dann auch für große Wert von n, z.B. $n = 200$, minimal, da der Rang des Gitters nur 5 ist.

Bemerkung: Warum wurde bei der Definition des Gitters Λ nicht der Faktor $\mu = 1$ verwendet? Im Fall $\mu = 1$ haben die explizit angegebenen Vektoren ebenfalls Länge $\approx 2^{2n}$. Für den 1. Vektor einer LLL-reduzierten Gitterbasis erhält man allerdings die Abschätzung:

$$\|v_1\| \leq 2^{\frac{5-1}{4}} \det(\Lambda)^{\frac{1}{5}} = 2(\mu a_{i_1})^{\frac{4}{5}} \approx (2^{2n})^{\frac{4}{5}} = 2^{1.6n}.$$

Die explizit angegeben Vektoren sind also deutlich länger als der 1. Vektor einer LLL-reduzierten Basis. Daher kann man nicht erwarten, sie in einer Gitterbasis wiederzufinden.

4.4. Bestimmung von möglichen (U, M)'s aus $(a_1, \ldots, a_n)$, $(i_1, \ldots, i_5)$ und $(k_{i_1}, \ldots, k_{i_5})$. Wir haben die Gleichungen

$$Ua_{i_j} - Mk_{i_j} = b_j.$$

Aus $0 < b_j < M$ erhält man $Mk_{i_j} < Ua_{i_j} < M(k_{i_j} + 1)$ und damit

$$\frac{k_{i_j}}{a_{i_j}} < \frac{U}{M} < \frac{k_{i_j} + 1}{a_{i_j}}.$$

Nun hat b_1, b_2, b_3, b_4, b_5 die superincreasing-Eigenschaft. Damit erhält man folgende (notwendige) Abschätzungen:

- $b_1 < b_2$ impliziert $Ua_{i_1} - Mk_{i_1} < Ua_{i_2} - Mk_{i_2}$ und damit

$$\frac{U}{M} < \frac{k_{i_1} - k_{i_2}}{a_{i_1} - a_{i_2}} \text{ für } a_{i_1} > a_{i_2} \quad \text{bzw.} \quad \frac{k_{i_1} - k_{i_2}}{a_{i_1} - a_{i_2}} < \frac{U}{M} \text{ für } a_{i_1} < a_{i_2}.$$

- $b_1 + b_2 < b_3$ liefert $Ua_{i_1} - Mk_{i_1} + Ua_{i_2} - Mk_{i_2} < Ua_{i_3} - Mk_{i_3}$, also $U(a_{i_1} + a_{i_2} - a_{i_3}) < M(k_{i_1} + k_{i_2} - k_{i_3})$ und damit

$$\frac{U}{M} < \frac{k_{i_1} + k_{i_2} - k_{i_3}}{a_{i_1} + a_{i_2} - a_{i_3}} \text{ für } a_{i_1} + a_{i_2} - a_{i_3} > 0 \quad \text{bzw.} \quad > \text{ andernfalls.}$$

- $b_1 + b_2 + b_3 < b_4$ liefert analog

$$\frac{U}{M} < \frac{k_{i_1} + k_{i_2} + k_{i_3} - k_{i_4}}{a_{i_1} + a_{i_2} + a_{i_3} - a_{i_4}} \text{ für } a_{i_1} + a_{i_2} + a_{i_3} - a_{i_4} > 0 \quad \text{bzw.} \quad > \text{ andernfalls.}$$

- $b_1 + b_2 + b_3 + b_4 < b_5$ liefert analog

$$\frac{U}{M} < \frac{k_{i_1} + k_{i_2} + k_{i_3} + k_{i_4} - k_{i_5}}{a_{i_1} + a_{i_2} + a_{i_3} + a_{i_4} - a_{i_5}} \text{ für } a_{i_1} + a_{i_2} + a_{i_3} + a_{i_4} - a_{i_5} > 0 \quad \text{bzw.} \quad > \text{ andernfalls.}$$

Wir wählen nun ein $\frac{U}{M}$, das den obigen Ungleichungen genügt, z.B. in der Mitte des zugehörigen Intervalls, und testen mit dem zuvor beschriebenen Verfahren, ob sich aus $(a_1, \ldots, a_n)$, U und M ein privater Merkle-Hellman-Schlüssel ergibt. Wenn ja, sind wir fertig.

Bemerkung: Wir haben zum beschriebenen Verfahren eine Maple-Funktion 'mehe_angriff5' geschrieben, bei der a und $(i_1, i_2, i_3, i_4, i_5)$ einzugeben sind. Das Verfahren funktionierte in den meisten Fällen, wenn $(i_1, \ldots, i_5)$ richtig angegeben war.

Beispiel: Für das oben angegebene Beispiel mit $n = 5$ lieferte die Funktion 'mehe_ma' bei Eingabe von $(a_1, \ldots, a_{25})$ und $(i_1, \ldots, i_5)$ einen möglichen zugehörigen privaten Merkle-Hellman-Schlüssel:

```
[[54778707650056058076152112763, 154909163830097617851994112532, [
    305375331546498995505O, 7799179800517622893081, 15598585679620048767781,
    36305670307617290538836, 7163053639739766219371 7, 147920521899819773783140,
    29325447807599578676663 5, 58839990784872493888784 1,
    11825339753894313880193 95, 23588323789324928883756 06,
    47265329882497132674383 46, 94533869614403733460584 00,
    18908498220995154036475374, 37817210175441156866307499,
    75638009806573019339026576, 15127473012352040005563958 7,
    30255732826897706064935666 1, 60511117464596849692628373 9,
    12102253432175330394116630 36, 24204554318834417367536617 14,
    48409109220040803126625455 01, 96818205628858887459232321 30,
    19363644437918829312828450160, 38727291721527506134938105931,
    77454582537292955856121050006], [10, 2, 3, 19, 18, 6, 25, 13, 11, 21, 23, 7,
    20, 12, 15, 8, 1, 16, 14, 22, 5, 9, 17, 24, 4]]
```

4.5. Der Allgemeinfall. Wir haben jetzt folgende Situation: Haben wir einen öffentlichen Merkle-Hellman-Schlüssel $(a_1, \ldots, a_n)$ und haben wir (vermutungsweise) 5 Indizes $(i_1, \ldots, i_5)$ mit den zuvor angegebenen Eigenschaften, so erhalten wir auf dem vorgestellten Weg in den meisten Fällen (schnell) einen zugehörigen privaten Merkle-Hellman-Schlüssel.
Leider kennen wir in der Praxis $(i_1, \ldots, i_5)$ nicht. Was kann man also machen?
Theoretisch kann man nun alle Möglichkeiten für $(i_1, \ldots, i_5)$ durchprobieren. Da dies weniger als n^5 Fälle sind, hat man so insgesamt einen polynomialen Algorithmus um aus einem öffentlichen Merkle-Hellman-Schlüssel einen privaten zu berechnen. Für die Theorie ist dies ein ausreichender Grund um die Merkle-Hellman-Verschlüsselung als unsicher einzustufen.
Es wäre schön, einen Weg zu finden, bei dem man nicht alle Möglichkeiten für $(i_1, \ldots, i_5)$ durchprobieren muß.

Literatur: [**MeOoVa**], [**Od**], [**BrOd**].

KAPITEL 3

Gitterangriffe auf Rucksäcke mit kleiner Dichte

Brickell, Lagarias, Odlyzko und andere haben Gitter ins Spiel gebracht um Rucksackprobleme $x_1a_1 + \cdots + x_na_n = s$ anzugehen. Über das Funktionieren der Algorithmen kann man theoretische Aussagen machen, wenn die sogenannte Dichte des Rucksacks

$$d = \frac{n}{\log_2 \max(a_1, \ldots, a_n)}$$

hinreichend klein ist. (Bei unseren Merkle-Hellman-Beispielen ist $\log_2 a_i \approx 2n$, also $d \approx 0.5$.) Außerdem sollte man die kürzesten Gittervektoren $\neq 0$ schnell bestimmen können. Leider ist hierfür kein allgemeiner Algorithmus bekannt. Wir verwenden hier den LLL-Algorithmus, der zwar nicht sicher den kürzesten Gittervektor $\neq 0$ liefert, aber dies praktisch doch oft tut.

1. Ein erster Ansatz mit einem Gitter

Gegeben sei $(a_1, \ldots, a_n)$ und s. Gesucht sind $x_i \in \{0, 1\}$ mit

$$s = x_1a_1 + \cdots + x_na_n.$$

Wir betrachten ein Gitter Λ, das durch die Zeilen folgender Matrix definiert wird:

$$\begin{pmatrix} 1 & 0 & \ldots & 0 & a_1 \\ 0 & 1 & \ldots & 0 & a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & a_n \\ 0 & 0 & \ldots & 0 & s \end{pmatrix}.$$

Multipliziert man die i-te Zeile mit $y_i \in \mathbf{Z}$ und addiert alles auf, so erhält man den Gittervektor

$$(y_1, y_2, \ldots, y_n, a_1y_1 + \cdots + a_ny_n + sy_{n+1}).$$

Hat man eine Lösung $\sum_i x_ia_i = s$ des Rucksackproblems, so ist (mit $y_i = x_i$, $y_{n+1} = -1$)

$$\pm(x_1, \ldots, x_n, 0)$$

ein Gittervektor der Länge $\leq \sqrt{n}$. Findet man umgekehrt einen Gittervektor der Gestalt

$$(z_1, \ldots, z_n, z_{n+1}) \quad \text{mit} \quad z_{n+1} = 0 \text{ und } z_i \in \{0, 1\} \text{ oder } z_i \in \{0, -1\},$$

so erhält man sofort eine Lösung des Rucksackproblems.
Wenn die Vektoren, die zu Lösungen des Rucksackproblems gehören, unter den kürzesten Vektoren des Gitters sind, kann man versuchen, diese mit Hilfe des LLL-Algorithmus zu bestimmen. Dies liefert dann folgendes Verfahren, wobei wir noch einen Faktor μ als Gewicht für die letzte Komponente einführen:

Algorithmus I: Gegeben seien natürliche Zahlen $a_1, \ldots, a_n$ und s. Gesucht sind $x_i \in \{0, 1\}$ mit $\sum_i x_ia_i = s$. Weiter sei eine natürliche Zahl μ gegeben, z.B. $\mu = 1$ oder $\mu = \lfloor\sqrt{n}\rfloor + 1$. Das Verfahren liefert eine Lösung $(x_1, \ldots, x_n)$ oder endet ohne eine Lösung gefunden zu haben.

1. Sei Λ das durch die Zeilen der Matrix
$$\begin{pmatrix} 1 & 0 & \dots & 0 & \mu a_1 \\ 0 & 1 & \dots & 0 & \mu a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & \mu a_n \\ 0 & 0 & \dots & 0 & \mu s \end{pmatrix}$$
definierte Gitter im $\mathbf{Z}^{n+1}$.
2. Bestimme eine LLL-reduzierte Basis $b_1, \dots, b_{n+1}$ des Gitters Λ (mit $b_i = (b_{i1}, \dots, b_{in}, b_{i,n+1})$).
3. Führe für $i = 1, \dots, n+1$ folgende Schritte durch:
   (a) Gilt $b_{i,n+1} = 0$ und $b_{i1}, \dots, b_{in} \in \{0, 1\}$ und $b_{i1}a_1 + \dots + b_{in}a_n = s$, so ist $(b_{i1}, \dots, b_{in})$ eine Lösung des Rucksackproblems und man ist fertig.
   (b) Gilt $b_{i,n+1} = 0$ und $b_{i1}, \dots, b_{in} \in \{0, -1\}$ und $-b_{i1}a_1 - \dots - b_{in}a_n = s$, so ist $(-b_{i1}, \dots, -b_{in})$ eine Lösung des Rucksackproblems und man ist fertig.

Bemerkung: Wir haben zu dem Verfahren eine Maple-Funktion 'lda_I0' geschrieben, die als Ergebnis eine Lösung oder 'false' liefert.

Beispiel: Wir wählen den öffentlichen Merkle-Hellman-Schlüssel der Länge $n = 16$

```
a := [2665785540, 1493149584, 1918851846, 2342932803, 2323532286, 785565665,
    521004450, 3668428933, 185031638, 2445017726, 80220278, 466206904,
    2180010352, 785441492, 4128238584, 3953701490]
```

Für jedes $m \in \{1, \dots, 16\}$ haben wir zufällig 100 Vektoren $x = (x_1, \dots, x_{16}) \in \{0, 1\}^{16}$ gewählt mit m Einträgen $x_i \neq 0$. Dann haben wir getestet, ob Algorithmus I mit dem Gewicht $\mu = 1$ wieder den Vektor x liefert. In der folgenden Tabelle gibt die untere Zahl, die Anzahl der Erfolge an:

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#Erfolge	100	100	100	100	100	100	100	100	100	99	100	96	99	100	100	100

Man sieht, daß das Verfahren für $m \leq \frac{n}{2}$ erfolgreich war.

Bemerkungen:

1. Angenommen, Algorithmus I liefert keine Lösung des Rucksackproblems $x_1a_1 + \dots + x_na_n = s$. Findet man mit Algorithmus I eine Lösung $(y_1, \dots, y_n)$ des Rucksackproblems
$$y_1a_1 + \dots + y_na_n = \sum_i a_i - s,$$
so folgt
$$\sum_i (1 - y_i)a_i = s,$$
also hat man eine Lösung des ursprünglichen Rucksackproblems.
2. Die angesprochene Möglichkeit wird in der Maple-Funktion 'lda_I1' ausgeführt. Die Funktion 'lda_I' probiert zunächst 'lda_I0', bei Misserfolg dann 'lda_I1'.
3. Für die folgenden Experimente haben wir uns daher auf zufällige 0-1-Vektoren $x = (x_1, \dots, x_n)$ mit $m \leq \frac{n}{2}$ Einträgen $\neq 0$ beschränkt.

Beispiel: Wir betrachten wieder einen öffentlichen Merkle-Hellman-Schlüssel der Länge $n = 16$:

```
a=[2411288408, 3801966757, 2554273793, 2836038305, 3966924636, 392111572,
   2667184438, 231301127, 1350534123, 3915623983, 2663468858, 2835327513,
   2454963321, 2954191847, 2794064667, 1145454372]
```

Durch Probieren finden wir, daß Algorithmus I bei folgendem Vektor

$$x = [1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1]$$

und $s = \sum_i x_i a_i$ keinen Erfolg hat. Die folgende Tabelle gibt eine LLL-reduzierte Basis $b_1, \ldots, b_{17}$ an, wobei die letzte Spalte $\|b_i\|^2$ enthält. (Gewicht $\mu = 1$)

1	0	0	1	-1	0	-1	-2	1	1	2	-1	0	-2	0	0	0	19
1	-1	-2	3	-3	0	4	2	0	1	-3	1	0	0	0	0	0	55
0	1	1	3	-3	0	1	-1	3	-1	-3	0	1	0	0	0	1	43
0	0	-2	1	4	1	0	-3	0	-2	1	-2	-1	0	0	0	-1	42
-1	0	-3	-1	-2	0	0	-1	0	2	1	2	2	0	0	0	1	30
-2	1	0	0	-2	5	-3	2	0	1	2	1	1	0	0	0	2	58
-1	-1	1	-2	3	-1	3	0	1	2	-2	-2	-1	-2	0	0	0	44
1	1	-1	-1	0	-1	-3	3	-2	-1	0	1	2	1	1	0	2	39
2	-1	2	1	-1	-1	1	0	1	2	-3	-2	0	0	-1	0	2	36
1	-1	2	-2	-1	0	-3	0	0	-1	3	0	0	2	1	1	1	37
0	1	1	-3	0	-2	-2	1	0	0	0	0	0	1	1	2	0	26
-1	0	1	-3	-1	-3	-2	1	-1	-1	0	0	0	0	1	1	0	30
-1	1	1	-2	1	3	0	2	-1	0	0	-1	2	-1	-1	1	2	34
0	1	0	0	-3	-1	2	1	0	2	0	0	-1	-1	1	-2	1	28
0	0	-2	1	-1	1	2	2	2	1	1	-1	-1	0	-1	-1	-1	26
0	0	-1	4	-1	-1	1	3	-1	0	1	0	0	1	2	2	0	40
1	1	-1	-1	-2	2	1	1	1	1	1	2	-1	-3	0	1	4	48

Man sieht, daß der Vektor x nicht vorkommt, obwohl $8 = \|x\|^2 < \|b_i\|^2$ für alle Basisvektoren b_i gilt. Nun gewichten wir mit $\mu = 2$ und erhalten folgende LLL-reduzierte Basis:

-1	-1	0	0	-1	-1	0	0	-1	-1	0	0	0	-1	0	-1	0	8
-1	0	0	-1	1	0	1	2	-1	-1	-2	1	0	2	0	0	0	19
-1	0	-3	-1	-2	0	0	-1	0	2	1	2	2	0	0	0	2	33
0	-1	-1	3	0	2	2	-1	0	0	0	0	0	-1	-1	-2	0	26
-1	0	-2	3	0	-1	-1	-2	1	1	-3	-2	-2	-1	0	-1	0	41
1	0	-3	1	1	-1	2	0	0	-1	-2	-1	-1	1	1	2	-2	34
1	-2	1	1	-1	2	-1	-1	0	-1	3	0	0	1	0	-1	2	30
0	0	2	-1	-4	-1	0	3	0	2	-1	2	1	0	0	0	2	45
0	0	1	1	0	1	2	-2	1	-5	1	0	2	0	0	0	0	42
0	0	2	-1	1	-1	-2	-2	-2	-1	-1	1	1	0	1	1	2	29
-2	-1	0	0	0	1	0	0	2	3	0	0	-2	-1	1	-1	0	26
0	1	-2	1	1	0	2	-2	0	0	1	-2	-2	-1	1	-2	0	30
-2	2	-2	1	-2	0	-2	1	1	2	-1	2	-1	0	1	0	-2	38
-2	2	0	-2	-1	-1	-1	-3	1	2	-2	1	1	2	-2	1	0	44
-2	2	-1	2	-2	-2	0	0	0	2	-1	1	1	3	0	3	0	46
-2	2	0	1	-2	2	-3	-2	1	0	-1	-2	-1	0	-1	2	0	42
0	1	0	-1	0	-1	-2	-1	2	1	0	4	-1	-3	-1	1	2	45

Der gesuchte Vektor x findet sich bis aufs Vorzeichen als erster Vektor b_1 wieder, d.h. Algorithmus I ist mit Gewicht $\mu = 2$ erfolgreich.

Beispiel: Wir probieren Algorithmus I mit einem öffentlichen Merkle-Hellman-Schlüssel der Länge $n = 32$:

```
a := [15279645034382642658, 831644695304570761, 5735702772626609108,

    12754594012356351307, 392273463265471421, 16301363229070607403,

    8909831757415271530, 3175742223975326548, 8582791342480973350,

    606435942484842997, 17002872034206559966, 11660544045510572566,

    17410689172919087440, 8835780101066444898, 9023549972380682175,
```

```
8816642781298157435, 13138683111191919672, 15158412943156336229,

1180267859151047803 3, 12319912987978631220, 9753888420897689250,

16823640581506788673, 4267022527651865634, 13027412197136314545,

18212294035847720380, 15068628333017550717, 11007240465970678571,

13065880725434981818, 8297327455045111378, 10628010114337374575,

7938802153134876067, 4291495073015094616]
```

Für $m \in \{1, \ldots, 16\}$ wählen wir jeweils 100 zufällige Vektoren $x = (x_1, \ldots, x_{32})$ mit $x_i \in \{0, 1\}$ und $\sum_i x_i^2 = m$ und testen mit verschiedenen Gewichten, ob Algorithmus I Erfolg hat. Die Tabelle gibt jeweils die Anzahl der Erfolge an:

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mu = 1$	100	100	100	100	100	99	94	95	87	77	80	62	62	57	61	51
$\mu = 2$	100	100	100	100	100	100	99	95	87	80	82	78	76	76	67	64
$\mu = 3$	100	100	100	100	100	99	97	95	82	77	76	69	66	62	52	55
$\mu = 4$	100	100	100	100	100	100	99	98	87	83	85	81	70	75	62	63
$\mu = 5$	100	100	100	100	100	100	99	98	90	85	82	78	76	70	60	70
$\mu = 6$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 7$	100	100	100	100	100	100	99	99	89	86	86	80	78	77	70	66
$\mu = 8$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 9$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 10$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 11$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 12$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 13$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 14$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 15$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 16$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 17$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 18$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 19$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 20$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 21$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 22$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 23$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 24$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68
$\mu = 25$	100	100	100	100	100	100	99	98	89	80	83	79	74	72	63	68

Man sieht, daß es kein optimales Gewicht für alle Möglichkeiten von m gibt.

Bemerkung: Man kann folgendes zeigen: Könnte man in Algorithmus I den LLL-Algorithmus durch einen Algorithmus ersetzen, der mit polynomialer Laufzeit, d.h. schnell, den kürzesten Gittervektor $\neq 0$ liefert, und ist die Dichte $d = \frac{n}{\log_2 \max(a_1, \ldots, a_n)} < 0.6463\ldots$, so ist es sehr wahrscheinlich, daß Algorithmus I das Rucksackproblem bei Wahl von $\mu > \sqrt{n}$ löst.

2. Ein weiteres Gitter

Wir betrachten nun ein etwas modifiziertes Gitter, das durch die Zeilen folgender Matrix erzeugt wird:

$$\begin{pmatrix} 2 & 0 & \dots & 0 & \mu a_1 \\ 0 & 2 & \dots & 0 & \mu a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 2 & \mu a_n \\ 1 & 1 & \dots & 1 & \mu s \end{pmatrix}.$$

Durch Multiplikation der Matrix von links mit $(y_1, y_2, \dots, y_n, y_{n+1}) \in \mathbf{Z}^{n+1}$ erhält man den Gittervektor:

$$(2y_1 + y_{n+1}, 2y_2 + y_{n+1}, \dots, 2y_n + y_{n+1}, \mu \sum_i y_i a_i + \mu y_{n+1} s).$$

Ist $\sum_i x_i a_i = s$, so erhält man durch Wahl von $y_1 = x_1, \dots, y_n = x_n, y_{n+1} = -1$ den Vektor

$$(2x_1 - 1, 2x_2 - 1, \dots, 2x_n - 1, 0).$$

Außer dem letzten Eintrag sind alle Einträge ± 1.
Sei umgekehrt ein Gittervektor gegeben mit

$$(z_1, z_2, \dots, z_n, z_{n+1}) \quad \text{und} \quad z_{n+1} = 0 \text{ und } z_1, \dots, z_n \in \{1, -1\}.$$

Wir bilden nun

$$(\frac{z_1 + 1}{2}, \frac{z_2 + 1}{2}, \dots, \frac{z_n + 1}{2}) \quad \text{und} \quad (\frac{-z_1 + 1}{2}, \frac{-z_2 + 1}{2}, \dots, \frac{-z_n + 1}{2})$$

und testen, ob wir eine Lösung von $\sum_i x_i a_i = s$ erhalten.

Algorithmus II: Gegeben seien natürliche Zahlen $a_1, \dots, a_n$ und s. Gesucht sind $x_i \in \{0, 1\}$ mit $\sum_i x_i a_i = s$. Weiter sei eine natürliche Zahl μ gegeben, z.B. $\mu = 1$, $\mu = 2$ oder $\mu = \lfloor \sqrt{n} \rfloor + 1$. Das Verfahren liefert eine Lösung $(x_1, \dots, x_n)$ oder endet ohne eine Lösung gefunden zu haben.

1. Sei Λ das durch die Zeilen der Matrix

$$\begin{pmatrix} 2 & 0 & \dots & 0 & \mu a_1 \\ 0 & 2 & \dots & 0 & \mu a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 2 & \mu a_n \\ 1 & 1 & \dots & 1 & \mu s \end{pmatrix}$$

   definierte Gitter im $\mathbf{Z}^{n+1}$.
2. Bestimme eine LLL-reduzierte Basis $b_1, \dots, b_{n+1}$ des Gitters Λ (mit $b_i = (b_{i1}, \dots, b_{in}, b_{i,n+1})$).
3. Führe für $i = 1, \dots, n+1$ folgende Schritte durch:
   (a) Gilt $b_{i,n+1} = 0$ und $b_{i1}, \dots, b_{in} \in \{-1, 1\}$ und $\frac{b_{i1}+1}{2} a_1 + \dots + \frac{b_{in}+1}{2} a_n = s$, so ist $(\frac{b_{i1}+1}{2}, \dots, \frac{b_{in}+1}{2})$ eine Lösung des Rucksackproblems und man ist fertig.
   (b) Gilt $b_{i,n+1} = 0$ und $b_{i1}, \dots, b_{in} \in \{-1, 1\}$ und $\frac{-b_{i1}+1}{2} a_1 + \dots + \frac{-b_{in}+1}{2} a_n = s$, so ist $(\frac{-b_{i1}+1}{2}, \dots, \frac{-b_{in}+1}{2})$ eine Lösung des Rucksackproblems und man ist fertig.

Bemerkung: Zu Algorithmus II haben wir eine Maple-Funktion 'lda_II' geschrieben.

Bevor wir zu den Experimenten kommen, machen wir noch folgende theoretische Bemerkung:

Bemerkung: Man kann folgendes zeigen: Könnte man in Algorithmus II den LLL-Algorithmus durch einen Algorithmus ersetzen, der mit polynomialer Laufzeit den kürzesten Gittervektor $\neq 0$ liefert, und ist die Dichte $d = \frac{n}{\log_2 \max(a_1, \dots, a_n)} < 0.9408...$, so ist es sehr wahrscheinlich, daß Algorithmus II das Rucksackproblem bei Wahl von $\mu > \sqrt{n}$ löst. Dies ist eine Verbesserung gegenüber Algorithmus I.

Beispiel: Wir wählen einen öffentlichen Merkle-Hellman-Schlüssel der Länge $n = 32$ und wählen dann für jedes $m \in \{1, \dots, 16\}$ zufällig 100 Vektoren $x = (x_1, \dots, x_{32})$ mit $x_i \in \{0, 1\}$ und $\sum_i x_i^2 = m$. Dann

wird getestet, ob Algorithmus II das Rucksackproblem mit Parametern $(a_1, \ldots, a_n)$ und $\sum_i x_i a_i$ löst. In der Tabelle ist jeweils die Anzahl der erfolgreichen Versuche angegeben.

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mu = 1$	100	99	94	89	82	81	78	68	70	69	77	76	80	82	84	78
$\mu = 2$	100	98	94	98	88	93	92	95	91	97	99	99	96	97	96	96
$\mu = 3$	100	98	96	98	94	95	94	92	95	97	100	97	99	99	98	99
$\mu = 4$	100	98	94	99	96	95	97	97	99	98	100	98	100	99	100	99
$\mu = 5$	100	98	96	99	97	96	97	95	99	99	100	99	100	99	100	100
$\mu = 6$	100	97	94	96	89	91	90	95	89	99	97	94	100	98	95	99
$\mu = 7$	100	98	94	96	93	92	90	92	93	95	94	98	97	98	98	98
$\mu = 8$	100	98	95	98	95	96	96	96	94	98	100	98	100	99	98	98
$\mu = 9$	100	99	97	98	98	95	97	97	97	96	100	98	99	100	98	100
$\mu = 10$	100	97	91	96	92	93	92	97	95	98	99	99	100	100	100	100
$\mu = 11$	100	97	96	97	94	95	95	98	95	100	100	98	100	100	100	100
$\mu = 12$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 13$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 14$	100	97	96	97	94	95	95	98	95	100	100	98	100	100	100	100
$\mu = 15$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 16$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 17$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 18$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 19$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 20$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 21$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 22$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 23$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 24$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 25$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 26$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 27$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 28$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 29$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 30$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 31$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100
$\mu = 32$	100	97	93	97	91	93	93	97	95	99	100	98	100	100	100	100

3. Vergleichende Experimente

Wir wollen jetzt die Algorithmen I (I0 und I1) und II vergleichen. Zunächst wurde ein öffentlicher Merkle-Hellman-Schlüssel $(a_1, \ldots, a_n)$ der Länge n gewählt. Dann wurden 100 0-1-Vektoren $x = (x_1, \ldots, x_n)$ zufällig gewählt. Die Zahl m bezeichnet die Anzahl der Einträge $x_i \neq 0$. In Klammern bei I0, I1 und II steht das Gewicht μ. Die Einträge 'ja/nein' geben Auskunft, ob der Algorithmus erfolgreich war oder nicht. Die letzte Zeile enthält die Gesamtzahl der erfolgreichen Versuche.

Beispiel: $n = 32$.

m	I0(1)	I1(1)	II(1)	II(2)	I0($\lfloor\sqrt{n}\rfloor+1$)	I1($\lfloor\sqrt{n}\rfloor+1$)	II($\lfloor\sqrt{n}\rfloor+1$)
26	nein	ja	ja	ja	ja	ja	ja
4	ja	ja	ja	ja	ja	ja	ja
22	nein	ja	nein	ja	ja	ja	ja
4	ja	ja	ja	ja	ja	ja	ja
13	ja	nein	ja	ja	ja	ja	ja
10	nein	ja	ja	ja	nein	ja	ja
27	ja	ja	ja	ja	ja	ja	ja
18	ja	nein	ja	ja	ja	ja	ja
19	nein	ja	ja	ja	ja	ja	ja
28	ja	ja	ja	ja	ja	ja	ja
2	ja	ja	ja	ja	ja	ja	ja
1	ja	ja	ja	ja	ja	ja	ja
8	ja	nein	ja	ja	ja	ja	ja
14	nein	ja	ja	ja	nein	ja	ja
6	ja	ja	nein	nein	ja	ja	nein
26	ja	ja	ja	ja	ja	ja	ja
4	ja	ja	ja	ja	ja	ja	ja
32	ja	nein	ja	ja	ja	nein	ja
5	ja	nein	ja	ja	ja	nein	ja
26	ja	ja	ja	ja	ja	ja	ja
30	ja	ja	ja	ja	ja	ja	ja
5	ja	ja	ja	ja	ja	ja	ja
7	ja	ja	nein	ja	ja	ja	ja
7	ja	ja	ja	ja	ja	ja	ja
9	ja	ja	ja	ja	ja	ja	ja
15	ja	nein	nein	ja	ja	nein	ja
13	ja	nein	ja	ja	ja	nein	ja
14	nein	nein	ja	ja	ja	nein	ja
13	ja	ja	ja	ja	ja	ja	ja
17	nein	ja	ja	ja	nein	ja	ja
3	ja	ja	ja	ja	ja	ja	ja
2	ja	ja	ja	ja	ja	ja	ja
1	ja	ja	ja	ja	ja	ja	ja
5	ja	ja	nein	ja	ja	ja	ja
27	nein	ja	nein	ja	ja	ja	nein
5	ja	nein	ja	nein	ja	nein	ja
17	nein	nein	ja	ja	nein	nein	ja
30	ja	ja	ja	ja	ja	ja	ja
25	ja	ja	ja	ja	ja	ja	ja
15	nein	ja	ja	ja	nein	ja	ja
8	ja	nein	nein	ja	ja	nein	ja
18	ja	ja	ja	ja	ja	ja	ja
22	ja	ja	ja	ja	ja	ja	ja
7	ja	nein	ja	ja	ja	nein	ja
10	ja	nein	ja	ja	ja	nein	ja
4	ja	ja	ja	ja	ja	ja	ja
29	ja	ja	ja	ja	ja	ja	ja
5	ja	nein	ja	nein	ja	nein	ja
19	nein	ja	ja	ja	ja	ja	ja
24	ja	ja	ja	ja	ja	ja	ja
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	70	70	79	93	85	79	98

Beispiel: $n = 100$

m	I0(1)	I1(1)	II(1)	II(2)	I0($\lfloor\sqrt{n}\rfloor+1$)	I1($\lfloor\sqrt{n}\rfloor+1$)	II($\lfloor\sqrt{n}\rfloor+1$)
82	nein	nein	nein	nein	nein	nein	nein
32	nein	nein	nein	nein	nein	nein	nein
86	nein	nein	ja	ja	nein	nein	ja
70	nein	nein	nein	nein	nein	nein	nein
83	nein	nein	ja	ja	nein	nein	ja
79	nein	nein	nein	nein	nein	nein	nein
69	nein	nein	nein	nein	nein	nein	nein
93	nein	ja	nein	nein	nein	ja	nein
44	nein	nein	nein	nein	nein	nein	nein
85	nein	ja	ja	ja	nein	ja	ja
62	nein	ja	nein	nein	nein	ja	nein
89	nein	nein	nein	nein	nein	nein	nein
91	nein	ja	ja	ja	nein	ja	ja
61	nein	nein	nein	nein	nein	nein	nein
95	nein	ja	ja	ja	nein	ja	ja
94	nein	ja	ja	ja	nein	ja	ja
16	nein	nein	nein	nein	nein	nein	nein
70	nein	ja	ja	ja	nein	ja	ja
31	nein	nein	nein	nein	nein	nein	nein
71	nein	ja	ja	ja	nein	ja	ja
27	nein	nein	ja	ja	nein	nein	ja
15	nein	nein	ja	ja	nein	nein	ja
73	nein	nein	nein	nein	nein	nein	nein
37	ja	nein	ja	ja	ja	nein	ja
13	ja	nein	ja	ja	ja	nein	ja
97	nein	ja	ja	ja	nein	ja	ja
56	nein	nein	nein	nein	nein	nein	nein
58	nein	nein	nein	nein	nein	nein	nein
49	nein	nein	nein	nein	nein	nein	nein
19	nein	nein	nein	nein	nein	nein	nein
57	nein	nein	nein	nein	nein	nein	nein
27	nein	nein	nein	nein	nein	nein	nein
59	nein	nein	nein	nein	nein	nein	nein
29	nein	nein	nein	nein	nein	nein	nein
43	nein	nein	nein	nein	nein	nein	nein
6	ja	nein	ja	ja	ja	nein	ja
31	nein	nein	ja	ja	nein	nein	ja
83	nein	nein	ja	ja	nein	nein	ja
37	nein	nein	nein	nein	nein	nein	nein
58	nein	nein	nein	nein	nein	nein	nein
91	nein	ja	ja	ja	nein	ja	ja
4	ja	nein	ja	ja	ja	nein	ja
20	ja	nein	nein	nein	ja	nein	nein
91	nein	ja	nein	nein	nein	ja	nein
61	nein	nein	nein	nein	nein	nein	nein
4	ja	nein	ja	ja	ja	nein	ja
95	nein	ja	ja	ja	nein	ja	ja
1	ja	nein	ja	ja	ja	nein	ja
57	nein	nein	ja	ja	nein	nein	ja
20	nein	nein	nein	nein	nein	nein	nein
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	17	17	43	43	17	17	43

Beispiel: $n = 200$, $d = 0.4998763788$.

m	I0(1)	I1(1)	II(1)	II(2)	I0($\lfloor\sqrt{n}\rfloor+1$)	I1($\lfloor\sqrt{n}\rfloor+1$)	II($\lfloor\sqrt{n}\rfloor+1$)
82	nein	nein	nein	nein	nein	nein	nein
106	nein	nein	nein	nein	nein	nein	nein
110	nein	nein	nein	nein	nein	nein	nein
117	nein	nein	nein	nein	nein	nein	nein
112	nein	nein	nein	nein	nein	nein	nein
144	nein	nein	nein	nein	nein	nein	nein
12	ja	nein	ja	ja	ja	nein	ja
117	nein	nein	nein	nein	nein	nein	nein
10	ja	nein	ja	ja	ja	nein	ja
125	nein	nein	nein	nein	nein	nein	nein
65	nein	nein	nein	nein	nein	nein	nein
138	nein	nein	ja	ja	nein	nein	ja
194	nein	ja	ja	ja	nein	ja	ja
103	nein	nein	nein	nein	nein	nein	nein
138	nein	nein	nein	nein	nein	nein	nein
51	nein	nein	nein	nein	nein	nein	nein
160	nein	nein	nein	nein	nein	nein	nein
139	nein	nein	nein	nein	nein	nein	nein
27	nein	nein	ja	ja	nein	nein	ja
2	ja	nein	ja	ja	ja	nein	ja
200	nein	nein	ja	ja	nein	nein	ja
148	nein	nein	nein	nein	nein	nein	nein
167	nein	nein	nein	nein	nein	nein	nein
9	ja	nein	ja	ja	ja	nein	ja
2	ja	nein	nein	nein	ja	nein	nein
118	nein	nein	nein	nein	nein	nein	nein
185	nein	ja	nein	nein	nein	ja	nein
10	nein	nein	nein	nein	nein	nein	nein
175	nein	nein	nein	nein	nein	nein	nein
95	nein	nein	nein	nein	nein	nein	nein
49	nein	nein	nein	nein	nein	nein	nein
160	nein	nein	nein	nein	nein	nein	nein
75	nein	nein	nein	nein	nein	nein	nein
91	nein	nein	nein	nein	nein	nein	nein
122	nein	nein	nein	nein	nein	nein	nein
74	nein	nein	ja	ja	nein	nein	ja
111	nein	nein	nein	nein	nein	nein	nein
131	nein	nein	nein	nein	nein	nein	nein
108	nein	nein	nein	nein	nein	nein	nein
103	nein	nein	nein	nein	nein	nein	nein
178	nein	nein	nein	nein	nein	nein	nein
120	nein	nein	nein	nein	nein	nein	nein
181	nein	ja	nein	nein	nein	ja	nein
108	nein	nein	nein	nein	nein	nein	nein
18	ja	nein	ja	ja	ja	nein	ja
46	nein	nein	nein	nein	nein	nein	nein
153	nein	nein	nein	nein	nein	nein	nein
148	nein	nein	nein	nein	nein	nein	nein
34	nein	nein	nein	nein	nein	nein	nein
150	nein	nein	nein	nein	nein	nein	nein
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	11	13	25	25	11	13	25

Beispiel: $n = 300$, $d = 0.5002105527$.

m	I0(1)	I1(1)	II(1)	II(2)	I0($\lfloor\sqrt{n}\rfloor+1$)	I1($\lfloor\sqrt{n}\rfloor+1$)	II($\lfloor\sqrt{n}\rfloor+1$)
259	nein	ja	nein	nein	nein	ja	nein
126	nein	nein	nein	nein	nein	nein	nein
202	nein	nein	nein	nein	nein	nein	nein
276	nein	nein	ja	ja	nein	nein	ja
191	nein	nein	nein	nein	nein	nein	nein
176	nein	nein	nein	nein	nein	nein	nein
11	ja	nein	ja	ja	ja	nein	ja
178	nein	nein	nein	nein	nein	nein	nein
295	nein	ja	ja	ja	nein	ja	ja
198	nein	nein	nein	nein	nein	nein	nein
15	ja	nein	ja	ja	ja	nein	ja
255	nein	ja	nein	nein	nein	ja	nein
169	nein	nein	nein	nein	nein	nein	nein
146	nein	nein	nein	nein	nein	nein	nein
223	nein	nein	nein	nein	nein	nein	nein
235	nein	nein	nein	nein	nein	nein	nein
237	nein	nein	nein	nein	nein	nein	nein
85	nein	nein	nein	nein	nein	nein	nein
287	nein	nein	nein	nein	nein	nein	nein
229	nein	nein	nein	nein	nein	nein	nein
186	nein	nein	ja	ja	nein	nein	ja
13	nein	nein	nein	nein	nein	nein	nein
38	nein	nein	nein	nein	nein	nein	nein
214	nein	nein	nein	nein	nein	nein	nein
89	nein	nein	nein	nein	nein	nein	nein
224	nein	nein	nein	nein	nein	nein	nein
44	nein	nein	nein	nein	nein	nein	nein
222	nein	nein	nein	nein	nein	nein	nein
151	nein	nein	nein	nein	nein	nein	nein
32	nein	nein	nein	nein	nein	nein	nein
108	nein	nein	nein	nein	nein	nein	nein
49	nein	nein	nein	nein	nein	nein	nein
235	nein	nein	nein	nein	nein	nein	nein
58	nein	nein	nein	nein	nein	nein	nein
292	nein	ja	nein	nein	nein	ja	nein
52	nein	nein	nein	nein	nein	nein	nein
148	nein	nein	nein	nein	nein	nein	nein
153	nein	nein	nein	nein	nein	nein	nein
14	nein	nein	nein	nein	nein	nein	nein
152	nein	nein	nein	nein	nein	nein	nein
34	ja	nein	nein	nein	ja	nein	nein
288	nein	ja	ja	ja	nein	ja	ja
168	nein	nein	nein	nein	nein	nein	nein
136	nein	nein	nein	nein	nein	nein	nein
263	nein	nein	nein	nein	nein	nein	nein
262	nein	nein	nein	nein	nein	nein	nein
204	nein	nein	nein	nein	nein	nein	nein
232	nein	nein	nein	nein	nein	nein	nein
141	nein	nein	nein	nein	nein	nein	nein
151	nein	nein	nein	nein	nein	nein	nein

Man merkt, daß die praktische Erfolgswahrscheinlichkeit der Algorithmen I und II bei Zunahme von n deutlich abnimmt.

4. Anwendung auf die Merkle-Hellman-Verschlüsselung

Kennt man einen öffentlichen Merkle-Hellman-Schlüssel $a = (a_1, \ldots, a_n)$ und eine Nachricht, die damit zu $c_1, c_2, c_3, \ldots$ verschlüsselt wurde, so kann man nun mit den Algorithmen II und I versuchen, die Rucksackprobleme

$$x_{i1}a_1 + \cdots + x_{in} = c_i$$

zu lösen und damit die Ausgangsnachricht

$$x_{11}x_{12}\ldots x_{1n}x_{21}x_{22}\ldots x_{2n}x_{31}x_{32}\ldots x_{3n}\ldots$$

zu rekonstruieren.

Beispiel: Wir beginnen mit dem Ausgangstext:

```
B. Schneier, Angewandte Kryptographie (Addison-Wesley 1996)
19.2 Rucksackalgorithmen
Der erste Algorithmus fuer verallgemeinerte Public-Key-Verschluesselung
war der Rucksackalgorithmus (knapsack algorithm), der von Ralph Merkle
und Martin Hellman entwickelt wurde. Er konnte nur zur Verschluesselung
benutzt werden. Adi Shamir modifizierte das System jedoch spaeter fuer
den Einsatz bei digitalen Signaturen. Die Sicherheit von
Rucksackalgorithmen beruht auf dem Rucksackproblem, einem
NP-vollstaendigen Problem. Obwohl sich dieser Algorithmus spaeter als
unsicher erwies, ist er eine genauere Betrachtung wert, da er zeigt, wie
man ein NP-vollstaendiges Problem fuer Public-Key-Kryptographie
einsetzen kann.
```

Nun wählen wir (zufällig) öffentliche Merkle-Hellman-Schlüssel $a = (a_1, \ldots, a_n)$, verschlüsseln den Ausgangstext damit und versuchen dann den verschlüsselten Text mit obigem Ansatz zu entschlüsseln. Bei Misserfolg wurde '** ... *' eingefügt, Zeilenumbrüche wurden bei Bedarf eingefügt.

Mit einem Merkle-Hellman-Schlüssel der Länge $n = 32$:

```
B. Schneier, Angewandte Kryptographie (Addison-Wesley 1996)
19.2 Rucksackalgorithmen
Der erste Algorithmus fuer verallgemeinerte Public-Key-Verschluesselung
war der Rucksackalgorithmus (knapsack algorithm), der von Ralph Merkle
und Martin Hellman entwickelt wurde. Er konnte nur zur Verschluesselung
benutzt werden. Adi Shamir modifizierte das System jedoch spaeter fuer
den Einsatz bei digitalen Signaturen. Die Sicherheit von
****sackalgorithmen beruht auf dem Rucksackproblem, einem
NP-vollstaendigen Problem. Obwohl sich dieser Algorithmus spaeter als
unsicher erwies, ist er eine genauere Betrachtung wert, da er zeigt, wie
man ein NP-vollstaendiges Problem fuer Public-Key-Kryptographie
einsetzen kann.
```

Mit einem Merkle-Hellman-Schlüssel der Länge $n = 64$:

```
B. Schneier, Angewandte Kryptographie (Addison-Wesley 1996)
19.2 Rucksackalgorithmen
Der erste Algorithmus fuer verallgemeinerte Public-Key-Verschluesse********
der Rucksackalgorithmus (knapsack algorithm), der von Ralph Merkle
und Martin Hellman entwickelt wurde.********te nur zur Verschluesselung
benutzt werden. Adi Shamir modifizierte das System jedoch spaeter fuer
den Einsatz bei digitalen Signaturen. Die Sicherheit ********sackalgorithmen
beruht auf dem Rucksackproblem, einem NP-vollstaendigen Problem.
Obwohl sich dieser Algorithmus spaeter als unsicher erwies, ist er
eine genauer********htung wert, da e**************** ein
```

```
NP-********ndiges Problem fuer Public-Key-Kryptographie einsetzen kann.
```

Mit einem Merkle-Hellman-Schlüssel der Länge $n = 128$:

```
****************ewandte Kryptogr***************esley 1996)
19.2****************hmen
Der erste A*************************************************************************
****************************************er von Ralph Mer*********************
******************************************hluesselung
benu*********************************************************************************
****ei digitalen Sig*******************************sackalgorithmen ************
********************einem NP-vollsta*************** Obwohl sich dieser Algorithmus
*********************************************************************************
wie man ein NP-************************************************phie
einsetzen k***************
```

Mit einem Merkle-Hellman-Schlüssel der Länge $n = 256$:

```
*********************************************************************************
****************lgorithmus fuer verallgemeinerte*******************************
*********************************************************************************
*********************************************************************************
*********************************************************************************
*********************************************************************************
*********************************************************************************
*********************************************************************************
*********************************************************************ann.
```

5. Theoretische Überlegungen zu den Angriffen auf Rucksäcke mit kleiner Dichte

Jedem $a = (a_1, \ldots, a_n) \in \mathbf{N}^n$ und $u = (u_1, \ldots, u_n) \in \{0,1\}^n \setminus \{0\}$ ordnen wir ein Gitter $\Lambda(a,u) \subseteq \mathbf{Z}^n$ zu, das durch die Zeilen der Matrix

$$\begin{pmatrix} 1 & 0 & \ldots & 0 & \mu a_1 \\ 0 & 1 & \ldots & 0 & \mu a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & \mu a_n \\ 0 & 0 & \ldots & 0 & \mu s \end{pmatrix}$$

gegeben wird, wobei wir $\mu \in \mathbf{N}$ mit $\mu > \sqrt{n}$ fest gewählt haben, z.B. $\mu = \lfloor \sqrt{n} \rfloor + 1$, und $s = s(a,u) = a_1 u_1 + \cdots + a_n u_n$. Es gilt dann

$$\Lambda(a,u) = \{(z_1, z_2, \ldots, z_n, \mu(z_1 a_1 + \cdots + z_n a_n + z_{n+1} s)) : z_1, \ldots, z_n, z_{n+1} \in \mathbf{Z}\}.$$

Wählt man hier $z_1 = u_1, \ldots, z_n = u_n, z_{n+1} = -1$, so sieht man sofort

$$(u_1, \ldots, u_n, 0) \in \Lambda(a,u) \quad \text{und} \quad \|(u_1, \ldots, u_n, 0)\| = \|(u_1, \ldots, u_n)\| \leq \sqrt{n}.$$

Wir führen für $x = (x_1, \ldots, x_n) \in \mathbf{Z}^n$ die Notation $\hat{x} = (x_1, \ldots, x_n, 0) \in \mathbf{Z}^{n+1}$ ein.

Bemerkung: Uns interessant, wann $\hat{u} = (u_1, \ldots, u_n, 0)$ der kürzeste Vektor in $\Lambda(a,u)$ ist. Wir wollen daher die Menge

$$\Lambda(a,u) \cap \{x \in \mathbf{Z}^{n+1} : \|x\| \leq \|\hat{u}\|\}$$

betrachten, die natürlich die Vektoren 0, $\hat{u}$ und $-\hat{u}$ enthält. Bei Anwendung des LLL-Algorithmus ist auch interessant, unter welchen Bedingungen man $(u_1, \ldots, u_n, 0)$ in einer reduzierten Basis finden kann.

LEMMA. *Wir setzen $\mu > \sqrt{n}$ voraus.*

1. *Ist $x = (x_1, \ldots, x_n, x_{n+1}) \in \Lambda(a,u)$ und $x_{n+1} \neq 0$, so ist*

$$\|\hat{u}\| \leq \sqrt{n} < \mu \leq \|x\|,$$

insbesondere $x \notin \Lambda(a,u) \cap \{y \in \mathbf{Z}^{n+1} : \|y\| \leq \|\hat{u}\|\}$.

2.

$$\Lambda(a,u) \cap \{y \in \mathbf{Z}^{n+1} : \|y\| \leq \|\hat{u}\|\} = \{\hat{x} : x \in \mathbf{Z}^n \text{ mit } \|x\| \leq \|u\| \text{ und es gibt } \lambda \in \mathbf{Z} \text{ mit } \sum x_i a_i = \lambda s(a,u)\}.$$

Beweis:

1. Wegen $x_{n+1} \equiv 0 \bmod \mu$ folgt mit $x_{n+1} \neq 0$
$$\|(x_1, \ldots, x_n, x_{n+1})\| \geq |x_{n+1}| \geq \mu > \sqrt{n} \geq \|(u_1, \ldots, u_n, 0)\|.$$
2. Dies folgt aus 1. und der Charakterisierung der Elemente von $\Lambda(a,u)$. ■

Beispiel: Sei $a \in \mathbf{N}^n$ mit $a_3 = a_1 + a_2$ und $u = (1,1,0,\ldots,0) \in \{0,1\}^n$, $s = s(a,u) = a_1 + a_2 = a_3$. Für $x = (0,0,1,0,\ldots,0) \in \mathbf{Z}^{n+1}$ gilt dann

$$x \in \Lambda(a,u) \quad \text{und } \|x\| < \|\hat{u}\|,$$

d.h. $\pm\hat{u}$ ist nicht der kürzeste nichttriviale Vektor in $\Lambda(a,u)$.

Das Beispiel macht deutlich, daß man nicht erwarten kann, daß $\pm\hat{u}$ die kürzesten Vektoren des Gitters $\Lambda(a,u)$ sind. Wir werden im folgenden nicht ein einzelnes Gitter anschauen, sondern eine Menge von Gittern.

Für $n \in \mathbf{N}$ und $A \in \mathbf{N}$ betrachten wir

$$R(n,A) = \{((a_1,\ldots,a_n),(x_1,\ldots,x_n)) : a_i \in \{1,\ldots,A\}, x_i \in \{0,1\}, x \neq 0\}.$$

Die Menge $R(n,A)$ parametrisiert also eine Menge von Gittern $\Lambda(a,u)$. Interessant ist die Teilmenge der Gitter, bei denen $\pm\hat{u}$ nicht die kürzesten nichttrivialen Vektoren sind. Daher definieren wir die Menge

$$R_*(n,A) = \{(a,u) \in R(n,A) : \Lambda(a,u) \cap \{x \in \mathbf{Z}^{n+1} : \|x\| \leq \|u\|\} \neq \{0, \pm\hat{u}\}\}.$$

Mit obigem Lemma erhalten wir:

$$\begin{aligned} R_*(n,A) &= \{(a,u)) \in R(n,A), \text{ es gibt } x = (x_1,\ldots,x_n) \in \mathbf{Z}^n \text{ und } \lambda \in \mathbf{Z} \\ &\quad \text{mit } \sum a_i x_i = \lambda(\sum_i a_i u_i), \|x\| \leq \|u\| \text{ und } x \notin \{0,u,-u\}\} = \\ &= \{(a,u)) \in R(n,A), \text{ es gibt } x = (x_1,\ldots,x_n) \in \mathbf{Z}^n \text{ und } \lambda \in \mathbf{Z} \\ &\quad \text{mit } \sum a_i(x_i - \lambda u_i) = 0, \|x\| \leq \|u\| \text{ und } x \notin \{0,u,-u\}\}. \end{aligned}$$

Wir wollen abschätzen, wie oft ein Gitter $\Lambda(a,u)$ einen kürzeren Vektor als $\hat{u}$ enthält, d.h. wir interessieren uns für den Quotienten

$$\frac{\#R_*(n,A)}{\#R(n,A)},$$

wobei natürlich

$$\#R(n,A) = A^n \cdot (2^n - 1)$$

gilt. Leider scheint nicht klar zu sein, wie man $\#R_*(n,A)$ direkt abschätzen kann. Daher wird folgende Aufteilung von $R(n,A)$ eingeführt:

$$\begin{aligned} R_<(n,A) &= \{(a,u) \in R(n,A) : \sum a_i u_i < \frac{1}{n}\sum a_i\}, \\ R_\geq(n,A) &= \{(a,u) \in R(n,A) : \sum a_i u_i \geq \frac{1}{n}\sum a_i\}, \end{aligned}$$

Bemerkung: Für $(a,u) \in R(n,A)$ erhält man die Abschätzung

$$0 < \frac{\sum_i a_i u_i}{\sum_i a_i} \leq 1.$$

Daher sollte man erwarten, daß die Fälle mit $\frac{\sum_i a_i u_i}{\sum_i a_i} < \frac{1}{n}$ eher selten sind. Als Aufgabe zeige man, daß

$$\lim_{n\to\infty} \frac{\#R_<(n,A)}{\#R(n,A)} = 0$$

gilt. Dies bedeutet, daß es sehr unwahrscheinlich ist, bei zufälliger Wahl von a und u auf einen Fall mit $s(a,u) < \frac{1}{n}\sum_i a_i$ zu stoßen.
Wir haben etwas experimentiert:

n	Anzahl der Fälle mit $s < \frac{1}{n}\sum a_i$ bei jeweils 1000 Versuchen
2	284, 302, 309, 272, 266, 317, 306, 299, 287, 276
3	220, 209, 201, 217, 209, 220, 213, 178, 222, 229
4	148, 160, 131, 161, 152, 140, 174, 143, 150, 129
5	97, 96, 126, 97, 108, 111, 130, 100, 88, 114
6	61, 72, 63, 69, 68, 66, 76, 80, 60, 73
7	57, 44, 53, 45, 48, 51, 40, 52, 47, 38
8	36, 24, 38, 33, 24, 20, 35, 31, 25, 25
9	15, 15, 17, 22, 13, 25, 17, 19, 22, 14
10	12, 10, 9, 6, 7, 7, 6, 10, 10, 11
11	6, 6, 8, 8, 5, 6, 8, 10, 3, 4
12	1, 6, 3, 6, 3, 4, 2, 3, 3, 3
13	1, 5, 2, 0, 2, 0, 1, 1, 4, 1
14	0, 1, 2, 2, 2, 3, 2, 0, 0, 1
15	1, 1, 0, 0, 1, 1, 1, 0, 1, 0
16	0, 1, 1, 0, 1, 0, 1, 0, 0, 0
17	0, 0, 0, 2, 0, 0, 0, 0, 0, 1
18	0, 0, 0, 0, 1, 2, 0, 0, 0, 1
19	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
20	1, 0, 0, 0, 0, 0, 0, 0, 0, 0

Wir geben noch eine weitere Überlegung an: $\overline{a} = \frac{1}{n}\sum_i a_i$ ist der Mittelwert der a_i's. Ist nun $\sum_i a_i u_i < \overline{a}$, so folgt

$$a_i \geq \overline{a} \quad \Longrightarrow \quad u_i = 0,$$

d.h. alle a_i's mit $a_i \geq \overline{a}$ sind in diesem Fall uninteressant. Damit wird das Rucksackproblem auf ein Rucksackproblem kleinerer Dimension reduziert.
Aus diesen Gründen beschränken wir uns im folgenden auf die Teilmenge $R_\geq(n,A) \subseteq R(n,A)$.

Im folgenden bezeichnet $K_n(r)$ die Kugel vom Radius r im $\mathbf{R}^n$, d.h. $K_n(r) = \{x \in \mathbf{R}^n : \|x\| \leq r\}$.

Lemma.

$$\frac{\#R_*(n,A) \cap R_\geq(n,A)}{\#R(n,A)} \leq \frac{1}{A} \cdot (2n\sqrt{n}+1) \cdot \#(\mathbf{Z}^n \cap K_n(\sqrt{n})).$$

Beweis: Sei $(a,u) \in R_*(n,A) \cap R_\geq(n,A)$. Dann gibt es $x \in \mathbf{Z}^n$ und $\lambda \in \mathbf{Z}$ mit

$$\sum_i a_i(x_i - \lambda u_i) = 0, \quad \|x\| \leq \|u\|, \quad x \notin \{0, \pm u\}, \quad s(a,u) \geq \frac{1}{n}\sum_i a_i.$$

Wir erhalten

$$|\lambda| = \frac{|\sum_i a_i x_i|}{|\sum_i a_i u_i|} \leq \frac{|\sum_i a_i x_i|}{\frac{1}{n}\sum_i a_i} \leq n\frac{\sum_i a_i |x_i|}{\sum_i a_i} \leq n\frac{\sum_i a_i \|x\|}{\sum_i a_i} = n\|x\| \leq n\|u\| \leq n\sqrt{n}.$$

Also folgt

$$(a,u) \in \{(a,u) \in R(n,A) \quad : \quad \text{es gibt } x \in \mathbf{Z}^n \cap K_n(\sqrt{n}) \text{ und } \lambda \in \mathbf{Z} \text{ mit } |\lambda| \leq n\sqrt{n} \text{ mit } \sum_i a_i(x_i - \lambda u_i) = 0, \quad x \neq \lambda u\}.$$

Wir definieren für $u \in \{0,1\}^n \setminus \{0\}$, $x \in \mathbf{Z}^n \cap K_n(\sqrt{n})$ und $\lambda \in \mathbf{Z}$ mit $|\lambda| \le n\sqrt{n}$

$$S(u,x,\lambda) = \{a \in \mathbf{Z}^n : 1 \le a_i \le A, \sum_i a_i(x_i - \lambda u_i) = 0\}.$$

Für $x \neq \lambda u$ folgt $\#S(u,x,\lambda) \le A^{n-1}$, denn ist $x_i - \lambda u_i \neq 0$, so kann man sich $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ vorgeben und daraus dann a_i bestimmen. Wir erhalten

$$R_*(n,A) \cap R_{\ge}(n,A) \subseteq \bigcup_{u \in \{0,1\}^n \setminus \{0\}} \bigcup_{\substack{\lambda \in Z \\ |\lambda| \le n\sqrt{n}}} \bigcup_{\substack{x \in \mathbf{Z}^n \cap K_n(\sqrt{n}) \\ x \neq \lambda u}} \{(a,u) : a \in S(u,x,\lambda)\}.$$

Damit folgt

$$\#R_*(n,A) \cap R_{\ge}(n,A) \le (2^n - 1) \cdot (2n\sqrt{n} + 1) \cdot \#(\mathbf{Z}^n \cap K_n(\sqrt{n})) \cdot A^{n-1},$$

was mit $\#R(n,A) = A^n \cdot (2^n - 1)$ sofort die Behauptung liefert. ■

Wir geben jetzt (ohne Beweis) eine Abschätzung für $\#\mathbf{Z}^n \cap K_n(\sqrt{n})$ an:

LEMMA. *Für alle (hinreichend großen)* $n \in \mathbf{N}$ *gilt*

$$\#(\mathbf{Z}^n \cap K_n(\sqrt{n})) \le 2^{2.047099n}.$$

Beweis: Sei

$$N(n) = \#\mathbf{Z}^n \cap K_n(\sqrt{n}) = \#\{x \in \mathbf{Z}^n : \|x\| \le \sqrt{n}\}.$$

Die ersten Werte kann man explizit bestimmen:

n	$N(n)$	$\frac{\log_2 N(n)}{n}$
1	3	1.584963
2	9	1.584963
3	27	1.584963
4	89	1.618933
5	333	1.675876
6	1341	1.731516
7	5449	1.773111
8	21697	1.800651
9	84663	1.818827

Nach Mazo/Odlyzko gilt:

$$\lim_{n \to \infty} \frac{\log_2 N(n)}{n} \le 2.047096,$$

was dann die Behauptung liefert. ■

Zusammenfassend erhalten wir daher für alle hinreichend großen $n \in \mathbf{N}$

$$\frac{\#R_*(n,A) \cap R_{\ge}(n,A)}{\#R(n,A)} \le \frac{1}{A} \cdot (2n\sqrt{n} + 1) \cdot 2^{2.047099n}.$$

Damit erhält man folgenden Satz:

SATZ. *Ist* $A \ge 2^{2.0471n}$, *so folgt*

$$\lim_{n \to \infty} \frac{\#R_*(n,A) \cap R_{\ge}(n,A)}{\#R(n,A)} = 0.$$

Wählt man zufällig $(a,u) \in R(n,A)$, wo wird $A \approx \max(a_1, \ldots, a_n)$ gelten. Wir überlegen daher für $A = \max(a_1, \ldots, a_n)$:

$$\begin{aligned} A \ge 2^{2.0471n} &\iff \log_2 \max(a_1, \ldots, a_n) \ge 2.0471n \\ &\iff \frac{n}{\log_2 \max(a_1, \ldots, a_n)} \le \frac{1}{2.0471} \approx 0.4884959211. \end{aligned}$$

Interpretation: Ist n (hinreichend) groß, ist $(a,u) \in R(n,A)$ zufällig gewählt, so daß für die Dichte des Rucksacks

$$d(a_1,\ldots,a_n) \leq 0.488$$

gilt, so ist es sehr wahrscheinlich, daß $\pm\hat{u}$ die kürzesten Vektoren $\neq 0$ des Gitters $\Lambda(a,u)$ sind.

Bemerkungen:

1. Der Algorithmus I wurde so erweitert, daß nicht nur das Gitter $\Lambda(a,u)$, sondern auch $\Lambda(a,(1,\ldots,1)-u)$ verwendet wird. Wegen
$$\|u\|^2 \leq \frac{n}{2} \quad \text{oder} \quad \|(1,\ldots,1)-u\|^2 \leq \frac{n}{2}$$
kann man sich dann auf u's mit $\|u\| \leq \sqrt{\frac{n}{2}}$ beschränken. Die obigen Abschätzungen verbessern sich dann zu $d(a_1,\ldots,a_n) < 0.6463\ldots$. Für den Algorithmus II erhält man die Bedingung $d(a_1,\ldots,d_n) < 0.9408\ldots$.
2. Die angegebenen Abschätzungen hängen von der Dichte des Rucksacks ab, nicht jedoch von der Dimension. Wie kann man sich dann erklären, daß bei den durchgeführten Experimenten das Gelingen von Algorithmus I und II stark von der Dimension n abhängt? (Bei wachsendem n funktionieren die Algorithmen immer seltener.)

Wir wollen jetzt ähnlich wie eben untersuchen, wann der LLL-Algorithmus bei Anwendung auf das Gitter $\Lambda(a,u)$ den Vektor u als den ersten reduzierten Gittervektor liefert.

LEMMA. *Gilt*

$$\Lambda(a,u) \cap K_{n+1}(2^{n/2}\sqrt{n}) = \{0,\hat{u},-\hat{u}\}.$$

d.h. sind $0,\pm\hat{u}$ die einzigen Gittervektoren mit einer Länge $\leq 2^{n/2}\sqrt{n}$, und ist $b_1,\ldots,b_{n+1}$ eine LLL-reduzierte Gitterbasis, so ist $b_1 = \pm\hat{u}$.

Beweis: Für alle $x \in \Lambda(a,u)$, $x \neq 0$ gilt die Abschätzung

$$\|b_1\| \leq 2^{n/2}\|x\|,$$

insbesondere folgt

$$\|b_1\| \leq 2^{n/2}\|\hat{u}\| \leq 2^{n/2}\sqrt{n}.$$

Da nach unserer Voraussetzung $0,\pm\hat{u}$ die einzigen Gittervektoren mit einer Länge $\leq 2^{n/2}\sqrt{n}$ sind, folgt $b_1 = \pm\hat{u}$, was zu zeigen war. ■

Wählen wir jetzt $\mu > 2^{n/2}\sqrt{n}$, so erhalten wir auf gleiche Weise wie früher:

LEMMA.

$$\Lambda(a,u) \cap K_{n+1}(2^{n/2}\sqrt{n}) = \{\hat{x} : x \in \mathbf{Z}^n, \lambda \in \mathbf{Z}, x \neq \lambda u, \sum_i a_i(x_i - \lambda u_i) = 0\}.$$

Wir definieren jetzt die Teilmenge von $R(n,A)$, bei der man nach obigem Lemma nicht sicher sein kann, daß der LLL-Algorithmus $\pm\hat{u}$ als ersten Vektor einer reduzierten Basis liefert.

$$\begin{aligned} R_*^{\mathrm{LLL}}(n,A) &= \{(a,u) \in R(n,A) : \Lambda(a,u) \cap K_{n+1}(2^{n/2}\sqrt{n}) \neq \{0,\hat{u},-\hat{u}\}\} = \\ &= \{(a,u) \in R(n,A) : \text{ es gibt } x \in \mathbf{Z}^n \text{ mit } \|x\| \leq 2^{n/2}\sqrt{n} \text{ und } \lambda \in \mathbf{Z} \text{ mit } x \neq \lambda u \\ &\qquad \text{und } \sum_i a_i(x_i - \lambda u_i) = 0\} \end{aligned}$$

Wie zuvor beschränken wir uns auf Elemente von $R_{\geq}(n,A)$ und erhalten das Lemma:

LEMMA.

$$\frac{\#R_*^{\mathrm{LLL}}(n,A) \cap R_{\geq}(n,A)}{\#R(n,A)} \leq \frac{1}{A} \cdot (2 \cdot 2^{n/2} n\sqrt{n} + 1) \cdot \#(\mathbf{Z}^n \cap K_n(2^{n/2}\sqrt{n})).$$

Beweis: Der Beweis verläuft wie der frühere. Ist $(a,u) \in R_*^{\mathrm{LLL}}(n,A) \cap R_{\geq}(n,A)$ und $x \in \mathbf{Z}^n$, $\lambda \in \mathbf{Z}$ mit $\|x\| \leq 2^{n/2}\sqrt{n}$, $x - \lambda u \neq 0$, $\sum_i a_i(x_i - \lambda u_i) = 0$, so folgt

$$|\lambda| \leq 2^{n/2} n\sqrt{n},$$

was den Faktor $2 \cdot 2^{n/2} n\sqrt{n} + 1$ liefert. ■

LEMMA.

$$\#\mathbf{Z}^n \cap K_n(2^{n/2}\sqrt{n}) \leq (2 \cdot 2^{n/2}\sqrt{n} + 1)^n.$$

Beweis: Ist $x \in K_n(2^{n/2}\sqrt{n})$, so ist x Element des Würfels

$$\{(x_1, \ldots, x_n) \in \mathbf{R}^n : |x_i| \leq 2^{n/2}\sqrt{n}\}.$$

Das das Intervall $-y..y$ höchstens $2y + 1$ ganze Zahlen enthält, folgt daraus sofort die angegebene Abschätzung. ■

Damit kann man jetzt abschätzen:

$$\begin{aligned}
\frac{\#R_*^{\mathrm{LLL}}(n,A) \cap R_{\geq}(n,A)}{\#R(n,A)} &\leq \frac{1}{A} \cdot (2 \cdot 2^{n/2}\sqrt{n} + 1)^n \cdot (2 \cdot 2^{n/2} n\sqrt{n} + 1) \leq \\
&\leq \frac{1}{A} \cdot (4 \cdot 2^{n/2}\sqrt{n})^n \cdot (4 \cdot 2^{n/2} n\sqrt{n}) = \\
&= \frac{1}{A} \cdot (2^{2 + \frac{n}{2} + \frac{1}{2}\log_2 n})^n \cdot 2^{2 + \frac{n}{2} + \frac{3}{2}\log_2 n} = \\
&= \frac{1}{A} \cdot 2^{2n + \frac{1}{2}n^2 + \frac{1}{2}n\log_2 n + 2 + \frac{1}{2}n + \frac{3}{2}\log_2 n}
\end{aligned}$$

und erhält

LEMMA. *Ist $\varepsilon > 0$ und $A \geq 2^{(\frac{1}{2}+\varepsilon)n^2}$, so gilt*

$$\lim_{n\to\infty} \frac{\#R_*^{\mathrm{LLL}}(n,A) \cap R_{\geq}(n,A)}{\#R(n,A)} = 0.$$

Was bedeutet $A \geq 2^{(\frac{1}{2}+\varepsilon)n^2}$ mit $A = \max(a_1, \ldots, a_n)$?

$$A \geq 2^{(\frac{1}{2}+\varepsilon)n^2} \quad \Longleftrightarrow \quad \log_2 \max(a_1, \ldots, a_n) \geq (\frac{1}{2} + \varepsilon)n^2 \quad \Longleftrightarrow \quad d(a_1, \ldots, a_n) \leq \frac{2}{n(1+2\varepsilon)}.$$

Interpretation: Man kann sicher sein, daß der LLL-Algorithmus sehr wahrscheinlich $\pm\hat{u}$ als den ersten reduzierten Gittervektor liefert, falls für die Dichte des Rucksacks $d(a_1, \ldots, a_n) \lessapprox \frac{2}{n}$ gilt.

Bemerkung: Da bei unseren Merkle-Hellman-Beispielen die Rucksackdichte ≈ 0.5 war, wird verständlich, weshalb für wachsendes n die Algorithmen I und II immer seltener Erfolg haben.

Literatur: [**Od**], [**CoLaOdSc**], [**MaOd**], [**Sc**].

KAPITEL 4

Das Chor-Rivest-Verschlüsselungsverfahren

1. Der Satz von Bose-Chowla

Gegeben seien natürliche Zahlen n und h. Wir sagen, eine Folge natürlicher Zahlen $a_0, \ldots, a_{n-1}$ hat die Eigenschaft $S(n,h)$, wenn gilt: Sind $x_i, y_i \in \mathbf{N}_0$ mit $\sum_{i=0}^{n-1} x_i = \sum_{i=0}^{n-1} y_i = h$ und $(x_0, x_1, \ldots, x_{n-1}) \neq (y_0, y_1, \ldots, y_{n-1})$, so folgt $\sum_{i=0}^{n-1} x_i a_i \neq \sum_{i=0}^{n-1} y_i a_i$, d.h. die Summen von je h Zahlen aus $\{a_0, \ldots, a_{n-1}\}$ sind alle verschieden, wobei Wiederholungen erlaubt sind.
Definieren wir für das Tripel $(n, h, (a_0, \ldots, a_{n-1}))$ eine Abbildung

$$\begin{aligned} f_{(n,h,(a_0,\ldots,a_{n-1}))} : \{(x_0, x_1, \ldots, x_{n-1}) \in \mathbf{N}_0^n : \sum_{i=0}^{n-1} x_i = h\} &\to \mathbf{Z}, \\ (x_0, x_1, \ldots, x_{n-1}) &\mapsto x_0 a_0 + x_1 a_1 + \cdots + x_{n-1} a_{n-1}, \end{aligned}$$

so hat die Folge $a_0, \ldots, a_{n-1}$ genau dann die Eigenschaft $S(n,h)$, wenn $f_{(n,h,(a_0,\ldots,a_{n-1}))}$ injektiv ist.

Bemerkungen:

1. Offensichtlich hat $a_0, \ldots, a_{n-1}$ genau dann die Eigenschaft $S(n,1)$, wenn alle a_i's verschieden sind.
2. Man sieht leicht folgende Implikationen:

$$S(n,h) \implies S(n,h-1) \implies S(n,h-2) \implies \ldots \implies S(n,2) \implies S(n,1).$$

3. $a_0 \in \mathbf{N}$ hat für alle $h \geq 1$ die Eigenschaft $S(1,h)$.
4. Für $n = 2$ sieht man aus dem Ansatz $x_0 + x_1 = h$ und

$$f_{(2,h,(a_0,a_1))}(x_0, x_1) = x_0 a_0 + x_1 a_1 = x_0 a_0 + (h - x_0) a_1 = x_0 (a_0 - a_1) + h a_1,$$

daß a_0, a_1 genau dann die Eigenschaft $S(2,h)$ hat, wenn $a_0 \neq a_1$ gilt.

Der folgende Satz gibt ein Beispiel einer Folge mit der Eigenschaft $S(n,h)$:

SATZ. *Seien $n, h \in \mathbf{N}$ gegeben mit $h \geq 2$ und*

$$a_0 = 1, \quad a_1 = h, \quad a_2 = h^2, \quad \ldots, \quad a_{n-1} = h^{n-1}.$$

Dann hat die Folge $a_0, a_1, \ldots, a_{n-1}$ die Eigenschaft $S(n,h)$, nicht jedoch die Eigenschaft $S(n,h+1)$ für $n \geq 3$.

Beweis: Wir nehmen zunächst an, die Folge $a_0, \ldots, a_{n-1}$ besitzt nicht die Eigenschaft $S(n,h)$. Dann gibt es $x_i, y_i \in \mathbf{N}_0$ mit $\sum_i x_i = \sum_i y_i = h$, $(x_0, \ldots, x_{n-1}) \neq (y_0, \ldots, y_{n-1})$ und

$$x_0 + x_1 h + x_2 h^2 + \cdots + x_{n-1} h^{n-1} = y_0 + y_1 h + y_2 h^2 + \cdots + y_{n-1} h^{n-1}.$$

Also findet sich ein Index $0 \leq k \leq n-1$ mit $x_0 = y_0$, $x_1 = y_1$, $\ldots$, $x_{k-1} = y_{k-1}$ und $x_k \neq y_k$. Durch Subtraktion von $x_0 + x_1 h + \cdots + x_{k-1} h^{k-1} = y_0 + y_1 h + \cdots + y_{k-1} h^{k-1}$ und Division durch h^k folgt die Gleichung

$$x_k + x_{k+1} h + x_{k+2} h^2 + \cdots + x_{n-1} h^{n-1-k} = y_k + y_{k+1} h + y_{k+2} h^2 + \cdots + y_{n-1} h^{n-1-k}.$$

Modulo h erhält man $x_k \equiv y_k \bmod h$. Wegen $0 \le x_k, y_k \le h$ und $x_k \neq y_k$ folgt o.E. $x_k = h$ und $y_k = 0$. Aus $\sum_i x_i = h$ ergibt sich $x_0 = \cdots = x_{k-1} = x_{k+1} = \cdots = x_{n-1} = 0$, insbesondere $y_0 = \cdots = y_{k-1} = y_k = 0$, und damit

$$h^{k+1} = h \cdot h^k = \sum_{i=0}^{n-1} x_i h^i = \sum_{i=0}^{n-1} y_i h^i = y_{k+1}h^{k+1} + \cdots + y_{n-1}h^{n-1} \ge (y_{k+1} + \cdots + y_{n-1})h^{k+1} = h \cdot h^{k+1},$$

ein Widerspruch. Also war die Annahme falsch, d.h. die Folge $a_0, \ldots, a_{n-1}$ hat die Eigenschaft $S(n,h)$. Andererseits gilt

$$(h+1)a_1 = (h+1)h = h + h^2 = h \cdot 1 + 1 \cdot h^2 = h \cdot a_0 + 1 \cdot a_2,$$

also hat $a_0, \ldots, a_{n-1}$ nicht die Eigenschaft $S(n, h+1)$, sofern $n \ge 3$ ist. ■

LEMMA. *Die Folge $a_0, \ldots, a_{n-1}$ habe die Eigenschaft $S(n,h)$. Dann gilt*

$$\max(a_0, \ldots, a_{n-1}) \ge \begin{cases} \frac{1}{h}\binom{n+h-1}{h} = \frac{(n+h-1)(n+h-2)\ldots(n+1)n}{h \cdot h!} \ge \frac{1}{h \cdot h!} n^h \\ \frac{1}{h}\binom{h+n-1}{n-1} = \frac{(h+n-1)(h+n-2)\ldots(h+1)}{h \cdot (n-1)!} \ge \frac{1}{(n-1)!} h^{n-2}. \end{cases}$$

Beweis: Schreiben wir abkürzend

$$M_{n,h} = \{(x_0, \ldots, x_{n-1}) \in \mathbf{N}_0^n : \sum_i x_i = h\},$$

dann besagt Eigenschaft $S(n,h)$, daß die früher definierte Abbildung $f_{(n,h,(a_0,\ldots,a_{n-1}))} : M_{n,h} \to \mathbf{Z}$ injektiv ist. Also hat man

$$\{f_{(n,h,(a_0,\ldots,a_{n-1}))}(x) : x \in M_{n,h}\} \subseteq \mathbf{N} \quad \text{und} \quad \#\{f_{(n,h,(a_0,\ldots,a_{n-1}))}(x) : x \in M_{n,h}\} = \#M_{n,h}$$

und damit

$$\max\{f_{(n,h,(a_0,\ldots,a_{n-1}))}(x) : x \in M_{n,h}\} \ge \#M_{n,h}.$$

Nun gilt

$$\begin{aligned} f_{(n,h,(a_0,\ldots,a_{n-1}))}((x_0, \ldots, x_{n-1})) &= x_0a_0 + x_1a_1 + \cdots + x_{n-1}a_{n-1} \le \\ &\le (x_0 + x_1 + \cdots + x_{n-1}) \max(a_0, a_1, \ldots, a_{n-1}) = \\ &= h \max(a_0, a_1, \ldots, a_{n-1}) \end{aligned}$$

und

$$\#M_{n,h} = \#\{(x_0, x_1, \ldots, x_{n-1}) \in \mathbf{N}_0^n : \sum_{i=0}^{n-1} x_i = h\} = \binom{h+n-1}{h} = \binom{h+n-1}{n-1},$$

was zur Abschätzung

$$\begin{aligned} \max(a_0, \ldots, a_{n-1}) &= \frac{1}{h} \cdot h \max(a_0, \ldots, a_{n-1}) \ge \frac{1}{h} \max\{f_{(n,h,(a_0,\ldots,a_{n-1}))}(x) : x \in M_{n,h}\} \ge \\ &\ge \frac{1}{h}\#M_{n,h} = \frac{1}{h}\binom{n+h-1}{h} = \frac{1}{h}\binom{h+n-1}{n-1} \end{aligned}$$

führt. Dies zeigt die Behauptung. ■

Bemerkung: Das Lemma zeigt, daß für Folgen $a_0, \ldots, a_{n-1}$ mit der Eigenschaft $S(n,h)$ das Maximum $\max(a_0, \ldots, a_{n-1})$ bei festem h wegen

$$\max(a_0, \ldots, a_{n-1}) \ge \frac{1}{h \cdot h!} n^h$$

mindestens polynomial in n wächst, bei festem n ist das Wachstum wegen

$$\max(a_0, \ldots, a_{n-1}) \ge \frac{1}{(n-1)!} h^{n-2}$$

mindestens polynomial in h. Das Beispiel $a_0 = 1, a_1 = h, \ldots, a_{n-1} = h^{n-1}$ des Satzes liefert wegen $\max(a_0, \ldots, a_{n-1}) = h^{n-1}$ ein Beispiel für polynomiales Wachstum in h. Die Frage ist nun: Kann man

auch Folgen $a_0, \ldots, a_{n-1}$ mit der Eigenschaft $S(n,h)$ konstruieren, so daß $\max(a_0, \ldots, a_{n-1})$ nur polynomial in n wächst. Der folgende Satz bzw. der Beweis des Satzes gibt hierfür eine Konstruktionsmöglichkeit (im Spezialfall $n = p$ mit einer Primzahl p):

SATZ (Bose-Chowla). *Sei p eine Primzahl und h eine natürliche Zahl. Dann gibt es eine Folge $a_0, a_1, \ldots, a_{p-1}$ natürlicher Zahlen, die die Eigenschaft $S(p,h)$ hat und die Ungleichung*

$$1 \le a_i \le p^h - 1 \text{ für } i = 0, \ldots, p-1$$

erfüllt.

Beweis: Im endlichen Körper $\mathbf{F}_{p^h}$ wählen wir ein Element ξ vom Grad h über $\mathbf{F}_p$, d.h. $\mathbf{F}_{p^h} = \mathbf{F}_p[\xi]$ und ein Element γ, das Ordnung $p^h - 1$ in der multiplikativen Gruppe $\mathbf{F}_{p^h}^*$ hat, also diese erzeugt. Daher gibt es für jedes i mit $0 \le i \le p-1$, d.h. $i \in \mathbf{F}_p$, eine Zahl a_i mit $1 \le a_i \le p^h - 1$ und

$$\gamma^{a_i} = \xi + i.$$

Angenommen wir haben jetzt $\sum_{i=0}^{p-1} x_i a_i = \sum_{i=0}^{p-1} y_i a_i$ mit $x_i, y_i \in \mathbf{N}_0$ und $\sum_i x_i = \sum_i y_i = h$. Wir definieren Polynome $f(x), g(x) \in \mathbf{F}_p[x]$ durch

$$f(x) = \prod_{i=0}^{p-1} (x+i)^{x_i}, \quad g(x) = \prod_{i=0}^{p-1} (x+i)^{y_i}.$$

$f(x)$ und $g(x)$ bestehen aus jeweils h Linearfaktoren der Gestalt $x + i$, insbesondere sind also $f(x)$ und $g(x)$ normierte Polynome vom Grad h. Nun folgt

$$f(\xi) = \prod_{i=0}^{p-1} (\xi+i)^{x_i} = \gamma^{\sum_{i=0}^{p-1} x_i a_i} = \gamma^{\sum_{i=0}^{p-1} y_i a_i} = \prod_{i=0}^{p-1} (\xi+i)^{y_i} = g(\xi).$$

$f(x) - g(x)$ ist ein Polynom vom Grad $\le h-1$ mit Koeffizienten in $\mathbf{F}_p$, das ξ als Nullstelle hat. Da aber ξ Grad h über $\mathbf{F}_p$ hat, kann $f(x) - g(x)$ nur das Nullpolynom sein. Also ist $f(x) = g(x)$. Die Eindeutigkeit der Primfaktorzerlegung im Polynomring $\mathbf{F}_p[x]$ liefert dann $(x_0, \ldots, x_{p-1}) = (y_0, \ldots, y_{p-1})$, was gezeigt werden sollte. ■

Beispiel: Wir wählen $p = 11$, $h = 4$ und ξ mit dem Minimalpolynom $f(x) = x^4 + 7x^3 + 5x^2 + 6x + 2$. Mit $\gamma = \xi$ erhalten wir $\gamma^{a_i} = \xi + i$ für $0 \le i \le 10$ mit

$$a_0 = 1, \quad a_1 = 12179, \quad a_2 = 108, \quad a_3 = 766, \quad a_4 = 4746,$$

$$a_5 = 1950, \quad a_6 = 12367, \quad a_7 = 2447, \quad a_8 = 4954, \quad a_9 = 307, \quad a_{10} = 6085.$$

Die Folge $a_0, \ldots, a_{10}$ hat also die Eigenschaft $S(11,4)$.

Chor und Rivest haben Folgen $a_0, \ldots, a_{p-1}$, wie sie im Beweis des Satzes von Bose-Chowla konstruiert wurden, benutzt, um daraus eine Rucksackverschlüsselung zu basteln.

2. Kombinatorische Vorbereitung

Die im Beweis des Satzes von Bose-Chowla konstruierten Folgen $a_0, \ldots, a_{p-1}$ haben die Eigenschaft $S(p,h)$, d.h.

$$\{(x_0, \ldots, x_{p-1}) \in \mathbf{N}_0^p : \sum_{i=0}^{p-1} x_i = h\} \to \mathbf{Z}, \quad (x_0, \ldots, x_{p-1}) \mapsto x_0 a_0 + \cdots + x_{p-1} a_{p-1}$$

ist injektiv. Um aus $a_0, \ldots, a_{p-1}$ wie früher eine Rucksackverschlüsselung zu erhalten, bräuchten wir die Injektivität der Abbildung

$$\{0,1\}^p \to \mathbf{Z}, \quad (x_0, \ldots, x_{p-1}) \mapsto x_0 a_0 + \cdots + x_{p-1} a_{p-1},$$

was man aber im allgemeinen nicht erwarten kann. Wir können nur die Injektivität von

$$\{(x_0, \ldots, x_{p-1}) \in \{0,1\}^p : \sum_i x_i = h\} \to \mathbf{Z}, \quad (x_0, \ldots, x_{p-1}) \mapsto x_0 a_0 + \cdots + x_{p-1} a_{p-1}$$

folgern. Daher werden wir uns bei der Verschlüsselung auf Bitfolgen

$$(x_0, \ldots, x_{p-1}) \in \{0,1\}^p \text{ mit } \sum_i x_i = h,$$

d.h. Bitfolgen der Länge p mit Hamming-Gewicht h, beschränken.
Die Frage ist nun, wie man Text in Bitfolgen der Länge p mit genau h Einsen, also in Folgen

$$M = (M_0 M_1 \ldots M_{p-1}) \in \{0,1\}^p \quad \text{mit} \quad \sum_{i=1}^{p} M_i = h$$

umsetzt. Dazu überlegen wir folgendes:

Es gibt genau $\binom{p}{h}$ binäre Folgen der Länge p mit Hamming-Gewicht h. Daher gibt es eine Bijektion

$$\{m \in \mathbf{N} : 0 \le m \le \binom{p}{h} - 1\} \quad \longleftrightarrow \quad \{(M_0 M_1 \ldots M_{p-1}) \in \{0,1\}^p : \sum_{i=1}^{p} M_i = h\}.$$

Indem man links die Ordnungsrelation der natürlichen Zahlen, rechts die lexikographische Anordnung benutzt, erhält man eine explizite Bijektion der beiden Mengen, die wir mit $T_{p,h}$ bezeichnen.

Beispiel: Wir betrachten den Fall $p = 5$ und $h = 2$ mit $\binom{p}{h} = 10$. Die folgende Tabelle gibt die Bijektion wieder:

$T_{5,2}(m)$	m
(11000)	9
(10100)	8
(10010)	7
(10001)	6
(01100)	5
(01010)	4
(01001)	3
(00110)	2
(00101)	1
(00011)	0

Es gilt

$$\begin{aligned} \#\{(M_0 M_1 \ldots M_{p-1}) \in \{0,1\}^p : \sum_{i=0}^{p-1} M_i = h, M_0 = 1\} &= \binom{p-1}{h-1}, \\ \#\{(M_0 M_1 \ldots M_{p-1}) \in \{0,1\}^p : \sum_{i=0}^{p-1} M_i = h, M_0 = 0\} &= \binom{p-1}{h}. \end{aligned}$$

Da die Bijektion $T_{p,h}$ die Ordnungsrelationen respektiert, folgt

$$T_{p,h}(\{0,1,2,\ldots,\binom{p-1}{h} - 1\}) = \{(0 M_1 \ldots M_{p-1}) \in \{0,1\}^p : \sum_i M_i = h\}$$

und damit

$$T_{p,h}(m) = (1 M_1 \ldots M_{p-1}) \quad \Longleftrightarrow \quad m \ge \binom{p-1}{h}.$$

Daraus erhält man dann folgenden Algorithmus:

Algorithmus zur Bestimmung von $T_{p,h}(m)$ und $T_{p,h}^{-1}(M)$: Gegeben seien natürliche Zahlen p und h mit $1 \le h < p$.

1. Gegeben sei $m \in \mathbf{N}$ mit $0 \le m \le \binom{p}{h} - 1$. Berechnet wird $T_{p,h}(m) = (M_0 M_1 \ldots M_{p-1})$.
   (a) Setze $l := h$.
   (b) Für $i = 1, \ldots, p$ führe man folgendes aus:
   Ist $m \ge \binom{p-i}{l}$, setze $M_{i-1} := 1$, $m := m - \binom{p-i}{l}$, $l := l - 1$.
   Ansonsten setze $M_{i-1} := 0$.

2. Gegeben sei $M = (M_0 M_1 \dots M_{p-1}) \in \{0,1\}^p$ mit $\sum_{i=1}^{p} M_i = h$. Bestimmt wird $m = T_{p,h}^{-1}(M)$.
   (a) Setze $m := 0$, $l := h$.
   (b) Führe für $i = 1, \dots, p$ folgendes aus:
   Ist $M_{i-1} = 1$, setze $m := m + \binom{p-i}{l}$, $l := l - 1$.

Bemerkung: Um Beispiele zu rechnen, haben wir für die Abbildungen $T_{p,h}$ und $T_{p,h}^{-1}$ Maple-Funktionen 'T_ph' und 'T_ph_i' geschrieben.

3. Algebraische Vorbereitungen

Sei p eine Primzahl und $h \geq 1$. Ist $f(x) \in \mathbf{F}_p[x]$ ein irreduzibles normiertes Polynom vom Grad h, so wird die Menge

$$F(p,h,f(x)) = \{a(x) \in \mathbf{F}_p[x] : \deg a(x) \leq h-1\}$$

durch die Operationen

$$(a(x), b(x)) \mapsto a(x) + b(x), \quad (a(x), b(x)) \mapsto a(x)b(x) \bmod f(x)$$

zu einem Körper mit p^h Elementen. Bis auf Isomorphie gibt es nur einen Körper mit p^h Elementen, der auch mit $\mathbf{F}_{p^h}$ bezeichnet wird.
Man kann schnell testen, ob ein Polynom $f(x) \in \mathbf{F}_p[x]$ irreduzibel ist. (Die Maple-Funktion 'Irreduc(f) mod p' führt einen entsprechenden Test aus.) Daher kann man sich schnell einen Körper $F(p,h,f(x)) \simeq \mathbf{F}_{p^h}$ konstruieren.

Die multiplikative Gruppe $\mathbf{F}_{p^h}^*$ des Körpers $\mathbf{F}_{p^h}$ ist zyklisch von der Ordnung $p^h - 1$, d.h. es gibt ein Polynom $g(x)$ vom Grad $\leq h-1$ mit

$$\{g(x)^n \bmod f(x) : 0 \leq n \leq p^h - 2\} = F(p,h,f(x)) \setminus \{0\}.$$

Genau dann hat ein Polynom $g(x)$ die angegebene Eigenschaft, wenn

$$g(x)^{\frac{p^h-1}{p_i}} \not\equiv 1 \bmod f(x) \text{ für alle Primteiler } p_i \text{ von } p^h - 1$$

gilt. Dies ist einfach zu testen, wenn man $p^h - 1$ faktorisieren kann. (Für das schnelle Potenzieren $g(x)^n \bmod f(x)$ gibt es die Maple-Funktion 'Powmod(g,n,f,x) mod p'.)

Ist dann $k(x) \not\equiv 0 \bmod f(x)$, so gibt es eine Zahl ℓ mit

$$g(x)^\ell \equiv k(x) \bmod f(x) \quad \text{und} \quad 0 \leq \ell \leq p^h - 2.$$

ℓ heißt der diskrete Logarithmus von $k(x)$ zur Basis $g(x)$ und wird auch mit $\log_g k$ bezeichnet.

Die Berechnung diskreter Logarithmus ist im allgemeinen sehr schwierig. (Diese Schwierigkeit nutzt man zur Konstruktion von Kryptosystemen wie Diffie-Hellman-Schlüsselaustausch oder ElGamal-Verschlüsselung.) Ein allgemeines probabilistisches Verfahren ist folgendes:

Das Pollardsche ρ-Verfahren zur Berechnung diskreter Logarithmen in $\mathbf{F}_{p^h}$: Gegeben sei

- ein irreduzibles Polynom $f(x) \in \mathbf{F}_p[x]$ vom Grad h,
- ein $g(x) \in F(p,h,f(x))$,
- die Ordnung n von $g(x)$ in der multiplikativen Gruppe $F(p,h,f(x))^*$, d.h. die kleinste natürliche Zahl n mit $g(x)^n \equiv 1 \bmod f(x)$,
- ein Polynom $k(x) \in F(p,h,f(x)) \setminus \{0\}$ (mit $k(x)^n \equiv 1 \bmod f(x)$).

Bestimmt wird eine Zahl $\ell \in \mathbf{N}_0$ mit $g(x)^\ell \equiv k(x) \bmod f(x)$ oder das Verfahren wird ohne Erfolg beendet.

1. Definiere rekursiv $x_i \in F(p,h,f(x))$, $a_i, b_i \in \mathbf{N}_0$ mit

$$x_i \equiv g(x)^{a_i} k(x)^{b_i} \mod f(x)$$

durch $x_0 = 1$, $a_0 = 0$, $b_0 = 0$ und

$$x_{i+1} \equiv \begin{cases} x_i^2 \bmod f(x), & \text{falls } x_i(0) \equiv 0 \bmod 3, \\ k(x) \cdot x_i \bmod f(x), & \text{falls } x_i(0) \equiv 1 \bmod 3, \\ g(x) \cdot x_i \bmod f(x), & \text{falls } x_i(0) \equiv 2 \bmod 3. \end{cases}$$

Dann wird

$$a_{i+1} \equiv \begin{cases} 2a_i \bmod n, \\ a_i \bmod n, \\ a_i + 1 \bmod n, \end{cases} \qquad b_{i+1} \equiv \begin{cases} 2b_i \bmod n, \\ b_i + 1 \bmod n, \\ b_i \bmod n, \end{cases} \quad \text{falls} \quad x_i(0) \equiv \begin{cases} 0 \bmod 3, \\ 1 \bmod 3, \\ 2 \bmod 3. \end{cases}$$

Dabei soll $x_i(0) \bmod 3$ bedeuten, daß der konstante Term $x_i(0) \in \mathbf{F}_p$ als ganze Zahl zwischen 0 und $p-1$ aufgefaßt und dann modulo 3 reduziert wird. Ist $g(x)^\ell \equiv k(x) \bmod f(x)$, so ist $x_i \equiv g(x)^{a_i + \ell b_i} \bmod f(x)$.

2. Mit obigen Rekursionsformeln berechnet man parallel (x_i, a_i, b_i) und (x_{2i}, a_{2i}, b_{2i}) und testet für jedes $i \geq 1$, ob $x_i = x_{2i}$ gilt.
3. Gilt $x_i = x_{2i}$ und ist $k(x) \equiv g(x)^\ell \bmod f(x)$, so folgt $a_i + \ell b_i = a_{2i} + \ell b_{2i} \bmod n$, also

$$\ell(b_i - b_{2i}) \equiv a_{2i} - a_i \bmod n.$$

Ist $\mathrm{ggT}(b_i - b_{2i}, n) = 1$, so folgt

$$\ell \equiv \frac{a_{2i} - a_i}{b_i - b_{2i}} \bmod n$$

und man ist fertig. Im Fall $\mathrm{ggT}(b_i - b_{2i}, n) > 1$ beenden wir das Verfahren erfolglos.

Bemerkungen:

1. Man hofft, daß sich die Folge x_i wie eine Zufallsfolge verhält. Dann ist es wahrscheinlich, daß nach etwa $\sqrt{n}$ Schritten der Fall $x_i = x_{2i}$ eintritt.
2. Durch die Betrachtung von $x_i(0) \bmod 3$ wird $F(p, h, f(x))$ in drei Teilmengen aufgeteilt, man kann aber auch eine andere Aufteilung wählen.
3. Ist die Ordnung n von $g(x)$ in $F(p, h, f(x))^*$ eine Primzahl, so folgt aus $x_i = x_{2i}$ und $\mathrm{ggT}(b_i - b_{2i}, n) > 1$ sofort $(x_i, a_i, b_i) = (x_{2i}, a_{2i}, b_{2i})$, woraus keine sinnvolle Information für den diskreten Logarithmus folgt. Da in unseren Anwendungen n meist eine Primzahl sein wird, beenden wir im obigen Fall das Verfahren ergebnislos (ohne weitere Überlegungen).

Beispiel: Wir wählen $p = 101$, $h = 3$, $f(x) = x^3 + 21x^2 + 58x + 99$. Es ist $p^h - 1 = 2^2 \cdot 5^2 \cdot 10303$. Das Polynom $g(x) = 10x^2 + 42x + 76$ hat Ordnung $n = 10303$ in der multiplikativen Gruppe $F(p, h, f(x))^*$. Für $k(x) = 43x^2 + 6x + 11$ gilt auch $k(x)^n \equiv 1 \bmod f(x)$. Mit den Bezeichnungen des ρ-Verfahrens erhalten wir:

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
1	$43x^2 + 6x + 11$	0	1	$56x^2 + 59x + 42$	1	1
2	$56x^2 + 59x + 42$	1	1	$100x^2 + 42x + 27$	4	4
3	$6x^2 + 78x + 24$	2	2	$52x^2 + 39x + 78$	9	8
4	$100x^2 + 42x + 27$	4	4	$71x^2 + 43x + 77$	36	32
5	$19x^2 + 78x + 14$	8	8	$71x^2 + 99x + 44$	38	32
6	$52x^2 + 39x + 78$	9	8	$37x^2 + 56x + 65$	39	33
7	$84x^2 + 23x + 12$	18	16	$41x^2 + 82x + 19$	41	33
8	$71x^2 + 43x + 77$	36	32	$83x^2 + 84x + 94$	41	35
9	$89x^2 + 25x + 26$	37	32	$19x^2 + 78x + 14$	41	37
10	$71x^2 + 99x + 44$	38	32	$84x^2 + 23x + 12$	84	74
11	$47x^2 + 8x + 52$	39	32	$89x^2 + 25x + 26$	169	148
12	$37x^2 + 56x + 65$	39	33	$47x^2 + 8x + 52$	171	148
13	$9x^2 + 50x + 92$	40	33	$9x^2 + 50x + 92$	172	149

Es folgt

$$\log_{g(x)} k(x) \equiv \frac{a_{26} - a_{13}}{b_{13} - b_{26}} \equiv \frac{172 - 40}{33 - 149} \equiv 5328 \bmod 10303.$$

Hat $p^h - 1$ nur (verhältnismäßig) kleine Primteiler, so kann man folgendes Verfahren anwenden:

Silver-Pohlig-Hellman-Verfahren zur Berechnung diskreter Logarithmen in $\mathbf{F}_{p^h}$: Gegeben sei

- ein irreduzibles Polynom $f(x) \in \mathbf{F}_p[x]$ vom Grad h,

- die Primfaktorzerlegung
$$p^h - 1 = p_1^{e_1} \dots p_r^{e_r},$$
- ein $g(x) \in F(p,h,f(x))$, das die multiplikative Gruppe $F(p,h,f(x))^*$ erzeugt, d.h. die Bedingungen
$$g(x)^{\frac{p^h-1}{p_i}} \not\equiv 1 \bmod f(x) \text{ für } i = 1, \dots, r$$
erfüllt,
- ein Polynom $k(x) \in F(p,h,f(x)) \setminus \{0\}$.

Bestimmt wird eine Zahl $\ell \in \mathbf{N}_0$ mit $g(x)^\ell \equiv k(x) \bmod f(x)$, also $\ell = \log_g k$.

1. Für $i = 1, \dots, r$ bestimmt man eine Zahl ℓ_i mit
$$\left(g(x)^{\frac{p^h-1}{p_i^{e_i}}}\right)^{\ell_i} \equiv k(x)^{\frac{p^h-1}{p_i^{e_i}}} \bmod f(x) \quad \text{und} \quad 0 \le \ell_i \le p_i^{e_i} - 1,$$
indem man z.B. nacheinander probiert, welches $\ell_i = 0, 1, 2, \dots$ die Gleichung erfüllt, oder indem man das Pollardsche ρ-Verfahren anwendet. ($g(x)^{\frac{p^h-1}{p_i^{e_i}}}$ hat Ordnung $p_i^{e_i}$ in $F(p,h,f(x))^*$.)
2. Mit dem chinesischen Restsatz konstruiert man sich eine Zahl ℓ mit
$$\ell \equiv \ell_1 \bmod p_1^{e_1}, \quad \ell \equiv \ell_2 \bmod p_2^{e_2}, \quad \dots, \quad \ell \equiv \ell_r \bmod p_r^{e_r}.$$
Dann gilt
$$g(x)^\ell \equiv k(x) \bmod f(x),$$
d.h. ℓ ist der diskrete Logarithmus von k zur Basis g.

Bemerkung: Um das Silver-Pohlig-Hellman-Verfahren zur Logarithmenberechnung anwenden zu können, muß man zunächst $p^h - 1$ faktorisieren:
$$p^h - 1 = p_1^{e_1} \dots p_r^{e_r},$$
was nicht immer einfach ist. Der zeitaufwendige Schritt ist dann die Suche nach den richtigen ℓ_i's. Die Schrittzahl wächst bei naiver Bestimmung der ℓ_i's im allgemeinen mindestens wie die Summe
$$p_1^{e_1} + \dots + p_r^{e_r},$$
bei Verwendung des Pollardschen ρ-Verfahrens erwartet man eine Schrittzahl, die mindestens wie
$$\sqrt{p_1^{e_1}} + \dots + \sqrt{p_r^{e_r}}$$
wächst. Ist einer der Primteiler p_i von $p^h - 1$ zu groß, wird das Silver-Pohlig-Hellman-Verfahren unpraktikabel.

Beispiel: Wir wählen $p = 19$, $h = 5$ und das modulo 19 irreduzible Polynom
$$f(x) = x^5 + 18x^4 + 4x^2 + 15x + 9.$$
Es ist
$$p^h - 1 = 2 \cdot 3^2 \cdot 151 \cdot 911$$
und
$$x^{\frac{p^h-1}{2}} \equiv 18 \bmod f(x), \quad x^{\frac{p^h-1}{3}} \equiv 11 \bmod f(x), \quad x^{\frac{p^h-1}{151}} \equiv 9x^3 + 6x^2 + x + 11 \bmod f(x),$$
$$x^{\frac{p^h-1}{911}} \equiv 9x^2 + 6x + x + 7 \bmod f(x), \quad \left(x^{\frac{p^h-1}{9}} \equiv 5 \bmod f(x)\right).$$
Daher erzeugt x die multiplikative Gruppe von $F(p,h,f(x))$. Wir wollen den diskreten Logarithmus $\log_x(x+7)$ berechnen. Es gilt
$$(x+7)^{\frac{p^h-1}{2}} \equiv 18 \bmod f(x), \quad (x+7)^{\frac{p^h-1}{9}} \equiv 17 \bmod f(x), \quad (x+7)^{\frac{p^h-1}{151}} \equiv 12x^3+16x^2+14x+13 \bmod f(x),$$
$$(x+7)^{\frac{p^h-1}{911}} \equiv 10x^4 + 15x^3 + 3x^2 + 13 \bmod f(x).$$
Durch Probieren findet man
$$\ell_1 = 1, \quad \ell_2 = 4, \quad \ell_3 = 59, \quad \ell_4 = 57 \quad \text{mit} \quad x^{\frac{p^h-1}{p_i^{e_i}}\ell_i} \equiv (x+7)^{\frac{p^h-1}{p_i^{e_i}}} \bmod f(x).$$

Der chinesische Restsatz liefert dann $\ell = 1045885$ mit $\ell \equiv \ell_i \bmod p_i^{e_i}$, also

$$x^{1045885} \equiv x + 7 \bmod f(x).$$

Bemerkung: Zur Berechnung von diskreten Logarithmus mit dem Silver-Pohlig-Hellman-Verfahren haben wir die Maple-Funktion 'dislog(k,g,f,p)' geschrieben. Bei Bedarf werden die Funktionen 'dislog_pollard_ntl(k,g,f,p)' oder 'dislog_naiv_ntl(k,g,f,p)' aufgerufen, die die NTL-Programme 'dlog_pollard_ntl.c' oder 'dlog_naiv_ntl.c' benutzen.

4. Chor-Rivest-Verschlüsselung

Nach den kombinatorischen und algebraischen Vorbereitungen können wir jetzt das Chor-Rivest-Verschlüsselungsverfahren vorstellen.

Das Chor-Rivest-Verschlüsselungsverfahren:

1. **Schlüsselerzeugung:** Jeder Teilnehmer A wählt bzw. bestimmt sich folgende Größen:
   (a) Eine Primzahl p und eine natürliche Zahl $h < p$.
   (b) Ein zufälliges normiertes irreduzibles Polynom $f(x) \in \mathbf{F}_p[x]$ vom Grad h.
   (c) Ein zufälliges Polynom $g(x) \in \mathbf{F}_p[x]$, das Ordnung $p^h - 1$ modulo $f(x)$ hat.
   (d) Für jedes $i \in \mathbf{F}_p$ eine ganze Zahl a_i mit $0 \le a_i \le p^h - 2$ und
   $$g(x)^{a_i} \equiv x + i \bmod f(x),$$
   also $a_i = \log_g(x + i)$.
   (e) Eine zufällige Permutation π der Menge $\{0, 1, \ldots, p-1\}$.
   (f) Eine zufällige ganze Zahl d mit $0 \le d \le p^h - 2$.
   (g) Ganze Zahlen c_i mit
   $$c_i \equiv a_{\pi(i)} + d \bmod p^h - 1 \quad \text{für} \quad 0 \le i \le p-1.$$
   A's öffentlicher Schlüssel ist $((c_0, c_1, \ldots, c_{p-1}), p, h)$, der private Schlüssel $(f(x), g(x), \pi, d)$.
2. **Verschlüsselung:** Will B eine Nachricht verschlüsselt an A schicken, geht B folgendermaßen vor:
   (a) B besorgt sich den öffentlichen Schlüssel $((c_0, \ldots, c_{p-1}, p, h)$ von A.
   (b) B wandelt die Nachricht in eine Folge von 0-1-Vektoren $M = (M_0, \ldots, M_{p-1})$ der Länge p mit Hamming-Gewicht h nach einem festgelegten Verfahren um.
   (c) $c \equiv \sum_{i=0}^{p-1} M_i c_i \pmod{p^h - 1}$ ist die Verschlüsselung von $(M_0, \ldots, M_{p-1})$ und wird an A geschickt.
3. **Entschlüsselung:** A empfängt von B die verschlüsselte Nachricht c und bestimmt nacheinander folgende Größen:
   (a) $r \equiv c - hd \bmod p^h - 1$.
   (b) $u(x) \equiv g(x)^r \bmod f(x)$.
   (c) $s(x) = u(x) + f(x)$. ($s(x) \in \mathbf{F}_p[x]$ ist normiert und hat Grad h.)
   (d) Die Ausgangsbitfolge $M = (M_0, M_1, \ldots, M_{p-1})$ ergibt sich nun aus
   $$M_i = \begin{cases} 1 & \text{für } s(-\pi(i)) = 0, \\ 0 & \text{für } s(-\pi(i)) \neq 0. \end{cases}$$

Beweis für die Richtigkeit der Entschlüsselung: Modulo $p^h - 1$ gilt:

$$c \equiv \sum_{i=0}^{p-1} M_i c_i \equiv \sum_{i=0}^{p-1} M_i (a_{\pi(i)} + d) = \sum_{i=0}^{p-1} M_i a_{\pi(i)} + d \sum_{i=0}^{p-1} M_i = \sum_{i=0}^{p-1} M_i a_{\pi(i)} + hd \pmod{p^h - 1}.$$

Wegen $g(x)^{p^h - 1} \equiv 1 \bmod f(x)$ folgt modulo $f(x)$

$$\begin{aligned} s(x) &= u(x) + f(x) \equiv u(x) \equiv g(x)^r \equiv g(x)^{c - hd} \equiv \\ &\equiv g(x)^{\sum_{j=0}^{p-1} M_j a_{\pi(j)}} = \prod_{\substack{0 \le j \le p-1 \\ M_j = 1}} g(x)^{a_{\pi(j)}} \equiv \prod_{\substack{0 \le j \le p-1 \\ M_j = 1}} (x + \pi(j)) \bmod f(x). \end{aligned}$$

Sowohl die linke als auch die rechte Seite sind normierte Polynome vom Grad h, also folgt

$$s(x) = \prod_{\substack{0 \le j \le p-1 \\ M_j = 1}} (x + \pi(j)).$$

Somit hat man für $i \in \mathbf{F}_p$

$$s(-\pi(i)) = 0 \quad \Longleftrightarrow \quad \prod_{\substack{0 \le j \le p-1 \\ M_j = 1}} (\pi(j) - \pi(i)) = 0 \quad \Longleftrightarrow \quad M_i = 1,$$

was die Behauptung beweist. ■

Beispiel: Wir wählen $p = 11$, $h = 4$, dann (zufällig)

$$f = x^4 + 6x^3 + 10x^2 + 3x + 8, \quad g = 6x + 10.$$

Die diskreten Logarithmen a_i mit $g^{a_i} \equiv x + i \bmod f$ sind

$$a_0 = 9647, a_1 = 3525, a_2 = 9285, a_3 = 3407, a_4 = 3631, a_5 = 5977, a_6 = 13334, a_7 = 6161,$$

$$a_8 = 10334, a_9 = 13177, a_10 = 7471.$$

Als Permutation wählen wir

$$\pi(0) = 1, \quad \pi(1) = 10, \quad \pi(2) = 4, \quad \pi(3) = 8, \quad \pi(4) = 2, \quad \pi(5) = 6,$$

$$\pi(6) = 5, \quad \pi(7) = 0, \quad \pi(8) = 3, \quad \pi(9) = 9, \quad \pi(10) = 7$$

und weiter die Zahl $d = 4773$. Damit ergibt sich der öffentliche Schlüssel

$$(c, p, h) = ((8298, 12244, 8404, 467, 14058, 3467, 10750, 14420, 8180, 3310, 10934), 11, 4)$$

sowie der private Schlüssel

$$(f, g, \pi, d) = (x^4 + 6x^3 + 10x^2 + 3x + 8, \quad 6x + 10, \quad [1, 10, 4, 8, 2, 6, 5, 0, 3, 9, 7], \quad 4773).$$

Verschlüsselt man $M = (00000110101)$ mit dem öffentlichen Schlüssel, erhält man die natürliche Zahl 4051.
Beim Entschlüsseln von 4051 mit dem privaten Schlüssel ergeben sich folgende Werte:

$$r = 14239, \quad u = 4x^3 + 8x^2 + 6, \quad s = 10x^3 + 7x^2 + 3 + x^4 + 3x.$$

Durch Probieren findet man, daß $s(-\pi(i)) \equiv 0 \bmod p$ für $i = 5, 6, 8, 10$ gilt. Also erhält man schließlich $M = (00000110101)$ zurück.

Bemerkung: Für Schlüsselerzeugung, Verschlüsselung und Entschlüsselung haben wir Maple-Funktionen 'choriv_key', 'choriv_en0' und 'choriv_de0' geschrieben und damit die nachfolgenden Beispiele gerechnet.

5. Chor-Rivest-Schlüssel mit $h = 12$ und $h = 24$

Chor und Rivest schlagen als geeignete Schlüsselparameter unter anderem $p = 197$, $h = 12$ vor. Sie geben an, einen zugehörigen Schlüssel nach einigen Vorbereitungen in ungefähr 8 Stunden berechnet zu haben.

Bei unserer Maple-Schlüsselerzeugungsfunktion 'choriv_key' wird die meiste Rechenzeit für die Berechnung der p Logarithmen $\log_{g(x)}(x + i)$, $i = 0, \ldots, p-1$ benötigt. Zugrunde liegt jeweils das Silver-Pohlig-Hellman-Verfahren. Für die hierzu nötige Berechnung der Teillogarithmen ℓ_i mit $g(x)^{\frac{p^h-1}{p_i^{e_i}}} \equiv (x + i)^{\frac{p^h-1}{p_i^{e_i}}} \bmod f(x)$ haben wir drei verschiedene Wege ausprobiert:

- naive Berechnung mit Maple,
- naive Berechnung mit NTL,
- das Pollardsche ρ-Verfahren mit NTL.

Bei der naiven Logarithmenberechnung braucht man für einen Logarithmus ungefähr $\sum_i p_i^{e_i}$ Schritte, für alle Logarithmen wächst dann die Laufzeit mindestens wie

$$p \cdot \sum_{i=1}^{r} p_i^{e_i},$$

wenn $p^h - 1 = p_1^{e_1} \ldots p_r^{e_r}$ gilt.

Wir haben zunächst nur mit Maple ohne Zuhilfenahme von NTL zufällige Schlüssel $K_{p,h}$ mit dem Parameter $h = 12$ erzeugt. Die Schlüssel sind

$$K_{13,12}, K_{17,12}, K_{19,12}, K_{23,12}, K_{29,12}, K_{31,12}, K_{37,12}, K_{41,12}, K_{43,12}, K_{47,12}, K_{53,12}, K_{103,12}, K_{197,12}.$$

und sind explizit im Anhang angegeben. Die Rechenzeiten stehen in direktem Bezug zu $\log_2(p \cdot \sum_i p_i^{e_i})$ mit $p^{12} - 1 = \prod_i p_i^{e_i}$, wie man aus folgender Tabelle ersieht.

p	h	$\log_2(p \sum p_i^{e_i})$	$K_{p,h}$	Rechenzeit
13	12	18.51	$K_{13,12}$	00:03:20
17	12	20.44	$K_{17,12}$	00:10:11
19	12	14.92	$K_{19,12}$	00:00:15
23	12	17.48	$K_{23,12}$	00:01:32
29	12	15.11	$K_{29,12}$	00:00:19
31	12	24.77	$K_{31,12}$	04:02:03
37	12	22.35	$K_{37,12}$	00:45:03
41	12	17.72	$K_{41,12}$	00:01:55
43	12	27.13	$K_{43,12}$	24:26:19
47	12	18.00	$K_{47,12}$	00:02:14
53	12	18.98	$K_{53,12}$	00:04:30
103	12	21.83	$K_{103,12}$	00:30:22
197	12	23.10	$K_{197,12}$	01:25:29

Als nächstes haben wir für $h = 24$ die Teillogarithmen mit dem NTL-Programm 'dlog_naiv_ntl.c' naiv berechnet. Die Rechenzeit hängt wesentlich von $\log_2(p \cdot \sum_i p_i^{e_i})$ ab, wenn $p^{24} - 1 = \prod_i p_i^{e_i}$ ist. In der folgenden Tabelle finden sich die ersten 10 Primzahlen mit $24 < p < 600$ mit den kleinsten Werten $\log_2(p \cdot \sum_i p_i^{e_i})$ und den Rechenzeiten für die so berechneten Schlüssel $K_{43,24}$, $K_{79,24}$, $K_{139,24}$.

p	h	$\log_2(p \cdot \sum_i p_i^{e_i})$	$K_{p,h}$	Rechenzeit
139	24	25.83	$K_{139,24}$	05:01:51
79	24	28.14	$K_{79,24}$	22:16:04
43	24	28.48	$K_{43,24}$	30:55:39
47	24	28.82		
41	24	29.53		
181	24	29.76		
29	24	30.60		
277	24	31.27		
197	24	31.55		
149	24	32.43		

Schließlich haben wir für $h = 24$ die Teillogarithmen beim Silver-Pohlig-Hellman-Verfahren mit der Pollardschen ρ-Methode berechnet unter Verwendung des NTL-Programms 'dlog_pollard_ntl.c'. Die erwartete Schrittzahl wächst mindestens wie

$$p(\sqrt{p_1^{e_1}} + \cdots + \sqrt{p_r^{e_r}}),$$

wenn $p^{24} - 1 = p_1^{e_1} \ldots p_r^{e_r}$ gilt. Wir haben daher in folgender Tabelle die ersten Primzahlen p zwischen 24 und 1000 aufgelistet, und zwar nach der Größe von $\log_2(p(\sqrt{p_1^{e_1}} + \cdots + \sqrt{p_r^{e_r}})$. Die Tabelle enthält auch die Rechenzeiten für die berechneten Schlüssel $K_{p,h}$.

p	h	$\log_2(p \cdot \sum_i \sqrt{p_i^{e_i}})$	$K_{p,h}$	Rechenzeit
43	24	17.49	$K_{43,24}$	00:08:05
79	24	17.68		
41	24	17.71		
139	24	17.73		
47	24	17.78		
29	24	17.87		
181	24	19.33		
149	24	20.24		
197	24	20.47	$K_{197,24}$	01:08:01
89	24	20.49		
277	24	20.69	$K_{277,24}$	01:16:50
113	24	20.90		
157	24	21.37		
103	24	21.59		
421	24	21.79		
509	24	21.87		
109	24	22.00		
211	24	22.01	$K_{211,24}$	03:10:13

($K_{43,24}$ wurde bereits vorher erzeugt, allerdings mit wesentlich höherer Rechenzeit.)

6. Ein Angriff auf das Chor-Rivest-Kryptosystem mit verbesserter Gitterreduktion

Ist $((c_0, \ldots, c_{p-1}), p, h)$ ein öffentlicher Schlüssel eines Chor-Rivest-Kryptosystems, so ist $\max(c_0, \ldots, c_{p-1}) \approx p^h$, also gilt für die Dichte des Rucksacks $(c_0, \ldots, c_{p-1})$:

$$d(c_0, \ldots, c_{p-1}) = \frac{p}{\log_2 \max(c_0, \ldots, c_{p-1})} \approx \frac{p}{\log_2 p^h} = \frac{p}{h \log_2 p}.$$

Wir erhalten beispielsweise

p	53	103	103	197	197
h	12	12	24	12	24
d	0.77	1.28	0.64	2.15	1.08

Durch geeignete Wahl von p und h kann man erreichen, daß die früher vorgestellten Angriffe auf Rucksäcke mit kleiner Dichte (low density attacks) erfolglos bleiben.

Schnorr und Hörner haben eine Modifizierung des LLL-Algorithmus angegeben und damit dann unter anderem Angriffe auf das Chor-Rivest-System mit den Parametern $p = 103$, $h = 12$ durchgeführt. Das Verfahren war in 42% der Fälle erfolgreich, wobei 50 Versuche durchgeführt wurden. (Jeder einzelne Versuch benötigte ungefähr 1.5 Stunden Rechenzeit.)

Da der nachfolgende algebraische Angriff allerdings wesentlich effektiver ist, verzichten wir hier auf eine Darstellung des Schnorr-Hörner-Angriffs.

7. Ein algebraischer Angriff

Vaudenay hat einen Angriff auf das Chor-Rivest-Kryptosystem beschrieben, wobei aus dem öffentlichen Schlüssel ein privater konstruiert wird. Nach eigenen Aussagen funktioniert das z.B. für $h = 24$ in ungefähr 15 Minuten.

Wir wollen im folgenden beispielhaft die Fälle $h = 12$ und $h = 24$ behandeln.

7.1. Algebraische Vorbereitungen. Ein Chor-Rivest-Kryptosystem besitzt einen öffentlichen Schlüssel $((c_1,\dots,c_{p-1}),p,h)$ und einen privaten Schlüssel $(f(x),g(x),\pi^{-1},d)$.

Das Polynom $f(x)\in\mathbf{F}_p[x]$ dient der Beschreibung des endlichen Körpers $\mathbf{F}_{p^h}$ mittels des Ringhomomorphismus

$$\psi:\mathbf{F}_p[x]\to\mathbf{F}_{p^h}$$

mit Kern $f(x)\mathbf{F}_p[x]$. Ist

$$\psi(x)=\xi,$$

so ist $f(x)$ das Minimalpolynom von ξ. Außerdem gilt (nach Voraussetzung) $\mathbf{F}_{p^h}=\mathbf{F}_p[\xi]$. Sei

$$\psi(g)=\gamma.$$

Dann ist $\gamma=g(\xi)$. Das Element γ hat (nach Voraussetzung) Ordnung p^h-1 in der multiplikativen Gruppe $\mathbf{F}_{p^h}^*$. Dann wurden bei der Schlüsselerzeugung diskrete Logarithmen

$$a_i\equiv\log_\gamma(\xi+i)\pmod{p^h-1},\qquad\text{also}\qquad\gamma^{a_i}=\xi+i\qquad\text{für}\qquad i\in\mathbf{F}_p,$$

berechnet. Nach Wahl einer Permutation π (von $\mathbf{F}_p$) und einer Zahl $d\in\{0,1,\dots,p^h-2\}$ erhielt man den öffentlichen Schlüssel aus der Gleichung

$$c_i\equiv a_{\pi(i)}+d\pmod{p^h-1},$$

d.h.

$$\gamma^{c_i-d}=\xi+\pi(i).$$

Die wesentliche Aufgabe besteht nun darin, bei Kenntnis eines öffentlichen Schlüssels $((c_0,\dots,c_{p-1}),p,h)$ Größen $\gamma,\xi\in\mathbf{F}_{p^h}$, π, d zu bestimmen mit

$$\gamma^{c_i-d}=\xi+\pi(i),$$

wobei ξ Grad h haben sollte, denn dann folgt die Entschlüsselung einer Nachricht $\sum_i M_ic_i$ ($\sum_i M_i=h$) mittels der Formel

$$\gamma^{(\sum_i M_ic_i)-hd}=\gamma^{-hd}\prod_{\substack{0\le i\le p-1\\ M_i=1}}\gamma^{c_i}=\prod_{\substack{0\le i\le p-1\\ M_i=1}}\gamma^{c_i-d}=\prod_{\substack{0\le i\le p-1\\ M_i=1}}(\xi+\pi(i)).$$

Wir werden im folgenden auch die Größen γ, ξ, π, d als privaten Chor-Rivest-Schlüssel bezeichnen.

Wir wollen jetzt sehen, daß man die Permutation π normalisieren kann.

LEMMA. *γ,ξ,π,d seien ein privater Schlüssel zu $((c_0,\dots,c_{p-1}),p,h)$. Definiert man*

$$\widetilde{\xi}=\frac{\xi+\pi(0)}{\pi(1)-\pi(0)},\quad\widetilde{\pi}(i)=\frac{\pi(i)-\pi(0)}{\pi(1)-\pi(0)},\quad\widetilde{d}\equiv d+\log_\gamma(\pi(1)-\pi(0)),$$

so ist auch $\gamma,\widetilde{\xi},\widetilde{\pi},\widetilde{d}$ ein privater Chor-Rivest-Schlüssel und $((c_0,\dots,c_{p-1}),p,h)$ ist der zugehörige öffentliche Schlüssel. Dabei gilt

$$\widetilde{\pi}(0)=0,\quad\widetilde{\pi}(1)=1.$$

Beweis: $\widetilde{\xi}$ hat genauso wie ξ Grad h über $\mathbf{F}_p$. Da π eine Permutation von $\mathbf{F}_p$ ist, gilt dies auch für $\widetilde{\pi}$. Außerdem sieht man $\widetilde{\pi}(0)=0$, $\widetilde{\pi}(1)=1$. Es gilt

$$\begin{aligned}\gamma^{c_i-\widetilde{d}}&=\gamma^{c_i-d-\log_\gamma(\pi(1)-\pi(0))}=\gamma^{c_i-d}\cdot\frac{1}{\pi(1)-\pi(0)}=\\&=\frac{\xi+\pi(i)}{\pi(1)-\pi(0)}=\frac{\xi+\pi(0)+\pi(i)-\pi(0)}{\pi(1)-\pi(0)}=\widetilde{\xi}+\widetilde{\pi}(i),\end{aligned}$$

was die Behauptung beweist. ■

Man kann also die Permutation immer so wählen, daß $\pi(0)=0$, $\pi(1)=1$ gilt.

LEMMA. *Sei γ, ξ, π, d ein privater Chor-Rivest-Schlüssel und $((c_0, \ldots, c_{p-1}), p, h)$ der zugehörige öffentliche Schlüssel mit $\pi(0) = 0$, $\pi(1) = 1$. Dann gilt:*

$$\xi = \frac{1}{\gamma^{c_1 - c_0} - 1}, \quad d \equiv c_0 - \log_\gamma \xi \pmod{p^h - 1}, \quad \pi(i) = \gamma^{c_i - d} - \xi \text{ für alle } i.$$

(Also kann man (γ, ξ, π, d) bestimmen, wenn man $((c_0, \ldots, c_{p-1}), p, h)$ und γ kennt.)

Beweis: Es gelten die Formeln

$$\gamma^{c_0 - d} = \xi, \quad \gamma^{c_1 - d} = \xi + 1,$$

was durch Division

$$\gamma^{c_1 - c_0} = 1 + \frac{1}{\xi}, \quad \text{also} \quad \xi = \frac{1}{\gamma^{c_1 - c_0} - 1}$$

liefert. Weiter ergibt sich dann

$$d \equiv c_0 - \log_\gamma \xi \pmod{p^h - 1} \quad \text{und dann} \quad \pi(i) = \gamma^{c_i - d} - \xi,$$

was zu zeigen war. ■

Bemerkung: Das letzte Lemma zeigt, daß die Kenntnis von γ reicht, um den gesamten privaten Schlüssel zu rekonstruieren. Allerdings ist kein Weg bekannt, um direkt an γ zu kommen, wenn p^h hinreichend groß ist.

Bemerkung: Der Frobenius-Automorphismus $x \mapsto x^p$ führt die Gleichung

$$\gamma^{c_i - d} = \xi + \pi(i)$$

in die Gleichung

$$(\gamma^p)^{c_i - d} = \xi^p + \pi(i)$$

über, da $\pi(i)$ als Element von $\mathbf{F}_p$ fest bleibt. Also kann man statt des Schlüssels (γ, ξ, π, d) auch die konjugierten Schlüssel $(\gamma^p, \xi^p, \pi, d)$, $(\gamma^{p^2}, \xi^{p^2}, \pi, d)$, $\ldots$, $(\gamma^{p^{h-1}}, \xi^{p^{h-1}}, \pi, d)$ benutzen. Die Frage ist, ob γ modulo Konjugation eindeutig bestimmt ist.

Bevor wir daran gehen, aus dem öffentlichen Schlüssel den privaten zu rekonstruieren, erinnern wir noch an einige Tatsachen über endliche Körper.

1. Zu jedem Teiler r von h gibt es genau einen Zwischenkörper

   $$\mathbf{F}_p \subseteq \mathbf{F}_{p^r} \subseteq \mathbf{F}_{p^h}.$$

   Dabei hat $\mathbf{F}_{p^r}$ Grad r über $\mathbf{F}_p$, der Körper $\mathbf{F}_{p^h}$ hat Grad $\frac{h}{r}$ über $\mathbf{F}_{p^r}$.
2. Sei r ein Teiler von h. Die einzigen Körperautomorphismen von $\mathbf{F}_{p^h}$, die $\mathbf{F}_{p^r}$ elementweise fest lassen, sind Potenzen des Frobeniusautomorphismus

   $$x \mapsto x^{p^r},$$

   genauer kann man sagen

   $$\begin{aligned} \mathrm{Aut}(\mathbf{F}_{p^h} | \mathbf{F}_{p^r}) &= \{\alpha \mapsto \alpha, \alpha \mapsto \alpha^{p^r}, \alpha \mapsto \alpha^{p^{2r}}, \alpha \mapsto \alpha^{p^{3r}}, \alpha \mapsto \alpha^{p^{4r}}, \ldots\} = \\ &= \{\alpha \mapsto \alpha^{p^{ir}} : i = 0, 1, \ldots, \frac{h}{r} - 1\}. \end{aligned}$$

3. Die Normabbildung läßt sich daher ganz einfach berechnen:

   $$\begin{aligned} \mathrm{N}_{\mathbf{F}_{p^h} | \mathbf{F}_{p^r}}(\alpha) &= \alpha \cdot \alpha^{p^r} \cdot \alpha^{p^{2r}} \cdot \cdots \cdot \alpha^{p^{(\frac{h}{r}-1)r}} = \\ &= \alpha^{\sum_{i=0}^{\frac{h}{r}-1} (p^r)^i} = \alpha^{\frac{p^h - 1}{p^r - 1}}. \end{aligned}$$

7.2. Die Hilfsgrößen $\gamma_r \in \mathbf{F}_{p^r}$ und $Q_r(x) \in \mathbf{F}_{p^r}[x]$ für Teiler $r|h$. Wir definieren jetzt für jeden Teiler $r|h$

$$\gamma_r = \mathrm{N}_{\mathbf{F}_{p^h}|\mathbf{F}_{p^r}}(\gamma) = \gamma^{\frac{p^h-1}{p^r-1}}.$$

Folgende Eigenschaften sind sofort klar:

1. $\gamma_r \in \mathbf{F}_{p^r}$.
2. γ_r erzeugt die multiplikative Gruppe $\mathbf{F}^*_{p^r}$, d.h. hat Ordnung $p^r - 1$.
3. Sind r', r zwei Teiler von h mit $r'|r$, so folgt

$$\gamma_{r'} = \gamma_r^{\frac{p^r-1}{p^{r'}-1}} = \mathrm{N}_{\mathbf{F}_{p^r}|\mathbf{F}_{p^{r'}}}(\gamma_r).$$

Für jeden Teiler r von h definieren wir das Polynom

$$Q_r(x) = \gamma_r^d \prod_{i=0}^{h/r-1} (x + \xi^{p^{ri}}) = \gamma_r^d (x+\xi)(x+\xi^{p^r})(x+\xi^{p^{2r}}) \ldots (x+\xi^{p^{h-r}}).$$

LEMMA. *$Q_r(x)$ ist ein Polynom vom Grad $\frac{h}{r}$ und hat Koeffizienten in $\mathbf{F}_{p^r}$. Es gilt*

$$Q_r(\pi(i)) = \gamma_r^{c_i}.$$

Beweis: $\mathrm{grad} Q_r(x) = \frac{h}{r}$ ist klar. Durch Anwendung des Frobeniusautomorphismus $\alpha \mapsto \alpha^{p^r}$ auf die Koeffizienten von $Q_r(x)$ sieht man, daß $Q_r(x)$ Koeffizienten in $\mathbf{F}_{p^r}$ hat. Für $i \in \mathbf{F}_p$ gilt

$$\begin{aligned} Q_r(i) &= \gamma_r^d \prod_{j=0}^{\frac{h}{r}-1} (\xi^{p^{rj}} + i) = \gamma_r^d (\xi+i)(\xi^{p^r}+i)(\xi^{p^{2r}}+i)\ldots(\xi^{p^{h-r}}+i) = \\ &= \gamma_r^d \mathrm{N}_{\mathbf{F}_{p^h}|\mathbf{F}_{p^r}}(\xi+i) = \gamma_r^d \mathrm{N}_{\mathbf{F}_{p^h}|\mathbf{F}_{p^r}}(\gamma^{a_i}) = \gamma_r^d (\mathrm{N}_{\mathbf{F}_{p^h}|\mathbf{F}_{p^r}}(\gamma))^{a_i} = \gamma_r^d \gamma_r^{a_i} = \gamma_r^{a_i+d}, \end{aligned}$$

was sofort $Q_r(\pi(i)) = \gamma_r^{c_i}$ liefert. ■

Um weitere Eigenschaften von γ_r zu beweisen, brauchen wir folgendes allgemeine Lemma:

LEMMA. 1. *Für $1 \le e \le p-2$ gilt in $\mathbf{F}_p$.*

$$\sum_{i \in \mathbf{F}_p} i^e = 0.$$

(Für $e = 0$ hat man analog $\sum_{i \in \mathbf{F}_p} 1 = 0$.)

2. *Ist $F(x)$ ein Polynom vom Grad $\le p-2$ und Koeffizienten in einem Oberkörper von $\mathbf{F}_p$, so folgt*

$$\sum_{i \in \mathbf{F}_p} F(i) = 0.$$

Beweis:

1. Es gibt ein $i_0 \in \mathbf{F}_p^*$ mit $i_0^e \ne 1$, da $\mathbf{F}_p^*$ zyklisch von Ordnung $p-1$ ist. Mit i durchläuft auch $i_0 i$ die Menge $\mathbf{F}_p$. Also folgt

$$\sum_{i \in \mathbf{F}_p} i^e = \sum_{i \in \mathbf{F}_p} (i_0 i)^e = i_0^e \sum_{i \in \mathbf{F}_p} i^e$$

und damit

$$(1 - i_0^e) \sum_{i \in \mathbf{F}_p} = 0,$$

was wegen $i_0^e \ne 1$ sofort $\sum_{i \in \mathbf{F}_p} i^e = 0$ und damit die Behauptung liefert.

2. Wir schreiben $F(x) = a_0 + a_1 x + \cdots + a_{p-2}x^{p-2}$ und erhalten dann

$$\begin{aligned}\sum_{i\in\mathbf{F}_p} F(i) &= \sum_{i\in\mathbf{F}_p}(a_0 + a_1 i + a_2 i^2 + \cdots + a_{p-2}i^{p-2}) = \\ &= a_0\sum_{i\in\mathbf{F}_p} 1 + a_1\sum_{i\in\mathbf{F}_p} i + a_2\sum_{i\in\mathbf{F}_p} i^2 + \cdots + a_{p-2}\sum_{i\in\mathbf{F}_p} i^{p-2} = 0,\end{aligned}$$

wie behauptet. ■

Bemerkung: Die einschränkende Bedingung $\leq p-2$ des Lemmas ist nötig wegen

$$\sum_{i=0}^{p-1} i^{p-1} = \sum_{i=1}^{p-1} i^{p-1} \equiv \sum_{i=1}^{p-1} 1 = p-1 \equiv -1 \bmod p.$$

Wir geben nun eine wichtige Eigenschaft von γ_r an:

SATZ. *Für alle e mit $1 \leq e < \frac{(p-1)r}{h}$ gilt*

$$\sum_{i=0}^{p-1} \gamma_r^{ec_i} = 0.$$

Beweis: Für den Grad des Polynoms $Q_r(x)^e$ gilt mit der Voraussetzung $e < \frac{(p-1)r}{h}$

$$\mathrm{grad} Q_r(x)^e = \frac{h}{r}\cdot e < \frac{h}{r}\cdot\frac{(p-1)r}{h} = p-1.$$

Daher folgt mit dem letzten Lemma

$$0 = \sum_{i\in\mathbf{F}_p} Q_r(i)^e = \sum_{i\in\mathbf{F}_p} Q_r(\pi(i))^e = \sum_{i\in\mathbf{F}_p} \gamma_r^{ec_i},$$

was zu zeigen war. ■

Bemerkung: Für $\gamma = \gamma_h$ gelten also die $p-2$ Gleichungen

$$\sum_{i=0}^{p-1} \gamma^{ec_i} = 0 \quad \text{für} \quad 1 \leq e \leq p-2,$$

was es neben experimentellen Daten nahelegt, daß γ durch den öffentlichen Schlüssel bis auf Konjugation eindeutig bestimmt ist.

7.3. Praktische Bestimmung einiger γ_r's. Wir wählen ein Element $\alpha \in \mathbf{F}_{p^h}$, das Ordnung $p^h - 1$ in der multiplikativen Gruppe $\mathbf{F}_{p^h}^*$ hat. Ist r ein Teiler von h, so $\alpha^{\frac{p^h-1}{p^r-1}} \in \mathbf{F}_{p^r}$ und hat Ordnung $p^r - 1$. Da γ_r Ordnung $p^r - 1$ hat, kann man schreiben

$$\gamma_r = \alpha^{\frac{p^h-1}{p^r-1}u_r} \quad \text{mit} \quad \mathrm{ggT}(u_r, p^r-1) = 1 \quad \text{und} \quad 0 \leq u_r \leq p^r - 2.$$

Für $r'|r|h$ gilt

$$\alpha^{\frac{p^h-1}{p^{r'}-1}u_{r'}} = \gamma_{r'} = \mathrm{N}_{\mathbf{F}_{p^r}|\mathbf{F}_{p^{r'}}}(\gamma_r) = \alpha^{\frac{p^h-1}{p^{r'}-1}u_r},$$

was

$$u_r \equiv u_{r'} \pmod{p^{r'}-1}$$

liefert.

Beispiel: Für $6|h$ erhalten wir mit obigem Ansatz folgende Bedingungen für die u_i's:

$$\begin{aligned}&0 \leq u_1 \leq p-2, \quad \mathrm{ggT}(u_1, p-1) = 1,\\ &0 \leq u_2 \leq p^2-2, \quad \mathrm{ggT}(u_2, p^2-1) = 1, \quad u_2 \equiv u_1 \pmod{p-1},\\ &0 \leq u_3 \leq p^3-2, \quad \mathrm{ggT}(u_3, p^3-1) = 1, \quad u_3 \equiv u_1 \pmod{p-1},\\ &0 \leq u_6 \leq p^6-2, \quad \mathrm{ggT}(u_6, p^6-1) = 1, \quad u_6 \equiv u_2 \pmod{p^2-1}, \quad u_6 \equiv u_3 \pmod{p^3-1}.\end{aligned}$$

Für $1 \le e < \frac{(p-1)r}{h}$ gilt

$$\sum_{i=0}^{p-1} \gamma_r^{ec_i} = 0, \quad \text{also} \quad \sum_{i=0}^{p-1} \alpha^{\frac{p^h-1}{p^r-1}u_r ec_i} = 0.$$

Damit erhält man folgende Idee:

1. Teste, welche Zahlen $u_1 \in \{0,1,\ldots,p-2\}$ mit $\mathrm{ggT}(u_1, p-1) = 1$ alle Gleichungen
$$\sum_{i=0}^{p-1} \alpha^{\frac{p^h-1}{p-1}u_1 ec_i} = 0 \quad \text{für} \quad 1 \le e < \frac{p-1}{h}$$
erfüllen. (Ein gesuchtes u_1 befindet sich unter den gefundenen Zahlen.) Die Anzahl der getesteten u_1's ist durch p beschränkt, in jedem Versuch müssen Summen mit p Summanden berechnet werden.
2. Teste im Fall $r|h$, welche Zahlen $u_r \in \{0,1,\ldots,p^r-2\}$ mit $\mathrm{ggT}(u_r, p^r-1) = 1$ alle Gleichungen
$$\sum_{i=0}^{p-1} \alpha^{\frac{p^h-1}{p^r-1}u_r ec_i} = 0 \quad \text{für} \quad 1 \le e < \frac{(p-1)r}{h}$$
erfüllen, wobei man nur solche u_r's testen muß, die für alle Teiler $r'|r$ eine Kongruenz $u_r \equiv u_{r'} \bmod p^{r'} - 1$ mit einem zuvor gefundenen $u_{r'}$ erfüllen.

Wir wollen kurz überlegen, wieviele u_r's zu testen sind. Im Fall $2|h$ hat man für u_2 die Einschränkungen $u_2 \in \{0,1,\ldots,p^2-2\}$, $\mathrm{ggT}(u_2, p^2-1) = 1$, $u_2 \equiv u_1 \bmod p-1$, so daß ungefähr p Fälle zu testen sind.
Im Fall $3|h$ ergeben sich aus $u_3 \in \{0,1,\ldots,p^3-2\}$, $\mathrm{ggT}(u_3, p^3-1) = 1$, $u_3 \equiv u_1 \bmod p-1$ ungefähr p^2 Testfälle für u_3.
Für $4|h$ hat man für u_4 die Bedingungen $u_4 \in \{0,1,\ldots,p^4-2\}$, $\mathrm{ggT}(u_4, p^4-1) = 1$, $u_4 \equiv u_2 \bmod p^2-1$, so daß es ungefähr p^2 Möglichkeiten für u_4 gibt.
Im Fall $6|h$ ist $u_6 \in \{0,1,\ldots,p^6-2\}$, $\mathrm{ggT}(u_6, p^6-1) = 1$, $u_6 \equiv u_2 \bmod p^2-1$ und $u_6 \equiv u_3 \bmod p^3-1$, so daß ungefähr

$$\frac{p^6-1}{\mathrm{kgV}(p^2-1, p^3-1)} = \frac{p^6-1}{\frac{(p^2-1)(p^3-1)}{\mathrm{ggT}(p^2-1,p^3-1)}} \approx \frac{(p^6-1)(p-1)}{(p^2-1)(p^3-1)} \approx p^2$$

Fälle für u_6 zu testen sind.

Beispiel: Wir betrachten für $p = 11$, $h = 4$ den öffentlichen Chor-Rivest-Schlüssel

$$((c_0,\ldots,c_{p-1}),p,h) = (8298, 12244, 8404, 467, 14058, 3467, 10750, 14420, 8180, 3310, 10934), 11, 4).$$

Wir wollen einen zugehörigen privaten Schlüssel konstruieren. Wir wählen $\alpha \in \mathbf{F}_{11^4}$ mit dem Minimalpolynom

$$f = x^4 + 6x^3 + 10x^2 + 3x + 8.$$

Für u_1 testen wir alle Zahlen aus $\{1,3,5,7,9\}$. Es bleibt nur $u_1 = 3$ übrig.
Für u_2 testen wir alle Zahlen mit $u_2 \equiv u_1 \bmod p-1$, $\mathrm{ggT}(u_2, p^2-1) = 1$, also die Zahlen der Menge

$$\{13, 23, 43, 53, 73, 83, 103, 113\}.$$

Den Test bestehen aber nur $u_2 = 13$ und $u_2 = 23$. Für u_4 testen wir alle Zahlen mit $u_4 \equiv 13$ oder $23 \bmod p^2-1$, $\mathrm{ggT}(u_4, p^4-1) = 1$, $0 \le u_4 \le p^4-2$. Dies gibt 240 Möglichkeiten, beginnend mit

$$13, 133, 253, 373, 493, 613, 733, 853, 973, 1093, 1213, 1333, 1453, 1573, 1693, 1813, 1933, 2053, 2173, 2293, 2413, 2533, 2653, 277$$

Den Test bestehen nur die Zahlen

$$u_4 = 3973, 12253, 3023, 14423.$$

Wir wählen $u_4 = 3973$ und berechnen

$$\gamma = \alpha^{u_4} = \alpha^3 + 10\alpha^2 + 8.$$

Damit wird

$$\xi = \frac{1}{\gamma^{c_1-c_0}-1} = 10\alpha^3 + \alpha^2 + 7$$

und

$$d \equiv (c_0 - \log_\gamma \xi) \pmod{p^h - 1}, \quad \text{also} \quad d = 10629.$$

Aus $\pi(i) = \gamma^{c_i - d} - \xi$ folgt dann

$$\pi = [0, 1, 4, 2, 5, 3, 9, 6, 10, 7, 8].$$

Das Minimalpolynom von ξ ist

$$F = x^4 + 10x^3 + 5x^2 + 10x + 2.$$

Wegen $\mathrm{ggT}(c_6 - d, p^h - 1) = 1$ und $121 \cdot (c_6 - d) \equiv 1 \bmod p^h - 1$ folgt

$$\gamma^{c_6 - d} = \xi + 9, \quad \gamma = (\xi + 9)^{121} = 10\xi + 4.$$

Also ist ein privater Schlüssel

$$(F, G, \pi, d) = (x^4 + 10x^3 + 5x^2 + 10x + 2, 10x + 4, [0, 1, 4, 2, 5, 3, 9, 6, 10, 7, 8], 10629).$$

Man überprüft leicht, daß dies tatsächlich den angegebenen öffentlichen Schlüssel liefert.

Bemerkung: Erfüllt u_r den Test

$$\sum_{i=0}^{p-1} \alpha^{\frac{p^h-1}{p^r-1} u_r e c_i} = 0 \quad \text{für} \quad 1 \le e < \frac{(p-1)r}{h},$$

so folgt durch Potenzieren mit p

$$\sum_{i=0}^{p-1} \alpha^{\frac{p^h-1}{p^r-1} p u_r e c_i} = 0 \quad \text{für} \quad 1 \le e < \frac{(p-1)r}{h},$$

also erfüllt auch $pu_r \pmod{p^r - 1}$ den Test und damit alle r Zahlen

$$u_r, \quad pu_r, \quad p^2 u_r, \quad \ldots, \quad p^{r-2} u_r, \quad p^{r-1} u_r \pmod{p^r - 1}.$$

Beim Durchprobieren möglicher u_r's können wir uns also auf einen Kandidaten aus jeder Konjugationsklasse beschränken.

Wir haben eine Maple-Funktion 'gamma_r_test' zur Bestimmung möglicher γ_r's bzw. u_r's geschrieben. Für $h|6$ gibt es die Funktion 'u_6_liste', die nacheinander Möglichkeiten für u_1, u_2, u_3, u_6 durchprobiert und dabei die Kongruenzbedingungen berücksichtigt. Ausgegeben wird dann eine Liste möglicher Werte für u_6. In unseren Beispielen war u_6 jedesmal bis auf Konjugation eindeutig bestimmt.

Beispiele: Wir haben für unsere Schlüssel $K_{p,h}$ mit Maple γ_6 bzw. u_6 (nach Wahl eines Elements α) bestimmt.

$K_{p,h}$	Stunden:Minuten:Sekunden
$K_{13,12}$	00:00:04
$K_{17,12}$	00:00:04
$K_{19,12}$	00:00:06
$K_{23,12}$	00:00:11
$K_{29,12}$	00:00:23
$K_{31,12}$	00:00:24
$K_{37,12}$	00:00:45
$K_{41,12}$	00:01:08
$K_{43,12}$	00:01:13
$K_{47,12}$	00:01:30
$K_{53,12}$	00:02:26
$K_{103,12}$	00:16:11
$K_{197,12}$	02:35:43
$K_{43,24}$	00:01:14
$K_{79,24}$	00:08:17
$K_{139,24}$	00:50:02
$K_{197,24}$	02:34:48
$K_{211,24}$	03:12:55
$K_{277,24}$	07:33:21

7.4. Bestimmung von π und $Q_6(x)$ mit Hilfe von γ_6 im Fall $h = 12$. Wir nehmen an, wir haben mit dem zuvor beschriebenen Verfahren γ_6 bestimmt.

$Q_6(x)$ ist ein Polynom vom Grad $\frac{h}{r} = \frac{12}{6} = 2$ mit $Q_6(\pi(i)) = \gamma_6^{c_i}$. Das Polynom $Q_6(x)$ ist durch die 3 Stellen $(\pi(i), Q_6(\pi(i))) = (\pi(i), \gamma_r^{c_i})$, $i = 0, 1, 2$ bestimmt. Die Polynominterpolationsformel liefert

$$Q_6(x) = \gamma_6^{c_0} \frac{(x-\pi(1))(x-\pi(2))}{(\pi(0)-\pi(1))(\pi(0)-\pi(2))} + \gamma_6^{c_1} \frac{(x-\pi(0))(x-\pi(2))}{(\pi(1)-\pi(0))(\pi(1)-\pi(2))} + \gamma_6^{c_2} \frac{(x-\pi(0))(x-\pi(1))}{(\pi(2)-\pi(0))(\pi(2)-\pi(1))}.$$

Insbesondere folgt dann für $i = 0, \ldots, p-1$

$$\gamma_6^{c_i} = \gamma_6^{c_0} \frac{(\pi(i)-\pi(1))(\pi(i)-\pi(2))}{(\pi(0)-\pi(1))(\pi(0)-\pi(2))} + \gamma_6^{c_1} \frac{(\pi(i)-\pi(0))(\pi(i)-\pi(2))}{(\pi(1)-\pi(0))(\pi(1)-\pi(2))} + \gamma_6^{c_2} \frac{(\pi(i)-\pi(0))(\pi(i)-\pi(1))}{(\pi(2)-\pi(0))(\pi(2)-\pi(1))}.$$

Wir kennen dabei $\gamma_6^{c_i}$, nicht jedoch $\pi(2)$. Praktisch findet man durch Koeffizientenvergleich $m_{0i}, m_{1i}, m_{2i} \in \mathbf{F}_p$ mit

$$\gamma_6^{c_i} = \gamma_6^{c_0} m_{0i} + \gamma_6^{c_1} m_{1i} + \gamma_6^{c_2} m_{2i}.$$

Wir schreiben $x_i = \pi(i)$ und haben dann

$$m_{0i} = \frac{(x_i - x_1)(x_i - x_2)}{(x_0 - x_1)(x_0 - x_2)}, \quad m_{1i} = \frac{(x_i - x_0)(x_i - x_2)}{(x_1 - x_0)(x_1 - x_2)}, \quad m_{2i} = \frac{(x_i - x_0)(x_i - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Daraus ergibt sich $m_{i0} + m_{i1} + m_{i2} = 1$ und

$$\begin{aligned} x_i^2 - (x_1 + x_2)x_i + x_1 x_2 &= m_{0i}(x_0 - x_1)(x_0 - x_2), \\ x_i^2 - (x_0 + x_2)x_i + x_0 x_2 &= m_{1i}(x_1 - x_0)(x_1 - x_2), \\ x_i^2 - (x_0 + x_1)x_i + x_0 x_1 &= m_{2i}(x_2 - x_0)(x_2 - x_1). \end{aligned}$$

Subtraktion der ersten beiden Gleichungen liefert

$$x_i = m_{0i} x_0 + m_{1i} x_1 + m_{2i} x_2$$

Setzt man dies wieder in die erste Gleichung ein, erhält man mit $x_0 = \pi(0) = 0$, $x_1 = \pi(1) = 1$

$$(m_{1i}^2 - m_{1i}) + 2m_{1i}m_{2i}x_2 + (m_{2i}^2 - m_{2i})x_2^2 = 0.$$

Dies sind p quadratische Gleichungen für x_2. Daraus kann man $x_2 = \pi(2)$ bestimmen und damit dann

$$\pi(i) = m_{1i} + m_{2i}\pi(2).$$

Mit der Kenntnis von $\pi(2)$ kann man nun $Q_6(x)$ bestimmen.

In unseren Beispielen mit den Schlüsseln $K_{p,12}$ hat dies immer funktioniert.

7.5. Bestimmung von π und $Q_6(x)$ mit Hilfe von γ_6 im Fall $h = 24$. Im Fall $h = 24$ ist $Q_6(x)$ ein Polynom vom Grad 4 mit Koeffizienten in $\mathbf{F}_{p^6}$. Es gilt $Q_6(\pi(i)) = \gamma_6^{c_i}$ für alle i. Da $Q_6(x)$ Grad 4 hat, ist $Q_6(x)$ durch die 5 Stellten $(\pi(i), \gamma_6^{c_i})$, $i = 0, 1, 2, 3, 4$ eindeutig bestimmt. Schreiben wir $x_i = \pi(i)$, so gilt

$$\begin{aligned} Q_6(x) &= \gamma_6^{c_0}\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)(x_0-x_4)} + \gamma_6^{c_1}\frac{(x-x_0)(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)(x_1-x_4)} \\ &+\gamma_6^{c_2}\frac{(x-x_0)(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)(x_2-x_4)} + \gamma_6^{c_3}\frac{(x-x_0)(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)(x_3-x_4)} \\ &+\gamma_6^{c_4}\frac{(x-x_0)(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_0)(x_4-x_1)(x_4-x_2)(x_4-x_3)}. \end{aligned}$$

Wir setzen

$$m_{0i} = \frac{(x_i-x_1)(x_i-x_2)(x_i-x_3)(x_i-x_4)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)(x_0-x_4)}, \quad m_{1i} = \frac{(x_i-x_0)(x_i-x_2)(x_i-x_3)(x_i-x_4)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)(x_1-x_4)},$$
$$m_{2i} = \frac{(x_i-x_0)(x_i-x_1)(x_i-x_3)(x_i-x_4)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)(x_2-x_4)}, \quad m_{3i} = \frac{(x_i-x_0)(x_i-x_1)(x_i-x_2)(x_i-x_4)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)(x_3-x_4)},$$
$$m_{4i} = \frac{(x_i-x_0)(x_i-x_1)(x_i-x_2)(x_i-x_3)}{(x_4-x_0)(x_4-x_1)(x_4-x_2)(x_4-x_3)}.$$

Damit gilt dann

$$\gamma_6^{c_i} = \gamma_6^{c_0}m_{0i} + \gamma_6^{c_1}m_{1i} + \gamma_6^{c_2}m_{2i} + \gamma_6^{c_3}m_{3i} + \gamma_6^{c_4}m_{4i}.$$

Die Elemente $\gamma_6^{c_i}$ liegen in $\mathbf{F}_{p^6}$. Wir nehmen an, $\gamma_6^{c_0}, \gamma_6^{c_1}, \gamma_6^{c_2}, \gamma_6^{c_3}, \gamma_6^{c_4}$ sind linear unabhängig über $\mathbf{F}_p$. (In den gerechneten Beispielen war dies stets der Fall.) Dann sind $m_{0i}, m_{1i}, m_{2i}, m_{3i}, m_{4i} \in \mathbf{F}_p$ eindeutig bestimmt und können bei Kenntnis von γ_6 und $c_0, \ldots, c_{p-1}$ bestimmt werden.

Wir kennen also $m_{0i}, \ldots, m_{4i}$, nicht jedoch $x_i = \pi(i)$ (bis auf $\pi(0) = 0$ und $\pi(1) = 1$). Wir haben dann die Gleichungen

$$\begin{aligned} g_{0i} &= (x_i-x_1)(x_i-x_2)(x_i-x_3)(x_i-x_4) - m_{0i}(x_0-x_1)(x_0-x_2)(x_0-x_3)(x_0-x_4), \\ g_{1i} &= (x_i-x_0)(x_i-x_2)(x_i-x_3)(x_i-x_4) - m_{1i}(x_1-x_0)(x_1-x_2)(x_1-x_3)(x_1-x_4), \\ g_{2i} &= (x_i-x_0)(x_i-x_1)(x_i-x_3)(x_i-x_4) - m_{2i}(x_2-x_0)(x_2-x_1)(x_2-x_3)(x_2-x_4), \\ g_{3i} &= (x_i-x_0)(x_i-x_1)(x_i-x_2)(x_i-x_4) - m_{3i}(x_3-x_0)(x_3-x_1)(x_3-x_2)(x_3-x_4), \\ g_{4i} &= (x_i-x_0)(x_i-x_1)(x_i-x_2)(x_i-x_3) - m_{4i}(x_4-x_0)(x_4-x_1)(x_4-x_2)(x_4-x_3). \end{aligned}$$

Wir haben also $5p$ Gleichungen mit p bzw. $p-2$ Unbekannten x_i.
Wir bilden

$$h_{1i} = \frac{g_{0i}-g_{1i}}{x_0-x_1}, \quad h_{2i} = \frac{g_{0i}-g_{2i}}{x_0-x_2}, \quad h_{3i} = \frac{g_{0i}-g_{3i}}{x_0-x_3}, \quad h_{4i} = \frac{g_{0i}-g_{4i}}{x_0-x_4},$$

was wieder polynomial in $x_0, x_1, x_2, x_3, x_4, x_i$ ist, dann

$$k_{2i} = \frac{h_{1i}-h_{2i}}{x_1-x_2}, \quad k_{3i} = \frac{h_{1i}-h_{3i}}{x_1-x_3}, \quad k_{4i} = \frac{h_{1i}-h_{4i}}{x_1-x_4},$$

was wieder polynomial in $x_0, x_1, x_2, x_3, x_4, x_i$ ist, schließlich

$$l_{3i} = \frac{k_{2i}-k_{3i}}{x_2-x_3}, \quad l_{4i} = \frac{k_{2i}-k_{4i}}{x_2-x_4}.$$

Nun ist

$$l_{3i} - l_{4i} = (1 - m_{0i} - m_{1i} - m_{2i} - m_{3i} - m_{4i})(x_3 - x_4).$$

Da $x_3 \neq x_4$ gelten muß, folgt

$$m_{0i} + m_{1i} + m_{2i} + m_{3i} + m_{4i} = 1.$$

Damit ergibt sich aus l_{3i} oder l_{4i}

$$x_i = m_{0i}x_0 + m_{1i}x_1 + m_{2i}x_2 + m_{3i}x_3 + m_{4i}x_4.$$

Wir haben

$$\begin{aligned} g_{0i} = g_{1i} = g_{2i} = g_{3i} = g_{4i} = 0 \quad &\Longleftrightarrow \quad g_{0i} = 0 \text{ und } g_{0i} - g_{1i} = g_{0i} - g_{2i} = g_{0i} - g_{3i} = g_{0i} - g_{4i} = 0 \\ &\Longleftrightarrow \quad g_{0i} = 0 \text{ und } h_{1i} = h_{2i} = h_{3i} = h_{4i} = 0 \\ &\Longleftrightarrow \quad g_{0i} = h_{1i} = 0 \text{ und } h_{1i} - h_{2i} = h_{1i} - h_{3i} = h_{1i} - h_{4i} = 0 \\ &\Longleftrightarrow \quad g_{0i} = h_{1i} = 0 \text{ und } k_{2i} = k_{3i} = k_{4i} = 0 \\ &\Longleftrightarrow \quad g_{0i} = h_{1i} = k_{2i} = 0 \text{ und } k_{2i} - k_{3i} = k_{2i} - k_{4i} = 0 \\ &\Longleftrightarrow \quad g_{0i} = h_{1i} = k_{2i} = 0 \text{ und } l_{3i} = l_{4i} = 0. \end{aligned}$$

Da $l_{3i} = l_{4i} = 0$ bereits gelöst wurde, bleiben nur die Gleichungen

$$g_{0i} = h_{1i} = k_{2i} = 0,$$

wobei g_{0i} homogen vom Grad 4, h_{1i} homogen vom Grad 3 und k_{2i} homogen vom Grad 2 in x_0, x_1, x_2, x_3, x_4 ist.

Wir setzen $x_0 = \pi(0) = 0$, $x_1 = 1 = \pi(1)$ und haben dann die $3p$ Gleichungen $g_{0i} = h_{1i} = k_{2i} = 0$ in x_2, x_3, x_4. Um x_2, x_3, x_4 zu bestimmen sind wir praktisch so vorgegangen: Bei festem i bilden wir die Resultanten

$$\begin{aligned} r_{02}(x_2, x_3) &= \text{Resultante}_{x_4}(g_{0i}(x_2, x_3, x_4), k_{2i}(x_2, x_3, x_4))(x_2, x_3), \\ r_{12}(x_2, x_3) &= \text{Resultante}_{x_4}(h_{1i}(x_2, x_3, x_4), k_{2i}(x_2, x_3, x_4))(x_2, x_3), \\ r_{0212}(x_2) &= \text{Resultante}_{x_3}(r_{02}(x_2, x_3), r_{12}(x_2, x_3))(x_2). \end{aligned}$$

Die Nullstellen von r_{0212} geben Möglichkeiten für $x_2 = \pi(2)$ an. Bei Betrachtung einiger i's war dann x_2 jedesmal eindeutig bestimmt. Auf ähnliche Weise kann man dann x_3 und schließlich x_4 bestimmen.

Damit erhält man dann die Permutation π und das Polynom $Q_6(x)$.

7.6. Bestimmung von ξ, γ und d. Nun ist

$$Q_6(x) = \begin{cases} \gamma_6^d (x + \xi)(x + \xi^{p^6}) & \text{im Fall } h = 12, \\ \gamma_6^d (x + \xi)(x + \xi^{p^6})(x + \xi^{p^{12}})(x + \xi^{p^{18}}) & \text{im Fall } h = 24. \end{cases}$$

Bestimmt man die Nullstellen von $Q_6(x)$ in $\mathbf{F}_{p^h}$, so erhält man bis auf Konjugation $-\xi$.
Wir bestimmen das Minimalpolynom $F(X) \in \mathbf{F}_p[X]$ von ξ über $\mathbf{F}_p$, z.B. mit dem Maple-Befehl

$$F(X) := \text{resultant}(X - \xi(x), f(x), x) \bmod p,$$

wobei $f(x)$ das Minimalpolynom von α über $\mathbf{F}_p$ war.
Jetzt kann man alles in Abhängigkeit von ξ ausdrücken, wobei ξ der Variablen X entspricht und man dann modulo $F(X)$ rechnet.

Es gilt

$$\gamma^{c_i - c_j} = \frac{\gamma^{c_i - d}}{\gamma^{c_j - d}} = \frac{\xi + \pi(i)}{\xi + \pi(j)}.$$

Wir suchen jetzt Indizes $i \neq j$ mit $\text{ggT}(c_i - c_h, p^h - 1) = 1$. Dann erhalten wir

$$\gamma = \left(\frac{\xi + \pi(i)}{\xi + \pi(j)} \right)^{\frac{1}{c_i - c_j} \pmod{p^h - 1}} = G(\xi),$$

wobei $G(X)$ ein Polynom vom Grad $\leq h - 1$ in X ist.

Aus

$$\gamma^{c_0 - d} = \xi$$

erhalten wir durch Logarithmenberechnung

$$d \equiv c_0 - \log_\gamma \xi.$$

(Leider ist hier eine Logarithmenberechnung nötig!)

Damit sind alle Größen γ, ξ, π, d bestimmt, (F, G, π, d) ist ein privater Chor-Rivest-Schlüssel zu $((c_0, \ldots, c_{p-1}), p, h)$.

Bemerkung: Zum dargestellten Verfahren haben wir Maple-Funktionen 'angriff_h12' und 'angriff_h24' geschrieben, die zu einem öffentlichen Chor-Rivest-Schlüssel mit $h = 12$ bzw. $h = 24$ einen zugehörigen privaten Schlüssel bestimmen.

7.7. Beispiele. Wir haben auf die öffentlichen Schlüssel von $K_{p,h}$ die Maple-Funktionen 'angriff_h12' bzw. 'angriff_h24' angewandt und einen zugehörigen privaten Schlüssel erhalten. Die Tabelle gibt die Rechenzeiten an, wobei die wesentliche Zeit zur Bestimmung von u_6 verwendet wird.

$K_{p,h}$	Berechnung von u_6	Restzeit
$K_{13,12}$	00:00:04	00:00:01
$K_{17,12}$	00:00:04	00:00:01
$K_{19,12}$	00:00:06	00:00:01
$K_{23,12}$	00:00:11	00:00:01
$K_{29,12}$	00:00:23	00:00:02
$K_{31,12}$	00:00:24	00:00:02
$K_{37,12}$	00:00:45	00:00:02
$K_{41,12}$	00:01:08	00:00:02
$K_{43,12}$	00:01:13	00:00:03
$K_{47,12}$	00:01:30	00:00:02
$K_{53,12}$	00:02:26	00:00:02
$K_{103,12}$	00:16:11	00:00:03
$K_{197,12}$	02:35:43	00:00:07
$K_{43,24}$	00:01:14	00:00:23
$K_{79,24}$	00:08:17	00:00:28
$K_{139,24}$	00:50:02	00:00:29
$K_{197,24}$	02:34:48	00:00:33
$K_{211,24}$	03:12:55	00:00:44
$K_{277,24}$	07:33:21	00:00:51

Literatur: [**ChRi**], [**ScHo**], [**Va**].

KAPITEL 5

Angriffe auf RSA-Schlüssel mit kleinem privaten Exponenten

1. Das RSA-Kryptosystem

Wir erinnern an die RSA-Verschlüsselung:

1. Es liegt folgender mathematische Sachverhalt zugrunde: Sind p und q verschiedene ungerade Primzahlen und $N = pq$, wählt man natürliche Zahlen e und d mit

   $$ed \equiv 1 \bmod \varphi(N) \quad (\text{und } \varphi(N) = \varphi(pq) = (p-1)(q-1)),$$

   so ist die Abbildung

   $$E_{(N,e)} : \mathbf{Z}/N\mathbf{Z} \to \mathbf{Z}/N\mathbf{Z}, \quad x \mapsto x^e \bmod N$$

   bijektiv mit der Umkehrabbildung

   $$D_{(N,d)} : \mathbf{Z}/N\mathbf{Z} \to \mathbf{Z}/N\mathbf{Z}, \quad y \mapsto y^d \bmod N.$$

2. Kennt man das Paar (N, e), kennt man aber die Faktorisierung von N nicht bzw. kann man diese nicht bestimmen, so kann man im allgemeinen auch (N, d) nicht bestimmen.
3. Dies benutzt man zur Konstruktion des RSA-Public-Key-Kryptosystems mit öffentlichen Schlüsseln (N, e), privaten Schlüsseln (N, d), Verschlüsselungsabbildungen $E_{(N,e)}$, Entschlüsselungsabbildungen $D_{(N,d)}$. Bei der Schlüsselkonstruktion sollten die Primzahlen p und q so gewählt werden, daß man nicht erwarten kann, daß einem gängigen Faktorisierungverfahren die Faktorisierung von N in angemessener Zeit gelingt.

Bemerkung: Wir haben Maple-Funktionen 'rsa_encrypt' und 'rsa_decrypt' für RSA-Verschlüsselung und RSA-Entschlüsselung.

Ist $N = pq$, wählt man d mit $1 < d < \varphi(N)$ und $\mathrm{ggT}(d, \varphi(N)) = 1$, so erhält man mit $e \equiv \frac{1}{d} \bmod \varphi(N)$ ein RSA-Schlüsselpaar $((N, e), (N, d))$. Wir wollen untersuchen, wie groß d mindestens sein sollte, damit man aus der Kenntnis von (N, e) nicht auf (N, d) schließen kann, wobei vorausgesetzt wird, daß man N praktisch nicht faktorisieren kann.

1. d sollte nicht zu klein sein, so daß man durch Probieren von $d = 3, 5, 7, \ldots$ auf d schließen kann. (Man testet z.B., ob $2^{ed} \equiv 2 \bmod N$ gilt.)
2. Wiener hat gezeigt, daß man im Fall

   $$d \lessapprox N^{0.25}$$

   aus (N, e) mit einem Kettenbruchverfahren (N, d) berechnen kann. Also sollte d sicherheitshalber (deutlich) größer als $N^{0.25}$ sein.
3. Boneh und Durfee haben Verfahren mit Gittern beschrieben, mit denen man eventuell aus (N, e) den privaten Schlüssel (N, d) berechnen kann, wenn

   $$d \lessapprox N^{0.284} \text{ (1. Gittervariante)} \quad \text{bzw.} \quad d \lessapprox N^{0.292} \text{ (2. Gittervariante)}$$

   gilt und alles gut geht. Also sollte d sicherheitshalber (deutlich) größer als $N^{0.292}$ gewählt werden.

Die Angriffe von Wiener und Boneh/Durfee werden in den folgenden Abschnitten beschrieben.

Version vom 22. Juli 2002

2. Kettenbrüche und der Satz von Wiener

Unter einem Kettenbruch versteht man einen Ausdruck der Form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots \cfrac{1}{a_{n-2} + \cfrac{1}{a_{n-1} + \cfrac{1}{a_n}}}}}}}$$

Aus schreibtechnischen Gründen ist dafür auch die Bezeichnung

$$[a_0, a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}, a_n]$$

gebräuchlich.

Beispiel:

$$[2,3,5,7] = 2 + \cfrac{1}{3 + \cfrac{1}{5 + \cfrac{1}{7}}} = 2 + \cfrac{1}{3 + \cfrac{1}{\frac{36}{7}}} = 2 + \cfrac{1}{3 + \frac{7}{36}} = 2 + \cfrac{1}{\frac{115}{36}} = 2 + \frac{36}{115} = \frac{266}{115}$$

Kettenbruchalgorithmus: Gegeben sei eine reelle Zahl α.

1. Man setzt $\alpha_0 = \alpha$ und berechnet rekursiv für $i \geq 0$

   $$a_i = \lfloor \alpha_i \rfloor \quad \text{und} \quad \alpha_{i+1} = \frac{1}{\alpha_i - a_i}, \text{ solange } \alpha_i \notin \mathbf{Z}.$$

   Ist $\alpha_n \in \mathbf{Z}$, so sagt man, die Kettenbruchentwicklung bricht ab.
2. Ist $a_i = \lfloor \alpha_i \rfloor \neq \alpha_i$, so ist $0 < \alpha_i - a_i < 1$, also $\alpha_{i+1} = \frac{1}{\alpha_i - a_i} > 1$ und damit $a_{i+1} \geq 1$. Also: $a_0 \in \mathbf{Z}$ und $a_i \in \mathbf{N}$ für $i \geq 1$.
3. Nach Konstruktion gilt

   $$\alpha_i = a_i + \frac{1}{\alpha_{i+1}}$$

   und damit

   $$\alpha = \alpha_0 = a_0 + \frac{1}{\alpha_1} = a_0 + \cfrac{1}{a_1 + \frac{1}{\alpha_2}} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \frac{1}{\alpha_3}}} = \cdots = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots \cfrac{1}{a_{n-1} + \frac{1}{\alpha_n}}}}}$$

   bzw. mit der einfacheren Notation

   $$\alpha = [\alpha_0] = [a_0, \alpha_1] = [a_0, a_1 + \frac{1}{\alpha_2}] = [a_0, a_1, \alpha_2] = \cdots = [a_0, a_1, a_2, \ldots, a_{n-1}, \alpha_n].$$

4. Man kann zeigen, daß genau die rationalen Zahlen endliche Kettenbruchentwicklungen haben, d.h. für $\alpha \in \mathbf{Q}$ liefert der Kettenbruchalgorithmus $a_0 \in \mathbf{Z}$, $a_1, \ldots, a_n \in \mathbf{N}$ mit

   $$\alpha = [a_0, a_1, \ldots, a_n].$$

Bemerkung: Die Kettenbruchentwicklung einer rationalen Zahl kann man mit den Maple-Funktionen 'kbe' oder 'numtheory[cfrac]' bestimmen.

Von zentraler Wichtigkeit in der Theorie der Kettenbrüche ist folgender Satz:

SATZ. *Sei α eine rationale Zahl mit der Kettenbruchentwicklung $\alpha = [a_0, a_1, \ldots, a_n]$. Ist β eine rationale Zahl mit $\beta = \frac{x}{y}$, $x \in \mathbf{Z}$, $y \in \mathbf{N}$ und*

$$\left|\alpha - \frac{x}{y}\right| < \frac{1}{2y^2},$$

so gibt es ein m mit $0 \leq m \leq n$ und

$$\beta = \frac{x}{y} = [a_0, a_1, \ldots, a_m].$$

(Man nennt $[a_0, a_1, \ldots, a_m]$ einen Näherungsbruch für α.)

Wir wollen diesen Satz jetzt auf das RSA-Problem anwenden.

Überlegung: Sei (N, e) ein öffentlicher RSA-Schlüssel. Dann gibt es verschiedene ungerade Primzahlen p und q mit $N = pq$, eine natürliche Zahl d mit $ed \equiv 1 \bmod \varphi(N)$ (und o.E. $1 < e, d < \varphi(N)$). Die Kongruenz liefert eine natürliche Zahl k mit

$$ed - k\varphi(N) = 1.$$

Daher ist

$$\frac{e}{\varphi(N)} - \frac{k}{d} = \frac{1}{d\varphi(N)}$$

und somit

$$\frac{e}{\varphi(N)} \approx \frac{k}{d}.$$

Gilt nun $N \approx \varphi(N)$, so folgt

$$\frac{e}{N} \approx \frac{k}{d}$$

und man kann untersuchen, ob $\frac{k}{d}$ als Näherungsbruch in der Kettenbruchentwicklung von $\frac{e}{N}$ vorkommt. Dies geschieht in nachfolgendem Satz:

SATZ (M. Wiener). *Sei $N = pq$ mit $q < p < 2q$ und $d < \frac{1}{3}N^{\frac{1}{4}}$. Dann gilt*

$$\left|\frac{e}{N} - \frac{k}{d}\right| < \frac{1}{2d^2},$$

also taucht $\frac{k}{d}$ als Näherungsbruch in der Kettenbruchentwicklung von $\frac{e}{N}$ auf.

Beweis: Aus $q < p$ und $N = pq$ folgt $q < \sqrt{N}$. Mit $p < 2q$ folgt $p < 2\sqrt{N}$, was schließlich zu

$$0 < N - \varphi(N) = N - (N + 1 - p - q) = p + q - 1 < 3\sqrt{N}$$

führt. Mit $e < \varphi(N)$ ergibt sich $ke < k\varphi(N) = ed - 1 < ed$ und damit $k < d$.
Mit den obigen Bezeichnungen und $9d^2 < \sqrt{N}$ gilt:

$$\begin{aligned} \left|\frac{e}{N} - \frac{k}{d}\right| &= \left|\frac{ed - kN}{Nd}\right| = \left|\frac{ed - k\varphi(N) + k\varphi(N) - kN}{Nd}\right| = \left|\frac{1 - k(N - \varphi(N))}{Nd}\right| = \\ &= \frac{k(N - \varphi(N)) - 1}{Nd} < \frac{k(N - \varphi(N))}{Nd} < \frac{d \cdot 3\sqrt{N}}{Nd} = \frac{3}{\sqrt{N}} < \frac{3}{9d^2} = \frac{1}{3d^2} < \frac{1}{2d^2}, \end{aligned}$$

was gezeigt werden sollte. ■

Beispiel: Bei den folgenden Werten

$$\begin{aligned} N &= 588432708985352048988609608877396213831699739855070740014173675325399134289 53, \\ p &= 172924831312834108056472927590512528249, \\ q &= 340282366920938463463374607431768211297, \\ d &= 5118143743528119887, \\ e &= 368205529222454797902388474401748768026679569273728283371254098130263139281 11, \\ k &= \frac{ed-1}{(p-1)(q-1)} = 3202624186156992207 \end{aligned}$$

sind die Voraussetzungen des Satzes von Wiener erfüllt. (N hat 256 Bits.) Für die Kettenbruchentwicklungen von $\frac{e}{N}$ und $\frac{k}{d}$ findet man

$$\begin{aligned} \frac{e}{N} &= [0,1,1,1,2,20,1,3,8,1,1,1,1,1,1,8,7,1,2,1,3,3,7,197,40,1,2,2,3,8,7,1,3,2,11,2,6,1,1, \\ &\quad 15,4,2,5,1,1,3,2,8,18,1,1,1,4,17,1,5,1,1,1,2,2,2,4,2,1,1,3,2,4,1,1,1,13,403,19,1, \\ &\quad 2,2,15,1,2,11,1,4,1,2,2,25,13,1,2,1,1,7,2,5,1,2,2,1,3,4,3,19,4,40,1,12,22,1,1,3,1, \\ &\quad 1,76,1,5,24,2,7,35,5,2,5,1,1,1,1,1,27,2,15,2,3,7,1,1,2,3,5,1,2] \\ \frac{k}{d} &= [0,1,1,1,2,20,1,3,8,1,1,1,1,1,1,8,7,1,2,1,3,3,7,197,40,1,2,2,3,8,7,1,3,2,11,2] \end{aligned}$$

und man sieht, daß tatsächlich $\frac{k}{d}$ ein Näherungsbruch von $\frac{e}{N}$ ist.

Bemerkung: Ist (N,e) gegeben, so kann man die Kettenbruchentwicklung von $\frac{e}{N} = [a_0, a_1, \ldots, a_n]$ und die zugehörigen Näherungsbrüche $\frac{k_i}{d_i} = [a_0, a_1, \ldots, a_i]$ bilden. Nun löst man die beiden Gleichungen (für jedes i)

$$N - pq = 0 \quad \text{und} \quad ed_i - 1 - k_i(p-1)(q-1) = 0$$

mit den beiden Unbekannten p_i, q_i. Findet man eine Lösung mit $p_i, q_i \in \mathbf{Z}$, $p_i, q_i \geq 2$, so hat man eine Faktorisierung von N und man kennt sofort den privaten RSA-Schlüssel (N,d). Wir haben dazu eine Maple-Funktion 'kb_rsa' geschrieben.

3. Der Ansatz von Boneh und Durfee

Boneh und Durfee geben Gitterverfahren an, mit denen man aus einem öffentlichen RSA-Schlüssel (N,e) den zugehörigen privaten (N,d) bzw. die Faktorisierung $N = pq$ erhalten kann, wenn alles gut geht und in etwa

$$d \lessapprox N^{0.284} \text{ (1. Verfahren)} \quad \text{bzw.} \quad d \lessapprox N^{0.292} \text{ (2. Verfahren)}$$

gilt. Das Funktionieren kann man bis jetzt noch nicht (theoretisch) beweisen, außerdem muß eventuell der Gitterrang so hoch gewählt werden, daß eine praktische Berechnung schwierig wird. Wir skizzieren im folgenden den Weg zu $d \lessapprox N^{0.284}$.

Überlegungen:

1. Gegeben sei ein öffentlicher RSA-Schlüssel (N,e), Ziel eines Angreifers ist es, den zugehörigen privaten Schlüssel (N,d) zu bestimmen. Ist $N = pq$ und $s = p+q$, so folgt $\varphi(N) = (p-1)(q-1) = N + 1 - s$, also mit $ed \equiv 1 \bmod \varphi(N)$

   $$d \equiv \frac{1}{e} \pmod{N+1-s}.$$

   Also genügt es, s zu bestimmen. (Dann kann man auch N faktorisieren, da p und q Nullstellen des quadratischen Polynoms $x^2 - sx + N$ sind.)
2. Aus $ed \equiv 1 \bmod \phi(N)$ folgt die Existenz einer ganzen Zahl k mit $ed = 1 + k\phi(N) = 1 + k(N+1-s)$ und somit ist

   $$k(N+1-s) + 1 \equiv 0 \bmod e.$$

   (Es gilt $1 \leq k = \frac{ed-1}{\phi(N)} < \frac{ed}{\phi(N)} < d$ wegen $e < \phi(N)$.) Definieren wir ein Polynom

   $$f(x,y) = x(N+1-y) + 1 \in \mathbf{Z}[x,y],$$

so gilt

$$f(k,s) \equiv 0 \bmod e,$$

d.h. (k,s) ist eine Lösung der Gleichung $f(x,y) \equiv 0 \bmod e$.

3. Leider hat die Gleichung $f(x,y) \equiv 0 \bmod e$ zuviele Lösungen, die man auch leicht angeben kann:

$$\{(x_0,y_0) : 0 \le x_0, y_0 \le e-1, f(x_0,y_0) \equiv 0 \bmod e\} =$$
$$= \{(x_0, (N+1+\frac{1}{x_0}) \bmod e) : 0 \le x_0 \le e-1, \mathrm{ggT}(x_0,e) = 1\}.$$

Soll die Lösung $(x,y) = (k,s)$ ausgezeichnet sein, so braucht man noch einschränkende Bedingungen.

4. Wir wollen die Größe der Lösung (k,s) von $f(x,y) \equiv 0 \bmod e$ genauer anschauen. Dabei machen wir ein paar Einschränkungen, wobei das Zeichen $\approx$ keine zu präzise Bedeutung haben soll:
   - $e \approx N$ und $p \approx q \approx \sqrt{N}$.
   - Dann ist $s \approx 2\sqrt{N} \approx s^{0.5}$.
   - Ist $\delta < 1$ und $d \le N^\delta$, so ist $k \lessapprox e^\delta$.

   Ist δ klein, so suchen wir also nach (im Verhältnis zu e) kleinen Lösungen (k,s) von $f(x,y) \equiv 0 \bmod e$. Boneh und Durfee haben skizziert, wie man hier vorgehen kann.

Das folgende Lemma präzisiert einige der groben Abschätzungen der Vorüberlegung.

LEMMA. *Seien $p \neq q$ ungerade Primzahlen, $N = pq$ und e,d ganze Zahlen mit $1 < d,e < \varphi(N)$ und $ed \equiv 1 \bmod \phi(N)$, $k \in \mathbf{N}$ mit $ed = 1 + k\varphi(N)$, $s \in \mathbf{N}$ mit $s = p+q$. Sei δ eine reelle Zahl mit $0 < \delta < 1$ und $d \le N^\delta$. Wir setzen voraus, daß*

$$p < q < 2p$$

gilt. Dann gilt die Gleichung

$$k(N+1-s) + 1 \equiv 0 \bmod e$$

und für k und s haben wir folgende Abschätzungen:

1. *Es gilt*
$$s \le \lfloor \frac{3}{\sqrt{2}}\sqrt{N} \rfloor \quad \text{und} \quad k \le \lfloor \frac{eN^\delta}{N - \lfloor \frac{3}{\sqrt{2}}\sqrt{N} \rfloor} \rfloor.$$
2. $$k \le \frac{1}{1 - \sqrt{\frac{9}{2N}}} e^\delta,$$
   insbesondere $k \le 1.01e^\delta$ für $N \ge 100000$.
3. *Gilt $e > \frac{1}{10}N$, so folgt*
$$s \le \sqrt{45e} \le 6.71e^{\frac{1}{2}}.$$

Außerdem sind p und q Nullstellen der Gleichung $x^2 - sx + N = 0$, können also bei Kenntnis von s und N berechnet werden.

Beweis: Zunächst ist

$$\varphi(N) = (p-1)(q-1) = pq - (p+q) + 1 = N + 1 - s$$

und damit

$$ed = 1 + k\varphi(N) = 1 + k(N+1-s),$$

was modulo e die behauptete Gleichung $k(N+1-s)+1 \equiv 0 \bmod e$ ergibt.
Mit $p < q < 2p$ erhält man durch etwas Funktionsdiskussion die Abschätzung

$$s = p + q < (\frac{1}{\sqrt{2}} + \sqrt{2})\sqrt{N} = \frac{3}{\sqrt{2}}\sqrt{N}, \quad \text{also} \quad s \le \lfloor \frac{3}{\sqrt{2}}\sqrt{N} \rfloor.$$

Damit ergibt sich

$$k = \frac{ed-1}{\varphi(N)} = \frac{ed-1}{N+1-s} \le \frac{ed}{N-s} \le \frac{eN^\delta}{N - \lfloor \frac{3}{\sqrt{2}}\sqrt{N} \rfloor}.$$

Wir können dies auch anders formulieren

$$k \le \frac{ed}{N-s} \le \frac{eN^\delta}{N-\sqrt{\frac{9N}{2}}} = \frac{e^\delta e^{1-\delta} N^\delta}{N-\sqrt{\frac{9N}{2}}} \le \frac{e^\delta N^{1-\delta} N^\delta}{N-\sqrt{\frac{9N}{2}}} = \frac{e^\delta N}{N-\sqrt{\frac{9N}{2}}} = \frac{1}{1-\sqrt{\frac{9}{2N}}} e^\delta.$$

Mit $e > \frac{1}{10}N$ folgt $N < 10e$ und damit

$$s \le \frac{3}{\sqrt{2}}\sqrt{N} = \sqrt{\frac{9N}{2}} \le \sqrt{\frac{90e}{2}} = \sqrt{45e} \le 6.71 e^{\frac{1}{2}},$$

was alles beweist. ■

Überlegungen:

1. Zu einem öffentlichen RSA-Schlüssel (N, e) bilden wir das Polynom $f(x, y) = x(N + 1 - y) + 1$, wofür mit $N = pq$, $s = p + q$, $k = \frac{ed-1}{\varphi(N)}$ die Gleichung
$$f(k, s) \equiv 0 \bmod e$$
gilt. Wir betrachten jetzt für $m \in \mathbf{N}$ die Menge
$$P_m = \{F(x, y) \in \mathbf{Z}[x, y] : F(k, s) \equiv 0 \bmod e^m\}$$
aller Polynome, die modulo e^m die Nullstelle (k, s) haben. Man kann viele Polynome aus P_m angeben, auch wenn man nur den öffentlichen Schlüssel (N, e) kennt, z.B. gilt für beliebige $a_i(x, y) \in \mathbf{Z}[x, y]$
$$\sum_{i=0}^{m} a_i(x, y) e^{m-i} f(x, y)^i = a_0(x, y)e^m + a_1(x, y)e^{m-1} f(x, y) + \cdots + a_m(x, y) f(x, y)^m \in P_m,$$
denn
$$\sum_{i=0}^{m} a_i(k, s) e^{m-i} f(k, s)^i = \sum_{i=0}^{m} a_i(k, s) e^{m-i} (ed)^i = \sum_{i=0}^{m} a_i(k, s) d^i e^m \equiv 0 \bmod e^m.$$
2. Wir betrachten jetzt folgende Teilmenge von P:
$$P_\infty = \{F(x, y) \in \mathbf{Z}[x, y] : F(k, s) = 0\}.$$
Findet man $F_1, F_2 \in P_\infty$ mit $\mathrm{ggT}(F_1, F_2) = 1$, so ist
$$(k, s) \in \{(x, y) \in \mathbf{R}^2 : F_1(x, y) = F_2(x, y) = 0\} \quad \text{und}$$
$$\#\{(x, y) \in \mathbf{R}^2 : F_1(x, y) = F_2(x, y) = 0\} < \infty,$$
bzw. nach Elimination einer Variablen (Resultantenbildung)
$$\mathrm{Resultante}_x(F_1, F_2)(s) = 0, \quad \mathrm{Resultante}_y(F_1, F_2)(k) = 0,$$
sodaß man leicht den privaten Schlüssel (N, d) bestimmen kann. Die Frage ist nun, wie man Elemente von P_∞ finden kann.
3. Es gilt
$$P_1 \supseteq P_2 \supseteq P_3 \supseteq \cdots \supseteq P_{m-1} \supseteq P_m \supseteq P_{m+1} \supseteq \cdots \supseteq P_\infty$$
und
$$\cap_{m \ge 1} P_m = P_\infty.$$

Das folgende Lemma von Howgrave-Graham gibt ein Kriterium, wann kleine Nullstellen modularer Gleichungen Nullstellen über $\mathbf{Z}$ sind.

LEMMA (Howgrave-Graham). *Sei $h(x, y) = \sum_{i=1}^{w} c_i x^{a_i} y^{b_i} \in \mathbf{Z}[x, y]$ ein Polynom und seien M, X, Y natürliche Zahlen. Für $x_0, y_0 \in \mathbf{Z}$ gilt dann:*

$$\begin{cases} h(x_0, y_0) \equiv 0 \bmod M \\ |x_0| \le X, |y_0| \le Y \\ ||h(Xx, Yy)|| < \frac{M}{\sqrt{w}} \end{cases} \implies h(x_0, y_0) = 0 \quad (\textit{in } \mathbf{Z}).$$

Beweis: Wir erhalten mit der Schwarzen Ungleichung (für das Skalarprodukt)

$$\begin{aligned} |h(x_0,y_0)| &= |(a_1 x_0^{a_1} y_0^{b_1}, \ldots, a_w x_0^{a_w} y_0^{b_w}) \cdot \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}| \leq ||(a_1 x_0^{a_1} y_0^{b_1}, \ldots, a_w x_0^{a_w} y_0^{b_w})|| \cdot || \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} || = \\ &= \sqrt{\sum_{i=0}^{w} a_i^2 x_0^{2a_i} y_0^{2b_i}} \cdot \sqrt{w} \leq \sqrt{\sum_{i=0}^{w} a_i^2 X^{2a_i} Y^{2b_i}} \cdot \sqrt{w} = ||h(Xx,Yy)|| \cdot \sqrt{w}. \end{aligned}$$

Die Voraussetzung $||h(Xx,Yy)|| < \frac{M}{\sqrt{w}}$ liefert nun $|h(x_0,y_0)| < M$. Da $h(x_0,y_0)$ durch M teilbar ist, folgt $h(x_0,y_0) = 0$, wie behauptet. ■

Überlegungen: Wir verwenden die vorangegangenen Bezeichnungen.

- Für alle Polynome $h(x,y) \in P_m$ gilt $h(k,s) \equiv 0 \bmod e^m$. Bei geeigneten Voraussetzungen (wie im zugehörigen Lemma) kann man abschätzen $|k| \leq 1.01e^{\delta}$, $|s| \leq 6.71e^{0.5}$. Setzt man also $X = 1.01e^{\delta}$, $Y = 6.71e^{0.5}$ und gilt die Normungleichung $\|h(Xx,Yy)\| < e^m/\sqrt{w}$ (mit geeignetem w), so folgt mit dem Lemma von Howgrave-Graham sofort $h(k,s) = 0$. Findet man zwei Polynome $h_1(x,y), h_2(x,y)$ mit dieser Eigenschaft, so ist $h_1(k,s) = h_2(k,s) = 0$ und man kann hoffen, daß man damit $s = p + q$ bestimmen kann.
- Die Bedingung $\|h(Xx,Yy)\| < e^m/\sqrt{w}$ zeigt, daß man nach Elementen $h(x,y) \in P_m$ mit kleiner Norm $\|h(Xx,Yy)\|$ suchen sollte. Hier liegt die Verwendung von Gittern nahe, da man für LLL-reduzierte Gitterbasen einige Abschätzungen kennt.
- Wir definieren im folgenden ein Gitter

  $$\Lambda(N,e,m,t) \subseteq P_m,$$

  abhängig von einigen Parametern, und testen, ob Elemente h_1, h_2 einer LLL-reduzierten Gitterbasis die Eigenschaft $h_1(k,s) = h_2(k,s) = 0$ haben.

Das Gitter $\Lambda(N,e,m,t,X,Y)$: Seien $m \geq 1$, $t \geq 0$,

$$\begin{aligned} f(x,y) &= x(N+1-y)+1, \\ g_{ik}(x,y) &= e^{m-k} x^i f(x,y)^k, \quad k = 0,\ldots,m, \quad i = 0,\ldots,m-k, \\ h_{jk}(x,y) &= e^{m-k} y^j f(x,y)^k, \quad k = 0,\ldots,m, \quad j = 1,\ldots,t, \end{aligned}$$

und mit diesen Bezeichnungen

$$\Lambda(N,e,m,t,X,Y) = \sum_{k=0}^{m} \sum_{i=0}^{m-k} \mathbf{Z} g_{ik}(Xx,Yy) + \sum_{k=0}^{m} \sum_{j=1}^{t} \mathbf{Z} h_{jk}(Xx,Yy).$$

Bestimmung von Rang und Determinante des Gitters $\Lambda(N,e,m,t,X,Y)$: Die folgenden Monome $x^u y^v$ treten bei den Polynomen des Gitters auf, die wir gleich folgendermaßen anordnen:

- Zunächst werden die Monome mit $u \geq v$ lexikographisch aufgelistet, wobei zuerst y berücksichtigt wird:

  $$1, x, \ldots, x^m, \quad xy, x^2y, \ldots, x^m y, \quad \ldots, \quad x^{m-1}y^{m-1}, x^m y^{m-1}, \quad x^m y^m.$$

  $(v = 0,\ldots,m, u = v,\ldots,m)$
- Dann werden alle Monome mit $u < v$ lexikographisch aufgelistet, wobei zuerst x berücksichtig wird:

  $$y, y^2, \ldots, y^t, \quad xy^2, xy^3, \ldots, xy^{1+t}, \quad \ldots, \quad x^m y^{m+1}, x^m y^{m+2}, \ldots, x^m y^{m+t}.$$

  $(u = 0,\ldots,m, v = u+1,\ldots,u+t.)$

Also

$$\Lambda(N,e,m,t,X,Y) \subseteq \sum_{v=0}^{m}\sum_{u=v}^{m} \mathbf{Z}x^u y^v + \sum_{u=0}^{m}\sum_{v=u+1}^{u+t} x^u y^v.$$

Listen wir die Polynome in der oben angegeben Reihenfolge auf, also

$$g_{00}, g_{10}, \ldots, g_{m0}, \quad g_{01}, g_{11}, \ldots, g_{m-1,1}, \quad \ldots, \quad g_{0,m-1}, g_{1,m-1}, \quad g_{0m},$$

$$h_{10}, h_{20}, \ldots, h_{t0}, \quad h_{11}, h_{21}, \ldots, h_{t1}, \quad \ldots, \quad h_{1m}, h_{2m}, \ldots, h_{tm},$$

und tragen wir die Koeffizienten zeilenweise in eine Matrix $M = M(N,e,m,t,X,Y)$ ein, so sieht man aus

$$\begin{aligned} g_{ik} &= \cdots + e^{m-k}(-1)^k x^{i+k} y^k, \\ h_{jk} &= \cdots + e^{m-k}(-1)^k x^k y^{j+k}, \end{aligned}$$

daß $M(N,e,m,t,X,Y)$ eine (quadratische) Diagonalmatrix wird.
Für den Rang gilt

$$\mathrm{Rang}(\Lambda) = \sum_{v=0}^{m}\sum_{u=v}^{m} 1 + \sum_{u=0}^{m}\sum_{v=u+1}^{u+t} 1 = \frac{1}{2}(m+1)(m+2t+2).$$

Für die Determinante erhält man durch Aufmultiplizieren der Diagonalelemente von $M(N,e,m,t,X,Y)$

$$\begin{aligned} \det\Lambda &= |\prod_{k=0}^{m}\prod_{i=0}^{m-k} e^{m-k}(-1)^k X^{i+k}Y^k \cdot \prod_{k=0}^{m}\prod_{j=1}^{t} e^{m-k}(-1)^k X^k Y^{j+k}| = \\ &= e^{\sum_{k=0}^{m}\sum_{i=0}^{m-k}(m-k)+\sum_{k=0}^{m}\sum_{j=1}^{t}(m-k)} \\ &\quad \cdot X^{\sum_{k=0}^{m}\sum_{i=0}^{m-k}(i+k)+\sum_{k=0}^{m}\sum_{j=1}^{t}k} Y^{\sum_{k=0}^{m}\sum_{i=0}^{m-k}k+\sum_{k=0}^{m}\sum_{j=1}^{t}(j+k)} = \\ &= (eX)^{\frac{1}{6}m(m+1)(2m+3t+4)} \cdot Y^{\frac{1}{6}(m+1)(m^2+2m+3tm+3t^2+3t)}. \end{aligned}$$

Beispiel: Hier ist die Matrix $M(N,e,3,1,X,Y)$, wobei $A = N+1$ ist:

$$\begin{array}{cccccccccccccc}
e^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & e^3X & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & e^3X^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & e^3X^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
e^2 & e^2XA & 0 & 0 & -e^2XY & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & e^2X & e^2X^2A & 0 & 0 & -e^2X^2Y & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & e^2X^2 & e^2X^3A & 0 & 0 & -e^2X^3Y & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
e & 2eXA & eX^2A^2 & 0 & -2eXY & -2eX^2YA & 0 & eX^2Y^2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & eX & 2eX^2A & eX^3A^2 & 0 & -2eX^2Y & -2eX^3YA & 0 & eX^3Y^2 & 0 & 0 & 0 & 0 & 0 \\
1 & 3XA & 3X^2A^2 & X^3A^3 & -3XY & -6X^2YA & -3X^3YA^2 & 3X^2Y^2 & 3X^3Y^2A & -X^3Y^3 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & e^3Y & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & e^2XYA & 0 & 0 & 0 & 0 & 0 & e^2Y & -e^2Y^2X & 0 & 0 \\
0 & 0 & 0 & 0 & 2eXYA & eX^2YA^2 & 0 & -2eX^2Y^2A & 0 & 0 & eY & -2eY^2X & eY^3X^2 & 0 \\
0 & 0 & 0 & 0 & 3XYA & 3X^2YA^2 & X^3YA^3 & -6X^2Y^2A & -3X^3Y^2A^2 & 3X^3Y^3A & Y & -3Y^2X & 3Y^3X^2 & -Y^4X^3
\end{array}$$

Die vorgestellten Überlegungen führen jetzt zu folgendem Verfahren:

Boneh-Durfee-Angriff: Gegeben sei ein öffentlicher RSA-Schlüssel (N,e).

1. Wähle Parameter $m \geq 1$, $t \geq 0$ und Zahlen X, Y, so daß $k \lessapprox X$, $s \lessapprox Y$ zu erwarten ist, z.B. $X = 1.01e^{\delta}$, $Y = 6.71e^{0.5}$ mit einem Parameter δ.
2. Erstelle die Matrix $M(N,e,m,t,X,Y)$. LLL-Reduktion liefert Polynome $h_1(Xx,Yy)$, $h_2(Xx,Yy)$, $h_3(Xx,Yy)$, $\ldots$, die den Elementen der reduzierten Gitterbasis entsprechen.
3. Bilde nun Resultanten

   $$r_{ij}(y) = \mathrm{Resultante}_x(h_i(x,y), h_j(x,y)),$$

   Bestimme im Fall $r_{ij} \neq 0$ die ganzzahligen Nullstellen $\widetilde{y}$ von $r_{ij}(y)$ und teste, ob solche mit der Annahme $\widetilde{y} = p+q$ zu einer Faktorisierung von $N = pq$ führen.

Bemerkung: Wir haben zu dem Verfahren eine Maple-Funktion 'bodu_angriff'(N, e, m, t, δ) geschrieben, wobei zur Gitterreduktion das NTL-Programm 'lll_ntl.c' benutzt wird. Dabei wird in Abhängigkeit von N, e und δ

$$Y = \lfloor \frac{3}{\sqrt{2}} \sqrt{N} \rfloor \quad \text{und} \quad X = \lfloor \frac{eN^{\delta}}{N - \lfloor \frac{3}{\sqrt{2}} \sqrt{N} \rfloor} \rfloor$$

gesetzt.

Bevor wir weitere theoretische Überlegungen anstellen, sollen Beispiele das Funktionieren des Boneh-Durfee-Angriffs illustrieren.

Beispiele: Wir betrachten 30 (zufällig gewählte) Beispiele $(N_i, p_i, q_i, d_i, e_i)$, bei denen N_i jeweils 256 Bits hat und $0.25 < \ln(d_i)/\ln(N_i) < 0.30$ gilt. Die Zahlenwerte der Beispiele sind im Anhang zu finden.
Hier sind zunächst ein paar numerische Daten der Beispiele zusammengestellt:

i	$N_i/2^{256}$	q_i/p_i	$s_i/\lfloor\sqrt{N_i}\rfloor$	$\ln d_i/\ln N_i$	e_i/N_i	$\ln k_i/\ln e_i$	$\ln s_i/\ln e_i$
1	.530153	1.729949	2.075572	.252818	.823092	.251994	.504686
2	.522488	1.417601	2.030522	.251921	.185530	.244724	.508855
3	.778248	1.786816	2.084819	.251269	.787919	.250261	.504825
4	.617248	1.400991	2.028490	.255815	.350253	.251377	.507003
5	.722782	1.478429	2.038338	.257428	.514503	.254631	.505919
6	.537527	1.861969	2.097388	.258151	.153262	.250198	.509594
7	.742815	1.619201	2.058346	.263108	.728729	.261790	.504977
8	.905896	1.841912	2.093998	.264969	.187240	.257959	.508976
9	.595634	1.813758	2.089282	.260943	.982418	.260869	.504215
10	.663110	1.556111	2.049082	.268693	.988090	.268643	.504086
11	.743670	1.687228	2.068796	.265160	.521323	.262448	.505964
12	.661991	1.713817	2.072995	.267256	.018099	.250265	.515807
13	.599903	1.209125	2.009022	.271834	.732372	.270550	.504832
14	.679841	1.730668	2.075688	.273807	.059226	.262027	.512302
15	.582417	1.887987	2.101821	.272090	.345592	.267691	.507245
16	.713130	1.671251	2.066302	.277389	.202617	.270816	.508683
17	.712670	1.438494	2.033142	.278961	.815222	.278128	.504589
18	.510519	1.573366	2.051572	.275722	.822736	.274921	.504622
19	.577604	1.595391	2.054798	.283050	.624651	.281138	.505416
20	.709465	1.565934	2.050495	.280970	.810860	.280118	.504652
21	.634932	1.838277	2.093386	.280227	.086772	.270147	.511235
22	.588707	1.209998	2.009091	.289298	.918366	.288956	.504186
23	.717132	1.831631	2.092270	.287210	.149088	.279468	.509645
24	.751773	1.877935	2.100103	.287921	.036061	.274311	.513825
25	.568258	1.566611	2.050593	.292965	.174629	.285920	.509083
26	.774167	1.556143	2.049087	.290905	.974658	.290803	.504122
27	.757951	1.554997	2.048923	.291712	.379873	.287822	.506818
28	.594897	1.414206	2.030103	.299473	.105303	.290446	.510497
29	.864552	1.794047	2.086012	.298002	.493872	.295198	.506161
30	.538079	1.252035	2.012644	.295057	.459580	.291944	.506181

Zur nächsten Tabelle:

- Ein * in der Spalte 'Wiener' gibt an, ob die Wiener-Attacke mit 'kb_rsa' erfolgreich war.
- Mit der Funktion 'bodu_angriff' wurden mit den Parametern $(m,t) = (3,1)$ und $(m,t) = (5,2)$ bei verschiedener Wahl des Parameters δ Angriffe durchgeführt, $\delta = 0.24, ..., 0.30$. Bei Erfolg steht 3 bzw. 5 an der entsprechenden Stelle.
- Für $(m,t) = (3,1)$ dauerte die LLL-Reduktion jedesmal $\approx$ 15 sec (Gitterrang: 14), die Resultantenbildungen mit Maple (bis) $\approx$ 25 sec. Im Fall $(m,t) = (5,2)$ lag die Rechenzeit für die LLL-Reduktion zwischen 17 und 40 Minuten (Gitterrang: 33). Die Resultantenbildung kann recht zeitaufwendig sein.

i	$\ln(d_i)/\ln(N_i)$	Wiener	0.24	0.25	0.26	0.27	0.28	0.29	0.30
1	0.2528	*	3	3,5	3,5	3,5	3,5	3,5	
2	0.2519	*	3	3,5	3,5	3,5	3,5	3,5	3,5
3	0.2513	*	3	3,5	3,5	3,5	3,5	3,5	
4	0.2558	*	3	3,5	3,5	3,5	3,5	3,5	5
5	0.2574	*	3	3,5	3,5	3,5	3,5	5	
6	0.2582	*	3	3,5	3,5	3,5	3,5	5	5
7	0.2631			5	5	3,5	5		
8	0.2650			5	3,5	5	5		
9	0.2609			5	5	5			
10	0.2687					5			
11	0.2652			5	5	5			
12	0.2673	*	3	3,5	3,5	3,5	3,5	5	
13	0.2718								
14	0.2738					5			
15	0.2721					5			
16	0.2774								
17	0.2790								
18	0.2757								
19	0.2831								
20	0.2810								
21	0.2802								
22	0.2893								
23	0.2872								
24	0.2879								
25	0.2930								
26	0.2909								
27	0.2917								
28	0.2995								
29	0.2980								
30	0.2951								

Zur folgenden Tabelle:

- Hier wurde immer der Paramter $\delta = 0.27$ gewählt und (m,t) variiert.
- Rechenzeiten für die LLL-Reduktion mit NTL: $(m,t) = (3,1)$: zwischen 12 und 18 Sekunden (Gitterrang: 14); $(m,t) = (4,1)$: zwischen 2:49 und 3:25 Minuten (Gitterrang: 20); $(m,t) = (5,1)$: zwischen 18:17 und 23:04 Minuten (Gitterrang: 27); $(m,t) = (5,2)$: zwischen 29:35 und 34:57 Minuten (Gitterrang: 33).

i	$\ln(d_i)/\ln(N_i)$	Wiener	(3,1)	(4,1)	(5,1)	(5,2)
1	0.2528	*	*	*	*	*
2	0.2519	*	*	*	*	*
3	0.2513	*	*	*	*	*
4	0.2558	*	*	*	*	*
5	0.2574	*	*	*	*	*
6	0.2582	*	*	*	*	*
7	0.2631		*	*	*	*
8	0.2650			*	*	*
9	0.2609			*	*	*
10	0.2687					*
11	0.2652			*	*	*
12	0.2673	*	*	*	*	*
13	0.2718					
14	0.2738			*	*	*
15	0.2721					*
16	0.2774					
17	0.2790					
18	0.2757					
19	0.2831					
20	0.2810					
21	0.2802					
22	0.2893					
23	0.2872					
24	0.2879					
25	0.2930					
26	0.2909					
27	0.2917					
28	0.2995					
29	0.2980					
30	0.2951					

Schließlich findet sich in der folgenden Tabelle die Anzahl aller Polynome h_{ij}, die wir in den Fällen $(m,t,\delta) = (3,1,0.27), (4,1,0.27), (5,1,0.27), (5,2,0.27)$ erhalten haben mit $h_{ij}(k_i, s_i) = 0$.

i	Anzahl	i	Anzahl	i	Anzahl
1	61	11	37	21	0
2	61	12	59	22	0
3	61	13	2	23	0
4	61	14	8	24	0
5	61	15	4	25	0
6	61	16	0	26	0
7	46	17	0	27	0
8	55	18	0	28	0
9	48	19	0	29	0
10	3	20	0	30	0

Wir wenden dies auf das bisher offengebliebene 13. Beispiel an: Nimmt man das 8. Polynom aus dem Fall $(m,t,\delta) = (5,1,0.27)$ und das 19. Polynom aus dem Fall $(m,t,\delta) = (5,2,0.27)$, so erhält man die Faktorisierung von N_{13}.

Um einen Eindruck von der Größenordnung der auftretenden Polynome zu bekommen, geben wir hier für das 15. Beispiel und $(m,t,\delta) = (5,2,0.27)$ die Polynome h_5, h_{10} und $r(y) = \text{Resultante}_x(h_5(x,y), h_{10}(x,y))$ explizit wieder:

```
                    302                   300  4                 321  3
h_5 := .5337941381 10    x + .6642166791 10    x  + .2487137350 10    x

                    165                 8  2                 280  5
     - .2666916346 10    y - .99820538 10  y  - .6952193685 10    x

                    340  2                 264                 282  3
     + .1564628703 10    x  + .1736720426 10    + .1402433640 10    x  y

                    222  4  2                 243  3  2
     - .6949235655 10    x  y  - .1335773709 10    x  y

                    165    2                 87  5  5
     + .8000749037 10    x y  + .2693330774 10   x  y

                    203  5  2                 126  5  4
     + .3170413671 10    x  y  - .1457468899 10    x  y

                    164  5  3                 146  4  4
     - .4968817861 10    x  y  + .1364311409 10    x  y

                    205  3  3                 107  4  5                9  3  5
     - .1120937604 10    x  y  + .3549701754 10    x  y  + .998205380 10  x  y

                    165  3  4                 9  2  4                 165  2  3
     + .2666916346 10    x  y  - .998205380 10  x  y  - .8000749037 10    x  y

                   9    3                 48  5  6                9  4  6
     + .499102690 10  x y  - .2899907874 10   x  y  - .499102690 10  x  y

                  8  5  7                 264                     302  2
     + .99820538 10  x  y  - .3473440851 10    x y - .5337941381 10    x  y

                    262  4                   241  5
     - .1864810053 10    x  y + .3765727134 10    x  y

                    264  2  2                 184  4  3
     + .1736720426 10    x  y  - .2871419261 10    x  y

                     302                   300  4                 320  3
h_10 := .4883852381 10    x - .6233244196 10    x  - .1783087329 10    x

                    165                   9  2                 280  5
     + .7109701225 10    y - .450429205 10  y  + .3971695757 10    x

                    340  2                 264                 282  3
     - .5988799235 10    x  - .1123908243 10    - .2086127611 10    x  y

                    223  4  2                 244  3  2
     - .1686364308 10    x  y  + .1926850003 10    x  y

                    166    2                 86  5  5
     - .2132910367 10    x y  + .7095631190 10   x  y
```

```
                  202  5  2                125  5  4
     + .8206397112 10    x  y  - .9957477611 10    x  y

                  164  5  3                146  4  4
     - .1610987991 10    x  y  - .1528155520 10    x  y

                  204  3  3                107  4  5
     - .2068441875 10    x  y  - .7929938976 10    x  y

                  10  3  5                 165  3  4
     + .4504292050 10   x  y  - .7109701225 10    x  y

                  10  2  4                 166  2  3                  10    3
     - .4504292050 10   x  y  + .2132910367 10    x  y  + .2252146025 10   x y

                  48  5  6                 10  4  6                9  5  7
     + .3579737584 10   x  y  - .2252146025 10   x  y  + .450429205 10  x  y

                  264                     302  2                   262  4
     + .2247816487 10    x y - .4883852381 10    x  y + .2100077013 10    x  y

                  241  5                   264  2  2
     - .8958577814 10    x  y - .1123908243 10    x  y

                  185  4  3
     + .1330994745 10    x  y

                 1910                    39                  928  5
r  := .1537855993 10     (y - .5458236588 10  ) (-.8107753091 10    y

                  851  7                 73  27                503  16
     - .3889211275 10    y  + .1439355849 10   y   - .1108194826 10    y

                  1081                   1043  2                967  4
     - .1069740259 10     y - .1648179509 10     y  + .2177792872 10    y

                  426  18                1004  3                192  24
     + .4113639709 10    y   - .6033520664 10     y  + .6140632499 10    y

                  464  17                231  23                697  11
     + .2081204748 10    y   - .3257605190 10    y   - .4466306506 10    y

                  734  10                114  26                774  9
     + .4795386582 10    y   - .5510318466 10    y   + .5126378824 10    y

                  812  8                 154  25                387  19
     - .2907796784 10    y  - .1434024450 10    y   - .5846884079 10    y

                  349  20                581  14                271  22
     - .1005042247 10    y   - .1690713387 10    y   + .1022710092 10    y

                  542  15                658  12                620  13
     - .4017607163 10    y   + .2483313473 10    y   + .1857007789 10    y
```

```
                  310  21                  1119                  890  6
+ .1322298973 10    y   + .4575901979 10     + .1475065426 10    y )
```

Abschließend bemerken wir zu den Beispielen noch folgendes:

1. Bei allen Beispielen mit $\log d_i / \log N_i < 0.275$ war der Boneh-Durfee-Angriff mit den angegebenen Parametern erfolgreich.
2. Beim 18. Beispiel mit $\log d_{18}/\log N_{18} = 0.2757$ lieferte auch der Angriff mit $(m,t,\delta) = (7,3,0.27)$ (Gitterrang: 60) nach 16:18:45 Stunden kein Polynom $h(x,y)$ mit $h(k_{18}, s_{18}) = 0$.

Wir geben jetzt einen Satz an, der zeigt, warum der Boneh-Durfee-Angriff im Fall

$$\frac{\ln d}{\ln N} < \frac{7}{6} - \frac{1}{3}\sqrt{7} \approx 0.284$$

bei geeigneter Parameterwahl erfolgversprechend ist.

SATZ. 1. *Für $m \geq 1$, $t \geq 0$ sei*

$$\Delta(m,t) = \frac{m^3 + 3m^2t + 3m^2 - 3mt^2 - 10m - 3t^2 - 3t}{2m(m+1)(2m+3t+4)}.$$

Für

$$\begin{aligned} t_m &= \frac{\sqrt{7m^4 + 30m^3 + 79m^2 + 60m + 4} - 2m^2 - 6m - 4}{3m+3} = \\ &= \frac{\sqrt{7}-2}{3}m - \frac{28 - 8\sqrt{7}}{21} + \frac{36\sqrt{7}}{49}\frac{1}{m} + O(\frac{1}{m^2}) = \\ &= 0.2153\,m - 0.3254 + \frac{1.9438}{m} + O(\frac{1}{m^2}) \end{aligned}$$

gilt

$$\Delta(m,t_m) = \frac{7-2\sqrt{7}}{6} - \frac{16\sqrt{7}-35}{42m} + O(\frac{1}{m^2}) \approx 0.2847 - \frac{0.1746}{m} + O(\frac{1}{m^2})$$

und

$$\Delta(m,t) \leq \Delta(m,t_m) < \frac{7-2\sqrt{7}}{6} \quad \textit{und} \quad \sup_{m\geq 1, t\geq 0} \Delta(m,t) = \frac{7-2\sqrt{7}}{6}.$$

2. *Zu m,t,δ mit $0 < \delta < \Delta(m,t)$ gibt es ein $e_0(m,t,\delta)$, so daß nachfolgende Eigenschaft erfüllt ist.*
3. *Sei (N,e) ein öffentlicher RSA-Schlüssel, (N,d) der zugehörige private Schlüssel und $N = pq$, $1 < e,d < \varphi(N)$ mit $p < q < 2p$, $e > \frac{1}{10}N$, sowie $k = \frac{ed-1}{\varphi(N)}$, $s = p+q$. Seien m,t,δ gewählt mit $m \geq 1$, $t \geq 0$, $0 < \delta < \Delta(m,t)$ und $X = 1.01e^{\delta}$, $Y = 6.71e^{0.5}$. Seien $h_1(Xx,Yy), h_2(Xx,Yy)$ die beiden ersten Vektoren einer LLL-reduzierten Basis des Gitters $\Lambda(N,e,m,t,X,Y)$. Gilt jetzt*

$$d \leq N^{\delta} \quad \textit{und} \quad e \geq e_0(m,t,\delta),$$

so folgt

$$h_1(k,s) = h_2(k,s) = 0 \quad \textit{und} \quad \text{Resultante}_x(h_1(x,y), h_2(x,y))(s) = 0.$$

Beweis: Schreiben wir $\Lambda = \Lambda(N,e,m,t,X,Y)$ und

$$\begin{aligned} w = w(m,t) &= \frac{1}{2}(m+1)(m+2t+2), \\ A = A(m,t) &= \frac{1}{6}m(m+1)(2m+3t+4), \\ B = B(m,t) &= \frac{1}{6}(m+1)(m^2+2m+3tm+3t^2+3t), \end{aligned}$$

so gilt

$$\dim \Lambda = w, \quad \det \Lambda = (eX)^A \cdot Y^B.$$

Weiter ist nach Voraussetzung

$$\begin{aligned} X &= 1.01e^{\delta} = e^{\delta+\log 1.01/\log e}, \\ Y &= 6.71e^{\frac{1}{2}} = e^{\frac{1}{2}+\log 6.71/\log e}, \\ \det\Lambda &= (eX)^A \cdot Y^B = e^{(1+\delta+\log 1.01/\log e)A+(\frac{1}{2}+\log 6.71/\log e)B}. \end{aligned}$$

$\Delta(m,t)$ wurde genau so definiert, daß gilt

$$\frac{1}{w-1}\left((1+\Delta(m,t))A+\frac{1}{2}B\right) = m.$$

Ist nun $0 < \delta < \Delta(m,t)$, so folgt

$$\frac{1}{w-1}\left((1+\delta)A+\frac{1}{2}B\right) < m.$$

Dann ist klar, daß es ein $e_0(m,t,\delta)$ gibt, so daß für alle $e \geq e_0(m,t,\delta)$ die Ungleichung

$$\frac{1}{\log e}\left(\frac{1}{4}w\log 2+\frac{A\log 1.01+B\log 6.71}{w-1}+\frac{1}{2}\log w\right)+\frac{1}{w-1}\left((1+\delta)A+\frac{1}{2}B\right) < m$$

gilt. (Natürlich kann man $e_0(m,t,\delta)$ dann explizit als Funktion von m,t,δ angeben.)
Nun gelten die Äquivalenzen:

$$\begin{aligned} & 2^{w/4}(\det\Lambda)^{1/(w-1)} < e^m/\sqrt{w} \\ \iff\quad & \frac{w}{4}\frac{\log 2}{\log e}+\frac{1}{w-1}\frac{\log\det\Lambda}{\log e} < m-\frac{\log w}{2\log e} \\ \iff\quad & \frac{w}{4}\frac{\log 2}{\log e}+\frac{1}{w-1}\left((1+\delta+\log 1.01/\log e)A+(\frac{1}{2}+\log 6.71/\log e)B\right) < m-\frac{\log w}{2\log e} \\ \iff\quad & \frac{1}{\log e}\left(\frac{1}{4}w\log 2+\frac{A\log 1.01+B\log 6.71}{w-1}+\frac{1}{2}\log w\right)+\frac{1}{w-1}\left((1+\delta)A+\frac{1}{2}B\right) < m. \end{aligned}$$

Sind $h_1(Xx,Yy), h_2(Xx,Yy)$ die ersten beiden Vektoren einer LLL-reduzierten Basis des Gitter Λ, so gelten die allgemeinen Abschätzungen

$$\|h_1(Xx,Yy)\| \leq 2^{(w-1)/4}(\det\Lambda)^{1/w}, \quad \|h_2(Xx,Yy)\| \leq 2^{w/4}(\det\Lambda)^{1/(w-1)}.$$

Mit obigen Äquivalenzen und unseren Voraussetzungen folgt

$$\|h_1(Xx,Yy)\| < e^m/\sqrt{w}, \quad \|h_2(Xx,Yy)\| < e^m/\sqrt{w}.$$

Nach Konstruktion von $\Lambda(N,e,m,t,X,Y)$ gilt mit $N=pq$, $s=p+q$, $k=\frac{ed-1}{\varphi(N)}$

$$h_1(k,s) \equiv 0 \bmod e^m, \quad h_2(k,s) \equiv 0 \bmod e^m$$

und mit unseren früheren Abschätzungen und den Voraussetzungen $p<q<2p$, $N>100000$, $e>\frac{1}{10}N$, $d \leq N^{\delta}$

$$|k| \leq 1.01e^{\delta}, \quad |s| \leq 6.71e^{1/2}.$$

Das Lemma von Howgrave-Graham liefert nun

$$h_1(k,s) = h_2(k,s) = 0,$$

was sofort auch $\text{Resultante}_x(h_1(x,y),h_2(x,y))(s) = 0$ liefert.
Wir betrachten jetzt die Funktion $\Delta(m,t)$. Es ist

$$\frac{\partial\Delta}{\partial t} = \frac{3}{2}\frac{(-3m-3)t^2+(-4m^2-12m-8)t+(m^3+m^2+8m-4)}{m(m+1)(2m+3t+4)^2}.$$

Bei festem m ist der Zähler von $\frac{\partial\Delta}{\partial t}$ eine nach unten geöffnete Parabel in t. Es ist für $m \geq 1$

$$\frac{\partial\Delta}{\partial t}(t=0) = \frac{3}{2}\frac{m^3+m^2+8m-4}{m(m+1)(2m+4)^2} > 0.$$

Die positive Nullstelle des Zählers von $\frac{\partial\Delta}{\partial t}$ ist

$$t_m = \frac{\sqrt{7m^4+30m^3+79m^2+60m+4}-2m^2-6m-4}{3m+3}.$$

Also nimmt $\Delta(m,t)$ bei festem $m \geq 1$ sein Maximum für $t = t_m$ an. Insbesondere folgt für $m \geq 1$, $t \geq 0$ sofort

$$\Delta(m,t) \leq \Delta(m,t_m).$$

Funktionsdiskussion von $m \mapsto \Delta(m,t_m)$ zeigt, daß diese Funktion für $m \geq 1$ streng monoton steigend ist mit

$$\lim_{m\to\infty} \Delta(m,t_m) = \frac{7-2\sqrt{7}}{6} \approx 0.2847.$$

Insbesondere folgt

$$\Delta(m,t) \leq \Delta(m,t_m) < \frac{7-2\sqrt{7}}{6},$$

womit alle Behauptungen gezeigt sind. ■

Beispiel: Wir stellen noch einige explizite Zahlenwerte für die im Satz verwendeten Größen zusammen:

m	t_m	$\Delta(m,\lfloor t_m \rfloor)$	$\Delta(m,t_m)$	$\Delta(m,\lceil t_m \rceil)$	$w(m,\lfloor t_m \rfloor)$	$w(m,\lceil t_m \rceil)$
1	.2361	$\Delta(1,0) = -.2500$	$-.2361$	$\Delta(1,1) = -.3333$	$w(1,0) = 3$	$w(1,1) = 5$
2	.4603	$\Delta(2,0) = 0.0000$	.0199	$\Delta(2,1) = 0.0000$	$w(2,0) = 6$	$w(2,1) = 9$
3	.6388	$\Delta(3,0) = .1000$	.1204	$\Delta(3,1) = .1154$	$w(3,0) = 10$	$w(3,1) = 14$
4	.8166	$\Delta(4,0) = .1500$	.1708	$\Delta(4,1) = .1700$	$w(4,0) = 15$	$w(4,1) = 20$
5	1.0000	$\Delta(5,1) = .2000$	.2000	$\Delta(5,1) = .2000$	$w(5,1) = 27$	$w(5,1) = 27$
6	1.1889	$\Delta(6,1) = .2180$	.2185	$\Delta(6,2) = .2110$	$w(6,1) = 35$	$w(6,2) = 42$
7	1.3824	$\Delta(7,1) = .2296$	.2311	$\Delta(7,2) = .2277$	$w(7,1) = 44$	$w(7,2) = 52$
8	1.5795	$\Delta(8,1) = .2373$	.2401	$\Delta(8,2) = .2388$	$w(8,1) = 54$	$w(8,2) = 63$
9	1.7795	$\Delta(9,1) = .2427$	.2467	$\Delta(9,2) = .2464$	$w(9,1) = 65$	$w(9,2) = 75$
10	1.9818	$\Delta(10,1) = .2465$	.2518	$\Delta(10,2) = .2518$	$w(10,1) = 77$	$w(10,2) = 88$
11	2.1859	$\Delta(11,2) = .2557$	.2558	$\Delta(11,3) = .2532$	$w(11,2) = 102$	$w(11,3) = 114$
12	2.3914	$\Delta(12,2) = .2585$	.2590	$\Delta(12,3) = .2578$	$w(12,2) = 117$	$w(12,3) = 130$
13	2.5982	$\Delta(13,2) = .2605$	.2617	$\Delta(13,3) = .2612$	$w(13,2) = 133$	$w(13,3) = 147$
14	2.8059	$\Delta(14,2) = .2620$	.2639	$\Delta(14,3) = .2638$	$w(14,2) = 150$	$w(14,3) = 165$
15	3.0145	$\Delta(15,3) = .2657$	.2657	$\Delta(15,4) = .2636$	$w(15,3) = 184$	$w(15,4) = 200$
16	3.2238	$\Delta(16,3) = .2672$	.2673	$\Delta(16,4) = .2661$	$w(16,3) = 204$	$w(16,4) = 221$
17	3.4337	$\Delta(17,3) = .2683$	.2686	$\Delta(17,4) = .2680$	$w(17,3) = 225$	$w(17,4) = 243$
18	3.6441	$\Delta(18,3) = .2691$	.2698	$\Delta(18,4) = .2696$	$w(18,3) = 247$	$w(18,4) = 266$
19	3.8550	$\Delta(19,3) = .2697$	.2708	$\Delta(19,4) = .2708$	$w(19,3) = 270$	$w(19,4) = 290$
20	4.0662	$\Delta(20,4) = .2717$	.2717	$\Delta(20,5) = .2706$	$w(20,4) = 315$	$w(20,5) = 336$
30	6.1921	$\Delta(30,6) = .2769$	.2769	$\Delta(30,7) = .2765$	$w(30,6) = 682$	$w(30,7) = 713$
40	8.3305	$\Delta(40,8) = .2792$	.2792	$\Delta(40,9) = .2791$	$w(40,8) = 1189$	$w(40,9) = 1230$
50	10.4742	$\Delta(50,10) = .2805$	.2805	$\Delta(50,11) = .2805$	$w(50,10) = 1836$	$w(50,11) = 1887$
60	12.6208	$\Delta(60,12) = .2813$	.2813	$\Delta(60,13) = .2813$	$w(60,12) = 2623$	$w(60,13) = 2684$
70	14.7690	$\Delta(70,14) = .2818$	.2819	$\Delta(70,15) = .2819$	$w(70,14) = 3550$	$w(70,15) = 3621$
80	16.9182	$\Delta(80,16) = .2822$	.2823	$\Delta(80,17) = .2823$	$w(80,16) = 4617$	$w(80,17) = 4698$
90	19.0682	$\Delta(90,19) = .2826$	.2826	$\Delta(90,20) = .2825$	$w(90,19) = 5915$	$w(90,20) = 6006$
100	21.2186	$\Delta(100,21) = .2828$	.2828	$\Delta(100,22) = .2828$	$w(100,21) = 7272$	$w(100,22) = 7373$

Bemerkungen:

1. Ein Schwachpunkt des Satzes ist, daß man nicht garantieren kann, daß die Resultante

   $$\text{Resultante}_x(h_1(x,y), h_2(x,y))$$

   von $h_1(x,y)$ und $h_2(x,y)$ von Null verschieden ist, d.h. es könnte passieren, daß $h_1(x,y)$ und $h_2(x,y)$ einen gemeinsamen Faktor haben. In diesem Fall kann man natürlich $s = p + q$ nicht aus $h_1(x,y)$ und $h_2(x,y)$ bestimmen.
2. Bei unseren Beispielen funktionierte das Verfahren deutlich besser als die obige Tabelle erwarten läßt.

3. Mit einem etwas modifizierten Gitter stellen Boneh und Durfee Überlegungen an und skizzieren, daß der entsprechende Angriff für

$$\delta < 1 - \frac{1}{2}\sqrt{2} \approx 0.292$$

erfolgversprechend ist.

Literatur: [**Wi**], [**BoDu**].

ANHANG A

Programme zur Vorlesung

1. Gitter und Gitterbasenreduktion

1.1. gitter_ma.

```
# gitter_ma
# Kapitel: Gitter und Gitterbasenreduktion
# Version: 16. Juli 2002
# Funktionen:
# v_mult
# v_add
# v_smult
# v_laenge
# gso
# mu
# gitter_det2
# gitter_det2_folge
# gitter_hoehe
# normalisierung
# cii1
# tausch
# komb
# lll
# zufallsmatrix
# gitter_ma2ntl
# zutra
# einh
# zeitausgabe
# rz_lll_ntl
# rz_lll_ma
# rz_lll
#
# Vektoren werden als Listen dargestellt

# Skalarprodukt zweier Vektoren
v_mult:=proc()
  local a, b, s, i;
  a:=args[1]; b:=args[2];
  s:=0;
  for i from 1 to nops(a) do
    s:=s+a[i]*b[i];
  od;
  s;
end;

# Addition zweier Vektoren
v_add:=proc()
```

```
  local a, b, c, i;
  a:=args[1]; b:=args[2]; c:=[];
  for i from 1 to nops(a) do
    c:=[op(c),a[i]+b[i]];
  od;
  c;
end;

# Multiplikation eines Skalars mit einem Vektor
v_smult:=proc()
  local k, a, b, i;
  k:=args[1]; a:=args[2];
  b:=[];
  for i from 1 to nops(a) do
    b:=[op(b),k*a[i]];
  od;
  b;
end;

# Laenge eines Vektors
v_laenge:=proc()
  local a;
  a:=args[1];
  sqrt(v_mult(a,a));
end;

# Gram-Schmidt-Orthogonalisierung
gso:=proc()
  local b, m, n, i, j, bb, bbi;
  b:=args[1];
  m:=nops(b); n:=nops(b[1]);
  bb:=[];
  for i from 1 to m do
    bbi:=b[i];
    for j from 1 to i-1 do
      bbi:=v_add(bbi,v_smult(-v_mult(b[i],bb[j])/v_mult(bb[j],bb[j]),bb[j]));
    od;
  bb:=[op(bb),bbi];
  od;
  bb;
end;

# mu_ij-Werte eines Gitters
mu:=proc()
  local b, m, n, i, j, bb, bbi, mm, mmm;
  b:=args[1];
  m:=nops(b); n:=nops(b[1]);
  bb:=gso(b);
  mm:=[];
  for i from 2 to m do
    mmm:=[];
    for j from 1 to i-1 do
      mmm:=[op(mmm),v_mult(b[i],bb[j])/v_mult(bb[j],bb[j])];
    od;
```

```
    mm:=[op(mm),mmm];
  od;
  mm;
end;

# Determinantenquadrat
gitter_det2:=proc()
  local b, m, n, M, i, j;
  b:=args[1];
  m:=nops(b); n:=nops(b[1]);
  M:=array(1..m,1..m);
  for i from 1 to m do
    for j from 1 to m do
      M[i,j]:=v_mult(b[i],b[j]);
    od;
  od;
  linalg[det](M);
end;

# Folge der Determinantenquadrate der Untergitter L_i=Z b_1+...+Z b_i
gitter_det2_folge:=proc()
  local b, m, bi, d, i;
  b:=args[1]; m:=nops(b); bi:=[]; d:=[];
  for i from 1 to m do
    bi:=[op(bi),b[i]];
    d:=[op(d),gitter_det2(bi)];
  od;
  d;
end;

# Maximum der Absolutbetraege der Eintraege einer Gitterbasis
gitter_hoehe:=proc()
  local b, m, n, H, i, j;
  b:=args[1];
  m:=nops(b); n:=nops(b[1]);
  H:=0;
  for i from 1 to m do
    for j from 1 to n do
      H:=max(H,abs(b[i][j]));
    od;
  od;
  H;
end;

# Basiswechsel, so dass |mu_ij|<=0.5 gilt
normalisierung:=proc()
  local b, m, n, i, j, bb, bbi, mui, bi, muij, k;
  b:=args[1];
  m:=nops(b); n:=nops(b[1]);
  bb:=[b[1]];
  for i from 2 to m do
    mui:=[]; bi:=b[i]; bbi:=[seq(0,i=1..n)];
    for j from i-1 to 1 by -1 do
      muij:=v_mult(bi,bb[j])/v_mult(bb[j],bb[j]);
```

```
      k:=floor(muij+1/2);
      if k<>0 then
        bi:=v_add(bi,v_smult(-k,b[j])); muij:=muij-k;
      fi;
      bbi:=v_add(bbi,v_smult(-muij,bb[j]));
      mui:=[muij,op(mui)];
    od;
    b:=subsop(i=bi,b);
    bbi:=v_add(bi,bbi);
    bb:=[op(bb),bbi];
  od;
  b;
end;

# Werte fuer c_i,i-1
cii1:=proc()
  local b, m, bb, c, i, muii1, cii1;
  b:=args[1];
  m:=nops(b);
  bb:=gso(b);
  c:=[];
  for i from 2 to m do
    muii1:=v_mult(b[i],bb[i-1])/v_mult(bb[i-1],bb[i-1]);
    cii1:=v_mult(bb[i],bb[i])/v_mult(bb[i-1],bb[i-1])+muii1^2;
    c:=[op(c),cii1];
  od;
  c;
end;

# Vertauschen von Basisvektoren
tausch:=proc()
  local i1, i2, b, bi1, bi2;
  i1:=args[1]; i2:=args[2]; b:=args[3];
  bi1:=b[i1]; bi2:=b[i2];
  subsop(i1=bi2,i2=bi1,b);
end;

# Addiere zu b_i1 das c-fache von b_i2: komb(i1,c,i2,b)
komb:=proc()
  local i1, i2, c, b, bi1n;
  i1:=args[1]; c:=args[2]; i2:=args[3]; b:=args[4];
  bi1n:=v_add(b[i1],v_smult(c,b[i2]));
  subsop(i1=bi1n,b);
end;

# LLL-Reduktion bei gegebener Gitterbasis
lll:=proc()
  local b, m, n, i, j, bb, bbi, az, c, bi, muij, k, muii1, ci;
  b:=args[1];
  if nargs<2 then c:=3/4; else c:=args[2]; fi;
  m:=nops(b); n:=nops(b[1]);
  i:=1;
  while i<=m do
    if i=1 then
```

```
      bb:=[b[1]];
      i:=2;
    fi;
    bi:=b[i]; bbi:=[seq(0,i=1..n)];
    for j from i-1 to 1 by -1 do
      muij:=v_mult(bi,bb[j])/v_mult(bb[j],bb[j]);
      k:=floor(muij+1/2);
      if k<>0 then
        bi:=v_add(bi,v_smult(-k,b[j])); muij:=muij-k;
      fi;
      bbi:=v_add(bbi,v_smult(-muij,bb[j]));
      if j=i-1 then muii1:=muij; fi;
    od;
    b:=subsop(i=bi,b);
    bbi:=v_add(bi,bbi);
    bb:=[op(bb),bbi];
    ci:=v_mult(bb[i],bb[i])/v_mult(bb[i-1],bb[i-1])+muii1^2;
    if ci<c then
      az[i-1]:=az[i-1]+1;
      b:=tausch(i,i-1,b);
      bb:=subsop(i=NULL,bb); bb:=subsop(i-1=NULL,bb);
      i:=i-1;
    else
      i:=i+1;
    fi;
  od;
  b;
end;

# Erzeugung einer m x n - Zufallsmatrix mit |Eintraege|<=M
zufallsmatrix:=proc()
  local m, n, M, zz, a, i, aa, j;
  m:=args[1]; n:=args[2]; M:=args[3];
  zz:=rand(-M..M);
  a:=[];
  for i from 1 to m do
    aa:=[];
    for j from 1 to n do
      aa:=[op(aa),zz()];
    od;
    a:=[op(a),aa];
  od;
  a;
end;

# Ein Gitter, gegeben durch eine Gitterbasis, wird in NTL-Format in eine
# Datei geschrieben
gitter_ma2ntl:=proc()
  local M, aus, m, n, i, j;
  M:=args[1]; aus:=args[2];
  m:=nops(M); n:=nops(M[1]);
  # Die Matrix M wird in NTL-Format in 'aus' geschrieben.
  fprintf(aus,"[\n");
  for i from 1 to m do
```

```
    fprintf(aus,"[");
    for j from 1 to n do
      fprintf(aus,"%d ",M[i][j]);
    od;
    fprintf(aus,"]\n");
  od;
  fprintf(aus,"]\n");
  fclose(aus);
end;

# Zufallstransformationen: zutra(Anzahl,Hoehe,b)
zutra:=proc()
  local az, H, b, zz, zzh, i1, i2, c;
  az:=args[1]; H:=args[2]; b:=args[3];
  zz:=rand(1..nops(b));
  zzh:=rand(-H..H);
  while az>0 do
    i1:=zz(); i2:=zz(); while i1=i2 do i2:=zz(); od;
    printf("i1=%d i2=%d\n",i1,i2);
    tausch(i1,i2,b);
    i1:=zz(); i2:=zz(); while i1=i2 do i2:=zz(); od;
    c:=zzh();
    b:=komb(i1,c,i2,b);
    az:=az-1;
  od;
  b;
end;

# n x n - Einheitsmatrix
einh:=proc()
  local n, b0, b, i;
  n:=args[1];
  b0:=[seq(0,i=1..n)]; b:=[];
  for i from 1 to n do
    b:=[op(b),subsop(i=1,b0)];
  od;
  b;
end;

# Zeitausgabe wandelt Sekunden in das Format Stunden:Minuten:Sekunden um
zeitausgabe:=proc()
  local z, stunden, minuten, sekunden, zeit;
  z:=round(args[1]);
  stunden:=floor(z/3600); z:=z-3600*stunden;
  minuten:=floor(z/60); z:=z-60*minuten;
  sekunden:=z;
  if stunden<10 then
    stunden:=cat("0",convert(stunden,string));
  else
    stunden:=convert(stunden,string);
  fi;
  if minuten<10 then
    minuten:=cat("0",convert(minuten,string));
  else
```

```
    minuten:=convert(minuten,string);
  fi;
  if sekunden<10 then
    sekunden:=cat("0",convert(sekunden,string));
  else
    sekunden:=convert(sekunden,string);
  fi;
  zeit:=cat(stunden,":",minuten,":",sekunden);
  zeit;
end;

# Rechenzeit der LLL-Reduktion mit NTL
# Eingabe: Matrix, Ausgabedatei
rz_lll_ntl:=proc()
  local b, m, n, gh, aus;
  global lll_tmp1, lll_tmp2, b_zeit, b_det2, b_red;
  b:=args[1]; m:=nops(b); n:=nops(b[1]); aus:=args[2];
  gh:=gitter_hoehe(b);
  printf("m=%d n=%d log_10(gh)=%f\n",m,n,log[10](gh));
  # Die Gitterbasis b wird in die Datei lll_tmp1 geschrieben:
  gitter_ma2ntl(b,lll_tmp1);
  # LLL-Reduktion mit NTL:
  system("lll_ntl lll_tmp1 lll_tmp2");
  read lll_tmp2;
  printf("Rechenzeit: %s\n",b_zeit);
  printf("log_10(det)=%f\n",log[10](sqrt(b_det2)));
  fopen(aus,APPEND);
  fprintf(aus,"%d & %d & %.2f & %s\\\\\n",m,n,log[10](gh),b_zeit);
  fclose(aus);
end;

# Rechenzeit der LLL-Reduktion mit Maple
# Eingabe: Matrix, Ausgabedatei
rz_lll_ma:=proc()
  local b, m, n, gh, b_zeit, b_red, aus;
  b:=args[1]; m:=nops(b); n:=nops(b[1]); aus:=args[2];
  gh:=gitter_hoehe(b);
  printf("m=%d n=%d log_10(gh)=%f\n",m,n,log[10](gh));
  b_zeit:=time();
  b_red:=lattice(b);
  b_zeit:=time()-b_zeit;
  printf("Rechenzeit: %s\n",zeitausgabe(b_zeit));
  fopen(aus,APPEND);
  fprintf(aus,"%d & %d & %.2f & %s\\\\\n",m,n,log[10](gh),zeitausgabe(b_zeit));
  fclose(aus);
end;

# Rechenzeit der LLL-Reduktion mit eigenem Maple-Programm lll
# Eingabe: Matrix, Ausgabedatei
rz_lll:=proc()
  local b, m, n, gh, b_zeit, b_red, aus;
  b:=args[1]; m:=nops(b); n:=nops(b[1]); aus:=args[2];
  gh:=gitter_hoehe(b);
  printf("m=%d n=%d log_10(gh)=%f\n",m,n,log[10](gh));
```

```
  b_zeit:=time();
  b_red:=lll(b);
  b_zeit:=time()-b_zeit;
  printf("Rechenzeit: %s\n",zeitausgabe(b_zeit));
  fopen(aus,APPEND);
  fprintf(aus,"%d & %d & %.2f & %s\\\\\n",m,n,log[10](gh),zeitausgabe(b_zeit));
  fclose(aus);
end;
```

1.2. lll_ntl.c.

```
/* lll_ntl.c
   Version: 30.4.2002
   Uebersetzung: g++ lll_ntl.c -o lll_ntl -lntl -lgmp
   Die Eingabedatei muss eine Liste von Vektoren in der Form
   [[b11 b12 ... b1n][b21 b22 ... b2n] ... [bm1 bm2 ... bmn]]
   enthalten. Die Ausgabedatei enthaelt in Maple-Form die Determinante
   b_det, eine reduzierte Gitterbasis b_red und die benoetigte
   Rechenzeit zeit (als Zeichenkette).
*/

#include <fstream.h>
#include <NTL/LLL.h>

int main( int argc, char *argv[])
{
  if (argc!=3)
  {
    cout<<"Aufruf: 'lll_ntl Eingabedatei Ausgabedatei'\n";
    return 0;
  }

  ifstream(ein); ein.open(argv[1]);
  ofstream(aus); aus.open(argv[2]);

  ZZ b_det2; mat_ZZ b;
  ein>>b;

  int m=b.NumRows(), n=b.NumCols();
  cout<<"LLL-Gitterbasenreduktion eines Gitters vom Rang "<<m;
  cout<<" im R^"<<n<<"\n";
  cout<<"Eingabedatei: "<<argv[1]<<"\n";
  cout<<"Ausgabedatei: "<<argv[2]<<"\n";

  LLL(b_det2,b,0);

  double zeit=GetTime();

  aus<<"# Ergebnis einer LLL-Gitterbasenreduktion eines Gitters vom\n";
  aus<<"# Rang "<<m<<" im R^"<<n<<" mit dem Programm lll_ntl.c.\n";
  aus<<"# Dabei ist b_zeit die benoetigte Rechenzeit, b_det2 das ";
  aus<<"Quadrat \n# der Gitterdeterminante und b_red eine reduzierte ";
  aus<<"Gitterbasis.\n\n";
  aus<<"b_zeit:=\""; PrintTime(aus,zeit); aus<<"\";\n";

  aus<<"b_det2:="<<b_det2<<";\n";
```

```
  aus<<"b_red:=[\n[";
  for (int i=1;i<=m;i++)
  {
    for (int j=1;j<=n;j++)
    {
      aus<<b(i,j);
      if (j<n) aus<<","; else aus<<"]";
    }
    if (i<m) aus<<",\n["; else aus<<"\n];\n";
  }

  cout<<"Rechenzeit: "; PrintTime(cout,zeit); cout<<"\n";
  return 0;
}
```

2. Rucksackalgorithmen - Die Merkle-Hellman-Rucksackverschlüsselung

2.1. ru_ntl.c.

```
/* ru_ntl.c
   Version: 11.5.2002
   Uebersetzung: g++ ru_ntl.c -o ru_ntl -lntl -lgmp
   Naive Rucksackberechnungsmethode
   Die 1. Eingabedatei enthaelt [a_1 a_2 ... a_n], die 2. Eingabedatei
   eine Zahl s. Bestimmt werden x_i=0,1 mit x_1a_1+...+x_na_n=s.
   Die Ausgabe erfolgt in der im Vorlesungsskript benutzten Form.
*/

#include <fstream.h>
#include <NTL/LLL.h>

int main( int argc, char *argv[])
{
  if (argc!=3)
  {
    cout<<"Aufruf: 'ru_ntl 1.Eingabedatei 2.Eingabedatei'\n";
    return 0;
  }
  ifstream(ein1); ein1.open(argv[1]);
  ifstream(ein2); ein2.open(argv[2]);

  vec_ZZ a, x; ZZ s;

  ein1>>a; // a liefert den Rucksack a_1,...,a_n

  int n=a.length();
  x.SetLength(n);
  cout<<n<<" & ";

  // a_min und a_max sind Minimum und Maximum von a_1,...,a_n
  ZZ amin, amax;
  amin=a[0]; amax=a[0];
  for (int i=1;i<n;i++)
  {
    if (a[i]<amin) amin=a[i];
    if (a[i]>amax) amax=a[i];
```

```
  }
  cout<<""<<amin<<" & "<<amax<<" & ";

  ein2>>s;
  cout<<s<<" & ";

  ZZ xxmax;
  xxmax=1; for (int i=1;i<=n;i++) xxmax*=2; // xxmax=2^n

  ZZ xx, y, ss;
  xx=0;
  while (xx<xxmax)
  {
    y=xx; ss=0;
    for (int i=0;i<n;i++)
    {
      x[i]=y%2; y/=2;
      ss+=x[i]*a[i];
    }
    if (s==ss)
    {
      for (int j=0;j<n;j++) cout<<x[j]; cout<<" & ";
      xx=xxmax;
    }
    xx++;
  }

  double zeit=GetTime();
  PrintTime(cout,zeit); cout<<"\\\\\n";
  return 0;
}
```

2.2. mehe_ma.

```
# mehe_ma
# Kapitel: Rucksackalgorithmen -
#          Die Merkle-Hellman-Rucksackverschluesselung
# Version: 17. Juli 2002
# Funktionen:
# mehe_key
# byte2bits
# auffuellen
# wegstreichen
# mehe_en0
# mehe_en
# mehe_de0
# mehe_de
# v_mult
# gitter_ma2ntl
# superincreasing
# UM_kon
# sort_b
# aUM_test
# mehe_angriff5
# kleineindizes
# mehe_angriff
```

```
# Zufaellige Erzeugung eines Merkle-Hellman-Schluesselpaares
# Eingabe: Folgenlaenge n
#          Wird noch etwas eingegeben, wird Permutation=Identitaet
# Ausgabe: [k_pub,k_priv] mit k_priv=[U,M,b,perm]
mehe_key:=proc()
  local n, z, b, su, i, M, W, perm, j, k, perm1, pi, pj, a, k_pub, U,
        k_priv;
  n:=args[1];
  z:=rand(1..2^n);
  b:=[2^n+z()]; su:=b[1];
  for i from 2 to n do
    b:=[op(b),su+z()];
    su:=su+b[i];
  od;
  M:=su+z(); z:=rand(1..M);
  W:=0; while igcd(W,M)>1 do W:=z(); od;
  perm:=[];
  for i from 1 to n do
    perm:=[op(perm),i];
  od;
  if nargs=1 then
    for k from 1 to n do
      i:=z() mod n; if i=0 then i:=n; fi; pi:=perm[i];
      j:=z() mod n; if j=0 then j:=n; fi; pj:=perm[j];
      perm1:=subsop(i=pj,perm); perm:=subsop(j=pi,perm1);
    od;
  fi;
  a:=[];
  for i from 1 to n do
    a:=[op(a),(W*b[perm[i]]) mod M];
  od;
  k_pub:=a;
  U:=1/W mod M;
  k_priv:=[U,M,b,perm];
  [k_pub,k_priv];
end;

# Umwandlung eines Bytes in 8 Bits
byte2bits:=proc()
  local b, bi, i;
  b:=args[1];
  bi:=[];
  for i from 1 to 8 do
    bi:=[irem(b,2),op(bi)]; b:=iquo(b,2);
  od;
  bi;
end;

# Eine Byte-Folge wird aufgefuellt, damit die Anzahl der Folgenglieder
# durch eine gegebene Zahl k teilbar.
# Eingabe: Bytefolge bf, gewuenschte Blocklaenge k
auffuellen:=proc()
  local bf, k, zuviel, i;
```

```
  bf:=args[1]; k:=args[2];
  zuviel:=k-(nops(bf) mod k);
  for i from 1 to zuviel-1 do bf:=[op(bf),10]; od;
  bf:=[op(bf),zuviel];
end;

# wegstreichen - Umkehrabbildung zu 'auffuellen'
# Eingabe: Bytefolge bf
wegstreichen:=proc()
  local bf, zuviel, i;
  bf:=args[1];
  zuviel:=bf[nops(bf)];
  for i from 1 to zuviel do
    bf:=subsop(nops(bf)=NULL,bf);
  od;
  bf;
end;

# Merkle-Hellman-Verschluesselung
# Eingabe: Bitfolge mit n Bits, oeffentlicher Schluessel [a_1,...,a_n]
mehe_en0:=proc()
  local x, a, n, s, i;
  x:=args[1]; a:=args[2];
  n:=nops(a);
  s:=0;
  for i from 1 to n do
    s:=s+x[i]*a[i];
  od;
  s;
end;

# Merkle-Hellman-Verschluesselung
# Eingabe: Bytefolge, oeffentlicher Schluessel
mehe_en:=proc()
  local bytf, k_pub, n, bitf, bf, b, i;
  bytf:=args[1]; k_pub:=args[2]; n:=nops(k_pub);
  if n mod 8>0 then error "n sollte durch 8 teilbar sein!"; fi;
  # Die Bytefolge wird ergaenzt, damit die Anzahl der Bytes durch n/8
  # teilbar wird:
  bytf:=auffuellen(bytf,n/8);
  # Umwandlung der Bytefolge in eine Bitfolge:
  bitf:=map(x->op(byte2bits(x)),bytf);
  # Aufteilung der Bitfolge in Bloecke mit je n Bits:
  bf:=[]; b:=[];
  for i from 1 to nops(bitf) do
    b:=[op(b),bitf[i]];
    if i mod n=0 then
      bf:=[op(bf),b]; b:=[];
    fi;
  od;
  # Verschluesselung:
  map(x->mehe_en0(x,k_pub),bf);
end;
```

```
# Merkle-Hellman-Entschluesselung
mehe_de0:=proc()
  local s, k_priv, U, M, b, perm, n, x, t, j, y, i;
  s:=args[1]; k_priv:=args[2];
  U:=k_priv[1]; M:=k_priv[2]; b:=k_priv[3]; perm:=k_priv[4];
  n:=nops(b);
  x:=[];
  t:=U*s mod M;
  for j from n to 1 by -1 do
    if t>=b[j] then
      x:=[1,op(x)]; t:=t-b[j];
    else
      x:=[0,op(x)];
    fi;
  od;
  x;
  y:=[];
  for i from 1 to n do
    y:=[op(y),x[perm[i]]];
  od;
  y;
end;

# Merkle-Hellman-Entschluesselung
# Eingabe: Bytefolge, privater Schluessel
mehe_de:=proc()
  local s, k_priv, n, bitf, bytf, b, i;
  s:=args[1]; k_priv:=args[2]; n:=nops(k_priv[4]);
  # Elementweise Entschluesselung
  bitf:=map(x->op(mehe_de0(x,k_priv)),s);
  # Umwandlung der Bitfolge in eine Bytefolge:
  bytf:=[]; b:=0;
  for i from 1 to nops(bitf) do
    b:=2*b+bitf[i];
    if i mod 8=0 then
      bytf:=[op(bytf),b]; b:=0;
    fi;
  od;
  wegstreichen(bytf);
end;

# Skalarprodukt
v_mult:=proc()
  local a, b, s, i;
  a:=args[1]; b:=args[2];
  if nops(a)<>nops(b) then
    error "Die Vektoren sind nicht gleich lang!";
  fi;
  s:=0;
  for i from 1 to nops(a) do
    s:=s+a[i]*b[i];
  od;
  s;
end;
```

```
gitter_ma2ntl:=proc()
  local M, aus, m, n, i, j;
  M:=args[1]; aus:=args[2];
  m:=nops(M); n:=nops(M[1]);
  # Die Matrix M wird in NTL-Format in 'aus' geschrieben.
  fprintf(aus,"[\n");
  for i from 1 to m do
    fprintf(aus,"[");
    for j from 1 to n do
      fprintf(aus,"%d ",M[i][j]);
    od;
    fprintf(aus,"]\n");
  od;
  fprintf(aus,"]\n");
  fclose(aus);
end;

# Test, ob eine Folge [b_1,...,b_n] superincreasing ist oder nicht
superincreasing:=proc()
  local b, n, s, i;
  b:=args[1]; n:=nops(b); s:=0;
  for i from 1 to n do
    if s>=b[i] then
      return false;
    else
      s:=s+b[i];
    fi;
  od;
  true;
end;

UM_kon:=proc()
  local k, a, i, n, LSmax, RSmin, j, as, ks, l, t, UM, M, U, x;
  k:=args[1]; a:=args[2]; i:=args[3]; n:=nops(a);
  LSmax:=k[1]/a[i[1]]; RSmin:=(1+k[1])/a[i[1]];
  for j from 1 to 5 do
    LSmax:=max(LSmax,k[j]/a[i[j]]);
    RSmin:=min(RSmin,(1+k[j])/a[i[j]]);
  od;
  for j from 2 to 5 do
    as:=-a[i[j]]; ks:=-k[j];
    for l from 1 to j-1 do
      as:=as+a[i[l]]; ks:=ks+k[l];
    od;
    if as>0 then
      RSmin:=min(RSmin,ks/as);
    else
      LSmax:=max(LSmax,ks/as);
    fi;
  od;
  if LSmax>RSmin then return false; fi;
  t:=1/2;UM:=t*LSmax+(1-t)*RSmin; M:=denom(UM); U:=M*UM;
  x:=aUM_test(a,U,M);
```

```
  if x[1]=false then return false; fi;
  x[2];
end;

# Sortieren, so dass dann a[i]=b[p[i]] bzw. a_i=b_(p_i) gilt
# Eingabe: a=[a_1,...,a_n]
# Ausgabe: b=[b_1,...,b_n] und p=[p_1,...,p_n]
sort_b:=proc()
  local a, n, amax, d, s, i, b, pi, p, j;
  a:=args[1]; n:=nops(a);
  amax:=max(op(a));
  d:=0; while 10^d<=amax do d:=d+1; od;
  s:="%"||d||"d %d\n";
  for i from 1 to n do
    fprintf(sort_tmp1,s,a[i],i);
  od;
  fclose(sort_tmp1);
  system("sort sort_tmp1 > sort_tmp2");
  b:=[]; pi:=[];
  for i from 1 to n do
    b:=[op(b),op(fscanf(sort_tmp2,"%d"))];
    pi:=[op(pi),op(fscanf(sort_tmp2,"%d"))];
  od;
  fclose(sort_tmp2);
  p:=[];
  for i from 1 to n do
    for j from 1 to n do
      if pi[j]=i then p:=[op(p),j]; break; fi;
    od;
  od;
  [b,p];
end;

aUM_test:=proc()
  local a, U, M, n, aUM, i, bp, b, p, si, su;
  a:=args[1]; U:=args[2]; M:=args[3]; n:=nops(a);
  aUM:=[];
  for i from 1 to n do
    aUM:=[op(aUM),(U*a[i]) mod M];
  od;
  bp:=sort_b(aUM); b:=bp[1]; p:=bp[2];
  si:=superincreasing(b);
  if si=true then
    su:=0;
    for i from 1 to n do su:=su+b[i]; od;
    if su>=M then si:=false; fi;
  fi;
  [si,[U,M,b,p]];
end;

# Eingabe: oeffentlicher Schluessel a=[a1,....,an],
#          5 Indizes [i1,i2,i3,i4,i5]
mehe_angriff5:=proc()
  local a, i, n, mu, B, Bred, Bred1, Bred2, Bred2a, Bred2b, ka, kb,
```

```
        UMa, UMb;
  global lll_tmp1, lll_tmp2, b_red;
  a:=args[1]; i:=args[2]; n:=nops(a);
  mu:=2^n;
  B:=[[1,-mu*a[i[2]],-mu*a[i[3]],-mu*a[i[4]],-mu*a[i[5]]],
      [0,mu*a[i[1]],0,0,0],[0,0,mu*a[i[1]],0,0],
      [0,0,0,mu*a[i[1]],0],[0,0,0,0,mu*a[i[1]]]];
  # Bred:=lattice(B);
  gitter_ma2ntl(B,lll_tmp1);
  system("lll_ntl lll_tmp1 lll_tmp2");
  read lll_tmp2;
  Bred:=b_red;
  Bred1:=Bred[1];
  if Bred1[1]<0 then
    Bred1:=map(x->-x,Bred1);
  fi;
  if Bred1[1]<>a[i[1]] or
    Bred1[2]<>0 or Bred1[3]<>0 or Bred1[4]<>0 or Bred1[5]<>0 then
    printf("1. Vektor anders als erwartet!\n");
    return false;
  fi;
  Bred2:=Bred[2];
  if Bred2[1]<0 then
    Bred2:=map(x->-x,Bred2);
  fi;
  if Bred2[1]>=a[i[1]] then
    printf("2. Vektor in 1. Komponente nicht gut reduziert!\n");
    return false;
  fi;
  Bred2a:=Bred2;
  Bred2b:=map(x->-x,Bred2); Bred2b:=subsop(1=a[i[1]]-Bred2[1],Bred2b);
  ka:=convert(evalm(Bred2a&*B^(-1)),list);
  kb:=convert(evalm(Bred2b&*B^(-1)),list);
  UMa:=UM_kon(ka,a,i);
  UMb:=UM_kon(kb,a,i);
  if UMa<>false then printf("UMa o.k.\n"); fi;
  if UMb<>false then printf("UMb o.k.\n"); fi;
  if UMa<>false then
    return UMa;
  elif UMb<>false then
    return UMb;
  fi;
  false;
end;

# Eingabe: K=[k_pub,k_priv] - Merkle-Hellman-Schluesselpaar
# Ausgabe: [i1,i2,i3,i4,i5] mit b_i1<b_i2<b_i3<b_i4<b_i5<...
kleineindizes:=proc()
  local K, k_priv, p, i, j, l;
  K:=args[1];
  k_priv:=K[2]; p:=k_priv[4];
  i:=[];
  for j from 1 to 5 do
    for l from 1 to nops(p) do
```

```
      if p[l]=j then i:=[op(i),l]; break; fi;
    od;
  od;
  i;
end;

# Eingabe: k_pub oder k_pub,[i1,i2,i3,i4,i5]
mehe_angriff:=proc()
  local a, n, i1, i2, i3, i4, i5, L;
  if nargs>1 then
    return mehe_angriff5(args[1],args[2]);
  fi;
  a:=args[1];
  n:=nops(a);
  for i1 from 1 to n do
    for i2 from 1 to n do
      for i3 from 1 to n do
        for i4 from 1 to n do
          for i5 from 1 to n do
            if nops({i1,i2,i3,i4,i5})=5 then
              L:=mehe_angriff5(a,[i1,i2,i3,i4,i5]);
              if L<>false then return L; fi;
            fi;
          od;
        od;
      od;
    od;
  od;
  false;
end;
```

3. Gitterangriffe auf Rucksäcke mit kleiner Dichte

3.1. lda_ma.

```
# lda_ma
# Kapitel: Gitterangriffe auf Rucksaecke mit kleiner Dichte
# Version: 17. Juli 2002
# Funktionen:
# gitter_ma2ntl
# v_mult
# dichte
# lda_I0
# lda_I1
# lda_I
# lda_II
# lda_x
# zufbitf

gitter_ma2ntl:=proc()
  local M, aus, m, n;
  M:=args[1]; aus:=args[2];
  m:=nops(M); n:=nops(M[1]);
  # Die Matrix M wird in NTL-Format in 'aus' geschrieben.
  fprintf(aus,"[\n");
  for i from 1 to m do
```

```
    fprintf(aus,"[");
    for j from 1 to n do
      fprintf(aus,"%d ",M[i][j]);
    od;
    fprintf(aus,"]\n");
  od;
  fprintf(aus,"]\n");
  fclose(aus);
end;

# Skalarprodukt
v_mult:=proc()
  local a, b, s, i;
  a:=args[1]; b:=args[2];
  if nops(a)<>nops(b) then
    error "Die Vektoren sind nicht gleich lang!";
  fi;
  s:=0;
  for i from 1 to nops(a) do
    s:=s+a[i]*b[i];
  od;
  s;
end;

# Dichte eines Rucksacks
dichte:=proc()
  local a, n, d, i;
  a:=args[1]; n:=nops(a);
  d:=0;
  for i from 1 to n do
    d:=max(d,log[2](a[i]));
  od;
  d:=n/d;
  evalf(d);
end;

# low density attack
# Eingabe: s
#          [a_1,...,a_n] (oeffentlicher Rucksackschluessel)
#          evtl. mu
# Ausgabe: [x_1,...,x_n] mit x_i=0,1 und s=x_1a_1+...+x_na_n
#          oder
#          false, wenn das Verfahren nicht funktioniert
lda_I0:=proc()
  local s, a, n, b0, b, i, bi, az, S, x, mu;
  global lll_tmp1, lll_tmp2, b_zeit, b_det2, b_red;
  s:=args[1]; a:=args[2]; n:=nops(a);
  if nargs<3 then mu:=1; else mu:=args[3]; fi;
  b0:=[seq(0,i=1..n)]; b:=[];
  for i from 1 to n do
    bi:=subsop(i=1,b0);
    b:=[op(b),[op(bi),mu*a[i]]];
  od;
  b:=[op(b),[op(b0),mu*s]];
```

```
  # Die Gitterbasis b wird in die Datei lll_tmp1 geschrieben:
  gitter_ma2ntl(b,lll_tmp1);
  # LLL-Reduktion mit NTL:
  system("lll_ntl lll_tmp1 lll_tmp2");
  read lll_tmp2; # Die reduzierte Basis ist nun b_red.
  az:=0;
  for i from 1 to n+1 do
    if b_red[i][n+1]=0 then
      az:=az+1;
      S:=convert(b_red[i],set);
      if S={0,1} or S={0,-1} or S={1} or S={-1} then
        x:=subsop(n+1=NULL,b_red[i]);
        if S={0,-1} or S={-1} then x:=map(y->-y,x); fi;
        if v_mult(a,x)=s then return x; fi;
      fi;
    fi;
  od;
  false;
end;

lda_I1:=proc()
  s:=args[1]; a:=args[2];
  if nargs<3 then mu:=1; else mu:=args[3]; fi;
  n:=nops(a);
  sa:=0; for i from 1 to n do sa:=sa+a[i]; od;
  x:=lda_I0(sa-s,a,mu);
  if x<>false then return map(y->1-y,x); fi;
  false;
end;

lda_I:=proc()
  s:=args[1]; a:=args[2]; n:=nops(a);
  x:=lda_I0(s,a);
  if x<>false then return x; fi;
  sa:=0; for i from 1 to n do sa:=sa+a[i]; od;
  x:=lda_I0(sa-s,a);
  if x<>false then return map(y->1-y,x); fi;
  false;
end;

lda_II:=proc()
  local s, a, n, b0, b, i, bi, az, S, x, mu;
  global lll_tmp1, lll_tmp2, b_zeit, b_det2, b_red;
  s:=args[1]; a:=args[2]; n:=nops(a);
  if nargs<3 then mu:=1; else mu:=args[3]; fi;
  b0:=[seq(0,i=1..n)]; b:=[];
  for i from 1 to n do
    bi:=subsop(i=2,b0);
    b:=[op(b),[op(bi),mu*a[i]]];
  od;
  b:=[op(b),[seq(1,i=1..n),mu*s]];
  # Die Gitterbasis b wird in die Datei lll_tmp1 geschrieben:
  gitter_ma2ntl(b,lll_tmp1);
  # LLL-Reduktion mit NTL:
```

```
  system("lll_ntl lll_tmp1 lll_tmp2");
  read lll_tmp2; # Die reduzierte Basis ist nun b_red.
  az:=0;
  for i from 1 to n+1 do
    if b_red[i][n+1]=0 then
      y:=subsop(n+1=NULL,b_red[i]);
      az:=az+1;
      S:=convert(y,set);
      if S={-1,1} or S={-1} or S={1} then
        x:=map(z->(z+1)/2,y);
        if v_mult(a,x)=s then return x; fi;
        x:=map(z->(-z+1)/2,y);
        if v_mult(a,x)=s then return x; fi;
      fi;
    fi;
  od;
  false;
end;

# Low-density-attack
# Eingabe: [c_1,c_2,c_3,...] (verschluesselter Text),
#          [a_1,...,a_n] (oeffentlicher Rucksackschluessel)
# Ausgabe: Entschluesselter Text bzw. **...*, falls der
#          Entschluesselungsversuch misslang
lda_x:=proc()
  c:=args[1]; a:=args[2]; m:=[];
  for i from 1 to nops(c) do
    x:=lda_II(c[i],a); mi:=[];
    if x=false then x:=lda_I(c[i],a); fi;
    if x<>false then
      byt:=0;
      for j from 1 to nops(x) do
        byt:=2*byt+x[j];
        if j mod 8=0 then
          mi:=[op(mi),byt]; byt:=0;
        fi;
      od;
    else
      mi:=[seq(42,j=1..nops(a)/8)];
    fi;
    printf("%a",convert(mi,bytes));
    m:=[op(m),op(mi)];
  od;
  m;
end;

# Konstruktion eines zufaelligen 0-1-Vektors der Laenge n mit m
# Eintraegen 1
zufbitf:=proc()
  local n, m, z, x, i;
  n:=args[1]; m:=args[2];
  if m>n then error "m>n"; fi;
  z:=rand(1..n);
  x:=[seq(0,i=1..n)];
```

```
  while m>0 do
    i:=z();
    if x[i]=0 then x:=subsop(i=1,x); m:=m-1; fi;
  od;
  x;
end;
```

3.2. ku_ntl.c.

```
/* ku_ntl.c - 6.6.2002
   Berechnung der Anzahl der x aus Z^n mit |x|^2<=n
*/

#include <NTL/ZZ.h>

ZZ ku( int n)
{
  int h=SqrRoot(n), H=2*h+1, i;
  ZZ y, z, s, az, xi;
  for (z=0;z<power_ZZ(H,n);z++)
  {
    y=z; s=0;
    for (i=1;i<=n;i++)
    {
      xi=(y%H)-h; s+=xi*xi; y/=H;
    }
    if (s<=n) az++;
  }
  return az;
}

int main()
{
  for (int n=1;n<=20;n++)
  {
    cout<<"ku("<<n<<")="<<ku(n)<<"\n";
  }
  return 0;
}
```

4. Das Chor-Rivest-Verschlüsselungsverfahren

4.1. choriv_ma.

```
# choriv_ma
# Kapitel: Das Chor-Rivest-Verschluesselungsverfahren
# Version: 9.7.2002
# Funktionen:
# T_ph(m,p,h)
# T_ph_i(M,p,h)
# dislog(k,g,f,p)
# dislog_pollard(k,g,f,p,n)
# dislog_naiv_ntl(k,g,f,p)
# dislog_pollard_ntl(k,g,f,p,n)
# fph_gen(p,h)
# choriv_key(p,h)
# choriv_en0(M,k_pub)
```

```
# choriv_de0(m,k_priv)
# gamma_r_test(c,p,h,f,r,u_r_start)
# gamma_r_test_ntl(c,p,h,f,r,u_r_start)
# u_6_liste(c,p,h,f)
# angriff_h12([c,p,h]) oder [c,p,h],f oder [c,p,h],f,u6
# chrem_ex2([a1,a2],[m1,m2])
# inv(g,f,p)
# zeit_sec2hms(t)
# k_priv2pub(f,g,pi,d)
# k_pub_priv_test(k_pub,k_priv)
# angriff_h24([c,p,h]) oder [c,p,h],f oder [c,p,h],f,u6
# pi_norm(pi)

# T_ph wandelt eine ganze Zahl m mit 0<=m<=binomial(p,h)-1 in eine
# Bitfolge M der L"ange p mit Hamming-Gewicht h um. T_ph_i ist die
# inverse Abbildung.
# Eingabe: m, p, h mit 0<=m<=binomial(p,h)-1
# Ausgabe: T_ph(m)
T_ph:=proc()
  local m, p, h, l, M, i;
  m:=args[1]; p:=args[2]; h:=args[3];
  if m>=binomial(p,h) then error "m zu gross!"; fi;
  l:=h; M:=[];
  for i from 1 to p do
    if m>=binomial(p-i,l) then
      M:=[op(M),1]; m:=m-binomial(p-i,l); l:=l-1;
    else
      M:=[op(M),0];
    fi;
  od;
  M;
end;

# Eingabe: M, p, h, wobei M eine Bitfolge der Laenge p mit
#          Hamming-Gewicht h ist
# Ausgabe: T_ph^(-1)(M)
T_ph_i:=proc()
  local M, p, h, l, m, i;
  M:=args[1]; p:=args[2]; h:=args[3];
  m:=0; l:=h;
  for i from 1 to p do
    if M[i]=1 then
      m:=m+binomial(p-i,l); l:=l-1;
    fi;
  od;
  m;
end;

# Berechnung diskreter Logarithmen: g(x)^z=k(x) mod f(x) ueber F_p mit
# dem Silver-Pohlig-Hellman-Verfahren
# Eingabe: k(x), g(x), f(x), p
#          Werden mehr als 4 Argumente eingegeben, werden die
#          Logarithmen nur auf naive Weise mit Maple berechnet, sonst
#          wird auch NTL mit dem Pollardschen rho-Verfahren benutzt.
```

```
# Ausgabe: z=log_g(k)
dislog:=proc()
  global ntl_zeit;
  local k, g, f, x, p, h, ph, pp, i, zz, e, ge, ke, ze, geze;
  k:=args[1]; g:=args[2]; f:=args[3]; p:=args[4];
  x:=op(indets(f)); h:=degree(f,x);
  ph:=ifactors(p^h-1)[2];
  pp:=[];
  for i from 1 to nops(ph) do
    pp:=[op(pp),ph[i][1]^ph[i][2]];
  od;
  zz:=[];
  for i from 1 to nops(ph) do
    e:=(p^h-1)/pp[i];
    ge:=Powmod(g,e,f,x) mod p;
    ke:=Powmod(k,e,f,x) mod p;
    if pp[i]<100 or nargs>4 then
      ze:=0; geze:=1;
      while geze<>ke do
        ze:=ze+1; geze:=Rem(geze*ge,f,x) mod p;
        if ze>pp[i] then error "dislog exisitiert nicht!"; fi;
      od;
    else
      ze:=dislog_pollard_ntl(ke,ge,f,p,pp[i]);
      # Falls das Pollarsche Verfahren nicht funktioniert hat:
      if ze<0 then
        ze:=dislog_naiv_ntl(ke,ge,f,p);
        printf("!\n");
      fi;
    fi;
    zz:=[op(zz),ze];
  od;
  chrem(zz,pp) mod (p^h-1);
end;

# Berechnung diskreter Logarithmen mit dem Pollardschen rho-Verfahren
# Eingabe: a(x), g(x), f(x), p, n=ord(g(x)) in F_p[x]/f(x)
# Ausgabe: dlog mit g(x)^dlog=a(x) mod f(x) ueber F_p oder
#          dlog<0, falls das Verfahren nicht funktioniert.
#          k(x)^n=1 mod f(x) sollte sichergestellt sein.
dislog_pollard:=proc()
  local k, g, f, p, n, x, x_i, a_i, b_i, x_i1, a_i1, b_i1, x_2i, a_2i,
        b_2i, x_2i1, a_2i1, b_2i1, x_2i2, a_2i2, b_2i2;
  k:=args[1]; g:=args[2]; f:=args[3]; p:=args[4]; n:=args[5];
  x:=op(indets(f));
  x_i:=1; a_i:=0; b_i:=0; x_2i:=1; a_2i:=0; b_2i:=0;
  do
    if subs(x=0,x_i) mod 3=0 then
      x_i1:=Rem(x_i*x_i,f,x) mod p;
      a_i1:=2*a_i mod n; b_i1:=2*b_i mod n;
    elif subs(x=0,x_i) mod 3=1 then
      x_i1:=Rem(k*x_i,f,x) mod p;
      a_i1:=a_i; b_i1:=(b_i+1) mod n;
    else
```

```
      x_i1:=Rem(g*x_i,f,x) mod p;
      a_i1:=(a_i+1) mod n; b_i1:=b_i;
    fi;
    x_i:=x_i1; a_i:=a_i1; b_i:=b_i1;

    if subs(x=0,x_2i) mod 3=0 then
      x_2i1:=Rem(x_2i*x_2i,f,x) mod p;
      a_2i1:=2*a_2i mod n; b_2i1:=2*b_2i mod n;
    elif subs(x=0,x_2i) mod 3=1 then
      x_2i1:=Rem(k*x_2i,f,x) mod p;
      a_2i1:=a_2i; b_2i1:=(b_2i+1) mod n;
    else
      x_2i1:=Rem(g*x_2i,f,x) mod p;
      a_2i1:=(a_2i+1) mod n; b_2i1:=b_2i;
    fi;

    if subs(x=0,x_2i1) mod 3=0 then
      x_2i2:=Rem(x_2i1*x_2i1,f,x) mod p;
      a_2i2:=2*a_2i1 mod n; b_2i2:=2*b_2i1 mod n;
    elif subs(x=0,x_2i1) mod 3=1 then
      x_2i2:=Rem(k*x_2i1,f,x) mod p;
      a_2i2:=a_2i1; b_2i2:=(b_2i1+1) mod n;
    else
      x_2i2:=Rem(g*x_2i1,f,x) mod p;
      a_2i2:=(a_2i1+1) mod n; b_2i2:=b_2i1;
    fi;
    x_2i:=x_2i2; a_2i:=a_2i2; b_2i:=b_2i2;

    if x_i=x_2i then break; fi;
  od;
  if igcd(b_i-b_2i,n)>1 then return -igcd(b_i-b_2i,n); fi;
  (a_2i-a_i)/(b_i-b_2i) mod n;
end;

# Berechnung diskreter Logarithmen mit NTL auf naive Weise
# Eingabe: a(x), g(x), f(x), p
# Ausgabe: dlog mit g(x)^dlog=a(x) mod f(x) ueber F_p
dislog_naiv_ntl:=proc()
  global dlog_tmp1, dlog_tmp2, dlog, dlog_zeit, ntl_zeit;
  local a, g, f, p, x, h, j;
  a:=args[1]; g:=args[2]; f:=args[3]; p:=args[4];
  x:=op(indets(f)); h:=degree(f,x);
  # p, f(x), g(x), a(x) werden in die Datei dlog_tmp1 geschrieben
  fprintf(dlog_tmp1,"%d\n[",p);
  for j from 0 to h do fprintf(dlog_tmp1,"%d ",coeff(f,x,j)); od;
  fprintf(dlog_tmp1,"]\n[");
  for j from 0 to h-1 do fprintf(dlog_tmp1,"%d ",coeff(g,x,j)); od;
  fprintf(dlog_tmp1,"]\n[");
  for j from 0 to h-1 do fprintf(dlog_tmp1,"%d ",coeff(a,x,j)); od;
  fprintf(dlog_tmp1,"]\n");
  fclose(dlog_tmp1);
  # dlog_ntl berechnet den diskreten Logarithmus log_k(a) und schreibt
  # ihn zusammen mit der verbrauchten Rechenzeit dlog_zeit in die Datei
  # dlog_tmp2.
```

```
  system("dlog_naiv_ntl dlog_tmp1 dlog_tmp2");
  read dlog_tmp2;
  ntl_zeit:=ntl_zeit+dlog_zeit;
  dlog;
end;

# Berechnung diskreter Logarithmen mit NTL mit dem rho-Verfahren
# Eingabe: a(x), g(x), f(x), p, n=ord(g(x)) in F_p[x]/f(x)
# Ausgabe: dlog mit g(x)^dlog=a(x) mod f(x) ueber F_p oder
#          dlog<0, falls das Verfahren nicht funktioniert
dislog_pollard_ntl:=proc()
  global dlog_tmp1, dlog_tmp2, dlog, dlog_zeit, ntl_zeit;
  local a, g, f, p, x, h, j, n;
  a:=args[1]; g:=args[2]; f:=args[3]; p:=args[4]; n:=args[5];
  x:=op(indets(f)); h:=degree(f,x);
  # p, f(x), g(x), a(x), n werden in die Datei dlog_tmp1 geschrieben
  fprintf(dlog_tmp1,"%d\n[",p);
  for j from 0 to h do fprintf(dlog_tmp1,"%d ",coeff(f,x,j)); od;
  fprintf(dlog_tmp1,"]\n[");
  for j from 0 to h-1 do fprintf(dlog_tmp1,"%d ",coeff(g,x,j)); od;
  fprintf(dlog_tmp1,"]\n[");
  for j from 0 to h-1 do fprintf(dlog_tmp1,"%d ",coeff(a,x,j)); od;
  fprintf(dlog_tmp1,"]\n");
  fprintf(dlog_tmp1,"%d\n",n);
  fclose(dlog_tmp1);
  # dlog_pollard_ntl berechnet den diskreten Logarithmus log_k(a) und
  # schreibt ihn zusammen mit der verbrauchten Rechenzeit dlog_zeit in
  # die Datei dlog_tmp2.
  system("dlog_pollard_ntl dlog_tmp1 dlog_tmp2");
  read dlog_tmp2;
  ntl_zeit:=ntl_zeit+dlog_zeit;
  dlog;
end;

# Wird NTL benutzt, wird die Rechenzeit um ntl_zeit veraendert.
# Daher wird der Groesse zu Beginn der Wert 0 zugewiesen:
ntl_zeit:=0;

# Bestimmung eines modulo p irreduziblen Polynoms f vom Grad h, so
# dass x die multiplikative Gruppe von F_p[x]/(f) erzeugt.
# Eingabe: p, h
# Ausgabe: f(x)
fph_gen:=proc()
  local p, h, ph, z, fprim, f, i;
  p:=args[1]; h:=args[2];
  ph:=ifactors(p^h-1)[2];
  z:=rand(0..p-1);
  fprim:=0; f:=0;
  while (Irreduc(f) mod p=false) or fprim=0 do
    # Bestimmung eines zufaelligen normierten Polynoms
    f:=x^h;
    for i from h-1 to 0 by -1 do
      f:=f+z()*x^i;
    od;
```

```
    # Test, ob f primitiv ist
    if Irreduc(f) mod p=true then
      fprim:=1;
      for i from 1 to nops(ph) do
        if Powmod(x,(p^h-1)/ph[i][1],f,x) mod p=1 then
          fprim:=0; break;
        fi;
      od;
    fi;
  od;
  f;
end;

# Erzeugung eines zufaelligen Chor-Rivest-Schluessel-Paares
# (([c_0,...c_(p-1)],p,h),(f(x),g(x),pi,d))
# bei Eingabe von p, h
choriv_key:=proc()
  global ntl_zeit;
  local g, zeit0, p, h, ph, z, f, i, gprim, a, pi, i1, j1, i2, j2,
        d, c, zeit1;
  ntl_zeit:=0;
  zeit0:=time();
  p:=args[1]; h:=args[2];
  printf("Erzeugung eines zufaelligen Chor-Rivest-Schluessels:\n");
  printf("Parameter: Primzahl p=%d, Grad h=%d\n",p,h);
  ph:=ifactors(p^h-1)[2];
  z:=rand(0..p-1);
  # Bestimmung eines zufaelligen irreduziblen normierten Polynoms f vom
  # Grad h, das den endlichen Koerper mit p^h Elementen beschreibt
  printf("Bestimmung eines zufaelligen irreduziblen normierten ");
  printf("Polynoms vom Grad %d:\n",h);
  f:=0;
  while Irreduc(f) mod p=false do
    f:=x^h;
    for i from h-1 to 0 by -1 do
      f:=f+z()*x^i;
    od;
  od;
  printf("f=%a\n",f);
  # Bestimmung eines zufaelligen primitiven Elements g modulo f
  printf("Bestimmung eines zufaelligen Polynoms g, das primitiv ");
  printf("modulo f ist:\n");
  gprim:=0;
  while gprim=0 do
    g:=0;
    for i from h-1 to 0 by -1 do
      g:=g+z()*x^i;
    od;
    gprim:=1;
    if g=0 then
      gprim:=0;
    else
      for i from 1 to nops(ph) do
        if Powmod(g,(p^h-1)/ph[i][1],f,x) mod p=1 then
```

```
          gprim:=0; break;
        fi;
      od;
    fi;
  od;
  printf("g=%a\n",g);
  # Berechnung diskreter Logarithmen g^(a_i)=x+i mod f
  printf("Berechnung der diskreten Logarithmen a_i mit ");
  printf("g^(a_i)=x+i mod f:\n");
  a:=array(0..p-1);
  for i from 0 to p-1 do
    a[i]:=dislog(x+i,g,f,p);
    zeit1:=time();
    printf("a_%d=%d (%s)\n",i,a[i],zeit_sec2hms(zeit1-zeit0+ntl_zeit));
  od;
  # Zufaellige Permutation pi der Zahlen 0,...,p-1
  printf("Erzeugung einer zufaelligen Permutation ");
  printf("pi=[pi(0),pi(1),...,pi(p-1)]:\npi=[");
  pi:=array(0..p-1);
  for i from 0 to p-1 do pi[i]:=i; od;
  for i from 0 to p-1 do
    i1:=z(); j1:=pi[i1];
    i2:=z(); j2:=pi[i2];
    pi[i1]:=j2; pi[i2]:=j1;
  od;
  for i from 0 to p-1 do
    printf("%d",pi[i]);
    if i<p-1 then printf(","); else printf("]\n"); fi;
  od;
  # Zufaellige Zahl d mit 0<=d<=p^h-2
  printf("Wahl einer zufaelligen Zahl d mit 0<=d<=p^h-2\n");
  z:=rand(0..p^h-2);
  d:=z();
  printf("d=%d\n",d);
  # Oeffentlicher Schluessel
  printf("Berechnung von c_i=a_(pi(i)) mod (p^h-1)\n");
  c:=array(0..p-1);
  for i from 0 to p-1 do
    c[i]:=(a[pi[i]]+d) mod (p^h-1);
  od;
  zeit1:=time();
  printf("Ausgabe: Chor-Rivest-Schluesselpaar ");
  printf("((c,p,h),(f,g,pi^(-1),d))\n");
  printf("Rechenzeit: %.3f sec ",zeit1-zeit0+ntl_zeit);
  printf("(%s)\n",zeit_sec2hms(zeit1-zeit0+ntl_zeit));
  [[convert(c,list),p,h],[f,g,convert(pi,list),d]];
end;

# Chor-Rivest-Verschluesselung
# Eingabe: M, k_pub mit k_pub=[c,p,h] und M binaere Folge der Laenge p
# mit h Einsen
choriv_en0:=proc()
  local M, k_pub, c, p, h, cc, i;
  M:=args[1]; k_pub:=args[2]; c:=k_pub[1]; p:=k_pub[2]; h:=k_pub[3];
```

```
  cc:=0;
  for i from 1 to p do
    cc:=(cc+M[i]*c[i]) mod (p^h-1);
  od;
  cc;
end;

# Chor-Rivest-Entschluesselung: invers zu choriv_en0
choriv_de0:=proc()
  local cc, k_priv, f, g, perm, d, p, h, pi, i, r, u, s, M, t;
  cc:=args[1]; k_priv:=args[2];
  f:=k_priv[1]; g:=k_priv[2]; perm:=k_priv[3]; d:=k_priv[4];
  p:=nops(perm); h:=degree(f,x);
  pi:=array(0..p-1);
  for i from 0 to p-1 do pi[i]:=perm[i+1]; od;
  r:=(cc-h*d) mod (p^h-1);
  u:=Powmod(g,r,f,x) mod p;
  s:=(u+f) mod p;
  M:=array(0..p-1);
  for i from 0 to p-1 do
    if subs(x=-pi[i],s) mod p=0 then M[i]:=1; else M[i]:=0; fi;
  od;
  convert(M,list);
end;

#############################################################################
# gamma_r_test-Vergleich - Warum ist NTL schlechter?
#############################################################################
# p:=53; h:=12;
# c:=K53_12[1][1];
# f:=x^12+15*x^10+45*x^9+21*x^8+8*x^7+20*x^6+19*x^5+51*x^4+11*x^3
#    +36*x^2+x+50;
# U:=u_6_liste(c,p,h,f);
# Ergebnis: U:=[21490402573];
# Rechenzeiten
# (00:02:25) - gamma_r_test - nur mit Maple
# (00:07:27) - gamma_r_test_ntl - unter Verwendung von NTL
#############################################################################

# Liefert moegliche gamma_r=x^((p^h-1)/(p^r-1)*u_r) mod f
# Eingabe: c, p, h, f, r, u_r_start
# Ausgabe: u_r - Liste mit moeglichen Exponenten
gamma_r_test:=proc()
  local c, p, h, f, r, u_r_start, x, alpha_r, F, f_r, u_r, i, gamma_r,
        ok, e, s, j;
  c:=args[1]; p:=args[2]; h:=args[3]; f:=args[4];
  r:=args[5]; u_r_start:=args[6];
  x:=op(indets(f));
  c:=map(t->t mod (p^r-1),c); # c[i] wird nur mod p^r-1 benutzt
  alpha_r:=Powmod(x,(p^h-1)/(p^r-1),f,x) mod p;
  # Der Test spielt sich im von alpha_r erzeugten Unterkoerper ab
  F:=(Factors(resultant(X-alpha_r,f,x)) mod p)[2];
  if nops(F)=1 and F[1][2]=h/r then F:=F[1][1];
    else error "Das Minimalpolynom von alpha^((p^h-1)/(p^r-1) ...";
```

```
  fi;
  # f_r ist jetzt das Minimalpolynom von alpha_r mit der Variablen x_r
  f_r:=subs(X=x_r,F);
  u_r:=[];
  for i from 1 to nops(u_r_start) do
    gamma_r:=Powmod(x_r,u_r_start[i],f_r,x_r) mod p;
    ok:=1; e:=1;
    while e<(p-1)*r/h and ok=1 do
      s:=0;
      for j from 1 to nops(c) do
        s:=s+(Powmod(gamma_r,e*c[j],f_r,x_r) mod p);
      od;
      if s mod p<>0 then ok:=0; fi;
      e:=e+1;
    od;
    if ok=1 then u_r:=[op(u_r),u_r_start[i]]; fi;
  od;
  u_r;
end;

# Mit NTL
# Liefert moegliche gamma_r=x^((p^h-1)/(p^r-1)*u_r) mod f
# Eingabe: c, p, h, f, r, u_r_start
# Ausgabe: u_r - Liste mit moeglichen Exponenten
gamma_r_test_ntl:=proc()
  global gamma_r_test_tmp1, gamma_r_test_tmp2, gamma_r_test_ntl_zeit,
         u_r_ntl;
  local c, p, h, f, r, u_r_start, x, u_r, alpha_r, F, f_r, i;
  c:=args[1]; p:=args[2]; h:=args[3]; f:=args[4];
  r:=args[5]; u_r_start:=args[6];
  x:=op(indets(f));
  c:=map(t->t mod (p^r-1),c); # c[i] wird nur mod p^r-1 benutzt
  if r<2 then error "Voraussetzung ist r>=2"; fi;
  alpha_r:=Powmod(x,(p^h-1)/(p^r-1),f,x) mod p;
  # alles spielt sich im von alpha_r erzeugten Unterkoerper ab
  F:=(Factors(resultant(X-alpha_r,f,x)) mod p)[2];
  if nops(F)=1 and F[1][2]=h/r then F:=F[1][1];
    else error "Das Minimalpolynom von alpha^((p^h-1)/(p^r-1) ...";
  fi;
  f_r:=subs(X=x_r,F);
  # Ausgabe von c, p, h, r, f_r, u_r_start in Datei
  fprintf(gamma_r_test_tmp1,"[");
  for i from 1 to p do fprintf(gamma_r_test_tmp1,"%d ",c[i]); od;
  fprintf(gamma_r_test_tmp1,"]\n%d\n%d\n%d\n[",p,h,r);
  for i from 0 to r do
    fprintf(gamma_r_test_tmp1,"%d ",coeff(f_r,x_r,i));
  od;
  fprintf(gamma_r_test_tmp1,"]\n[");
  for i from 1 to nops(u_r_start) do
    fprintf(gamma_r_test_tmp1,"%d ",u_r_start[i]);
  od;
  fprintf(gamma_r_test_tmp1,"]\n");
  fclose(gamma_r_test_tmp1);
  system("gamma_r_test_ntl gamma_r_test_tmp1 gamma_r_test_tmp2");
```

```
  read gamma_r_test_tmp2;
  u_r:=u_r_ntl;
end;

# u_6_liste bestimmt bis auf Konjugation alle Kandidaten fuer u6 mit
# gamma_6=x^((p^h-1)/(p^6-1)*u6) mod f
u_6_liste:=proc()
  global gamma_r_test_ntl_zeit;
  local zeit0, c, p, h, f, x, u_1, i, zeit1, u_2, U_2, u, u_3, U_3,
        u_6, U_6, ggT, kgV, u2, j, u3;
  zeit0:=time();
  c:=args[1]; p:=args[2]; h:=args[3]; f:=args[4];
  if h mod 6<>0 then error "h ist nicht durch 6 teilbar!"; fi;
  x:=op(indets(f));
  ########################################################################
  # Erstellung der u_1-Liste
  u_1:=[];
  for i from 0 to p-2 do
    if igcd(i,p-1)=1 then u_1:=[op(u_1),i]; fi;
  od;
  u_1:=gamma_r_test(c,p,h,f,1,u_1);
  zeit1:=time();
  printf("u_1=%a (%s)\n",u_1,zeit_sec2hms(zeit1-zeit0));
  ########################################################################
  # Erstellung der u_2-Liste
  u_2:={};
  for i from 1 to nops(u_1) do
    u:=u_1[i];
    while u<p^2-1 do
      if igcd(u,p^2-1)=1 and member((p*u) mod (p^2-1),u_2)=false then
        u_2:=u_2 union {u};
      fi;
      u:=u+(p-1);
    od;
  od;
  u_2:=convert(u_2,list);
  u_2:=gamma_r_test(c,p,h,f,2,u_2);
  zeit1:=time();
  printf("u_2=%a (%s)\n",u_2,zeit_sec2hms(zeit1-zeit0));
  ########################################################################
  # Erstellung der u_3-Liste
  u_3:={};
  for i from 1 to nops(u_1) do
    u:=u_1[i];
    while u<p^3-1 do
      if igcd(u,p^3-1)=1 and member((p*u) mod (p^3-1),u_3)=false
                         and member((p^2*u) mod (p^3-1),u_3)=false then
        u_3:=u_3 union {u};
      fi;
      u:=u+(p-1);
    od;
  od;
  u_3:=convert(u_3,list);
  u_3:=gamma_r_test(c,p,h,f,3,u_3);
```

```
  zeit1:=time();
  printf("u_3=%a (%s)\n",u_3,zeit_sec2hms(zeit1-zeit0));
  ######################################################################
  # Erstellung der u_6-Liste
  u_6:={}; ggT:=igcd(p^2-1,p^3-1); kgV:=ilcm(p^2-1,p^3-1);
  for i from 1 to nops(u_2) do
    u2:=u_2[i];
    for j from 1 to nops(u_3) do
      u3:=u_3[j];
      if (u2-u3) mod ggT=0 then
        u:=chrem_ex2([u2,u3],[p^2-1,p^3-1]) mod kgV;
        while u<p^6-1 do
          if igcd(u,p^6-1)=1 and member((u*p) mod (p^6-1),u_6)=false
                             and member((u*p^2) mod (p^6-1),u_6)=false
                             and member((u*p^3) mod (p^6-1),u_6)=false
                             and member((u*p^4) mod (p^6-1),u_6)=false
                             and member((u*p^5) mod (p^6-1),u_6)=false
            then u_6:=u_6 union {u};
          fi;
          u:=u+kgV;
        od;
      fi;
    od;
  od;
  u_6:=convert(u_6,list);
  printf("Startliste fuer u_6 bestimmt (%d Elemente)! ",nops(u_6));
  zeit1:=time(); printf("(%s)\n",zeit_sec2hms(zeit1-zeit0));
  u_6:=gamma_r_test(c,p,h,f,6,u_6);
  #u_6:=gamma_r_test_ntl(c,p,h,f,6,u_6);
  #zeit0:=zeit0-gamma_r_test_ntl_zeit;
  zeit1:=time(); printf("(%s)\n",zeit_sec2hms(zeit1-zeit0));
  u_6;
end;

# Eingabe: Oeffentlicher Schluessel (c,p,h) oder (c,p,h),f oder
# (c,p,h),f,u6 mit h=12
angriff_h12:=proc()
  local zeit0, k_pub, c, p, h, f, x, u_6, r, u_r, gamma_r, ci,
        gamma_r_c, m0, m1, m2, i, mm0, mm1, mm2, gg, R, RRi, Ri, j, pi,
        Q_6, alpha, Q_6wurzeln, xi, F, i0, i1, gamma, zeit1, d, k_priv;
  global gamma_r_test_ntl_zeit, dlog_zeit, ntl_zeit;
  zeit0:=time(); ntl_zeit:=0;
  k_pub:=args[1];
  c:=k_pub[1]; p:=k_pub[2]; h:=k_pub[3];
  if h<>12 then error "h<>12"; fi;
  ######################################################################
  # Wahl eines Polynoms f, so dass x modulo f Ordnung p^h-1 hat
  if nargs>=2 then
    f:=args[2];
  else
    f:=fph_gen(p,h);
  fi;
  x:=op(indets(f));
  ######################################################################
```

```
# u_6-Liste
if nargs>=3 then
  u_6:=args[3];
else
  u_6:=u_6_liste(c,p,h,f);
fi;
#########################################################################
# Bestimmung der Koeffizienten m0[i], m1[i], m2[i] mit
# gamma_6_c[i]=gamma_6_c[0]*m0[i]+...+gamma_6_c[2]*m2[i]
if nops(u_6)>1 then
  printf("u_6 hat %d Elemente. Nur u_6[1] wird benutzt.\n",nops(u_6));
fi;
r:=6; u_r:=u_6;
gamma_r:=Powmod(x,(p^h-1)/(p^r-1)*u_r[1],f,x) mod p;
ci:=array(0..p-1); gamma_r_c:=array(0..p-1);
m0:=array(0..p-1); m1:=array(0..p-1); m2:=array(0..p-1);
for i from 0 to p-1 do
  ci[i]:=c[i+1];
  gamma_r_c[i]:=Powmod(gamma_r,ci[i],f,x) mod p;
od;
for i from 0 to p-1 do
  mm0:='mm0'; mm1:='mm1'; mm2:='mm2';
  gg:={coeffs(expand(gamma_r_c[i]-mm0*gamma_r_c[0]-mm1*gamma_r_c[1]
                     -mm2*gamma_r_c[2]),x)} mod p;
  assign(msolve(gg,p));
  m0[i]:=mm0; m1[i]:=mm1; m2[i]:=mm2;
od;
#########################################################################
# Bestimmung von pi mit pi[0]=0, pi[1]=1, xi=pi[i]
# Mit m0[i], m1[i], m2[i] hat man viele Gleichungen fuer x2=pi[2]
R:={seq(i,i=0..p-1)};
for i from 3 to p-1 do
  RRi:=Roots((m1[i]^2-m1[i])+2*m1[i]*m2[i]*x2+(m2[i]^2-m2[i])*x2^2)
       mod p;
  Ri:={};
  for j from 1 to nops(RRi) do
    Ri:=Ri union {RRi[j][1]};
  od;
  R:=R intersect Ri;
  if nops(R)<=1 then break; fi;
od;
pi:=array(0..p-1);
pi[0]:=0; pi[1]:=1; pi[2]:=op(R);
for i from 3 to p-1 do
  pi[i]:=(m1[i]+m2[i]*pi[2]) mod p;
od;
if nops(convert(pi,set))<>p then error "pi stimmt nicht!"; fi;
#########################################################################
Q_6:=gamma_r_c[0]*(X-pi[1])*(X-pi[2])/(pi[0]-pi[1])/(pi[0]-pi[2])
    +gamma_r_c[1]*(X-pi[0])*(X-pi[2])/(pi[1]-pi[0])/(pi[1]-pi[2])
    +gamma_r_c[2]*(X-pi[0])*(X-pi[1])/(pi[2]-pi[0])/(pi[2]-pi[1]);
#########################################################################
# Q_6 hat -xi und seine Konjugierten ueber F_p^6 als Nullstellen in
# F_p^h. So wird xi berechnet.
```

```
  alpha:=RootOf(f,x);
  Q_6:=subs(x=alpha,Q_6);
  Q_6wurzeln:=Roots(Q_6,alpha) mod p;
  if nops(Q_6wurzeln)<>h/6 then error "Fehler!"; fi;
  xi:=(-Q_6wurzeln[1][1]) mod p;
  xi:=subs(alpha=x,xi);
  # Das Minimalpolynom von xi wird F
  F:=resultant(X-xi,f,x) mod p; F:=subs(X=x,F);
  # Von hier an entspricht xi der Variablen x und es ist modulo F zu
  # rechnen
  i0:=0; i1:=1;
  while igcd(ci[i0]-ci[i1],p^h-1)>1 do
    i1:=i1+1;
    if i1=p then i0:=i0+1; i1:=i0+1; fi;
    if i1>=p then error "Methode funktioniert so nicht!"; fi;
  od;
  gamma:=Powmod(Rem((x+pi[i0])*inv(x+pi[i1],F,p),F,x) mod p,
                1/(ci[i0]-ci[i1]) mod (p^h-1),F,x) mod p;
  zeit1:=time(); printf("Zeit: %s\n",zeit_sec2hms(zeit1-zeit0));
  printf("Logarithmenberechnung: log_gamma(xi)\n");
  d:=(ci[0]-dislog(x,gamma,F,p)) mod (p^h-1);
  # k_priv ist der gefundene private Schluessel
  k_priv:=[F,gamma,convert(pi,list),d];
  # Nun wird noch getestet, ob k_pub und k_priv zusammenpassen
  if k_pub_priv_test(k_pub,k_priv)=false then error "Fehler!"; fi;
  zeit1:=time();
  printf("Zeit: %s\n",zeit_sec2hms(zeit1-zeit0+ntl_zeit));
  k_priv;
end;

# Chinesischer Restsatz fuer x=a1 mod m1, x=a2 mod m2 ohne
# die Voraussetzung ggT(m1,m2)=1
chrem_ex2:=proc()
  local a, m, a1, a2, m1, m2, m12, aa, mm, pe, i, q;
  a:=args[1]; m:=args[2];
  a1:=a[1]; a2:=a[2]; m1:=m[1]; m2:=m[2];
  if (a1-a2) mod igcd(m1,m2)<>0 then return []; fi;
  m12:=ilcm(m1,m2); aa:=[]; mm:=[];
  pe:=ifactors(m12)[2];
  for i from 1 to nops(pe) do
    q:=pe[i][1]^pe[i][2];
    if m1 mod q=0 then
      aa:=[op(aa),a1 mod q]; mm:=[op(mm),q];
    else
      aa:=[op(aa),a2 mod q]; mm:=[op(mm),q];
    fi;
  od;
  [aa,mm];
  chrem(aa,mm);
end;

# Berechnung von 1/g mod f ueber F_p
# Eingabe: g,f,p
inv:=proc()
```

```
  local g, f, p, x, ggT, s, t;
  g:=args[1]; f:=args[2]; p:=args[3]; x:=op(indets(f));
  ggT:=Gcdex(f,g,x,'s','t') mod p;
  t/ggT mod p;
end;

# Zeitumwandlung: sec -> hms
# Eingabe: t (in Sekunden)
zeit_sec2hms:=proc()
  local t, st, mi, se, z;
  t:=round(args[1]);
  st:=iquo(t,3600);
  t:=t-3600*st;
  mi:=iquo(t,60);
  t:=t-60*mi;
  se:=t;
  z:="";
  if st<10 then z:=cat(z,"0"); fi;
  z:=cat(z,convert(st,string),":");
  if mi<10 then z:=cat(z,"0"); fi;
  z:=cat(z,convert(mi,string),":");
  if se<10 then z:=cat(z,"0"); fi;
  z:=cat(z,convert(se,string));
end;

# k_priv2pub berechnet aus einem privaten Chor-Rivest-Schluessel
# (f(x),g(x),pi,d) den zugehoerigen oeffentlichen Schluessel (c,p,h)
# Eingabe: (f,g,pi,d)
# Ausgabe: (c,p,h)
k_priv2pub:=proc()
  local k_priv, f, g, pi_liste, d, x, p, h, pi, i, a, c;
  k_priv:=args[1];
  f:=k_priv[1]; g:=k_priv[2]; pi_liste:=k_priv[3]; d:=k_priv[4];
  printf("f=%a\n",f);
  printf("g=%a\n",g);
  printf("pi=%a\n",pi_liste);
  printf("d=%d\n",d);
  x:=op(indets(f)); p:=nops(pi_liste); h:=degree(f,x);
  # Permutation
  pi:=array(0..p-1);
  for i from 0 to p-1 do pi[i]:=pi_liste[i+1]; od;
  a:=array(0..p-1); c:=array(0..p-1);
  for i from 0 to p-1 do
    a[i]:=dislog(x+i,g,f,p);
    printf("a_%d=%d\n",i,a[i]);
  od;
  for i from 0 to p-1 do
    c[i]:=(a[pi[i]]+d) mod (p^h-1);
  od;
  [convert(c,list),p,h];
end;

# k_pub_priv_test testet, ob oeffentlicher und privater Schluessel
# zueinander passen.
```

```
# Eingabe: k_pub, k_priv
k_pub_priv_test:=proc()
  local k_pub, k_priv, p, h, m, M1, c, M2;
  k_pub:=args[1]; k_priv:=args[2];
  p:=k_pub[2]; h:=k_pub[3];
  # Getestet werden einige kleine Zahlen
  for m from 0 to min(10,binomial(p,h)-1) do
    M1:=T_ph(m,p,h);
    c:=choriv_en0(M1,k_pub);
    M2:=choriv_de0(c,k_priv);
    if M1<>M2 then return false; fi;
  od;
  true;
end;

# Eingabe: Oeffentlicher Schluessel (c,p,h) oder (c,p,h),f mit h=24
# oder (c,p,h),f_p_24,u_6_p_24
angriff_h24:=proc()
  local zeit0, k_pub, c, p, h, f, x, u_6, alpha_6, F, f_6, gamma_6,
        gamma_6_c, i, m0, m1, m2, m3, m4, mm0, mm1, mm2, mm3, mm4, gg,
        g0, h1, k2, Wx2, g0i, h1i, k2i, r02, r12, r0212, RRx2, Rx2, j,
        Wx3, RRx3, Rx3, Wx4, RRx4, Rx4, pi, ci, Q_6, alpha, Q_6wurzeln,
        xi, i0, i1, gamma, zeit1, d, k_priv;
  global ntl_zeit;
  zeit0:=time(); ntl_zeit:=0;
  k_pub:=args[1];
  c:=k_pub[1]; p:=k_pub[2]; h:=k_pub[3];
  if h<>24 then error "h<>24"; fi;
  # Wahl eines Polynoms f, so dass x modul f Ordnung p^h-1 hat
  if nargs>=2 then
    f:=args[2];
  else
    f:=fph_gen(p,h);
  fi;
  x:=op(indets(f));
  ########################################################################
  # Bestimmung von Kandidaten fuer u_6 bzw. gamma_6
  if nargs>=3 then
    u_6:=args[3];
  else
    u_6:=u_6_liste(c,p,h,f);
  fi;
  ########################################################################
  # Bestimmung der Koeffizienten m
  # gamma_6^c_i=m_0i*gamma_6^c_0+...+m_4i*gamma_6^c_4
  # Da sich alles in F_p^6 abspielt, rechnen wir dort
  alpha_6:=Powmod(x,(p^h-1)/(p^6-1),f,x) mod p;
  F:=(Factors(resultant(X-alpha_6,f,x)) mod p)[2];
  if nops(F)=1 and F[1][2]=4 then F:=F[1][1];
    else error "Das Minimalpolynom von alpha^((p^h-1)/(p^6-1) ...";
  fi;
  f_6:=subs(X=x_6,F);
  gamma_6:=Powmod(x_6,u_6[1],f_6,x_6) mod p;
  gamma_6_c:=array(0..p-1);
```

```
for i from 0 to p-1 do
  gamma_6_c[i]:=Powmod(gamma_6,c[i+1],f_6,x_6) mod p;
od;
m0:=array(0..p-1); m1:=array(0..p-1); m2:=array(0..p-1);
m3:=array(0..p-1); m4:=array(0..p-1);

for i from 0 to p-1 do
  mm0:='mm0'; mm1:='mm1'; mm2:='mm2'; mm3:='mm3'; mm4:='mm4';
  gg:={coeffs(expand(gamma_6_c[i]-mm0*gamma_6_c[0]-mm1*gamma_6_c[1]
     -mm2*gamma_6_c[2]-mm3*gamma_6_c[3]-mm4*gamma_6_c[4]),x_6)} mod p;
  assign(msolve(gg,p));
  m0[i]:=mm0; m1[i]:=mm1; m2[i]:=mm2; m3[i]:=mm3; m4[i]:=mm4;
od;
#########################################################################
# Bestimmung der Permutation pi mit pi[0]=0, pi[1]=1, xi=pi[i]
# Es gibt Gleichungen fuer x2,x3,x4: g0i=h1i=k2i=0

g0:=m4i^2*x4^2*x3-m1i*x2*x4-m1i*x2*x3-m2i*x2^2*x4-m2i*x2^2*x3
-3*m4i^2*x4^2*m2i*x2-3*m4i^2*x4^2*m3i*x3-3*m2i^2*x2^2*m4i*x4
+2*m1i*x2*m3i*x3+2*m1i*x2*m4i*x4+2*m2i*x2^2*m3i*x3+2*m2i*x2^2*m4i*x4
-3*m2i^2*x2^2*m3i*x3-3*m2i*x2*m3i^2*x3^2+2*m2i*x2*m3i*x3^2
-3*m3i^2*x3^2*m4i*x4+2*m3i*x3^2*m4i*x4+2*m1i*m2i*x2*x3
+12*m4i^2*x4^2*m2i*x2*m3i*x3+12*m2i^2*x2^2*m4i*x4*m3i*x3
-6*m1i*x2^2*m3i*x3*m2i-6*m1i*x2*m3i*x3*m4i*x4-6*m1i*x2^2*m4i*x4*m2i
-6*m2i*x2^2*m3i*x3*m4i*x4-3*m2i^2*x2^3*m4i*x4-3*m2i*x2^2*m4i^2*x4^2
+2*m2i*x2^2*m4i*x4^2+4*m2i^3*x2^3*m3i*x3+6*m2i^2*x2^2*m3i^2*x3^2
+4*m2i*x2*m3i^3*x3^3-3*m2i^2*x2^2*m3i*x3^2-3*m2i*x2*m3i^2*x3^3
+4*m3i^3*x3^3*m4i*x4-3*m3i^2*x3^2*m4i*x4^2-3*m3i^2*x3^3*m4i*x4
+2*m3i*x3^2*m4i*x4^2-m4i^3*x4^3*x3+m4i^2*x4^3*x3+m1i^2*x2*x3
+m2i^2*x2^3*x3+6*m1i^2*m2i^2*x2^2+4*m1i*m2i^3*x2^3+4*m1i^3*m3i*x3
+6*m1i^2*m3i^2*x3^2+4*m1i*m3i^3*x3^3-3*m1i^2*m3i*x3^2-3*m1i*m3i^2*x3^3
+4*m1i^3*m4i*x4+6*m1i^2*m4i^2*x4^2-3*m1i^2*m4i*x4^2-m3i^3*x2*x3^3
-m4i^3*x2*x4^3+m4i^2*x2*x4^3+2*m1i*m4i*x4^2-3*m1i^2*x2^2*m2i
-3*m1i*x2^3*m2i^2+4*m4i^3*x4^3*m1i-3*m4i^2*x4^3*m1i-m2i^3*x2^3*x3
+m3i^2*x2*x3^3-3*m1i^2*m2i*x2+m1i^4-6*m2i*x2*m4i*x4*m3i*x3
-3*m1i*m2i^2*x2^2-3*m1i^2*m3i*x3-3*m1i*m3i^2*x3^2+2*m1i*m3i*x3^2
-3*m1i^2*m4i*x4-6*m2i*x2*m4i*x4*m1i+2*m2i*x2*m4i*x4*x3+2*m1i*m4i*x4*x3
+2*m2i*x2*m4i*x4^2-3*m4i^2*x4^2*x3^2*m3i+m4i^2*x2*x3*x4^2
-m4i*x2*x3*x4^2+2*m1i*x2^2*x3*m2i+2*m1i*x2*x3^2*m3i+2*m2i*x2^2*x3^2*m3i
+6*m4i^2*x4^2*m2i^2*x2^2+4*m4i^3*x4^3*m2i*x2-3*m4i^2*x4^3*m2i*x2
+6*m4i^2*x4^2*m3i^2*x3^2+4*m4i^3*x4^3*m3i*x3-3*m4i^2*x4^3*m3i*x3
+4*m2i^3*x2^3*m4i*x4-3*m2i^2*x2^2*m4i*x4^2-3*m1i^2*x2*m3i*x3
-3*m1i*x2*m3i^2*x3^2-3*m1i^2*x2*m4i*x4-3*m1i*x2*m4i^2*x4^2
+2*m1i*x2*m4i*x4^2-3*m2i^2*x2^3*m3i*x3-3*m2i*x2^2*m3i^2*x3^2
+2*m4i*x2^2*x3*x4*m2i+2*m4i*x2*x3^2*x4*m3i-m1i^3*x4+m1i^2*x4
+m4i^4*x4^4-m4i^3*x4^4-m1i^3*x2+m3i^2*x2*x3^2+m4i^2*x2*x4^2
-m1i^3-3*m4i^2*x4^2*x3*m2i*x2-6*m1i*m2i*x2*m3i*x3-6*m1i*m3i*x3*m4i*x4
+2*m1i*x2^2*m2i+2*m3i*x2*x3*m4i*x4-3*m4i^2*x4^2*m1i+m2i^2*x2^2*x3
-m4i^3*x4^3+m1i^2*x2+m1i^2*x3+m2i^2*x2^3-m2i^3*x2^3-m3i^3*x3^3
+m3i^2*x3^3+m4i^2*x4^3-m3i*x3^2*x4-m3i*x2*x3^2-6*m1i*m2i*x2*m3i*x3*x4
-m3i^3*x3^3*x4+m3i^2*x3^3*x4+m2i^2*x2^2*x4+m3i^2*x3^2*x4-m2i^3*x2^3*x4
+12*m4i^2*x4^2*m2i*x2*m1i+12*m4i^2*x4^2*m3i*x3*m1i
+12*m2i^2*x2^2*m4i*x4*m1i+2*m1i*x2*m3i*x3*x4+2*m2i*x2^2*m3i*x3*x4
+12*m2i^2*x2^2*m3i*x3*m1i-3*m2i^2*x2^2*m3i*x3*x4
```

```
+12*m2i*x2*m3i^2*x3^2*m1i-3*m2i*x2*m3i^2*x3^2*x4-6*m2i*x2*m3i*x3^2*m1i
+2*m2i*x2*m3i*x3^2*x4+12*m3i^2*x3^2*m4i*x4*m1i-6*m3i*x3^2*m4i*x4*m1i
+2*m1i*m2i*x2*x3*x4+24*m2i*x2*m4i*x4*m3i*x3*m1i+2*m4i*x2*x3*x4*m1i
+2*m2i*x2*m3i*x3*x4+m1i^2*x3*x4+m2i^2*x2^3*x4+2*m3i*x2*x3*m4i*x4^2
-m1i^3*x3-m2i^3*x2^4+m2i^4*x2^4+m3i^4*x3^4-m3i^3*x3^4
-6*m1i*m3i*x3*m4i*x4^2+2*m3i*x3*m4i*x4^2-3*m4i^2*x4^2*x3*m1i
-m2i*x2^2*x3*x4-3*m1i^2*m2i*x2*x4-3*m1i*m2i^2*x2^2*x4-3*m1i^2*m3i*x3*x4
-3*m1i*m3i^2*x3^2*x4+2*m1i*m3i*x3^2*x4+m3i^2*x2*x3^2*x4+2*m1i*m2i*x2*x4
+2*m1i*m3i*x3*x4+2*m1i*x2^2*m2i*x4+m2i^2*x2^2*x3*x4-m3i*x2*x3^2*x4
-6*m1i*m2i*x2*x3*m4i*x4+12*m1i^2*m2i*x2*m3i*x3+12*m1i^2*m2i*x2*m4i*x4
-6*m2i*x2*m4i*x4^2*m3i*x3+12*m1i^2*m3i*x3*m4i*x4-6*m2i*x2*m4i*x4^2*m1i
-3*m2i^2*x2^2*m4i*x4*x3+2*m2i*x2*m4i*x4^2*x3-3*m3i^2*x2*x3^2*m4i*x4
-3*m4i^2*x2*x4^2*m3i*x3+12*m2i*x2*m3i^2*x3^2*m4i*x4
-6*m2i*x2*m3i*x3^2*m4i*x4-3*m1i^2*m2i*x2*x3-3*m1i*m2i^2*x2^2*x3
-3*m1i^2*m4i*x4*x3+2*m1i*m4i*x4^2*x3+4*m1i^3*m2i*x2+m1i^2*x2*x4
-m1i*x3*x4-m4i*x4^2*x2-m4i*x4^2*x3;
h1:=-m1i-m4i^2*x4^2*x3+m1i*x4+m1i*x3+m1i*x2+m2i*x2^2*x4+m2i*x2^2*x3
+3*m4i^2*x4^2*m2i*x2+3*m4i^2*x4^2*m3i*x3+3*m2i^2*x2^2*m4i*x4
-2*m1i*x2*m3i*x3-2*m1i*x2*m4i*x4-2*m2i*x2^2*m3i*x3-2*m2i*x2^2*m4i*x4
+3*m2i^2*x2^2*m3i*x3+3*m2i*x2*m3i^2*x3^2-2*m2i*x2*m3i*x3^2
+3*m3i^2*x3^2*m4i*x4-2*m3i*x3^2*m4i*x4-2*m1i*m2i*x2*x3-2*m1i*m4i*x4^2
+3*m1i^2*m2i*x2+6*m2i*x2*m4i*x4*m3i*x3+3*m1i*m2i^2*x2^2+3*m1i^2*m3i*x3
+3*m1i*m3i^2*x3^2-2*m1i*m3i*x3^2+3*m1i^2*m4i*x4+6*m2i*x2*m4i*x4*m1i
-2*m2i*x2*m4i*x4*x3-2*m1i*m4i*x4*x3-2*m2i*x2*m4i*x4^2-m1i^2*x4
-m3i^2*x2*x3^2-m4i^2*x2*x4^2+m1i^3+6*m1i*m2i*x2*m3i*x3
+6*m1i*m3i*x3*m4i*x4-2*m1i*x2^2*m2i-2*m3i*x2*x3*m4i*x4
+3*m4i^2*x4^2*m1i-m2i^2*x2^2*x3+m4i^3*x4^3-m1i^2*x2-m1i^2*x3
-m2i^2*x2^3+m2i^3*x2^3+m3i^3*x3^3-m3i^2*x3^3-m4i^2*x4^3+m3i*x3^2*x4
+m3i*x2*x3^2-m2i^2*x2^2*x4-m3i^2*x3^2*x4-2*m2i*x2*m3i*x3*x4
-2*m3i*x3*m4i*x4^2-2*m1i*m2i*x2*x4-2*m1i*m3i*x3*x4+m4i*x4^2*x2
+m4i*x4^2*x3;
k2:=-m1i+m4i^2*x4^2+2*m2i*x2*m4i*x4-m4i*x4^2+2*m1i*m2i*x2
+2*m1i*m3i*x3+2*m1i*m4i*x4+m1i^2-m2i*x2^2+2*m2i*x2*m3i*x3-m3i*x3^2
+m2i^2*x2^2+m3i^2*x3^2+2*m3i*x3*m4i*x4;

Wx2:={seq(i,i=0..p-1)}; # Moegliche Werte fuer x2
for i from 5 to p-1 do
  g0i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],g0) mod p;
  h1i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],h1) mod p;
  k2i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],k2) mod p;
  r02:=resultant(g0i,k2i,x4) mod p;
  r12:=resultant(h1i,k2i,x4) mod p;
  r0212:=resultant(r02,r12,x3) mod p; # Polynom in x2
  if r0212<>0 then
    RRx2:=Roots(r0212) mod p; Rx2:={};
    for j from 1 to nops(RRx2) do
      Rx2:=Rx2 union {RRx2[j][1]};
    od;
    Wx2:=Wx2 intersect Rx2;
  fi;
  if nops(Wx2)=1 then break; fi;
od;
# Wx2 enthaelt den einzig moeglichen Wert fuer x2
```

```
g0:=subs(x2=op(Wx2),g0) mod p;
h1:=subs(x2=op(Wx2),h1) mod p;
k2:=subs(x2=op(Wx2),k2) mod p;

Wx3:={seq(i,i=0..p-1)}; # Moegliche Werte fuer x3
for i from 5 to p-1 do
  g0i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],g0) mod p;
  h1i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],h1) mod p;
  k2i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],k2) mod p;
  r02:=resultant(g0i,k2i,x4) mod p; # Polynom in x3
  r12:=resultant(h1i,k2i,x4) mod p; # Polynom in x3
  if r02<>0 then
    RRx3:=Roots(r02) mod p; Rx3:={};
    for j from 1 to nops(RRx3) do
      Rx3:=Rx3 union {RRx3[j][1]};
    od;
    Wx3:=Wx3 intersect Rx3;
  fi;
  if r12<>0 then
    RRx3:=Roots(r12) mod p; Rx3:={};
    for j from 1 to nops(RRx3) do
      Rx3:=Rx3 union {RRx3[j][1]};
    od;
    Wx3:=Wx3 intersect Rx3;
  fi;
  if nops(Wx3)=1 then break; fi;
od;
# Wx3 enthaelt den einzig moeglichen Wert fuer x3

g0:=subs(x3=op(Wx3),g0) mod p;
h1:=subs(x3=op(Wx3),h1) mod p;
k2:=subs(x3=op(Wx3),k2) mod p;

Wx4:={seq(i,i=0..p-1)}; # Moegliche Werte fuer x4
for i from 0 to p-1 do
  g0i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],g0) mod p;
  h1i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],h1) mod p;
  k2i:=subs(m1i=m1[i],m2i=m2[i],m3i=m3[i],m4i=m4[i],k2) mod p;
  if g0i<>0 then
    RRx4:=Roots(g0i) mod p; Rx4:={};
    for j from 1 to nops(RRx4) do
      Rx4:=Rx4 union {RRx4[j][1]};
    od;
    Wx4:=Wx4 intersect Rx4;
  fi;
  if h1i<>0 then
    RRx4:=Roots(h1i) mod p; Rx4:={};
    for j from 1 to nops(RRx4) do
      Rx4:=Rx4 union {RRx4[j][1]};
    od;
    Wx4:=Wx4 intersect Rx4;
  fi;
  if k2i<>0 then
    RRx4:=Roots(k2i) mod p; Rx4:={};
```

```
    for j from 1 to nops(RRx4) do
      Rx4:=Rx4 union {RRx4[j][1]};
    od;
    Wx4:=Wx4 intersect Rx4;
  fi;
  if nops(Wx4)=1 then break; fi;
od;
# Wx4 enthaelt den einzig moeglichen x4-Wert

pi:=array(0..p-1);
pi[0]:=0; pi[1]:=1;
pi[2]:=op(Wx2); pi[3]:=op(Wx3); pi[4]:=op(Wx4);
for i from 5 to p-1 do
  pi[i]:=(m1[i]+m2[i]*pi[2]+m3[i]*pi[3]+m4[i]*pi[4]) mod p;
od;
if nops(convert(pi,set))<>p then error "pi stimmt nicht!"; fi;
#############################################################################

ci:=array(0..p-1);
for i from 0 to p-1 do ci[i]:=c[i+1]; od;

gamma_6:=Powmod(x,(p^h-1)/(p^6-1)*u_6[1],f,x) mod p;
for i from 0 to p-1 do
  gamma_6_c[i]:=Powmod(gamma_6,ci[i],f,x) mod p;
od;

Q_6:=gamma_6_c[0]*(X-pi[1])*(X-pi[2])*(X-pi[3])*(X-pi[4])/
     ((pi[0]-pi[1])*(pi[0]-pi[2])*(pi[0]-pi[3])*(pi[0]-pi[4]))
    +gamma_6_c[1]*(X-pi[0])*(X-pi[2])*(X-pi[3])*(X-pi[4])/
     ((pi[1]-pi[0])*(pi[1]-pi[2])*(pi[1]-pi[3])*(pi[1]-pi[4]))
    +gamma_6_c[2]*(X-pi[0])*(X-pi[1])*(X-pi[3])*(X-pi[4])/
     ((pi[2]-pi[0])*(pi[2]-pi[1])*(pi[2]-pi[3])*(pi[2]-pi[4]))
    +gamma_6_c[3]*(X-pi[0])*(X-pi[1])*(X-pi[2])*(X-pi[4])/
     ((pi[3]-pi[0])*(pi[3]-pi[1])*(pi[3]-pi[2])*(pi[3]-pi[4]))
    +gamma_6_c[4]*(X-pi[0])*(X-pi[1])*(X-pi[2])*(X-pi[3])/
     ((pi[4]-pi[0])*(pi[4]-pi[1])*(pi[4]-pi[2])*(pi[4]-pi[3]));

# Q_6 hat -xi und seine Konjugierten ueber F_p^6 als Nullstellen in
# F_p^h. So wird xi berechnet.
alpha:=RootOf(f,x);
Q_6:=subs(x=alpha,Q_6);
Q_6wurzeln:=Roots(Q_6,alpha) mod p;
if nops(Q_6wurzeln)<>h/6 then error "Fehler!"; fi;
xi:=(-Q_6wurzeln[1][1]) mod p;
xi:=subs(alpha=x,xi);

# Das Minimalpolynom von xi wird F
F:=resultant(X-xi,f,x) mod p; F:=subs(X=x,F);
# Von hier an entspricht xi der Variablen x und es ist modulo F zu
# rechnen

i0:=0; i1:=1;
while igcd(ci[i0]-ci[i1],p^h-1)>1 do
  i1:=i1+1;
```

```
    if i1=p then i0:=i0+1; i1:=i0+1; fi;
    if i1>=p then error "Methode funktioniert so nicht!"; fi;
  od;

  gamma:=Powmod(Rem((x+pi[i0])*inv(x+pi[i1],F,p),F,x) mod p,
                1/(ci[i0]-ci[i1]) mod (p^h-1),F,x) mod p;
  zeit1:=time(); printf("Zeit: %s\n",zeit_sec2hms(zeit1-zeit0));
  printf("Logarithmenberechnung: log_gamma(xi)\n");
  d:=(ci[0]-dislog(x,gamma,F,p)) mod (p^h-1);

  # Nun ist der private Schluessel bestimmt
  k_priv:=[F,gamma,convert(pi,list),d];
  # Jetzt wird getestet, ob k_pub und k_priv zusammenpassen
  if k_pub_priv_test(k_pub,k_priv)=false then
    error "k_pub und k_priv passen nicht zusammen!";
  fi;
  zeit1:=time();
  printf("Zeit: %s\n",zeit_sec2hms(zeit1-zeit0+ntl_zeit));
  k_priv;
end;

# Normalisierung der Permutation
pi_norm:=proc()
  local pi_liste, p, pi, i, pi_n;
  pi_liste:=args[1];
  p:=nops(pi_liste);
  pi:=array(0..p-1);
  for i from 0 to p-1 do
    pi[i]:=pi_liste[i+1];
  od;
  pi_n:=array(0..p-1);
  for i from 0 to p-1 do
    pi_n[i]:=(pi[i]-pi[0])/(pi[1]-pi[0]) mod p;
  od;
  convert(pi_n,list);
end;
```

4.2. dlog_naiv_ntl.c.

```
/* dlog_naiv_ntl.c
   Version: 9.7.2002
   Uebersetzung: g++ dlog_naiv_ntl.c -o dlog_naiv_ntl -lntl -lgmp
   Eingabe: p, f(x), g(x), a(x)
   Berechnet wird log_g(a) in F_p[x]/(f)
   Eingabe: p [f0 f1 ... fh] [g0 g1 ...] [a0 a1 ...]
*/

#include <fstream.h>
#include <NTL/ZZ_pX.h>

int main( int argc, char *argv[])
{
  if (argc!=3)
  {
    cout<<"Aufruf: 'dlog_ntl Eingabedatei Ausgabedatei'\n";
    return 0;
```

```
  }
  ifstream(ein); ein.open(argv[1]);
  ofstream(aus); aus.open(argv[2]);

  ZZ p, n;
  ein>>p;

  ZZ_p::init(p);

  ZZ_pX f, g, a, gn(0,1);
  ein>>f; ein>>g; ein>>a;

  ZZ_pXModulus F(f);
  ZZ_pXMultiplier G(g,F);

  while (gn!=a)
  {
    n++;
    MulMod(gn,gn,G,F);
  }

  aus<<"dlog:="<<n<<";\n";

  double zeit=GetTime();
  aus<<"dlog_zeit:="<<zeit<<";\n";

  return 0;
}
```

4.3. dlog_pollard_ntl.c.

```
/* dlog_pollard_ntl.c
   Version: 8.7.2002
   Uebersetzung: g++ dlog_pollard_ntl.c -o dlog_pollard_ntl -lntl -lgmp
   Eingabe: p, f(x), g(x), k(x), n, wobei n die Ordnung von g(x) mod
            f(x) ist. Ausserdem sollte k(x)^n=1 mod f(x) gelten.
   Eingabeform: p [f0 f1 ... fh] [g0 g1 ...] [k0 k1 ...] n
   Ausgegeben wird z mit g(x)^z=k(x) mod f(x) oder ein z<0, falls das
   Verfahren nicht funktioniert.
   (Mit 'modular arithmetic with pre-conditioning' war das Programm
   langsamer als in der vorliegenden Fassung.)
*/

#include <fstream.h>
#include <NTL/ZZ_pX.h>

int main( int argc, char *argv[])
{
  if (argc!=3)
  {
    cout<<"Aufruf: 'dlog_pollard_ntl Eingabedatei Ausgabedatei'\n";
    return 0;
  }
  ifstream(ein); ein.open(argv[1]);
  ofstream(aus); aus.open(argv[2]);
```

```
  ZZ p, n, z, a_i, a_i1, a_2i, a_2i1, a_2i2, b_i, b_i1, b_2i, b_2i1,
     b_2i2;

  ein>>p;
  ZZ_p::init(p);

  ZZ_pX f, g, k, x_i(0,1), x_i1, x_2i(0,1), x_2i1, x_2i2;

  ein>>f>>g>>k>>n;

  do
  {
    switch(rep(ConstTerm(x_i))%3)
    {
      case 0: x_i1=(x_i*x_i)%f; a_i1=(2*a_i)%n; b_i1=(2*b_i)%n;
        break;
      case 1: x_i1=(k*x_i)%f; a_i1=a_i; b_i1=(b_i+1)%n;
        break;
      case 2: x_i1=(g*x_i)%f; a_i1=(a_i+1)%n; b_i1=b_i;
        break;
    }
    switch(rep(ConstTerm(x_2i))%3)
    {
      case 0: x_2i1=(x_2i*x_2i)%f; a_2i1=(2*a_2i)%n; b_2i1=(2*b_2i)%n;
        break;
      case 1: x_2i1=(k*x_2i)%f; a_2i1=a_2i; b_2i1=(b_2i+1)%n;
        break;
      case 2: x_2i1=(g*x_2i)%f; a_2i1=(a_2i+1)%n; b_2i1=b_2i;
        break;
    }
    switch(rep(ConstTerm(x_2i1))%3)
    {
      case 0: x_2i2=(x_2i1*x_2i1)%f; a_2i2=(2*a_2i1)%n; b_2i2=(2*b_2i1)%n;
        break;
      case 1: x_2i2=(k*x_2i1)%f; a_2i2=a_2i1; b_2i2=(b_2i1+1)%n;
        break;
      case 2: x_2i2=(g*x_2i1)%f; a_2i2=(a_2i1+1)%n; b_2i2=b_2i1;
        break;
    }
    x_i=x_i1; a_i=a_i1; b_i=b_i1;
    x_2i=x_2i2; a_2i=a_2i2; b_2i=b_2i2;
  } while (x_i!=x_2i);

  if (GCD(b_i-b_2i,n)>1) z=-GCD(b_i-b_2i,n);
    else z=(InvMod((b_i-b_2i)%n,n)*(a_2i-a_i))%n;

  aus<<"dlog:="<<z<<";\n";

  double zeit=GetTime();
  aus<<"dlog_zeit:="<<zeit<<";\n";

  return 0;
}
```

4.4. gamma_r_test_ntl.c.

```
/* gamma_r_test_ntl.c
   Version: 26.6.2002
   Uebersetzung: g++ gamma_r_test_ntl.c -o gamma_r_test_ntl -lntl -lgmp
   Eingabe: c, p, h, r, f_r, u_r_start aus Datei
   Bestimmt werden die x^u_r_start[i] mod f_r, die den gamma_r-Test
   bestehen
*/

#include <fstream.h>
#include <NTL/ZZ_pX.h>

int main( int argc, char *argv[])
{
  if (argc!=3)
  {
    cout<<"Aufruf: 'gamma_r_test_ntl Eingabedatei Ausgabedatei'\n";
    return 0;
  }
  ifstream(ein); ein.open(argv[1]);
  ofstream(aus); aus.open(argv[2]);

  vec_ZZ c; ein>>c; // Einlesen von c

  ZZ p; ein>>p; ZZ_p::init(p); // Einlesen von p und Initialisierung

  int h, r; ein>>h; ein>>r; // Einlesen von h und r

  ZZ pr1; pr1=power(p,r)-1;

  ZZ_pX f; ein>>f; // Einlesen von f_r

  vec_ZZ u_r_start; ein>>u_r_start; // Einlesen von u_r_start

  ZZ_pXModulus F(f);

  ZZ_pX alpha_r(1,1); // alpha_r entspricht der Variablen x

  ZZ_pX gamma_r, s;
  int i, j, az=0, ok, e;
  vec_ZZ u_r; u_r.SetLength(0);

  for (i=0;i<u_r_start.length();i++)
  {
    gamma_r=PowerMod(alpha_r,u_r_start[i],F);
    ok=1; e=1;
    while (ok==1 && (h*e)<((p-1)*r))
    {
      s=0;
      for (j=0;j<c.length();j++) s+=PowerMod(gamma_r,(e*c[j])%pr1,F);
      if (s!=0) ok=0;
      e++;
    }
    if (ok==1)
    {
```

```
      az++;
      u_r.SetLength(az);
      u_r[az-1]=u_r_start[i];
    }
  }

  aus<<"u_r_ntl:=[";
  for (i=0;i<u_r.length()-1;i++) aus<<u_r[i]<<",";
  aus<<u_r[u_r.length()-1]<<"];\n";

  double zeit=GetTime();
  aus<<"gamma_r_test_ntl_zeit:="<<zeit<<";\n";

  return 0;
}
```

5. Angriffe auf RSA-Schlüssel mit kleinem privaten Exponenten

5.1. rsa_d_ma.

```
# rsa_d_ma
# Kapitel: Angriffe auf RSA-Schluessel mit kleinem privaten Exponenten
# Version: 21. Juli 2002
# Funktionen:
# bits
# zz
# rsa_bsp_Npq
# rsa_bsp_Npqde
# rsa_encrypt
# rsa_decrypt
# kbe
# kb_rsa
# Lambda
# poly2koeff
# koeff2poly
# gitter_ma2ntl
# bodu_angriff

Digits:=100:

#########################################################################
# RSA-Schluesselerzeugung
#########################################################################

# Anzahl der Bits einer natuerlichen Zahl
bits:=proc()
  local n;
  n:=0;
  while 2^n<=args[1] do n:=n+1; od;
  n;
end;

# Zufallszahlengenerator mit globaler Variablen zzz
zzz:=0:
zz:=proc() #
  global zzz;
```

```
  local a, b;
  a:=10054713583559140643528886738829985246218310689795775468915757617131209360212765783124990604871
  b:=12606538508356856350745686164386563346746497638516951351694753918707638262775160439501534753834
  zzz:=a*zzz+b mod 2^333;
end;

# Erzeugung einer zufaelligen RSA-Zahl N=pq mit n Bits und 1.1<q/p<1.9
# Eingabe: n (Bitzahl)
# Ausgabe: [N,p,q]
rsa_bsp_Npq:=proc()
  local n, P, Q, p, q;
  n:=args[1]; p:=1; q:=1;
  while q/p<1.1 or q/p>1.9 do
    P:=isqrt(2^(n-1));
    p:=nextprime(zz() mod P);
    Q:=iquo(2^n,p);
    q:=3;
    while (bits(p*q)<>n) do q:=nextprime(zz() mod Q); od;
  od;
  printf("p=%d q=%d\n",p,q);
  printf("q/p=%f\n",q/p);
  printf("Bitzahlen von N,p,q: %d,%d,%d\n",bits(p*q),bits(p),bits(q));
  [p*q,p,q];
end;

# Eingabe: [N,p,q],delta oder n,delta
#          delta fuer d=N^delta
rsa_bsp_Npqde:=proc()
  local Npq, N, p, q, delta, d, e;
  if type(args[1],integer)=true then
    Npq:=rsa_bsp_Npq(args[1]);
  else
    Npq:=args[1];
  fi;
  N:=Npq[1]; p:=Npq[2]; q:=Npq[3];
  delta:=args[2];
  d:=floor(N^(delta-zz()/2^340));
  while igcd(d,(p-1)*(q-1))>1 do d:=d+1; od;
  printf("ln(d)/ln(N)=%.4f\n",ln(d)/ln(N));
  e:=1/d mod (p-1)*(q-1);
  [N,p,q,d,e];
end;

##########################################################################
# RSA-Verschluesselung
##########################################################################

# `bf:=convert(zk,bytes);' wandelt eine Zeichenkette `zk' in eine
# Bytefolge `bf' um.
# `bf:=readbytes("datei",infinity);' liefert die Bytefolge `bf' der
# Datei `datei'.
# `zk:=convert(bf,bytes);' wandelt eine Bytefolge `bf' in eine
# Zeichenkette `zk' um.
# writebytes("datei",bf);' schreibt die Bytefolge `bf' in eine Datei
```

```
# 'datei'.

# Eingabe: Bytefolge, N, e
rsa_encrypt:=proc()
  local bl, bf, zuviel, i, zf, a, j, b, N, e;
  bf:=args[1]; N:=args[2]; e:=args[3];
  # bl ist die Blocklaenge
  bl:=0; while 256^(bl+1)<N do bl:=bl+1; od; if bl>255 then bl:=255; fi;

  # Die Bytefolge wird ergaenzt, damit Anzahl=0 mod Blocklaenge ist.
  # 'zuviel' Bytes muessen spaeter wieder weggestrichen werden.
  zuviel:=bl-(nops(bf) mod bl);
  for i from 1 to zuviel-1 do bf:=[op(bf),10]; od;
  bf:=[op(bf),zuviel];

  zf:=[]; # zf wird die auszugebende Zahlenfolge
  for i from 1 to nops(bf)/bl do
    # jeweils bl Bytes werden in eine Zahl a umgewandelt
    a:=0;for j from 1 to bl do a:=256*a+bf[(i-1)*bl+j]; od;
    b:=Power(a,e) mod N; # Verschluesselung b=a^e mod N
    zf:=[op(zf),b];
  od;
  zf;
end;

rsa_decrypt:=proc()
  local bl, bf, i, b, a, bf1, j, zuviel, zahlenfolge, N, d;
  zahlenfolge:=args[1]; N:=args[2]; d:=args[3];
  bl:=0; while 256^(bl+1)<N do bl:=bl+1; od; if bl>255 then bl:=255; fi;

  bf:=[]; # bf wird die auszugebende Bytefolge
  for i from 1 to nops(zahlenfolge) do
    b:=zahlenfolge[i];
    a:=Power(b,d) mod N;
    bf1:=[]; # a wird in die bl-elementige Bytefolge bf1 umgewandelt
    for j from 1 to bl do
      bf1:=[a mod 256,op(bf1)];
      a:=iquo(a,256);
    od;
    bf:=[op(bf),op(bf1)];
  od;
  zuviel:=bf[nops(bf)]; # 'zuviel' Bytes werden weggestrichen
  for i from 1 to zuviel do bf:=subsop(nops(bf)=NULL,bf); od;
  bf;
end;

##############################################################################
# Wiener-Angriff
##############################################################################

# Kettenbruchentwicklung einer rationalen Zahl
kbe:=proc()
  local b0, b1, k, b2, a;
  b0:=numer(args[1]); b1:=denom(args[1]); k:=[];
```

```
  while b1>0 do
    b2:=b0 mod b1; a:=(b0-b2)/b1;
    k:=[op(k),a];
    b0:=b1; b1:=b2;
  od;
  k;
end;

# kb_rsa versucht aus dem oeffentlichen RSA-Schluessel (N,e) mittels
# Kettenbruechen die Faktorisierung N=pq zu bestimmen. Gelingt dies,
# wird [p,q,d] mit dem privaten Schluessel d ausgegeben, sonst [].
# (Wintersemester 2001/2002)
kb_rsa:=proc()
  local N, e, w4N, b0, b1, ki, ki1, di, di1, i, b2, ai, k, d, si, Di,
        p, q;
  N:=args[1]; e:=args[2];
  w4N:=isqrt(4*N); if w4N^2>4*N then w4N:=w4N-1; fi;
  b0:=e; b1:=N-w4N;
  ki:=1; ki1:=0;
  di:=0; di1:=1;
  i:=-1;
  while b1>0 do
    i:=i+1;
    b2:=b0 mod b1; ai:=(b0-b2)/b1; b0:=b1; b1:=b2;
    k:=ai*ki+ki1; ki1:=ki; ki:=k;
    d:=ai*di+di1; di1:=di; di:=d;
    if i>0 and (e*di-1) mod ki=0 then
      si:=N+1-(e*di-1)/ki;
      Di:=si^2-4*N;
      if issqr(Di) then
        printf("e=%d d%d=%d k%d=%d s%d=%d\n",e,i,di,i,ki,i,si);
        p:=(si-isqrt(Di))/2; q:=(si+isqrt(Di))/2;
        printf("ln(d)/ln(N)=%f\n",ln(di)/ln(N));
        return([p,q,di]);
      fi;
    fi;
  od;
  [];
end;

##############################################################################
# Boneh-Durfee-Angriff
##############################################################################

Lambda:=proc()
  local N, e, m, t, X, Y, f, M, k, i, g_ik, j, h_jk;
  N:=args[1]; e:=args[2]; m:=args[3]; t:=args[4];
  X:=args[5]; Y:=args[6];
  f:=X*x*(N+1-Y*y)+1;
  M:=[];
  for k from 0 to m do
    for i from 0 to m-k do
      g_ik:=e^(m-k)*(X*x)^i*f^k;
      M:=[op(M),g_ik];
```

```
    od;
  od;
  for k from 0 to m do
    for j from 1 to t do
      h_jk:=e^(m-k)*(Y*y)^j*f^k;
      M:=[op(M),h_jk];
    od;
  od;
  M:=map(g->poly2koeff(g,m,t),M);
end;

poly2koeff:=proc()
  local g, m, t, ko, u, v;
  g:=expand(args[1]); m:=args[2]; t:=args[3];
  ko:=[];
  for v from 0 to m do
    for u from v to m do
      ko:=[op(ko),coeff(coeff(g,x,u),y,v)];
    od;
  od;
  for u from 0 to m do
    for v from u+1 to u+t do
      ko:=[op(ko),coeff(coeff(g,x,u),y,v)];
    od;
  od;
  ko;
end;

koeff2poly:=proc()
  local ko, m, t, g, i, v, u;
  ko:=args[1]; m:=args[2]; t:=args[3];
  g:=0; i:=0;
  for v from 0 to m do
    for u from v to m do
      i:=i+1; g:=g+ko[i]*x^u*y^v;
    od;
  od;
  for u from 0 to m do
    for v from u+1 to u+t do
      i:=i+1; g:=g+ko[i]*x^u*y^v;
    od;
  od;
  g;
end;

gitter_ma2ntl:=proc()
  local M, aus, m, n, i, j;
  M:=args[1]; aus:=args[2];
  m:=nops(M); n:=nops(M[1]);
  # Die Matrix M wird in NTL-Format in 'aus' geschrieben.
  fprintf(aus,"[\n");
  for i from 1 to m do
    fprintf(aus,"[");
    for j from 1 to n do
```

```
      fprintf(aus,"%d ",M[i][j]);
    od;
    fprintf(aus,"]\n");
  od;
  fprintf(aus,"]\n");
  fclose(aus);
end;

# bodu_angriff(N,e,m,t,delta)
bodu_angriff:=proc()
  local zeit1, N, e, m, t, delta, s_max, k_max, M, M_red, H, i, j, R,
        S, l, s, pq, p, q, d, k, zeit2;
  global lll_tmp1, lll_tmp2, b_red, b_zeit;
  zeit1:=time();
  N:=args[1]; e:=args[2]; m:=args[3]; t:=args[4]; delta:=args[5];
  s_max:=floor(3/sqrt(2)*sqrt(N));
  k_max:=floor(e*N^delta/(N-s_max));
  if type(s_max,integer)=false or type(k_max,integer)=false then
    printf("s_max=%a\nk_max=%a\n",s_max,k_max);
    error "s_max oder k_max nicht ganzzahlig!";
  fi;
  printf("m=%d t=%d delta=%f\n",m,t,delta);
  M:=Lambda(N,e,m,t,k_max,s_max);
  gitter_ma2ntl(M,lll_tmp1);
  system("lll_ntl lll_tmp1 lll_tmp2");
  read lll_tmp2;
  printf("LLL-Reduktion in %s sec\n",b_zeit);
  M_red:=b_red;
  M_red:=map(z->koeff2poly(z,m,t),M_red);
  H:=subs(x=x/k_max,y=y/s_max,M_red);

  # return H; # falls die gefundenen Polynome ausgegeben werden sollen

  for i from 1 to nops(H)-1 do
    for j from i+1 to nops(H) do
      R:=resultant(H[i],H[j],x);
      if degree(R,y)>0 then
        S:=roots(R); # Moegliche s-Werte
        if S<>[] then
          for l from 1 to nops(S) do
            s:=S[l][1]: # x^2-s*x+N sollte als (x-p)(x-q) faktorisieren
            if issqr(s^2-4*N) then
              pq:=roots(x^2-s*x+N):
              p:=pq[1][1]: q:=pq[2][1]:
              if p>1 and q>1 then
                printf("i=%d j=%d\n",i,j);
                printf("p=%d\nq=%d\n",p,q);
                d:=1/e mod (p-1)*(q-1);
                printf("d=%d\nln(d)/ln(N)=%f\n",d,ln(d)/ln(N));
                k:=(e*d-1)/(p-1)/(q-1);
                printf("k/k_max=%f s/s_max=%f\n",k/k_max,s/s_max);
                zeit2:=time();
                printf("NTL-Zeit: %s - ",b_zeit);
                printf("Maple-Zeit: %f sec\n",zeit2-zeit1);
```

```
                return [p,q];
              fi;
            fi;
          od;
        fi;
      fi;
    od;
  od;
  zeit2:=time();
  printf("NTL-Zeit: %s - ",b_zeit);
  printf("Maple-Zeit: %f sec\n",zeit2-zeit1);
end;
```

ANHANG B

Schlüssel-Beispiele

1. Chor-Rivest-Schlüssel

```
# choriv_keys - 10.7.2002
# Chor-Rivest-Schluessel-Paare [[c,p,h],[f,g,pi,d]]:
# K13_12, K17_12, K19_12, K23_12, K29_12, K31_12, K37_12, K41_12,
# K43_12, K47_12, K53_12, K103_12, K197_12, K43_24, K79_24, K139_24,
# K197_24, K211_24, K277_24
#
# Zur Rekonstruktion eines privaten Schluessels aus dem oeffentlichen
# Chor-Rivest-Schluessel sind Polynome fp_h angegeben, sodass x in
# F_p[x]/(fp_h(x)) Ordnung p^h-1 hat:
# f13_12, ..., f277_24
# u_6_p_h liefert gamma_6=x^((p^h-1)/(p^6-1)*u_6_p_h) mod f(x):
# u_6_13_12, ..., u_6_277_24

K13_12 := [[[338560977252, 10057992617473, 10346841287109, 13453024336223,
1351323739305, 6315935813546, 16330673815797, 6147507914386, 9762470128256,
15490511956693, 6528612130286, 2244399240225, 13591124139592], 13, 12],
[x^12+4*x^11+2*x^10+8*x^8+10*x^7+5*x^6+9*x^5+7*x^4+10*x^3+5*x^2+7*x+7, 2*x^11+9*x^10
+x^9+10*x^8+x^7+4*x^6+4*x^5+x^4+9*x^3+2*x^2+4*x, [6, 5, 10, 9, 12, 11, 8, 3, 0
, 7, 2, 4, 1], 18322694093419]];
K17_12 := [[[291776003530658, 569782245647717, 392072384979845,
454436167342541, 581379191908324, 34915864083781, 356995543583280,
142036538023568, 151666118463485, 129771323457287, 100220804719060,
365535901934025, 332014988645029, 26050257838699, 436874454575156,
17291689062691, 358902446205616], 17, 12], [x^12+10*x^11+7*x^10+8*x^9+13*x^8+5
*x^7+x^6+3*x^5+13*x^4+5*x^3+6*x^2+15*x+10, 4*x^11+5*x^10+14*x^9+11*x^8+7*x^7+
12*x^6+x^5+5*x^4+x^3+5*x^2+3*x+6, [8, 1, 2, 16, 4, 11, 10, 13, 6, 12, 3, 7, 9,
0, 5, 15, 14], 70810410128684]];
K19_12 := [[[1374796601506597, 1053037480415132, 2099468220227531,
491910988580181, 1075218393929062, 113380695291477, 1258428228971246,
647490641959557, 842680817158125, 1039279683856515, 772486397366184,
1277983770467703, 821540399345513, 1854460036620867, 748905115209258,
968153080781727, 610051914682277, 1778095857946542, 635634294316316], 19, 12],
[x^12+5*x^11+x^10+18*x^9+2*x^8+13*x^7+3*x^6+15*x^5+18*x^4+13*x^3+5*x^2+17*x+14
, 9*x^11+15*x^10+8*x^9+11*x^8+13*x^7+2*x^6+2*x^4+3*x^3+15*x^2+12*x+11, [5, 11,
12, 14, 13, 17, 7, 2, 18, 1, 9, 16, 4, 10, 6, 15, 8, 3, 0], 905174724293475]];
K23_12 := [[[4450225587507959, 12000132711618141, 17938019942923688,
4037848730656300, 5662765685759936, 7950626703074793, 6738646691131088,
7408961135985140, 3007590491436258, 4406182263098408, 8283925608002404,
3148094571042218, 16830226806871596, 19383526237977945, 17355782533100649,
7163457639570294, 2136954542721743, 15974337528305793, 15634178476271862,
20936209115700674, 3476170968360036, 60012430297377, 11433598262315425], 23,
12], [x^12+22*x^11+13*x^9+16*x^8+8*x^7+17*x^6+14*x^5+9*x^4+4*x^3+12*x^2+9*x+7,
6*x^11+14*x^10+6*x^9+6*x^8+2*x^7+12*x^6+13*x^5+2*x^4+13*x^3+2*x^2+12*x+18, [15
, 8, 20, 2, 21, 18, 12, 4, 9, 3, 10, 1, 0, 16, 17, 22, 6, 7, 11, 19, 14, 5, 13
], 14989997360708469]];
K29_12 := [[[28182448092215054 6, 183201716463659996, 120208154730838161,
348458414510747927, 10845661605481133, 291952398139303064, 179705998376087465,
332641856322418683, 269787416088134104, 129773405167784502, 255184605807658328
, 129481675738238776, 220901718089530040, 77837207585440186,
285727562749818052, 47137264650657041, 35741036505865835, 281815577827005139,
252031818353906782, 82673381409283926, 8811987675378110, 218091579503681422,
50710987126317215, 337096387052675690, 189112841839186052, 276606814068893282,
28858843664545113, 103080374802028621, 337664158402816332], 29, 12], [x^12+19*
x^11+28*x^10+11*x^9+28*x^8+18*x^7+25*x^6+4*x^5+6*x^4+17*x^3+13*x^2+5*x+9, 2*x^
11+13*x^9+23*x^8+8*x^7+13*x^6+9*x^5+10*x^4+20*x^3+14*x^2+14, [26, 6, 15, 28, 3
, 8, 4, 23, 17, 25, 10, 11, 12, 5, 0, 21, 19, 16, 18, 2, 22, 1, 13, 14, 24, 20
, 9, 27, 7], 109512480020094246]];
K31_12 := [[[256687482481782809, 198027369017555170, 86175732123171289,
343687918591207903, 73451086354700853, 256661214683819574, 557563560698862503,
85033573355977564, 473726468990237563, 64259284163400863 0, 639011558834138864,
35206471210802749, 438557736124530195, 417518147259327613, 647953041920553942,
87812621737192862, 393160884541408720, 348516573380280659, 727336948787236663,
716381077698832380, 103727080765177013, 671195289331189607, 652445611478843017
, 333112883542645245, 127430955348541656, 462156757172821596,
477759226091241623, 695282500796941093, 316582040115093182, 656878066603314 79
, 233313299635284862], 31, 12], [x^12+3*x^11+15*x^10+8*x^9+18*x^8+29*x^7+24*x^
6+7*x^5+23*x^4+25*x^3+10*x^2+8, 25*x^11+15*x^10+2*x^9+30*x^8+29*x^7+7*x^6+10*x
^5+20*x^4+21*x^3+28*x^2+30*x+3, [20, 19, 5, 1, 4, 25, 14, 7, 21, 22, 18, 29,
12, 9, 26, 24, 0, 17, 11, 6, 15, 8, 2, 23, 3, 10, 16, 27, 28, 13, 30],
297017278158988564]];
K37_12 := [[[264040218261213988, 578810481147876 1616, 513001129119020 6372,
225444381634954 6885, 329561266064745716, 184477500183803 9705,
348484332749094 6401, 599409194755680 5316, 317266882460567 8278,
352239572882837 9596, 351308640178004 4419, 290415058971377 6704,
```

```
217987881976592233, 286625139742132O316, 2970573691833627542,
3999223150122206376, 372953348209289107 3, 5746426627195219502,
2630321869047434479, 2386564102321346308, 379043198109672072,
2830140054087012405, 5855758571780818194, 1134136046787398139,
3702533192391856959, 3271052192892188069, 5642753568698472503,
4452864519906973751, 762438965968895553, 3466193372959026093,
1823770069953061280, 894642270430122402, 42387813649671739 8,
3421914430089858089, 4099470833163007568, 120397527657163586,
40585344951695180 71], 37, 12], [x^12+19*x^11+x^10+16*x^9+2*x^8+29*x^7+25*x^6+
16*x^5+13*x^4+35*x^3+27*x^2+4*x+3, 26*x^11+29*x^9+29*x^8+3*x^7+34*x^6+20*x^5+
33*x^4+30*x^3+4*x^2+3*x+1, [13, 10, 12, 35, 2, 30, 29, 27, 15, 3, 26, 11, 24,
9, 20, 23, 0, 8, 18, 22, 14, 1, 21, 6, 4, 25, 5, 16, 28, 7, 17, 36, 32, 33, 34
, 31, 19], 5315654521724324595]];
K41_12 := [[[21960093209858057951, 8569821294864755493, 12199228987435622786,
15707623071703125792, 13524774058929095952, 18299291688647884965,
3468449731630900873, 7605258823989029070, 11017475581382089818,
9204220729626035 36, 3499525741095110619, 21757889670154279101,
10502636950095097106, 8302587740752346605, 10104879912575359102,
16511650065288023915, 15929903072134702826, 13011878832169952915,
18504128269665545984, 11562766707601935919, 10519601685793165863,
6062231653787464233, 3900986933726233560, 1867011409612979008,
4510735586453851883, 22530810681628278970, 8939483396609475257,
19731581483650699286, 14571379275416101045, 8495108889919185869,
7938455594330559389, 8066410039214306229, 1908498168733310 5493,
2051553337028766447, 9234197751505978125, 603113029644110211,
8249342954403899405, 21922043266537279385, 13835662743200437948,
1426473344378175 2657, 2083390598934310 3521], 41, 12], [x^12+30*x^11+15*x^10+4*
x^9+30*x^8+13*x^7+5*x^6+8*x^5+19*x^4+38*x^3+38*x^2+21*x+12, 21*x^11+27*x^10+34
*x^8+22*x^7+6*x^6+4*x^5+8*x^4+40*x^3+4*x^2+8*x+33, [23, 30, 14, 3, 13, 20, 21,
6, 35, 33, 40, 11, 12, 24, 15, 2, 16, 28, 26, 0, 7, 34, 32, 17, 22, 29, 19, 27
, 5, 9, 31, 18, 36, 39, 1, 10, 4, 25, 37, 38, 8], 2010184140875250035 6]];
K43_12 := [[[35693880026098986667, 24602109616388799349, 19834172326186580496,
8083461124620612514, 1274257831953441789, 15672771121784313778,
2093748149628792 8123, 3077110481809837022 5, 1027643407626109587,
1720330016402647636, 3017316489786332 9535, 3543556102440863 7558,
3477392220096086 9277, 1972675483326816 6875, 3125693747938243 0126,
3836150543193158 0304, 4962639686288589323, 1806511231443629 6273,
1835918416087413 2378, 1931287431659290 8209, 3529667688271203 2314,
2379710126369217 6497, 1155666383611072 2747, 5858468769856604442,
1577784568556176 6821, 1675569900528202 0100, 2397485974063929 5569,
2220793137890052894, 1080883525855777 6086, 3856503751739914 1357,
5553492644754592777, 5602079459508148604, 6428986331259233749,
1887389643620901 3998, 1703065091456831 9085, 2214348138705543 5750,
6724839511533848349, 2663421903074994 4182, 7559007646278404 05,
1641841432450444411, 2240154535686183 7239, 3405575778411678 5905,
3455381368734477 6276], 43, 12], [x^12+6*x^11+4*x^10+21*x^9+20*x^8+33*x^7+17*x^
6+5*x^5+41*x^4+3*x^3+3*x^2+25*x+26, 39*x^11+28*x^10+31*x^9+13*x^8+18*x^7+6*x^6
+38*x^5+16*x^4+38*x^3+37*x^2+31*x+14, [33, 29, 7, 21, 22, 5, 6, 11, 30, 9, 42,
3, 1, 13, 39, 27, 34, 28, 18, 0, 36, 37, 32, 2, 20, 16, 4, 15, 38, 40, 24, 31,
41, 25, 14, 17, 8, 35, 19, 10, 23, 26, 12], 1857494696480871705 4]];
K47_12 := [[[76184303085263454583, 78908640038775925316, 43142244350540698 03,
2459625943743255 1150, 6101800689818928 7502, 2954394454261309 0964,
7384346233856916 4293, 8871362084580474 4591, 9443862172916867 0911,
2205380248868555 7801, 2245826923732560 2825, 5150205599332250 4322,
8970346112785369 1529, 8533447216784895 2068, 8337196022731818 4365,
8633443982468558067, 7122375466623838 4635, 5830454902352741 0253,
1818512940910602 3982, 1033131667104588 01510, 8256740774115448 0194,
5009115967094662 4354, 2413352221181772 4548, 9608310484979858 1385,
6600667020048294 6308, 3520339849142981 0044, 5217698844748401 2955,
3209079446050559 8193, 5306222767114523 6130, 3162171363257736 3453,
1151120105654038 54103, 4402771846996725 9123, 9323382218740548 7742,
5629653496630632 3855, 750262716568650163, 6816925221900864389,
5906549939556699 4313, 5521677645016155535, 5097648216383934 8552,
8009824149704047731, 3139383474502831 7136, 5348396087059563 5349,
2141998955454831 9237, 1900972557890757 0444, 1042544401080192 56809,
2859553681964823512, 5408901006998420 2000], 47, 12], [x^12+24*x^11+20*x^10+39*
x^9+44*x^8+35*x^7+35*x^6+23*x^5+28*x^4+25*x^3+38*x^2+5*x+46, 3*x^11+11*x^10+20
*x^9+37*x^8+27*x^6+16*x^5+33*x^4+3*x^3+41*x^2+38*x+7, [0, 11, 12, 3, 19, 46, 9
, 28, 29, 6, 15, 43, 20, 27, 14, 33, 31, 5, 36, 44, 32, 8, 25, 30, 24, 1, 26,
7, 13, 22, 23, 4, 21, 40, 34, 39, 35, 16, 41, 37, 18, 38, 17, 45, 2, 10, 42],
2590681158935131641 4]];
K53_12 := [[[42925696011522684779 6, 32920062202720285221 0,
1124418158207453118 51, 1578643727271738363 43, 3264680741702537140 45,
3195743922240303997 9, 2980634969776949093 32, 1153131474050545157675,
4688964910139597870 85, 1033306503374917231 57, 7994910748277410064 7,
3103847304884634710 91, 3906464930652134414 00, 2734809684312661282 86,
4061610244652636381 30, 2779656993161036672 9, 9813595227895073808,
1953504239958767893 52, 3945679141371226261 53, 3158174806283438022 19,
1669086329659421728 74, 3402854799151479929 24, 3683338778175726421 03,
3588550715347221767 27, 4418947861850211418 8, 9186577709529461958 5,
1563697121431213396 73, 3649188503630935633 69, 4166936721735250780 44,
4307259626770029031 04, 2317902148686115144 62, 4909956392021486882 55,
1217600667900777225 55, 4001853962618350281 44, 3534890465569768106 12,
4283058240103495537 94, 1869502730534707274 34, 4305339900535971935 7,
1479038769202790641 26, 4067524644689912950 9, 2812742319510652281 11,
3947629737284909567 14, 3492852531860944739 83, 3606028869469100422 96,
1408855730902950293 75, 7538949284495291945 6, 2806306339685585870 3,
9148697256128710594, 1716085110353537363 80, 894581179168555003 0,
1085293186431750801 2, 3529479332865888823 0, 4884568721313622717 51], 53, 12],
[x^12+13*x^11+30*x^10+52*x^9+24*x^8+28*x^7+46*x^6+8*x^5+45*x^4+44*x^3+8*x^2+12
*x+6, 10*x^11+x^10+32*x^9+41*x^8+50*x^7+16*x^6+6*x^5+34*x^4+8*x^3+47*x^2+51*x+
37, [47, 46, 7, 6, 38, 17, 26, 48, 9, 14, 1, 27, 3, 50, 5, 35, 42, 29, 13, 25,
0, 43, 22, 10, 24, 49, 16, 21, 28, 18, 30, 19, 23, 33, 34, 39, 36, 11, 45, 20,
```

```
15, 41, 51, 8, 44, 4, 40, 52, 12, 32, 2, 31, 37], 2089027756128278038O]];
K103_12 := [[[43208599195340394591 1816, 8257311865662423079 7428,
2214863875316477872391 9, 6541806873916074916541 39, 7082762481178203201417 14,
1066028968367059429363515, 1495828944093857453098 02, 5038704578451612785948 78,
7082732685408935436156 99, 8726630290452785829948 65, 2557871634367394323315 66,
5659337808779811769712 40, 1337538777064937130408439, 5023105061253987169768 0,
1913418723274814329209 39, 8631228213929094501318 48, 2695799716046028010851 617,
4857055445608527284219 74, 1364371032344157833970112, 9361420068667155444798 21,
1090155352357054346511118, 9471674945101381049845 28, 1321369316335652241919829
, 1132996061818041532611338, 1417574788424290195814943,
9238386035690222840359 76, 1262807113652470897508419, 8789445333019825606535 36,
6222264346257397902488 15, 4585587550308636504467 44, 3683750250519341041369 68,
4444242192862958655511 92, 8068115045945789539129 56, 1122158164806408758182 865,
2737272401978128282433 24, 8049606648543911848712 03, 3819193611076227498982 6,
5948820214441504823705 76, 5859607460618700083643 3, 7316556296849734144749 47,
6160794222832867555636 079, 1354521112646502733736711, 1038680961848732230819 678
, 9198341093060740939097 2, 1269460840908967517218 009,
9233949976422357661106 64, 1012073072335586347303336, 1169079425349933647875 742
, 1905341322524777172853 671, 1248527222806667469239 49, 9153799543782965910181 43
, 1061229483944589019966 518, 1178126030782331225331 958,
8907601831895611366269 00, 8761676824301117602789 90, 6710083246176681383689 03,
7693751602061884187620 42, 6162461393014105782766 70, 4913644212631633345602 03,
1271663264116345450976 929, 2641700279787033486527 80, 8482170031007824994522 6,
6143856930992496690351 41, 1363035771553538932573 3, 1127675946443781021418 25,
1485712886463960264787 09, 3764833049279474062094 3, 1468773422199257682018 9,
1495181161731996869839 73, 3362616634118168022947 36, 4945566614030677173313 97,
4552921404497957179487 99, 1192668186872817004378 43, 1301460563783564257033 10,
7227188851888909423091 58, 1269712836971972490029 700, 5210237759597127898288 335,
1215463907997713191462 313, 3024661449615022162833 13, 4828770169644235305848 69,
7445702025325429053458 27, 1087853773254049084340877, 6158667109589149941456 66,
1252709988575387873027 238, 8388894494149558553456 12, 6945458627702613331352 90,
4119415368527990687480 04, 1360568045640836525430 43, 8360553118108669593994 54,
1189218497291227186549 164, 9924161309065005765 7998, 1197847396406271788602 540
, 8431347948768541847583 25, 3956609784993965471115 69, 8081048113087395040883 48
, 1075676818809063788961 20, 1313597774061157690713 279,
5465094909539410824878 55, 5565878512796267544990 9, 1144270918229833952078 533,
3287186089343225275984 49, 5664284222199468950729 04, 6896205016096031156780 75],
103, 12], [x^12+35*x^11+89*x^10+31*x^9+96*x^8+33*x^7+69*x^6+63*x^5+44*x^4+72*x
^3+12*x^2+34*x+73, 22*x^11+16*x^10+8*x^9+x^8+91*x^7+57*x^6+18*x^5+80*x^4+31*x^
3+55*x^2+52*x+61, [35, 85, 13, 96, 79, 5, 27, 7, 18, 9, 82, 12, 92, 73, 22, 94
, 46, 60, 62, 74, 20, 21, 99, 102, 75, 32, 72, 52, 2, 84, 97, 70, 68, 16, 3, 0
, 10, 37, 25, 87, 40, 83, 26, 23, 44, 24, 59, 47, 48, 81, 50, 6, 86, 54, 42,
55, 61, 88, 89, 58, 51, 30, 29, 53, 64, 80, 66, 67, 101, 76, 33, 71, 57, 56,
98, 100, 77, 19, 43, 78, 4, 15, 31, 63, 8, 1, 41, 45, 11, 14, 90, 95, 39, 17,
28, 49, 34, 93, 38, 65, 36, 91, 69], 161036146700639926316168]];
K197_12 := [[[2292243632176490159604514279, 3133119521432565891001860597, 1174184916858889094132238721, 2137354666401469462770260990,
7908626934078090910425954 56, 1067650189138626009082708322, 1334845200250780569583513600, 5181364418491748455586790558,
2077030796591215839545235193, 3106978276778432335042034986, 2378759629175293490126912999, 2997109543980881026108299449,
4985878750350001078154677 15, 1861799458241029471666324905, 1902201183436047855881307312, 1976561166083529303838142306,
2516779976184927655220299032, 1498633106685224985379292 37, 2330905707974592688202710936, 2454233172999961805597196578,
2545189910368683597521078775, 1316620370136853382195677732, 2446826955652148439827140848, 3006889146922241428283877047,
3788232169123326073071762 62, 2818080620985010261185861724, 8926539285386712893828388 25, 1036857713210890635249811 21,
1802723171420707113457911800, 1951242064977516973471845 56, 2160612893491213598391175822, 1957426258387911915545034390,
2007775575610758014781440431, 3313815447054886604410058650, 5280952513108757311636357 49, 4116679269122066922709686 73,
1442168893119877966831713786, 1855617992359398876169351908, 4957851607082183966927541 34, 2206529435043128661520668338,
2402835292735222711569214767, 3086060249087916564512140945, 1014132131194233767378748499, 1548711200338838126712084029,
1464955135919673848685980222, 5766314076275243764607693 35, 7937231638520459048843921 45, 1212920222501914810927662 17,
1871432063913721563365548295, 4985749018485320072191 7457, 3400523185313977123934152252, 3107798538317647758769295901,
9996809442677280764880578 9, 2969163503285377383342254567, 1903974441447679903519297622, 3105661861300539706679108369,
6793465589495591043300911 18, 1761303704316268766329782 71, 1074290293516676420590700 615, 3637117625009739867093433 133,
1793887500821090971186955019, 2570640871559024277703615286, 6201659810533532682647485 66, 2361171093619599513915366320,
1277675741026369070776277467, 1549790451709374933921654 63, 1420397559555354706387369279, 8258711281102429773555679941,
3543073990728975677002996 5, 3741094023350809063379134 13, 1093822381104482388950336903, 1148211350239319517193852906,
4149623811848268252161390262, 2048381195531378005709086568, 1680699353761826799811903629, 8307211548650257815058970 10,
1462279108159918429400137812, 2287618319107151912086669226, 1713617024110313646295339603, 1871071470865334180536274832,
2908732695572993650384291622, 1880477807373018413012457152, 2641756870050858058459880104, 1912712481618532421136908898,
3092834726999955812377236581, 1872657538422265324742205658, 1497833985345357620139019808, 2279062440366053030238153424,
2091946609841463747256739642, 1755434448289262634522515883, 4908804834922784788261780 34, 2978353744380384816920604189,
1587568160639590512693168 69, 1495729850459506437391514433, 1293835612328461117376532322, 1052272081071277215833954400,
2175096260254788866760009838, 2053747633317466910208650047, 3378378556608844969197241 19, 2741858822454057727746351584,
9938524399476337772219544 95, 1094892515216619106513589956, 4246668155598455777087181 2, 2980528749590083794132915391,
2661605424745880212478806273, 7005747576233339277455398217, 1380800155229721956784453 35, 1461976865322906370173828708,
1188997334151010479559662515, 3323011284012576313590380685, 1155436046931730106251733368, 9871249005132930527755376266,
2587589437632496582785278844, 2374595064709184213970009806, 8814145862312309802995394 64, 1011629215608560640338470531,
1902077366118507815569746259, 8568880814852430700495355 35, 2694806229109473070838295987, 2976372509137032276572571257,
6235481070442908459719266 47, 4525959392693768271537286 7, 1047427877752966754224603986, 2031084367242259934734023183,
1000475687435126184894379540, 2190480850975687170839859628, 1003233725837441986246108902, 2624885612288982891858774299,
2668933736580608778847291611, 2776551514601587438612285581, 7176025502094175926227080 98, 2149724551337394618471103077,
1583793580362115084508256081, 2346791492241977246022005097, 5808888069171385149112848 78, 1182894576020571676044954095,
2106555890961806021304754094, 2327143527933381388011564588, 2420483434269250213211832102, 2298710854576582849919272756,
2517597254638752725005216359, 2071291344423896392248223659, 3126637553807743371937053684, 4548318665334535552062354508,
2606839234063199545786561942, 2039714541470691763488742852, 1741040088387833730955143494, 2602000593364607578322859227,
7885317891422655739698274391, 1876799675845413301984224728, 1780249063615482018004344878, 9678440569115306198992155656,
3393064554230742698895499865, 1122245890083332026012560 08, 1839254857726848040299349115, 2782868628530727082904915177,
1052922423914686017504734101, 2575075207571517956780630434, 1179083145937332528039787408, 1945477073731988409300176988 8,
1727728503018213256822097237, 1824640780649244402424389 3, 4319456785368692151054478 58, 1055145285063286409585489 97,
3860138406482327055354436 59, 3136855326024394994991208 65, 3185825056433601554947321577, 3392342986833671450752463 92,
2556659938735777678826706796, 2453298181148924093394288528, 1856795436407827207108732663, 1417870320611589784664354 55,
3185535953303953590848309427, 2996993666010448075770594684, 1928750268944456301266905925, 3250175995727595107453930572,
6457211279293544044343364837, 1530196227530702950246634 96, 2999181375835752743651794642, 8217095818933156979847755 23,
1967563649806208903714279026, 1931988216907728091867586288, 3351614434259289227270938049, 2021594568873153409228782486,
1660810357444432372288649164, 1356140240070524230356035456, 3039160442350814923449823112, 2200107939852686262497979594,
2017594840053765245179510532, 3099088617742283200415297025, 1928131485307765700495662756, 1782144193075656297800479 01,
```

```
279948107267016804406492064 4, 749316269986001055329759 75, 226002077913058416367916 0181, 152523153549965080329987 1158,
22360893724852959197270337 32], 197, 12], [x^12+156*x^11+146*x^10+20*x^9+88*x^8
+187*x^7+136*x^6+34*x^5+74*x^4+78*x^3+45*x^2+138*x+13, 130*x^11+161*x^10+131*x
^9+105*x^8+90*x^7+104*x^6+140*x^5+124*x^4+67*x^3+126*x^2+107, [151, 131, 2,
155, 161, 63, 164, 110, 93, 9, 193, 11, 50, 86, 105, 144, 1, 47, 39, 67, 12,
88, 22, 18, 16, 122, 81, 27, 80, 168, 58, 66, 32, 111, 6, 189, 153, 140, 38,
109, 40, 104, 55, 191, 44, 75, 7, 37, 156, 51, 142, 46, 146, 133, 59, 42, 127,
92, 129, 29, 169, 115, 65, 17, 73, 19, 21, 64, 8, 69, 49, 89, 52, 183, 96, 41,
167, 57, 76, 160, 3, 43, 82, 97, 121, 143, 95, 87, 190, 137, 90, 23, 56, 79,
77, 61, 106, 83, 31, 53, 100, 101, 20, 15, 33, 10, 68, 117, 166, 158, 70, 130,
91, 74, 114, 94, 45, 179, 14, 126, 192, 185, 107, 123, 176, 125, 119, 84, 36,
181, 178, 128, 5, 149, 163, 48, 116, 124, 186, 195, 13, 138, 145, 148, 120, 72
, 35, 147, 30, 172, 150, 159, 152, 173, 154, 26, 34, 157, 132, 118, 60, 62,
162, 139, 24, 165, 25, 134, 102, 99, 28, 85, 98, 175, 194, 171, 4, 177, 135,
136, 180, 0, 182, 103, 184, 170, 112, 187, 54, 113, 174, 188, 71, 78, 141, 108
, 196], 1385688817087855623251543351]];

K43_24 := [[[82584220954059529228298155879720697718 5, 139044836008736945778231779945646461982 0,
150114251040446707650013048028618257318 3, 109651404865178763285421519661335375420,
118126216459048714927994850188980608602 3, 758522565853412588292476389413899696 07,
257297497861525990289739145001982205690, 186672060338707387394871056667395830694,
125468636522833431661388607145096057 72, 338590174412954382508460144427663453749,
129634648716748401441877500286711154322 7, 109182409057823165363351488764046371 4974,
289032933899775184645952060963445894427, 211204087244398171787025571136767627463,
484969730174254550951392566611872755724, 105061925563626585062582936820419223965,
789929777615872744202088020424158703062, 201336201318653316245780640542241761 59,
367231801367710217319050022443095830984, 825465033842223712501075851838334921180,
857864337239442748938179553739415630610, 539928823254836726065674627038148530047,
148161535518099585104100949382631053313 5, 131457933614200636252693697778892320 8868,
446137503141166158404330442680983888033, 611056690042636146185055978599352650661,
100230383041144637169529940123089451751 8, 502249445707643484073907223897746605007,
131602965405186958943055400909606309785 7, 121086992659681257024577511185866645559 3,
261217153610364381059805476335671631363, 162412011566513729461828729662102429269,
465105664788284329255989054648092971446, 140234847387872557676085606889159239384 1,
917416059751065981171968187916387990005, 758712545617002985897611535101557106298,
267972646484002885799901984140221500142, 967949141063582404361810904636870274673,
824065895078426415437425898391302712891, 119693625513881465331849923409996325492,
333956832850814167397369346683298725056, 304871726189666302540144501501527704229,
489690394655735487595159599812221854188], 43, 24], [x^24+31*x^23+39*x^22+41*x^
21+8*x^20+16*x^19+41*x^18+3*x^17+4*x^16+3*x^15+13*x^14+26*x^13+13*x^12+x^11+35
*x^10+32*x^8+7*x^7+6*x^6+39*x^5+12*x^4+14*x^3+16*x^2+25*x+23, 20*x^23+3*x^22+
42*x^21+32*x^20+3*x^19+36*x^18+27*x^17+3*x^16+34*x^15+3*x^14+38*x^13+20*x^12+
36*x^11+31*x^9+8*x^8+11*x^7+40*x^6+4*x^5+32*x^4+40*x^3+40*x^2+9*x+18, [0, 12,
21, 37, 11, 38, 41, 27, 8, 31, 25, 6, 10, 5, 30, 15, 26, 22, 16, 9, 33, 18, 7,
40, 2, 36, 1, 4, 14, 29, 39, 32, 13, 24, 34, 35, 20, 3, 17, 28, 23, 19, 42],
29209609682833173109734739100823495154 9]];
K79_24 := [[[19351717242314442711059858384938754046423526639, 319697050331337120319033225403551805398107586 6,
322644735073307708513543517224180957391958721 8, 187588649303808210356996113966438778689768144 3,
201581513533777465915492539571619748541377838 7, 230540734972596323805632479714359203332405910 3,
188956482022837304290782537724896616724963125 9, 303025957682404186444456683466305500120635071 5,
235287951017295884574126574672814667349992180 8, 212550922397255634611866834104164878902490961,
347512708355705107850230004916764259861023337 6, 207212305542399412534242612386988884209980983 8,
688796106059952357365799686582710281373544035, 201061939152187613315381936988002027603897246 2,
171849258466175228559371175078785768164989383 2, 313618966362298585492897644032854901741351158 2,
342500291755260293805404263088057701617205900 4, 136558618791663860298437588227169706408178042,
235936787112240328296173958044404275265420246 5, 197295602750829812052907948430450867696392153 5,
521268401732223975700854327437838197578490267, 229805405844496276260085048890365495055873802 5,
207373598788602595259389450552325428861675810 1, 900197212739458556986244394423894899718151779,
132214216317327826112718798735449584717050422 1, 296957337683988403526651914466911624991393279 7,
158076991611497308719588309071146617030709911 3, 107478076275035327146357457790150769657653947,
120882475203186534900944153642994467049344073 3, 207198464996026610957566632240509809528505673 4,
277044107302484819412919983106595740078623420 2, 238698558388516148025494510889503122338349694 6,
129910381108806013453707767624304330633449063 54, 321711756022616340982088118875783277515076590 2,
162407198722121539494533740989549367132585991 7, 276436753472164567825992262996931870556603840 7,
174648023916174456958342767601763213510684186 2, 798864412292257599652022614302325231112614 40,
767216951868280742668045530489776385545873536, 119430246435524042165738339779514771588267677 1,
125270392564567139510381779561442685441130466 3, 317455221301637820073602000698772314874773413 41,
602902458439206439203888064223883509929410040, 196680104178611298551365136405434616519509553 6,
235260621197500462502234874800561553792538934 8, 274249476016163250827114626479283585145610145 7,
256811784026500302180658822570661772351443230 1, 330831363832239291800915285335105640775972201,
279090114605771429650093284703919641679623628, 232268844547950839462051892520849942014911134 8,
293270870553832990104250877703213129932259125 1, 235678834067775571943153315974166683775894364 2,
121974605968787981141009383749243595273781095 0, 425396905901829996547124515590386603659774415,
214718440693055548269155685088208412105583313 4, 297785976212947857025504144052127003767531627 0,
340036515917804628635280458611384945669114471 5, 326958488737296528942471413131845355174349929 2,
305158240627422523786831829202939482977026396 5, 130800541700871162789899415441297439163307239,
146940330977404732562007338344765950304158323 5, 224405729803846511387856296185150190663774121 2,
211242419516799201612250346689616870346468063 8, 532173477764585117897715006881937957527530028,
265362529296280750089632301429316538567842506 4, 231684278181849161286774943799716727233795604 1,
122792989399214350216882315207047349805911176 1, 171253693100684038026859344147476707876096618 1,
256426690895186130890709799905833761622362638 2, 446914047655967492753306220951431692502830130,
181096964567432378587989274841371956604992960 3, 367299463582509807433341963276394431348562531,
179143600714780916972605819031150362206064520 2, 145075232088733552083610711119845824030310603 0,
205123034739614360651711885514542781820220706 5, 321235911145916450233363038283958614078693672 2,
161497055894288539993232382459093255263767064 5, 347246517412041285472310064744004574514507001 4,
136262906044862930355961765774913914900123478 7], 79, 24], [x^24+62*x^23+x^22+
63*x^21+3*x^20+35*x^19+33*x^18+40*x^17+68*x^16+10*x^15+4*x^14+67*x^13+21*x^12+
46*x^11+36*x^10+60*x^9+48*x^8+68*x^6+42*x^5+35*x^4+64*x^3+51*x^2+64*x+63, 58*x
^23+76*x^22+67*x^21+62*x^20+75*x^19+13*x^18+21*x^17+23*x^16+27*x^15+9*x^14+35*
x^13+32*x^12+22*x^11+60*x^10+72*x^9+23*x^8+78*x^7+70*x^6+11*x^5+22*x^4+5*x^3+
70*x^2+51*x+4, [35, 37, 68, 24, 4, 39, 12, 63, 55, 5, 34, 36, 73, 52, 76, 15,
0, 17, 40, 71, 20, 7, 51, 23, 69, 14, 72, 62, 11, 64, 30, 78, 32, 2, 27, 16,
22, 9, 38, 77, 18, 13, 42, 45, 33, 10, 46, 75, 3, 59, 25, 60, 61, 53, 54, 19,
```

```
47, 29, 70, 67, 74, 41, 21, 6, 48, 65, 57, 49, 1, 8, 56, 43, 31, 66, 50, 26,
44, 28, 58], 167953082470569221958110668043966548222583685 9]];
K139_24 := [[[13547106973736773376941044327298691787972880840996 5, 14254821922593487523798662172076292028375934179246 63,
19055630283182829147672991906387480841921391810000 48, 28771320220461361997330852913351949483010436888446 44,
11531157678443050672346511929878395087798576125444 5, 10687514529380793114154776524239764753256271326726 96,
21372456234035120041349286431922996773150958154222 9, 16299786397639614768643785320891121754987245869123 81,
11670173634077955382901174697098837108417369789015 3, 94877559456417408709964886237352496510749248914175 7,
18919142742908842510617092266177204814747337157590556 , 84770018460820016815902274542888253488078979554564 ,
31059555946784894231417121655182640337223706672329 4, 73396335870273958220185419335248314663284754890913 5,
61316977383242837033352385775188888133611508112403 4, 24031704700922478252347316572218057898660961045909 54,
22270571712861587394508122050098798023487044248592 03, 20862708019850780611010294040504674262567339613698 280,
18432532096588785797076624475198138171548556318470 21, 22102134118065392810199363484597660554818663200684 96,
21753286912402285265260608605348847409192522838467 95, 41460497833372208213686743440885879611906865595068 08,
70988616945812476319409649745849880256100987642710 , 16713414143836362910887594044715679302930330069701 48,
15991840849899004881889991659042545881378104825875 00, 23078209188507238760054289985000708067011496111047 886,
33206939465108282341817001535340006785464970245305 9, 15796513445626650421472982673944183532731031464619 16,
95722002518387604056417341124676451490265422283856 6, 15671119673734802098446476572699471017067879528466 36,
24562267587005980096970905733947455209898642312994 34, 23098852997822182887139679949643941647941008555954 849,
19703875275547857596180500697225490458144052832373 89, 15000626151448785285259435753872572082924400916563 78,
76738934289726230607393293308987934995301308762255 2, 11764913384131139854717337766706486861345511622489 41,
26835406232946575644444029436512154373634546475744 37, 90266153626181719942301235570558203366367307040034 4,
18328241299374719050197762273179675820509062870515 0, 21072206746382928106464997730848294861253882579141 28,
13614521275626954525569253853276940154236837575847 31, 75758045526588611946991750511916892019478309040443 8,
20947187819174745425236200239527228493001667524916 56, 21567224428807587013008467439749662614294385559238 688,
11885767236586241792383046036616843592168657483850 3, 10547074532276616483034492686104529303929194245110 21,
27005681296130243802486166056262665882908865878861 75, 25646652474020637677009475134544999937931232877301 87,
99242668709919617504639436523044999869289825500387 , 19156773449307840131398135892098459714177602070919 96,
18408549817001063922581452720482386136450437547586 04, 24217290350068392745517913584536474193611733354826 04,
18427151833630959877681156452176735818490534083537 62, 20276537670409501717798777306549214796722106308031 57,
22186193065092049442764786691422220676375589777082 22, 26768679468261205892797051705857149243297779340058 315,
19419257220854628595255283789972530789226491737262 40, 72313283821687486276115752565851037994767844425178 4,
22808581442730055312480840508638211419482832793455 22, 77800791428191270201815429538946952378643219113951 2,
15034586247922791758782497702328081471264372156956 66, 18863593040333778721111893138403558221051428090172 78,
18022623647290035476767139085516587804369835985054 1, 18319549554188902738480832455049478653907793879134 8,
20959425247235842542933722299187163975896911840453 74, 27178344673692068742160405587476004410337778290631 5,
15281016026269763228114433413306333657056591953478 62, 37385180469072823521981147394310840247236406111691 57,
19006588135270321171329083730748946255404804933575 075, 21796673282215029722734186836858018955647930133037 2,
21524043239663064157447421732447062229407480570495 5, 16487859455399412333239556798171771313859359440714 60,
25501553926909996762607043574964198961116635488197 , 23766740621585749068058586765268157149725030989879 6,
19754154037346313258186314335204052579135834717599 88, 11314279147164991802421147966177026721165561695971 62,
80226927046722711016756191951919263920211244472774 , 22957356794167730546866582920665373916388921810088 31,
62798024277127771244571365117854360930761557336695 4, 11123112152247635163329510088348912987633009808093 ,
36433525329438196175141802967383577088336090406400 8, 26127044141963105131637553678874899725851321552336 25,
12507827853815711502623763248891711499936817890701 76, 99376186837619797855278380768814851257017611104284 7,
16299522349985567622614857861098640459555155940612 57, 44445554774389539449041157620188348822835105614066 90,
17901371169366596364939850109924322786149678763101 91, 14273861000403054867472340561320806792041205405728 79,
30649124758502865423438639481561347749729044711991 3, 67977301016164437266658607488018219441379142964700 7,
19088494549024398102013800482681573318395221637504 45, 73658240977370165172604742036722376975759766377980 ,
18366894295519059805668398297777909761288760463988 83, 14080472822581197638916943242674427660582068072716 09,
19911872568197980255431229773027505855638660472126199 , 21387212400395940689242973545150918360908904672591 82,
22116001660663756526804568236497222885313968414774 28, 14196580710003267915033331697919534484357459644796 41,
13018684137332903191854788096722441408894515352431 09, 19481091722542124843372503755420649767492145969998 86,
96890640214371434810324608070265473449698816138991 1, 91270303151302463157758658908707758127173896182251 1,
13421609314001833803080313933517863102257415665215 1, 10794886613284708844584865496163363774353576351448 26,
15268030648872517419460435716974797222834954224233 09, 23915969249593497115784457718438123848960336911779 28,
19417930539546229911799893669284581191473062405849 36, 93188062607377476637913843240651973984690177485792 ,
17063429922443587934432556069310825171751005227765 , 72420851314807287883592800029587775744380357310792 8,
20328417597528698222958574606361998313235336111377656 , 18038376779991211093936768439369665733869906551139 54,
23294971863634945409027618130416520555305384033505 72, 23029245754185468816225683092210974946691581120174 77,
61099762399101838127166403814906876063826214480881 8, 58863692216151792280269681070685142940498437572596 5,
24244323276161554135238996199597148395878280150406 , 16618953401597221824667271710308131132292333094760802 ,
11574831595308254286892310567731121587417538136903 95, 17835907923498136739089974386149112691761766128818 9,
24612001748601096683187198707087465809570853337111 95, 25748178476114665402037361349144024382559220769060 60,
22675085702513514102899307222530571196215733934354767 , 58476610061048769878418879354590070365216576705502 6,
12833731843791128870172612109258983572824696536053 69, 25587354784264422090938932611705737895097422193212 83,
21991257768432942451297209247649543436483264504164893 , 12020626209122977197746021959693538880746251179079 52,
16858322277179889932118116814449195720482535502646669 , 54904186908073914543165819736678064082402444955108 ,
18320386110007205312366501016895405828526806908823 16, 44564312661705274863250683585680940549194503321 9,
26516731859086753365586997393956998191837254226462 04, 31653467539875124406402947822230178242589188423390 9,
12796396517249994578988626457199953509443745305358988 , 17861617423689652664634309806724036040377781951348 57,
25951285507987265123940795709559198033655200137018 80, 18500792167278315952506681869863212310447686873826 6,
20700699108244377579826801739455303603763281094264 29], 139, 24], [81+38*x+55*x
^12+84*x^11+54*x^10+60*x^8+99*x^7+135*x^6+40*x^5+36*x^4+115*x^3+125*x^2+56*x^9
+100*x^21+15*x^20+101*x^19+33*x^18+59*x^17+41*x^16+86*x^15+96*x^14+93*x^13+x^
24+124*x^22+135*x^23, 136*x^23+85*x^22+137*x^21+4*x^20+48*x^19+79*x^18+134*x^
17+48*x^16+100*x^15+60*x^14+111*x^13+118*x^12+100*x^11+71*x^10+51*x^9+129*x^8+
80*x^7+108*x^6+79*x^5+18*x^4+73*x^3+3*x^2+130*x+100, [127, 101, 132, 95, 133,
5, 28, 7, 1, 88, 10, 108, 65, 66, 53, 123, 27, 17, 87, 124, 35, 103, 22, 112,
126, 136, 96, 50, 111, 18, 9, 86, 32, 83, 104, 63, 85, 37, 94, 93, 11, 106, 44
, 13, 76, 82, 20, 47, 61, 49, 120, 51, 75, 30, 0, 105, 56, 134, 25, 3, 137, 89
, 21, 78, 15, 113, 107, 67, 39, 138, 114, 118, 97, 80, 122, 52, 38, 69, 54, 58
, 29, 81, 102, 31, 41, 42, 71, 2, 135, 91, 8, 64, 92, 4, 68, 79, 46, 60, 12,
121, 77, 90, 59, 55, 6, 62, 45, 116, 48, 74, 70, 110, 24, 98, 19, 115, 119, 16
, 109, 99, 14, 40, 23, 36, 57, 125, 72, 34, 129, 43, 73, 131, 84, 100, 33, 130
, 128, 26, 117], 77941568287321560682444015692729640119793594246858 0]];

K197_24 := [[[72253477537553828488290175978714886386788040153905660646 , 30530208558459768527372120692448274770758687797557523 10,
28699482888045155462570910485638173215588686836195395611 , 19335011823414033939709373868618222062892127259963566 75,
11134021335714014652717811745515959662900557222099928640 , 11024633299419144494326290827181071976986508033236146 470,
28429575896993218533344512640706661374183569054026245 85, 60208617070988285587570369243295575198298129727103099 05,
36273365315066506151987025121752673530794544493505647 49, 55216412599172243813613637923299496772573175711602255 83,
11294900121996749906797974668646270557377841140996581 885, 60966730580234905943083483541683500120836670579132623 47,
39027781942341978695334690465455449567223332599634421 71, 55343783853893061067752344682256449944724745513178685 24,
```

```
64769889811539948842021594165515250134759402158693866 12, 39082364493541827497495070218404439860112061705789724 40,
10363177885336071205596556786603036298679873548642928884, 34949017821067283660350803715332805417508220205800161 35,
49581911093582186131153639394713565148356773782196926 45, 61176781719184485797959876938871179145850073841601298 35,
10843241062123011789507600212683187260515464584934045688, 10209160189139896487727582695523601695913911120271382229,
19711313934179607755528750609677618407010153221061193 40, 53291644407637660574734070007235232423560351944637464 80,
20699500987543874683521121129824489907379512374762177 09, 10824540693186455717499724271651996041843720777532467957,
39601156500377049151359349045780337300945608732460640 63, 24786411489346956423852049120906660470262762354945262 17,
10694329780419599272462968414398680336725596352156266632, 69701900596295413112009445336836845536123462046932945 80,
11441088682126487370226564356270250977236726648542413803, 41077267951015287952653655319738711539587199735577591 6,
85043052385615362093944019072340147175342316945027044 24, 47601645018927742327579268851999549606877778300411965 53,
10419141402118480785297592516512209656440808757929392267, 43066091087071884656987887406948972110106722100657376 96,
42726090439504368600011894290539414769496431218473106 03, 77757730810377331949617053807441642069668686842916913 3,
10061764367698680048635346826606436834304737428740692829, 69235339660454959476739713029050892170244753002897980 64,
30079248403301251485458296144318876909912064432732124 44, 12236761774021307807042834401430980767139664500293235 15,
10910813750690958951503490065539214913778573652682977817, 10810805107868344022187583190550124963174430785936316796,
92115950737723226195746616536382649708565496572008619 91, 10459334081799556077148329486227402905917265282270072429,
46787584018972585550975475998535001697739089746181271 263, 28796814445672750406160296348078789687190029521564391 61,
10740984259658161878554456528941555263338839406513715652, 69082711450212434022521140456796013770775745449505895 69,
58948729794161500367327144845596554776835859412307102 03, 55540181981886120770088519143975689306475331902654639 ,
19118717256763189989224936230616624882560045303759521 1, 50496954465530680313699565814137777211356387229184901 81,
92970886411203229602832713276932782303396796525065 8049, 96376944821301196007070444096771671958497600203014588 93,
68187984449282556631763485480222199082775106999168603 05, 29350576361190926763325731359698285103942754046641657 80,
44623097573810683362045627516029132363014214773422693 51, 11322883957212438649418998451014227051250015789566039668,
62553350635991079929015919693828313942049178696678788 36, 48853811093381300853757196104402372923621690946573696,
44758440081946456243714479702888270988462452092910523 82, 25584430641623168835632088216796829466365735472007584 64,
72633298254602522023530498018451348778492053262048990 33, 11399831323744721070767050903918500006892116828576094570,
30205376523447321644511054199231522087721327631614480522852580 67, 64348876776445512270583799934755085498455730464907540 0,
75815573831282470122489984792852522208685989671993749 29, 72668144181837395014013704202577862033180588272093468 70,
45943639171878235130280365388047890391137954895632115 96, 98288048418688798463136721001805854306762435401679128 05,
29417300536535279702463614742857702140953187869698260 55, 70177661712488863286098564567603754144586091281174699 0,
83739139813727594659251473436683648756211793455726978 18, 31802510192103318971248995358037972306015483626112575222 ,
11392518527057978503814054886369091204357572130631718400, 72356934703240815161726845506737137849914562543292457 67,
24550295683632287945143971961447003908067393354530306 20, 92918151852434376614015741831386989564587364810776201 12,
60881671676262256291507551663691226416691653279907201 72, 10705518467544315044969157367028250176240491133019584 124,
29257921017123211615923949834157266688473629190850698 53, 59980002253711386691500181706505484707452103304416335 ,
96982026090138395639387562207123276316144805228528506 7, 83688698708140133779029644232770060958250433669738496 26,
10463033544408634654788136627675506079917113508436892986, 16973823960179062198203157390099342349585392780506435 69,
53133592970802957387216393261511872733667240202299192 3, 77575567875611172733459525554049647290200718640898319 69,
89456689543758903076279770361324155908534566049134025 80, 29257700023627570801402258599983819606370073904555396 ,
12414278345008812887016725970325653302483193755646899 23, 85679984309939827776106786370121932157874754573643805 93,
28661195230464635383476821777369913358311601590272945 9, 22923617600722751641650231120526088526048119362240694 78,
24225657718091865661885339123033336550058271901340883 78, 28685883513873465431004464315634877589414943181524469 87,
95097997752777568336759459798896828781694183562870321 4, 31244976521184106409897269550806209585084498631646235 88,
10739309988148681804607968318676474661864534613109324 76, 45226687156324320776187635092426608892004059667935228225,
44980300535744438641948472928720164678830428671192193 60, 10343504498360086972979958647951775494486119838648648004,
10343527342255594012278204379332374380617029983391605862, 54896558841783070017338930699537260249978556024407336 08,
41657697081611228484610274847597707492376064885698404 70, 77105895209029492420983161878743294789668824700636 46,
68523563169046029064290052466878882896676425020597370 88, 11098306174009008418828339039409961217577166489075128601,
78685475814854698863775822584194974638674577836300468 21, 68682576217273448704434337790384470382422956130819119 70,
80631054739287195251814991294005600929210918902708585 73, 42457072282901688099980885646211839987564734782153874 8,
19809641852472010930667685127038867915398258294146557 89, 73168826091125354631328155767412485988237534986653866 53,
50383123723209296165445578717881925494295521504782108 83, 53423134399599970550742516765776550697758978537657023 06,
30304365165137427102211893429751766425678778134845961 36, 10521566414666929224313525594645887797554465344331688496,
95915842039376405220915751725809923600415821725102328 98, 58526703726486133081470888781819402853813056215806668 0,
10070446203379541595572561835090865779655748255563488054, 10325460889453952222737938559277214242849272790246526819,
56633107631024892841942898044098260488400051109077472 82, 59326540570413075552654337432591100515398670189064645 8,
98262657061899216632750643558619427713271765364328277 31, 11431281488970531176515994162851550435164888850801617426,
47341838455524695548032646561508037797233084220586357 42, 22015418981206992027083941668117827100693769431684818 4,
98497752952548598879754384448456730673504400842346268 50, 45446584098775006879285737878870435546642382017386271 86,
96519338596327785839669682882623293757789682826097810, 10898325954327858553574031640169553964880257431621271279,
26385386033455879061973049083783550718146835707162582 42, 54108329060860214221818071367714182721786570688657777 6,
94032350487227981630296432855117197845930581611993880 18, 70934695128097747835042786520164189825012358263330724 56,
91044527910006268165989845364438803248509303494274570 77, 68414186184822053999783273408273560310194323167890032 1,
46306280239780972419456711572892592618044644172192022 91, 73168874736894747270026500359107415521502549647429780 8,
11122374137510761966271353952259287677375177054038267052, 48372038773667194905055454074621546005773274005058611 11,
41436253863102152686172690888901512988090094600251168586, 66655546551301259884021516478872230484207580219886078510,
11132525749350015565582355773212474800297779331803124561, 17442276001884932388473562419440748155701708383111761 7,
11370349485660292145754948149876239404855872966861013 94, 11359313198689535465564140132675930564082106202869634692,
53975474668957581511161088341978617978089344613670847 81, 10202009506032968273135768981030412117875500035089011232,
10747013086965094921941087298210598584294834670249542216, 26557411406029918125910533642810777620820585656538693 47,
11649927355935220300516959867339856326768050040056287036, 78573444649994935817479502001596291367132670829367158146,
96664470596187280437061240159971419827182753018306406 93, 28628514404658274171168194859883911566375912201559209 3,
60993763617207357784505431675685393844421066739024995 52, 92694605280514083719548601813251198693177197563743053 15,
21910748943337752109489471134042035016272366913550264 53, 21119100427443359426183719314910132702179954176238136552,
18586057831907520299050918760993200571354803177580136 09, 74634163016665092785523521928929002227039226906537239 31,
44952246528708344175270208603205613404773237196752265 47, 37149473649417841682571053589779482044801826426599911788,
98651916640562964627668150896050448139480016952591495 24, 96786151415780987676682737699207592003157279399779356,
68225767749569346891857747837221706616418404682174196 26, 57070658176664657761341822365294450979473447301047466 8,
52592529622812217992309326830979792595989058194513746 85, 64742296306372313573914063440085691226564393126732225 00,
92703347966724176837365079881656514589690076604392653 00, 90954404019829976358503921772581099369087146859281545 12,
12903713940707748948823602759175269229930583442091364 43, 10831950154690756753549589421668107625829966981605437476,
42314905460662331293219351268959219187738330995190788 1, 84829145218174611931067834673042587244673475334321260 33,
73229023989686159770908904798126535139801568664656382, 44276741455203881916187648052051183629925613251785177657191 5,
10541508357813332467312586475654746048864039638259487 639, 89470808430567866094112354457246730473947164423891554 11,
11425266400986688066699168495292325626451077547591086911, 90818291656331499656180813507556238335286585285230556 79,
95583817765691052424550115770196899758621298967746032 44, 94218759298186443380694736549724863518619246513251785177657191 5,
29128528955213793421349315173147401416715376021737455 08, 32877804651827456627720535353642786130041429013612951 90,
43178135930252982602408455062773316596424573722873970, 35313192705512485279047039685100885149408266126808758 23,
98581852728156137063719160606099946853751906653726649 15, 10302288724072054192287074349579771541258434494428755 97,
57809518576300967860441338063209136470113790304189556 85, 40542439471820194622259274764410905213218884775606384 60,
48926073173135174985693146731857428984896782849747936 84, 89600608713258955035525878955075105928803787530066679 11,
15833937793541621243609738970491460943340355004774407 00], 197, 24], [173+20*x+
147*x^23+4*x^22+2*x^21+24*x^20+112*x^19+169*x^18+180*x^17+170*x^16+22*x^15+195
```

```
*x^14+64*x^13+115*x^12+59*x^11+189*x^10+133*x^9+91*x^8+73*x^7+9*x^6+149*x^5+11
*x^4+45*x^3+68*x^2+x^24, 116*x^23+3*x^22+112*x^21+156*x^20+69*x^19+190*x^18+
161*x^17+138*x^16+156*x^15+50*x^14+81*x^13+167*x^12+51*x^11+184*x^10+159*x^9+
132*x^8+72*x^7+41*x^6+150*x^5+70*x^4+122*x^3+124*x^2+12*x+84, [0, 43, 142, 155
, 153, 112, 57, 138, 92, 125, 170, 122, 61, 13, 95, 53, 7, 164, 18, 116, 195,
21, 67, 163, 24, 42, 180, 141, 115, 73, 30, 28, 120, 50, 102, 35, 34, 86, 117,
8, 72, 193, 178, 146, 105, 37, 33, 47, 148, 49, 101, 66, 145, 151, 177, 128,
182, 93, 194, 59, 17, 2, 184, 63, 78, 169, 99, 22, 176, 9, 106, 6, 191, 29, 74
, 75, 111, 31, 156, 1, 60, 3, 81, 11, 84, 85, 160, 124, 88, 159, 83, 108, 185,
143, 104, 189, 137, 45, 70, 36, 14, 136, 118, 71, 174, 44, 119, 54, 98, 123,
131, 172, 167, 76, 158, 62, 97, 187, 161, 152, 55, 121, 165, 103, 139, 134,
126, 130, 173, 52, 127, 190, 132, 114, 4, 27, 113, 39, 109, 48, 140, 100, 157,
38, 144, 107, 96, 135, 46, 41, 58, 23, 154, 69, 183, 133, 77, 40, 12, 15, 79,
19, 162, 179, 168, 192, 25, 65, 89, 5, 10, 171, 51, 188, 20, 68, 149, 16, 94,
166, 196, 181, 186, 82, 56, 90, 91, 147, 32, 175, 110, 87, 129, 150, 26, 80,
64], 980482969277832811677592590355236298052987326733874415 6]];

K211_24 := [[[14731271674614532079038588046790870439932816694900269726, 381575895369372331103666922746052399777964910330339025 45,
587490506339856215141114028701660706445443283377018011922,  469693587974786161449216871989123358426575301865115184822,
448249498489623975520113924590366562897531114813469653177,  699647224631920376042314291793855510004069738965527418 6,
397328880167439558295274885210470398159408035765754425047,  477447881571476683685292986319220812003025092761384569 35,
340884652087284349888335146130689646412824896588924579366,  760167521494960830738560212479974162106718303519768948 5,
247778806306428853391309470056837479755377421916735645 5,  413982040531626216390093374218872866600012579566888369256,
431980496606229037258542060373615584852045872298398206 6,  136847435061472762097543373996617679203808236304510840 6,
492591282784932562408455438148712478983442977790092018 95,  481495132602213046108295016857072248831686704557905178 19,
497887813272221284563904000532681771704341726303729396 20,  379334182119239111964377777604355244230681456281817577452,
592157477768860637377284756781714977941107580611904298 1,  949510630505716006806756805500015384390936421917427286 8,
480889694436020139813753463938913149807752959511521478 18,  541711786339081605086291500959545230512027225546935461 6,
131524159712500049687999101455823613542461855097955171 99,  555381515204411810056595267320456795635210740945087251 24,
285408041027823673936217753698218738697313616542688008 00,  433475301342683714522411496220472632926387654694478820,
164777629032033561965170442269383360358396229057066265833,  600521688637731715507092232618810667488221822411423267 28,
858539526046770114695553524101458043949994103348446 1899,  241711625120560432727877074734529379432567750501317709 05,
475546925322039838568456731587575276476316338302184133 77,  206330268177460336393717648521551073424502775507513649 15,
218357223045746286702150523565194172110513339967620312 0,  536195659601629784791950661601827619912636649165357711 42,
504081762180809153553040660545821176131429473600768190 24,  470427450622227242099397673025788395331111020456074425 32,
458917242783784272578399324034625713263992840362619620 45,  240561104432684239064968506162123757794193903440472651,
458325538705303152649625465250706426929861682312114344 6,  670182802949486225717548458851063704612126598960158550 0,
663617677036287185246483864346467183259712459190324758,  557148165723905978792153776257728878410943799987264512 32,
542326689459299981771449457697382189288982470039892886 01,  624810398175744228889914996937807414099405697931429378 4,
786340660090004636946036908322718854973249185567407448,  588219761330288079321132806650905268202905391805123607 67,
414482853452427904029569030729187094682487621216272282 52,  137816493192356035468159368289705085757048955843778707 56,
111810485453261020702223356357224467034092970290273279 31,  360232376618147805015018464764169436609682670116795733 03,
183457960663875156109427708348949236016803529298280159 6,  306558653283071663616968334448396874817607622019833076 91,
129375915070086701065058886916276820416659283504086427 17,  406291648722030180260143565955157535009815463016807147 68,
104621996787196742223999362867971340687405063064702822 23,  254846639074979662093764498482019855748152088836011551 59,
288898340500917982283155402271591178490515693859141990 16,  369731807926848160248117197170012091441021079296165747 78,
333381379655631121382485509640817551447500707699452626 19,  573378130930413389879693919635850849177688723966414375 44,
115119680856453464040594344601897702574412512384769305 72,  518381556737304662418624486565620204219576346857041161 13,
355562479328233316311225278009780616453078872675306238 02,  195211327388895927247657958670456361414685723496727749 73,
261127914542806024638073074976700244997942293716506527 5,  450726147086969266810600353027210646234113390651764747 07,
324502838241594782612819476617970646335190139809048486 01,  554803572998005112854062419860172844026243582383058544 02,
436745055827737652715764917126255088583806563979593230 722,  375938914176459373797366522869617070510553533819854092 83,
160993056474489825494058393143464116698849597730566507 34,  269347129470400627949390471015354491267808726850628201 7,
395180866619165700230463803324872620788442353515281202 06,  280997118085253035337306597486060275259717763590628165 07,
249317185521801247939779989252955913701051997891514536, 911324927608233467186271561227577375931622538566743738,
295166597353597325864944667676054106656431361763686349 26,  604029013525153937214483481100171918819923973457424723 78,
352881756494496512641618066824378983788331121637363940 40,  204526069144052903732772773698705458842379583906949483 11,
479759156307164065100113218245238724029556602325285080 735,  533550671068863780180482282219348385882745627478602948 81,
138928442278448928533300520331625296958886673482429635 7,  230299885035247787443743311128501840769436998581003286 17,
744842146304803361595298465937358556457424913835420903 98,  750264251102490156851574138148596652573429573127229848 0,
115280101045633891332319579125073268926887206805879787 77,  379841514142266653893750047171827374116017328917528528 9,
315405450014305577810664807580796942520692626842105639 68,  286205013349047479699732575592208959058483980894826860 60,
537171920290398758149753522353663981727172769693664314 93,  359566019886462468249917055737694444192648222399237648 17,
485157490440518329061779777639478188502851015081610194 78,  355546403853546682687884720401471395315509500614089524 88,
272261698617344838767097046730637605805355812009955325 19,  214127025793967341451904812664278364770408967254911601 82,
266849715803897405671116177063641920041102496557144866 78,  578663650518220271417608421345742830901256786311937905 90,
253043963511987135324631198198324290447201778440324658 5,  378808538036655276207703298477041649032586333798784749 48,
239335985121313198017064568869798478345260229911725758 28,  321960624749527830244986117734528773826530889991866812 54,
505606695328969207469279651177815613606737337523207499 84,  595359344387285698928385675380726132451451072373354944 08,
764641036937765306169089355053256527543293264354235754 7,  107875933082840056387432348448808360633496608725406189 92,
541664219803891123473756168264948773369290192585559284 215,  214156076797626676221512386351874078651754735274926402 39,
150846793893039670009741083877962546289332250811543846 70,  346812094585857692583586514737751650136798883445157894 55,
374996700960237404889269126628854843549037945662525824 25,  448962026619462704031551354526988928687898744807786393 56,
266652179515021896954781868070782120387405753360891967 4,  417628112123167195174995587261534922085067352210122014 4,
223316150993656055202541675753249848677647283012769448 23,  172647148885840945042071902753473813805683863316056801 72,
352193538922689921276044516703076034415295753188128405 41,  278006575572285879611949853893407664929954762070591393 09,
423367898388282563476075053332259757767286504262313226 91,  235945326994360540267386411090449354361959533469576107,
525898028823553771947209458796343619644141483573346659 10,  603676807244593203404600415939694845575725627983314375 5,
329642521171111390602115286125473053810282482043784975 03,  376717622330013725484553882591623715287502630280606553 07,
596518249319801323070684001508972978350271884935920386 28,  338690908029337767864744662779394424198129516559747268 5,
492034216035952043948037384539675086925592777389153168 8,  149734399742591612031892689797711875065765991812282910 64,
492317626918113389414288689439321957661344893832432201 57,  523778527267443857811664004377404226473060293397074586 90,
302526872618901537946964363045927719953856134314903981 19,  464318748487875598672400146568786205732274730165626804 88,
304093825180426295284169679640317096749836670446639588, 116090593200189735016369521442493796824289730785046744 35,
511556731195650059474807338396178716554990691089228770 9,  249836955743109178126184442403313671858990623357555214 42,
300705685660139033104208856134097410322468493566011244 59,  596036836752965979608721582663425816973920043591970651 901,
422570791819918485159856747007053878377450245691920053 87,  278601870821559997295005967261715596446719507479784612 75,
532000803528388196864340762698731678394219823004233033 67,  154656045222287631653044648683485654344565222959090293 58,
208946239241085690290307654861234314569550020826613237 03,  403406357605890102152603454442484963796049239458431241 14,
364791089906065983299884520084438874891803004591838416 94,  355153554877832501425494691304462503918304319574373119 43,
199520726139871351512672246056955879174876472882894787 61,  526158318676768113051463996322681466322695511971216628 193,
396337619988815220332645411427673672453345663507620755 33,  596950937892670299932999677229504880496325967393589282 62,
303023863049704953023464882238096777151144021327027423 46,  161776534662165485701539229978143066004862178391899222 69,
```

```
337644427361087502280896748133281788239231931749547238 28, 232490329851449279937877013895777438251325579402988001 25,
429593770312329352173037100980024145319986580343069492 98, 202141752856652090656104927734840091395076044392761290 21,
333218958111384360991605820243668652323941584090473764 25, 194679325533564164406288458096810981199082214905906288 01,
472730711288469680584868039974611447527843956514544944 91, 242355722216377751195220800283726410749546610956440636 03,
457484300098253186232140910481055252674929214107063866 47, 211072855842932069428780731841850261224625381978391240 89,
165222773058551220197097705378858364490486169200369916 3, 135839632876075912921845472165917388183138603784571321 64,
337723899313299277145580907585085199313391704185752647 25, 524414053034138358639777083968840297382006942252045714 13,
386076884495593279597158271559289216278468431194053048 25, 295832114219357156070428792337292574803374534213928305 1,
280527164187106551485231421709198312656829670505406723 74, 224814901958878237884900842841098232749168012465436236 02,
563428097672863655846712174751407164541084883549954383 25, 249404460067226340881678146091971960761944020792708212 2,
363636374215632370185721174303985176974942677070405182 780, 145926475156180445075129956548309722773327328836446 463,
616145581541231557011264676891675682502326338765357254 0, 149865613737718262832950373394402576956185270739666478 91,
411869141440328513367971856473643823363897885064564481 08, 478667820906085379703722061206403642014381187181721284 86,
392508737381961960954322681031181950071564648562057171 88, 236314664504028944239753827413131908896938006840322692 28,
259790621368744281550700653199587324047105086473809648 81, 510352797101328219421101265793503554569903751160532827 00,
343394185014522923894546715876625655557084476972427155 72, 128332469289864510278560371852141788709119295042021027 93,
148166882280055461439271803698278463261283354879410203 10, 519546558340131350984033381580277572070323462229154825 99,
528560161248449933677081371245605005863409820796266377 10, 148523691719185434195785796632799256877543199945557122 71,
236679508913293072587706826381357044397311295391353190 99, 187826619015962050151210411070605858153624530163572749 26,
435218701247061374164312539961362330239503032226505318 95, 547309785248219551880047429354839999440732531626740614 055,
183942052061258558903662391364938218041823572824498195 14, 365467877149744277137459608687877195829989051224422347 54,
369592871791969517562959828000217510966324555497899017 53, 234727012559550394508006658920975108329681106370890154 24,
362941891340505342457679906761389213970316346352861859 84, 486822296282870916340068147756056743150012593083565668 37,
251455690472833132345142725690936987090690383805286789 69, 335326118482595470409804596396422049126117297885150183 3,
377123227232404742675727618046176895313161791496601226 54, 573831324831631357811946804311822397491237146928105333 95,
581510550270135415387483107448054822646401025169674348, 39707451551677927546911500338128195261817383073724910 60,
307552379882501089497676179261799520138246401203861411 13, 400736483644519540365664330338056855277971945918810989,
297243379225949923208058481519249212076319942512996318 78, 252631148763866279396910739090418629486565150426782152 24,
419371675243310243256203448067697584847349193168904665 97, 120342854525505636755654378503863549951243032335920636 70,
492907324271412710462827675421189039098704614185024812 50], 211, 24], [157+59*x
+6*x^3+149*x^2+109*x^11+158*x^10+34*x^9+91*x^8+180*x^7+42*x^6+65*x^5+184*x^4+
131*x^16+20*x^15+205*x^14+186*x^13+110*x^12+x^24+77*x^23+206*x^22+116*x^21+85*
x^20+78*x^19+54*x^18+42*x^17, 167*x^23+159*x^22+37*x^21+40*x^20+163*x^19+7*x^
18+28*x^17+205*x^16+22*x^15+143*x^14+158*x^13+209*x^12+114*x^11+198*x^10+59*x^
9+165*x^8+135*x^7+73*x^6+121*x^5+61*x^4+139*x^3+178*x^2+8*x+42, [60, 189, 24,
3, 179, 12, 13, 163, 8, 138, 136, 195, 190, 169, 64, 203, 48, 82, 55, 150, 199
, 21, 88, 10, 108, 122, 141, 92, 37, 209, 95, 160, 87, 178, 176, 4, 200, 17,
38, 61, 40, 0, 29, 130, 15, 102, 46, 28, 121, 96, 33, 170, 52, 51, 66, 126, 25
, 206, 58, 106, 93, 115, 19, 23, 112, 65, 42, 56, 111, 68, 196, 54, 177, 86,
31, 75, 162, 78, 148, 194, 77, 70, 53, 83, 134, 173, 18, 72, 27, 20, 165, 43,
22, 140, 49, 36, 131, 90, 98, 99, 191, 204, 151, 79, 73, 149, 57, 91, 67, 201,
110, 129, 128, 187, 114, 168, 132, 103, 105, 85, 120, 154, 116, 119, 101, 125,
184, 127, 124, 107, 167, 26, 32, 175, 41, 139, 158, 137, 9, 135, 186, 198, 81,
80, 144, 145, 205, 147, 161, 97, 59, 192, 207, 153, 30, 142, 156, 71, 157, 159
, 1, 152, 174, 155, 104, 146, 166, 62, 6, 5, 44, 181, 113, 171, 172, 16, 39,
133, 47, 100, 180, 69, 182, 45, 50, 74, 35, 84, 193, 94, 117, 63, 34, 188, 210
, 109, 202, 197, 143, 185, 123, 164, 118, 14, 11, 2, 183, 76, 89, 7, 208],
181468777279541570339928242512301802198023662780996325 27]];

K277_24 := [[[46030529922907525999571741781497342782990176813190819523 32, 14793257678458602195013842969822863804149235599901955783104,
340626312966318704268845782730677687921883686214401059991 40, 181235105364220114726269825196452184601084606924735926225 11,
256504722041786150606725470713272488406915167934599722790, 337344734608643803285711498344518342297174152074874584877 68,
224701077317512880644215152506035077316478262839990316245 6, 469548045270711108441278067275074903793633336763764168 239,
282313370252530178792227594908313285659979824512570350996 30, 473352778645397173320299505373046227091228154070467522110 5,
140505408565949682200484452715288109856204971017689815956 93, 276480139634891703736529342059102118486778934106760575121 30,
360536598149134145817865363161163016313583001565907387130 09, 516879934243168217967137331152513271107552849975855426507 27,
213780592310321338758353660653652450095087176626180931600 17, 507605278083078602460175623090811300304056138092951927 567,
207880716801046602729385089731768323324326837961581884306 86, 746598095046143562293972080008959864529006511844996369 9360,
206730043118148183683258689836102410628638329148011988359 4, 258794845154612826758595169464468658590307385805537491814 89,
357556693453518023158859512070905755542990400194911296965 61, 945157736262052908455779683751659106201871722699762738 2337,
198812645183463994729189312556507427072840222917012102 6426414, 381098882296731948273685198359824003057113915408279425970 13,
437466041482634478319127876806269708761123790888546065545 3, 401258068804053296745164499092083133377387593840163091925 38,
341732408128104407032509452997215701389286402350402584366 8, 336359680738515819784422030583553618654768448233897327 1584,
412231080373862024974433612609031728774788116990109415928 53, 372424877064938411405320548322769917037165167642610251 20644,
292523742578595116669407022318036527331903966290388667862 24, 254936392713765001565734322263787055833996405280173073835 13,
275643026541620459403965203139022730920408799317 1386569724, 760120918422346803669169493704497456643071685808629922915 38,
664058074100938915952993993041775042322948446769579726666 7, 405907625341499445109455290221182871190547740092384824747 672,
270475192061170352094598187331098796460254127762606087248 91, 233506419206487675256561393432376426710670757919464218767 3,
127956285566147644056964217627997786771962420393245667744 37, 310626553906465461515227939394131198549566650298856931639 7,
162160077227352369054615854629129512980822357258176136381 34, 372639948971833163460796066642485411163136853987949616146 94,
401541001582630624651631107052873875380446233475243585934916, 177283339574350913720064002532133188020900923256470305781 48,
248547018330543116176327764197910787475378784102437643233 51, 114719087382790003141429111657458616935249011830900601830 16,
184411446364897168748724954296673412331768850932479365529 33, 385167182942483355007672892293494760503556918275446437953 31,
343656822483947917331551878948189354064712519492989902557 39, 272855851353766120102342484309970170815200386873494222764 56,
374166488293452303526037408241765358041864873250334011004 32, 332277411020970343662312253921553051423780483496742941866 77,
957318729217061540099644882060591171043896140895809227 181, 104838521619425488433855024799726051307724601680901349227 95,
257564127066399736864075636213556812204910094428356607146 72, 411848160925056026906767360682614076874917392569508561213 61,
302528620435332265679796166768644624357630144850426419019 01, 317274562813050146072822101021741358106936138570700068280,
110498401359523400214486692960872941644672102736659589431 19, 300523090087075285721879970917965930892413011764003213224 19,
101138109863563401523902641478592179399533078663415785790 53, 118585747854986916377530279109889260679697351155117631853 76,
162840956238046468063860807098015999328063297654114407827 84, 305393933827024250692798434866713493323612893028397915945 24,
394234120315038817762833155177273566863008868912252720246 89, 386329155902105946549616581082526767272404573100931680009 00,
387653643208708099733161541869264679815516151405875673964 39, 346418124859203638660559726776770763581564691974216943456 9,
167189087042290733011524562593276789914029080866859679 80902, 277877772836464652384285161973320733777007167252221348701 70,
818840647961187849864675462960332652788377259677625886 69539, 364792924643196798277886634195798347296934835123883973466 74,
409106829420851979612983439178532266549429250062625142 91802, 207633790973666619675612490515261473263848966282484664931 90,
347653183925292522632286117877562564281047478530789160741 4, 356042240711136815372599561066534908864888269216883682315 63,
141729070976047889337108339182415353203732367930697409267 32, 376752862657185196236748726942849833469476367002999178457 98,
897061628752981112465445489821964539825963970334204129027 6, 289746955586170413488821939585027515655186442536489781334,
325653141113384124978605403216506363477302364259644320715 57, 634753139993911821183437812907507199378072306432386022924,
369266971214697583826171783785820012233662151580559554885 7, 869675667033035673465673372530089432151017298021809137011 0,
314293807182110224986801052014592679600459784807056237849 42, 197961398628492054674993140689515949500214227548970632737 96,
773818735274065709045537115585403349842893344474675771892, 121218907696155261944647912773472284088191211831635884003 59,
```

237418603349022056841769898180342831767892339987107178830 3, 210814472981383423738218666924054626946702756551430057656 07,
998584628331495972222378577357630480693956323719795745599 3, 856487549614428145121622963158681881041236883422547750041 9,
375407410412935154633752563252031191936716986627142540530 15, 109409789180084624127181684836873495646201274567411936536 68,
240866315767257315583088972382104894559162782505988260012 01, 202608553699957176378385061195919019874005875715235153694 15,
233155695842040818347191933269533063651318098153021971995 31, 413190356345658668518149027011813562819079071087337658063 06,
354903492203942992231367932316463789767979510575219335161 73, 158719872258929755698978398857685394238827997315381826983 18,
397470452708434538940732399879995219859547014299276372526 48, 785979732630078064407604131149345509121897493831928784584 9,
415078346038573227948781490890908708810708429593013588178 93, 382877615859241751417381705674496814503233011543073451188 5,
409284500112809110630676781693641354906742678076341632206 62, 465535899668726007293042760225088830536317367484921587205 ,
381357381268240752696290434068500845260361409735429569537 93, 118994807428425637945602958677763942040807067566401601061 98,
308225445708609021969192176008858972891367247132150022424 92, 417357808199159794733489620410575782505865301039940021181 0,
157940532602745774012780674497153608844032430493267987474 14, 299972337846185315669285405020585472815316003320494680739 ,
119539761480843639147993499874096910990879347249974455800 92, 205291739850533089236188519015315492189391589709845360184 51,
293736903378833773139312811581205354650970675872246681514 23, 361007195102110968119488502366530344652686226634536752308 487,
356377409914786952960861987729090816485174354041058828735 19, 248873378144553871121797101264338784227587269775101949829 67,
108451091073264576104544963796550528878241464000405347415 , 971132361074272127981226812457276900003079628930177268618 7,
173736694027367455666836976424614128711687363783367432047 49, 170518217110710304442477656528370951435853944776356474848 98,
314281406550419625065376806122142155525867233282620709835 16, 348154131924845591799866455813690288734819651444068516887 77,
405456821951762782140064891819618301843782221212225892720 09, 176241486908814664155863659569788694766425139363444723420 84,
336516461416843588984150398974120967841596147449071142294 00, 155383358204179214964531013208374506957012757348247539488 25,
141414131697908799157905883914159640720804265711538715180 6, 414387012468695575340103573714210407188328846226765334458 64,
123831207646294821672223725173468385639820710527285959353 17, 508374097702149566390927599524889895976589485390679423340 3,
338209761139099372563279165011060047388743360884673228852 28, 389529859752183534158788130934635949027780300961930945565 23,
341725276904495638313274578061914389166702484618935399709 86, 416365572122497549257043418482747483766062195214220856111 81,
180564615656264809272376006196967621611579681136455730117 5, 387746592308459606693234765262874908639443309351027895537 38,
160673846222792530466985558180922815342760390591978170342 82, 932979731538967324694002633022753528645433623074330837479 7,
249265414885882935539939175511160774049532528298434123649 21, 109607878895374706664945899535864956686335479683824710537 ,
388585836040058812145869857848433050233040430272561506576 27, 387656546656585175206492463269726192011587574021176407733 8,
243176594199325506262940148980121718826136198321664905757 49, 250726477020755267095834421654375148046657995977843163742 14,
391947191114679978569097735678260309556217927247011492091 09, 344689796585303836941058183817536294528614028416917193967037 ,
162816536132147964919241226477485015472036266593187723316 24, 168675137118721125958388267460521120750914622931281927077 41,
146374358831452314355967282946777854242328926564291194187 8, 636021542174867967119358148279303582467584641696189618857 7,
218713221568112735050109808112543075384556471123988630238 85, 132557447033003152964951519638421563451991108133225073400 35,
332702922639390076110357964692971709228345282023459962122 11, 209682487840955705241087881627931147506045657659589453935 08,
368473948326376924550922027971502513474613148118063216049 34, 181040726311849697237975925573130192647333819976469220914 42,
339972306032958874153998591870636555629194228987784773053 46, 733230571683614754413616786747342680058856077145460099635 2,
161404607916061552610348669382590669676022763779233556240 73, 282450818001396633694262109636799505501850772859143160306 55,
256560305675715278771611926215642943674294514326654046806 40, 790936942028491745741418259757333084876081695156724352852 5,
389434001920365127161971741981734831240261099260241472897 95, 117942421741661835186630839286762956707801963774767486113 68,
267213089592163605077747987935921824065425163669710970469 98, 197911045643737791790497598501505144702806592048966844283 30,
372768523881209481920911585862484779443313438572188921278 8, 123303892925304161422000813763312821040432749955852163694 06,
336510452341157709003530510329067503540497958361528287898 01, 240974153226678896066766132126234580654010714036946290705 3,
209553311829257327497517196122290425614986394111047442464 53, 115856577054141535185708422244802812783780055757089899154 10,
520323940777980220636568944885013856404522465100541877738 , 110426458764432534561814444298686842481942117810301573184 60,
314049861536104737759393542175072777254329922275263046471 25, 720552310472801271293459857374390538476741176175206099738 2,
147564833401451578829215998795654341332537935299869043435 46, 393994877398244879195497513836959770451914375918801586539 30,
312138051751090954599949811069087860935705836923094308624 3, 744856386108231605934279425894406932475067846984210435889 4,
720565949487253377685877710696285285586164046538067967560 , 225941980405121266003046986591708896472077617741418798259 75,
219565563625258143932280089779765446910118810895413574164 15, 190209747510432880413336376010468420306042893016362894184 08,
295374184456026999118990153654336911308469949821420332590 94, 318806155405715016854680956806967068193366794673268385795 4,
406492786739725118585483887229963519546191269688225123417 91, 238516166050005621457066254152888500098820032037161681221 ,
606530632093206136203666187117541458891598579329267020354 , 195351349145931087140683255630831683273656801451292388359 97,
177217902757676730005628887863303774012960182048415011304 74, 375511883194424167244332099486606572941534579359245265822 71,
170689102474506329489752214020585215645260278043551036471 47, 300231701898279979876424512285931713808772751779694608907 17,
333325514792520308556060504247345183607383869075854303664 96, 758976922384540054009912587552232486770165591170454017832 3,
314467143324226197507621730165954137514680117943980919480 34, 156755694975720707979687940451884333216380439608131286244 56,
126354192678070537718556803061128109384694439640166959607 61, 268708032585465603716253016139664767555053669420737492780 34,
620076365098752473013844473695292866223518853286569290703 1, 189101613027321146121795617013883700076513270831600668519 51,
400391946025216185938147519542645489490936329736342125095375, 32695714960907390342175893249066572563536380047799616329 45,
361060502083898307174038673892342828207983143870187483661 57, 209351898547312702497267423609373640051368601969283642250356 ,
227877811177587966475444321725779238476360616727789220707 97, 338802450056092030227644466929546724008853249559199310800 8,
230651801359991055164796459935319026651393048922381805201 98, 360308557938820342351504809093345144814301453873039898668751 ,
179065364337837313548383175363396500048781094950276733997682 , 372054287511643047182862164754496646872019039697250524419 44,
684580650830224695952513311630643408453147330470593581447 1, 336207677926543785095945284075919193461760395985415052893 67,
414206319556815975986404498396652991628173285237205583033 12, 302651842435518576262267537423805937255404225039429815791 96,
390696217192838076647770899223312632821153466512678072870 1, 218472949679385252200481864852743063306598975379064708736 01,
225849357512536271819357123184595799332815392731127746502 03, 114137790683997156451098734128363408260768169278466227902 32,
290761145848605191623867473473950282330008597273880814191479 , 243725963690285771993307405134893053740884763764819490899 9,
246700780359743565928616354080387960004169134035301452667 38, 133074666831331039759203365576656294211262230556226586607 ,
376546304323332748473054555313977698270577743161374460544 66, 190633623587317400066823775682543927384440422810143964877 88,
881664043438168460087461472451872657975845793842326990781 8, 313706784874952031014439237181254212211071725276069321644 0,
414335812320978609774257207212493178929705898686946646 4580208 , 114136971863357795858268611560059834514704929494568580242 79,
186488173061840592013104399527301818201048440604472678162 53, 228266413379135070188498598997758907541584367577242442703 34,
340367626000014141125264912477006950726898936491624377684703 , 373889628534361725812574637444686813643129612644594496513 41,
131666102511008366643614347093167040475001629852940655818 32, 456285806876721094070303161771016726722742220922571166393 ,
144860031148611393983005198354790562172845433521914724998 91, 260092595777093470528559012193594595177310002559107327952 39,
174998687950526547882825203434265303138244723823479518074 07, 401036303014300968452063277088393237090933866813379679459 6,
501269094917614864484486457564293096682103182751767839497 1, 306330018302619551997645004547482216581879785435280576326 09,
146078708546373005944282182644596911169364178652794794898 24, 340349600048245137662684644126288910107722511071293778948 2,
411506645512797838486060503269384243482333829167456110705 30, 721378207733568560998029740717320031930152965270892721693 ,
499458704973927495889850853128932675969643811657513345219 3, 307262400712733077825683358066613213197135277008348645079 2,
354425533967846692724107092441375879811356340956030104919 36, 182003271145247769886771203704326403004166312697632214590 65,
866515121170723532595340328915780834966408910099459463327 , 253934571167312131013154654359861394690004596207653554773452 ,
188834518393941220327501485241012102339785394974364064978 65, 369108712272588197820030347430398335254357090341962393986 87,
167861864308912235686443295949162683837952570674693665549 28, 195700322614020813103096036696910634355541124818748498282959 ,
336387093619439593813920452269401190861274594262222579815 , 288707779575912354990769516981486495432802006472316448816035 ,
160466319744586444488753198532138600970641439643687836305 30, 147657374495100250568659514423610411080356735457031657398 53,
214268682638477128447395589518461363613386044643775833319 51, 721389464995435304065954162961525889184919682354259458140 0,
170998904284392990375473914357453637630288036137626651810 14, 301847329526207143552047124796304317864111367930959360122 ,
258316521569417222218240012715138539626993843505735210201 69, 108895361648693426718159876317371237042814354434802767413 80,
718671380958164394884713009736896971877736691742472703881 7, 409696588344420614792460273340672689501639158259463252520 8,
390153341587503612257740927010598012359535017629186752104800224296307382306697483051325 1,
391442940762530852156920512729545485067009633254261970734 67, 283657547138035557975813970259403322409563664433957374262 34,

```
14999034492182679561582662539605880565393487274884941209792, 13093962431520953879282884448881414282206329843151520834939,
16379886106173543293511089569585582021089096489095496172879], 277, 24], [19+
269*x+181*x^23+122*x^22+220*x^21+185*x^20+147*x^19+47*x^18+204*x^17+129*x^16+
13*x^15+275*x^14+148*x^13+28*x^12+101*x^11+164*x^10+205*x^9+9*x^8+49*x^7+14*x^
6+27*x^5+140*x^4+93*x^3+256*x^2+x^24, 152*x^23+122*x^22+131*x^21+55*x^20+94*x^
19+67*x^18+142*x^17+59*x^16+24*x^15+65*x^14+140*x^13+266*x^12+254*x^11+209*x^
10+98*x^9+56*x^8+245*x^7+271*x^6+193*x^5+134*x^4+194*x^3+153*x^2+90*x+88, [74,
239, 216, 36, 132, 185, 193, 162, 257, 65, 269, 157, 59, 182, 242, 71, 16, 124
, 154, 38, 88, 4, 184, 169, 91, 52, 133, 155, 261, 229, 56, 221, 107, 77, 34,
35, 265, 115, 96, 167, 266, 41, 84, 118, 112, 45, 51, 47, 48, 105, 113, 200,
62, 53, 54, 123, 90, 104, 258, 83, 68, 138, 22, 79, 1, 102, 273, 89, 70, 69,
268, 160, 111, 236, 24, 75, 43, 190, 2, 228, 93, 67, 217, 237, 80, 92, 50, 208
, 97, 161, 227, 42, 170, 21, 40, 151, 119, 141, 181, 211, 46, 58, 136, 215, 31
, 98, 226, 168, 174, 109, 5, 255, 116, 204, 139, 11, 246, 117, 78, 30, 120,
260, 3, 114, 17, 125, 196, 127, 61, 66, 275, 177, 12, 233, 218, 244, 85, 137,
259, 199, 163, 210, 172, 254, 126, 103, 146, 23, 95, 18, 33, 148, 235, 153,
149, 250, 87, 179, 26, 222, 180, 276, 7, 140, 144, 165, 166, 209, 186, 231,
147, 158, 189, 230, 142, 128, 27, 251, 212, 73, 82, 272, 13, 108, 241, 14, 60,
187, 183, 194, 106, 188, 192, 8, 219, 224, 271, 63, 39, 15, 152, 201, 202, 203
, 207, 205, 72, 173, 248, 178, 253, 263, 198, 213, 86, 150, 25, 29, 156, 214,
191, 44, 274, 223, 240, 176, 57, 49, 195, 131, 99, 101, 232, 171, 110, 129,
164, 175, 134, 220, 64, 81, 76, 9, 159, 245, 6, 135, 238, 32, 262, 249, 252,
206, 55, 247, 243, 19, 20, 130, 121, 28, 197, 122, 264, 225, 234, 267, 0, 256,
270, 100, 10, 94, 143, 145, 37],
21045880846925396966886276423092950528034548018513778037357]];

f13_12 := x^12+4*x^11+2*x^10+8*x^8+10*x^7+5*x^6+9*x^5+7*x^4+10*x^3
          +5*x^2+7*x+7;
f17_12 := x^12+5*x^11+12*x^10+13*x^9+3*x^8+6*x^7+11*x^5+5*x^4+2*x^3
          +8*x^2+9*x+14;
f19_12 := x^12+17*x^11+15*x^10+18*x^9+14*x^8+9*x^7+14*x^6+4*x^5+6*x^4
          +6*x^3+8*x^2+17*x+2;
f23_12 := x^12+6*x^11+20*x^10+2*x^9+3*x^8+11*x^7+12*x^6+13*x^5+8*x^4
          +7*x^3+17*x^2+7*x+14;
f29_12 := x^12+14*x^11+13*x^10+15*x^9+12*x^8+23*x^7+14*x^6+21*x^5
          +17*x^4+9*x^3+4*x^2+15*x+8;
f31_12 := x^12+22*x^11+23*x^10+17*x^9+16*x^8+7*x^7+17*x^6+4*x^5+15*x^4
          +28*x^3+4*x^2+16*x+22;
f37_12 := x^12+7*x^11+2*x^10+3*x^9+25*x^8+35*x^7+31*x^6+22*x^5+8*x^4
          +6*x^3+34*x^2+22*x+17;
f41_12 := x^12+31*x^11+38*x^10+32*x^9+24*x^8+37*x^7+39*x^6+2*x^5+38*x^4
          +38*x^3+3*x^2+10*x+13;
f43_12 := x^12+42*x^11+x^10+40*x^9+10*x^8+37*x^7+9*x^6+19*x^5+2*x^4
          +26*x^3+30*x^2+10*x+20;
f47_12 := x^12+17*x^11+39*x^10+36*x^9+25*x^8+4*x^7+32*x^6+11*x^5+33*x^4
          +17*x^3+34*x^2+28*x+10;
f53_12 := x^12+8*x^11+29*x^10+32*x^9+19*x^8+x^7+24*x^6+51*x^5+26*x^4
          +28*x^3+32*x^2+37*x+5;
f103_12 := x^12+84*x^11+38*x^10+60*x^9+63*x^8+4*x^7+12*x^6+41*x^5
           +89*x^4+65*x^3+94*x^2+55*x+54;
f197_12 := x^12+179*x^11+167*x^10+128*x^9+70*x^8+188*x^7+139*x^6+48*x^5
           +72*x^4+192*x^3+185*x^2+83*x+99;
f43_24 := x^24+2*x^23+x^22+29*x^21+33*x^20+42*x^19+2*x^18+35*x^17
          +28*x^16+29*x^15+20*x^14+11*x^12+2*x^11+33*x^10+4*x^9+35*x^8
          +12*x^7+10*x^6+4*x^5+23*x^4+20*x^3+9*x^2+35*x+3;
f79_24 := x^24+33*x^22+53*x^21+72*x^20+64*x^19+26*x^18+60*x^17+37*x^16
          +47*x^15+72*x^14+6*x^13+13*x^12+63*x^11+41*x^10+66*x^9+74*x^8
          +14*x^7+9*x^6+x^5+74*x^4+14*x^3+20*x^2+14*x+30;
f139_24 := 56+115*x+87*x^12+6*x^11+98*x^10+19*x^8+10*x^7+56*x^6+2*x^5
           +88*x^4+99*x^3+x^2+26*x^9+68*x^23+103*x^22+88*x^21+9*x^20
           +27*x^19+4*x^18+41*x^17+7*x^16+x^15+76*x^14+x^24+95*x^13;
f197_24 := x^24+18*x^23+146*x^22+100*x^21+27*x^20+163*x^19+64*x^17
           +139*x^16+98*x^15+139*x^14+34*x^13+134*x^12+11*x^11+56*x^10
           +68*x^9+182*x^8+31*x^7+43*x^6+103*x^5+59*x^4+81*x^3+21*x^2
           +182*x+58;
f211_24 := 202+201*x+x^24+181*x^23+42*x^22+28*x^21+17*x^20+5*x^19
           +13*x^18+170*x^17+175*x^16+56*x^15+128*x^14+143*x^13
           +152*x^12+87*x^11+32*x^10+24*x^9+95*x^8+44*x^7+168*x^6
           +181*x^5+31*x^4+201*x^3+109*x^2;
f277_24 := 127+233*x+x^24+26*x^23+57*x^22+210*x^21+267*x^20+118*x^19
           +240*x^18+19*x^17+97*x^16+205*x^15+14*x^14+49*x^13+181*x^12
           +204*x^11+102*x^10+91*x^9+29*x^8+68*x^7+118*x^6+24*x^5
           +230*x^4+274*x^3+7*x^2;

u_6_13_12 := [2146069];
u_6_17_12 := [9147461];
u_6_19_12 := [20261663];
u_6_23_12 := [40295485];
u_6_29_12 := [469718521];
u_6_31_12 := [573875891];
u_6_37_12 := [368048927];
u_6_41_12 := [2471414359];
u_6_43_12 := [5344795325];
u_6_47_12 := [3480254201];
u_6_53_12 := [13516761305];
u_6_103_12 := [622140681191];
u_6_197_12 := [25301312285267];
u_6_43_24 := [2140699621];
u_6_79_24 := [142816379921];
u_6_139_24 := [3289127455379];
u_6_197_24 := [35912495900639];
u_6_211_24 := [261697384122 89];
u_6_277_24 := [77574583728737];
```

2. RSA-Beispiele

```
# pqd ist eine Liste mit 30 RSA-Schluesseln [p,q,d], wobei N=p*q
# 256 Bits hat und 0.25<ln(d)/ln(N)<0.30 gilt
pqd:=[
[188374999728520210871080603607360213419,3258791190742271972136743628076284526 39,
 25905722225349661637],
[20658585182930079377091169597545185357 9,29285637432357020719138089711246406177 3,
 22024801136906754887],
[22457355670353761351856484051588840726 9,40127159871700796053694222799575822989 7,
 21695608100616834497],
[22586642748989942271487341238460006375 9,31643691385940500726188737440233543538 1,
 45754907352297080797],
[23792658189785281098209166386741782723 3,35175756469391578704139867847687757374 9,
 63391160468813863933],
[18283245105627152829963968171026069295 9,34042841363471670642157795334819081398 9,
 66753327148181927197],
[23047795760349549563673182112096925288 3,37319015061490083040597412816811726856 3,
 174635575560106336567],
[23864046054655888863089729645688087987 7,43955473924227424319648910772641974641 7,
 255932416271350477589],
[19500221029913861749684557166901440611 7,35368684857737578007303468081563773607 1,
 112327734315375269027],
[22213236822799507538753797499868828481 1,34566268337412801388321083886750248165 7,
 455512228672814695493],
[22591365601485995886493230874724687825 3,38116795261766224091345841780602961132 1,
 251243047948357683157],
[21148685318405696458608649219616174423 9,36244980602506936281325451226628121866 1,
 353048686012604807563],
[23968688455766961183213025138632772087 3,28981133907483277942067585735552877826 9,
 773077208755641942239],
[21327324855169616285392227801125013457 3,36910520219890422836547226172587297453 7,
 113418446451635634970 1],
[18899796527279457625336070171745391247 9,35682569352614430078406437087019450416 1,
 802336824063049910353],
[22228125240995635530946842973656327847 9,37148785509367711128381506464444705903 1,
 2167042358239658416289],
[23951324254432200691216300570949168616 7,34453845229989933019479036223149406822 1,
 2862105187103316043097],
[19383433032874878295872044840952344957 3,30497236447842567745700252883960263481 1,
 1471013299145662220969],
[20474866699616654482236700030928596285 1,32665411993444468887026747336375843597 9,
 5564242900616314338459],
[22904360261493502196540995574162402557 7,35866725345291389493354262057990320593 3,
 4080239307380162890731],
[19998507749205660226077487947488336914 9,36762798462779341595791897447265556743 3,
 3467921838253580955629],
[23735416657063756414440363375091457922 7,28719799454651132765350985017799004115 3,
 16896108675169172990125],
[21292172873756060212822496806330364395 9,38999400058510803035330130012830209733 9,
 12359880407518772763959],
[21529931685201266132376849791943463880 3,40431809544375724777003381335375383504 9,
 14208824835589388691903],
[20494231865685669218174589698160302455 7,32106487220355919718958813543223896388 3,
 3199056654043432222495 9],
[24001144912169210350270836278031986170 1,37349213798707674746400689379988630705 9,
 24316664735890768303427],
[23757195099048231344731728098738737264 7,36942356322028668758166769568497924618 3,
 27882173530119194277931],
[22070088631132409606104354075667743583 9,31211648943525265957508652677952830712 1,
```

```
 1025548912501677407560 87],
[2362207417262346632894206847695617636 19,4237910270490220592509573248131766306 19,
 8837546814979595576281 9],
[2230766986826299314665229342681477795 49,2792998243863780941084675356817679024 01,
 4557791409587623043799 7]
];
```

Literaturverzeichnis

[BoDu] D. Boneh, G. Durfee, Cryptanalysis of RSA with Private Key d Less Than $N^{0.292}$.

[BrOd] E. F. Brickell, A. M. Odlyzko, Cryptanalysis: A Survey of Recent Results.

[ChRi] B. Chor, R. L. Rivest, A Knapsack Type Public Key Cryptosystem Based on Arithmetic in Finite Fields, Advances in Cryptology: Proceedings of CRYPTO 84, Springer-Verlag, 1985, pp. 54-65.

[Co] H. Cohen, A Course in Computational Algebraic Number Theory, Graduate Texts in Mathematics **138**, Springer-Verlag, 1993.

[CoLaOdSc] M. J. Coster, B. A. LaMacchia, A. M. Odlyzko, C. P. Schnorr, An Improved Low-Density Subset Sum Algorithm.

[MaOd] J. E. Mazo, A. M. Odlyzko, Lattice Points in High-Dimensional Spheres.

[MeOoVa] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.

[Od] A. M. Odlyzko, The Rise and Fall of Knapsack Cryptosystems.

[Sc] B. Schneier, Angewandte Kryptographie, Addison-Wesley, 1996.

[ScHo] C. P. Schnorr, H. H. Hörner, Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction.

[Va] S. Vaudenay, Cryptanalysis of the Chor-Rivest Cryptosystem, Advances in Cryptology CRYPTO '98, Santa Barbara, California, USA, Lecture Notes in Computer Science No. 1462, pp. 243-256, Springer-Verlag, 1998. Journal of Cryptology **14** (2001), 87-100.

[Wi] M. Wiener, Cryptanalysis of short RSA secret exponents, IEEE Transactions on Information Theory **36** (1990), 553-558.

Version vom 21. Juli 2002