Chapter 26

Number theoretic transforms (NTTs)

We introduce the number theoretic transforms (NTTs). The routines for the fast NTTs are rather straightforward translations of the FFT algorithms. Radix-2 and radix-4 routines are given, there should be no difficulty to translate any given complex FFT into the equivalent NTT. For the translation of real-valued FFT (or FHT) routines, we need to express sines and cosines in modular arithmetic, this is presented in sections 39.12.6 and 39.12.7.

As no rounding errors occur with the underlying modular arithmetic, the main application of NTTs is the fast computation of exact convolutions.

26.1 Prime moduli for NTTs

We want to implement FFTs in $\mathbb{Z}/m\mathbb{Z}$ (the ring of integers modulo some integer m) instead of $\mathbb{C}$, the field of complex numbers. These FFTs are called *number theoretic transforms* (NTTs), *mod m FFTs* or (if m is a prime) *prime modulus transforms*.

There is a restriction for the choice of m: for a length-n NTT we need a primitive n-th root of unity. A number r is called an n-th root of unity if $r^n = 1$. It is called a *primitive n-th root* if $r^k \neq 1 \; \forall \, k < n$ (see section 39.5 on page 774).

In $\mathbb{C}$ matters are simple: $e^{\pm 2\pi i/n}$ is a primitive n-th root of unity for arbitrary n. For example, $e^{2\pi i/21}$ is a primitive 21st root of unity. Now $r = e^{2\pi i/3}$ is also 21st root of unity but not a primitive root, because $r^3 = 1$. A primitive n-th root of 1 in $\mathbb{Z}/m\mathbb{Z}$ is also called an *element of order n*. The 'cyclic' property of the elements r of order n lies in the heart of all FFT algorithms: $r^{n+k} = r^k$.

In $\mathbb{Z}/m\mathbb{Z}$ things are not that simple: for a given modulus m primitive n-th roots of unity do not exist for arbitrary n. They only exist for some maximal order R and its divisors d_i: r^{R/d_i} is a d_i-th root of unity because $(r^{R/d_i})^{d_i} = r^R = 1$. Therefore n, the length of the transform, must divide the maximal order R. This is the first condition for NTTs:

$$n \quad \backslash \quad R \tag{26.1-1}$$

The operations needed in FFTs are modular addition, subtraction and multiplication, as described in section 39.1 on page 764. Division is not needed, except for the division by n in the final normalization. Division by n is multiplication by the inverse of n, so n must be invertible in $\mathbb{Z}/m\mathbb{Z}$.

Therefore n, the length of the transform, must be *coprime* to the modulus m. This is the second condition for NTTs.

$$\gcd(n, m) \quad = \quad 1 \tag{26.1-2}$$

We restrict our attention to prime moduli, though NTTs are also possible with composite moduli. If the modulus is a prime p, then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = \mathrm{GF}(p)$: all elements except 0 have inverses and 'division is possible'. Thus the second condition (relation 26.1-2) is trivially fulfilled for all NTT lengths $n < p$: a prime p is coprime to all integers $n < p$.

Roots of unity are available for the maximal order $R = p-1$ and its divisors: Therefore the first condition (relation 26.1-1) is that n divides $p-1$. This restricts the choice for p to primes of the form $p = v\,n+1$: for length-$n = 2^k$ NTTs one will use primes like $p = 3 \cdot 5 \cdot 2^{27} + 1$ (31 bits), $p = 13 \cdot 2^{28} + 1$ (32 bits), $p = 3 \cdot 29 \cdot 2^{56} + 1$ (63 bits) or $p = 27 \cdot 2^{59} + 1$ (64 bits).

```
arg 1: 62 == wb  [word bits, wb<=63]  default=62
arg 2: 0.01 == deltab  [results are in the range [wb-deltab, wb]]  default=0.01
 minb = 61.99 = wb-0.01
arg 3: 44 == minx  [log_2(min(fftlen))]  default=44
  ---- x = 44: -----
4580495072570638337 = 0x3f91300000000001 = 1 + 2^44 * 83 * 3137          (61.9902 bits)
4581058022524059649 = 0x3f93300000000001 = 1 + 2^44 * 3 * 11 * 13 * 607  (61.9904 bits)
4582113536686724609 = 0x3f96f00000000001 = 1 + 2^44 * 3 * 7 * 79 * 157   (61.9907 bits)
4585702359639785473 = 0x3fa3b00000000001 = 1 + 2^44 * 3^2 * 11 * 2633    (61.9918 bits)
4587039365779161089 = 0x3fa8700000000001 = 1 + 2^44 * 7 * 193^2          (61.9923 bits)
4587391209500049409 = 0x3fa9b00000000001 = 1 + 2^44 * 3 * 17 * 5113      (61.9924 bits)
4588130081313914881 = 0x3fac500000000001 = 1 + 2^44 * 3 * 5 * 17387      (61.9926 bits)
4589572640569556993 = 0x3fb1700000000001 = 1 + 2^44 * 11 * 37 * 641      (61.9931 bits)
[--snip--]
4610999923171655681 = 0x3ffd900000000001 = 1 + 2^44 * 5 * 19 * 31 * 89   (61.9998 bits)
4611105476287922177 = 0x3ffdf00000000001 = 1 + 2^44 * 262111             (61.9998 bits)
  ---- x = 45: -----
4580336742896238593 = 0x3f90a00000000001 = 1 + 2^45 * 29 * 67^2          (61.9902 bits)
4581533011547258881 = 0x3f94e00000000001 = 1 + 2^45 * 3 * 5 * 8681       (61.9905 bits)
4584347761314365441 = 0x3f9ee00000000001 = 1 + 2^45 * 5 * 11 * 23 * 103  (61.9914 bits)
4587655092290715649 = 0x3faaa00000000001 = 1 + 2^45 * 3 * 7^2 * 887      (61.9925 bits)
[--snip--]
  ---- x = 48: -----
4585508845593296897 = 0x3fa3000000000001 = 1 + 2^48 * 11 * 1481          (61.9918 bits)
  ---- x = 49: -----
4582975570802900993 = 0x3f9a000000000001 = 1 + 2^49 * 7 * 1163           (61.991 bits)
4595360469778169857 = 0x3fc6000000000001 = 1 + 2^49 * 3^2 * 907          (61.9949 bits)
  ---- x = 50: -----
4601552919265804289 = 0x3fdc000000000001 = 1 + 2^50 * 61 * 67            (61.9968 bits)
```

Figure 26.1-A: Primes suitable for NTTs of lengths dividing 2^{44}.

```
     modulus (hex)       ==    factorization + 1     log(m-1)/log(2)
    0x3f40f80000000001  == 2^43.3^2.5^2.7^2.47+1         61.9831
    0x3c0eb50000000001  == 2^40.3^3.5^2.7^3.17+1         61.9083
    0x3d673d0000000001  == 2^40.3^2.5^3.7^2.73+1         61.9402
    0x3fc22b0000000001  == 2^40.3^2.5^2.7^2.379+1        61.9945
    0x3bf6190000000001  == 2^40.3^2.5^3.7.499+1          61.906
    0x3d1d690000000001  == 2^40.3^2.5^2.7.2543+1         61.9335
    0x3d8c270000000001  == 2^40.3^2.5^2.7.13.197+1       61.9436
    0x3e8e8d0000000001  == 2^40.3^2.5^2.7.19.137+1       61.9671
    0x3ee4af0000000001  == 2^40.3^2.5^2.7.2617+1         61.9748
    0x3ed23a0000000001  == 2^41.3^2.5^2.7.1307+1         61.9732
    0x3fafb60000000001  == 2^41.3^2.5^4.7.53+1           61.9929
    0x3c46140000000001  == 2^42.3^3.5^2.7.11.19+1        61.9135
    0x3e32440000000001  == 2^42.3^2.5^2.7.647+1          61.9588
    0x3d23900000000001  == 2^44.3^3.5^2.7.53+1           61.934
```

Figure 26.1-B: Primes suitable for NTTs of lengths dividing $2^{40}\,3^2\,5^2\,7$.

Primes suitable with NTTs (sometimes called *FFT-primes*) can be generated with the program [FXT: mod/fftprimes-demo.cc]. A shortened sample output is shown in figure 26.1-A. A few moduli that allow for transforms of lengths dividing $2^{40} \cdot 3^2 \cdot 5^2 \cdot 7$ are shown in figure 26.1-B, the data is taken from [FXT: mod/moduli.txt]. We note that primality of moduli suitable for NTTs can easily by tested using Proth's theorem, see section 39.11.3.1 on page 795.

26.2 Implementation of NTTs

To implement NTTs (modulo m, length n), we need to implement modular arithmetic and replace $e^{\pm 2\pi i/n}$ by a primitive n-th root r of unity in $\mathbb{Z}/m\mathbb{Z}$ in the code. A C++ class implementing modular arithmetic is [FXT: `class mod` in mod/mod.h].

For the inverse transform one uses the (mod m) inverse r^{-1} of r that was used for the forward transform. The element r^{-1} is also a primitive n-th root. Methods for the computation of the modular inverse are described in section 39.1.4 on page 767 (GCD algorithm) and in section 39.7.4 on page 781 (powering algorithm).

While the notion of the Fourier transform as a 'decomposition into frequencies' appears to be meaningless for NTTs the algorithms are denoted with 'decimation in time/frequency' in analogy to those in the complex domain.

The nice feature of NTTs is that there is no loss of precision in the transform as with the floating-point FFTs. Using the trigonometric recursion in its most naive form is mandatory, as the computation of roots of unity is expensive.

26.2.1 Radix-2 DIT NTT

Pseudocode for the radix-2 decimation in time (DIT) NTT (to be called with `ldn=log2(n)`):

```
1   procedure mod_fft_dit2(f[], ldn, is)
2   // mod_type f[0..2**ldn-1]
3   {
4       n := 2**ldn
5
6       rn := element_of_order(n)  // (mod_type)
7
8       if is<0 then  rn := rn**(-1)
9
10      revbin_permute(f[], n)
11
12      for ldm:=1 to ldn
13      {
14          m  := 2**ldm
15          mh := m/2
16
17          dw := rn**(2**(ldn-ldm))   // (mod_type)
18          w  := 1                    // (mod_type)
19
20          for j:=0 to mh-1
21          {
22              for r:=0 to n-m step m
23              {
24                  t1 := r + j
25                  t2 := t1 + mh
26
27                  v := f[t2] * w  // (mod_type)
28                  u := f[t1]      // (mod_type)
29
30                  f[t1] := u + v
31                  f[t2] := u - v
32              }
33
34              w := w * dw  // trig recursion
35          }
36      }
37  }
```

As shown in section 21.2.1 on page 412 it is a good idea to extract the `ldm==1` stage of the outermost loop: Replace

```
for ldm:=1 to ldn
{
```

by

```
for r:=0 to n-1 step 2
{
    { f[r], f[r+1] } := { f[r]+f[r+1], f[r]-f[r+1] }  // parallel assignment
}
```

```
    for ldm:=2 to ldn
    {
```

The C++ implementation is given in [FXT: ntt/nttdit2.cc]:

```
void
ntt_dit2_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dit2()
// Decimation in time (DIT) radix-2 FFT
// Input data must be in revbin_permuted order
// ldn := base-2 logarithm of the array length
// is := sign of the transform
{
    const ulong n = 1UL<<ldn;

    for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);

    for (ulong ldm=2; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);

        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r + j;
                const ulong t2 = t1 + mh;

                mod v = f[t2] * w;
                mod u = f[t1];

                f[t1] = u + v;
                f[t2] = u - v;
            }
            w *= dw;
        }
    }
}
```

```
void
ntt_dit2(mod *f, ulong ldn, int is)
// Radix-2 decimation in time (DIT) NTT
{
    revbin_permute(f, 1UL<<ldn);
    ntt_dit2_core(f, ldn, is);
}
```

The elements of order 2^k are precomputed at initialization of the `mod` class. The call to `mod::root2pow()` is a simple table lookup.

26.2.2 Radix-2 DIF NTT

Pseudocode for the radix-2 decimation in frequency (DIF) NTT:

```
procedure mod_fft_dif2(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{
    n := 2**ldn
    dw := element_of_order(n)  // (mod_type)

    if is<0 then  dw := rn**(-1)

    for  ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2

        w := 1  // (mod_type)

        for j:=0 to mh-1
        {
            for r:=0 to n-m step m
```

```
19              {
20                  t1 := r + j
21                  t2 := t1 + mh
22
23                  v := f[t2]  // (mod_type)
24                  u := f[t1]  // (mod_type)
25
26                  f[t1] :=  u + v
27                  f[t2] := (u - v) * w
28              }
29
30              w := w * dw  // trig recursion
31          }
32
33          dw := dw * dw
34      }
35
36      revbin_permute(f[], n)
37  }
```

As in section 21.2.2 on page 414 extract the `ldm==1` stage of the outermost loop: replace the line

```
for  ldm:=ldn to 1 step -1
```

by

```
for  ldm:=ldn to 2 step -1
```

and insert

```
for r:=0 to n-1 step 2
{
    { f[r], f[r+1] } := { f[r] + f[r+1], f[r] - f[r+1] }  // parallel assignment
}
```

before the call of `revbin_permute(f[],n)`.

The C++ implementation is given in [FXT: ntt/nttdif2.cc]:

```
1   void
2   ntt_dif2_core(mod *f, ulong ldn, int is)
3   // Auxiliary routine for ntt_dif2().
4   // Decimation in frequency (DIF) radix-2 NTT.
5   // Output data is in revbin_permuted order.
6   // ldn := base-2 logarithm of the array length.
7   // is := sign of the transform
8   {
9       const ulong n = (1UL<<ldn);
10      mod dw = mod::root2pow( is>0 ? ldn : -ldn );
11
12      for (ulong ldm=ldn; ldm>1; --ldm)
13      {
14          const ulong m = (1UL<<ldm);
15          const ulong mh = (m>>1);
16
17          mod w = mod::one;
18
19          for (ulong j=0; j<mh; ++j)
20          {
21              for (ulong r=0; r<n; r+=m)
22              {
23                  const ulong t1 = r + j;
24                  const ulong t2 = t1 + mh;
25
26                  mod v = f[t2];
27                  mod u = f[t1];
28
29                  f[t1] = (u + v);
30                  f[t2] = (u - v) * w;
31              }
32              w *= dw;
33          }
34          dw *= dw;
35      }
36
37      for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
38  }
```

```
1   void
2   ntt_dif2(mod *f, ulong ldn, int is)
```

```
3   // Radix-2 decimation in frequency (DIF) NTT
4   {
5       ntt_dif2_core(f, ldn, is);
6       revbin_permute(f, 1UL<<ldn);
7   }
```

26.2.3 Radix-4 NTTs

The radix-4 versions of the NTT are straightforward translations of the routines that use complex numbers. We simply give the C++ implementations

26.2.3.1 Decimation in time (DIT) algorithm

Code for a radix-4 decimation in time (DIT) NTT [FXT: ntt/nttdit4.cc]:

```
1   static const ulong LX = 2;
2
3   void
4   ntt_dit4_core(mod *f, ulong ldn, int is)
5   // Auxiliary routine for ntt_dit4()
6   // Decimation in time (DIT) radix-4 NTT
7   // Input data must be in revbin_permuted order
8   // ldn := base-2 logarithm of the array length
9   // is := sign of the transform
10  {
11      const ulong n = (1UL<<ldn);
12
13      if ( ldn & 1 )  // n is not a power of 4, need a radix-2 step
14      {
15          for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
16      }
17
18      const mod imag = mod::root2pow( is>0 ? 2 : -2 );
19
20      ulong ldm = LX + (ldn&1);
21      for ( ; ldm<=ldn ; ldm+=LX)
22      {
23          const ulong m = (1UL<<ldm);
24          const ulong m4 = (m>>LX);
25
26          const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
27          mod w = (mod::one);
28          mod w2 = w;
29          mod w3 = w;
30
31          for (ulong j=0; j<m4; j++)
32          {
33              for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
34              {
35                  const ulong i1 = i0 + m4;
36                  const ulong i2 = i1 + m4;
37                  const ulong i3 = i2 + m4;
38
39                  mod a0 = f[i0];
40                  mod a2 = f[i1] * w2;
41                  mod a1 = f[i2] * w;
42                  mod a3 = f[i3] * w3;
43
44                  mod t02 = a0 + a2;
45                  mod t13 = a1 + a3;
46
47                  f[i0] = t02 + t13;
48                  f[i2] = t02 - t13;
49
50                  t02 = a0 - a2;
51                  t13 = a1 - a3;
52                  t13 *= imag;
53
54                  f[i1] = t02 + t13;
55                  f[i3] = t02 - t13;
56              }
57
58              w *= dw;
59              w2 = w * w;
60              w3 = w * w2;
61          }
```

```
    }
}

void
ntt_dit4(mod *f, ulong ldn, int is)
// Radix-4 decimation in time (DIT) NTT
{
    revbin_permute(f, 1UL<<ldn);
    ntt_dit4_core(f, ldn, is);
}
```

26.2.3.2 Decimation in frequency (DIF) algorithm

Code for a radix-4 decimation in frequency (DIT) NTT [FXT: ntt/nttdif4.cc]:

```
static const ulong LX = 2;

void
ntt_dif4_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dif4().
// Decimation in frequency (DIF) radix-4 NTT.
// Output data is in revbin_permuted order.
// ldn := base-2 logarithm of the array length.
// is := sign of the transform
{
    const ulong n = (1UL<<ldn);

    const mod imag = mod::root2pow( is>0 ? 2 : -2 );

    for (ulong ldm=ldn; ldm>=LX; ldm-=LX)
    {
        const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w;
        mod w3 = w;

        for (ulong j=0; j<m4; j++)
        {
            for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            {
                const ulong i1 = i0 + m4;
                const ulong i2 = i1 + m4;
                const ulong i3 = i2 + m4;

                mod a0 = f[i0];
                mod a1 = f[i1];
                mod a2 = f[i2];
                mod a3 = f[i3];

                mod t02 = a0 + a2;
                mod t13 = a1 + a3;

                f[i0] = (t02 + t13);
                f[i1] = (t02 - t13) * w2;

                t02 = a0 - a2;
                t13 = a1 - a3;
                t13 *= imag;

                f[i2] = (t02 + t13) * w;
                f[i3] = (t02 - t13) * w3;
            }

            w *= dw;
            w2 = w * w;
            w3 = w * w2;
        }
    }

    if ( ldn & 1 )  // n is not a power of 4, need a radix-2 step
    {
        for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
    }
}
```

```
1   void
2   ntt_dif4(mod *f, ulong ldn, int is)
3   // Radix-4 decimation in frequency (DIF) NTT
4   {
5       ntt_dif4_core(f, ldn, is);
6       revbin_permute(f, 1UL<<ldn);
7   }
```

26.3 Convolution with NTTs

The NTTs are natural candidates for the computation of exact integer convolutions, as used in high precision multiplication algorithms. All computations are modulo m, the largest value that can be represented is $m-1$. Choosing a modulus that is greater than the maximal possible value of the result avoids any truncation.

If m does not fit into a single machine word, the modular arithmetic tends to be expensive. This may slow down the computation unacceptably. It is better to choose m as a product of mutually coprime moduli m_i that are all just below machine word size, compute the convolutions for each modulus m_i, and finally use the Chinese Remainder Theorem (see section 39.4 on page 772) to obtain the result modulo m. In [271] it is suggested to use three primes just below the word size. This method allows computing convolutions (almost) up to lengths that just fit into a machine word.

Routines for the NTT-based exact convolution are given in [FXT: ntt/nttcnvl.cc]. The routines are virtually identical to their complex equivalents given in section 22.1.2 on page 441. For example, the routine for cyclic self-convolution is

```
1   void
2   ntt_auto_convolution(mod *f, ulong ldn)
3   // Cyclic self-convolution.
4   // Use zero padded data for linear convolution.
5   {
6       assert_two_invertible();  // so we can normalize later
7
8       const int is = +1;
9
10      ntt_dif4_core(f, ldn, is);  // transform
11
12      const ulong n = (1UL<<ldn);
13      for (ulong i=0; i<n; ++i)  f[i] *= f[i];  // multiply element-wise
14
15      ntt_dit4_core(f, ldn, -is);  // inverse transform
16
17      multiply_val(f, n, (mod(n)).inv() );  // normalize
18  }
```

The revbin permutations are avoided as explained in section 22.1.3 on page 442.

For further applications of the NTT see the survey article [172] and the references given there.

Chapter 27

Fast wavelet transforms

The discrete wavelet transforms are a class of transforms that can be computed in linear time. We describe wavelet transforms whose basis functions have compact support. These are derived as a generalization of the Haar transform.

27.1 Wavelet filters

We motivate the *wavelet transform* as a generalization of the 'standard' Haar transform given in section 24.1 on page 497. The Haar transform will be reformulated as a sequence of filtering steps.

We consider only (moving average) filters F defined by n coefficients (filter *taps*) f_0, f_1, ..., f_{n-1}. Let A be the length-N sequence a_0, a_1, ..., a_{N-1}. Define $F_k(A)$ as the weighted sum

$$F_k(A) \quad := \quad \sum_{j=0}^{n-1} f_j \, a_{k+j \bmod N} \tag{27.1-1}$$

That is, $F_k(A)$ is the result of applying the filter F to the n elements a_k, a_{k+1}, a_{k+2}, ... a_{k+n-1}, possibly wrapping around.

Now assume that N is a power of 2. Let H be the low-pass filter defined by $h_0 = h_1 = +1/\sqrt{2}$ and G be the high-pass filter defined by $g_0 = +1/\sqrt{2}$, $g_1 = -1/\sqrt{2}$. A single filtering step of the Haar transform consists of

- computing the sums: $s_0 = H_0(A)$, $s_2 = H_2(A)$, $s_4 = H_4(4)$, ..., $s_{N-2} = H_{N-2}(A)$,
- computing the differences: $d_0 = G_0(A)$, $d_2 = G_2(A)$, $d_4 = G_4(4)$, ..., $d_{N-2} = G_{N-2}(A)$,
- writing the sums to the left half of A and the differences to the right half: $A = [s_0, s_2, s_4, s_6, \ldots, s_{N-2}, d_0, d_2, d_4, d_6, \ldots, d_{N-2}]$.

The Haar transform is computed by applying the filtering step to the whole sequence, then to its left half, then to its left quarter, ..., the left four elements, the left two elements. With the Haar transform no wrap-around occurs.

The analogous filtering step for the wavelet transform is obtained by defining two length-n filters H (low-pass) and G (high-pass) subject to certain conditions. We consider only filters with an even number n of coefficients.

Define the coefficients of G to be the reversed sequence of the coefficients of H with alternating signs:

$$\begin{aligned} g_0 = +h_{n-1}, \quad g_1 = -h_{n-2}, \quad g_2 = +h_{n-3}, \quad g_4 = -h_{n-3}, \quad \ldots \\ \ldots, \quad g_{n-3} = -h_2, \quad g_{n-2} = +h_1, \quad g_{n-1} = -h_0 \end{aligned} \tag{27.1-2}$$

We also require that the resulting transform is orthogonal. Let S be the matrix corresponding to one filtering step, ignoring the order:

$$S\,A \quad = \quad [s_0, d_0,\ s_2, d_2,\ s_4, d_4,\ s_6, d_6,\ \ldots,\ s_{N-2}, d_{N-2}] \tag{27.1-3}$$