Polynomarithmetik: Multiplikation

Helge Janetzko

Seminar: Computeralgebra

bei Herrn Prof. Dr. Iwanowski

Inhalt

1. Polynome im Überblick
2. Multiplikation: Der Karatsuba-Algorithmus
3. Schnelle Multiplikation mit FFT

Inhalt

1. Polynome im Überblick

- Definition Polynom:
  Summe von Vielfachen von Potenzen einer Variable x

$$a(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = \sum_{k=0}^{n} a_k x^k$$

- Grad des Polynoms ($\deg(a)$):
  höchster Exponent n, bei dem der Koeffizient $a_n \neq 0$
- Führender Koeffizient:
  Koeffizient vor höchstem Exponenten (a_n, wenn $a_n \neq 0$)

1. Polynome im Überblick

- Addition:
  - $\deg(a+b) \leq \max(\deg(a), \deg(b))$

$$\sum_{k=0}^{n} a_k x^k + \sum_{k=0}^{m} b_k x^k = \sum_{k=0}^{\max(m,n)} \left(a_k + b_k\right) x^k$$

- Multiplikation
  - $\deg(a \cdot b) = \deg(a) + \deg(b)$

$$\sum_{k=0}^{n} a_k x^k \cdot \sum_{k=0}^{m} b_k x^k = \sum_{k=0}^{m+n} c_k x^k$$

1. Polynome im Überblick

- Additions- und Multiplikationsalgorithmen von Zahlen auch auf Polynome anwendbar
  - Weglassen der Überträge:
    - Zahlen:

$$\begin{array}{rcc} & 7 & 5 \\ + & 1_1 & 8 \\ \hline = & 9 & 3 \end{array}$$

Polynome:

$$\begin{array}{rcc} & 7x^1 & 5x^0 \\ + & 1x^1 & 8x^0 \\ \hline = & 8x^1 & 13x^0 \end{array}$$

    - Vorteil: einfachere Implementierung
    - Nachteil: beliebig große Koeffizienten

Inhalt

2. Multiplikation: Der Karatsuba-Algorithmus

- Karatsuba-Algorithmus für Zahlen:
  - Zerlegung der Faktoren:

$$x = a \cdot 10^{n/2} + b$$

$$y = c \cdot 10^{n/2} + d$$

$$\begin{aligned} x \cdot y &= \left(a \cdot 10^{n/2} + b\right) \cdot \left(c \cdot 10^{n/2} + d\right) \\ &= a \cdot c \cdot 10^{n} + \left(a \cdot d + b \cdot c\right) \cdot 10^{n/2} + b \cdot d \end{aligned}$$

  - Anwenden der Identität (nur noch 3 Multiplikationen):

$$a \cdot d + b \cdot c = a \cdot c + b \cdot d + \left(a - b\right) \cdot \left(d - c\right)$$

$$x \cdot y = a \cdot c \cdot 10^{n} + \left(a \cdot c + b \cdot d + \left(a - b\right) \cdot \left(d - c\right)\right) \cdot 10^{n/2} + b \cdot d$$

2. Multiplikation: Der Karatsuba-Algorithmus

- Karatsuba-Algorithmus für Zahlen:
  - *Beispiel:*

$$x = 6394 \quad \Rightarrow \quad x = 63 \cdot 10^2 + 94 = (6 \cdot 10 + 3) \cdot 10^2 + (9 \cdot 10 + 4)$$

$$y = 8312 \quad \Rightarrow \quad x = 83 \cdot 10^2 + 12 = (8 \cdot 10 + 3) \cdot 10^2 + (1 \cdot 10 + 2)$$

2. Multiplikation: Der Karatsuba-Algorithmus

```
long multiply( long x, long y ) {
  long a, b, c, d, n, p1, p2, p3;

  // Maximum der Anzahl der Stellen
  n = max. Stellen von x und y, auf nächste 2er-Potenz erhöht;

  // wenn mehr als eine Stelle, Karatsuba
  if( n > 1 ) {

    // Zerlegung von x und y
    b = x mod 10^(n/2);
    a = (x - b) / 10^(n/2);
    d = y mod 10^(n/2);
    c = ( y - d ) / 10^(n/2);

    // die drei Produkte
    p1 = multiply( a, c );
    p2 = multiply( b, d );
    p3 = multiply( a - b, d - c );

    // Resultat berechnen
    return p1 * 10^n + ( p1 + p2 + p3 ) * 10^(n/2) + p2;
  }

  return x * y;
}
```

2. Multiplikation: Der Karatsuba-Algorithmus

- Karatsuba-Algorithmus für Polynome:
  - Koeffizientenlisten statt Zahlen
    (*hier:* führender Koeffizient am Listenanfang)
  - Alle Rechnungen ohne Überträge
  - *Beispiel:*
    x = [6;3;9;4] → a = [6;3] und b = [9;4]
    y = [8;3;1;2] → c = [8;3] und d = [1;2]

    weitere Zerlegung der Listen, bis Listen nur noch aus einem Element bestehen ...

2. Multiplikation: Der Karatsuba-Algorithmus

```
list multiply ( list x, list y ) {
  list a, b, c, d, p1, p2, p3,
       midterm, tab, result;
  int n;

  n = max. Anzahl Koeffizienten von x und
      y, auf nächste 2er-Potenz erhöht;

  if( n > 1 ) {

    // Listen mit Nullen bis n auffüllen
    while( x.size() < n ) x.addFirst( 0 );
    while( y.size() < n ) y.addFirst( 0 );

    // Zerlegung der Koeffizienten-Listen
    a = subList( x, 0, n/2 - 1 );
    b = subList( x, n/2, n );
    c = subList( y, 0, n/2 - 1 );
    d = subList( y, n/2, n );

    // die drei Produkte
    p1 = multiply( a, c );
    p2 = multiply( b, d );
    p3 = multiply( subtract(a, b),
                   subtract(d, c) );

    //---- Resultat berechnen ----
    // Liste der Länge n/2,
    // nur mit Nullen
    tab = new list();
    for( int i = 0; i < n/2; ++i )
      tab.addFirst( 0 );

    // p1 + p2 + p3
    midterm = plus( p1, p2, p3 );

    // p1 * 10^n
    p1.append( tab );
    p1.append( tab );

    // midterm * 10^(n/2)
    midterm.append( tab );

    return plus( p1, midterm, p2 );
    //----------------------------
  }

  result = new list();
  result.addFirst( x.getFirst() *
                   y.getFirst() );
  return result;
}
```

2. Multiplikation: Der Karatsuba-Algorithmus

- Komplexität:
  - Addition: $O(n)$
  - Schulmultiplikation: $O(n^2)$
  - Karatsuba-Algorithmus: $O(n^{\log_2 3}) \approx O(n^{1,58})$

Ziel: $O(n \log_2 n)$

Inhalt

3. Schnelle Multiplikation mit FFT

3.1 Stützstellendarstellung

- Bisher: Koeffizientendarstellung
  - $a(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = \sum_{k=0}^{n} a_k x^k$
- Stützstellendarstellung:
  - eindeutige Darstellung des Polynoms a durch n Werte $y_0, \ldots, y_{n-1}$ an n vorgegebenen verschiedenen Stellen $x_0, \ldots, x_{n-1}$, wenn $\deg(a) = n - 1$

3. Schnelle Multiplikation mit FFT

3.1 Stützstellendarstellung

- Auswertung des Polynoms an der Stelle x_0 (Vektorschreibweise):
  - $y_0 = \begin{pmatrix} a_0 & \dots & a_{n-1} \end{pmatrix} \cdot \begin{pmatrix} {x_0}^0 \\ \vdots \\ {x_0}^{n-1} \end{pmatrix}$
  - *Beispiel:*

$$a(x) = 3x^2 + 2x + 1$$

$$sei\ x_0 = 2 \quad \Rightarrow \quad a(2) = 3 \cdot 2^2 + 2 \cdot 2 + 1 = 17$$

$$a(2) = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 2^2 \end{pmatrix} = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 2^2 = 17$$

3. Schnelle Multiplikation mit FFT

3.1 Stützstellendarstellung

- Auswertung des Polynoms an n Stellen $x_0, ..., x_{n-1}$ (Vektor-Matrix-Multiplikation):
  - $$\begin{pmatrix} y_0 & \dots & y_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 & \dots & a_{n-1} \end{pmatrix} \cdot \begin{pmatrix} x_0^{\,0} & \dots & x_{n-1}^{\,0} \\ \vdots & & \\ x_0^{\,n-1} & \dots & x_{n-1}^{\,n-1} \end{pmatrix}$$

    $$\begin{pmatrix} y_0 & \dots & y_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 & \dots & a_{n-1} \end{pmatrix} \cdot T$$

    T ist Transformationsmatrix

- Rücktransformation in Koeffizientendarstellung:
  - $$\begin{pmatrix} y_0 & \dots & y_{n-1} \end{pmatrix} \cdot T^{-1} = \begin{pmatrix} a_0 & \dots & a_{n-1} \end{pmatrix}$$

3.1 Stützstellendarstellung

- Multiplikation in Stützstellendarstellung:
  - Einfache Multiplikation der Werte (Voraussetzung: gleiche Stützstellen bei beiden Polynomen)
  - Komplexität ist $O(n)$
- Problem:
  - Komplexität der Transformation ist $O(n^2)$
- Lösung:
  - Vielfache der primitiven n-ten Einheitswurzel als Stützstellen verwenden (Diskrete Fouriertransformation)
  - Komplexität nur noch $O(n \cdot \log n)$

3.2 Einheitswurzel

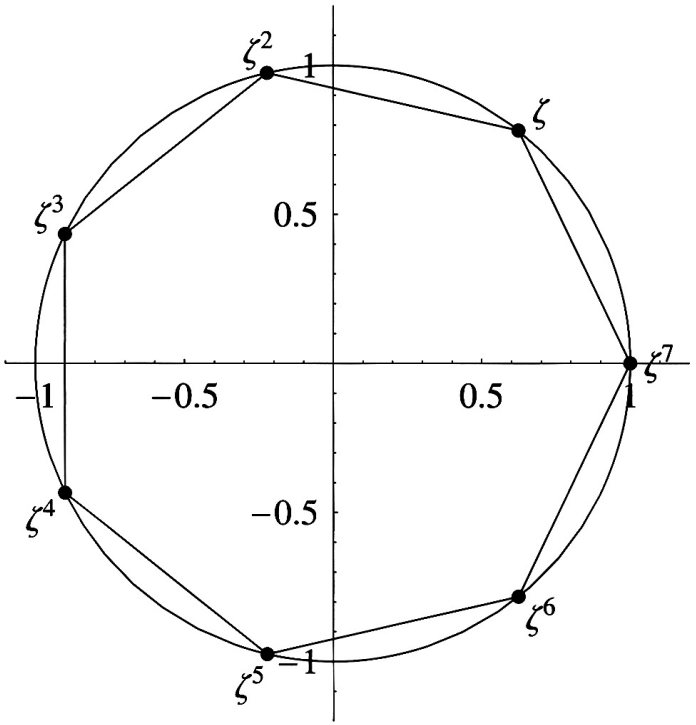
- *Definition:*
  Sei R ein kommutativer Ring mit Einselement 1 und $n \geq 1$ eine Natürliche Zahl. $\zeta \in R$ heißt *n-te Einheitswurzel*, wenn
  $\zeta^n = 1$
  und *primitive n-te Einheitswurzel*, wenn zusätzlich
  $\zeta^k \neq 1 \; für \; 0 < k < n$
- *Beispiel 1 - $R = \mathbb{Z}_9$:*
  2 und 5 sind primitive 6-te Einheitswurzeln in R,
  - denn $2^6 = \quad 64 \bmod 9 = 1$
  - denn $5^6 = 15625 \bmod 9 = 1$

3.2 Einheitswurzel

- *Beispiel 2 -* $R = \mathbb{C}$ *(komplexe primitive* n*-te Einheitswurzel):*

$$\zeta = e^{\frac{2\pi i}{n}}, \quad denn \quad \zeta = e^{\frac{2\pi i n}{n}} = e^{2\pi i} = \cos(2\pi) + i \cdot \sin(2\pi) = 1 + i \cdot 0 = 1$$

  - Punkte ζ^k *für* $0 < k < n$ bilden ein regelmäßiges n-Eck auf dem Einheitskreis der *Gaußschen Zahlenebene*:



3.2 Einheitswurzel

- *Beispiel 2 - $R = \mathbb{C}$ (komplexe primitive n-te Einheitswurzel):*

$$\zeta = e^{\frac{2\pi i}{n}}, \quad denn \quad \zeta = e^{\frac{2\pi i n}{n}} = e^{2\pi i} = \cos(2\pi) + i \cdot \sin(2\pi) = 1 + i \cdot 0 = 1$$

  - Ist n gerade so gilt:

$$\zeta^{\frac{n}{2}+k} = -\zeta^{k}$$

  - Das Quadrat einer primitiven n-ten Einheitswurzel ist primitive $n/2$-te Einheitswurzel:

$$\left(e^{\frac{2\pi i}{n}} \right)^2 = e^{\frac{2\pi i \cdot 2}{n}} = e^{\frac{2\pi i \cdot 2}{2 \cdot n/2}} = e^{\frac{2\pi i}{n/2}}$$

3.3 Schnelle Fouriertransformation

- Transformation eines Polynoms a in Stützstellendarstellung mit $\deg(a) = n - 1$ und

  $\zeta^k \quad für \quad 0 \leq k < n \quad mit \quad \zeta = e^{\frac{2\pi i}{n}}$

  als Stützstellen ergibt eine symmetrische Transformationsmatrix (Fouriermatrix):

$$\begin{pmatrix} x_0^{\,0} & \cdots & x_{n-1}^{\,0} \\ \vdots & & \\ x_0^{\,n-1} & \cdots & x_{n-1}^{\,n-1} \end{pmatrix} \overset{n=4}{\Rightarrow} \begin{pmatrix} \zeta^{0\cdot 0} & \zeta^{1\cdot 0} & \zeta^{2\cdot 0} & \zeta^{3\cdot 0} \\ \zeta^{0\cdot 1} & \zeta^{1\cdot 1} & \zeta^{2\cdot 1} & \zeta^{3\cdot 1} \\ \zeta^{0\cdot 2} & \zeta^{1\cdot 2} & \zeta^{2\cdot 2} & \zeta^{3\cdot 2} \\ \zeta^{0\cdot 3} & \zeta^{1\cdot 3} & \zeta^{2\cdot 3} & \zeta^{3\cdot 3} \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

$$F_{ij} = \zeta^{i \cdot j}$$

3.3 Schnelle Fouriertransformation

- Idee des FFT-Algorithmus:
  - Zerlegung des Polynoms in ein Polynom mit geraden und eins mit ungeraden Koeffizientenindex:

$$a(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$

$$a^{[1]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}$$

$$a^{[2]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$$

$$\Rightarrow \quad a(x) = a^{[1]}(x^2) + x \cdot a^{[2]}(x^2)$$

  - Nur noch Auswertung der Teilpolynome notwendig
  - Rekursive Fortsetzung der Zerlegung

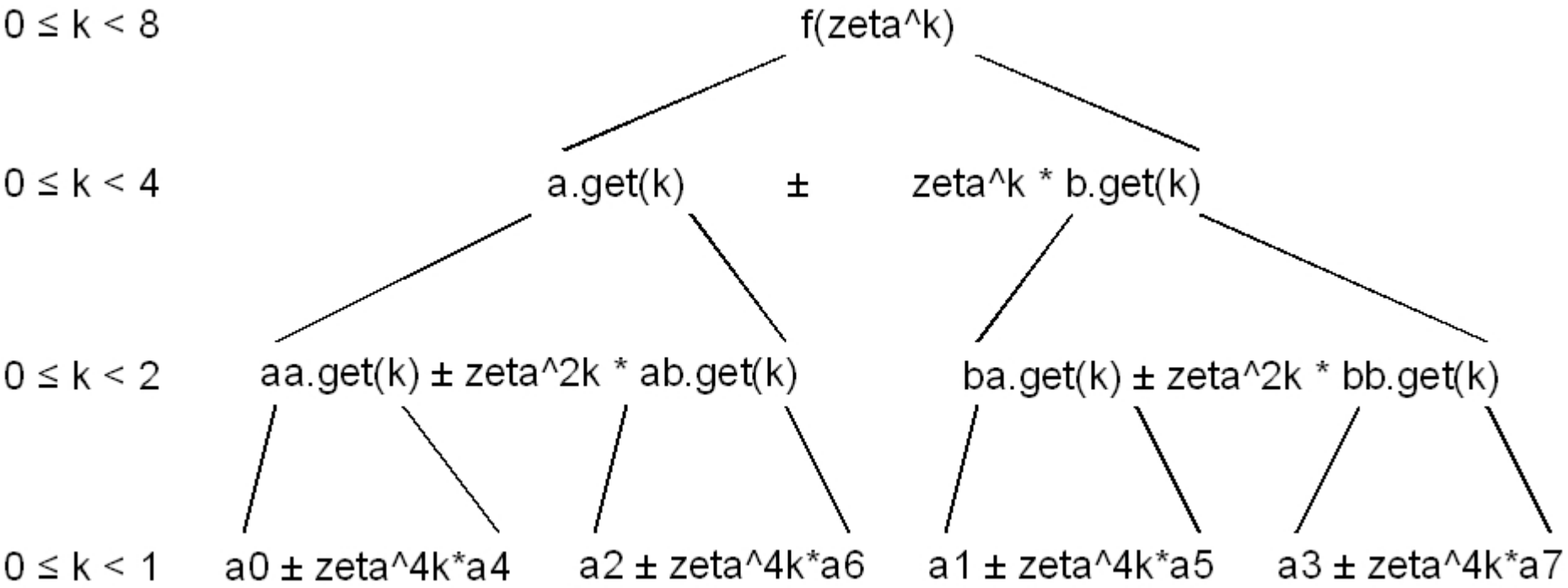
3. Schnelle Multiplikation mit FFT

3.3 Schnelle Fouriertransformation

```
// Führender Koeffizient
// am Listenende !
//
// Voraussetzung:
// Koeffizientenanzahl
// ist 2er-Potenz

list fft ( list l ) {
  complex zeta, c1, c2;
  list
    evenList = new list(),
    oddList = new list(),
    a, b,
    tab1 = new list(),
    tab2 = new list();

  // Anzahl Koeffizienten
  n = l.size();

  if( n > 1 ) {

    // primitive n-te
    // Einheitswurzel
    zeta = e^( 2*π*i / n );

    // Zerlegung der Koeffizientenliste
    for( int j = 0; j < n-1; j+=2 )
      evenList.addLast( l.get( j ) );
    for( int j = 1; j < n  ; j+=2 )
      oddList.addLast( l.get( j ) );

    a = fft( evenList );
    b = fft( oddList );

    // Werte an den Stellen zeta^k bestimmen
    for( int k = 0; k < n/2; ++k )
    {
      c1 = a.get( k ) + zeta^k *  b.get( k );
      c2 = a.get( k ) + zeta^k * -b.get( k );

      tab1.addLast( c1 );
      tab2.addLast( c2 );
    }

    // Liste tab2 an Liste tab1 hängen
    tab1.addAll( tab2 );
    return tab1;
  }

  return l;
}
```

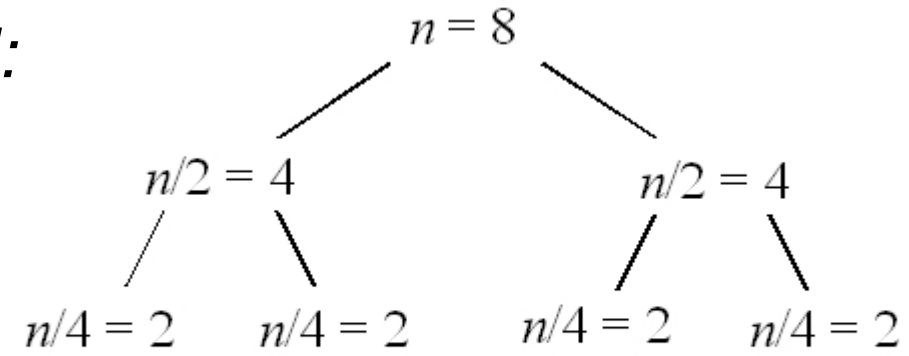
3.3 Schnelle Fouriertransformation

- *Beispiel:*
  FFT für Polynom f mit $\deg(f) = 7$;
  d.h. Es gibt 8 Koeffizienten (a_0 bis a_7)



3.3 Schnelle Fouriertransformation

- Komplexitätsanalyse:
  - Anzahl Rekursionsstufen (p):
    $p = \log_2 n$, wenn $n = 2^p$ die Anzahl der Koeffizienten ist
  - $O(n)$ Operationen pro Rekursionsstufe
  - Komplexität des FFT-Algorithmus:
    $O(n \cdot p) = O(n \cdot \log_2 n)$
  - *Beispiel:*

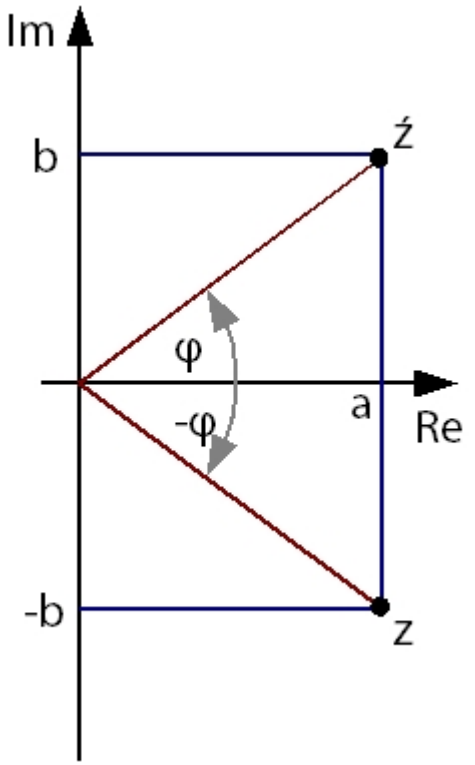


$$8 + 2 \cdot 4 + 4 \cdot 2 = 8 \cdot 3 = 8 \cdot \log_2 8 = n \cdot \log_2 n$$

3.3 Schnelle Fouriertransformation

- Inverse Schnelle Fouriertransformation (IFFT):
  - Analog zur FFT, nur andere Transformationsmatrix
  - Inverse Fouriermatrix:
    - Statt n-ter Einheitswurzel, inverse n-te Einheitswurzel verwenden (konjugiert komplexe n-te Einheitswurzel)
    - Elemente durch n dividieren

$$F^{-1} \overset{n=4}{=} \frac{1}{4} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}, \qquad F^{-1}_{ij} = \frac{\zeta^{-i \cdot j}}{n}$$



3. Schnelle Multiplikation mit FFT

3.3 Schnelle Fouriertransformation

```
list multiply( list l1, list l2 ) {
  list result = new list();

  // Grad der Multiplikation ist n+m
  int m   = l1.size();
  int n   = l2.size();
  int num = (n + m) auf nächste 2er-Potenz erhöht;
  while( l1.size() < num ) l1.addLast( 0 );
  while( l2.size() < num ) l2.addLast( 0 );

  // Fouriertransformation, O(n·logn)
  l1 = fft( l1 );
  l2 = fft( l2 );

  // Multiplikation der Werte, O(n)
  for( int i = 0; i < l1.size(); ++i )
    result.addLast( l1.get( i ) * l2.get( i ) );

  // Inverse Fouriertransformation, O(n·logn)
  result = ifft( result );

  // Koeffizienten durch n dividieren
  for( int i = 0; i < result.size(); ++i )
    result.set( i, result.get( i ) / n );

  return result;
}
```

Polynomarithmetik: Multiplikation

Vielen Dank für
die Aufmerksamkeit!