Fast convolution using polynomial transforms

by

Jens Hee
https://jenshee.dk

January 2004

Change log

3. November 2003

1. Document started.

22. November 2003

1. Program example added.

1. January 2004

1. Program example with new font.

13. March 2020

1. Meta data added.

Contents

Chapter 1

Introduction

In the following an algorithm for fast computation of the convolution of two sequences of equal length is reviewed. Fast convolution is of great importance in digital filtering and it is in this area most research has been made. However, algorithms for fast multiplication of large numbers can also benefit from the theory of convolution especially convolution of sequences of rational numbers.
The method was first proposed by H. J. Nussbaumer who has given a description in his book "Fast Fourier Transforms and Convolution Algorithms", together with other efficient methods for fast convolution. A more thorough description can be found in his paper "Fast Polynomial Transform Algorithms for Digital Convolution" IEEE Trans. ASSP april 1980.
The method is based on the computation of polynomial products using the Chinese remainder theorem and polynomial transforms. It does not require any real number operations and is therefore well suited for computations on sequences of integers and rational numbers. Although it may seem rather cumbersome it turns out to give substantial savings in the number of arithmetic operations required, the asymptotic behavior being of the order $NlogN$ where N is the length of the input sequences.
The presentation given in the following will hopefully add to the understanding of the method.

Chapter 2

Basic facts

Before introducing the algorithm, some basic facts are given about how convolution and integer multiplication can be viewed as a product of two polynomials.

2.1 Polynomial products and convolution

In general the convolution of two sequences is given by:

$$y_k = \sum_{n=-\infty}^{\infty} h_n x_{k-n} \qquad -\infty < k < \infty$$

For finite sequences of length N the convolution is a sequence of length $2N-1$ given by:

$$y_k = \sum_{n=0}^{N-1} h_n x_{k-n} \qquad k = 0, ..., 2N-2$$

where it is assumed that $x_k = 0$ for $n < 0$ and $n > N-1$.

It will now be shown that this is equivalent to finding the coefficients of the product of two polynomials.

Given two polynomials of degree $N-1$:

$$X(Z) = \sum_{n=0}^{N-1} x_n Z^n$$

$$H(Z) = \sum_{n=0}^{N-1} h_n Z^n$$

then the product of the two polynomials is a polynomial of degree $2N-2$ given by:

$$\begin{aligned} Y(Z) &= H(Z)X(Z) \\ &= \sum_{n=0}^{2N-2} y_n Z^n \end{aligned}$$

$$
\begin{aligned}
&= \sum_{n=0}^{N-1} h_n Z^n \sum_{n=0}^{N-1} x_n Z^n \\
&= \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} h_n x_m Z^{n+m} \\
&= \sum_{n=0}^{N-1} \sum_{k=n}^{N-1+n} h_n x_{k-n} Z^k
\end{aligned}
$$

If we let $x_n = 0$ for $n < 0$ and $n > N-1$, then $Y(Z)$ can be written:

$$Y(Z) = \sum_{k=0}^{2N-2} \sum_{n=0}^{N-1} h_n x_{k-n} Z^k$$

and the coefficients y_k are given by:

$$y_k = \sum_{n=0}^{N-1} h_n x_{k-n} \qquad k = 0, ..., 2N-2$$

This shows that the sequence y is the convolution of the sequences x and h, or inversely, the convolution of two sequences can be computed by a polynomial product.

2.2 Polynomial products and multiplication

Any two N-digit integers A and B can in a positional number system with base b be written:

$$A = \sum_{n=0}^{N-1} x_n b^n$$

$$B = \sum_{n=0}^{N-1} h_n b^n$$

substituting Z for b one obtains two polynomials:

$$X(Z) = \sum_{n=0}^{N-1} x_n Z^n$$

$$H(Z) = \sum_{n=0}^{N-1} h_n Z^n$$

and consequently:

$$AB = H(Z)X(Z) \qquad \text{for } Z = b$$

This shows that multiplication of two integers can be computed by a polynomial product followed by an evaluation of the resulting polynomial for $Z = b$.

Chapter 3

Circular convolution

As seen from the previous section the problem of convoluting two sequences as well as multiplication of two integers can be stated:

$$y_k = \sum_{n=0}^{N-1} h_n x_{k-n} \qquad k = 0, ..., 2N-2$$

or alternatively:

$$Y(Z) = H(Z)X(Z)$$

However, the algorithm described in the following section can only be used to compute the circular convolution given by:

$$y_k = \sum_{n=0}^{N-1} h_n x_{k-n} \qquad k = 0, ..., N-1$$

where all indices in the summation are modulo N.

or alternatively:

$$Y(Z) \equiv H(Z)X(Z) \qquad \text{modulo}(Z^N - 1)$$

here and in the following the term on the left hand side denotes the polynomial in the equivalence class having the degree L, where $0 \leq L \leq N-1$.
The limitation to circular convolution is not a serious problem since an ordinary convolution is identical to a circular convolution if the sequences x and h are extended with zeros to the double length.

Chapter 4

Computation of polynomial products modulo $(Z^N - 1)$

The problem considered in this section is the computation of polynomial products modulo (Z^N-1):

$$Y(Z) \equiv H(Z)X(Z) \qquad \text{modulo}(Z^N - 1)$$

In the following only values of $N = 2^t$ are considered although the basic idear behind the algorithm can be used for other values of N.
The first step is to reduce the complexity of the problem by using the Chinese remainder theorem (see appendix A). Since $N = 2^t$, $(Z^N - 1)$ can be factorized as:

$$Z^N - 1 = (Z^{N/2} + 1)(Z^{N/2} - 1)$$

and $Y(Z)$ can be written:

$$Y(Z) \equiv \frac{1}{2}(Z^{N/2} + 1)Y_1(Z) - (Z^{N/2} - 1)Y_2(Z) \qquad \text{modulo } (Z^N - 1)$$

where:

$$\begin{aligned} Y_1(Z) &\equiv H_1(Z)X_1(Z) \qquad \text{modulo } (Z^{N/2} - 1) \\ X_1(Z) &\equiv X(Z) \qquad \text{modulo } (Z^{N/2} - 1) \\ H_1(Z) &\equiv H(Z) \qquad \text{modulo } (Z^{N/2} - 1) \end{aligned}$$

$$\begin{aligned} Y_2(Z) &\equiv H_2(Z)X_2(Z) \qquad \text{modulo } (Z^{N/2} + 1) \\ X_2(Z) &\equiv X(Z) \qquad \text{modulo } (Z^{N/2} + 1) \\ H_2(Z) &\equiv H(Z) \qquad \text{modulo } (Z^{N/2} + 1) \end{aligned}$$

The computation of Y from Y_1 and Y_2 is strait forward and the problem is reduced to computing Y_1 and Y_2.
Y_1 is a polynomial product modulo $(Z^{N/2}-1)$ and can be recursively computed by use of the above method. Y_2, on the other hand, is a polynomial product modulo $(Z^{N/2} + 1)$. The computational procedure for this problem is described in the following section.

Chapter 5

Computation of polynomial products modulo $(Z^N + 1)$

The problem considered in this section is the computation of polynomial products modulo (Z^N+1):

$$Y(Z) \equiv H(Z)X(Z) \qquad \text{modulo}(Z^N + 1)$$

where $N = 2^t$.

First X, H and Y are rewritten as two-dimensional polynomials:

$$X^{'}(Z, Z_1) = \sum_{n=0}^{L_1-1} \sum_{m=0}^{L_2-1} x_{L_1 m+n} Z_1^m Z^n = \sum_{n=0}^{L_1-1} Q_n(Z_1) Z^n$$

$$H^{'}(Z, Z_1) = \sum_{n=0}^{L_1-1} \sum_{m=0}^{L_2-1} h_{L_1 m+n} Z_1^m Z^n = \sum_{n=0}^{L_1-1} R_n(Z_1) Z^n$$

$$Y^{'}(Z, Z_1) = \sum_{n=0}^{L_1-1} \sum_{m=0}^{L_2-1} y_{L_1 m+n} Z_1^m Z^n = \sum_{n=0}^{L_1-1} S_n(Z_1) Z^n$$

where Q_n, R_n and S_n are polynomials in Z_1 of degree $L_2 - 1$, $Z_1 = Z^{L_1}$ and $L_1 L_2 = N$.

Since:

$$\begin{aligned} X^{'}(Z, Z_1) &\equiv X(Z) \qquad \text{modulo } (Z^{L_1} - Z_1) \\ H^{'}(Z, Z_1) &\equiv H(Z) \qquad \text{modulo } (Z^{L_1} - Z_1) \\ Y^{'}(Z, Z_1) &\equiv Y(Z) \qquad \text{modulo } (Z^{L_1} - Z_1) \\ Z_1^{L_2} + 1 &\equiv Z^N + 1 \qquad \text{modulo } (Z^{L_1} - Z_1) \end{aligned}$$

the polynomial product HX modulo $(Z^N + 1)$ can be expressed by:

$$\begin{aligned} P^{'}(Z, Z_1) &\equiv H^{'}(Z, Z_1) X^{'}(Z, Z_1) \qquad \text{modulo } (Z^{L_1} - Z_1) \\ Y^{'}(Z, Z_1) &\equiv P^{'}(Z, Z_1) \qquad \text{modulo } (Z_1^{L_2} + 1) \end{aligned}$$

or alternatively:

$$\begin{aligned} U^{'}(Z,Z_1) &\equiv H^{'}(Z,Z_1)X^{'}(Z,Z_1) \qquad \text{modulo } (Z_1^{L_2}+1) \\ V^{'}(Z,Z_1) &\equiv U^{'}(Z,Z_1) \qquad \text{modulo } (Z^{L_1}-Z_1) \\ Y^{'}(Z,Z_1) &\equiv V^{'}(Z,Z_1) \qquad \text{modulo } (Z_1^{L_2}+1) \end{aligned}$$

The computation of $Y^{'}$ from $U^{'}$ is strait forward and the problem is reduced to finding $U^{'}$.

$U^{'}(Z,Z_1)$ can be written:

$$U^{'}(Z,Z_1) = \sum_{n=0}^{2L_1-2} U_n(Z_1)Z^n$$

where U_n is a polynomial in Z_1 of degree L_2-1.

and consequently:

$$U_k(Z_1) \equiv \sum_{n=0}^{L_1-1} R_n(Z_1)Q_{k-n}(Z_1) \qquad \text{modulo } (Z_1^{L_2}+1) \qquad k=0,...,2L_1-2$$

if we in the summation let $Q_n(Z)=0$ for $n<0$ and $n>L_1-1$.

Now letting $R_n(Z_1)=Q_n(Z_1)=0$ for $L_1 \le n \le 2L_1-1$, $U_{2L_1-1}(Z_1)=0$ and taking all indices modulo $2L_1$ then $U_k(Z_1)$ can be written:

$$U_k(Z_1) \equiv \sum_{n=0}^{2L_1-1} R_n(Z_1)Q_{k-n}(Z_1) \qquad \text{modulo } (Z_1^{L_2}+1) \qquad k=0,...,2L_1-1$$

This shows that the sequence U_k is a circular convolution of the sequences R_n and Q_n and consequently can be computed using polynomial transforms (see Appendix B). As a result one has:

$$U_k(Z_1) \equiv \frac{1}{2L_1} \sum_{n=0}^{2L_1-1} \overline{R}_n(Z_1)\overline{Q}_n(Z_1)Z_1^{-(L_2/L_1)nk} \qquad \text{modulo } (Z_1^{L_2}+1) \qquad k=0,...,2L_1-1$$

where:

$$\overline{Q}_n(Z_1) = \sum_{k=0}^{2L_1-1} Q_k(Z_1)Z_1^{(L_2/L_1)kn} \qquad \text{modulo } (Z_1^{L_2}+1) \qquad n=0,...,2L_1-1$$

$$\overline{R}_n(Z_1) = \sum_{k=0}^{2L_1-1} R_k(Z_1)Z_1^{(L_2/L_1)kn} \qquad \text{modulo } (Z_1^{L_2}+1) \qquad n=0,...,2L_1-1$$

$L_2 \ge L_1$ since L_2/L_1 must be an integer. It has been shown that the best choice for L_1 and L_2 is the one that gives the least quotient (see Nussbaumer).
The polynomial products $\overline{Q}_n\overline{R}_n$ modulo $Z_1^{L_2}+1$ is computed by recursive use of the above method and the final problem is to find an efficient method for computing the polynomial transforms. This is described in the next section.

Chapter 6

Computation of polynomial transforms

The problem considered in this section is the computation of polynomial transforms of the form:

$$Y_k(Z) \equiv \sum_{n=0}^{N-1} X_n(Z) Z^{Pkn} \qquad \text{modulo } (Z^L + 1) \qquad k = 0, ..., N-1$$

where $P = 2L/N$, P being an integer.

The polynomial transform is similar to the DFT and can be computed correspondingly, that is, using an FFT-like algorithm. The presentation and notation given in the next section is very similar to the presentation given in standard textbooks on FFT.

6.1 Decimation-in-Time Algorithm

By splitting the polynomial sequence X into two sequences using the even and odd indices one obtains:

$$\begin{aligned} Y_k(Z) &\equiv \sum_{n=0}^{N-1} X_n(Z) Z^{Pkn} \\ &\equiv \sum_{n\ even} X_n(Z) Z^{Pkn} + \sum_{n\ odd} X_n(Z) Z^{Pkn} \\ &\equiv \sum_{n=0}^{N/2-1} X_{2n}(Z) Z^{2Pkn} + Z^{Pk} \sum_{n=0}^{N/2-1} X_{2n+1}(Z) Z^{2Pkn} \qquad \text{modulo } (Z^L + 1) \end{aligned}$$

and

$$\begin{aligned} Y_{k+N/2}(Z) &\equiv \sum_{n=0}^{N/2-1} X_{2n}(Z) Z^{2Pkn} + Z^{Pk} Z^{PN/2} \sum_{n=0}^{N/2-1} X_{2n+1}(Z) Z^{2Pkn} \\ &\equiv \sum_{n=0}^{N/2-1} X_{2n}(Z) Z^{2Pkn} - Z^{Pk} \sum_{n=0}^{N/2-1} X_{2n+1}(Z) Z^{2Pkn} \qquad \text{modulo } (Z^L + 1) \end{aligned}$$

By recursively using the procedure on the new sequences, one obtains an algorithm equivalent to the Decimation-in-Time FFT-algorithm as shown on Figure 6.1.

6.2 Decimation-in-Frequency Algorithm

By splitting the polynomial sequence X into two sequences using the first and second half one obtains:

$$\begin{aligned} Y_k(Z) &\equiv \sum_{n=0}^{N-1} X_n(Z) Z^{Pkn} \\ &\equiv \sum_{n=0}^{N/2-1} X_n(Z) Z^{Pkn} + \sum_{n=N/2}^{N-1} X_n(Z) Z^{Pkn} \\ &\equiv \sum_{n=0}^{N/2-1} X_n(Z) Z^{Pkn} + Z^{PkN/2} \sum_{n=0}^{N/2-1} X_{n+N/2}(Z) Z^{Pkn} \\ &\equiv \sum_{n=0}^{N/2-1} (X_n(Z) + (-1)^k X_{n+N/2}(Z)) Z^{Pkn} \qquad \text{modulo } (Z^L + 1) \end{aligned}$$

and

$$\begin{aligned} Y_{2k}(Z) &\equiv \sum_{n=0}^{N/2-1} (X_n(Z) + X_{n+N/2}(Z)) Z^{2Pkn} \qquad \text{modulo } (Z^L + 1) \\ Y_{2k+1}(Z) &\equiv \sum_{n=0}^{N/2-1} (X_n(Z) - X_{n+N/2}(Z)) Z^{Pn} Z^{2Pkn} \qquad \text{modulo } (Z^L + 1) \end{aligned}$$

By recursively using the procedure on the new sequences, one obtains an algorithm equivalent to the Decimation-in-Frequency FFT-algorithm as shown on Figure 6.2.

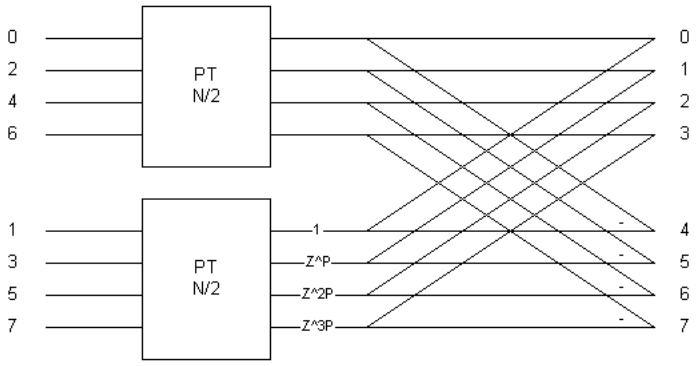

Figure 6.1: First step of Decimation-in-Time algorithm

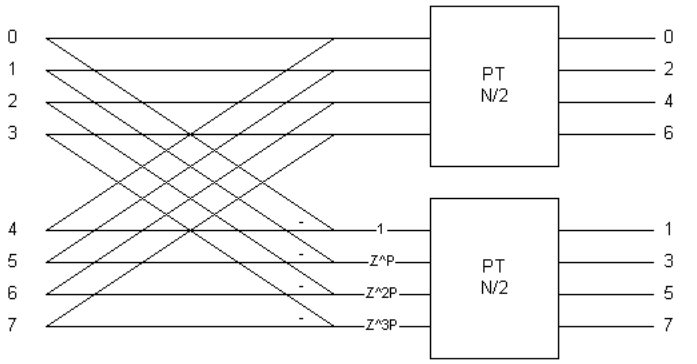

Figure 6.2: First step of Decimation-in-Frequency algorithm

Appendix A

The Chinese Remainder Theorem

Traditionally the Chinese remainder theorem has been used to solve a set of k linear congruences:

$$x \equiv r_i \qquad \text{modulo } p_i \qquad i = 1, ..., k$$

This problem has a unique solution if p_i are relative prime in pairs:

$$x = \sum_{i=1}^{k} P_i T_i r_i \qquad \text{modulo } P$$

where

$$P = \prod_{i=1}^{k} p_i$$

$$P_i = \prod_{\substack{j=1 \\ j \neq i}}^{k} p_j$$

$$P_i T_i \equiv 1 \qquad \text{modulo } p_i$$

The equivalent theorem for polynomials is then:

$$X(Z) \equiv \sum_{i=1}^{k} P_i(Z) T_i(Z) R_i(Z) \qquad \text{modulo } P(Z)$$

if

$$R_i(Z) \equiv X(Z) \qquad \text{modulo } p_i(Z) \qquad i = 1, ..., k$$

$$P(Z) = \prod_{i=1}^{k} p_i(Z)$$

$$P_i(Z) = \prod_{\substack{j=1 \\ j \neq i}}^{k} p_j(Z)$$

$$P_i(Z) T_i(Z) \equiv 1 \qquad \text{modulo } p_i(Z)$$

The $p_i(Z)$ may not have common factors (usually called relatively prime polynomials).

Appendix B

The Polynomial Transform

The polynomial transform of a sequence of polynomials is defined by:

$$\overline{X}_k(Z) \equiv \sum_{n=0}^{N-1} X_n(Z) G^{kn}(Z) \qquad \text{modulo } P(Z) \qquad k = 0, ..., N-1$$

If the following conditions hold:
1. $G^N(Z) \equiv 1$ modulo $P(Z)$
2. $N \neq 0$
3. $G(Z)$ has an inverse modulo $P(Z)$
4. $\sum_{k=0}^{N-1} G^{qk}(Z) \equiv \begin{cases} 0 \text{ modulo } P(Z) \text{ for } q \not\equiv 0 \text{ modulo } N \\ N \text{ modulo } P(Z) \text{ for } q \equiv 0 \text{ modulo } N \end{cases}$

then the well known convolution property of the DFT also holds for the polynomial transform.

That is, if:

$$Y_k(Z) \equiv \sum_{n=0}^{N-1} H_n(Z) X_{k-n}(Z) \qquad \text{modulo } P(Z) \qquad k = 0, ..., N-1$$

then:

$$Y_k(Z) \equiv \frac{1}{N} \sum_{n=0}^{N-1} \overline{H}_n(Z) \overline{X}_n(Z) G^{-kn}(Z) \qquad \text{modulo } P(Z) \qquad k = 0, ..., N-1$$

where:

$$\overline{X}_k(Z) \equiv \sum_{n=0}^{N-1} X_n(Z) G^{kn}(Z) \qquad \text{modulo } P(Z) \qquad k = 0, ..., N-1$$

$$\overline{H}_k(Z) \equiv \sum_{n=0}^{N-1} H_n(Z) G^{kn}(Z) \qquad \text{modulo } P(Z) \qquad k = 0, ..., N-1$$

note that all indices are taken modulo N.

As a special case if we let $G(Z) = Z^P$ and $P(Z) = Z^L + 1$ where $2L = PN$ then it is easily seen that the above conditions for the convolution theorem to be valid are met.

Appendix C

Program example

```
#include "stdio.h"
#include "math.h"

const NMax = 65536;
const SomeMore = 300;
const PolyTempMax = 256;

#define NUMBERTYPE long

NUMBERTYPE * Global = new NUMBERTYPE[4*NMax + SomeMore];

long CountAdd;
long CountMul;

// Standard circular convolution
void CircConv(long N, NUMBERTYPE *X, NUMBERTYPE *H, NUMBERTYPE *Y)
{
long i, j;

for (i = 0; i < N ;i++)
Y[i] = 0;
for (i = 0; i < N ;i++)
for (j = 0; j < N ;j++)
if (i + j < N)
Y[i + j] += X[j] * H[i];
else
Y[i + j - N] += X[j] * H[i];
}

void WriteResult(long N, NUMBERTYPE *X)
{
long i;

for (i = 0; i < N; i++)
{
printf("%8d", X[i]);
if ((i % 10) == 9)
printf("\n");
}
printf("\n");
}

void ModuloPlusMinus(long N, long X)
{
long i;
NUMBERTYPE Temp;

for (i = 0; i < N; i++)
{
```

```
Temp = Global[X + i];
Global[X + i] = Temp - Global[X + i + N];
Global[X + i + N] = Temp + Global[X + i + N];

CountAdd += 2;
}
}

void MulAddDiv(long N, long X1, long X2)
{
long i;
NUMBERTYPE Temp;

for (i = 0; i < N; i++)
{
Temp = Global[X1 + i];
Global[X1 + i] = (Temp + Global[X2 + i]) / 2;
Global[X1 + i + N] = (-Temp + Global[X2 + i]) / 2;

CountAdd += 2;
}
}

void PermutePolySub(long N, long K, long X, NUMBERTYPE * Y, long Z)
{
long i;
long adr;
long ex;

for (i = 0; i < N; i++)
{
adr = (i + K + 2 * N * N) % N;
ex = ((i + K + 2 * N * N) / N) % 2;
if (ex == 0)
{
Global[Z + adr] = Global[X + i] - Y[i];

CountAdd ++;
}
else
{
Global[Z + adr] = -Global[X + i] + Y[i];

CountAdd ++;
}
}
}

void PolyAdd(long N, long X, NUMBERTYPE * Y, long Z)
{
long i;

for (i = 0; i < N; i++)
{
Global[Z + i] = Global[X + i] + Y[i];

CountAdd ++;
}
}

void PolySubPermute(long N, long K, long X, NUMBERTYPE * Y, long Z)
{
long i;
long adr;
long ex;

for (i = 0; i < N; i++)
{
```

```
adr = (i + K + 2 * N * N) % N;
ex = ((i + K + 2 * N * N) / N) % 2;
if (ex == 0)
{
Global[Z + adr] = Global[X + adr] - Y[i];

CountAdd ++;
}
else
{
Global[Z + adr] = Global[X + adr] + Y[i];

CountAdd ++;
}
}
}

void PolyAddPermute(long N, long K, long X, NUMBERTYPE * Y, long Z)
{
long i;
long adr;
long ex;

for (i = 0; i < N; i++)
{
adr = (i + K + 2 * N * N) % N;
ex = ((i + K + 2 * N * N) / N) % 2;
if (ex == 0)
{
Global[Z + adr] = Global[X + adr] + Y[i];

CountAdd ++;
}
else
{
Global[Z + adr] = Global[X + adr] - Y[i];

CountAdd ++;
}
}
}

void PolyTransNB(long N, long L2, long K, long X)
{
long i;
long M;
long m1;
long P;
long p1;
long Q;
long q1;
long adr;
NUMBERTYPE *Temp = new NUMBERTYPE[PolyTempMax];;

M = (long)floor(0.5 + log(N) / log(2));
P = 1;
Q = N / 2;
for (m1 = 0; m1 < M; m1++)
{
for (p1 = 0; p1 < P; p1++)
for (q1 = 0; q1 < Q; q1++)
{
adr = q1 + 2 * p1 * Q;
for (i = 0; i < L2; i++)
Temp[i] = Global[X + L2 * (adr +Q) + i];
PermutePolySub(L2, P * q1 * K, X + L2 * adr, Temp, X + L2 * (adr + Q));
PolyAdd(L2, X + L2 * adr, Temp, X + L2 * adr);
}
```

```
P *= 2;
Q /= 2;
}
delete []Temp;
}

void PolyTransBN(long N, long L2, long K, long X)
{
long i;
long M;
long m1;
long P;
long p1;
long Q;
long q1;
long adr;
NUMBERTYPE *Temp = new NUMBERTYPE[PolyTempMax];;

M = (long)floor(0.5 + log(N) / log(2));
P = N / 2;
Q = 1;
for (m1 = 0; m1 < M; m1++)
{
for (p1 = 0; p1 < P; p1++)
for (q1 = 0; q1 < Q; q1++)
{
adr = q1 + 2 * p1 * Q;
for (i = 0; i < L2; i++)
Temp[i] = Global[X + L2 * (adr + Q) + i];
PolySubPermute(L2, P * q1 * K, X + L2 * adr, Temp, X + L2 * (adr + Q));
PolyAddPermute(L2, P * q1 * K, X + L2 * adr, Temp, X + L2 * adr);
}
P /= 2;
Q *= 2;
}
delete []Temp;
}

void NegaConvolution(long N, long X, long H, long Y)
{
long i, k;
long N2, N3, N4;
long L1, L2;
NUMBERTYPE Tp0, Tp1, Tp2;

if (N == 2)
{
Tp0 = (Global[X] + Global[X + 1]) * Global[H];
Tp1 = (Global[H] + Global[H + 1]) * Global[X + 1];
Tp2 = (Global[H] - Global[H + 1]) * Global[X];

CountAdd +=3;
CountMul +=3;

Global[X] = Tp0 - Tp1;
Global[X+1] = Tp0 - Tp2;

CountAdd +=2;
}
else
{
N2 = 2 * N;
N3 = 3 * N;
N4 = 4 * N;

L2 = (long)floor(0.5 + sqrt(N));
if (L2 * L2 != N)
L2 = (long)floor(0.5 + sqrt(N2));
```

```
L1 = N / L2;

for (k = 0; k < L1; k++)
for (i = 0; i < L2; i++)
{
Global[Y + L2 * k + i] = Global[X + L1 * i + k];
Global[Y + N2 + L2 * k + i] = Global[H + L1 * i + k];
Global[Y + N + L2 * k + i] = 0;
Global[Y + N3 + L2 * k + i] = 0;
}

PolyTransNB(2 * L1, L2, L2 / L1, Y);
PolyTransNB(2 * L1, L2, L2 / L1, Y + N2);

for (k = 0; k < 2 * L1; k++)
NegaConvolution(L2, Y + L2 * k, Y + N2 + L2 * k, Y + N4);

PolyTransBN(2 * L1, L2, -(L2 / L1), Y);

for (k = 0; k < L1; k++)
{
for (i = 1; i < L2; i++)
{
Global[X + L1 * i + k] = (Global[Y + L2 * k + i] +
                          Global[Y + N + L2 * k + (i - 1)]) / (2 * L1);

CountAdd++;
}

Global[X + k] = (Global[Y + L2 * k] - Global[Y + N + L2 * k + L2 - 1]) / (2 * L1);

CountAdd++;
}



}

}

void CircConvolution(long N, long X, long H)
{
long M;
long N2;
long X1, X2;
NUMBERTYPE Tp0, Tp1, Tp2;

N2 = 2 * N;
M = N / 2;
do
{
ModuloPlusMinus(M, X);
ModuloPlusMinus(M, H);
NegaConvolution(M, X, H, N2);
X += M;
H += M;
M /= 2;
} while (M != 1);

Tp0 = (Global[X] + Global[X + 1]) * Global[H];
Tp2 = (Global[H] - Global[H + 1]);
Tp1 = Tp2 * Global[X + 1];
Tp2 = Tp2 * Global[X];

CountAdd +=2;
CountMul +=3;

Global[X] = Tp0 - Tp1;
```

```
Global[X+1] = Tp0 - Tp2;

CountAdd +=2;

M = 2;
X1 = X - M;
X2 = X;
do
{
MulAddDiv(M, X1, X2);
X2 -= M;
M *= 2;
X1 -= M;
} while (M != N);

}

void main()
{
NUMBERTYPE * sequenceOne = new NUMBERTYPE[NMax];
NUMBERTYPE * sequenceTwo = new NUMBERTYPE[NMax];
NUMBERTYPE * resultFast = new NUMBERTYPE[NMax];
NUMBERTYPE * resultSlow = new NUMBERTYPE[NMax];

long N;
long X,H;
long i;

do
{
printf("Length of sequence = \n");  // N must be >= 4 and power of 2
scanf("%d", &N);

X = 0;
H = N;

for (i = 0; i < N; i++)
{
sequenceOne[i] = i; // just an example
sequenceTwo[i] = i + 10; // just an example
}

for (i = 0; i < N; i++)
{
Global[X + i] = sequenceOne[i];     // First part of Global must contain 1. sequence
Global[H + i] = sequenceTwo[i];     // Second part of Global must contain 2. sequence
}

CountAdd = 0;
CountMul = 0;

// Fast convolution
CircConvolution(N, X, H);
WriteResult(N, Global); // First part of Global contains the convolution
printf("\n");

// Standard convolution just for comparison
CircConv(N, sequenceOne, sequenceTwo, resultSlow);
WriteResult(N, resultSlow);

printf("Additions: %d \n", CountAdd);
printf("Multiplications: %d \n", CountMul);

}while (true);

}
```