DIPLOMARBEIT

Thema

Schnelle Algorithmen für große Zahlen und ihre Implementierung in C#

Ausgeführt am Institut für

Diskrete Mathematik und Geometrie

der Technischen Universität Wien

unter der Anleitung von

Ao. Univ. Prof. Dipl. -Ing. Dr. techn. Johann Wiesenbauer

durch

Hannes Glavanovits
Name

7461 Mönchmeierhof 54
Anschrift

Inhaltsverzeichnis

Große Zahlen spielen in der heutigen Computerarithmetik eine entscheidende Rolle. Sie finden Verwendung in der Kryptographie, Faktorisierungsproblemen, Primzahltests, ... In diesen Gebieten werden Berechnungen mit sehr großen Zahlen durchgeführt. Zu diesem Zweck sind korrekte Lösungen mit einer schnellen Implementation nötig. Ein symbolisch korrekter Algorithmus kann eine nichtakzeptable Laufzeit besitzen. Im folgenden werden die bekanntesten und am meisten verbreitesten Algorithmen für die Multiplikationen, Potenzen sowie dem Finden des größten gemeinsamen Teilers vorgestellt. Weiters erfolgt eine Implementierung in C# innerhalb einer Klassenstruktur, welche die Basisoperationen bereitstellt.

1 Einführung (Basisoperationen)

Definition 1.1 *Die Basis-B Darstellung einer positiven, ganzen Zahl x ist die Folge von Werten $0 \leq x_i < B$, für die gilt*

$$x = \sum_{i=0}^{D-1} x_i B^i.$$

Definition 1.2 *Die balancierte Basis-B Darstellung einer positiven, ganzen Zahl ist die Folge von ganzzahligen Stellen $(\bar{x}_i)$, wobei für die $\bar{x}_i$ gilt $-\lfloor \frac{B}{2} \rfloor \leq \bar{x}_i \leq \lfloor \frac{B-1}{2} \rfloor$,*

$$x = \sum_{i=0}^{D-1} \bar{x}_i B^i.$$

Die balancierte Darstellung wird in einigen Algorithmen zur Effizienzsteigerung genutzt.

Die grundlegendsten arithmetischen Operationen sind die Addition, Subtraktion und Multiplikation. Diese, bereits aus der Schule bekannten (klassischen) Methoden zur Berechnung sind simpel und lauten ($z_{-1} = 0$):

- Addition $z = x + y$: $z_n = x_n + y_n +$ dem Übertrag von z_{n-1}
- Subtraktion $z = x + y$ mit $x > y$: $z_n = x_n - y_n -$ dem Übertrag z_{n-1}
- Multiplikation $z = x \cdot y$:

$$z_n = \sum_{i+j=n} x_i y_j \qquad \forall n. \tag{1}$$

  + dem Übertrag k von z_{n-1}.

Das Quadrat $z = x^2$ lautet nach (1)

$$z_n = \sum_{i+j=n} x_i x_j = \sum_{i=0}^{n} x_i x_{n-i} \qquad n \in [0, 2D-2],$$

$$= 2 \sum_{i=0}^{\lfloor n/2 \rfloor} x_i x_{n-i} - \delta_n \qquad \delta_n = \left\{ \begin{array}{cc} 0 & n \text{ ungerade} \\ x_{n/2}^2 & n \text{ gerade} \end{array} \right\} \tag{2}$$

Für das Resultat x^2 fehlt noch der Übertrag der z_n in der Darstellung. Durch diese Symmetrie werden nur halb soviele Multiplikationen benötigt, wie bei der klassischen Multiplikation.

1.1 Division und mod

Definition 1.3 *Bei ganzen Zahlen $x, y \in \mathbb{Z}$ ist die Division als ganzzahliger Anteil von x/y und die modulo-Operation als Rest der Division definiert.*

$$x, y \in \mathbb{Z}, \quad \frac{x}{y} := \left\lfloor \frac{x}{y} \right\rfloor, \quad x \bmod N = x - N \left\lfloor \frac{x}{N} \right\rfloor$$

Die klassische Division für $\frac{x}{N}$:

1. Finden eines $m = B^b N \leq x < B^{b+1} N$
2. Berechnung von $\lfloor \frac{x}{m} \rfloor \in [1, B-1]$
3. Der Quotient $\lfloor \frac{x}{m} \rfloor = q_b$ ist die b-Stelle des Resultats q
4. Reduktion von $x = x - m \lfloor \frac{x}{m} \rfloor$
5. Division von m durch B, $b = b - 1$
6. Sollte $m < N$ dann ist $\lfloor \frac{x}{m} \rfloor = q = \sum_{i=0} q_i B^i$
7. sonst gehe zu (2)

Es existieren viele Implementationen des Divisionsalgorithmus für den Fall $B = 2$, da Optimierungen in diesem Fall direkt die Laufzeit des Algorithmus verkürzen. Im folgenden bezeichnet ein Shift, eine binäre Operation, um links oder rechts, eine Multiplikation bzw. eine Division um 2. Der Vorteil solcher Shifts ist die hardwarenahe Implementierung.

Algorithmus 1.4 (Binäre Division) *berechnet* $\lfloor \frac{x}{N} \rfloor, x \geq N > 0$

```
BinaryDiv {
    // Finden von b  m = 2^b N ≤ x < 2^{b+1} N                    (1)
    m = N, b = 0, c = 0
    while (x ≥ M)
        m = 2m = m << 1, b = b + 1                    Finden von m
    m = m/2 = m >> 1, b = b − 1                       Korrektur
    for 0 ≤ j ≤ b                                              (7)
        c = 2c                          jeweilige Stelle mit 2^j (3)
        a = x − m
        if (a ≥ 0)
            c = c + 1                                          (2)
            x = a                                              (4)
        m = m/2                                                (5)
    return c }
```

Beispiel: $x = 297, N = 59 \Rightarrow b = 2, m = 236, c = 0$

j	$c = 2c$	$a = x - m$	c	x	m
0	0	61	1	61	118
1	2	-57	$-$	$-$	59
2	4	2	5	2	$-$

$\Rightarrow c = \lfloor \frac{297}{59} \rfloor = 5$ und $a = 297 \bmod 59 \equiv 2$

1.2 Newton - Methode

Die Division ist eine Operation mit großen Aufwand, die Vermeidung der Division, hat eine Verkürzung der Laufzeit zur Folge.
Lösung: Newton-Methode, welche ein allgemeines div und mod Schema nur durch Multiplikationen realisiert

Gegeben: $f(x)$
Gesucht: Lösung von $f(x) = 0$ mit geeigneten Startwert x_0

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad n = 0, 1, 2 \ldots \tag{3}$$

konvergiert, für den geeigneten Startwert x_0, gegen die gesuchte Lösung.
Der Umkehrwert $\frac{1}{a}$ einer reellen Zahl $a > 0$ ist die Lösung der Gleichung $\frac{1}{x} = a$ und die Newton-Iteration lautet

$$x_{n+1} = 2x_n - ax_n^2 \tag{4}$$

Um die Newton-Methode auch für ganze Zahlen anzuwenden, wird ein allgemeiner Umkehrwert benötigt.

Definition 1.5 *$B(n)$ liefert die Anzahl der Bits der Zahl n.*
$B(0) = 0, \quad B(1) = 1, \quad B(2) = B(3) = 2, \ldots$

Definition 1.6 *Der allgemeiner Umkehrwert, (Reciprocal), $R(N)$ ist definiert für positive, ganze Zahlen N als $\lfloor \frac{4^{B(N-1)}}{N} \rfloor$.*

Lemma 1.7 *x, N positive ganze Zahlen und $R(N)$ wie in 1.6, dann gilt*

$$\left\lfloor \frac{x}{N} \right\rfloor \sim \left\lfloor \frac{xR}{4^{B(R)-1}} \right\rfloor \tag{5}$$

und

$$x \bmod N \sim x - N \left\lfloor \frac{xR}{4^{B(R)-1}} \right\rfloor, \tag{6}$$

der Fehler ist cN (c klein).

Beweis:
$\lfloor \frac{x}{N} \rfloor = \lfloor \frac{xA}{NA} \rfloor \geq \lfloor \lfloor \frac{A}{N} \rfloor \frac{x}{A} \rfloor = \lfloor \lfloor \frac{4^b}{N} \rfloor \frac{x}{4^b} \rfloor \qquad 2^{b-1} \leq N < 2^b$
$\lfloor 4^{B(N-1)} \rfloor \approx 4^b \Rightarrow \lfloor \frac{4^b}{N} \rfloor \approx R \Rightarrow R = 2^{b+1} + \epsilon \qquad \epsilon$ ist der Fehler
$b = B(N-1) = B(R) - 1 \quad \Rightarrow \lfloor \frac{x}{N} \rfloor \sim \lfloor \frac{xR}{4^{B(R)-1}} \rfloor$

Algorithmus 1.8 (Reciprocal) *berechnet $R(N)$ für N*

```
Reciprocal (N) {
    b = B(N − 1)      r = 2^b      s = r − 1                    Initialisierung
    while (r > s)                                   Diskrete Newton Iteration
        s = r
        r = 2r − ⌊(N/2^b)⌊r^2/2^b⌋⌋
    s = 4^b − Nr                                    Korrektur des Resultats
    while (s < 0)
        r = r − 1
        s = s + N
    return r }
```

Satz 1.9 *Der Algorithmus (1.8) berechnet $R(N)$.*

Beweis: Zitat aus Referenz 1.
Es gilt $2^{b-1} < N \leq 2^{b-1}$, $c = \frac{4^b}{N}$ und somit $R(N) = \lfloor c \rfloor$. Definiere

$$f(r) = 2r - \left\lfloor \frac{N}{2^b} \left\lfloor \frac{r^2}{2^b} \right\rfloor \right\rfloor, \qquad g(r) = 2r - \frac{Nr^2}{4^b} = 2r - \frac{r^2}{c}.$$

Es gilt $\frac{r^2}{c} = \frac{N}{2^b}\frac{r^2}{2^b} = \frac{N}{2^b}(\lfloor \frac{r^2}{2^b} \rfloor + r_1)$, $0 \leq r_1 < 1$ und $\frac{N}{2^b} \leq 1$
$\Rightarrow \lfloor \frac{N}{2^b} \lfloor \frac{r^2}{2^b} \rfloor \rfloor + r_1 + r_2 \quad 0 \leq r_2 < 1$

$$\Rightarrow \left\lfloor \frac{N}{2^b} \left\lfloor \frac{r^2}{2^b} \right\rfloor \right\rfloor > \frac{Nr^2}{4^b} - 2 \qquad \Rightarrow g(r) \leq f(r) < g(r) + 2.$$

Annahme: $f(r) < c + 2$

$$g(r) = c - \frac{(c-r)^2}{c} \Rightarrow c - \frac{(c-r)^2}{c} \leq f(r) < c - \frac{(c-r)^2}{c} + 2 < c + 2$$

Annahme: $r < c => f(r) \geq g(r) = 2r - \frac{r^2}{c} > r \Rightarrow f(r) > r$
D.h. die Folge der Diskreten Newton Iteration in 1.8 von $2^b, f(2^b), f(f(2^b))$, ist streng monoton steigend bis zum Wert s mit $c \leq s < c+2$. Das Resultat der Diskreten Newton Iteration in 1.8 ist $r = f(s), c \leq r < c + 2$. Falls $N = 2^j$ stoppt der Algorithmus 1.8 nach dem ersten Durchlauf und liefert das Ergebnis $r = N$. Der Algorithmus 1.8 terminiert mit $\lfloor c \rfloor$.

Lemma 1.10 *Die Anzahl der Schritte in der Diskreten Newton Iteration in Algorithmus 1.8 ist* $O(\ln(b+1)) = O(\ln\ln(N+2))$. *Die Durchläufe der while-Schleife in Korrektur des Resultats ist* ≤ 2.

Algorithmus 1.11 (Division-free mod) *berechnet* $x \bmod N$ *und* $\lfloor \frac{x}{N} \rfloor$. *Der Umkehrwert* $R = R(N)$ *wird vorher mit Algorithmus 1.8 berechnet.*

DivisionFreeMod {
$s = 2(B(R-1) - 1)$
$q = \lfloor \frac{xR}{2^s} \rfloor$ — Näherung für $\lfloor \frac{x}{N} \rfloor$
$r = x - Nq$ — Näherung für $x \bmod N$
while $(r \geq N)$ — Reduktion des Fehlers cN
$r = r - N, \qquad q = q + 1$
return (q, r) }

Beispiel: $x = 25695, N = 403 \Rightarrow R(N) = 650$
$s = 2(10-1) = 18$, $q = \frac{403 \cdot 650}{2^{18}} = 63$, $r = 25695 - 63 \cdot 403 = 306$

Der Vorteil dieser Methode liegt in den wenigen (≤ 2) Durchläufen der while-Schleife und einfachen Berechnung der Näherung mittels Rightshifts.

Eine weitere naheliegende Anwendung von Newton ist das ganzzahlige Wurzelziehen. Zur Berechnung der ganzzahligen Wurzel $\lfloor \sqrt{a} \rfloor$ verwendet man die Newton-Iteration

$$x_{n+1} = \frac{x_n}{2} + \frac{a}{2x_n}. \tag{7}$$

Algorithmus 1.12 *berechnet* $\lfloor \sqrt{N} \rfloor$ *für ganze, positive Zahlen*

NewtonSquareRoot {
$y = 2^{\lceil \frac{B(N)}{2} \rceil}$ — Initialisierung mit Startwert $> \lceil \sqrt{N} \rceil$
do — Newton-Iteration
$x = y, \qquad y = \lfloor \frac{x + \lfloor N/x \rfloor}{2} \rfloor$
while $(x > y)$
return x }

Beispiel: $N = 1993 \Rightarrow x = 2^{\lceil \frac{11}{2} \rceil = 6} = 64, \quad y = \frac{1}{2}(64 + \frac{1993}{64}) = 47$
$x = 47, \quad y = \frac{1}{2}(47 + \frac{1993}{47}) = 44 \Rightarrow x = 44, y = 44, x \not> y \Rightarrow \sqrt{1993} = 44$

Die Laufzeit von (1.12) beträgt $O(\ln \ln N)$.

1.3 Restklassenring mod$p$

Die Operationen in einem Restklassenring mod$p = \mathbb{Z}_p$ lauten für $x, y \in \mathbb{Z}_p$:

- Addition : $(x + y) \bmod p = \left\{ \begin{array}{cc} x + y - p & x + y \geq p \\ x + y & x + y \leq p \end{array} \right\}$
- Subtraktion : $(x - y) \bmod p = \left\{ \begin{array}{cc} x - y + p & x < y \\ x - y & x \geq y \end{array} \right\}$
- Multiplikation : $(xy) \bmod p$

Für die Multiplikation existieren verschiedene Ansätze. Man kann zuerst das Produkt xy bilden und danach die modulo-Reduktion durchführen. Die Reduktion kann mittels eines Divisionsalgorithmus bzw. einer schnelleren Methode, z.B. der Barrett-Methode, erfolgen. Ein anderer Ansatz ist die modulo - Reduktion während der Multiplikation durchzuführen (z.B. Binäre mul-mod). Jedes Teilresultat wird um mod$p$ reduziert .

1.3.1 Binäre mul-mod

Berechnung von $xy \bmod N$ für $0 \leq x, y < N$ und $x_0, x_1, \ldots, x_{D-1}$ die Binärdarstellung von x, $x_{D-1} > 0$ höchstes Bit
Lösung: „bitweises“ Multiplizieren und Shiften
$s = x_{D-1}y, \quad s = 2s + x_{D-2}y, \quad s = 2s + x_{D-3}y, \ldots \Rightarrow s = xy$
In jedem Zwischenschritt wird $s \bmod N$ gebildet, somit gilt $s < N$.

Algorithmus 1.13 (Binäre mul-mod)

BinaryMulMod {
  $s = 0$ — Initialisierung
  for $(D - 1 \geq j \geq 0)$ — beginnend mit dem höchsten Bit
    $s = 2s$
    if $(s \geq N)$ $s = s - N$ — mod N
    if $(x_j == 1)$ $s = s + y$ — $x_j == 0 \Rightarrow s = s$
    if $(s \geq N)$ $s = s - N$ — mod N
  return s }

Beispiel: $x = 11, y = 31 \Rightarrow 11 \cdot 31 \equiv 46 \bmod 59$

x_j	1	0	1	1
$s = 2s$	0	$31 \cdot 2 = 62 \equiv 3$	$3 \cdot 2 = 6$	$37 \cdot 2 = 74 \equiv 15$
$x_j == 1, s+ = y$	31	–	$6 + 31 = 37$	$15 + 31 = 46$

1.3.2 Modulare Barrett Reduktion

Die Barrett Reduktion berechnet $x \bmod m$ mit x, m positive, ganzzahlige Werte, wobei die Länge von $x < 2 \cdot$ Länge von m in $B(>3)$-Basisdarstellung d.h. $x = (x_{2k-1}, x_{2k-2}, \ldots, x_1, x_0)$ und $m = (m_{k-1}, m_{k-2}, \ldots, m_1, m_0)$ mit $m_{k-1} \neq 0$. Es wird eine vorberechnete Konstante $\mu = \lfloor \frac{B^{2k}}{m} \rfloor$ benötigt.

Algorithmus 1.14 *x, m positive, ganzzahlige Werte in B-Basisdarstellung, k Länge von m bzgl. $B(>3)$ und $\mu = \lfloor \frac{B^{2k}}{m} \rfloor$*

BarrettReduction(x, m) {
  Näherung für Quotienten Q
  $q_1 = \lfloor \frac{x}{B^{k-1}} \rfloor$, $q_2 = q_1 \cdot \mu$, $q_3 = \lfloor \frac{q_2}{B^{k+1}} \rfloor$
  Näherung für Rest $R = x \bmod m$
  $r_1 = x \bmod B^{k+1}$, $r_2 = q_3 m \bmod B^{k+1}$, $r = r_1 - r_2$
  Korrektur des Resultats r
  if $(r < 0)$ $\quad r = r + B^{k+1}$
  while $(r \geq m)$ $\quad r = r - m$
  return (r) }

Beispiel: $x = 3955117, m = 2556 \Rightarrow k = 12$, $\mu = \frac{2^{24}}{2556} = 6563$

Quotienten:	$q_1 = \frac{x}{2^{11}} = 1931$, $q_3 = \frac{1931 \cdot 6563}{2^{13}} = 1547$
Reste :	$r_1 = x \bmod 2^{13} = 6573, \quad r_2 = q_3 \cdot 2556 \bmod 2^{13} = 5588$
Resultat:	$r_1 - r_2 = 6573 - 5588 = 985 \equiv 3955117 \bmod 2556$

Funktionsweise von 1.14:
Die Barrett-Reduktion beruht auf der Darstellung von $x = Qm + R$.

- Nährung für den Quotienten Q:
  Der Quotient lautet $Q = \lfloor \frac{x}{m} \rfloor = \lfloor \frac{x}{B^{k-1}} \cdot \frac{B^{2k}}{m} \cdot \frac{1}{B^{k+1}} \rfloor$, q_3 ist eine Näherung für den Quotienten und es gilt $Q - 2 \leq q_3 \leq Q$.

- Nährung für den Rest R:
  Für die Differenz der Reste $r_1 - r_2$ gilt $-B^{k+1} < r_1 - r_2 < B^{k+1}$ und die Kongruenz $r_1 - r_2 \equiv (Q - q_3) \cdot m + R \bmod B^{k+1}$
  $\Rightarrow 0 \leq (Q - q_3) \cdot m + R < 3m < B^{k+1}$.

- Korrektur des Resultats r:
  Sollte $r_1 - r_2 \geq 0 \Rightarrow r_1 - r_2 = (Q - q_3)m + R$, falls $r_1 - r_2 < 0 \Rightarrow r_1 - r_2 + B^{k+1} = (Q - q_3)m + R$. Die Anzahl der Durchläufe der while-Schleife beträgt maximal 2, da $0 \leq r < 3m$.

Bei der Barrettreduktion sind die Divisionen äquivalent zu Rechtsshifts bzgl. der Basis B und $\bmod B^{k+1}$ enspricht den niedrigwertigen $k+1$ Stellen d.h. die Multiplikation $\bmod B^{k+1}$ entspricht einer reduzierten Multiplikation von $k+1 \times k+1$ Stellen.

1.4 Moduli spezieller Form

Bei dem modulo N Berechnung mit speziellen Aufbau

$$N = 2^q + c$$

mit $|c|$ „klein“, existieren effiziente mod N Algorithmen als das klassische mod Schema. Ein Anwendungsbereich dieser Verbesserungen liegt bei den Mersenne Primzahlen $2^q - 1$ und Fermat Zahlen $2^{2^n} + 1$.

Satz 1.15 *Für $N = 2^q + c$, c „klein“, q positive ganze Zahl gilt*

$$x \equiv (x \bmod 2^q) - c\lfloor \frac{x}{2^q} \rfloor (\mathrm{mod} N). \tag{8}$$

Beweis:
$a = \lfloor \frac{x}{2^q} \rfloor$ werden die q niedrigsten Bits von x gelöscht.
$b = x \bmod 2^q$ sind die q niedrigsten Bits von x.
$\Rightarrow x = a2^q + b, \qquad 2^q \equiv -c \bmod N \Rightarrow x \equiv -ac + b \bmod N$ und $x > -ac + b$

Lemma 1.16 *Folgerung von 1.15 für Multiplikation mit $2^k \bmod N$:*

- *Mersenne Primzahl : $2^q - 1$ d.h. $c = -1$*
  Multiplikation mit $2^k \bmod N$ ist gleich einem links-zirkulären Shift um k Bits bzw. falls $k < 0$ einem rechts-zirkulären Shift um k Bits.
  Beispiel: $(2^2 \cdot 5) \bmod 2^3 - 1 = (101)_2 << 2 = (110)_2 = 6$
- *Fermat Zahlen : $2^{2^n} + 1$ d.h. $c = 1, q = 2^n$*
  Multiplikation mit $2^k \bmod N, k > 0$ ist gleich einem links-zirkulären Shift, wobei die rechtsanzufügenden Bits vom Zwischenresultat subtrahiert werden, und dem Vorzeichen $(-1)^{\lfloor \frac{k}{q} \rfloor}$.
  Beispiel: $(2^5 \cdot 11) \bmod 2^4 + 1 \Rightarrow$
  $(1011)_2 << 5 = (0110)_2 - (1)_2 = (0101)_2 = 5 \Rightarrow -5 \equiv 12 \bmod 17$

Algorithmus 1.17 *berechnet $x \bmod N$ für $N = 2^q + c, x > 0$ mit $B(|c|) < q$. Die Methode ist effizienter für kleine $|c|$.*

```
FastModSpecialForm{
  while (B(x) > q)                    rekursiver Aufruf von Satz (1.15)
    y = x >> q                                  Rightshift = ⌊x/2^q⌋
    x = x − (y << q)                  oder x&(2^q − 1) = x mod 2^q
    x = x − cy
  if (x ≥ N)        x = x − N
  if (sgn(x) < 0)   x = x + N                   −1, 0, 1 für x <, =, > 0
  return x }
```

Beispiel: $x = 15630$, $N = 2^7 + 3 = 131$

$B(x) = 14 > 7$		
$y = \frac{15630}{2^7} = 122$	$x = 15630 - 122 \cdot 2^7 = 14$	$x = 14 - 3 \cdot 122 = -352$
$B(-352) = 9 > 7$		
$y = \frac{-352}{2^7} = -3$	$x = -352 + 3 \cdot 2^7 = 32$	$x = 32 + 3 \cdot 3 = 41$

$\Rightarrow 15630 \equiv 41 \bmod 131$

Für den Satz 1.15 gelten folgende Abschätzungen

$$x \bmod 2^q < 2^q \; und \; -c\lfloor\frac{x}{2^q}\rfloor \sim 2^{B(x)-q}.$$

d.h. in jedem Schritt der while-Schleife wird x um q Bits reduziert $\Rightarrow$ Durchläufe der While-Schleife beträgt $O(\lceil\frac{B(x)}{q}\rceil)$.
Die Mersenne und Fermatzahlen bilden aufgrund ihrer speziellen Bauart in einigen anderen Anwendungsgebieten Verwendung, z.B. Kryptographie, die Elliptische Kurven Methode.

Eine weitere Spezialform ist die Proth Form

$$N = k \cdot 2^q + c.$$

Um $x \bmod N$ zu berechnen, kann man einen Schätzer $\tilde{d}$ für $\lfloor\frac{x}{N}\rfloor$ verwenden $\Rightarrow x \bmod N = x - dN, \qquad \tilde{d} = d - \Delta d$

Der Schätzer $\tilde{d}$ ist $\tilde{d} = \lfloor\frac{x}{k \cdot 2^q}\rfloor$ und es gilt

$$\lfloor\frac{x}{k2^q + c}\rfloor = \lfloor\frac{x}{k2^q}\rfloor + bN, \qquad b = \left\{ \begin{array}{ll} b > 0 & c < 0 : \lfloor\frac{x}{k2^q+c}\rfloor \geq \lfloor\frac{x}{k2^q}\rfloor \\ b < 0 & c > 0 : \lfloor\frac{x}{k2^q+c}\rfloor \leq \lfloor\frac{x}{k2^q}\rfloor \end{array} \right\}$$

Die Größenordnung von b ist mit

$$\lfloor\frac{x}{k2^q}\rfloor - \lfloor\frac{x}{k2^q + c}\rfloor \leq \frac{x}{k2^q} - \frac{x}{k2^q + c} + 2 = \frac{xc}{k2^q(k2^q + c)} + 2$$

$$k2^q(k2^q + c) < N^2 \qquad \frac{x}{k2^q(k2^q + c)} \sim 1 \Rightarrow tc + 2$$

$O(c)$ beschränkt.

Algorithmus 1.18 *berechnet* $x \bmod N$ *mit* $N = k2^q + c, 0 < x < N^2, q > 0, B(|c|) < q$ *und* c *beliebig*

```
FastModProthForm {
    d = ⌊(x>>q)/k⌋                          Schätzer d̃
    t = Nd = ((kd) << q) + cd
    if (c < 0)                              b > 0 oder
        while (t ≤ x)
            d = d + 1
            t = Nd
        d = d − 1                           Korrektur des Resultats
        t = Nd
    else                                    b < 0
        while (t > x)
            d = d − 1
            t = Nd
    return x − t }
```

Beispiel: $x = 11627, N = 3 \cdot 2^6 - 3 = 189 \Rightarrow d = \lceil \frac{11627}{3 \cdot 2^6} \rceil = 60, t = 11340$
$c < 0 \wedge t \leq x \Rightarrow d = 61, t = 11529, x - t = 98$

Die Anzahl der Durchläufe der while-Schleife ist mit $O(b) = O(c)$ beschränkt. Der Algorithmus 1.18 ist effizienter für „kleine“ $k, |c|$.

2 Potenzen

Potenzen gehören zu den elementaren arithmetischen Operationen in einem Körper. Die klassische Schulmethode für x^y ist das wiederholte Multiplizieren mit x und Verringerung des Exponenten y um 1, solange $y > 0$.
Es existieren effizientere Algorithmen für x^y bzw. $x^y \bmod N$.

Algorithmus 2.1 *klassische Potenz*

```
Pow {
    r = 1
    while (y > 0)
        r = r · x
        y = y − 1
    return x }
```

Der Aufwand der klassichen Potenz beträgt $C \sim y \cdot M$.
Die Komplexität C eines Algorithmus setzt sich aus der Anzahl von S (Komplexität des Quadrates x^2 bzw. $x^2 \bmod N$) und der Anzahl der Multiplikationen M (Komplexität der Multiplikation xy bzw. $xy \bmod N$) zusammen. Für die klassischen Methoden gilt das Verhältnis $S/M = 1/2$, d.h. der Aufwand für die Berechnung eines Quadrat ist nur die Hälfte des Aufwandes einer Multiplikation.

Gesucht : Algorithmus für $x^y \bmod N$
Im folgenden werden Algorithmen für x^y angegeben, die Erweiterung auf $x^y \bmod N$ erfolgt mittels

- $(x^y) \bmod N$, Berechnung von x^y und Resultat $\bmod N$
- während des Algorithmus wird jedes Teilresultat $\bmod N$ reduziert
  Operationen: $\cdot, +, - \rightarrow \cdot, +, - \bmod N$

Als Beispiel werden die binären Methoden auf $x^y \bmod N$ erweitert. Die Reduktion erfolgt mittels der Barrett-Methode (1.14). Es existieren spezielle Algorithmen für $x^y \bmod N$, z.B. die Montgomery-Methode, welche in $\mathbb{Z}_N$ operiert.

2.1 Binäre Methoden

Es existieren 2 Formen von nichtrekursiven binären Methoden, „Links-Rechts“ und „Rechts-Links“ Formen. Die Binärdarstellung von y ist $(y_0, \ldots, y_{D-1})$ mit dem höchsten Bit y_{D-1}. Es gilt:

$$x^y = x^{y_{D-1},\ldots,y_0} = x^{2^{D-1}y_{D-1}} \cdot x^{2^{D-2}y_{D-2}} \cdots x^{2y_1} \cdot x^{y_0} = \prod_{j=0}^{D-1} x^{2^j y_j}$$

2.1.1 Links-Rechts-Form

Der Algorithmus (2.2) berechnet die Potenz x^y indem die Bits des Exponenten y von links nach rechts durchlaufen werden. Dieser Algorithmus (2.2) ist bekannt als „Square and Multiply“.

$$x^y = \prod_{j=0}^{D-1} x^{2^j y_j} = ((\dots((x^{y_{D-1}})^2 \cdot x^{y_{D-2}})^2 \cdots x^{y_1})^2 \cdot x^{y_0} \tag{9}$$

Algorithmus 2.2 (Links-Rechts-Form) *berechnet* x^y

PowLeftRight {
  $z = x$ Initialisierung
  Schleife über die Bits von y, beginnend mit den zweithöchsten Bit y_{D-2}
  for $(D-2 \geq j \geq 0)$
    $z = z^2$ Quadrieren in Horner-Schema
    if $(y_j == 1)$ $z = zx$ Multiplikation mit x^{y_j}
  return z }

Die Anzahl der Quadrate ($z = z^2$) ist $D-1$ und die Anzahl der Multiplikationen ($z = zx$) ist die Anzahl der 1er Bits des Exponenten y, $\#1(y) - 1$.

Beispiel: $31^{11}, 11 = 1011_2 \Rightarrow z = 31$

	z^2	$z \cdot 31$
$x_2 = 0$	$31^2 = 961$	
$x_1 = 1$	$961^2 = 923521$	28629151
$x_0 = 1$	$28629151^2 = 819628286980801$	25408476896404831

$\Rightarrow 31^{11} = 25408476896404831$

2.1.2 Rechts-Links-Form

Der Algorithmus (2.3) berechnet die Potenz x^y indem die Bits des Exponenten y von rechts nach links durchlaufen und für die Bits y_j mit $(x^{2^j})^{y_j}$ multipliziert werden.

$$x^y = \prod_{j=0}^{D-1} x^{2^j y_j} = x^{\sum_{j=0}^{D-1} 2^j y_j} = x^{y_0}(x^2)^{y_1}(x^4)^{y_2} \cdots (x^{2^{D-1}})^{y_{D-1}} \tag{10}$$

Algorithmus 2.3 (Rechts-Links-Form) *berechnet* x^y

PowRightLeft {
  $z = x \quad a = 1$ Initialisierung, a = Ergebnis
  Schleife über die Bits von y, beginnend mit den niedrigsten Bit y_0
  for $(0 \leq j < D-1)$
    if $(y_j == 1)$ $a = az$ Multiplikation mit $(x^{2^j})^{y_j=1}$
    $z = z^2$ Quadrieren
  return az } Multiplikation mit $x^{2^{D-1}}$

Die Anzahl der Quadrate ($z = z^2$) ist $D-1$ und die Anzahl der Multiplikationen ($a = az$) ist $\#1(y)-1$, da bei $a = z \cdot 1$ nur eine Zuweisung stattfindet.

Beispiel: $31^{11}, 11 = 1011_2 \Rightarrow z = 31, a = 1$

	a	z^2
$x_0 = 1$	31	$961 = 31^2$
$x_1 = 1$	29791	$923521 = 31^4$
$x_2 = 0$	–	$852891037441 = 31^8$

$\Rightarrow 31^{11} = a \cdot z = 29791 \cdot 852891037441 = 25408476896404831$

2.1.3 Komplexität

Ein Vorteil der „Links-Rechts-Form“ gebenüber der „Rechts-Links-Form“ ist die Operation $z = zx$ mit einem fixen Multiplikant. Dieser Vorteil wirkt sich besonders für x mit „vielen“ 1ern aus.
Beispiel: Primalitätstest „kleiner Fermat“ für einen Zeugen a

$$a^{p-1} \equiv 1 \bmod p \quad a_{\text{„klein“}}$$

d.h. für das Resultat der sukzessive Multiplikation gilt $< aN$, die Reduktion kann mittels weniger Subtraktionen erfolgen. Für $a = 2$ kann die Multiplikation durch eine Addition ersetzt werden. Die beiden Formen besitzen dieselbe Anzahl von Quadraten $D-1$ und Multiplikationen $\#1(y)-1$ für fixe x und y. Die Komplexität C der binären Methoden ist

$$C \sim (\lg y) \cdot S + \#1(y) \cdot M \tag{11}$$

Für die Anzahl der 1er im Exponenten y ist der „Best Case“ 1 und der „Worst Case“ n. Gesucht: „Average-Case“ = Erwartungswert von $\#1(y)$

$$\mathbb{E}(\#1(y)) = \sum_{k=0}^{n=\lg y} \mathbb{E}(\text{ Stelle } y_i = 1) = \sum_{k=0}^{\lg y} \mathbb{P}(y_i == 1) = \sum_{k=0}^{\lg y} \frac{1}{2} = \frac{\lg y}{2}$$

d.h. durchschnittlich die Hälfte der Bits des Exponenten = 1

$$\Rightarrow C \sim (\lg y) \cdot S + \frac{\lg y}{2} \cdot M \tag{12}$$

2.1.4 Binäre Methoden für $x^y \bmod N$

Die Erweiterung der binären Methoden auf $\bmod N$ erfolgt durch die Ersetzung der Operationen durch Operationen $\bmod N$, d.h. z^2, zx werden mittels Barrett-Methode (Algorithmus 1.14) reduziert.

Algorithmus 2.4 (Links-Rechts-Form) *berechnet* $x^y \bmod N$

PowLeftRightMod {
$z = x$ Initialisierung
for $(D-2 \geq j \geq 0)$
$z =$BarrettReduction(z^2, N) Quadrieren und Barrett-Reduktion
Multiplikation mit x^{y_j} und Barrett-Reduktion
if $(y_j == 1)$ $z =$BarrettReduction(zx, N)
return z }

Algorithmus 2.5 (Rechts-Links-Form) *berechnet* $x^y \bmod N$

PowRightLeftMod {
$z = x \quad a = 1$ Initialisierung, a = Ergebnis
for $(0 \leq j < D-1)$
Multiplikation mit $(x^{2^j})^{y_j=1}$ und Barrettreduktion
if $(y_j == 1)$ $a =$BarrettReduction(az, N)
$z =$BarrettReduction(z^2, N)
return BarrettReduction(az, N) } Multiplikation mit $x^{2^{D-1}}$

Die Komplexität setzt sich aus der Komplexität der binären Methoden und der Barrett-Reduktion zusammen,

$$C \sim (\lg y) \cdot S + \#1(y) \cdot M + (\lg y + \#1(y)) \cdot C_{Barrett} \tag{13}$$

mit $C_{Barrett}$, die Komplexität der Barrett-Reduktion. Die durchschnittliche Komplexitätsabschätzung beträgt somit

$$\Rightarrow C \sim (\lg y) \cdot S + \frac{\lg y}{2} \cdot M + \frac{3}{2} \lg y \cdot C_{Barrett}. \tag{14}$$

2.2 B - äre Methoden

Die B-äre Methoden werden in der Literatur oft als „windowing“ bezeichnet. Die Grundidee ist das Erweitern der Basis von 2 auf B.

2.2.1 B - äre Methode mit fixer Basis B

Beispiel: Gegenüberstellung der binären Methode (2.2) und Darstellung der Basis $B = 4$. Gegeben sind bei $B = 4$ die Potenzen $1, x^1, x^2, x^3$.

	x^{79}	x^{127}
Exponent	$79 = 1001111_2 = 1033_4$	$127 = 1111111_2 = 1333_4$
$B = 2$	$(((x^{2^3}x)^2x)^2x)^2x$	$(((((x^2x)^2x)^2x)^2)x^2x)^2x)$
Aufwand $B = 2$	$6S + 4M$	$6S + 6M$
$B = 4$	$(x^{4^2}x^3)^4x^3$	$((x^4x^3)^4x^3)^4x^3$
Aufwand $B = 4$	$6S + 2M$	$6S + 3M$

Der Aufwand von $B = 4$ erhöht sich um $2M$ für die Vorberechnung von x^2, x^3. Eine Verringerung der Multiplikationen (inklusive der Vorberechnung der Multiplikationen) gegenüber der Binärmethode ergibt sich für Potenzen mit Exponenten > 79.

Vorteil: Für Ziffern der B-Basisdarstellung $y_{D-1}, \dots y_1, y_0$ mit $y_i > 2$ werden die y_i Multiplikationen $\times x$ der Binärmethode durch eine Multiplikation $\times x^{y_i}$ ersetzt $\Rightarrow$ Verringerung der Multiplikationen.

Algorithmus 2.6 *berechnet x^y mit $y > 0$. Die B-Basisdarstellung $B = 2^b$ ist $y_{D-1}, \dots y_1, y_0$ mit $0 \le y_i < B$. Die Potenzen x^d sind vorberechnet.*

```
PowWindow {
    z = 1                                          Initialisierung
    for (D − 1 ≥ i > 0) {                          Schleife über die Stellen
        if  (y_i > 0)      z = z x^{y_i}           Multiplikation
        z = z^{2^b} }                              Quadrate
    if  (y_0 > 0)      z = z x^{y_0}               Multiplikation
    return z }
```

Die Komplexität der B-ären Methode ist $C = a \cdot S + b \cdot M$. Für die Konstante a gilt, dass pro Stelle b Quadrate gebildet werden, d.h. $a = Db \sim \lg y$.
b entspricht dem Erwartungswert der Multiplikationen $= \mathbb{E}(\#M)$.

$$\mathbb{E}(\#M) = \mathbb{E}(Stellen \neq 0) = \sum_{i=0}^{n=\frac{\lg y}{b}} \mathbb{E}(y_i > 0) =$$

$$= \sum_{i=0}^{n} (1 - \mathbb{P}(y_i = 0) = \sum_{i=0}^{n} 1 - 2^{-b} = \frac{\lg y}{b}(1 - 2^{-b})$$

Die Komplexität von der B-ären Methode lautet

$$C \sim (\lg y)S + \frac{\lg y}{b}(1 - 2^{-b})M \tag{15}$$

ohne die Vorberechnung der x^d. Für die Basis $B = 2 = 2^1$ gilt $C \sim (\lg y)S + \frac{\lg y}{2}M$, das „windowing“ wird zur binären Methode.

2.2.2 Gleitende B-äre Methode

Die gleitende B-äre Methode, „sliding windowing“ ändert die Basis während der Berechnung.

Der Exponent y ist in Binärdarstellung $y_{D-1}, \ldots, y_1, y_0$ gegeben. Die Bits werden absteigend durchlaufen, bei $y_i = 0$ wird quadriert. Bei $y_i = 1$ wird ein Block der Länge l, $[y_i, y_{i-1}, \ldots, y_{i-l+1}]$, mit $y_{i-l+1} = 1$ gebildet und das Ergebnis mit $x^{y_i, y_{i-1}, \ldots, y_{i-l+1}}$ multipliziert. Die Blocklänge l ist $\leq k$, der maximalen Blocklänge. Als Vorberechnung müssen die Potenzen $x^d < x^{2^k}$ gebildet werden.

Der Exponent $y = y_{D-1}, \ldots y_1, y_0$ mit $0 \leq y_i < B = 2^b$ wird stellenweise durchlaufen.

- $y_i == 0 \Rightarrow z = z^{2^b}$
- $y_i > 0$ wird der Block über maximal k Stellen gebildet

Algorithmus 2.7 *berechnet x^y. $y = (y_{D-1}, \ldots y_1, y_0)$ in Binärdarstellung gegeben und k ist die maximale Blocklänge.*

PowSlidingWindow {
  $i = D - 1 \qquad z = 1$ — Initialisierung
  while $(i \geq 0)$ — Schleife
    if $(y_i == 0)$
      $z = z^2, \quad i = i - 1$ — Quadrieren
    else
      $j = i - k + 1$ — max. Länge
      Bitblock $y_i, \ldots, y_j$ beginnt und endet mit 0
      while $(y_j == 0) \quad \{ \quad ++j \quad \}$ — solange Endbit = 1
      $d = (y_i, \ldots y_j)$
      $z = z^{2^{i-j+1}}$ — Potenz
      $z = z(x^d)$ — Multiplikation
      $i = j - 1$ — nächste Stelle
  return z }

Beispiel: $3^{852} \bmod 7$ mit der maximalen Blocklänge $k = 3$
Vorberechnung der $x^d = 3^i < 3^8$

i	1	2	3	4	5	6	7
$3^i \bmod 7$	3	2	6	4	5	1	3

Durchlauf des Algorithmus für $852_2 = 1101010100 \Rightarrow 3^{852} \equiv 1 \bmod 7$

y_i	d	l	z^{2^l}	$z \cdot 3^d$	z
11	3	2	$(3^2)^2 \equiv 4 \bmod 7$	$4 \cdot 3^3 \equiv 3 \bmod 7$	3
0	–	–	$3^2 \bmod 7$		2
101	5	3	$(2^2)^3 \equiv 1 \bmod 7$	$1 \cdot 3^5 \equiv 5 \bmod 7$	5
0	–	–	$5^2 \bmod 7$		4
1	1	1	$4^2 \equiv 2 \bmod 7$	$2 \cdot 3 \equiv 6 \bmod 7$	6
0	–	–	$6^2 \bmod 7$		1
0	–	–	$1^2 \bmod 7$		1

Satz 2.8 *Die Anzahl der Durchläufe t der while-Schleife von 2.7 ist mit $\frac{\lg y}{k} \leq t \leq \lg y$ beschränkt.*

Beweis: Im besten Fall wird in jedem Durchlauf ein Block mit maximaler Länge k gefunden. Der Worst-Case tritt bei $y = 2^{D-1}$ ein, die Schrittweite beträgt $1 \Rightarrow \lg y$ Quadrate.

2.2.3 Vorberechnung:

Vollständige Berechnung aller $x^e \, \forall e : 0 \leq e < B$

Methode		$C \sim$
1	$x^e = x^{e-1} \cdot x$	BM
2	$x^e = \begin{cases} (x^{\frac{e}{2}})^2 & \text{e gerade} \\ x^{\frac{e-1}{2}} \cdot x^{\frac{e+1}{2}} & \text{e ungerade} \end{cases}$	$\frac{B}{2}S + \frac{B}{2}M$

Methode 3: Vorberechnung der x^k nur für ungerade k:
$\forall e : 0 \leq e < B$ gilt $e = 2^c k$, k ungerade

Die Multiplikation mit x^e wird ersetzt durch $(x^k)^{2^c}$, k ungerade. Der Aufwand einer Multiplikation des Algorithmus beträgt somit eine Multiplikation und c Quadrate.

Beispiel: für $B = 16$

								c	Anzahl
1	3	5	7	9	11	13	15	$c = 0$	$\frac{B}{2}$
2	6	10	14					$c = 1$	$\frac{B}{4}$
4	12							$c = 2$	$\frac{B}{8}$
8								$c = 3$	$\frac{B}{16}$

$$\mathbb{P}(e == 2^c k) = \frac{\#e = 2^c k < B}{\#e < B} = \frac{\frac{B}{2^{c+1}}}{B} = \frac{1}{2^{c+1}}$$

Gesucht: durchschnittlicher Wert für c = Erwartungswert von $c = \mathbb{E}(c) =$

$$\sum_{i=1}^{\lg e} i \cdot \mathbb{P}(e == 2^i k) = \sum_{i=1}^{\lg e} \frac{i}{2^i} = \frac{1}{2} \sum_{i=1}^{\lg e} \frac{i}{2^i}, 1 \leq \sum_{i=1}^{\lg e} \frac{i}{2^i} \leq 2 \Rightarrow \mathbb{E} \approx \frac{1}{2} 2 = 1$$

d.h. für jede Multiplikation im Algorithmus wird eine zusätzliches Quadrat gebildet. Die Komplexität des Algorithmus ohne Vorberechnung beträgt $C(A) = x_S S + x_M M$ und mit Vorberechung

$$C(V) \sim \frac{B}{2} M \Rightarrow C \sim (x_S + x_M) S + x_M M + \frac{B}{2} M$$

2.3 Montgomery Methode für $x^y \bmod N$

Die Montgomery-Methode führt eine Exponentation in Restklassenring $\mathbb{Z}_N$ aus. Die Idee ist die Transformation des Problem in ein Residuum, wo eine schnellere Lösung möglich ist.

Definition 2.9 (Montgomery Reduktion) *Für R, N teilerfremde, positive, ganzzahlige Zahlen und $x \in \mathbb{Z}_N = (0 \leq x < N)$ ist das (R, N)-Residuum von x, $\bar{x} = xR \bmod N$.*

Die Abbildung $x \to \bar{x}$ ist verträglich mit der Addition für alle $x, y \in \mathbb{Z}_N$: $\overline{x+y} = (x+y)R \equiv xR + yR \bmod N = \bar{x} + \bar{y}$, jedoch unverträglich mit der Multiplikation $\overline{xy} = (xy)R \bmod N = \bar{x} \cdot \bar{y}R$

Definition 2.10 (Montgomery Produkt) *$\star$ zweier Zahlen a, b lautet $a \star b = abR^{-1} \bmod N$.*

$(\mathbb{Z}_N, +, \star)$ ist ein kommutativer Ring:

- $(\mathbb{Z}_N, +)$ ist kommutative Gruppe
- $(\mathbb{Z}_N, \star)$ assoziativ und $\exists$ Einselement bzgl. $\star$
  - Assoziativität $abcR^{-2} \equiv (abR^{-1}) \cdot (cR^{-1}) \equiv (a \star b) \star c \equiv (aR^{-1}) \cdot (bcR^{-1}) \equiv a \star (b \star c) \bmod N$
  - $\exists$ Einselement R $u \star R \equiv uR^{-1}R \equiv u \bmod N$
- Distributivgesetz $(a+b) \star c \equiv (a+b)cR^{-1} \equiv acR^{-1} + bcR^{-1} \equiv a \star c + b \star c$

Satz 2.11 *Es gilt $(\mathbb{Z}_N, +, \cdot) \simeq (\mathbb{Z}_N, +, \star)$, der Isomorphismus ist gegeben durch $x \to \bar{x} = xR \bmod N$ und der inverse durch $x \to \bar{x}^{-1} = xR^{-1} \bmod N$.*

Beweis:

- $x \to \bar{x}$
  $\overline{x+y} \equiv (x+y)R \equiv xR + yR \equiv \bar{x} + \bar{y}$
  $\overline{xy} \equiv (xy)R \equiv (xR \cdot yR)R^{-1} \equiv \bar{x} \star \bar{y}$
- $x \to \bar{x}^{-1}$
  $\overline{\bar{x} + \bar{y}}^{-1} \equiv (\bar{x} + \bar{y})R^{-1} \equiv (xR + yR)R^{-1} \equiv x + y$
  $\overline{\bar{x} \star \bar{y}}^{-1} \equiv (\bar{x} \star \bar{y})R^{-1} \equiv ((xR \cdot yR)R^{-1})R^{-1} \equiv xyR^2R^{-2} \equiv xy$

Die Multiplikation von $\mathbb{Z}_N$ mit R oder R^{-1} ergibt ein (R, N)-Residuum, die Idee ist die Potenz in diesem Residuum zu berechnen.

Verfahrensweise für u^k:

1. Montgomery-Reduktion von u
2. Bildung der Potenz mit $\bar{u}^{*k}$ mittels Montgomery-Multiplikationen
3. Rücktransformation von $\bar{u}^{*k}$

$$\begin{array}{ccccc} & & u^k & & \\ u \in \mathbb{Z}_N & & \longrightarrow & & u^k \\ uR \quad \downarrow & & & & \uparrow \quad \bar{u}R^{-1} \\ \bar{u} & & \longrightarrow & & \bar{u}^{*k} \\ & \bar{u}^{*k} = \underbrace{\bar{u} \star \ \bar{u} \star \ldots \star \bar{u}}_{k} & & & \end{array}$$

Das Montgomery Produkt $a \star b = abR^{-1} \bmod N$ wird im folgenden in der Funktion M, $M(a,b) = a \star b$, berechnet und $M(\bar{a}, 1) = \bar{a}R^{-1}$.
Gesucht: schnelle Implementation der M Funktion, siehe 2.3.1

Algorithmus 2.12 (Montgomery Potenz) *liefert x^y mod N für N, R wie in 2.9 und $(y_0, \ldots y_{D-1})$ die Bits von y.*

```
Montgomery(x,y,N) {
    x̄ = xR mod N,   p̄ = R mod N                    Montgomery-Reduktion
    for (D − 1 ≥ j ≥ 0)                  sukzessive Montgomery-Multiplikation
        p̄ = M(p̄, p̄)
        if (y_j == 1)      p̄ = M(p̄, x̄)
    return M(p̄, 1) }                          Rücktransformation von p̄
```

Beispiel: $x = 314, y = 9 = 1001_2, N = 797, R = 2^{10}$

		$\bar{x} = 314 \cdot 2^{10} \bmod 797 = 345$	$\bar{p} = 2^{10} \bmod 797 =^2 27$
3	1	$\bar{p} = M(227, 227) = 227$	$\bar{p} = M(227, 345) = 345$
2	0	$\bar{p} = M(345, 345) = 735$	
1	0	$\bar{p} = M(735, 735) = 38$	
0	1	$\bar{p} = M(38, 38) = 210$	$\bar{p} = M(210, 345) = 586$

$\Rightarrow 314^9 = M(586, 1) = 136$ Die Effizienz der Montgomery Potenz ist von der M-Funktion abhängig.

Satz 2.13 *Sei N eine ungerade, positive, ganze Zahl. Dann wird die modulare Potenz $x^y \bmod N$ mit 2 Montgomery-Reduktionen (x, p) und $2 \cdot \log_2 y$ Montgomery-Multiplikationen berechnet $\Rightarrow O(\log_2 y)$ Multiplikationen.*

2.3.1 Implementation des Montgomery Produktes

Satz 2.14 (Montgomery) *$x = a \cdot b$, N, R wie in 2.9, $N' = (-N^{-1}) \bmod R$*

$$z = x + N((xN') \bmod R) \tag{16}$$

ist ohne Rest durch R teilbar und es gilt

$$\frac{z}{R} \equiv xR^{-1} \bmod N \tag{17}$$

Für $0 \leq x < RN$ ist die Differenz, $y/R - ((xR^{-1}) \bmod N)$, 0 oder N
$\Rightarrow \frac{z}{R} \bmod N \equiv a \star b$.

Beweis:
(16) : $z = x + t \cdot N \equiv x + xN' \cdot N \equiv x - x \cdot N^{-1}N \equiv x - x \equiv 0 \bmod R$
(17) :$z = x + t \cdot N \equiv x \bmod N \Rightarrow \frac{x+t\cdot N}{R} \equiv xR^{-1} \bmod N$
Für $0 \leq x < RN$ gilt $\frac{y}{R} - (xR^{-1} \bmod N) = \frac{x+t\cdot N}{R} - \frac{x}{R} = \frac{tN}{R}$
$0 \leq t \leq R \Rightarrow \frac{t}{R} \cdot N = 0, N$

Algorithmus 2.15 (Montgomery Produkt) *berechnet $M(a, b)$, $0 \leq a, b < N$ mit N ungerade und $R > N$.*

```
M(c,d) {
    x = a · b
    z = y/R                                   Satz Montgomery (16)
    if (z ≥ N)        z = z − N
    return z }
```

Der Aufwand von $M(a, b)$ beträgt 3 Multiplikationen mit Größe $\leq N$ und einer Division ($\frac{z}{R}$). Eine Verbesserung des Aufwandes ist durch eine geschickte Wahl von R möglich.

Lemma 2.16 *$R = 2^s, N$ ungerade $\Rightarrow \frac{y}{R} = (x + N \cdot ((x \cdot N')\&(R-1))) >> s$*

Für $R = 2^s$ ist die Division durch R gleich einem Rightshift $>>$ um s Bits und mod R sind die niedrigsten s Bits d.h. & (bitweises Und) mit $R-1$. Für $0 \leq x = a \cdot b < RN$ ist xR^{-1} mod N mit 3 Multiplikationen zu berechnen (die Bitoperationen sind vernachlässigbar).

Es existiert eine Variation des Satzes 2.14 für Langzahlarithmetik zur Basis $B = 2^s$, N in B-Darstellung $N = \sum_{i=0}^{n-1} n_i B^i$ und $R = B^n$. Der Vorteil ist, dass die Reduktion mit $n' = -N^{-1} \bmod B$ statt modulo R operieren kann.

Satz 2.17 *$z, N = \sum_{i=0}^{n-1} n_i B^i$ in Basisdarstellung, $B = 2^n$, gegeben und $n' = -N^{-1} \bmod B$ dann gilt für z aus (2.14)*

$$z = z + \sum_{i=0}^{n-1} t_i N B^i \qquad t_i = z_i n' \bmod B \tag{18}$$

$\Rightarrow \frac{z}{R}$ ist die Montgomery-Reduktion von z.

Für $a \star b$ wird die Langzahlmultiplikation $z = a \cdot b$ und die Reduktion von z kombiniert,

$$z = a \cdot b = \left(\sum_{i=0}^{n-1} a_i B^i \right) b = \sum_{i=0}^{n-1} a_i B^i b.$$

Algorithmus 2.18 *Variablen laut 2.17*

M(a, b) {
  $z = 0$
  for $(0 \leq i \leq n-1)$ — Multiplikation und Reduktion
    $t_i = (z_i + a_i b) \cdot n' \bmod B$
    $z = z + (a_i b + t_i N) \cdot B^i$
  $z = \frac{z}{R} = z >> n$ — $R = B^n \Rightarrow$ Rightshift
  if $(z \geq N)$ $\quad z = z - N$
  return z }

Satz 2.19 *Algorithmus (2.18) berechnet das Montgomery-Produkt $z = a \star b$ mit $0 \leq a, b < N$ in $2n(n+1)$ Multiplikationen von Faktoren aus $[0, B)$.*

2.4 Fixe-Basis Methoden

Die folgenden Algorithmen nehmen die Basis x als fix. Der Parameter ist der veränderliche Exponent y.

2.4.1 B-äre Methode für fixe Basis x

Vorberechnung: Sämtliche vorkommende mögliche Potenzen bzgl. der Basis B bis zur maximalen Stellenanzahl D werden berechnet.

$$x^{iB^j} : i \in [1, B-1], j \in [0, D-1]$$

d.h. $(B-1)(D-1)$ berechnete Potenzen.

Algorithmus 2.20 *berechnet x^y*
mit $y_{D-1}, \ldots y_0$ die Stellen des Exponenten y bzgl. der Basis B

PowFixXWindow {
  $z = 1$ — Initialisierung
  for $(0 \leq i < D)$ — Schleife über die Stellen
    if $(y_i > 0)$ $\quad z = z x^{y_i B^i}$ — Multiplikation
  return z }

Die Komplexität von Algorithmus (2.20) (ohne Vorberechnung) beträgt

$$C \sim DM \sim \frac{\lg y}{\lg B} M \tag{19}$$

mit der Vorberechnung

$$C \sim DM \sim \frac{\lg y}{\lg B} M + (B-1)(D-1)M \tag{20}$$

Beispiel: Vorberechnung für die Basis 3

$y_i \backslash i$	0	1	2	3
1	3^1	3^4	3^{16}	3^{64}
2	3^2	3^8	3^{32}	3^{118}
3	3^3	3^{12}	3^{48}	3^{192}

Potenzen: $3^6 = 3^{12_4} = 3^{2,0}3^{1,1} = 3^2 \cdot 3^4 = 3^6$,
$3^{159} = 3^{2133_4} = 3^{3,0} \cdot 3^{3,1} \cdot 3^{1,2} \cdot 3^{2,3} = 3^3 \cdot 3^{12} \cdot 3^{16} \cdot 3^{128} = 3^{159}$

2.4.2 Fixe Basis Euklidmethode

Gegeben: fixe Basis x, Basis B und der Exponenten $y = \sum y_i B^i$
Vorberechnung: die Potenzen $p_i = x^{B^i}$ für $i < D$

Idee: Euklidreduktion der Koeffizienten $y_i \Leftrightarrow$ Multiplikation der p_i
Es gilt $y = \sum y_i B^i$ und $x^y = \prod p_i^{y_i}$. Wähle $y_m, y_n \in \{y_0, \ldots, y_{D-1}\}$ mit $y_m \geq y_i \forall i$ und $y_n \geq y_i \forall i \neq m$, wende den Euklid an $y_m = qy_n + r$. Setze $\bar{y}_m = r$ und $\bar{p}_n = p_m^q p_n$. Dann gilt

$$\bar{y} = \sum_{i \neq m} y_i B^i + \bar{y}_m B^m < y$$

und für die Potenzen p_i gilt

$$(\prod_{i \neq n,m} p_i^{y_i}) p_m^{\bar{y}_m} \bar{p}_n^{y_n} = (\prod_{i \neq n,m} p_i^{y_i}) p_m^r (p_m^q p_n)^{y_n} = (\prod_{i \neq n,m} p_i^{y_i}) p_m^{qy_n + r} p_n^{y_n} = x^y$$

Setze $y_m = \bar{y_m} = r$, $p_n = \bar{p_n}$ und wiederhole den Vorgang bis $y_m > 0$ und $y_i = 0 \ \forall i \neq m \Rightarrow x^y = p_m^{y_m}$.

Algorithmus 2.21 *berechnet x^y*
mit $y = \sum y[i]B^i$ und $p[i] = x^{B^i}$ für $i \in I = \{0, 1, \ldots, D-1\}$

```
PowFixXEuklid {
    m = IndexMax(−1)              liefert den Index des höchsten y_i
    n = IndexMax(m)               liefert den Index des zweithöchsten y_i
    while (y_n ≠ 0)               y_n = 0 ⇒ y_m > 0 ∧ y_i = 0 ∀i ≠ m
        q = ⌊y_m / y_n⌋, p_n = (p_m)^q p_n   y_m ≡ y_m mod y_n   Euklid
        m = IndexMax(−1)          liefert den Index des höchsten y_i
        n = IndexMax(m)           liefert den Index des zweithöchsten y_i
    return p_m^{y_m} }
```

IndexMax (ausnahme) { liefert den Index k mit $y_k \geq y_i \ \forall i \neq$ ausnahme
$k = 0$
for $(0 \leq i < D)$
if $(y[k] < y[i]$ && $i \neq$ ausnahme) $k = i$
return k }

- Quadrate:
  Der Quotient q ist meistens 1, d.h. die Quadrate sind vernachlässigbar.
- Multiplikationen: Anzahl der Multiplikationen $(p_n = (p_m)^q p_n)$
  = Anzahl der Divisionen des Euklidischen Algorithmus
  $\sim$ Anzahl der Divisionen des Euklidischen Algorithmus von y_m und y_n

Satz 2.22 *Die Anzahl der Divisionen eines Euklidischen Algorithmus von* $a, b \leq N \approx 2 \ln N \Rightarrow$ *die Komplexität von (2.21) ist*

$$C \sim k \cdot S + 2 \cdot \ln(\max(y_i)) \cdot M \tag{21}$$

mit k klein und mit der Vorberechnung

$$C \sim k \cdot S + 2 \cdot \ln(\max(y_i)) \cdot M + (D-1)M \tag{22}$$

2.5 Fixe-Exponent Methode

Idee: Der Exponent wird additiv zerlegt $y = u_1 + u_2 + u_3 \Rightarrow x^{u_1} \cdot x^{u_2} \cdot x^{u_3} = x^y$

Definition 2.23 *Additive Zerlegung („addition chain") einer Zahl e der Länge s ist eine Folge von Zahlen* u_i *mit den zugehörigen* $w_i = (i_1, i_2)$ *für die gilt*

- $u_0 = 1, u_s = e$
- $\forall \quad 0 \leq i \leq s: \quad u_i = u_{i_1} + u_{i_2} \quad 0 \leq i_1, i_2 \leq s$

Definition 2.24 *„Lucas" Zerlegung („lucas chain") ist eine Spezialform der additiven Zerlegung für die gilt*

- $u_0 = 1, u_s = e$
- $\forall \quad 0 \leq k \leq s: \quad u_k = u_i + u_j \quad 0 \leq i, j \leq s$
  und $u_i = u_j$ *oder* $u_i - u_j = u_m$ *mit* $m \leq i, j$

Beispiel: 2er Potenzen $(1, 2, 4, 8, \ldots)$ und die Fibonacci-Zahlen $F_n = F_{n-1} + F_{n-2} = (1, 2, 3, 5, 8, \ldots)$
Zerlegungen von $e = 15$:

	i	0	1	2	3	4	5	6
2er Potenzen :	w_i	–	$0,0$	$1,1$	$2,2$	$3,2$	$4,1$	$5,0$
	u_i	1	2	4	8	12	14	15
Fibonacci :	w_i	–	$0,0$	$1,0$	$2,1$	$3,2$	$4,3$	$5,1$
	u_i	1	2	3	5	8	13	15
$15 = 3 \cdot 5$:	w_i	–	$0,0$	$1,0$	$2,1$	$3,3$	$4,3$	
	u_i	1	2	3	5	10	15	

d.h. 2er Potenzen und Fibonacci Zerlegungen sind im allgemeinen nicht die kürzesten Zerlegungen. *Binärdarstellung* $e = \sum_{i=0}^{\lg e} e_i 2^i \Rightarrow$ *additive Zerlegung* u_i in 2er Potenzen

1. Bilde $u_i = 2^i = 2u_{i-1} \Rightarrow (u_0 = 1, \ldots, u_{\lg e} = 2^{\lg e}), w_i = (i, i)$
2. $\forall j : 0 \leq j < \lg e$ mit $e_j = 1$ bilde $u_i = u_{i-1} + u_j, w_i = (i - 1, j)$

Algorithmus 2.25 *liefert die Indexliste W der additiven Zerlegung aus der Binärdarstellung $y = (y_0, y_1, \ldots y_{D-1})$.*

```
BinaryAdditionChain {
    for (0 ≤ i < D)
        if (y_i == 1  und  i < D − 1)        W[l(W) + i] = (l(W) − 1, i)
        W[i] = (i − 1, i − 1)                                  2er Potenzen
    return W
```

Die Länge l von W (2.25) beträgt $l(W) = \lg y + \#1(y) - 1$.

Satz 2.26 *Die additive Zerlegung einer Zahl e mit der minimalen Länge s ist NP-schwer. Es gilt die Abschätzung*

$$s \geq \lg e + \lg \#1(e) - 2.13 \tag{23}$$

mit $\#1(e)$ die Anzahl der 1er in der Binärdarstellung von e.

Algorithmus 2.27 *berechnete x^y mit Hilfe der additiven Zerlegung*

```
PowFixYAdditionChain {
    p[0] = x                                   Liste der Potenzen p[i] = x^{u_i}
    W =BinaryAdditionChain(y)          Berechnung der additiven Zerlegung
    for  (1 ≤ i < l(W))   Durchlauf der Indizes W,l(W) = Länge der Liste
        (i_1, i_2) = W[i]
        p[i] = p[i_1] · p[i_2]                                  Multiplizieren
    return  p[l(W)]
```

Die Komplexität von (2.27) hängt nur von Multiplikationen M ab,

$$C \sim l(W) \cdot M \tag{24}$$

mit $l(W)$ = Länge der additiven Zerlegung.

Bei der Verwendung von Algorithmus (2.25) für die binäre additive Zerlegung existieren $\lg y$ Elemente mit (i, i), $\forall i < \lg y$.

Verbesserung 1 *: Quadrate in (2.27) d.h.*
$p[i] = p[i_1] \cdot p[i_2] \Rightarrow$ if $i_1 == i_2 \quad p[i] = p[i_1]^2 \quad$ else $\quad p[i] = p[i_1] \cdot p[i_2]$
Die Komplexität von (2.27) mit der Verbesserung 1 beträgt $C \sim \lg y \cdot S + \#1 \cdot M$ und somit gilt für den „Average Case“

$$C \sim \lg y \cdot S + \frac{\lg y}{2} \cdot M \tag{25}$$

Verbesserung 2*: „subtraction chain“*
In Ringen mit inversen Element bzw. Körpern können $u_i < 0$ angesetzt werden und somit die Länge verringern.
Beispiel: prime Restklassenringe $\mathbb{Z}_{p^n}$

2.6 Zusammenfassung

Die verschiedenen Exponentationsalgorithmen unterscheiden sich nach der Basis und dem Exponent, d.h. variabel oder fix, und ob eine Vorberechnung stattfindet oder nicht.

variable Basis und Exponent
Das Schema der binären und B-ären Methoden ist das Durchlaufen der Stellen des Exponenten y und je nach dem Wert y_i erfolgt eine Multiplikation bzw. die Bildung eines Quadrates. Der Unterschied besteht darin, dass die binären Methoden die Bits des Exponenten durchlaufen, während bei den B-ären Methoden die Stellen bzgl. einer wählbaren Basis $B \geq 2$ durchlaufen werden. Bei B-ären Methoden ist eine Vorberechnung der Werte $x^d < x^B$ nötig. Die durchschnittliche Komplexität der binären Methode lautet

$$C \sim (\lg y) \cdot S + \frac{\lg y}{2} \cdot M$$

und für die durchschnittliche Komplexität der B-ären Methoden gilt

$$C \sim (\lg y)S + \frac{\lg y}{b}(1 - 2^{-b})M$$

ohne die Vorberechnung der x^d.

Die Erweiterung der Methoden auf mod$N$ erfolgt durch die Ersetzung der Operationen durch Operationen mod N. Es existieren spezielle Algorithmen für $x^y \bmod N$, z.B. die Montgomery-Methode, welche in $\mathbb{Z}_N$ operiert.

Fixe Basis
Die Basis x bleibt unveränderlich, während die Exponenten frei wählbar sind. Es erfolgt eine Vorberechnung für $x^d < x^G$, G die Grenze für die vorberechneten Potenzen. Der Vorteil ist, dass die Vorberechnung nur am Anfang einer Folge von Potenzen stattfindet, d.h. für $x^{y_0}, x^{y_1}, x^{y_2}, \ldots$ erfolgt die Berechnung der Potenzen x^d bei $x^{y_0=\max(y_i)}$ und wird für $x^{y_1}, x^{y_2}, \ldots$ verwendet. Es existiert eine Variante der B-ären Methode für eine fixe Basis (2.20) mit der Komplexität (ohne Vorberechnung)

$$C \sim DM \sim \frac{\lg y}{\lg B} M$$

und mit der Vorberechnung von

$$C \sim \frac{\lg y}{\lg B} M + (B-1)(D-1)M.$$

Fixer Exponent
Für den fixen Exponenten wird eine additive Zerlegung erstellt. Mittels dieser additiven Zerlegung (2.23) können die Potenzen x^y für verschiedene x schnell berechnet werden. Das Finden der minimalen Länge s einer solchen Zerlegung ist NP-schwer. Die Komplexität von Algorithmus 2.27 hängt nur von Multiplikationen M ab und beträgt

$$C \sim l(W) \cdot M$$

mit $l(W)$ = Länge der additiven Zerlegung bzw. mit Verbesserung 1,

$$C \sim \lg y \cdot S + \frac{\lg y}{2} \cdot M.$$

S/M
Das Verhältnis von S/M gibt die Komplexität von S bzgl. M an. S/M unterscheidet sich für verschiedene Arten von Multiplikationen, z.B. bei FFT-Algorithmen lautet $S/M = 2/3$.

3 Größter gemeinsamer Teiler (ggT)

Definition 3.1 *Der größte gemeinsame Teiler ggT(x, y) zweier Zahlen x, y ist jener Teiler d für den gilt:* $d|x \wedge d|y$ *und* $\forall t : t|x \wedge t|y \Rightarrow t|d$

Definition 3.2 *Einen Integritätsring* $(R, +, \cdot)$ *bezeichnet man als Euklidischen Ring, wenn es eine Gradfunktion* $\delta : \mathbb{R}\backslash\{0\} \to \mathbb{N}$ *gibt und für beliebige* $x, y \in R$ *mit* $y \neq 0$ *stets Elemente* $q, r \in R$ *existieren mit* $x = qy + r, r = 0$ *oder* $\delta(r) < \delta(y)$.

Ein Euklidischer Ring besitzt die Primelementeigenschaft, d.h. die Darstellung mittels Primfaktoren p_i und der Einheit α

$$x = \alpha p_1^{e_1} p_2^{e_2} \cdots p_n^{e_n} \quad (e_i \geq 0) \qquad \textit{und} \qquad y = \beta p_1^{f_1} p_2^{f_2} \cdots p_n^{f_n} \quad (f_i \geq 0)$$

ist bis auf Reihenfolge und Assoziiertheit der Primfaktoren p_i eindeutig.

$$\Rightarrow ggT(x, y) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_n^{\min(e_n, f_n)}$$

3.1 Euklidischer Algorithmus

Der bekannteste Algorithmus zur Bestimmung des ggT ist der Euklidische Algorithmus.

Satz 3.3 (Euklidischer Algorithmus) $(R, +, \cdot)$ *ein Euklidischer Ring und* $x, y \in R$ *mit* $y \neq 0$. *Bildet man dann mit* $r_0 := x$ *und* $r_1 := y$ *die „Divisionskette“*

$$r_0 = q_0 r_1 + r_2 \text{ mit } \delta(r_2) < \delta(r_1)$$

$$r_1 = q_1 r_2 + r_3 \text{ mit } \delta(r_3) < \delta(r_2)$$

$$\vdots$$

$$r_{n-2} = q_{n-2} r_{n-1} + r_n \text{ mit } \delta(r_n) < \delta(r_{n-1})$$

$$r_{n-1} = q_{n-1} r_n$$

für $n > 0$ *ergibt die n-te Division den Rest* $0 \Rightarrow r_n = ggT(x, y)$.

Satz 3.4 (Lamé) *Sind* x, y *ganze Zahlen mit* $x > y > 0$ *und erfordert die Anwendung des Euklidischen Algorithmus auf* x *und* y *genau* n *Divisionsschritte, so gilt* $x \geq F_{n+2}$ *und* $y \geq F_{n+1}$.

Beweis: Fibonacci-Zahlen : $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$
Induktion nach n:
$n = 1 \qquad y \geq 1 = F_2, \quad x \geq 2 = F_3 \quad x > y$
$n - 1 \to n \qquad x = q_0 y + r_2$

$$\left.\begin{array}{l} y = q_1 r_2 + r_3 \\ \vdots \\ r_{n-1} = q_{n-1} r_n \end{array}\right\} n - 1 Divisionen$$

$$\left.\begin{array}{l} y \geq F_{(n-1)+2} = F_{n+1} \\ r_2 \geq F_{(n-1)+1} = F_n \end{array}\right\} \Rightarrow x = q_0 y + r_2 \geq y + r_2 \geq F_{n+1} + F_n = F_{n+2}$$

Satz 3.5 *Die Anzahl der Divisionen beim Euklidischen Algorithmus auf zwei ganzen Zahlen $y \leq x \leq N$ ist durch $|\log_\Phi(\sqrt{5}N)| - 2 \approx 2.078 \ln N + 1.672$ nach oben beschränkt.*

Beweis: Es gilt die Näherung für F_n

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right], n = 0, 1, 2, \ldots$$

$$\Phi = \frac{1+\sqrt{5}}{2} \qquad F_n \approx \frac{\Phi^n}{\sqrt{5}}$$

nach Lamé gilt $N \geq x \geq F_{n+2} \approx \frac{\Phi^{n+2}}{\sqrt{5}}$. $\Rightarrow n = \log_\Phi(\sqrt{5}N) - 2$

Algorithmus 3.6 (Euklid) *berechnet $ggT(x, y)$ für ganze Zahlen x, y*

```
Euclidean {
    while  (y ≠ 0)
        r = x mod y                                   Division
        x = y
        y = r
    return |x| }
```

Beispiel: $x = 7239, y = 4327 \Rightarrow ggT(7239, 4327) = 1$

x	y	$r = x \bmod y$
7239	4327	2912
4327	2912	1415
2912	1415	82
1415	82	21
82	21	19
21	19	2
19	2	1
2	1	0
1	0	–

3.2 Erweiterter euklidischer Algorithmus

Satz 3.7 *Sei $(R, +, \cdot)$ ein Euklidischer Ring und $x, y \in R$. Nach Abbruch des Euklidischen Algorithmus gilt*

$$r = ggT(x, y) = ax + by \quad \text{da} \quad r_i = a_i x + b_i y \tag{26}$$

Beweis: Induktion von a_k, b_k nach k mit Vorschrift $a_k = a_{k-2} - q_{k-2}a_{k-1}$ und $b_k = b_{k-2} - q_{k-2}b_{k-1}$

$k = 0, 1:$ $a_0 = 1, a_1 = 0$ und $b_0 = 0, b_1 = 1$
$r_0 = x = 1 \cdot x + 0 \cdot y$ und $r_1 = y = 0 \cdot x + 1 \cdot y$

$k - 1 \to k:$ $a_k \cdot x + b_k \cdot x = a_{k-2}x - q_{k-2}a_{k-1}x + b_{k-2}y - q_{k-2}b_{k-1}y =$
$(a_{k-2}x + b_{k-2}y) - q_{k-2}(a_{k-1}x + b_{k-1}y) =$
$r_{k-2} - q_{k-2}r_{k-1} = r_k$

Satz 3.8 (Erweiterter euklidischer Algorithmus) *Es werden die Vektoren $v_0 = (r_0, a_0, b_0)$, $v_1 = (r_1, a_1, b_1)$ und $v_2 = (r_2, a_2, b_2)$ verwendet.*

1. *Initialisierung: $v_0 = (x, 1, 0)$ und $v_1 = (y, 0, 1)$*
2. *Ist $r_1 = 0 \Rightarrow$ bricht der Algorithmus ab*
3. *Berechnung des Quotienten $q = \lfloor \frac{r_0}{r_1} \rfloor$*
4. *$v_2 = v_0 - qv_1, \quad v_0 = v_1, \quad v_1 = v_2$ gehe zu 2.*

Der Reduktionsschritt $v_2 = v_0 - qv_1$ wird für einen Vektor (3 Werte) ausgeführt, 3 Multiplikationen mit q und Subtraktionen, $\Rightarrow$ der Aufwand beträgt $O(C_{Euklid})$.

Lemma 3.9 *$x \in \mathbb{Z}_p, p$ prim $\Rightarrow ggT(x, p) = 1 = ax + bp \Rightarrow ax \equiv 1 \bmod p$*

Algorithmus 3.10 (Erweiterter Euklidischer Algorithmus)

```
ExtEuclidean {
    v0 = (x, 1, 0),        v1 = (p, 0, 1)          Initialisierung
    while  (v1[0] ≠ 0)
        q = ⌊v0[0] / v1[0]⌋                        Quotient q
        v2 = v0 − q · v1
        v0 = v1
        v1 = v2
    return (v0[1]) }                    return v0 = (ggT(x, p), a, b)
```

Der Algorithmus (3.10) liefert x für die Parameter (x, y) mit $ax + by = ggT(x, y)$ den Vektor $[ggT(x, y), a, b]$ und somit für (x, p) das Inverse x^{-1}.

Beispiel: $x = 31, y = 907$

v_0			v_1			t		
31	1	0	907	907	−30	−876	1	−1
						−438	454	−16
						−219	227	−8
31	1	0	219	680	−23	−188	228	−8
						−94	114	−4
						−47	57	−2
31	1	0	47	850	−29	−16	58	−2
						−8	29	−1
						−4	468	−16
						−2	234	−8
						−1	117	−4
31	1	0	1	790	−27	30	118	−4
						15	59	−2
15	59	−2	1	790	−27	14	176	−6
						7	88	−3
7	88	−3	1	790	−27	6	205	−7
						3	556	−19
3	556	−19	1	790	−27	2	673	−23
						1	790	−27
1	790	−27	1	790	−27	0	0	0

$\Rightarrow ggT(31, 907) = 1 = 790 \cdot 31 - 27 \cdot 907$

3.3 Lehmer-Variante des erweiterten euklidischen Algorithmus

Ziel: Effizientsteigerung des Algorithmus
Idee: Im euklidischen Algorithmus ist die Division die Rechenoperation mit den größten Aufwand, der Quotient q ist meistens „klein“.
Die Idee von Lehmer ist die Ersetzung der gesamten Division durch eine Division der führenden Stellen, da für den Quotienten gilt

$$q = \frac{a}{b} \approx \frac{a_k}{b_k} \tag{27}$$

mit a_k, b_k, die k führenden Stellen.

Lemma 3.11 *Die Wahrscheinlichkeit $P(q)$, dass ein Quotient im Euklidischen Algorithmus genau den Wert q hat, beträgt*

$$P(q) = \log \frac{(q+1)^2}{(q+1)^2 - 1} \tag{28}$$

z.B. $P(1) = 0.415, P(2) = 0.1699, P(3) = 0.0931, P(4) = 0.0589, \ldots$

Seien a und b ganze Zahlen mit $a \geq b > 0$ in Mehrfachgenauigkeit, d.h. gegeben bzgl. einer Basis M (z.B. $M = 2^{32}$ oder $M = 2^{64}$). Operationen in Mehrfachgenauigkeit sind durch Operationen in Einfachgenauigkeit (d.h. mit Variablen $< M$) zu ersetzen.

Mehrfachgenauigkeit → Einfachgenauigkeit
Dem euklidischen Algorithmus liegt eine lineare Transformation zugrunde

$$\begin{pmatrix} u \\ v \end{pmatrix} \rightarrow \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} Au + Bv \\ Cu + Dv \end{pmatrix}.$$

Der Ansatz der Lehmer-Methode ist die Berechnung der Quotienten $q(= \frac{x}{y})$ des euklidischen Algorithmus mit Hilfe der Variablen $\widetilde{x} \approx x, \widetilde{y} \approx y$ in Einfachgenauigkeit. Dazu wird eine Matrix $\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ schrittweise iteriert. Es wird $q = \lfloor \frac{\widetilde{x}+A}{\widetilde{y}+C} \rfloor$ und $\bar{q} = \lfloor \frac{\widetilde{x}+B}{\widetilde{y}+D} \rfloor$ berechnet, falls $q = \bar{q}$ folgt die Matrixmultiplikation

$$\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \cdot \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} C & D \\ A - qC & B - qD \end{pmatrix}$$

welche dem euklidischen Schritt entspricht. Danach erfolgt die Ersetzung der Matrix mit dem Resultat und $\widetilde{x} = \widetilde{y}$, $\widetilde{y} = \widetilde{x} - q\widetilde{y}$. Es gelten die Ungleichungen $0 \leq \widetilde{x} + A, \widetilde{y} + D \leq M$ und $0 \leq \widetilde{x} + B, \widetilde{y} + C < M \Rightarrow$ sämtliche Berechnungen erfolgen in Einfachgenauigkeit (Effizienzsteigerung). Die Iteration wird beendet falls $q \neq \bar{q}$ oder einer der Terme $\widetilde{y} + C, \widetilde{y} + D = 0$.
Bei Abbruch der Iteration und $B = 0$ wird eine Korrektur der Variablen in Mehrfachgenauigkeit durchgeführt, die Korrektur tritt mit einer Wahrscheinlichkeit von $\log(1 + \frac{1}{M})$ auf.
Die Iteration wird solange wiederholt bis $y < M \Rightarrow ggT$ mit anderer Methode in Einfachgenauigkeit berechnen.

Algorithmus 3.12 (Lehmer-Variante von Euklid)

```
LehmerEuclidean(x,y) {
    while (y ≥ M) {
        liefert die höchste Stelle bzw. = 0
        x̃ = x [ HighDigit(x) ]        ỹ = y [ HighDigit(x) ]
        A = 1, B = 0, C = 0, D = 1        Variablen in Einfachgenauigkeit
        while (ỹ + C ≠ 0 ∧ ỹ + D ≠ 0 ∧ q == q̄) {
            q = ⌊(x̃+A)/(ỹ+C)⌋,   q̄ = ⌊(x̃+B)/(ỹ+D)⌋        Berechnung der Quotienten
            if (q == q̄)                                  keine Korrektur der Variablen
                t = A − qC, A = C, C = t, t = B − qD, B = D, D = t
                t = x̃ − qỹ, x̃ = ỹ, ỹ = t }                Euklidischer Schritt
```

Korrektur der Variablen, Berechnung in Mehrfachgenauigkeit
if $(B == 0)$ $T = x \bmod y$, $x = y$, $y = T$
else $T = Ax + By$, $u = Cx + Dy$, $x = T$, $y = u$
if $y == 0$ return x
return $ggT(x, y)$ } Berechnung in Einfachgenauigkeit

Variablen in Einfachgenauigkeit: A, B, C, D, t
Variablen in Mehrfachgenauigkeit: T
Es werden 2 Folgen des Euklidischen Algorithmus A, C, q und $B, D, \bar{q}$ gebildet, falls sich die Quotienten unterscheiden, erfolgt eine Korrektur in Mehrfachgenauigkeit.

Ein Durchlauf der Lehmer-Variante entspricht k Reduktionsschritten des euklidischen Algorithmus $\Rightarrow$ die Anzahl der Durchläufe der Lehmer-Variante entspricht $k \cdot Anzahl_{Euklid} \Rightarrow O(Anzahl_{Euklid}) = O(2.078 \ln N + 1.672)$

Algorithmus 3.13 (Erweiterte Lehmer-Variante von Euklid)

ExtLehmerEuclidean(x,y) {
$Ma = [1, 0]$, $Mb = [0, 1]$ $ggT(x, y) = Ma[0]x + Ma[1]$
Algorithmus identisch mit 3.12 bis auf Korrektur in Mehrfachgenauigkeit
if $(B == 0)$
$qext = x/y$, $T = x \bmod y$, $x = y$, $y = T$
Berechnung für der Tabellen Ma, Mb für a, b
$MT = Ma - qext \cdot Mb$, $Ma = Mb$, $Mb = MT$
else
$T = Ax + By$, $u = Cx + Dy$, $x = T$, $y = u$
$MT = Ma \cdot A + Mb \cdot B$, $MU = Ma \cdot C + Mb \cdot D$
$Ma = MT$, $Mb = MU$
if $y == 0$ return $(x, Ma[0], Ma[1])$
Verwenden der berechneten Tabellen Ma, Mb für $ExtGcd$
return $ExtGcd(x, y, Ma, Mb)$ }

3.3.1 Modifzierte Lehmer-Variante

Es wird eine Modifikation der Lehmer-Variante besprochen mit den Funktionen ModLehmer und ML sowie einen zusätzlichen Parameter k.
Die Funktion ModLehmer führt die Korrektur in Mehrfachgenauigkeit aus (entspricht in 3.12 if $B = 0 \ldots$) und ruft die Funktion ML für Berechnungen in Einfachgenauigkeit auf. ML berechnet die Variablen in Einfachgenauigkeit d.h. Iteration aus 3.12.
Die Variablen U, V sind in Mehrfachgenauigkeit gegeben und $B(x)$ gibt die Anzahl der Bits von x an. Der Parameter k gibt die Anzahl der Bits für Einfachgenauigkeit an $(2^k = M)$, für die meisten Computer gelten die Grenzen $0 < k \leq 64$.

Algorithmus 3.14 $U \geq V > 0$ *und* $k > 0$

LehmerMod(U, V, k) {
  while $(V \neq 0)$ {
    Berechnungen in Einfachgenauigkeit ($\star$ Matrixmultiplikation)
    if $(B(U) - B(V) < \frac{k}{2})$ $\quad (U, V) = \mathsf{ML}(U, V, k) \star (U, V)$
    Korrektur in Mehrfachgenauigkeit
    $T = U \bmod V, \quad U = V, \quad V = T$ }
  return (U) }

Algorithmus 3.15 $U \geq V > 0$ *und* $k > 0$

ExtLehmerMod(U, V, k) {
  while $(V \neq 0)$ {
    Berechnungen in Einfachgenauigkeit ($\star$ Matrixmultiplikation)
    if $(B(U) - B(V) < \frac{k}{2})$
      $(U, V) = \mathsf{ML}(U, V, k) \star (U, V)$
      $MT = Ma \cdot M[0,0] + Mb \cdot M[0,1]$
      $MU = Ma \cdot M[1,0] + Mb \cdot M[1,1]$
      $Ma = MT,\ Mb = MU$
    Korrektur in Mehrfachgenauigkeit
    $Q = U/V, \quad T = U \bmod V, \quad U = V, \quad V = T$
    $MT = Ma - Q \cdot Mb, \quad Ma = Mb, \quad Mb = MT$ }
  return $(U, Ma[0], Ma[1])$ }

Die Operationen (alle in Einfachgenauigkeit) in ML ersetzen einige Schritte des euklidischen Algorithmus in einem einzigen Schritt. $\bar{u}, \bar{v}$ sind die k führenden Stellen von U, V. Die Berechnungen benutzen die Folgen q_i, x_i, y_i, u_i (alle in Einfachgenauigkeit). Die Restfolge u_i und die Quotientenfolge q_i sind definiert als

$$\begin{aligned} u_0 &= \bar{u} \\ u_1 &= \bar{v} \\ u_{i+2} &= u_i \bmod u_{i+1} \quad (i \geq 0) && (29) \\ q_{i+1} &= \left\lfloor \frac{u_i}{u_{i+1}} \right\rfloor \quad (i \geq 0) && (30) \end{aligned}$$

sowie die parallelen Folgen x_i, y_i für die gilt $u_i = x_i \cdot u + y_i \cdot v \quad (i \geq 0)$

$$\begin{aligned} (x_0, y_0) &= (1, 0) \\ (x_1, y_1) &= (0, 1) \\ (x_{i+2}, y_{i+2}) &= (x_i, y_i) - q_{i+1}(x_{i+1}, y_{i+1}) \quad (i \geq 0) && (31) \end{aligned}$$

mit der Identität Das Kriterium von Jebelean wird verwendet um festzustellen falls der Quotient q_i von $\frac{V}{U}$ abweicht. ML liefert eine Matrix $\begin{pmatrix} x_{i-1} & y_{i-1} \\ x_i & y_i \end{pmatrix}$ und U, V wird durch Matrixmultiplikation berechnet $(U, V) = \mathsf{ML}(U, V, k) \star (U, V) = (x_{i-1}U + y_{i-1}V, x_iU + y_iV)$.

Algorithmus 3.16

$\mathsf{ML}(U, V, k)$ {
  Umwandeln in Einfachgenauigkeit
  $h = B(U) - k, \quad \bar{u} = U >> h, \quad \bar{v} = V >> h$
  Initialisierung
  $(x_0, y_0) = (1, 0), \quad (x_1, y_1) = (0, 1), \quad i = 0, \quad done = false$
  while (not done) {
    $q = \lfloor \frac{\bar{u}}{\bar{v}} \rfloor$ Quotient
    $(x_{i+2}, y_{i+2}) = (x_i, y_i) - q(x_{i+1}, y_{i+1})$
    $(\bar{u}, \bar{v}) = (\bar{v}, \bar{u} - q\bar{v})$ Reduktion
    $i = i + 1$
    done = true ,falls q_i korrekt (Jebelean Kriterium), sonst false
    if i gerade $\quad done = \bar{v} < -x_{i+1} \vee \bar{u} - \bar{v} < y_{i+1} - y_i$
    else $\quad done = \bar{v} < -y_{i+1} \vee \bar{u} - \bar{v} < x_{i+1} - x_i$ }
  return $[[x_{i-1}, y_{i-1}], [x_i, y_i]]$

Satz 3.17 *Für $U > V \geq 0$ und $k > 0$ benötigt der Algorithmus (3.14)*

$$O\Big(k + \frac{\log V}{k}\Big)$$

Durchläufe der while-Schleife um $ggT(U, V)$ zu berechnen (Referenz 8.).

Beweis: Es exisitieren 3 Typen von Durchläufen der while-Schleife

1. $B(U) - B(V) \leq \frac{k}{2}$ und $B(V) \geq k$
2. $B(U) - B(V) > \frac{k}{2}$ und $B(V) \geq k$
3. $B(V) < k$

Die Anzahl kann für Typ 1 und 2 jeweils mit $O(1 + \frac{\log V}{k})$ und für Typ 3 mit $O(k)$ abgeschätzt werden.

d.h. für U, V mit der Länge von n Bits und $k = \lfloor \frac{\log n}{4} \rfloor$ kann die Komplexität des Algorithmus mit $O(\frac{n^2}{\log n})$ abgeschätzt werden.

3.4 Binärvariante des Euklidischen Algorithmus

Eine Variante des Euklidischen Algorithmus ist die Binärvariante von Stein. Verwendete Regeln für u, v:

- u, v beide gerade $\Rightarrow ggT(u,v) = 2 \cdot ggT(\frac{u}{2}, \frac{v}{2})$
- u gerade und v ungerade $\Rightarrow ggT(u,v) = ggT(\frac{u}{2}, v)$
- u, v beide ungerade $\Rightarrow ggT(u,v) = ggT(u-v, v) \wedge |u-v| < \max(u,v)$

Algorithmus 3.18 (Binärvariante von Euklid) *berechnet* $ggT(u,v)$
$v_2(m)$ *gibt die Anzahl der niederwertigen* 0 *in der Binärrepräsentation an*
$\Rightarrow 2^{v_2(m)} | m \wedge 2^{v_2(m)+1} \not| m$.

```
BinaryEuclidean  {
    β = min(v2(u), v2(v))                          2^β teilt ggT(u,v)
    u = u >> v2(u),    v = v >> v2(v)               ⇒ u, v ungerade
    while (u ≠ v) {
        (u, v) = (min{u, v}, |v − u| >> v2(|v − u|)) }
    return  2^β · u }                                  ggT(u,v)
```

Beispiel: $x = 2024, y = 1540 \Rightarrow \beta = 2, u = 506, v = 385$

$u = \min(u,v)$	$v = \lvert v-u\rvert >> v_2(\lvert v-u\rvert)$	u	v
506	385	385	121
121	$264 >> 3 = 33$	33	$88 >> 3 = 11$
11	$22 >> 1 = 11$		

$\Rightarrow ggT(u,v) = 2^2 \cdot 11 = 44$

In jedem Durchlauf wird eine der Variablen mindestens halbiert $\Rightarrow$ Anzahl der Durchläufe ist $O(\log_2 N)$. Der Vorteil der binären Methode ist das Vermeiden der Divisionen, da nur Shifts verwendet werden. Der Gesamtaufwand beträgt $O(\log^2 N)$ mit $a, b \leq N$.

Es existiert ein erweiterte Variante des binäre Algorithmus (3.19), der $[ggT(a,b), x, y]$ bzw. x^{-1} zurückliefert.

Algorithmus 3.19 (Erweiterter binärer euklidischer Algorithmus)

ExtBinaryEuclidean {
$\beta = \min\{v_2(x), v_2(y)\}, \quad x = x >> \beta, \quad y = y >> \beta$
$v_0 = (x, 1, 0), \qquad v_1 = (y, y, 1 - x)$ Initialisierung
if $(v_2(x) > 0) \qquad t = (x, 1, 0)$ x gerade
else $\qquad t = (-y, 0, -1)$ x ungerade

while $(t[0] \neq 0)$ { Rest $\neq 0$
while $(v_2(t[0]) > 0)$ { Rest gerade
if ($2|t[1]$ and $2|t[2]$) $\qquad t = t/2$ beide gerade
else $\qquad t = (t[0], t[1] + y, t[2] - x)/2$ }
if $(t[0] > 0) \qquad v_0 = t$
else $\qquad v_1 = (-t[0], y - t[1], -x - t[2])$
$t = v_0 - v_1$
if $(t[1] < 0) \qquad t = (t[0], t[1] + y, t[2] - x)$ }

return $(2^\beta v_0[0]), v_0[1], v_0[2])$ } $x^{-1} = v_0[1]$

Beispiel: $ggT(7239, 4327) = 1 = 7239 \cdot 2058 - 4327 \cdot 3443$

q	v_0			v_1			v_2		
1	7239	1	0	4327	0	1	2912	1	-1
1	4327	0	1	2912	1	-1	1415	-1	2
2	2912	1	-1	1415	-1	2	82	3	-5
17	1415	-1	2	82	3	-5	21	-52	87
3	82	3	-5	21	-52	87	19	159	-266
1	21	-52	87	19	159	-266	2	-211	353
9	19	159	-266	2	-211	353	1	2058	-3443
2	2	-211	353	1	2058	-3443	0	-4327	7239
	1	2058	-3443						

3.5 k-äre Variante des Euklidischen Algorithmus

Der k-äre ggT von Sorenson ist eine Verallgemeinerung des binären Algorithmus von Stein.
Idee: Ersetzung von 2 durch einen Faktor k.

Ersetzung von $2 \rightarrow k$

Binäre Regeln für (u, v)	k-äre Regeln für (u, v)
u, v beide gerade $\Leftrightarrow ggT(u, 2) = 2 \wedge ggT(v, 2) = 2$ $\Rightarrow ggT(u, v) = 2 \cdot ggT(u/2, v/2)$	$ggT(u, k) = d \wedge ggT(v, k) = d$ $\Rightarrow ggT(u, v) = d \cdot ggT(u/d, v/d)$
Genau eine Zahl ungerade $\Leftrightarrow ggT(u, 2) = 2 \wedge ggT(v, 2) = 1$ $\Rightarrow ggT(u, v) = ggT(u/2, v)$	$ggT(u, k) = d \wedge ggT(v, k) = 1$ $\Rightarrow ggT(u, v) = ggT(u/d, v)$
Beide Zahlen ungerade $\Leftrightarrow ggT(u, 2) = 1 \wedge ggT(v, 2) = 1$ $\Rightarrow ggT(u, v) = ggT(\lvert u - v \rvert, v)$	$ggT(u, k) = 1 \wedge ggT(v, k) = 1$ Finden von $a, b, au + bv \equiv 0 \bmod k$ $u > v \Rightarrow ggT(\lvert au + bv \rvert / k, v)$ $u < v \Rightarrow ggT(u, \lvert au + bv \rvert / k)$

$\Rightarrow$ Reduktionsschema:

1. Speichern $g = ggT(u, v, k)$ und weiter mit $\frac{u}{g}, \frac{v}{g}$
2. $u \rightarrow \frac{u}{ggT(u,k)}$ oder $v \rightarrow \frac{v}{ggT(v,k)}$
3. $(u, v) \rightarrow (min(u, v), |au + bv|/k)$
4. Wenn $u = 0 \vee v = 0$, Resultat $= g \cdot ggT(u, v)$, falls nicht weiter mit 2.

Lemma 3.20 *Für u, v, a, b ganze Zahlen, falls $g = ggT(u, v)$ und $h = ggT(v, au + bv)$, dann folgt $g|h$ und $\frac{h}{g}|a$.*

Beweis: Lemma folgt aus $h = ggT(v, au)$.

Damit Reduktionsschema korrekt abläuft, müssen **vor** der Reduktion die gemeinsamen Teiler von u, v, die mögliche Werte von a, b teilen, eliminiert werden. **Nach** der Reduktion müssen überflüssige Faktoren, durch die Teiler von a, b entstanden, im Resultat eliminiert werden.

Daraus folgt, dass die Teiler von a, b benötigt werden und dies führt zur maximalen Größe von a, b.

Lemma 3.21 *$k > 1$ ganze Zahl. Für jedes Paar (u, v) von ganzen Zahlen, existiert ein Paar von ganzen Zahlen (a, b) mit $0 < |a| + |b| \leq 2\lceil\sqrt{k}\rceil$ und $au + bv \equiv 0 \bmod k$.*

Beweis:
Die Abbildung $f := (a, b) \to au + bv \bmod k$ hat k mögliche Ergebnisse. Für die Menge $M = \{(a, b)\}$ mit $1 \leq a, b, \leq \lceil\sqrt{k}\rceil + 1$, gilt $|M| > k \Rightarrow \exists 2$ Paare $(a_1, b_1), (a_2, b_2) \in M$ mit $f(a_1, b_1) = f(a_2, b_2)$ und $a_1 \neq a_2$ oder $b_1 \neq b_2$.
$\Rightarrow f(a_1 - a_2, b_1 - b_2) = 0$ und $0 < |a_1 - a_2| + |b_1 - b_2| \leq 2\lceil\sqrt{k}\rceil$

Schema des k-ären Algorithmus (4 Phasen):

1. Vorberechnung von a, b und $ggT(x, k) \quad 0 \leq x < k$
2. Eliminieren und Speichern der Teiler d von u, v mit $d \leq \lceil\sqrt{k}\rceil + 1 \vee d|k$
3. Reduktionsschema
4. Eliminieren der überflüssigen Teiler $d < \sqrt{k} + 1$ im Resultat und multiplizieren mit Teilern in (2) $\Rightarrow ggT(u, v)$.

Beispiel: Berechnung von $ggT(u, v)$ für $u = 789, v = 453$ mit $k = 7$

3.5.1 Phase 1 : Vorberechnung

Gesucht: 4 Tabellen für k-äre Algorithmus
Die Länge der Tabellen ist mit $O(k)$ beschränkt. Für verschiedene u, v mit denselben k wird die Vorberechung nur einmal ausgeführt, da die Vorberechnung nur von k abhängig ist.

Tabelle G und I:
$G[x] = ggT(x, k)$, mit $0 \leq x < k$ falls $G[x] = 1 \Rightarrow I[x] = x^{-1} \bmod k$
Die Berechnung des ggT kann mit Hilfe des erweiterten euklidischen Algorithmus erfolgen ($O(k \log k)$).
Tabelle P:
P enthält alle Primzahlen und alle Teiler (prim) von k
$P = \{p \text{ mit } p <\approx \sqrt{k} + 1 \wedge p \in \mathbb{P}\} \cup \{p \text{ mit } p|k \wedge p \in \mathbb{P}\}$
Tabelle A und B:
Es gilt (u, v) relativ prim zu k, sei nun $x = \frac{u}{v} \bmod k \Rightarrow u - xv \equiv 0 \bmod k$.
In den Tabellen A, B stehen die a, b für jedes x d.h. $b - ax \equiv 0 \bmod k$ für $a = A[x], b = B[x]$. Laut Lemma (3.21) existieren a, b mit $0 < |a| + |b| \leq 2\sqrt{k}$ wobei $|a| = a \vee -a$ und $|a|_k = a \vee k - a$.

Algorithmus (3.22) : Finden von a, b, $|a| + |b|$ minimiert, mit der Laufzeit $O(k\sqrt{k})$. Suche ein a und $b = -ax \bmod k$ mit $a + |b|$ „klein“, dann setze $A[x] = a, B[x] = b$. Diese a, b erfüllen die Bedingung von Lemma (3.21).

Algorithmus 3.22 $A[x] = B[x] = k \quad \forall x : 0 \le x < k$

FindAB {
  for $(0 \le x < k)$
    for $1 \le a < \lceil\sqrt{k}\rceil$
      $b = |-a|_k x \bmod k$
      if $(a + |b| < |A[x]| + |B[x]|) \quad A[x] = a, B[x] = b$ }

Beispiel: $u = 627, v = 363$

x	0	1	2	3	4	5	6
G	7	1	1	1	1	1	1
I	–	1	−3	−2	2	3	−1
P	2	3	7				
A	–	1	1	2	2	1	1
B	–	1	−2	1	−1	2	1

3.5.2 Phase 2 : Eliminierung der Teiler

Die gemeinsamen Teiler d von u, v werden eliminiert, d.h. $(u, v) \to (\frac{u}{d}, \frac{v}{d})$ $\Rightarrow$ durchlaufen der Werte d der Tabelle P. Das Produkt der gemeinsamen Teiler wird in g gespeichert.

$g = 1$ Initialisierung des $ggT(u, v)$
for $(0 \le i < P.\text{Length})$ $\forall d \in P$
  while $(P[i] \,|\, u \wedge P[i] \,|\, v)$ Eliminierung des Teilers $d = P[i]$
    $g = g \cdot P[i], \quad u = \frac{u}{P[i]}, \quad v = \frac{v}{P[i]}$

Beispiel: $3|627, 3|363 \Rightarrow g = 3, u = 209, v = 121$

3.5.3 Phase 3 : Reduktion

Die Punkte 2 bis 4 des Reduktionsschemas:

while $(u \neq 0 \wedge v \neq 0)$
  $\bar{u} = u \bmod k, \quad \bar{v} = v \bmod k$ Reste sind Indizes für Tabellen
  if $G[\bar{u}] > 1 \quad u = \frac{u}{G[\bar{u}]}$ $ggT(u, k) = d \Rightarrow ggT(u, v) = ggT(u/d, v)$
  else if $G[\bar{v}] > 1 \quad v = \frac{v}{G[\bar{v}]}$ $ggT(v, k) = d \Rightarrow ggT(u, v) = ggT(u, v/d)$
    else $ggT(u, v) = ggT(\min(u, v), |au + bv|/k)$
      $x = \bar{u} \cdot I[\bar{v}]$ Hilfsvariable
      $t = |A[x] \cdot u + B[x] \cdot v|/k)$ Reduktion
      if $(u > v) \quad u = t$ else $v = t$

Beispiel: $u = 209, v = 121$

u	v	$\bar{u}$	$\bar{v}$	x	a	b	t
209	121	6	2	3	2	1	$\|418 + 121\|/7 = 77$
77	121	–	–	–	–	–	–
11	121	4	2	2	1	-2	$\|11 - 242\|/7 = 33$
11	33	4	5	5	1	2	$\|11 + 66\|/7 = 11$
11	11	4	4	1	1	-1	$\|11 - 11\|/7 = 0$
0	11	–	–	–	–	–	–

$\Rightarrow ggT(209, 121) = 11$

3.5.4 Phase 4 : Eliminieren der überflüssigen Teiler

d.h. die überfüssigen Teiler $\in P$ von a, b aus dem Resultat eliminieren $\Rightarrow$ Resultat $t \cdot g$ (Teiler aus Phase 2) $= ggT(u, v)$.

```
if (v = 0)   t = u else t = v
for  (0 ≤ i < P.Length)                                   ∀d ∈ P
    while  (P[i] | t)   t = t / P[i]          Eliminierung des Teilers d = P[i]
ggT(u, v) = t · g
```

Beispiel: 11 besitzt keine überflüssigen Teiler $\Rightarrow 3 \cdot 11 = ggT(627, 363) = 33$

3.5.5 Aufwand

Definition 3.23 *p prim und $n > 0$ dann gilt $p^e \| n$, falls $p^e | n$ und $p^{e+1} \nmid n$. $Q(n)$ ist definiert als $Q(n) = \min \{ p^e : p^e \| n$ und p prim $\}$.*

Zitat aus Referenz 9.

Satz 3.24 *Gegeben: u, v, $q = Q(k)$ und $n = \log_2(uv)$. Falls $k = O((\frac{n}{\log n})^2)$, dann beträgt der Aufwand des k-ären Algorithmus $O(n^2 \frac{\log k}{\log q})$ Bitoperationen.*

Beweis: Der Aufwand der Bitoperationen pro Phase

1. Vorberechnung : $O(k \log^2 k)$
2. Eliminierung der Teiler (Phase 2 + 4): Die beiden Phasen besitzen denselben Aufwand. Laut Primzahlen-Theorem existieren $O(\frac{\sqrt{k}}{\log k})$ Divisionen mit Rest $\neq 0$ für jeden möglichen Wert des Teilers d. Somit beträgt der Aufwand $O(n\sqrt{k})$. Der Aufwand für Divisionen mit Rest $= 0$ ist $O(n^2)$, da für die Teiler $d_1, \ldots d_r$ gilt $\sum_{i=1}^{r} \log d_i \leq n$. Der Aufwand beträgt $O(n^2 + n\sqrt{k})$.
3. Reduktion: Der Aufwand pro Iteration beträgt $O(n \log k)$, es existieren $O(\frac{n}{\log q})$ Iterationen, somit ergibt sich der Aufwand $O(n^2 \frac{\log k}{\log q})$.

Der Gesamtaufwand beträgt $O(n^2\frac{\log k}{\log q}+k\log^2 k+n\sqrt{k}) = O(n^2\frac{\log k}{\log q})$. Durchschnittlich beträgt der Aufwand somit $O(\frac{n^2}{\log k}) = O(\frac{n^2}{\log n})$. Für k eine Primzahlpotenz ergibt sich der schlechteste Fall von $O(n^2)$.

3.5.6 Finden von a, b

In der ursprünglichen Version des k-ären Algorithmus werden die a, b der Reduktion $au + bv \bmod k$ vorberechnet und in einer Tabelle gespeichert. Es existiert ein einfacher Algorithmus von Jebelean, Weber (Referenz 11.) welcher die a, b mit $0 < |a|, |b| < \sqrt{k}$ berechnet während der Laufzeit.

Der folgende Algorithmus berechnet „kleine“ (n, d) mit $ny \equiv dx \pmod k$, $0 < n, |d| < \sqrt{k}$. Die gesuchten a, b des k-ären Algorithmus sind $a = -d$ und $b = n$.

Algorithmus 3.25 (Jebelean Weber Algorithmus (JWA))
$x, y > 0, k > 1, ggT(k, x) = ggT(k, y) = 1$

JWA (x, y, k) {
  Initialisierung der Variablen
  $c = \frac{x}{y} \bmod k, \quad f_1 = (n_1, d_1) = (k, 0), \quad f_2 = (n_2, d_2) = (c, 1)$
  while $(n_2 \geq \sqrt{k})$ {
    $f_1 = f_1 - \lfloor \frac{n_1}{n_2} \rfloor f_2$
    Tauschen(f_1, f_2) }
  return f_2 }

Der schlechteste Fall (Worst Case) tritt bei den Fibonacci-Zahlen auf. Daraus ergibt sich die Abschätzung, siehe Euklid, von $(\log_2(k))^2$. Es existieren verschiedene Varianten bzw. Modifikationen des JWA.

Definition 3.26 $A_k = (0, \sqrt{k}), \quad B_k = (k - \sqrt{k}, \sqrt{k}), \quad U_k = A_k \cup B_k$
Für $(x, y) \in U_k \times U_k$ ist die T-Transformation definiert als

- $x, y \in A_k \Rightarrow T(x, y) = (x, y)$
- $x \in A_k, y \in B_k \Rightarrow T(x, y) = (x, y - k)$
- $x \in B_k, y \in A_k \Rightarrow T(x, y) = (k - x, -y)$
- $x, y \in B_k \Rightarrow T(x, y) = (k - x, k - y)$

Lemma 3.27 *Für jedes $(x, y) \in U_k \times U_k$ gilt für $x', y' = T(x, y)$*

1. $0 < x', |y'| < \sqrt{k}$
2. $x'y \equiv xy' \bmod k$

Durch Kombination von T-Transformation und JWA erhält man den Algorithmus JWAT. Der Algorithmus (3.28) besitzt die dieselbe „worst-case“ Abschätzung $(\log_2(k))^2$ wie der JWA, allerdings ist (3.28) im Durchschnitt schneller.

Algorithmus 3.28 $x, y > 0, k > 1, ggT(k, x) = ggT(k, y) = 1$

JWAT(x, y, k) {
  if $(a, b) \in U_k \times U_k \quad f_2 = T(a, b)$
  else $c = \frac{a}{b} \bmod k$
    if $c \in U_k \quad f_2 = T(c, 1)$
    else JWA
  return f_2 }

3.6 Rekursiver ggT für sehr große Zahlen

Der folgende Algorithmus von Schönhage entspricht einem rekursiven Reduktionsschema. Die zugrundlegende Idee ist die Berechnung einer 2×2 Matrix M mit $M \cdot (x, y)^T = (0, ggT(x, y))^T$. Die Matrix M ist ein Produkt einer Matrizenkette $M = M_n \cdot M_{n-1}, \cdots M_1$ mit 2×2 Matrizen M_i, M entspricht den Reduktionsschritten des Euklidischen Algorithmus.

Funktionsweise des Algorithmus:
Globale Variablen:

- lim: Grenze für die Berechnung mittels rekursiven Aufruf. Sollte $u, v \leq lim \Rightarrow$ Abbruch der rekursiven Berechnung und Aufruf von *cgcd*
- prec: maximale Bitlänge für *shgcd*
- G: globale Matrix = sukzessive Berechnung des Endresultats

Die Variablen *lim* und *prec* dienen nur zur Effizienzsteigerung.

rgcd(x,y) Hauptfunktion des Algorithmus d.h Reduktion der u, v

1. Initialisierung von $u = x, v = y$
2. Initialisierung von $G = ((1, 0), (0, 1))$
3. Aufruf von *hgcd(0,u,v)* $\Rightarrow$ liefert G
4. $(u_1, v_1)^T = G \cdot (u, v)^T$ mit der Hälfte der Stellenanzahl von (u, v)
5. falls $u, v < lim$ Rekursion beendet und Aufruf von *cgcd*
6. sonst $(u, v) = (u_1, v_1)$ und weiter mit (2).

hgcd(b,x,y) rekursive Funktion zur Berechnung von G
Die Funktion berechnet die erste Hälfte der Matrixkette $\approx M_n \cdots M_{n/2}$, wobei b die Anzahl der zu eliminierenden Bits von x, y angibt. Das Halbieren der Stellenanzahl ensteht durch mehrmaliges Aufrufen von *hgcd*:
x, y besitzen n Bits und *hgcd(0,x,y)* (erster Aufruf in *rgcd*).

hgcd(0,x,y)		
hgcd(n/2,x,y)	$G_1, (x_1, y_1)^T = G_1(x, y)^T$	$Size(x_1, y_1) = 3n/4$
hgcd(n/4,x,y)	$G_2, (x_2, y_2)^T = G_2(x_1, y_1)^T$	$Size(x_2, y_2) = n/2$

$\Rightarrow G = G_1 \cdot G_2$, halbiert die Stellenanzahl. Zur Effizienzsteigerung wird, falls die Bitanzahl unter die Grenze *prec* fällt, Auruf von *shgcd*.
shgcd(x,y) liefert G für „kleine“ x, y
Die Bitlänge von $x, y < prec$ d.h. die Matrix G kann einfach und ohne zusätzliche Rekursion gebildet werden.
cgcd(x,y) berechnet $ggT(x, y)$ für „kleine“ x, y mittels einer klassischen ggT-Methode (z.B. Euklid)

Falls die Variablen u, v in *rgcd* $< lim \Rightarrow$ Stop der Rekursion. Das Reduktionsschema ist schneller als verschiedene Varianten (wie binäre ggT-Formen mit oder ohne Lehmer-Erweiterungen) in der Region von $x, y \approx 2^{2^{16}}$, d.h. das Reduktionsschema ist für „sehr große“ Zahlen geeignet.

Aufwand: u, v von der Größe $\leq N$, $n = \log_2 N$ Anzahl der Bits von N
Die Anzahl der Durchläufe der while-Schleife von *rgcd* ist mit $O(\log n)$ beschränkt, da in jedem Durchlauf die Stellenanzahl halbiert wird.
Der Gesamtaufwand beträgt

$$C = O(C_{hgcd}(n) \log n). \tag{32}$$

Die Funktion *hgcd* enthält 2 rekursive Aufrufe. Es gilt somit

$$C_{hgcd}(n) = 2 \cdot C_{hgcd}\left(\frac{n}{2}\right) + O(C_M(n))$$

wobei $C_M(n)$ die Komplexität einer Multiplikation von 2 Zahlen mit Bitlänge n bezeichnet, $\Rightarrow C_{hgcd}$ hängt nur von $C_M(n)$ ab.

$$C = O(C_M(n) \log n) \tag{33}$$

und für $C_M = O(n \log n \log \log n)$ gilt somit

$$C = O(n \log^2 n \log \log n). \tag{34}$$

Algorithmus 3.29 (Rekursiver ggT)

$lim = 2^{256}$	Grenze für Rekursion
$prec = 32$	max. Bitlänge für shgcd

cgcd(x,y) = Euklid (x,y) klassische ggT-Methode

rgcd(x,y) { Hauptfunktion
$(u,v) = (x,y) \quad stop = true$ Initialisierung der Startwerte
while $(stop)$ Reduktionsschleife
$(u,v) = (|u|,|v|)$
if $(u < v) \quad (u,v) = (v,u)$
if $(v < lim) \quad stop = false$ Abbruch der Rekursion
else Berechnung von G
$G = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
hgcd(0,u,v) Beginn der Rekursion und Berechnung von G
$(u,v)^T = G \cdot (u,v)^T$
$(u,v) = (|u|,|v|)$
if $(u < v) \quad (u,v) = (v,u)$
if $(v < lim) \quad stop = false$ Abbruch der Rekursion
else $(u,v) = (v, u \bmod v)$
return *cgcd(u,v)* } Berechnung von $ggT(u,v)$ mit *cgcd*

hgcd(b,x,y) { rekursive Funktion
if $(y == 0)$ return Abbruch
$u = x >> b \quad v = y >> b$
$m = B(u)$ $B(u)$ ist Bitlänge von u
Bitlänge kleiner als max. Bitlänge ⇒ shgcd
if $(m < prec) \quad G = shgcd(u,v)$ return
$m = \lfloor m/2 \rfloor = m >> 1$
hgcd(m,u,v) Rekursion
$(u,v)^T = G \cdot (u,v)^T$
if $(u < 0) \quad (u, G_1, G_2) = (-u, -G_1, -G_2)$
if $(v < 0) \quad (u, G_3, G_4) = (-u, -G_3, -G_4)$
if $(u < v) \quad (u, v, G_1, G_2, G_3, G_4) = (v, u, G_3, G_4, G_1, G_2)$
if $(v \neq 0) \quad (u,v) = (v,u)$
$q = \lfloor v/u \rfloor$
$G = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} G$
$v = v - qu$
$m = \lfloor m/2 \rfloor = m >> 1$
$C = G$
hgcd(m,u,v) Rekursion
$G = GC$
return }

shgcd(x,y) Berechnung von G „kleiner“ x, y

$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
$(u, v) = (x, y)$
while $(v^2 > x)$
  $q = \lfloor u/v \rfloor$
  else $(u, v) = (v, u \bmod v)$
  $(A_1, A_3) = (A_3, A_1 - qA_3)$
  $(A_2, A_3) = (A_4, A_2 - qA_4)$
return A

3.7 Spezielle Inversionsalgorithmen

3.7.1 Spezielle Inversion basierend auf Euklidischen Alg.

Es existieren viele Varianten von Inversionsalgorithmen basierend auf den erweiterten euklidischen Algorithmus.
Folgender Algorithmus berechnet $x^{-1} \bmod p$ für p prim.

Algorithmus 3.30 *mit p prim und $x \not\equiv 0 \pmod p$*

InvSpecialEuclidean {
  $z = x \bmod p, \quad a = 1$ Initialisierung
  while $(z \neq 1)$ {
    $q = -\lfloor \frac{p}{z} \rfloor$ Quotient
    $z = p + qz$ Reduktion von z
    $a = qa \bmod p$ }
  return a } $a = x^{-1} \bmod p$

Beweis: Die einzelnen z bilden eine Folge z_j mit $\forall j : 0 < z_j < z_{j+1}$. p prim, $ggT(p, z_j) = 1 \Rightarrow p = \lfloor \frac{p}{z_j} \rfloor z_j + r$ mit $0 < r < z_j$. Somit ist $z_{j-1} = p + q \cdot z_j = p - \lfloor \frac{p}{z_j} \rfloor \cdot z_j = r$. $\Rightarrow z_j$ streng monoton fallende Folge mit $z_n = x, \ldots, z_0 = 1$ und $z_j > 0$.

Annahme: $z_i^{-1} = \prod_{j=1}^{i} q_j \quad 1 \leq i \leq n =$ Länge der z_j
Induktion nach i :
$i = 1$: $z_0 = 1 = p + q_1 \cdot z_1 \bmod p = q_1 \cdot z_1 \Rightarrow q_1 = z_1^{-1}$
$i \to i+1$: $z_i = p + q_{i+1} \cdot z_{i+1}$ und $z_i^{-1} = \prod_{j=1}^{i} q_j \Rightarrow z_i \cdot z_i^{-1} =$
$= p \cdot \prod_{j=1}^{i} q_j + z_{i+1} \cdot \prod_{j=1}^{i+1} q_j \bmod p \Rightarrow z_{i+1}^{-1} = \prod_{j=1}^{i+1} q_j$
$\Rightarrow x^{-1} = z_n^{-1} = \prod_{j=1}^{n} q_j = a$

Beispiel: : $x = 5038, p = 91033$

q	z	a	q	z	a
-18	349	-18	-260	293	4680
-310	203	-85305	-448	89	73813
-1022	75	-61562	-1213	58	27646
-1569	31	-44866	-2936	17	1825
-5354	15	-30519	-6068	13	28170
-7002	7	-68862	-13004	5	80860
-18206	3	-42517	-30344	1	16172

$x^{-1} = 16172$, $16172 \cdot 5038 \equiv 1 \bmod 91033$

Die Effizienz von Algorithmus 3.30 hängt von der Berechnung von $\lfloor \frac{p}{z} \rfloor$ ab.

Satz 3.31 *Die Anzahl der Durchäufe der while-Schleife ist $O(C_{Euklid}(p))$, Aufwand des Euklidischen Algorithmus. Der Gesamtaufwand beträgt*

$$O(C_{Euklid} \cdot C_{Div}) = O(\ln p \cdot C_{Div}) \tag{35}$$

mit C_{Div} (Aufwand einer Division).

Beweis:
Reduktion in while-Schleife entspricht ungefähr ($\approx$) Reduktion in Euklid.

Lemma 3.32
Für p zusammengesetzt, liefert (3.30) x^{-1} falls $ggT(x,p) = 1$.

Beweis: $ggT(z_i, p) = 1 \Rightarrow ggT(z_{i-1}, p) = 1 : ggT(z_i, p) = \mathsf{Euklid}(p, z_i)$
$= \mathsf{Euklid}(z_i, p - \lfloor \frac{p}{z_i} \rfloor z_i) = \mathsf{Euklid}(z_i, z_{i-1}) = ggT(z_{i-1}, z_i) \Rightarrow$ die Folge z_j erfüllt dieselben Voraussetzungen wie in Beweis für (3.30)

Lemma 3.33 *Für p zusammengesetzt und $ggT(z_i, p) = d > 1$, gilt in (3.30)* $d | z_j \, \forall j \leq i \quad \wedge \quad \exists z_m = d, m < i \Rightarrow z_{m-1} = 0$.

Beweis: $ggT(z_i, p) = d, \quad z_{i-1} = p + q \cdot z_i = d \cdot (\frac{p}{d} + q\frac{z_i}{d}) \Rightarrow d | z_{i-1}$
$z_{j-1} < z_j \wedge d | z_j \, \forall j \leq i \Rightarrow \exists z_m = d$, $z_{m-1} = p - \frac{p}{d} d = 0$

Durch Einfügen der Befehlszeile $\mathsf{if}\ (z = 0)\ \mathsf{return}\ 0$, liefert Algorithmus (3.30) auch für zusammengesetzte p, x^{-1} bzw. 0, falls x^{-1} nicht existiert.

3.7.2 Spezielle Inversion für Mersenne-Primzahl $M_p = 2^p - 1$

Folgender Algorithmus (3.34) beruht auf der speziellen Form von modM_p. Verwendete Regeln von Lemma 1.16:

1. $x \cdot 2^k \bmod M_p = x <<_p k$ d.h. Multiplikation mit 2^k entspricht einen links-zirkulären Shift (auf p Bits) von x um k Stellen
2. $x = 2^t \cdot k \bmod M_p$ mit $t \geq 0, k \geq 1 \Rightarrow x^{-1} = 2^{p-t} \cdot k^{-1}$
   Beweis: $x \cdot 2^{p-t}k^{-1} \equiv 2^t 2^{p-t} \cdot kk^{-1} \equiv 2^p kk^{-1} \equiv 1 \bmod M_p$

Algorithmus 3.34 *mit* $M_p = 2^p - 1$ *prim und* $x \not\equiv 0 \bmod M_p$

InvSpecialMersenne {
  $[a, a_n, y, y_n] = [1, 0, x, M_p]$ — Initialisierung
  while $(y_n \neq 1)$ {
    $e = \mathsf{LowBits}(y)$ — $y = 2^e \cdot k$, k ungerade
    $y = y/2^e = y >> e$ — Shift um e Stellen
    $a = 2^{p-e}a \bmod M_p$ — Links-zirkulärer Shift (1)
    Berechnung von a und y durch vorige Werte a_n, y_n
    $[a, a_n, y, y_n] = [a + a_n, a, y + y_n, y]$ }
  return a_n

Beweis: Bilde die Folgen y_n mit $y = 2^{e_n} y_n \Rightarrow y_n = \frac{y_{n-1}+y_{n-2}}{2^{t_n}}$ und a_n mit $a_n = 2^{p-e_n} \cdot (a_{n-1} + a_{n-2})$. Die Summenbildung erfolgt in $[a, a_n, y, y_n] = [a + a_n, a, y + y_n, y]$. Die Startwerte lauten $a_1 = 2^{p-e_1}, a_0 = 1$ und $y_1 = \frac{x}{2^{e_1}}, y_0 = M_p$, sie ergeben sich aus der Initialisierung.

Annahme: $a_n \cdot x \equiv y_n \bmod M_p$

Induktion nach n :

$n = 1$: $a_1 \cdot x \equiv 2^{q-e_1}x \equiv 2^q 2^{-e_1}x \equiv 2^q y_1 \equiv y_1 \bmod M_p$

$n \to n+1$: $a_{n+1} \cdot x \equiv 2^{q-e_{n+1}}(a_n + a_{n-1}) \cdot x \equiv$
$\equiv 2^{-e_{n+1}}(a_n x + a_{n-1}x) \equiv 2^{-e_{n+1}}(y_n + y_{n-1}) \equiv y_{n+1}$

$y_n \to 1 \Rightarrow a_n \cdot x \equiv 1 \Rightarrow x^{-1} = a_n$

Beispiel: $x = 671, M_p = 2^{13} - 1 = 8191$

e	a	a_n	y	y_n	e	a	a_n	y	y_n
0	1	0	671	8191	0	1	1	8862	671
1	4097	4096	5102	4431	1	10240	6144	6982	2551
1	11264	5120	6042	3491	1	10752	5632	6512	3021
4	6304	672	3428	407	2	2248	1576	1264	857
4	5812	4236	936	79	3	9058	4822	196	117
2	11182	6360	166	49	1	11951	5591	132	83
2	6531	940	116	33	2	8716	7776	62	29
1	12134	4358	60	31	2	11487	7129	46	15
1	8777	1648	38	23	1	1941	293	42	19
1	5359	5066	40	21	3	12903	7837	26	5
1	10193	2356	18	13	1	3357	1001	22	9
1	6775	5774	20	11	2	13611	7837	16	5
4	14319	6482	6	1					

$\Rightarrow x^{-1} = 6482$, $6482 \cdot 671 \equiv 1 \bmod (2^{13} - 1)$

Satz 3.35 *Die Anzahl der Durchäufe der while-Schleife ist $O(p)$. Es werden nur Addition und Shift-Operationen $O(\log_2 M_p)$ verwendet*
$\Rightarrow$ der Gesamtaufwand beträgt $O(p^2)$.

Beweis:
Heuristischer Ansatz: $y_k = \frac{y_{k-1}+y_{k-2}}{2^{t_k}}$, $\forall k : t_k \geq 1$, die Wahrscheinlichkeit für $\mathbb{P}(t_k == i) = \frac{\#\, 2^i | y_k}{\#\, 2 | y_k} = \frac{2^{p-1-i}}{2^{p-1}} = 2^{-i}$. Für $y_k = \frac{y_{k-1}+y_{k-2}}{2^{t_k}}$ gilt:

- $t_k = 1 : y_k = \frac{y_{k-1}+y_{k-2}}{2} \Rightarrow y_{k-1} \leq y_k \leq y_{k-2}$, dann existiert eine Folge $t_{k+1} \ldots t_{k+i-1} = 1$ und $t_{k+i} \geq 2 \Rightarrow y_{k+i} = \frac{y_{k+i-1}+y_{k+i-2}}{2^{t_{k+i}}} \leq \frac{y_{k-2}+y_{k-2}}{2^{t_{k+i}}} \leq \frac{y_{k-2}}{2}$
Der Erwartungswert für die Länge i dieser Folge ist $\sum_{i=1} \frac{i}{2^i} = 2$.
- $t_k \geq 2 : y_k = \frac{y_{k-1}+y_{k-2}}{2^{t_k}} \leq \frac{y_{k-2}}{2}$

d.h. es existiert eine Teilfolge y_j von y_k mit $y_{j+1} \leq \frac{y_j}{2}$ und $j = k-2, j+1 = k+i \vee k$. Die Länge dieser Teilfolge ist mit $\log_2 M_p = p$ beschränkt. Die Länge der gesamten Folge y_k beträgt somit $c \cdot p =$ Anzahl der Durchäufe.

3.8 Zusammenfassung

Die vorgestellten Argorithmen und Methoden basieren auf dem Prinzip des Euklidischen Algorithmus. Es erfolgt eine Reduktion des ggT der Zahlen x, y, auf x', y', mit $x', y' < x, y$.
Der Euklidische Algorithmus beinhaltet das Basismodell einer solchen Reduktion ($x \bmod y$). Allerdings ist diese Reduktion simpel und für größere Zahlen zu langsam. Darum wurden Verbesserungen von Euklid bzw. neue Ansätze entwickelt um die Effizienz zu steigern.

1. Basiswechsel
   Die Binärvariante von Euklid ist ein Beispiel für die Transformation eines Algorithmus auf dem Computer um eine bessere Leistung zu erzielen. Der Ansatz der k-ären Methode ist ein Basiswechsel von 2 auf k um die Operationen zu optimieren.

2. Optimierung der Reduktion (Lehmer)
   Die Idee von Lehmer ist die Optimierung einer Division mittels einer Näherung, dadurch wird der Aufwand der Operation reduziert und somit erfolgt eine Effizienzsteigerung.

3. spezielle Anwendung
   Der rekusive ggT ist ein spezieller Algorithmus für sehr große Zahlen d.h. die Operationen und Methoden sind für kleine Zahlen ungeeignet.

Für viele dieser Methoden existieren Variationen und Verbesserungen für spezielle Probleme, z.B. Inversionen.

4 Multiplikation

Im folgenden Kapitel werden Techniken für das schnelle Multiplizieren von großen Zahlen vorgestellt.

4.1 Potenzen in $(\mathbb{Z}, +)$

Die Idee besteht darin die Multiplikation durch Additionen zu ersetzen.

Satz 4.1 *Die Potenz x^y in $(\mathbb{Z}, \star)$ wird durch wiederholtes Ausführen der Operation $\star$ erzeugt, $\underbrace{x \star x \star \ldots \star x}_{y}$.*

- $(\mathbb{Z}, \cdot) \Rightarrow x^y = \prod_y x$
- $(\mathbb{Z}, +) \Rightarrow x^y = \sum_y x$

d.h. die Multiplikation entspricht der Potenz in $(\mathbb{Z}, +)$.

Der Algorithmus „Square and Multiply“ berechnet die Potenz x^y in $(\mathbb{Z}, \cdot)$ indem die Bits des Exponenten y von links nach rechts durchlaufen werden.

$$x^y = \sum_{j=0}^{D-1} x^{2^j y_j} = ((\ldots((x^{y_{D-1}})^2 \cdot x^{y_{D-2}})^2 \cdots x^{y_1})^2 \cdot x^{y_0} \tag{36}$$

In $(\mathbb{Z}, +)$ gilt nun :

$$x \cdot y = \sum_{j=0}^{D-1} x \cdot 2^j \cdot y_j = ((\ldots((x \cdot y_{D-1}) \cdot 2 + x \cdot y_{D-2}) \cdot 2 + \ldots x \cdot y_1) \cdot 2 + x \cdot y_0 \tag{37}$$

Algorithmus 4.2

```
PowAdd(x,y)  {
    s = 0                                   Initialisierung
    for (D − 1 ≥ i > 0)                     beginnend mit dem höchsten Bit
        s = s << 1
        if (y_i == 0)        s = s + y
    return s }
```

Die Multiplikation modulo in $(\mathbb{Z}, +)$ ist die binäre mul-mod Funktion, Algorithmus 1.13.

4.2 Karatsuba und Tom-Cook Methode

4.2.1 Karatsuba-Methode

Die Karatsuba - Methode basiert auf der $M = 2^m$ Darstellung zweier Zahlen

$$x = x_0 + x_1 M \quad x_0, x_1 \in [0, M-1] \qquad y = y_0 + y_1 M \quad y_0, y_1 \in [0, M-1]$$

und somit lautet das Produkt $xy = x_0 y_0 + (x_1 y_0 + x_0 y_1)M + x_1 y_1 M^2$ bzw.

$$xy = t_0(M+1) + t_1 M + t_2(M^2 + M)$$

mit

$$t_0 = x_0 y_0$$
$$t_1 = (x_1 - x_0)(y_0 - y_1)$$
$$t_2 = x_1 y_1$$

d.h. die Multiplikation zweier Zahlen der Größe M^2 kann durch 3 Multiplikationen von Zahlen der Größe M und einfachen binären Schiebeoperationen sowie Additionen ersetzt werden.
Für $L(n)$ = Laufzeit einer Multiplikation 2er Zahlen mit n Bits gilt:

$$L(2n) \leq 3L(n) + cn$$

Die Karatsuba-Methode besteht nun im rekursiven Aufruf dieser Aufteilung, d.h. x, y mit n Bits $\Rightarrow xy = t_0(M+1)+t_1 M+t_2(M^2+M)$, für die Berechung der t_i Aufruf von Karatsuba mit $\frac{n}{2}$ Bits, usw.

Satz 4.3 *Für die Multiplikation 2er Zahlen mit n Bits ist der Rechenaufwand mit $O(n^{\lg 3}) = O(n^{1.585})$ beschränkt.*

Algorithmus 4.4 *bei $x, y <$ Grenze $G = 2^g$ Abbruch der Rekursion*

Karatsuba(x, y) {
  if $x < G \wedge y < G$ return xy
  else
    Halbieren von x und y
    $M = 2^{\max(x,y).Length/2}$, $\quad x = x_1 M + x_0, \quad y = y_1 M + y_0$
    3 maliger rekursiver Aufruf
    return Karatsuba$(x_0 y_0) \cdot (M+1)$+Karatsuba
      $((x_1 - x_0)(y_0 - y_1))M$+ Karatsuba$(x_1 y_1) \cdot (M^2 + M)$

Die Karatsuba-Methode ist eine „theoretische Verbesserung“ gebenüber der Schulmethode, es ergeben sich Probleme bei der Implementation (Overhead, Speichermanagement für die rekursiven Aufrufe und die Rekombination der Subprodukte).

4.2.2 Tom-Cook Methode

Die Tom-Cook Methode basiert auf der Idee, dass für 2 Polynome

$$x(t) = x_0 + x_1 t + \ldots + x_{D-1} t^{D-1} \qquad y(t) = y_0 + y_1 t + \ldots + y_{D-1} t^{D-1}$$

das Polynomprodukt $z(t) = x(t) \cdot y(t)$ nur mittels $2D-1$ verschiedenen Werten für $j \in M$ (z.B. $[1-D, D-1]$) berechnet wird.

Symbolische Tom-Cook Methode:

1. 2 symbolische Polynome $x(t), y(t)$ mit den Grad $D-1$ und M ist eine Menge mit $2D-1$ verschiedenen Werten
2. Berechnung von $z(j) = x(j) \cdot y(j)$ für jedes $j \in M$, umwandeln von z_j in Terme von Koeffizienten der Polynome $x(t), y(t)$
3. Berechnung der Koeffizienten z_j durch Lösen des linearen Gleichungssystems von $2D-1$ Gleichungen: $z(t) = \sum_{k=0}^{2D-2} z_k t^k \quad t \in M$
4. Liste von $2D-1$ Relationen, wobei jede Relation z_j in Terme der Koeffizienten von x, y umwandelt

Das Resultat ist eine Menge von Formeln, welche die Koeffzienten von $z(t) = x(t) \cdot y(t)$ mittels Termen von Koeffizienten von $x(t), y(t)$ angibt.

Satz 4.5 *Durch rekursive Tom-Cook Aufruf, Punkt 2, kann der Aufwand für die Multiplikation 2er Zahlen mit Größe N mit $O((\ln N)^{\frac{\ln(2D-1)}{\ln D}})$ „kleinen" Multiplikationen (fixe Größe) abgeschätzt werden. Falls der Grad $d = D-1$ genügend groß ist, gilt für den Aufwand*

$$O(\ln^{1+\epsilon} N) \text{ mit } \epsilon > 0$$

(ohne Additionen und Multiplikationen mit einer Konstanten).

Implementierung für $B = 2^{32}$:

1. 2 Zahlen $x = x_0 + x_1 B + \ldots + x_{D_x-1} B^{D_x-1}$, $y = y_0 + y_1 B + \ldots + y_{D_y-1} B^{D_y-1}$ mit den Graden $D_x - 1, D_y - 1, D = \max(D_x, D_y)$ und $M = 0 \ldots 2D-1$
2. Berechnung von $z(j) = x(j) \cdot y(j)$ für jedes $j \in M$,
3. Differenzenschema für $z(j)$ und Speichern der Koeffizienten z_j für faktorielle Potenzen $B^{\underline{k}} = B \cdot (B-1) \cdots (B-k+1)$
4. Umwandeln von faktoriellen Potenzen $B^{\underline{k}}$ auf Potenzen B^k
5. $z = z_0 + z_1 B + \ldots + z_{D-1} B^{D-1} = x \cdot y$

Differenzenschema:
Gesucht: $z(x) = a_{2D-1}x^{\underline{2D-1}} + \ldots + a_1 x^{\underline{1}} + a_0$ in der Form von faktoriellen Potenzen $x^{\underline{k}} = x \cdot (x-1) \cdots (x-k+1)$
Die faktoriellen Potenzen besitzen die Eigenschaft, dass

$$\Delta x^{\underline{k}} = (x+1)^{\underline{k}} - x^{\underline{k}} = k x^{\underline{k-1}} \quad k = 0, 1, 2 \ldots$$

d.h. durch Bildung dieser Differenzen erhält man einen Term mit um 1 niedrigeren Grad.

$$\Delta z(x) = z(x+1) - z(x) = (2D-1)a_{2D-2}x^{\underline{2D-2}} + \ldots + 2a_2 x^{\underline{1}} + a_1$$

$$\Delta^2 z(x) = \Delta z(x+1) - \Delta z(x) = (2D-1)(2D-2)a_{2D-3}x^{\underline{2D-2}} + \ldots + 6a_3 x^{\underline{1}} + 2a_2$$

$$\Rightarrow a_k = \frac{1}{k!}\Delta^k z(0) \qquad k = 0, 1, 2 \ldots$$

Daraus folgt nachfolgendes **Differenzenschema** :

$$\begin{array}{rcl} a_{0j} = a_j \quad \forall j & & z_0 = a_{00} \\ a_{1j} = a_{0(j+1)} - a_{0j} \quad \forall j & & z_1 = a_{01} \\ a_{2j} = \frac{1}{2}(a_{1(j+1)} - a_{1j}) \quad \forall j & & z_2 = a_{02} \\ a_{3j} = \frac{1}{3}(a_{2(j+1)} - a_{2j}) \quad \forall j & & z_3 = a_{03} \\ \vdots & & \vdots \end{array}$$

Umwandeln von faktoriellen Potenzen, rekursiver Ansatz:
$x^{\underline{k}} = x \cdot (x-1) \cdots (x-k+1) = x^{\underline{k-1}} \cdot (x-k+1)$

Algorithmus 4.6 *liefert einen Vektor v_k mit $a_k x^{\underline{k}} = v[0] + v[1]x^1 + \ldots + v[k]x^k$, berechnet aus v_{k-1}*

FactorialPow(v_{old}, k) {
  if $(k == 0)$ return $[1]$
  if $(k == 1)$ return $[1, 0]$
  for $(1 \le i < Length(v_{old}))$
    $v_{new}[i] = v_{old}[i] \cdot (-k+1) + v_{old}[i-1]$
  return v_{new} }

Algorithmus 4.7 **($B = 2^{32}$)**

TomCook(x,y) {
  $D = \max(x.Count, y.Count)$ D-Stellen
  Auswerten an $2D-1$ Stellen : $0, 1, 2, \ldots 2D-1$
  $xt[0] = x[0], \quad yt[0] = y[0]$
  for $(1 \le i < 2D-1)$
    for $(0 \le j < D) \quad xt[i] = xt[i] + x[j] \cdot i^j$, $yt[i] = yt[i] + y[j] \cdot i^j$
  for $(0 \le i < 2D-1) \qquad zt[i] = xt[i] \cdot yt[i]$ Berechnung der z(t)
  Differenzenschema für Koeffizienten von faktorielle Potenzen

for $(0 \le i, k < zt.Count)$ $\quad zt[j] = \frac{1}{k}(zt[j] - zt[j-1])$
Berechnung von faktoriellen Potenzen zu B^i
for $(0 \le i < 2D-1)$
  $v =$FactorialPow(v, i)
  for $(0 \le j < v.Count)$ $\quad z[j] = z[j] + v[j] \cdot zt[i]$
return $[z[0], z[1], \ldots z[2D-1]]$ } Stellen von xy

4.3 Schnelle Fouriertransformationen (FFT)

Die vorigen Ergebnisse (Karatsuba, Tom-Cook) verringerten den Aufwand auf $O(\ln^{1+\epsilon} N)$ „kleine“ Multiplikationen, ohne die Anzahl der Additionen zu berücksichtigen.
Gesucht : Multiplikationen mit niedrigen Aufwand für alle Operationen
Solche Methoden basieren auf der diskreten Fourier Transformation (DFT) und der Repräsentation einer Zahl als Signal.

4.3.1 Algebraischer Ansatz

Seien $p(t) = \sum_{i=0}^{m} x_i t^i$ und $q(t) = \sum_{i=0}^{m} y_i t^i$ zwei Polynome, dann gilt:

$$p(t)q(t) = \sum_{k=0}^{2m} z_k t^k, \qquad z_k = \sum_{i=0}^{k} x_i y_{k-i}$$

Ein Polynom vom Grad $< n$ ist eindeutig durch seine Werte an n verschiedenen Stellen $\xi_0, \xi_1, \ldots, \xi_{n-1}$ bestimmt. Wegen $pq(\xi_k) = p(\xi_k)q(\xi_k)$ gilt für die Polynommultiplikation mit $n = 2m+1$ folgendes Schema :

1. Transformation :
   $(x_0, \ldots, x_m)(y_0, \ldots, y_m) \to (p(\xi_0), \ldots, p(\xi_{2m}))(q(\xi_0), \ldots, q(\xi_{2m}))$
2. punktweise Multiplikation :
   $(p(\xi_0), \ldots, p(\xi_{2m}))(q(\xi_0), \ldots, q(\xi_{2m})) \to (pq(\xi_0), \ldots, pq(\xi_{2m}))$
3. inverse Transformation : $(pq(\xi_0), \ldots, pq(\xi_{2m})) \to (z_0, \ldots, z_{2m})$

Die Punkte 1,2,3 entsprechen einer Polynommultiplikation von pq.

Untersuchung der Eigenschaften der Stützstellen:
Sei $p(t) = \sum_{i=0}^{n-1} x_i t^i$ und OBdA. $n = 2^r$, $r \in \mathbb{N}$:

$$p(t) = x_0 + x_1 t + x_2 t^2 + \ldots + x_{n-1} t^{n-1} =$$

$$= (x_0 + x_2 t^2 + \ldots + x_{n-2} t^{n-2}) + t(x_1 + x_3 t^2 + \ldots + x_{n-1} t^{n-1})$$

Durch Substitution von u für x^2 erhält man

$$= (x_0 + x_2 u + \ldots + x_{n-2} u^{\frac{n-2}{2}}) + t(x_1 + x_3 u + \ldots + x_{n-1} u^{\frac{n-2}{2}})$$

$= p_1(u) + tp_2(u)$, wobei der Grad(p_1)=Grad(p_2) $= \frac{n-2}{2} = 2^{r-1} - 1$.
$\Rightarrow$ Die Auswertung von p an den Stützstellen $\xi_0, \xi_1, \ldots, \xi_{n-1}$ kann durchgeführt werden, indem man p_1 und p_2 an den Stützstellen $\varphi_0, \varphi_1, \ldots \varphi_{n-1}$ mit $\varphi_i = \xi_i^2$ auswertet und dann jeweils eine Multiplikation und eine Addition ausführt.
Gesucht: Stützstellen mit der Eigenschaft, dass das jeweilige Quadrieren aller Stützstellen die Anzahl der verschiedenen Stützstellen halbiert.

Im folgenden bezeichent U einen kommutativen Ring mit Einselement.

Definition 4.8 *$\omega \in U$ heißt eine n-te Einheitswurzel ($n > 0$), wenn $\omega^n = 1$, und eine primitive n-te Einheitswurzel, wenn darüberhinaus nocht gilt, dass $n = 1 + 1 + \ldots + 1$ (n−mal) eine Einheit in U und $\omega^{\frac{n}{p}}$ für keinen Primteiler p von n ein Nullteiler von U ist.*

Ist U nullteilerfrei, z.B. ein Körper, so vereinfacht sich die Bedingung für eine primitive n-te Einheitswurzel und zwar, dass $\omega^n = 1$, aber $\omega^{\frac{n}{p}} \neq 1$ für jeden Primteiler p von n gelten muß($\Leftrightarrow \omega^k \neq 1$, $0 < k < n$).
Falls ω eine $2n$-te Einheitswurzel ist, dann ist ω^2 eine $n - te$ Einheitswurzel und damit halbiert sich, durch das Quadrieren aller Stützstellen ω^i, $0 \leq i < 2k$, die Anzahl der verschiedenen Stützstellen $\Rightarrow$ Stützstellenmenge ω^i besitzt die gewünschte Eigenschaft.

Im folgenden wird der allgemeine Begriff des Signals verwendet.

Definition 4.9 *Ein Signal x ist eine Folge von Elementen, $x = (x_n), n \in [0, D-1]$, mit Signallänge D und $x_n \in U$.*

Sätze von Signale sind transformationsunabhängig, d.h. die Entstehung des Signals ist nicht von Bedeutung.

$\Rightarrow$ Multiplikation 2er Zahlen a, b:

1. Zahlen a, b entsprechen Signalen x, y (Signalumwandlung $a \rightarrow_S x$)
2. Transformation des Signals x auf Signal X: $x \rightarrow_T X, y \rightarrow_T Y$
3. punktweise Multiplikation der transformierten Signale : $Z = X \cdot Y$
4. inverse Transformation auf Signal z : $Z \rightarrow_{T^{-1}} z$
5. Signal z enspricht Zahl c ($z \rightarrow_{S^{-1}} c$)

Die Signalumwandlung ist eine einfache Darstellung der Zahlen, z.B. Basisdarstellung $a = \sum_i x_i B^i$.

4.3.2 Diskrete Fourier Transformation (DFT)

Gegeben: Signal $x = (x_0, x_1, \dots x_{D-1})$ mit Länge D mit den Stützstellen $\xi_i = \omega^i$, ω eine D-te Einheitswurzel
Die Auswertungsabbildung lautet

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{D-1} \end{pmatrix} \rightarrow X = \begin{pmatrix} x_0 + x_1 + \ldots + x_{D-1} \\ x_0 + x_1\omega + \ldots + x_{D-1}\omega^{n-1} \\ x_0 + x_1\omega^2 + \ldots + x_{D-1}\omega^{2(n-1)} \\ \vdots \\ x_0 + x_1\omega^{n-1} + \ldots + x_{D-1}\omega^{(n-1)^2} \end{pmatrix} = DFT(x)$$

X Signal der Länge D bzw. in Matrizenschreibweise gilt $X = DFT_D \cdot x$,

$$DFT_D = (\omega^{ab})_{0 \leq a,b < D} = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)^2} \end{pmatrix}$$

die Diskrete-Fourier-Transformationsmatrix DFT_D der Länge D.

Definition 4.10 *x ein Signal der Länge D bestehend aus Elementen $\in U$, $D^{-1} \in U$ und ω eine primitive D-te Einheitswurzel von U. Die diskrete Fourier Transformation von x ist das Signal $X = DFT(x)$ mit den Elementen*

$$X_k = \sum_{j=0}^{D-1} x_j \omega^{-jk} \tag{38}$$

und die inverse $DFT^{-1}(X) = x$ lautet

$$x_j = \frac{1}{D} \sum_{k=0}^{D-1} X_j \omega^{jk}. \tag{39}$$

Beispiel: DFT in komplexe Körper, ω eine D-te Einheitswurzel $= \exp^{2\pi i/D}$

$$DFT_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

Neben der DFT existieren eine Menge von verschiedenen Transformationen, viele davon basieren auf Basisfunktionen und ermöglichen schnelle Algorithmen für reelle Signale.

Beispiele für Tranformationen:

- Walsh-Hadamard Transformation:
  verwendet keine Multiplikationen, nur Additionen
- Diskrete Cosinus Transformation (DCT) analog zu DFT
- verschiedene „wavelet“ Transformationen :
  ermöglichen sehr schnelle Algorithmen mit Laufzeit $O(N)$

4.3.3 Multiplikation mittels FFT

Gegeben: $a = \sum_{i=0} x_i B^i, b = \sum_{i=0} y_i B^i$ Zahlen $\in U$
Gesucht: $c = \sum_{i=0} z_i B^i$

Schema der Multiplikation mittels FFT:

$$\begin{array}{ccc} a, b & Multiplikation & c = a \cdot b \\ (x_0, x_1, \ldots, x_{D-1})(y_0, y_1, \ldots, y_{D-1}) & - - - - \rightarrow & (z_0, z_1, \ldots, z_{D-1}) \\ \Big| & & \uparrow \\ DFT \quad \Big| & & \Big| \quad DFT^{-1} \\ \downarrow & & \Big| \\ DFT(x) \cdot DFT(y) & - - - - \rightarrow & DFT(z) \\ & punktweise & \\ & Multiplikation & \end{array}$$

Algorithmus 4.11 (Grundlegende FFT Multiplikation)
a, b ganzzahlige Werte > 0 mit D Stellen bzgl. der Basis B und liefert $a \cdot b$ in B-Darstellung zurück

MultFFT(a, b) {
  Signale x, y sind die D Stellen von a, b
  Anhängen von 0en an x, y bis zur Länge $2D \Rightarrow \hat{x}, \hat{y}$ mit Länge $2D$
  $X = DFT(\hat{x}), Y = DFT(\hat{y})$ DFT-Transformation
  $Z = X \star Y$ elementweise Multiplikation
  $z = DFT^{-1}(Z)$ inverse DFT-Transformation
  $z = round(z)$ elementweises Runden zur nächsten ganzzahligen Zahl
  $carry = 0$ Übertrag
  for $(0 \leq n < 2D)$ Umwandeln auf B-Darstellung
    $v = z_n + carry \quad z_n = v \bmod B \quad carry = \lfloor \frac{v}{B} \rfloor$
  $z = reduce(z)$ Löschen der führenden Nullen
  return z }

Der Algorithmus (4.11) beschreibt das allgemeine Prinzip der FFT - Multiplikation mit den Transformationen, elementweisen Multiplizieren, Runden

und dem Umwandeln auf Basis B (=Rücktransformation). Eine große Fehlerquelle in (4.11) ist die die Genauigkeit der Gleitpunktarithmetik. Das Runden kann bei zu großen Gleitpunktfehlern zu einem falschen Wert von z_n führen.

4.3.4 Faltungen

Eine Zahl x, mit der Basisdarstellung $x = \sum_{i=0}^{D} x_i B^i$, kann als ein Signal mit $(x_0, x_1, \ldots x_{D-1})$ Werten aufgefasst werden.

Im folgenden wird die allgemeine Signalverarbeitung, ihre Methoden und Operationen betrachtet.

Definition 4.12 *Für 2 Signale x, y mit der Länge D gilt:*

$z = x + y$	*Summe 2er Signale*	$z_i = x_i + y_i \quad \forall i < D$
$z = q \star x$	*Produkt 2er Signale*	$z_i = x_i y_i \quad \forall i < D$
$z = q \cdot x$	*Produkt: Skalar · Signal*	$z_i = q x_i \quad \forall i < D$
$z = q \times x$	*„Faltung" 2er Signale*	$z_i = f(x, y) \quad \forall i < D$, *$f$ Funktion*

Für verschiedene Funktionen f existieren verschiedene Faltungen.

Definition 4.13 (Grundlegende Faltungen)
für 2 Signale x, y mit der Länge D, $(i, j = 0 \ldots D-1)$:
zyklische Faltung : $z = x \times y$

$$z_n = \sum_{i+j \equiv n \pmod D} x_i y_j \qquad n = 0 \ldots D-1$$

negazyklische Faltung : $v = x \times_- y$

$$v_n = \sum_{i+j=n} x_i y_j - \sum_{i+j=D+n} x_i y_j \qquad n = 0 \ldots D-1$$

azyklische Faltung : $u = x \times_A y$ und $u_{2D-1} = 0$

$$u_n = \sum_{i+j=n} x_i y_j \qquad n = 0 \ldots 2D-2$$

halbzyklische Faltung : $t = x \times_H y$ sind die ersten D Elemente der azyklischen Faltung $u = x \times_A y$ d.h. $t = (u_0, u_1, \ldots u_{D-1})$

Die Operation $x \cup y$ hängt das Signal y an x (Links-Rechts Reihenfolge) d.h. $x \cup y = (x_0, x_1, \ldots x_{D_x-1}, y_0, y_1, \ldots y_{D_y-1})$.
Für das Teilen eines Signals x in 2 Hälften, (Länge von x gerade $= 2D$), gilt die Notation $L(x) = (x_0, x_1, \ldots x_{D-1})$ für die erste Hälfte und $H(x) = (x_D, x_{D+1}, \ldots x_{2D-1})$ für die zweite Hälfte. Daraus ergibt sich die Identität $x = L(x) \cup H(x)$.

Mittels dieser Operationen und Definitionen gelten folgende Identitäten.

Satz 4.14 *x, y Signale mit derselben Länge D und es existert 2^{-1} für die Elemente der Signale* $\Rightarrow$

$$x \times_H y = \frac{1}{2}((x \times y) + (x \times_- y))$$

$$x \times_A y = (x \times_H y) \cup \frac{1}{2}((x \times y) - (x \times_- y))$$

Sollte D gerade und $x_j, y_j = 0$ für $j \geq \frac{D}{2}$, dann gilt

$$L(x) \times_A L(y) = x \times y = x \times_- y.$$

Diese Identiäten können genutzt werden um Algorithmen so zu modifzieren, damit eine andere Faltung berechnet werden kann. Eine häufig verwendete Methode ist das „zero-padding“, das Anhängen von D Nullen an ein Signal mit Länge D, um die 3. Identität von (4.14) anzuwenden.

Satz 4.15 *Signale x, y mit Länge D. Die zyklische Faltung von x, y lautet*

$$x \times y = DFT^{-1}(DFT(x) \star DFT(y)) \tag{40}$$

$$(x \times y)_n = \frac{1}{D} \sum_{k=0}^{D-1} X_k Y_k \omega^{kn} \qquad X = DFT(x), Y = DFT(y) \tag{41}$$

Eine FFT-Methode für große Zahlenarithmetik ist die Verwendung von DFTs aus (4.15) um die Multiplikation zweier Zahlen mit azyklischer Faltung zu berechnen. Die Idee (Schönhage, Strassen) funktioniert, wie folgt:
x, y ganzzahlige Werte repräsentiert durch Signale mit der Länge D mit den Stellen (B-Darstellung) von x, y, dann ist das Produkt xy eine azyklische Faltung der Länge $2D$ der „zero-padded“ Signale X, Y.

4.3.5 Inverse FFT und reelle Signale

Eine Berechnung der inversen FFT (FFT^{-1}) kann durch das Ersetzen der Einheitswurzel ω durch ω^{-1} und der Normalisierung durch $\frac{1}{D}$ (siehe Definition (4.10) erfolgen. Für komplexe Körper gilt $\Rightarrow \omega^{-1} = \overline{\omega}$.
D.h. die FFT^{-1} kann mittels

$x = \overline{x}$	Konjugieren des Signals
$X = FFT(x)$	FFT-Methode mit Einheitswurzel ω
$X = \overline{X}/D$	Konjugieren und normalisieren

berechnet werden $\Rightarrow$ die FFT^{-1} wird durch FFT berechnet.

Definition 4.16 *$x \in K$, (komplexes Signal $x \in \mathbb{C}^D$ bzw. $x \in \mathbb{F}_{p^2}^D$)*
Ein Signal x ist „pure real“, wenn das Signal nur aus reellen Elementen besteht d.h. $x_j = a_j + b_j i$, $\forall j \in [0, D-1]$ gilt $b_j = 0$.

Pure-real Signale besitzen nur die Hälfte der Werte eines vergleichbaren komplexen Signals. Daraus ergibt sich eine Methode zur Verringerung des Aufwandes für pure-real Signale.
Allgemeine Methode für die FFT eines pure-real Signals:

- Transformation eines pure-real Signals x der Länge D auf ein Signal y der Länge $\frac{D}{2}$, $\quad x \rightarrow_T y$
  Beispiel: x auf komplexes Signal y mit $y_j = x_j + ix_{j+\frac{D}{2}}$ abbilden
- FFT-Methode für y ausführen $\quad Y = FFT(y)$
- Rücktransformation des FFT-Resultat Y auf das korrekte FFT-Signal von $x \quad Y \rightarrow_{T^{-1}} X$

$\Rightarrow$ Aufwand für $FFT(x)$ halbiert (z.B. zyklische Faltung)

4.4 Radix 2 FFT-Algorithmen

Definition 4.17
Radix 2 FFTs bezeichnen Algorithmen mit der Länge $D = 2^d$.

4.4.1 Rekursiver FFT-Algorithmus

Der rekursive FFT-Algorithmus basiert auf folgendem Lemma:

Lemma 4.18 (Danielson-Lanczos) *Eine diskrete Fourier Transformation der Länge $n = 2m$ kann als Summe zweier diskreten Fourier Transformationen der Länge m, wobei die eine aus den Elementen mit geraden Index und die andere aus den Elementen mit ungeraden Index des Originalproblems gebildet werden. $DFT(x) = X = (X_0, X_1, \dots X_n)$:*

$$X_k = \sum_{i=0}^{n-1} x_i \omega^{-ik} = \sum_{i=2j}^{2m-1} x_i \omega^{-ik} + \sum_{i=2j+1}^{2m-1} x_i \omega^{-ik}$$

$$X_k = \sum_{j=0}^{n-1} x_j \omega^{-jk} = \sum_{j=0}^{m-1} x_{2j} (\omega^2)^{-jk} + \omega^{-k} \sum_{j=0}^{m-1} x_{2j+1} (\omega^2)^{-jk} \tag{42}$$

Eine DFT-Summe mit Länge D kann in 2 Summen mit Länge $\frac{D}{2}$ aufgespalten werden $\Rightarrow$ Lemma (4.18) ermöglicht eine rekursive Berechnung der Transformation DFT.
Für den Algorithmus (4.19) wird die Operation *len*, welche die Signallänge bestimmt, benötigt. $(a_j)_{j \in J}$ bezeichnet ein Signal, welches durch sukzessive Anhängen (Links-Rechts Reihenfolge) der a_j entsteht, wobei j die Menge J durchläuft.

Algorithmus 4.19 *berechnet die DFT eines Signals x mit $len(x) = D = 2^d$ und der D-ten Einheitswurzel ω*

FFTRecursiv(x) {
  $n = len(x)$
  if $(n == 1)$ return x — 1 Element
  $m = \frac{n}{2}$
  $X = (x_{2j})_{j=0}^{m-1}$ — gerader Teil von x
  $Y = (x_{2j+1})_{j=0}^{m-1}$ — ungerader Teil von x
  $X =$FFTRecursiv(x), $Y =$FFTRecursiv(x) — Rekursion
  $U = (X_{k \bmod m})_{k=0}^{n-1}$
  $V = (\omega^{-k} Y_{k \bmod m})_{k=0}^{n-1}$ — Multiplikation
  return $U + V$ } — Lemma (4.18) (Addition)

Der Aufwand C_D für $D = 2^d$ setzt sich aus 2 DFTs der Länge 2^{d-1} sowie einer Multiplikation und einer Addition von Signalen der Länge 2^{d-1} zusammen $\Rightarrow$

$$C_{2^r} = \underbrace{2C_{2^{r-1}}}_{Rekursionen} + \underbrace{2 \cdot 2^r}_{Addition+Multiplikation}$$

und da der Aufwand für ein Signal der Länge 1, $C_1 = 0$ beträgt, lautet die Lösung der Rekursion

$$C_{2^r} = r \cdot 2 \cdot 2^r$$

Die Anzahl der Operationen für ein Signal der Länge D von (4.19) beträgt

$$O(D \ln D),$$

diese Abschätzung gilt für Multiplikationen sowie für Additionen und Subtraktionen. Der rekursive FFT-Algorithmus (4.19) gehört zur Gruppe der „Divide and Conquer"-Algorithmen.

4.4.2 Cooley-Tukey Variante

Die Grundlage der Cooley-Tukey Variante der FFT ist Lemma (4.18), allerdings erfolgt **kein** rekursiver Aufruf, d.h. nur Schleifen.

Sei $n = 2m$ und ω die n-te primitive Einheitswurzel.

$$\begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_{D-1} \end{pmatrix} = DFT_n \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{D-1} \end{pmatrix}$$

Im folgenden bezeichnet D_n die DFT-Matrix mit n Zeilen und Spalten, d.h.

$$D_n \cdot x = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)^2} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

Durch die Anwendung des Danielson-Lanczos-Lemma (4.18) gilt:

$$D_n \cdot x = \left(\begin{array}{c|c} D_m & \Delta_m \\ \hline D_m & -\Delta_m \end{array}\right) \cdot \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{n-2} \\ \hline x_1 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix} = \left(\begin{array}{c|c} D_m & \Delta_m \\ \hline D_m & -\Delta_m \end{array}\right) \cdot \left(\begin{array}{c} x_{2j} \\ \hline x_{2j+1} \end{array}\right)_{j=0\dots m-1}$$

mit $\Delta_m = diag(1, \omega, \omega^2, \dots, \omega^{m-1})$, ($\omega^l = -1$ „Twiddle - Faktoren“) und x_i Folge der durch i indizierten Elemente von x. Mittels der Matrixschreibweise (I_m Identitätsmatrix mit m Elementen) gilt folgende Gleichung :

$$D_n \cdot x = \begin{pmatrix} I_m & \Delta_m \\ I_m & -\Delta_m \end{pmatrix} \cdot \begin{pmatrix} D_m & 0 \\ 0 & D_m \end{pmatrix} \cdot \left(\begin{array}{c} x_{2j} \\ \hline x_{2j+1} \end{array}\right)_{j=0\dots m-1} \tag{43}$$

Für m gerade gilt nun:

$$D_n \cdot x = \begin{pmatrix} I_m & \Delta_m \\ I_m & -\Delta_m \end{pmatrix} \cdot \begin{pmatrix} D_m & 0 \\ 0 & D_m \end{pmatrix} \cdot \left(\begin{array}{c} x_{2j} \\ \hline x_{2j+1} \end{array}\right)_{j=0\dots m-1} =$$

$$\underbrace{\begin{pmatrix} I_m & \Delta_m \\ I_m & -\Delta_m \end{pmatrix}}_{A_n} \cdot \left(\begin{array}{cc} \begin{array}{c|c} D_{\frac{m}{2}} & \Delta_{\frac{m}{2}} \\ \hline D_{\frac{m}{2}} & -\Delta_{\frac{m}{2}} \end{array} & 0 \\ 0 & \begin{array}{c|c} D_{\frac{m}{2}} & \Delta_{\frac{m}{2}} \\ \hline D_{\frac{m}{2}} & -\Delta_{\frac{m}{2}} \end{array} \end{array}\right) \cdot \left(\begin{array}{c} x_{4j} \\ \hline x_{4j+2} \\ \hline x_{4j+1} \\ \hline x_{4j+3} \end{array}\right)_{j=0\dots\frac{m}{2}-1}$$

$$A_n \cdot \underbrace{\left(\begin{array}{cc} \begin{array}{c|c} I_{\frac{m}{2}} & \Delta_{\frac{m}{2}} \\ \hline I_{\frac{m}{2}} & -\Delta_{\frac{m}{2}} \end{array} & 0 \\ 0 & \begin{array}{c|c} I_{\frac{m}{2}} & \Delta_{\frac{m}{2}} \\ \hline I_{\frac{m}{2}} & -\Delta_{\frac{m}{2}} \end{array} \end{array}\right)}_{A_m} \cdot diag(D_{\frac{m}{2}}) \cdot \left(\begin{array}{c} x_{4j} \\ \hline x_{4j+2} \\ \hline x_{4j+1} \\ \hline x_{4j+3} \end{array}\right)_{j=0\dots\frac{m}{2}-1}$$

Satz 4.20 *Für $D = 2^d$ ist $DFT_n \cdot x = A_d A_{d-1} \cdots A_1 P x$ mit P der Permutationsmatrix für die Elemente von x und der $D \times D$ Matrizen*

$$A_t = diag \begin{pmatrix} I_{\frac{t}{2}} & \Delta_{\frac{t}{2}} \\ I_{\frac{t}{2}} & -\Delta_{\frac{t}{2}} \end{pmatrix}.$$

$\Rightarrow$ Symbolischer Algorithmus von Cooley-Tukey :

$x = Px$

for $(1 \leq i \leq d)$ $\quad x = A_i x$

mit dem Resultat $FFT(x) = DFT \cdot x$.

Das Cooley-Tukey Schema („decimation - in - time“ DIT) erlaubt die Berechnung der FFT „in place“ d.h. das Originalsignal x wird durch die DFT-Werte ersetzt. „decimation - in - time“ bezieht sich auf das Lemma (4.18), wo ein „decimate“ des Index des Originalsignals stattfindet.

Für den Aufwand von DFT_n gelten folgende Abschätzungen:

- 2 Auswertungen von DFT_m :
  $DFT_m \cdot (x_{2j})_{j=0\ldots m-1}$ und $DFT_m \cdot (x_{2j+1})_{j=0\ldots m-1}$
- $\leq n = 2m$ Multiplikationen (wegen Diagonalmatrix) :
  $\Delta_m \cdot (DFT_m \cdot (x_{2j})_{j=0\ldots m-1})$ und $\Delta_m \cdot (DFT_m \cdot (x_{2j+1})_{j=0\ldots m-1})$
- $n = 2m$ Additionen/Subtraktionen:
  $DFT_m \cdot (x_{2j})_{j=0\ldots m-1} \pm \Delta_m DFT_m \cdot (x_{2j+1})_{j=0\ldots m-1}$

Der Aufwand C_s für $D = 2^d$ genügt der Rekursion $C_0 = 0, C_s = 2C_{s-1} + 2 \cdot 2^2$ ohne den Aufwand für die ω^j.

Satz 4.21 *Der Cooley-Tukey Algorithmus wertet die DFT-Matrix an x mit einem Aufwand von $2n \log n$ Elementaroperationen aus.*

Berechung der Permutation P:
Die Permutation der Elemente x_i wird durch eine Permutation der Bits „bit-scrambling“ der Indizes erreicht. Die Transposition eines Index $i \to s(i)$ entspricht einer rückwärtigen binären Indexierung, d.h. Binärdarstellung $i = b_D b_{D-1} \ldots b_1 b_0 \to b_0 b_1 \ldots b_{D-1} b_D$. Für ein Signal $x = (x_0, x_1, \ldots x_D)$ ist Px gleich dem „scrambled“ Signal $x_P = (x_{s(0)}, x_{s(1)}, \ldots x_{s(D)})$. Der folgende Algorithmus (4.22) überschreibt die Elemente x_i eines Signal x mit $x_{s(i)} \Rightarrow$ Nach scramble lautet das Signal $x = (x_{s(0)}, x_{s(1)}, \ldots x_{s(D)})$.

Algorithmus 4.22 *in place, reverse-binary element scrambling*

```
Scramble(x)  {
    n = x.Length    j = 0
    for (0 ≤ i < n − 1)                                  Initialisierung
        if (i < j)        (x_i, x_j) = (x_j, x_i)        Tauschen
        k = ⌊n/2⌋
        while (k ≤ j)
            j = j − k,        k = ⌊k/2⌋
        j = j + k }
```

Algorithmus (4.23) ist eine Implementation des Cooley-Tukey Schemas mit „in order“, d.h. natürliche Reihenfolge der DFT-Werte.

Algorithmus 4.23 (Cooley-Tukey) *in place, in order, bitscramble*

FFTCooleyTukey(x) {
  Scramble(x) „bitscramble“ der Indizes von x
  $n = x.Length$
  for $(m = 1; m < n; m = 2m)$ m läuft über 2-Potenzen
    for $(0 \leq j < m)$
      $a = \omega^{-jn/(2m)}$ ω, n-te Einheitswurzel
      for $(i = j; i < n; i = i + 2m)$
        $(x_i, x_{i+m}) = (x_i + ax_{i+m}, x_i - ax_{i+m})$
  return x }

4.4.3 Gentleman-Sande FFT-Methode

Der Gentleman-Sande Algorithmus gehört zur „decimation in frequency“ Klasse, wo eine Variation der Indizes der transformierten Elemente X_k stattfindet. Der Gentleman-Sande basiert auf den selben Überlegungen und Methoden wie Cooley-Tukey ($D = 2^d$, $DFT_n \cdot x = A_d A_{d-1} \cdots A_1 P x$).

Für die Matrizen gelten folgende Gleichungen:

- $D_n = D_n^T$: $D_n = (\omega^{ab})_{0 \leq a,b < n}$ und Transponieren vertauscht die Elemente $[a, b] \leftrightarrow [b, a]$ d.h. $\omega^{ab} = \omega^{ba}$
- $P = P^T$: „dünn“ besetzte Matrix $P[i, j] = \begin{cases} 1 & \text{falls } j == s(i) \\ 0 & \text{sonst} \end{cases}$
  Transposition $s(i) : s(s(i)) = i \Rightarrow P[i, s(i)] = 1 = P[s(i), i] \Rightarrow P = P^T$

Mittels dieser Umformungen gilt folgende Gleichung :

$$D_n \cdot x = D_n^T \cdot x = (A_d A_{d-1} \cdots A_1 P)^T x = P A_1^T \cdots A_{d-1}^T A_d^T x \qquad (44)$$

$\Rightarrow$ Symbolischer Algorithmus von Gentleman-Sande :
  for $(1 \leq i \leq d)$ $\quad x = A_i^T x$
  $x = Px$

Algorithmus 4.24 (Gentleman-Sande) *in place, in order, bitscramble*

FFTGentlemanSande(x) {
  $n = x.Length$
  for $(m = \frac{n}{2}; m \geq 1; m = \frac{m}{2})$ m läuft über 2-Potenzen
    for $(0 \leq j < m)$
      $a = \omega^{-jn/(2m)}$ ω, n-te Einheitswurzel
      for $(i = j; i < n; i = i + 2m)$
        $(x_i, x_{i+m}) = (x_i + x_{i+m}, a(x_i - x_{i+m}))$
  Scramble(x) „bitscramble“ der Indizes von x
  return x }

Satz 4.25 *Der Aufwand für die Berechnung der FFT für $D = 2^d$ durch Cooley-Tukey bzw. Gentleman-Sande kann mit*

$$C = O(D \ln D) \tag{45}$$

abgeschätzt werden.

4.5 Stockham FFT und Implementierung

Zum Unterschied der vorigen Algorithmen basiert die Stockham FFT - Methode **nicht** auf dem Danielson-Lanczos Lemma (4.18).

Die Stockham FFT-Methode findet Verwendung falls

- kein „Scrambling“ gewünscht wird bzw.
- fortlaufender Speicherzugriff erfolgt.

4.5.1 Stockham Formulierung

Gegeben: Signal x der Länge $D = 2^d$
Sei $X_0(j) = x_j$, $0 \leq j < D$ und α_t eine 2^t-te primitive Einheitswurzel.

Radix 2 Stockham Formulierung der FFT:

for $1 \leq t \leq d$
  for $0 \leq j < 2^{d-t}, \quad 0 \leq k < 2^{t-1}$ Iterationen für j, k

$$\begin{aligned} X_t(j2^t + k) &= X_{t-1}(j2^{t-1} + k) + \alpha_t^k X_{t-1}(j2^{t-1} + \frac{n}{2} + k) \\ X_t(j2^t + 2^{t-1} + k) &= X_{t-1}(j2^{t-1} + k) - \alpha_t^k X_{t-1}(j2^{t-1} + \frac{n}{2} + k) \end{aligned}$$

Das Resultat $X_d(k)$, $0 \leq k < n$ entspricht der gesuchten $DFT(x)$.

Eine Implemenationsvariante ist das „Überschreiben“, die Lösungen in der rechten Seite werden in dasselbe Array eingetragen. Diese Modifikation liefert das korrekte Ergebnis, allerdings ist ein „bit-scrambling“ nach Beendigung der Iteration von t nötig ($X_m(k_s)$ ist der korrekte DFT-Wert von $DFT(x)$ [$\mathsf{Scramble}(k_2)$].
Viele Implementationen führen eine Vorberechung der Potenzen von α und legen sie in einem Array U ab, $U(k) = \alpha$, $0 \leq k < \frac{n}{2}$, dann gilt α_t^k wird durch $U(k2^{d-t})$ indiziert.

Erweiterung: Radix 4 Stockham Variante:

Gegeben: Signal x der Länge $D = 4^d$
Sei $X_0(j) = x_j$, $0 \leq j < D$ und $\beta_t = \exp^{-2\pi i/4^t}$ eine 4^t-te primitive Einheitswurzel über $\mathbb{C}$.

$\mathsf{for}\ \ (1 \leq t \leq d)$
$\quad \mathsf{for}\ 0 \leq j < 4^{d-t}, \quad 0 \leq k < 4^{t-1}$

$$
\begin{aligned}
c_0 &= X_{t-1}(j4^{t-1} + k) \\
c_1 &= \beta_t^k X_{t-1}(j4^{t-1} + \frac{n}{4} + k) \\
c_2 &= \beta_t^{2k} X_{t-1}(j4^{t-1} + \frac{2n}{4} + k) \\
c_3 &= \beta_t^{3k} X_{t-1}(j4^{t-1} + \frac{3n}{4} + k) \\
d_0 &= c_0 + c_2 \\
d_1 &= c_0 - c_2 \\
d_2 &= c_1 + c_3 \\
d_3 &= i(c_1 - c_3) \\
X_t(j4^t + k) &= d_0 + d_2 \\
X_t(j4^t + 4^{t-1} + k) &= d_1 + d_3 \\
X_t(j4^t + 2 \cdot 4^{t-1} + k) &= d_0 - d_2 \\
X_t(j4^t + 3 \cdot 4^{t-1} + k) &= d_1 - d_3
\end{aligned}
$$

Der Vorteil der Radix 4 Variante ist von der Hardware abhängig. Bei einer vektororientierten Verarbeitung (CPU verwendet Vektorregister) erfolgt der Zugriff sehr viel schneller als vom Speicher. Radix 4 Iterationen leisten dasselbe wie 2 Radix 2 Iterationen allerdings mit nur einem Speicherzugriff auf die Datenarrays im Speicher, d.h. es erfolgt eine Reduktion der langsameren Speicherzugriffe.

4.5.2 Stockham FFT : ping-pong Variante

Eine Möglichkeit der Implementierung der Stockham FFT ist die „ping pong“ Variante für $D = 2^d$, Zitat aus Referenz 1.
Die Methode vermeidet das „bit-scrambling“, beinhaltet einen Speicherdurchlauf und verwendet eine Kopie des Signals. Die Berechnung verläuft 1 Schritt rückwärts und 4 Schritte vorwärts zwischen dem Originalsignal und der Kopie des Signals.

Folgender Algorithmus (4.26) verwendet Zeiger X, Y auf die Signale x und der Kopie des Signals y d.h. $X[0]$ ist das erste Element von x, durch die Addition von 4 zum Zeiger $X = X + 4 \Rightarrow X[0] = x_4$, die Arithmetik bezieht sich auf die Speicherplätze.

Algorithmus 4.26 *ping-pong Variante, in order, no bitscramble*

FFTPingPong(x) {
  $J = 1$ — Initialisierung
  $X = x, Y = y = Copy(x)$ — Zuweisung der Zeiger auf die Signale
  for $(d \geq i > 0)$ $m = 0$
    while $(m < \frac{D}{2})$
      $a = \omega^{-m}$ — ω, D-te Einheitswurzel
      for $(J \geq j > 0)$
        $Y[0] = X[0] + X[\frac{D}{2}], \quad Y[J] = a(X[0] - X[\frac{D}{2}])$
        $X = X + 1, \quad Y = Y + 1$ — weiterschalten der Zeiger um 1
      $Y = Y + J, \quad m = m + J$ — Zeiger Y und m um J weiter
    $J = 2J$
    $X = X - \frac{D}{2}, \quad Y = Y - D$ — Manipulation der Zeiger
    $(X, Y) = (Y, X)$ — Vertauschen der Zeiger
  if (d gerade) — FFT-Resultat steht in X oder Y
    return Komplexe Daten von X
  return Komplexe Daten von Y

Ein Vorteil von (4.26) ist der kontinuierliche Verlauf von J (abnehmend) für eine vektororientierte Verarbeitung d.h. die Daten als Vektoren im Speicher abgelegt.

4.6 Parallele (=4 step) FFT

Der parallele Algorithmus wird unter folgenden Bedingungen eingesetzt:

- „großes“ Signal und der Speicher zur Berechnung ist limitiert
- Signal auf verschiedene Bereiche abgelegt (Zerlegung).

Diese Bedingungen sind von der verwendeten Maschine (disk-memory, processors, ...) abhängig. Bei mehreren Prozessoren kann man die Teilberechnungen auf die verschiedenen Prozessoren verteilen. Die Idee ist

- Zerlegung des Signals
- Berechnung der einzelnen Teile
- Kombination der Teilresultate $\Rightarrow$ Endresultat FFT des Signals

Die parallele FFT basiert darauf, dass eine DFT_D der Länge $D = W \cdot H$ mittels einem Durchlauf der Zeilen und Spalten einer $H \times W$ Matrix berechnet werden kann.
Algebraische Reduktion der DFT, $X = DFT(x) =$

$$= \left(\sum_{j=0}^{D-1} x_j \omega^{-jk} \right)_{k=0}^{D-1} = \left(\sum_{J=0}^{W-1} \left(\sum_{M=0}^{H-1} x_{J+MW} \omega_H^{-MK} \right) \omega^{-JK} \omega_W^{-JN} \right)_{K+NH=0}^{D-1}$$

mit $\omega, \omega_H, \omega_W$ Einheitswurzeln vom Rang WH, H, W und den Indizes $(K + NH)$ mit $K \in [0, H-1], N \in [0, W-1]$.
Folgender Algorithmus und Beispiel sind Zitate aus Referenz 1.

Algorithmus 4.27 *parallel, four-step*

FFTParallel(x, W, H) {
  Initialisierung von $H \times W$ Matrix T (spaltenweise)
  for $(0 \leq M < H)$ Spaltendurchlauf
    for $(0 \leq J < W)$ Reihendurchlauf
      $T_{M,J} = x_{JH+M}$
  1. H in place, DFT der Länge W, für alle Reihen von T
  for $(0 \leq M < H)$ $T^{(M)} = DFT(T^{(M)})$
  2. Transponieren und Multiplikation der Elemente mit ω^{-MJ} von T
  for $(0 \leq J < W)$ obere Dreiecksmatrix
    for $(0 \leq M \leq J)$
      $t = T_{MJ}, \quad T_{MJ} = T_{JM}\omega^{-MJ}, \quad T_{JM} = t\omega^{-MJ}$
  3. W in place, DFT der Länge H, für alle Reihen von T
  for $(0 \leq J < W)$ $T^{(J)} = DFT(T^{(J)})$
  4. liefert $DFT(x)$ als Elemente (spaltenweise) zurück
  return T T_{MJ} ist $DFT(x)_{JH+M}$

Beispiel: FFT für ein Signal der Länge $D = 4 = 2 \cdot 2$ über $\mathbb{C}$ und der primitiven 4-ten Einheitswurzel $\omega_4 = \exp^{2\pi i/4} = i$

$$x = (x_0, x_1, x_2, x_3) \rightarrow T = \begin{pmatrix} x_0 & x_2 \\ x_1 & x_3 \end{pmatrix}$$

1. $H = 2$ Reihen von FFTs mit Länge $W = 2$

$$T = \begin{pmatrix} x_0 + x_2 & x_0 - x_2 \\ x_1 + x_3 & x_1 - x_3 \end{pmatrix}$$

2. Transposition und punktweise Multiplikation mit G

$$G = \begin{pmatrix} 1 & 1 \\ 1 & i \end{pmatrix} \Rightarrow T = \begin{pmatrix} x_0 + x_2 & x_1 + x_3 \\ x_0 - x_2 & -i(x_1 - x_3) \end{pmatrix}$$

3. $W = 2$ Reihen von FFTs mit Länge $H = 2$

$$T \rightarrow \begin{pmatrix} X_0 & X_2 \\ X_1 & X_3 \end{pmatrix}$$

4. liefert $DFT(x)$

Erweiterung auf „six step“:
Der Transpositionsschritt von (4.27) kann zur Umwandlung auf spaltenweise Indizierung benutzt werden.

1. Transponieren und umwandeln auf spaltenweise Indizierung
2. H in place, DFT der Länge W, für alle Reihen von T
3. Multiplikation der Elemente mit ω^{-MJ} von T
4. W in place, DFT der Länge H, für alle Reihen von T
5. Rücktransposition von T
6. liefert $DFT(x)$ von lexikographischer Ordnung zurück

Zur Effizienzsteigerung sollten die Signale in der spaltenweisen Indizierung gespeichert werden.

4.7 Diskrete gewichtete Transformationen (DWT)

Die diskrete gewichtete Transformation (DWT) ist eine Variante der DFT-basierenden Faltung und bildet ein Werkzeug zur Realisierung von Faltungen sowie der Durchführung von Berechnungen für sehr große Zahlen.

Definition 4.28 (Diskrete gewichtete Transformation)
x ein Signal der Länge D und a das Signal mit den Gewichten derselben Länge D, wobei jedes a_j invertierbar ist. Dann ist die diskrete gewichtete Transformation $X = DWT(x, a)$, das Signal mit den Elementen

$$X_k = \sum_{j=0}^{D-1} (a \star x)_j \omega^{-jk} \tag{46}$$

und der inversen $DWT^{-1}(X, a) = x$ mit

$$x_j = \frac{1}{Da_j} \sum_{k=0}^{D-1} X_k \omega^{jk}. \tag{47}$$

Die gewichtete zyklische Faltung zweier Signale ist das Signal $z = x \times_a y$ mit

$$z_n = \frac{1}{a_n} \sum_{j+k \equiv n \bmod D} (a \star x)_j (a \star y)_k \tag{48}$$

$\Rightarrow DWT = DFT(a \star x)$. Die DWT ermöglicht die Realisierung alternativer Faltungen, z.B. zur Vermeidung von „zero-padding“.

Satz 4.29 (Gewichtete Faltung) *Signale x, y,Gewichtssignal a mit derselben Länge D. Dann lautet die gewichtete zyklische Faltung von x, y*

$$x \times_a y = DWT^{-1}(DWT(x, a) \star DWT(y, a), a)$$

d.h.

$$(x \times_a y)_n = \frac{1}{Da_n} \sum_{k=0}^{D-1} (X \star Y)_k \omega^{kn}$$

Durch die Wahl des Gewichtssignals $a = (A^j)$, $j \in [0, D-1]$, A ist eine primitive $2D$-te Einheitswurzel folgt die Identität

$$x \times_{\_} y = x \times_a y,$$

d.h. die zyklische entspricht der negazyklischen Faltung.

Ein Anwendungsgebiet der DWT ist die Multiplikation 2er Zahlen modulo einer Fermat-Zahl $F_n = 2^{2^n} + 1$. Es existieren 3 Ansätze für die Berechnung (xy) mod F_n mittels einer Faltung von Signalen der Länge D und der Basis B mit $F_n = B^D + 1$:

1. zyklische Faltung: „zero padding“ der Signale x, y auf Länge $2D \Rightarrow$ Standard - FFT- Methode

2. negazyklische Faltung: Gewichtete Faltung für die Länge D mit dem Gewichtssignal $a = (A^j)$,A $2D$-te Einheitswurzel
   = Multiplikation modF_n

3. „right-angle“ Faltung: Erstellen von Signalen der Länge $\frac{D}{2}$ mit $x' = L(x) + iH(x)$ und y', gewichtete Faltung für x', y', A $4D$-te Einheitswurzel.

Der Vorteil der 3. Variante liegt im Vermeidung des „zero-padding“ und dem Halbieren der Längen.

Ein weiteres Beispiel für die DWT ist die Entdeckung neuer Mersenne-Primzahlen. Die Mersenne-Primzahlen $= 2^q - 1$ sind die größten bekannten Primzahlen, als Beispiel $2^{1398269} - 1$ oder $2^{6972593} - 1$. Für die Berechnung solcher enorm großer Zahlen, in der Größenordnung $\approx 2^{1000000}$, wird ein schneller Algorithmus benötigt.

Der gesuchte Algorithmus heißt IBDWT, irrational-basierende, diskret gewichtete Transformation, da eine spezielle, irrationale Basis verwendet wird.

Gegeben: die Mersenne-Primzahl $p = 2^q - 1$ und eine ganze Zahl x in Darstellung zur Basis $= 2$, $x = \sum_{i=0}^{q-1} x_j 2^j$. Bei sehr großen Zahlen sind Optimierungen zur Effizienzsteigerung nötig, z.B. bei der Standard-FFT wird ein „zero-padding“ der Signale auf Länge 2^l verwendet.

Satz 4.30 *Für $p = 2^q - 1$ (p nicht unbedingt prim) und die ganzen Zahlen $0 \leq x, y, < p$ wähle eine Signallänge $1 < D < q$. x besitzt die variablen Basisdarstellung*

$$x = \sum_{j=0}^{D-1} x_j 2^{\lceil qj/D \rceil} = \sum_{j=0}^{D-1} x_j 2^{\sum_{i=1}^{j} d_i}$$

mit

$$d_i = \lceil qi/D \rceil - \lceil q(i-1)/D \rceil,$$

jede Stelle $x_j \in [0, 2^{d_{j+1}-1}]$, genauso für y. Das Gewichtssignal a mit Länge D ist definiert als

$$a_j = 2^{\lceil qj/D \rceil - qj/D}.$$

Die gewichtete zyklische Faltung $x \times_a y$ ist ein Signal mit ganzen Zahlen, äquivalent (ohne Übertrag) zur variablen Basisdarstellung von xy mod p.

Vorteile:

- Länge $= 2^t \Rightarrow$ erlaubt Verbesserungen
- Arithmetik auf einer größeren Basis als $B = 2$

Die Länge $D = 2^k$ wird so gewählt, dass $\lfloor 2^{q/D} \rfloor$ eine akzeptable Größe erreicht, aber klein genug ist um zu große numerische Fehler (Rundung) zu vermeiden.

Der Satz führt direkt zum Algorithmus 4.31 für eine Mersennezahl $p = 2^q - 1$ (nicht unbedingt prim) und positiven ganzen Zahlen x, y, welcher in variabler Basisdarstellung die Lösung $xy \bmod p$ liefert.
Im folgenden wird eine Implementation des IBDWT und ein Beispiel für die Parameter gegeben. (Zitat aus Referenz 1.)

Algorithmus 4.31

IBDWTMersenne(x, y, p) {
  Erstellen der Signale x, y, a aus Satz 4.30
  $X = DWT(x, y), \quad Y = DWT(y, a)$ — FFT-Methode
  $Z = X \star Y$ — punktweise Multiplikation
  $z = DWT^{-1}(Z, a)$ — inverse Transformation
  $z = round(z)$ — elementweises Runden
  $carry = 0$ — Übertrag in variabler Basis
  for $(0 \leq n < len(z))$ {
    $B = 2^{d_{n+1}}$
    $v = z_n + carry$
    $z_n = v \bmod B$
    $carry = \lfloor \frac{v}{B} \rfloor$ }
  if $(carry > 0)$ — höchste Stellen von $z > 0$
    $z = z \bmod p$ — modulare Reduktion
  return z }

Zur Erklärung ein Beispiel für $p = 2^{521} - 1$, $q = 521$ und die Signallänge wird mit $D = 16$ angenommen. Dann lauten die Signale aus Satz 4.30

$$d = (33, 33, 32, 33, 32, 33, 32, 33, 33, 32, 33, 32, 33, 32, 33, 32),$$

und das gewichtete Signal

$$a = (1, 2^{7/16}, 2^{7/8}, 2^{5/16}, 2^{3/4}, 2^{3/16}, 2^{5/5}, 2^{1/16}, 2^{1/2},$$
$$2^{15/16}, 2^{3/8}, 2^{13/16}, 2^{1/4}, 2^{11/16}, 2^{1/8}, 2^{9/16}).$$

Für eine typische Gleitpunkt FFT liefert das Signal a keine exakten Elemente $\Rightarrow$ korrektes Ergebnis durch Rundung. Die Länge D wird entsprechend klein gewählt, um die Rundungsfehler nicht zu groß werden zu lassen.

4.8 Transformationsmethoden in $\mathbb{F}_p$

Endliche Körper $\mathbb{F}_{p^n}$ sind algebraische Strukturen mit endlich vielen Elementen und den Operationen $+, -, \cdot$.
Ein Beispiel sind die Restklassenringe $\mathbb{Z}_n$.

Algebraische Aspekte:

- p Primzahl, $n \geq 1 \Rightarrow \exists$ Körper mit p^n Elemente $\mathbb{F}_{p^n}$
- Alle Körper mit p^n Elementen sind isomorph zueinander.
- Die Menge der multiplikativ inversen Elemente $\mathbb{F}_{p^n}^* = \mathbb{F}_{p^n} \ \{0\}$ bildet eine zyklische Gruppe. Es handelt sich dabei um alle Lösungen der Gleichung
$$x^{p^n-1} = 0$$
- $\mathbb{F}_{p^d} \subseteq \mathbb{F}_{p^n} \Leftrightarrow d|n$
- Für jeden Teiler d von $p^n - 1$ existieren $\phi(d)$, d-te Einheitswurzeln.
- Finden von Einheitswurzeln:
p prim, $n \Rightarrow \exists t$ mit $n|p^t - 1$ d.h. $\mathbb{F}_{p^t}$ enthählt die n-ten Einheitswurzeln

Laut Definition (4.10) kann die DFT über endlichen Ringen und Körper durchgeührt werden, solange die genannten Vorraussetzungen in $\mathbb{F}_{p^n}$ erfüllt sind. Ein Vorteil von endlichen Körpern liegt bei den Einheitswurzeln, wobei die Existenz und Anzahl bestimmter Einheitswurzeln nicht immer einfach zu beantworten ist. Die arithmetischen Operationen $+, -, \cdot$ operieren rundungsfehlerfrei.

Die DFT der Länge D ermöglicht eine zyklische Faltung, falls D^{-1} und die D-te Einheitswurzel ω in der algebraischen Umgebung existieren.

Satz 4.32 *$\mathbb{F}_p$, p Primzahl und Teiler $d|p-1$ lautet die Tranformation*

$$X_k = \sum_{j=0}^{(p-1)/d-1} x_j h^{-jk} \bmod p, \tag{49}$$

mit h ein Element mit multiplikativer Ordnung $(p-1)/d$ in $\mathbb{F}_p$. Die inverse Transformation lautet

$$x_j = -d \sum_{k=0}^{(p-1)/d-1} X_k h^{jk} \bmod p, \qquad (\frac{p-1}{d})^{-1} \bmod p \equiv -d. \tag{50}$$

Der Satz wird in der Literatur oft als NTT (number-theoretical transform) bezeichnet. Die bisherigen Überlegungen und Sätze sind bei erfüllten Voraussetzungen auf die endlichen Körper übertragbar. Es sind jedoch einige Verbesserungen und Effizienzsteigerungen durch die speziellen Eigenschaften von endlichen Körpern möglich.

Beispiel: Diskrete Galois Transformation (DGT)
definiert für einen endlichen Körper $\mathbb{F}_{p^2}$ für $p = 2^q - 1$ einer Mersenne-Primzahl. In solchen Körpern ist die multiplikative Ordnung $|\mathbb{F}^*_{p^2}| = p^2 - 1 = 2^{q+1}(2^{q-1} - 1)$, womit primitive Einheitswurzeln der Ordnung $N = 2^k$ unter der Bedingung $k \leq q+1$ existieren. Somit lautet die diskrete Transformation der Länge N

$$X_k = \sum_{j=0}^{N-1} x_j h^{-jk} \bmod p \tag{51}$$

Diese Transformation ist ein „pure-real“ Signal und somit ist eine Erweiterung auf komplexe (Gaussche) Zahlen möglich, dadurch wird die Länge halbiert.

$$N = 2^k, \quad x_j = Re(x_j) + iIm(x_j), \quad h = Re(h) + iIm(h),$$

wobei h ein Element mit multiplikativer Ordnung N in $\mathbb{F}_{p^2}$ ist. Das Element X_k ist eine Gaussche Zahl mod$p$. Zur Implementierung ist nun ein Element mit multiplikativer Ordnung 2^{q+1} nötig, Satz 4.33.

Satz 4.33 (Creutzburg und Tasche) *Sei $p = 2^q - 1$ eine Mersenne - Primzahl mit q ungerade. Dann ist*

$$g = 2^{2^{q-2}} + i(-3)^{2^{q-2}}$$

ein Element von Ordnung 2^{q+1} in $\mathbb{F}^*_{p^2}$.

DGT-Faltung der Signale x, y in $\mathbb{F}_{p^2}$:

- Berechnung von $h = g^{2^{q+2-k}}$ mit Ordnung $N/2$,
- der Aufspaltung der Signale in Real und Imaginärteil
- Durchführung der DFT

Ein Vorteil der DGT-Faltung ist, dass alle mod Operationen für eine Mersenne - Primzahl durchgeführt werden, der spezielle Aufbau der Primzahl erlaubt Effizienzsteigerungen.

4.9 Schönhage Methode

Untersuchung von Restklassenringe $\mathbb{Z}_n$ mit n einer Fermatzahl $F_n = 2^{2^n} + 1$.

Definition 4.34 *FNT, (Fermat number transform), bezeichnet die DFT für eine prime Fermatzahl F_n*

Die FNT ist sehr interessant, da bei dieser Methode viele Verbesserungen der Effizienz möglich sind. Sie besitzt einen sehr niedrigen Aufwand.

Vorteile der FNT:

- Einfaches Finden von Einheitswurzeln in $\mathbb{Z}_{F_n}$:
  Für jeden Teiler d von $p - 1 = 2^{2^n}$ existiert eine Einheitswurzel, d.h. als Teiler kommen nur 2-Potenzen in Frage, 2 ist eine 2^{n+1}-te Einheitswurzel, 2^2 eine 2^n-te Einheitswurzel und usw.
- Wahl der Basis $= 2^k$ erlaubt verschiedene Reduktionsvarianten
- Vereinfachung der Arithmetik für eine Basis B
- spezielle Moduli modF_n da $2^q \equiv -1 \bmod (2^q + 1)$
- Multiplikationen und Divisionen 2^i = Shifts und Subtraktionen
- Länge der Signale sind $D = 2^d$, welche Optimierungen für FFT erlaubt

Die Erstellung der DFT wird nur durch simple Schiebeoperationen, Einheitswurzeln sind Potenzen von 2, erreicht. Die einzige „echte“ Multiplikation erfolgt in der punktweisen Multiplikation. Durch die Wahl der Basis $B = 2^k$ ergeben sich weitere Verbesserungen auf den Computer.

Der Ansatz von Schönhage ist die Berechnung einer Multiplikation von $xy \bmod F_n$ mittels FFT mit den oben erwähnten Vorteilen. Für die Berechnung von xy in $\mathbb{Z}$ wird eine Fermatzahl so gewählt, dass gilt $n \geq \lceil \lg x \rceil + \lceil \lg y \rceil$. Dann entspricht $xy \bmod F_n$ gleich dem Produkt xy in $\mathbb{Z}$.
Die Realisierung dieses Ansatzes ist die negazyklische Faltung dieser „zero-padded“ Signale mittels einer DWT.

Satz 4.35 *Die obere Schranke für den Aufwand einer FNT beträgt $O(DFT)$ simplen Schiebeoperationen.*

Diese Schranke wird bereits durch einfache Verbesserungen der FFT - Algorithmen unterschritten.

4.10 Nussbaumer Methode

Die Nussbaumer-Faltung im Ring R wird mittels der Identität 52 berechnet. Der zugrundeliegende Ring R besitzt die Eigenschaft, $x = y \Leftrightarrow 2x = 2y$, d.h. durch 2 kürzbar.

Satz 4.36 *Für zyklische Faltung von Signalen x, y mit gerader Länge D gilt*

$$2(x \times y) = [(u_+ \times v_+) + (u_- \times_- v_-)] \cup [(u_+ \times v_+) - (u_- \times_- v_-)] \qquad (52)$$

mit den Signalen u, v basierend auf den Hälften der Signale x, y

$$u_\pm = L(x) \pm H(x), \quad v_\pm = L(y) \pm H(y).$$

d.h. die zyklische Faltung setzt sich aus einer zyklischen und negazyklischen Faltung zusammen.

Beweis: Rekursionsformel mittels Polynomalgebra beweisen

Die Idee von Nussbaumer ist die Signale x, y in einem Polynomring zu transformieren, dort die Mulitplikation durchzuführen und das Resultat rücktransformieren. Bei dieser Methode werden die Rundungsfehler vermieden. Die Rekursion ist nur effizient, falls ein schnelle negazyklische Faltung existiert, um die zyklische Faltung direkt zu berechnen.

Satz 4.37 *Sei $D = 2^k = mr$, $m|r$. Die negazyklische Faltung von Signalen der Länge D mit Elementen aus einem Ring R, ist äquivalent zu, Polynomkoeffizienten verknüpft mit Signalelementen, zu der Multiplikation in einem Polynomring*

$$S = R[t]/(t^D + 1).$$

Dieser Ring ist isomorph zu $T[t]/(z - t^m)$ mit T einem Polynomring $T = R[z]/(z^r + 1)$. $z^{r/m}$ ist die m-te Wurzel von -1 in T.

Gesucht: Produkt $z = xy$

Schema der Transformationen:

1. Signalelemente $x_j \in R$ sind die Koeffizienten von $x(t)$

$$x = (x_0, x_1, \ldots, x_{D-1}) \leftrightarrow x(t) = x_0 + x_1 t + \ldots + x_{D-1} t^{D-1},$$

2. Polynommultiplikation im Polynomring S

$$x(t)y(t) \in S \text{ mit } x(t) = \sum_{j=0}^{m-1} X_j(t^m) t^j,$$

3. Polynome $X_j, Y_j \in T$ mit $X_j(z) = x_j + x_{j+m}z + \ldots + x_{j+m(r-1)}z^{r-1} \Rightarrow$ zyklische Faltung in T

$$Z = (X_0, X_1, \ldots, X_{m-1}, 0, \ldots, 0) \times (Y_0, Y_1, \ldots, Y_{m-1}, 0, \ldots, 0)$$

4. Das Resultat z sind die Koeffizienten von $z(t) = x(t)y(t)$ bzgl. t^i

$$z(t) = x(t)y(t) = \sum_{j=0}^{2m-2} Z_j(t^m)t^j,$$

Folgender Algorithmus berechnet die Nussbaumer Faltung, zitiert aus Referenz 1.

Algorithmus 4.38 (Nussbaumer Faltung, zyklisch und negazyklisch) *Signale x, y mit Länge $D = 2^k$ und Elemente $\in$ Ring R, R erlaubt das Kürzen durch 2. Algorithmus 4.38 berechnet die zyklische $(x \times y)$ in cyc und die negazyklische Faltung $(x \times_- y)$ in neg. Die negazyklische Funktion smallneg führt die negazyklische Faltung für Längen unterhalb einer Schranke (Schulmethode, Karatsuba, ...) durch.*

Initialisierung der Faktoren r, m und der Schranke $blen$
$r = 2^{\lceil k/2 \rceil}$, $m = D/r$, $blen = 16$

cyc(x, y) { rekursive, zyklische Faltung
Aufruf von cyc und neg mit halbierter Länge und
Berechnung der zyklischen Faltung mittels der Identität 52 }

neg(x, y) { rekursive, negazyklische Faltung
if $(len(x) \leq blen)$ return $smallneg(x, y)$

Transpositionsschritt
Erstellen von $2m$ Arrays X_j, Y_j mit der Länge r
„Zero-padding“ die X, Y Mengen mit $|M| = 2m$ Polynomen
Verwende die Wurzel $g = z^{r/m}$ um mit 2 DFTs der Länge $2m$
die Transformationen $\hat{X}, \hat{Y}$ zu berechnen
for $(0 \leq h < 2m)$ $\hat{Z}_h = neg(\hat{X}_h, \hat{Y}_h)$ punktweise Multiplikation
$g = z^{r/m}$, berechne die inverse DFT $(2m)$ um aus $\hat{Z}$,Z zu berechnen
Berechnungen im Ring S
Finden der Koeffizienten z_n von t^n, $n \in [0, D-1]$ von **??**
return (z_n)

Satz 4.39 *Für die Komplexität der Nussbaumer-Faltung gilt*

$$O(D \ln D)$$

Operationen im Ring R, dies ist äquivalent zu Gleitpunkt FFT - Methoden.

4.11 Zusammenfassung

Die Multiplikation ist, im Vergleich zu anderen arithmetischen Operationen (außer Division), mit einem erheblich größeren Aufwand verbunden. Als Beispiel sei hier die Funktion PowAdd genannt:
PowAdd : Ersetzen der Multiplikation durch Additionen
Die Ersetzung führt in der Praxis zu einer erheblichen Aufwandsreduktion (im Vergleich zur klassischen Multiplikation), natürlich gibt es weit aus effizientere Algorithmen.

Gesucht: schnelle Algorithmen für die Multiplikation

Den Anfang machten Karatsuba und Tom-Cook mit ihrer theoretischen Verbesserung der Multiplikation, durch Aufspalten in kleiner Terme und somit kleinere Multiplikationen. Der Aufwand dieser Methoden kann mit

$$O(n^\alpha) = O(\ln^\alpha N)$$

für die Schulmethode mit $\alpha = 2$, für Karatsuba und Tom-Cook je nach Variante < 2 Bitoperationen abgeschätzt werden. Der Nachteil dieser Verbesserungen liegt in der komplizierten Implementierung, d.h. in Praxis ergeben sich Probleme, z.B. Rekombination der Subprodukte.

Eine viel schnelleren und einfacheren Ansatz bildet die FFT-Methode. Der Vorteil der FFT-Methode liegt ihn ihrem einfachen Handling, d.h.

- allgemeine Definition : FFT für Signale definiert
- Vorraussetzung: Existenz einer primitiven Einheitswurzel, sowie invertierbarer Elemente
- Reduktion des Aufwandes auf eine punktweise Multiplikation

Die Signaldarstellung erlaubt eine Spezialisierung auf verschiedene Körper bzw. Variation der Methoden um die FFT zu verbessern. Die Vorraussetzungen werden für die meisten bekannte Körper bzw. Ringe erfüllt, z.B. Gleitpunkt-FFTs operieren gewöhnlich in $\mathbb{C}$. Das Schema der FFT ist

- Transformation eines Signal
- punktweise Multiplikation
- Rücktransformation des Resultates

Varianten der FFT :

1. Radix - 2 -FFTs:
   - rekursiver Ansatz mit $O(D \ln D)$ durch Danielson-Lanczos
   - Cooley-Tukey Variante vom rekursiven Ansatz (nur Schleifen)
   - Gentleman-Sande
2. Stockham Formulierung
3. parallele FFT

Diese Methoden sind so genannte Gleitpunkt-FFTs, d.h. über reellen bzw. komplexen Ring, Körper gebildet. Innerhalb dieser Methoden gibt es ebenfalls Optimierungen, z.B. „pure-real“ Signale in komplexe Signale umwandeln und damit die Länge halbieren.

Spezielle FFT-Methoden für ganzzahlige Ringe und Körper:

1. NTT (number theoretical transform)
2. FNT (Fermat number transform) für Restklassenringe $\bmod F_n$
3. Schönhage
4. Nussbaumer

Der Schönhage-Algorithmus nutzt die DFT , meist FNT, für einen Restklassenring $\mathbb{Z}_N$, indem das N so vergrößert wird, dass das ganzzahlige Produkt gleich dem Produkt $\bmod N$ ist. Zu diesem Zweck wird eine besondere DFT, die FNT, verwendet, da durch die Fermat-Zahl die Operationen vereinfacht werden. Der Ansatz der Nussbaumerfaltung bezieht sich auf eine Identität für Faltungen um somit zyklische und negazyklische Faltungen zu berechnen. Dies geschieht mit Polynomtransformationen.

Zusammenfassend gelten folgende Komplexitätsabschätzungen für Operanden mit n Bits (Länge $D = n/b$) und der Basis $B = 2^b$:

Algorithm	optimal B	Komplexität
Standard FFT, fixe Basis	$\dots$	$O_{op}(D \ln D)$
Standard FFT, variable Basis	$O(\ln n)$	$O(n(C \ln n)(C \ln \ln n) \dots)$
Schönhage	$O(n^{1/2})$	$O(n \ln n \ln \ln n)$
Nussbaumer	$O(n/\ln n)$	$O(n \ln n \ln \ln n)$

mit C einer Konstante und $\ln \ln \dots n \geq 1$, O_{op} bezieht sich auf die Operationen im Algorithmus, alle anderen Abschätzungen auf Bitoperationen.

5 Implementierung in C#

5.1 Zahlenimplementation

Für die Realisierung in C# wurde eine Klassenstruktur entwickelt um die Darstellung einer ganzzahligen Zahl mit unbestimmter Länge zu ermöglichen. Übersicht der Klassen:

- LNumber: enthält die Daten und Operationen $(+, -, \ldots)$
- Kernel: Hilfsklasse zur schnellen Berechnung
- LNMore: enthält zusätzliche Methoden und einige Algorithmen aus Kapitel 1.
- LNBit: Klasse zur Bitmanipulation von LNumber
- LNArray<Typ>: Generische Klasse für ein Array
- LNMatrix<Typ>: Generische Klasse für eine Matrix
- LNList<Typ>: Generische Klasse für eine Liste

LNumber
LNumber speichert die Zahl in einer 2^{32}-Darstellung innerhalb eines uint-Feldes. Dieses Feld repräsentiert die Zahl und wird manipuliert um die verschiedenen arithmetischen Operationen $+, -, \cdot, /$ zu realisieren.

Kernel
Kernel führt grundlegende Operationen mittels Zeigern durch $\Rightarrow$ unsafe - Code. D.h. die Programmteile verwenden die einfachsten (=schnellsten), maschinennahen Implementationen.

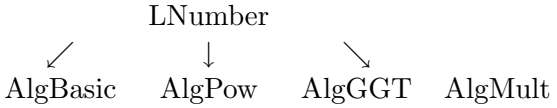
LNMore
LNMore enthählt zusätzliche Methoden wie Zufallszahlenerzeugung bzw. einen Befehlszeileninterpreter, aber auch Algorithmen aus Kapitel 1.

LNBit
Für spezielle Algorithmen ist eine schnelle Bitmanipulation notwendig. Zu diesem Zweck kann LNBit die Daten aus LNumber in Hinblick auf die Bitdarstellung manipulieren.

LNArray<Typ>, LNMatrix<Typ>, LNList<Typ>
Generische Klassen, d.h. der Typ wird vom Benutzer definiert, um verschiedene Strukturen Array, Matrix mit Operatoren, Liste zu implementieren.

5.2 Algorithmen

Die vorgestellten Algorithmen werden, ähnlich dem Pseudo-Code, implementiert unter zu Hilfenahme von LN-Klassen.



Die Klassen LNumber, LNArray<Typ>, ... repräsentieren eine ganze Zahl mit unbeschänkter Länge und die zugehörigen arithmetischen Operationen. Die Algorithmen in AlgBasic, AlgExp, AlgGGT verwenden diese Klassen. D.h. die Daten und grundlegenden Operationen werden in der Klasse LNumber gekapselt. Algorithmen verwenden LNumber zur Berechnung des Resultats (in AlgExp die Potenzen und AlgGGT den größten gemeinsamen Teiler). Die Algorithmen können vom Typ statisch, d.h. Funktion ohne Erstellen des Objektes aufrufbar bzw. nicht statisch sein. Nicht statische Methoden benötigen ein Objekt um die Berechnungen auszuführen.

statisch	nicht statisch
AlgXXX.Function()	AlgXXX obj = new AlgXX() cc obj.Function()

Die Variablen in diesen Klassen speichern die Daten für eine Vorberechung oder einem mehrteiligen Funktionenaufruf.

Beispiel:

```
public void KaryPrecomputation(int k) {
    TableGI(k);
    TableP(k);
    TableAB(k,FindAB"); }

public LNumber KaryGCD(LNumber x, LNumber y,int k)
    if (x == y)
        return new LNumber(x);
    EliminateDivisorsBegin(x,y);
    KaryReduction(k);
    return EliminateDivisorsEnd(); }
```

Die Containter-Klassen LNList<Typ>... sind für Typen mit einem Default-Konstruktor definiert.
Nachfolgend sind die oben erwähnten Klassen in C# implementiert (Prototypen und maximal 2 Beispielroutinen).

5.3 Klasse in C# : AlgBasic

```
public class AlgBasic{

public AlgBasic()        // Default-Konstruktor
/// <summary> berechnet $basis^2$ in dem die Symmetrien ausgen
/// uetzt, d.h. 2x mal schneller als basis*basis </summary>
public static LNumber Square(LNumber basis)
/// <summary> dividiert mittels der binaeren Division
///  mit der Vorschrift |dividend| >= |divisor| > 0 </summary>
public static LNumber BinaryDiv(LNumber dividend,
  LNumber divisor)
/// <summary> berechnet xy mod N mittels binaeren mul-mod
///  Vorraussetzungen: 0 <= x,y < N </summary>
public static LNumber BinaryMulMod(LNumber x, LNumber y,
                                   LNumber N)
/// <summary> berechnet Reciprocal (allgemeiner Umkehrwert)
/// fuer Newton-Division (DivisionFreeMod) N>0</summary>
/// <param name="N"> fuer N allg. Umkehrwert  </param>
/// <returns> R(N) = 4^(B(N-1))/N </returns>
public static LNumber Reciprocal(LNumber N)

/// <summary> berechnet x/N und x mod N mit allg. Umkehrwert
/// Vorraussetzung: 0 kleiner gleich x kleiner N </summary>
/// <param name="x"> Dividend </param>
/// <param name="N"> Divisor </param>
/// <returns> [x/N,x mod N]</returns>
public static LNArray<LNumber> DivisionFreeMod(LNumber x,
  LNumber N) {
  LNumber rest , quotient;
  LNumber R = Reciprocal(N); // allgemeiner Umkehrwert
  int s=2*(R-1).Bitlength()-2;
  quotient = (x*R)>>s;     // Naeherung fuer x/N
  rest = x - N*quotient;   //Naeherung fuer x mod N
  while (rest>=N)   // Reduktion des Fehlers cN
  { rest-=N;
    ++quotient;  }
  return new LNArray<LNumber>(2,quotient,rest); }

// liefert bei i =0 x/N und i=1 x mod N von DivisionFreeMod
public static LNumber GetDivisionFreeMod(int i,
  LNumber x, LNumber N)
/// <summary> berechnet die Wurzel von N, 0 < N </summary>
/// <param name="N"> zu wurzelziehende Zahl </param>
```

```
/// <returns> Wurzel von N</returns>
public static LNumber NewtonSquareRoot(LNumber N)
/// <summary> berechnet z mod N fuer  N=2^q+c </summary>
public static LNumber FastModSpecialForm(LNumber z,int q,int c)
{ LNumber y = new LNumber();
  LNumber x = new LNumber(z);
  LNumber N = new LNumber(1,2);
  N=(N<<(q-1))+c;
  while (x.Bitlength() > q)  // rekursiver Aufruf des Satzes
  { y = x >> q;
    x -= y << q;
    x -= c*y;  }
  if (x>=N) x -= N;    // Korrektur auf positiv
  if (x.Vorzeichen<0)  x += N;
  return x; }

/// <summary> berechnet x mod N fuer N=k2^q+c </summary>
public static LNumber FastModProthForm(LNumber x,
  LNumber k, int q, int c)   }}
```

5.4 Klasse in C# : AlgExp

```
public class AlgExp    {

// Variablen
// beinhaltet vorberechnete Potenzen
public LNList<LNumber> vorBerechnet;
// koeff fuer Euklid
public LNList<uint> m_koeff;
// Liste der Chainelemente
public LNList<int> m_chain;
// Art der Methode 1,2,3
public int vorberechnungMethode;

public AlgExp()
#region statische Potenzfunktionen = 1 Funktion
public static LNumber PowLeftRight(LNumber x, LNumber y)
public static LNumber PowRightLeft(LNumber x, LNumber y)
#endregion
#region allgemeine Potenzen mit Vorberechnungsfunktionen
/// <summary> x fuer die Vorberechnung </summary>
/// <param name="x"> Potenzen x^i gesucht </param>
/// <param name="methode"> d = 1,2 oder 3</param>
public void VorberechnungInit(LNumber x, int methode)
```

```
/// <summary> berechnet alle Potenzen x^i unter der
///  Bedingung      x^i kleiner x^{2^k} </summary>
/// <param name="k"> x^2^k > x^i in vorBerechnet </param>
public void Vor2Powk(int k)
/// <summary> haengt anz von Potenzen an d.h. [1,x,x^2,..,x^j]
///  => [1,x,x^j,x^j+1..,x^j+anz-1] </summary>
/// <param name="anz"> Anzahl der neuen Potenzen </param>
public void VorAppend(int anz)
/// <summary>berechnet die Anzahl der Stellen bzgl.
/// der Basis 2^basisLaenge </summary>
/// <param name="basisLaenge"> Basis = 2^basisLaenge </param>
/// <returns> D fuer y=(y0,y1,..yD-1) </returns>
public int GetAnzStellen(LNumber y,int basisLaenge)
/// <summary> liefert die durch vorberechneten Potenz
/// in vorBerechnet </summary>
/// <param name="index"> Potenz x^index </param>
/// <returns> LNumber x^index</returns>
public LNumber GetBMethode(int index)
#endregion

#region B-aere Methoden
/// <summary>berechnet die Potenz x^y mittels der b-aeren Var-
/// iante, Vor: mind. Vor2Powk(basisLaenge) </summary>
public LNumber PowWindow(LNumber x, LNumber y, int basisLaenge)
/// <summary>berechnet die Potenz x^y mittels der variablen
/// b-aeren Variante Vor: mind. Vor2Powk(k) </summary>
/// <param name="k"> Basis = 2^k, maximale Blocklaenge </param>
public LNumber PowSlidingWindow(LNumber x,LNumber y, int k)
{ // Repraesentation der Bits von y
  LNBit bitsOfy = new LNBit();
  bitsOfy.Set(y);
  LNumber res = new LNumber(1,1);
  // Beginn des Durchlaufs am hoechsten Bit
  int i=bitsOfy.Bitlength-1, j;
  // Durchlauf aller Bits
  while (i>=0)
  { // i-tes Bit = 0 => Quadrieren
    if ( !bitsOfy.GetBool(i) )
    { res = res.Square();
      --i;  }
    else
    { // ist letztes Bit des k-Blockes
      j = Math.Max(i-k+1,0);
      // sucht j-tes Bit = 1 solange EndBit = 1
```

```
      while ( !bitsOfy.GetBool(j) )   ++j;
      // Multiplikation res mit vorberechneter Potenz
      // x^[yi,..yj]
      res = res^(2<<(i-j));
      res = res*vorBerechnet[(int)bitsOfy[j,i]];
      i=j-1;
    }
  }
  return res;  }
#endregion

#region spezielle Potenzen fixe Basis
/// <summary> berechnet die Potenz x^y unter der Bedingung vor
/// berechnete Potenzen x^(B^D) > x^i, Basis = 2^b, Vorberech
/// nung: Vor2Powk(bLaenge*GetAnzStellen(y)) </summary>
/// <param name="bLaenge"> Basis = 2^bLaenge </param>
public LNumber PowFixXWindow(LNumber x,LNumber y,int bLaenge)
/// <summary> speichert fortlaufend die Potenz in vorBerechnet
/// d.h. [1,x] => [1,x,x^exp,x^(exp^2),x^8,...x^(exp^(anz-1))]
/// und Erweiterung um anz Potenzen [1,..x^i] =>
/// [1..x^i,x^i^exp,..x^i^(exp^anz-1)] </summary>
/// <param name="anz"> Anzahl der Potenzen </param>
/// <param name="exponent"> Exponent fuer Potenzen </param>
public void VorPow(int anz,int exponent)
/// <summary> berechnet die Potenz mit Hilfe des Euklidischen
/// Algorithmus (angewendet auf Koeffizientliste), Vor:
/// VorberechnungInit(x^2^blaenge,[1,2]),  mind.
/// VorPow(GetAnzStellen(y,blange)-2,2^blaenge) </summary>
public LNumber PowFixXEuklid(LNumber x,LNumber y, int bLaenge)
/// <summary> bestimmt Index des hoechsten bzw. zweithchsten
/// Elementes der Liste fuer PowFixXEuklid </summary>
/// <param name="ausnahme"> -1 => hoechstes Element,
/// >= 0 zweithoechstes </param>
/// <returns> Index, bei Fehler = -1 </returns>
private int IndexMax(int ausnahme)
#endregion

#region spezielle Potenzen fixer Exponent
/// <summary> erstellt eine AdditionChain aufgrund der
/// Binaerdarstellung d.h. 2er-Potenzen-Chain </summary>
/// <param name="y"> AdditionChain fuer Exponenten y </param>
public void BinaryAdditionChain(LNumber y)
/// <summary> liefert das index-te Element der AdditionChain
/// <param name="index"> index-te Element </param>
```

```
/// <returns> [i1,i2] </returns>
public int[] GetAdditionChain(int index)
/// <summary> setzt die AdditionChain aufgrund des Parameters
/// <param name="l"> neue AdditionChain</param>
public void SetAdditionChain(LNList<int> l)

/// <summary> berechnet x^y mit AdditionChains </summary>
public LNumber PowFixYAdditionChain(LNumber x,LNumber y)
{ // speichert die Potenzen p[i] = x^ui
  LNList<LNumber> p = new LNList<LNumber>();
  p.Add(x);
  // fuer aktuelles Element der AdditionChain
  int[] element = new int[2];
  // Durchlaufen der AdditionChain m_chain
  for (int i=1; i<m_chain.Count/2; i++)
  {
    element = GetAdditionChain(i);
    // gleiche Werte => Quadrieren
    if (element[0] == element[1])
      p.Add(p[element[0].Square());
    // Multiplikation
    else
    p.Add(p[element[0]*p[element[1]);
  }
  return p[p.Count-1];
} #endregion }
```

5.5 Klasse in C# : AlgGGT

```
public class AlgGGT {

#region Variablen fuer k-aere Algorithmus
// TabelleG = ggt(x,k) mit x<k und I[x] = x^-1, G[x]=1 sonst 0
private LNArray<LNumber> TabelleG, TabelleI;
// alle Primzahlen p < Wurzel(k), TabelleP = p < Wurzel(k)
// und p teilt k
private LNList<int> PrimeList, TabelleP;
// speichern der Werte a,b fuer au+bv = 0 mod k
private LNArray<int> TabelleA, TabelleB;
// speichern eliminierten Teiler aus Phase 2
private LNumber elimTeiler;
private LNArray<LNumber> uv;
#endregion
```

```
#region Variablen fuer rekursiven ggT
// Grenze fuer Rekursion des rekursiven ggT
public LNumber lim;
public int prec; //maximale Bitlaenge fuer shgcd
public LNMatrix<LNumber> G; // rekursiv zu berechnende Matrix
#endregion

public AlgGGT()  // Default-Konstruktor

#region Euklidischer Algorithmus und Binaervariante
/// <summary> liefert den ggT(x,y) aufgrund des Euklidischen
/// Algorithmus zurueck </summary>
public static LNumber Euclidean(LNumber x,LNumber y)
/// <summary> berechnet den [ggT(x,y),a,b], xa+yb=ggT(x,y)
/// mit den erweiterten Euklidischen Algorithmus </summary>
/// <returns> [ggT(x,y),a,b] mit xa+yb=ggT(x,y) </returns>
public static LNMatrix<LNumber> ExtEuclidean(LNumber x,
                                             LNumber y)
public static LNumber InvEuclidean(LNumber x,LNumber y)

/// <summary> liefert den ggT(x,y) aufgrund des binaeren Euklid
/// Algorithmus von Stein zurueck </summary>
public static LNumber BinaryEuclidean(LNumber x,LNumber y)
public static LNMatrix<LNumber> ExtBinaryEuclidean(LNumber xp,
                                                   LNumber yp)
{ int beta = Math.Min(LNBit.Low2Bits(xp),LNBit.Low2Bits(yp));
  LNumber x = LNMore.Abs(xp) >> beta; // Initialisierung
  LNumber y = LNMore.Abs(yp) >> beta;
  LNMatrix<LNumber> v0 = new LNMatrix<LNumber>(3,x,1,0);
  LNMatrix<LNumber> v1 = new LNMatrix<LNumber>(3,y,y,1-x);
  LNMatrix<LNumber> t = new LNMatrix<LNumber>(3);
  if (LNBit.Low2Bits(x) > 0) // x gerade
     t.Set(0,0,x,1,0);
  else // x ungerade
     t.Set(0,0,LNMore.Negative(y),0,-1);

  while (t[0] != 0) // solange der Rest !=0
  {
    while (LNBit.Low2Bits(t[0]) > 0) // Rest gerade
    {
      if ((LNBit.Low2Bits(t[1]) > 0 || t[1].Vorzeichen == 0)&&
         (LNBit.Low2Bits(t[2]) > 0 || t[2].Vorzeichen == 0))
        t.Set(0,0,t[0]>>1,t[1]>>1,t[2]>>1);
      else
```

```
        t.Set(0,0,t[0]>>1,(t[1]+y)>>1,(t[2]-x)>>1);
    }
    if (t[0] > 0)
      v0.Set(0,0,t);
    else
      v1.Set(0,0,LNMore.Negative(t[0]),y-t[1],
    LNMore.Negative(x)-t[2]);
    t = v0 - v1;
    if (t[1] < 0)
      t.Set(0,0,t[0],t[1]+y,t[2]-x);
    }
  }
  v0[0] = v0[0] << beta;
  return v0;  }
public static LNumber InvBinaryEuclidean(LNumber x,LNumber y)
#endregion

#region Lehmervariante von Euklid
/// <summary> berechnet ggT(xp,yp) mit Lehmer-Variante
/// von Euklid mit der Grenze fuer Einfachgenauigkeit
/// M = 2^32 (Typ des Elementes von LNumber) </summary>
public static LNumber LehmerEuclidean(LNumber xp,LNumber yp)
public static LNMatrix<LNumber> ExtLehmerEuclidean(LNumber xp,
                                                   LNumber yp)
public static LNumber InvLehmerEuclidean(LNumber x,LNumber y)
/// <summary> liefert den ggT unter Verwendung der
/// modifizierten Lehmer-Variante des Euklid </summary>
/// <param name="x"> x >= y > 0</param>
/// <param name="y"> x >= y </param>
/// <param name="k"> Anzahl der Bits fuer Einfachgenauigkeit
///  kleiner 64 </param>
public static LNumber LehmerMod(LNumber x,LNumber y, int k)
/// <returns> [ggT(x,y),a,b] mit xa+yb=ggT(x,y) </returns>
public static LNMatrix<LNumber> ExtLehmerMod(LNumber x,
                                        LNumber y,int k)
public static LNumber InvLehmerMod(LNumber x,LNumber y,int k)
/// <summary> fuehrt die euklidischen Reduktionschritte des
///  modifizierten
/// Lehmer Variante in Einfachgeauigkeit aus </summary>
/// <param name="U"> Variable in Mehrfachgenauigkeit </param>
/// <param name="V"> Variable in Mehrfachgenauigkeit </param>
/// <param name="k"> Anzahl der Bits  < 64 </param>
/// <returns> Matrix in Einfachgenauigkeit [[x1,y1],[x2,y2]]
/// </returns>
```

```
public static LNMatrix<LNumber> ML(LNumber U,LNumber V,int k)
{ int h = Math.Max(U.Bitlength()-k,0) , i=2;
  // fuehrende k Stellen = Einfachgenauigkeit
  long uquer = U >> h, vquer = V>>h, q, t;
  LNMatrix<long> xy = new LNMatrix<long>(3,2,1,0,0,1,0,0);
  bool done=false;  // Variable fuer Schleifendurchlauf
  while ( !done )
  {
    q = uquer / vquer;  // Quotient
    xy.SetRow(i%3,xy.GetRow((i-2)%3)-q*xy.GetRow((i-1)%3));
    t = uquer - q*vquer;      // Reduktion
    uquer = vquer;
    vquer = t;
    ++i;
    // ueberpruefen auf Korrektheit mittels Jebelean Kriterium
    if (i%2 == 0)
         done = vquer < -xy[(i-1)%3,0] ||
             uquer-vquer < xy[(i-1)%3,1]-xy[(i-2)%3,1];
    else
         done = vquer < -xy[(i-1)%3,1] ||
             uquer-vquer < xy[(i-1)%3,0]-xy[(i-2)%3,0];
  }

  return new LNMatrix<LNumber>(2,2,xy[(i-1)%3,0],xy[(i-1)%3,1],
         xy[(i-2)%3,0],xy[(i-2)%3,1]);
} #endregion

#region k-aere Algorithmus
/// <summary> erstellt die Tabellen G und I fuer die k-aere
/// Variante des Euklidischen Algorithmus mit Hilfe des erw.
/// binaeren Alg. G[x]: ggT(x,k) und I[x] = x^-1 </summary>
public void TableGI(int k)
/// <summary> erstellt die Tabelle P ( = alle Primzahlen
///  < Wurzel(k) und p Primzahl teilt k </summary>
public void TableP(int k)
/// <summary> true falls wert prim ist, sonst false </summary>
/// <param name="wert"> zu ueberpruefende  Primzahl</param>
private bool Prime(int wert)
/// <summary> berechnet die Tabellen A und B fuer k mit der
/// Algorithmenmethode method
/// JWA.. : TabelleG und I vorberechnet </summary>
public void TableAB(int k,string method)
/// <summary> setzt die Tabellen A und B : Durchlauf
/// von a (1..wurzelk) und suche Minimum von Abs(a)+Abs(b)
```

```
/// (lt. Lemma existieren a,b, mit Abs(a)+Abs(b)
/// kleinergleich 2*wurzelk) </summary>
/// <param name="x"> Index fuer Tabelle </param>
private void FindAB(int x,int k,int wurzelk)
/// <summary> Jebelean-Weber-Algorithmus
/// berechnet die a,b fuer den Quotienten von x/y </summary>
/// <param name="x"> x>0, ggT(x,k) = 1</param>
/// <param name="y"> y>0, ggT(y,k) = 1</param>
/// <param name="wurzelk"> Wurzel von k</param>
private int[] JWA(int x,int y,int k,int wurzelk)
/// <summary> JWA mit T-Transformation </summary>
private int[] JWAT(int x,int y,int k,int wurzelk)
/// <summary>liefert die T-Transformation von (x,y) zurueck
/// ueberprueft ob x,y in A_k = (0,wurzelk)
/// oder in B_k=(k-wurzelk,k)  </summary>
private int[] TransformT(int x,int y,int k,int wurzelk)
/// <summary> eliminiert die moeglichen Teiler von uv aus
///  Tabelle P </summary>
public void EliminateDivisorsBegin(LNumber x,LNumber y)
/// <summary> Reduktion des k-aeren Algorithmus </summary>
public void KaryReduction(int k)
/// <summary> eliminiert die moeglichen Teiler </summary>
/// <returns> liefert ggT(u,v) zurueck</returns>
public LNumber EliminateDivisorsEnd()
/// <summary> fuehrt Vorberechnung fuer k-aere Alg</summary>
public void KaryPrecomputation(int k)
/// <summary> liefert ggT(x,y) des k-aeren Alg </summary>
public LNumber KaryGCD(LNumber x, LNumber y,int k)
#endregion

#region spezielle Inversionsalgorithmen
/// <summary> berechnet das Inverse x^-1 fuer x in Zp
/// Vorraussetzung : x und relativ prim</summary>
public static LNumber InvSpecialEuclidean(LNumber x,
                                          LNumber p)
/// <summary> berechnet x^-1 in ZMp fuer Mp </summary>
public static LNumber InvSpecialMersenne(LNumber x,
                                 LNumber Mp,int p)
{ LNumber a = new LNumber(1,1), an = new LNumber(0,0);
  LNumber y = new LNumber(1,x), yn = new LNumber(1,Mp);
  LNumber t = new LNumber();
  int e=0;
  while ( yn != 1)
  { e = LNBit.Low2Bits(y); // Eliminieren um 2^e
```

```
    y = y>>e;
    if (a>Mp)
    { // a>Mp => 1 Reduktionsschritt mod Mp
      a = LNBit.And(a,Mp)+(a>>p);    }
    a = LNBit.LeftCircularShift(a,p-e,p); //=2^(p-e)*a mod Mp
    t = a; a = a + an; an = t;
    t = y; y = y + yn; yn = t;
  }
  return an;
}
#endregion

#region rekursiver ggT fuer sehr grosse Zahlen
/// <summary> berechnet den gcd mit Schoenhage-Strassen
/// d.h. rekursiver ggT nur fuer sehr grosse Zahlen mit
/// den beiden Grenzen
/// explim und expprec zur Effizienzsteigerung</summary>
/// <param name="explim"> fuer Zahlen kleiner 2^explim,
///    Standardmethode</param>
/// <param name="expprec"> Grenze fuer die Anzahl Bits</param>
/// <returns> gcd(x,y) </returns>
public LNumber RecursivGcd(LNumber x,LNumber y,
                           int explim,int expprec)
/// <summary> Hauptfunktion des rekursiven gcd, ruft
/// Funktion hgcd und Hilfsfunktion cgcd auf </summary>
/// <returns> ggT(x,y) </returns>
private LNumber rgcd(LNumber x,LNumber y)
/// <summary> rekursive Funktion welche die Matrix G
public void hgcd(int b,LNumber x,LNumber y)
/// <summary> zur Berechnung von G fuer kleine x,y </summary>
/// <returns> neues G </returns>
private LNMatrix<LNumber> shgcd(LNumber x,LNumber y)
/// <summary> berechnet den gcd fuer small (d.h. kleiner lim)
///  Terme mit der Standardmethode Euclidean </summary>
private LNumber cgcd(LNumber x,LNumber y)
#endregion  }
```

5.6 Klasse in C# : AlgMult

```
public class AlgMult {

public AlgMult()
/// <summary> uv mittels additiven Potenzieren </summary>
public static LNumber PowAdd(LNumber u,LNumber v)
```

```
/// <summary> berechnet uv mod N mittels binaeren mul-mod
/// Vorraussetzungen: 0 <= u,v < N </summary>
/// <param name="u"> u kleiner N </param>
/// <param name="v"> v kleiner N </param>
/// <param name="N"> mod N  </param>
/// <returns> liefert uv mod N zurueck </returns>
public static LNumber PowAddMod(LNumber u, LNumber v,
                                LNumber N)

/// <summary> fuehrt eine Multiplikation nach rekursiver
/// Karatsuba-Methode durch </summary>
/// <returns> x*y </returns>
public static LNumber Karatsuba(LNumber x,LNumber y)
{ // Zahlen unterhalb der Grenze
  if (x.Count() <=1 && y.Count() <= 1)    return x*y;
  else
  { int l, r=Math.Max(x.Count(),y.Count());
    if (r % 2 == 0)     l = r/2;
    else
      l = (r-1)/2;
    // Aufteilen in Haelften
    LNumber x1 = Part(x,l,r-1), x0 = Part(x,0,l-1);
    LNumber y1 = Part(y,l,r-1), y0 = Part(y,0,l-1);
    // rekursive Aufrufe
    LNumber res= Karatsuba(x0,y0);
    LNumber erg = res<<(l*32);              erg +=res;
    res = Karatsuba(x1-x0,y0-y1)<<(l*32);  erg +=res;
    res = Karatsuba(x1,y1);
    erg += res<<(64*l);           erg += res<<(32*l);
    return erg;
  }
}

/// <summary>  Teilzahl von z[anf,..ende] zurueck </returns>
private static LNumber Part(LNumber z,int anf,int ende)
/// <summary> fuehrt den Tom-Cook-Algorithmus mit Basis B=2^32
/// zur Berechnung von x*y </summary>
public static LNumber TomCook(LNumber x,LNumber y)
{ // D-Stellen = Maximum
  int D = Math.Max(x.Count(),y.Count());
  int anzStellen = 2*D-1, i,j,k;
  LNArray<LNumber> xt = new LNArray<LNumber>(anzStellen,0);
  LNArray<LNumber> yt = new LNArray<LNumber>(anzStellen,0);
  LNArray<LNumber> zt = new LNArray<LNumber>(anzStellen,0);
```

```
  LNArray<LNumber> Wx = new LNArray<LNumber>();
  LNumber pwert = new LNumber();

  // Auswerten an 2D-1 Stellen : 0,1,2,...2D-1
  xt[0] = x[0]; yt[0] = y[0];
  for (i=1; i<anzStellen; i++)
  {
    for (j=0; j<D; j++)
    {
      pwert = i^j;
      xt[i] += x[j]*pwert;
      yt[i] += y[j]*pwert;
    }
  }
  for (i=0; i<anzStellen; i++)     // Berechnung der z(t)
    zt[i] = xt[i]*yt[i];
  // Differenzenschema fuer Koeffizienten
  // fuer faktorielle Potenzen
  for (i=0,k=1; i<zt.Count-1; i++,k++)
    for (j=zt.Count-1; j>i; j--)
      zt[j] = zt[j] - zt[j-1])/k;
  // Berechnung von faktoriellen Potenzen zu B^i
  LNArray<LNumber> z = new LNArray<LNumber>(anzStellen,0);
  for (i=0; i<anzStellen; i++)
  {
    Wx = FactorialPow(Wx,i);
    for (j=0; j<Wx.Count; j++)
      z[j] += Wx[j]*zt[i];
  }
  // Speichern der Koeffzienten (+uebertrag)
  LNumber res = new LNumber(0,0);
  for (i=0; i<anzStellen; i++)
    res += (z[i]<<(32*i));
  res.Vorzeichen = x.Vorzeichen * y.Vorzeichen;
  return res;  }

/// <summary> berechnet faktorielle Potenz
/// x^k = x*(x-1)*..x(-k+1), rekursiv aus x^k-1</summary>
/// <param name="w"> Liste von Koeffizienten fuer x^k-1 </param>
/// <param name="k"> Exponent > 0 </param>
/// <returns> Liste von Koeffizienten fuer x^k </returns>
private static LNArray<LNumber> FactorialPow
       (LNArray<LNumber> w, int k)     }
```

5.7 Klasse in C# : LNumber

```
public class LNumber {

#region Daten
// beinhaltet die Zahl als Liste mit Elementen zur Basis 2^32
private uint[] FeldZahl;
// Vorzeichen der Zahl
private int FeldVorzeichen;
// Laenge der belegten Elemente
private int FeldLaenge;
// Hilfsvariable fuer Subtraktion
private long Grenzwert = (long)uint.MaxValue +1; #endregion

#region Konstruktoren
// Default Konstruktor : Initialisierung von FeldZahl
public LNumber()
{ FeldZahl = new uint[1];
  FeldLaenge = 0;  }
// Konstruktor : Initialisierung mit anz Element = 0
public LNumber(int anz)
// Konstruktor fuer elementweise Zuweisung uint
// wertvorzeichen : Vorzeichen der Zahl
// wert : Parameter uint-Elemente fuer Zahl
public LNumber(int wertvorzeichen,params uint[] wert)
// Konstruktor : konvertiert Parameter string zu LNumber
public LNumber(string paramstr)
/// <summary> Copy-Konstruktor : mit LNumber </summary>
public LNumber(LNumber zahl)#endregion

#region Manipulation
// setzt oder liefert index-te Element der FeldZahl
public uint this [ int index ]
// setzt oder liefert das Vorzeichen der FeldZahl
public int Vorzeichen
// liefert die Laenge der FeldZahl
public int Count()
// reduziert fuehrende Nullen
public void Reduce(int maxlaenge)
/// kopiert die Elemente zahl[zanf,..,zende]
/// in aktuelle LNumber beginnend bei Index [anf]
public void Copy(LNumber zahl,int zanf,int zende,int anf)
// liefert die Anzahl der Bits von FeldZahl
public int Bitlength()
```

```
// liefert Bit der aktuellen Position
public uint Bit(int index)
// NotANumber : ueberprueft ob die aktuelle Zahl definiert ist
// true = undefiniert, sonst false
public bool IsNaN()
//ueberprueft ob die aktuelle Zahl = 0 ist
public bool IsZero()
#endregion
public override bool Equals(object obj)
public override int GetHashCode()
public override string ToString()

#region    arithmetische Operatoren
public static LNumber operator + (LNumber sum1,LNumber sum2)
public static LNumber operator - (LNumber sub1,LNumber sub2)
public static LNumber operator * (LNumber mult1,LNumber mult2)
public static LNumber operator / (LNumber wert,LNumber teiler)
public static LNumber operator % (LNumber wert,LNumber teiler)
// berechnet den Rest von wert durch teiler und
// liefert Ergebnis zurueck (auch negative Reste zugelassen)
public void Mod(LNumber wert,LNumber teiler)
//berechnet eine Potenz mit Hilfe des Square and Multiply
public static LNumber operator ^(LNumber basis, LNumber
  exponent)
// liefert das Quadrat
public LNumber Square()
public static LNumber operator >>(LNumber zahl, int anzbits)
public static LNumber operator << (LNumber zahl, int anzbits)
#endregion

#region Vergleiche
// true, falls aufgerufener Vergleich zutrifft, sonst false
public static bool operator > (LNumber v1, LNumber v2)
public static bool operator < (LNumber v1, LNumber v2)
public static bool operator >= (LNumber v1, LNumber v2)
public static bool operator <= (LNumber v1, LNumber v2)
public static bool operator == (LNumber v1, LNumber v2)
public static bool operator != (LNumber v1, LNumber v2)
// bestimmt Vergleichsrelation zwischen FeldZahl und vergleich
// ohne Vorzeichen </summary>
// Parameter zum vergleichen
//    1  fuer FeldZahl groesser vergleich,
//    0  fuer FeldZahl = vergleich,
//   -1  fuer FeldZahl kleiner vergleich </returns>
```

```
public int CompareAbsolut(LNumber vergleich)
// bestimmt Vergleichsrelation mit Vorzeichen
public int Compare(LNumber vergleich)
#endregion

#region casten
//Konvertiert uebergebene Parameter in LNumber
public static implicit operator LNumber (string paramstr)
public static implicit operator LNumber (ulong paramzahl)
public static implicit operator LNumber (uint paramzahl)
public static implicit operator LNumber (long paramzahl)
public static implicit operator LNumber (int paramzahl)
// konvertiert LNumber auf den jeweiligen Typen
public static implicit operator uint (LNumber paramLargeZahl)
public static implicit operator int (LNumber paramLargeZahl)
public static implicit operator ulong (LNumber paramLargeZahl)
public static implicit operator long (LNumber paramLargeZahl)
#endregion  }
```

5.8 Klasse in C# : Kernel

```
/// <summary> low-level-Functions for LNumber </summary>
public sealed partial class Kernel  {
/// <remarks> This code is unsafe!
/// It is the caller's responsibility to make sure that it is
/// safe to access x [xOffset:xOffset+xLen],y [yOffset:
/// yOffset+yLen], and d [dOffset:dOffset+xLen+yLen]</remarks>

#region Addition and Subtraction
/// <summary> addiert die Daten x [xOffset:xOffset+xLen] und
/// y [yOffset:yOffset+yLen] und speichert das Resultat in
/// d [dOffset:dOffset+xLen+1] </summary>
public static unsafe void Addition(uint[] x,uint xOffset,uint
  xLen,uint[] y,uint yOffset, uint yLen,uint[] d,uint dOffset)
{ fixed(uint* xx= x, yy=y, dd=d) {
    uint *dP = dd+dOffset, bP,bE,sP,sE; // Zeiger dP=Ergebnis
    // Zeiger : b = bigger Number und s = smaller Number
    if (xLen >= yLen )
    { bP = xx + xOffset;  bE = bP + xLen;
      sP = yy + yOffset;  sE = sP + yLen;  }
    else
    { bP = yy + yOffset;  bE = bP + yLen;
      sP = xx + xOffset;  sE = sP + xLen; }
```

```
    bool notequal = (dP != bP);        // true falls kein +=
    ulong mcarry = 0;
    for (; sP < sE; sP++,bP++,dP++) // Durchlauf der < Zahl
    { mcarry += (ulong)*sP + *bP;
      *dP = (uint)mcarry;
      mcarry >>= 32;   }
    // uebertrag fuer restliche Elemente der groesseren Zahl
    for (; mcarry != 0 && bP < bE; bP++,dP++)
    { mcarry += *bP;
      *dP = (uint)mcarry;
      mcarry >>= 32;  }
    // Anfuegen eines Elementes = uebertrag
    if (mcarry != 0)    *dP = (uint)mcarry;
    // Kopieren bzw. bei bP += sP fertig
    if (notequal)
      for (; bP < bE; bP++,dP++)        *dP = *bP;}

/// <summary> subtrahiert von b [bOffset:bOffset+xLen],y
/// [yOffset:yOffset+yLen] und speichert das Resultat in d[dO
/// ffset:dOffset+xLen] !! x > y (sonst Fehler) </summary>
public static unsafe void Subtraction(uint[] b,uint bOffset,
  uint bLen,uint[] s,uint sOffset, uint sLen,
  uint[] d,uint dOffset)
#endregion
#region Shiften
/// <summary> Shiftet die Daten im Bereich von x [xOffset:
/// xOffset+xLen] um n (>0) Bits nach links und speichert das
/// Resultat in d [dOffset:dOffset+xLen+1]. </summary>
public static unsafe void LeftShift (uint [] x, uint xOffset,
   uint xLen,int n, uint [] d, uint dOffset)
/// <summary> Shiftet die Daten im Bereich von x [xOffset:
/// xOffset+xLen] um n (>0) Bits nach rechts und speichert
/// das Resultat in d [dOffset:dOffset+xLen]. </summary>
public static unsafe void RightShift (uint [] x, uint xOffset,
  uint xLen,int n, uint [] d, uint dOffset)
#endregion

#region Multiplication
/// <summary> Multipliziert die Daten in von x [xOffset:
/// xOffset+xLen] mit dem Bereich y [yOffset:yOffset+yLen] und
/// speichert das Resultat x*y in d [dOffset:dOffset+xLen+yLen]
public static unsafe void Multiply (uint [] x, uint xOffset,
  uint xLen,uint [] y, uint yOffset, uint yLen,
  uint [] d, uint dOffset)
```

```
{ fixed (uint* xx = x, yy = y, dd = d)  {
    uint* xP = xx + xOffset,       // Durchlaufadresse von x
    xE = xP + xLen,               // Endadresse von x
    yB = yy + yOffset,      // Beginnadresse von y
    yE = yB + yLen,         // Endadresse von y
    dB = dd + dOffset;      // Beginnadresse des Resultat x*y

  for (; xP < xE; xP++, dB++)
  { // Durchlauf fuer Teilmultiplikation
    // keine Multiplikation noetig d.h. weiterspringen
    if (*xP == 0) continue;

    ulong mcarry = 0;
    uint* dP = dB;
    for (uint* yP = yB; yP < yE; yP++, dP++)
    {// Teilmultiplikation von xp mit y
      mcarry += ((ulong)*xP * (ulong)*yP) + (ulong)*dP;
      *dP = (uint)mcarry;
      mcarry >>= 32;  }
      if (mcarry != 0)    *dP = (uint)mcarry;
    }
  }
}

/// <summary> Multipliziert die Daten von x [xOffset:
/// xOffset+xLen] mit dem Bereich y [yOffset:yOffset+yLen] und
/// das  x*y in x [xOffset:xOffset+xLen+yLen]. </summary>
public static unsafe void MultiplyEqual (ref uint [] x, uint
  xOffset, ref uint xLen, uint [] y, uint yOffset, uint yLen)
/// mit uint-Wert y und speichert das
/// Resultat x*y in d [dOffset:dOffset+xLen+1]. </summary>
public static unsafe void MultiplySmall (uint [] x, uint
  xOffset, uint xLen,uint y, uint [] d,uint dOffset)
#endregion
#region Squaring
/// <summary> Bildet das Quadrat der Daten im Bereich von x
/// [xOffset:xOffset+xLen]
/// speichert das Resultat in z [zOffset:zOffset+2*xLen].
public static unsafe void Square(uint[] x,uint xOffset,uint
  xLen,uint[] z,uint zOffset)
/// <summary> Bildet das Quadrat der Daten im Bereich von x
/// [xOffset:xOffset+xLen]
/// speichert das Resultat in x [xOffset:xOffset+2*xLen].
public static unsafe void SquareEqual (ref uint [] x, uint
```

```
  xOffset,ref uint xLen)
#endregion

#region Division, Compare, Shift of one
/// <summary> dividiert x durch d, z = Quotient und r = Rest
public static unsafe void DivideModSmall(uint [] x, uint
  xOffset, uint xLen, uint d,
  uint [] z, uint zOffset, out uint r)
/// <summary> liefert eine Vergleichsrelation  </summary>
public static Signum Compare(uint[] x,ref uint xLen,uint[] y,
  ref uint yLen)
/// <summary> Shiftet die Daten im Bereich von x [xOffset:
/// xOffset+xLen+1] um 1 Bit nach rechts. </summary>
public static unsafe void RightShiftOne (ref uint [] x,
  uint xOffset,ref uint xLen)
/// <summary> Shiftet die Daten im Bereich von x [xOffset:
/// xOffset+xLen] um 1 Bit nachlinks und speichert Resultat
/// in d [dOffset+xLen]. </summary>
public static unsafe void LeftShiftOne (uint [] x, uint
  xOffset,uint xLen, uint[] d,uint dOffset)
#endregion }
```

5.9 Klasse in C# : LNMore

```
public class LNMore {

private Random zufall;
private int m_correct;
public LNMore() /// <summary> Konstruktor </summary>

#region Zufallsgenerator fuer LNumber
/// <summary> initialisiert und definiert die Randomvariable
/// zufall fuer die Zufallszahlenerzeugung </summary>
public void RandomSeed( int seed )
/// <summary> liefert eine zufaellige Zahl LNumber indem die
/// einzelne Elemente mit der Randomvariable zufall erzeugt
/// werden </summary>
public LNumber Random(int anz)
#endregion

#region Befehlszeileninterpreter
/// <summary> ueberprueft die Befehlszeile auf korrekte Syntax
/// und liefert das Ergebnis mittels Interpreter(str) zurueck
/// nur Zahlen und Operatoren (+-/*^!) erlaubt hoechste Ord:
```

```
/// ^! dann */ und niedrigste Ordnung +- </summary>
/// <param name="str"> zu ueberpruefende Befehlszeile </param>
/// <returns> korrekte Syntax : Resulat, Fehler:NaN </returns>
public LNumber BefehlToLNumber(string str)
/// <summary> rekursive Funktion zur Berechnung der Befehls-
/// zeile => bei Multiplikation und Division werden die Terme
/// sukzessive,Potenzen rekursiv, Fakultaet u. Zahl aufgeloest
/// <param name="str"> zu berechnende Befehlszeile </param>
/// <returns> liefert das Resultat zurueck </returns>
private LNumber Interpreter(string str)
#endregion

#region zusaetzliche arithmetische Funktionen
/// <summary> liefert das Maximum der Parameter = LNumber
/// <param name="paramlist"> Anzahl von LNumber </param>
public static LNumber Max(params LNumber[] paramlist)
/// <summary> liefert das Minimum der Parameter = LNumber
/// <param name="paramlist"> Anzahl von LNumber </param>
public static LNumber Min(params LNumber[] paramlist)
/// <summary> liefert -zahl zurueck </summary>
public static LNumber Negative(LNumber zahl)
/// <summary> eine Kopie von zahl mit Absolutbetrag zurueck
/// 1 Parameter : Vorzeichen auf + aendern
/// 2 Parameter : x mod N => x>0 aendern </summary>
/// <param name="zahl"> zu kopierende Zahl </param>
/// <returns> Absolutbetrag(zahl) </returns>
public static LNumber Abs(params LNumber[] paramzahl)
/// <summary> berechnet x mod N, N>0 fuer x > 0 </summary>
public static LNumber AbsMod(LNumber x,LNumber N)
/// <summary> berechnet n! und setzt ZahlenListe </summary>
public static LNumber Factorial(int n)
#endregion }
```

5.10 Klasse in C# : LNBit

```
public class LNBit {

private BitArray bitListe;
private int bitLaenge;

/// <summary> Standardkonstruktor </summary>
public LNBit()
/// <summary> Konst.mit anz Elemente = wert => = 0 </summary>
public LNBit(int anz, bool wert)
```

```
/// <summary> liefert die Anzahl der Bits zurueck bzw. setzt
/// die Anzahl der Bits  </summary>
public int Bitlength
{ get { return bitLaenge; }
  set { bitListe.Length = value;
  bitLaenge = value;  } }
public override bool Equals(object obj)
public override int GetHashCode()
public override string ToString()
/// <summary> set, get das Bit von bitListe[index] </summary>
public uint this [ int index ]
/// <summary> liefert oder setz die Bits[beginn...ende] als
/// uint-Repraesentation zurueck  </summary>
public uint this [ int beginn, int ende ]
/// <summary>  bool-Wert von bitListe[index] </summary>
public bool GetBool(int index)
/// <summary> liefert die Bits[beginn...beginn+laenge-1]
/// <param name="beginn"> Startindex </param>
/// <param name="laenge"> Laenge der Bits
///             negativ: rechts von Startindex
///    positiv: links von Startindex </param>
/// <returns>liefert den uint-wert zurueck</returns>
public uint Get(int beginn, int laenge)
/// <summary> wandelt [beginn...ende] in eine LNumber um
/// und liefert sie zurueck </summary>
public LNumber ToLNumber(int beginn, int ende)
public LNumber ToLNumberAll()
/// <summary> setzen des Bitarray BitListe = wert </summary>
public void Set(int wert)
public void Set(uint wert)
public void Set(LNumber zahl)
/// <summary> sucht die Bitfolge find in bitListe </summary>
/// <param name="beginn"> Startindex zu suchen </param>
/// <param name="richtung"> negativ: rechts von Startindex,
///   positiv: links von Startindex </param>
/// <param name="laenge"> max. Anzahl von Stellen </param>
/// <param name="find"> zu suchendes LNBit </param>
/// <returns> index >0 mit bitListe[index,index+1,..] = tofind,
///         sonst -1 </returns>
public int SearchBits(int beginn, int richtung, int laenge,
  LNBit find)

#region statische Bitoperationen
/// <summary> berechnet die logische Operation & zwischen
```

```
/// term1 und term2 und liefert das Resultat zurueck </summary>
/// <param name="term1,term2"> Paramtertyp: LNumber </param>
public static LNumber And(LNumber term1, LNumber term2)
/// <summary> berechnet die logische Operation | zwischen
/// term1 und term2 und liefert das Resultat zurueck </summary>
/// <param name="term1,term2"> Paramtertyp: LNumber </param>
public static LNumber Or(LNumber term1, LNumber term2)
/// <summary> berechnet die logische Operation XOR zwischen
/// term1 und term2 und liefert das Resultat zurueck </summary>
/// <param name="term1,term2"> Paramtertyp: LNumber </param>
public static LNumber Xor(LNumber term1, LNumber term2)
/// <summary> berechnet ! von t und liefert zurueck </summary>
/// <param name="term"> Parametertyp: LNumber</param>
public static LNumber Not(LNumber t)

/// <summary>  zirkulaeren Linksshift durch </summary>
/// <param name="zahl"> zu shiftende Zahl </param>
/// <param name="anzbits"> anzbits nach links shiften </param>
/// <param name="stellenbits"> Bereich wird geshiftet </param>
public static LNumber LeftCircularShift(LNumber zahl,
  int anzbits, int stellenbits)
/// <summary> zirkulaeren Rechtsshift durch </summary>
public static LNumber RightCircularShift(LNumber zahl,
  int anzbits, int stellenbits)
/// <summary> liefert die Anzahl von niedrigen Bits = 0
/// = 2^v teilt zahl und 2^(v+1) teilt nicht zahl </summary>
/// <returns> hoechste 2-Potenz in Zahl</returns>
public static int Low2Bits(LNumber zahl)
#endregion  }
```

5.11 Klasse in C# : LNArray, LNMatrix, LNList

```
public class LNArray<T> : object where T:new()     {

// speichert anzahlFeld von T Elementen
protected T[] numberFeld;
protected int anzahlFeld;

public LNArray()         // Default-Konstruktor
/// <summary> Konstruktor mit anz von Nullelementen </summary>
public LNArray(int anz)
/// <summary> Konstruktor mit Parametern vom Typ T </summary>
public LNArray(int anz,params T[] init)
/// <summary> Copy-Konstruktor </summary>
```

```
public LNArray(LNArray<T> kopie)

/// <summary> liefert bzw. setzt die Laenge Arrays </summary>
public int Count
public T this[int index]
public override bool Equals(object obj)
public override int GetHashCode()
public override string ToString()
/// <summary> set des Arrays beginnend bei anfang </summary>
public void Set(int anfang,params T[] init)
public void Set(int anfang,LNArray<T> kopie)
public void SetAll(T wert)
/// <summary> eine Kopie von Array[imin,..imax] </summary>
public LNArray<T> CopyTo(int imin,int imax)
public LNArray<T> CopyTo()
/// <summary> liefert einen Index bzgl. des Typs T </summary>
public int GetTypeInt() }

/// <summary> Matrix bzw. Vektoroperationen fuer die Typen
///    int, long, double und LNumber </summary>
public class LNMatrix<T> : LNArray<T> where T:new()    {
// Parameter fuer die Zeilen und Spalten
private int zeilenAnzahl, spaltenAnzahl;
#region Konstruktoren
public LNMatrix() : base()    {    }
public LNMatrix(int z) : base(z)    { SetParam(z,1); }
public LNMatrix(int z,int s) : base(z*s)    { SetParam(z,s); }
public LNMatrix(int z,params T[] init) : base(z,init)
  { SetParam(z,1); }
public LNMatrix(int z,int s,params T[] init) : base(z*s,init)
  { SetParam(z,s); }
public LNMatrix(LNMatrix<T> kopie)
public LNMatrix(int z, int s,LNArray<T> kopie) : base(kopie)
  { SetParam(z,s); } #endregion

public override bool Equals(object obj)
public override int GetHashCode()
public override string ToString()

/// <summary> liefert die Zeilenanzahl zurueck </summary>
public int Row
/// <summary> liefert die Spaltenanzahl zurueck </summary>
public int Column
/// <summary> liefert bzw. setzt [z,s] Element</summary>
```

```
public T this[int z,int s]
/// <summary> setzt die Parameter des Arrays </summary>
private void SetParam(int zinit,int sinit)
/// <summary> setzen des Arrays beginnend bei [z,s] </summary>
public void Set(int z,int s,params T[] init)
public void Set(int z,int s,LNArray<T> kopie)
/// <summary> setzt die Zeile i </summary>
public void SetRow(int i,params T[] init)
public void SetRow(int i,LNArray<T> kopie)
public void SetColumn(int i,params T[] init)
public void SetColumn(int i,LNArray<T> kopie)
/// <summary> liefert die Zeile[i] zurueck  </summary>
public LNMatrix<T> GetRow(int i)
public LNMatrix<T> GetColumn(int i)
/// <summary> liefert eine Matrix mit mit oberen rechten Ende
///  [z1,s1]und linkere unteren Ende [z2,s2] </summary>
public LNMatrix<T> GetMatrix(int z1,int s1,int z2,int s2)
// Casten
public static implicit operator LNMatrix<int>(LNMatrix<T> l)
public static implicit operator LNMatrix<long>(LNMatrix<T> l)
public static implicit operator LNMatrix<double>(LNMatrix<T>l)
public static implicit operator LNMatrix<LNumber>(LNMatrix<T>l)
//  sowie alle Operationen +,-,*,/ mit einem Skalar
//  und die Matrixmultiplikation ^ als Beispiel + und Typ int
public static LNMatrix<int> operator + (LNMatrix<int> sum1,
                                        LNMatrix<T> sum2)
public static LNMatrix<int> operator + (LNMatrix<T> sum,
                                          int skalar)
public static LNMatrix<int> operator + (int skalar,
                                          LNMatrix<T> sum) }


public class LNList<T> : object where T:new() {

// generische Liste
private List<T> numberList;
/// <summary> Default-Konstruktor </summary>
public LNList()

/// <summary> Konstruktor mit anz von Nullelementen </summary>
public LNList(int anz)
/// <summary> Konstruktor mit Parametern vom Typ T </summary>
public LNList(params T[] paramzahl)
/// <summary> Copy-Konstruktor </summary>
```

```
public LNList(LNList<T> kopie)

/// <summary> liefert bzw. setzt die Laenge der Liste
/// Setzen: loescht Elemente bzw. fuegt sie ein bis
/// korrekte Laenge erreicht ist </summary>
public int Count
/// <summary> get or set index-te Element </summary>
public T this[int index]
public override bool Equals(object obj)
public override int GetHashCode()
public override string ToString()
public string ToString(int index)
/// <summary> setzt die aktuelle Liste auff l </summary>
public void Set(LNList<T> l)
/// <summary> fuegt Elemente vom Typ T am Ende ein  </summary>
public void Add(params T[] paramzahl)
public void Add(LNList<T> l)
/// <summary> fuegt T an der Stelle index ein </summary>
public void Insert(int index, params T[] paramzahl)
public void Insert(int index, LNList<T> l)
/// <summary> loescht das index-te Element </summary>
public void Remove(int index)
public void RemoveRange(int index, int laenge)
public void Clear()
/// <summary> liefert eine Kopie von der Liste
///  beginnend bei anfang bis ende</summary>
public virtual LNList<T> CopyTo(int anfang,int ende)
public LNList<T> CopyTo()
/// <summary> liefert den Index von ersten Auftreten von
/// paramzahl in der Liste zurueck </summary>
/// <param name="paramzahl"> zu suchende T </param>
/// <param name="startindex"> Beginn der Suche (=0)</param>
/// <param name="laenge"> Laenge des Suchbereichs </param>
/// <returns> liefert den Index, sonst -1 </returns>
public int IndexOf(T paramzahl,int startindex,int laenge)
public int IndexOf(T paramzahl,int startindex)
public int IndexOf(T paramzahl)
/// <summary> liefert den Index von letzten Auftreten von
/// paramzahl in der Liste zurueck </summary>
/// <param name="paramzahl"> zu suchende T </param>
/// <param name="startindex"> Beginn der Suche (=0)</param>
/// <param name="laenge"> Laenge des Suchbereichs </param>
/// <returns> liefert den Index, sonst -1 </returns>
public int LastIndexOf(T paramzahl,int startindex,int laenge)
```

```
public int LastIndexOf(T paramzahl,int startindex)
public int LastIndexOf(T paramzahl)
/// <summary> sortiert die Liste [start,start+laenge-1]
/// aufgrund von pcomp </summary>
/// <param name="start"> Beginn des Bereichs </param>
/// <param name="laenge"> Laenge des Bereichs </param>
public void Sort(int start, int laenge, LNComparer<T> pcomp)
/// <summary> sortiert die gesamte Liste nach pcomp </summary>
/// <param name="pcomp"> Sortierung aufgrund von
/// LNComparer paramcomp </param>
public void Sort(LNComparer<T> pcomp) }

public class LNComparer<T> :
  System.Collections.Generic.IComparer<T> where T:new() {
// gibt an ob aufsteigend = 1 oder absteigend = -1 sortiert
private int Reihenfolge;
// mit Vorzeichen = true oder ohne (Absolutbetrag) sortieren
private bool Absolutbetrag;

// Konstruktor r = true aufsteigend sonst absteigend sortieren
//             a = true, Absolutbetrag
public LNComparer(bool r,bool a)
/// <summary> fuehrt einen Vergleich bzgl den
/// Absolutbetraegen von x und y </summary>
public int ComparisonTypeAbsolut(T x, T y)
// ueberschriebene Methode von IComparer fuer Sortierung
int IComparer<T>.Compare( T x, T y )        }
```

6 Literaturverzeichnis der Referenzen:

1. Richard. E. Crandall und Carl Pomerance, *Prime Number. A Computational Perspective*, Springer Verlag

2. A.J. Menezese, P.C. van Oorschot and S.A. Vanstone, *Handbook of applied Cryptography*, CRC Press, www.cacr.math.uwaterloo.ca/hac

3. Johann Wiesenbauer, Ao.Univ.Prof.Dipl.-Ing.Dr.techn. an der TU - Wien, Skriptum zur Vorlesung Analyse von Algorithmen

4. D.E. Knuth, *The Art of Computer Programming, Seminumerical algorithms*, volume 2, Addison-Wesley, Reading, Mass, 3rd Edition, 1998

5. Jonathan Sorenson, *Two fast GCD algorithms*, Journal of Algorithms, 16:110-114, 1994

6. Jonathan P. Sorenson, *Lehmer's algorithm for very large numbers*, ANTS VI poster presentation; abstract appeared in SIGSAM Bulletin 38,3:102-104,2004.

7. Jonathan P. Sorenson, *An Analysis of the Generalized Binary GCD Algorithm*,High Primes and Misdemeanors: Lectures in Honour of the 60th Birthday of Hugh Cowie Williams, Alf van der Poorten and Andreas Stein ed., Banff, Alberta, Canada, 2004.

8. Jonathan P. Sorenson, *An analysis of Lehmer's Euclidean GCD algorithm*, 1995 ACM International Symposium on Symbolic and Algebraic Computation, A. H. M. Levelt ed., Montreal, Canada, pages 254-258, 1995. ACM Press

9. Jonathan Sorenson, *The K-ary GCD Algorithm*, Computer Sciences Technical Report, November 1990

10. Kenneth Weber, *The Accelerated Integer Algorithm*, ACM Transactions on Mathematical Software, Vol. 21, No. 1, März 1995, 111-122

11. J.W. Cooley and J.W. Tukey, *An algorithm for machine calculation of compley Fourier series*, Math. Comp. 19 (1965)

12. A.Schönhage. *Schnelle Berechnung von Kettenbruchentwicklungen*, Acta Informatica, 1971

13. A.Schönhage und V.Strassen, *Schnelle Multipliaktion grosser Zahlen*, Computing. 1971

14. D.H. Bailey, *A High Performance Fast Fourier Transform Algorithm for the Cray 2*, Journal of Supercomputing, vol. 1 (1987)

15. D.H. Bailey, *A High Performance FFT Algorithm for Vector Supercomputers*, International Journal of Supercomputer Applications, vol. 2 (1988)

16. D.H. Bailey, *FFTs in external or hierarchical memory*, Journal of Supercomputing, vol 4. no. 1, 1990

17. Richard E.Crandall, Ernst W. Mayer, Jason S. Papadopoulos , *The twenty-fourth fermat number is composite*, Mathematics of Computation archive, Volume 72 , Issue 243 : 1555 - 1572 , (July 2003), American Mathematical Society

18. H.J. Nussbaumer, *Linear Filtering Technique for Computinger Mersenne und Fermat Number Transforms*, IBM J.Res. Develop. 334-339, Juli 1997