

# Big Integer Arithmetic based on Fast Fourier Transforms

by [rattle](#), [.aware](#) computer security research

**Abstract.** Most asymmetric cryptographic algorithms require operations on very large subsets of the integer numbers, such as prime fields. For these purposes, we aim to implement a set of arbitrary precision integer arithmetic routines on the x86 architecture. Since addition and subtraction can trivially be implemented in linear time, we will focus on an effective multiplication algorithm. As we will see, integer division routines can then be derived from an existing, fast multiplication.

|  |           |
|--|-----------|
| <b>§1 - Discrete Fourier Transform</b>   | <b>2</b>  |
| Basic definitions: Discrete Fourier transform and discrete Convolution Theorem.  |           |
| <b>§2 - The Fast Fourier Transform</b>   | <b>4</b>  |
| A fast algorithm to calculate the Fourier transform of a given function with complex values.   |           |
| <b>§3 - Real Domain Transforms</b>   | <b>8</b>  |
| An extension of the FFT Algorithm particularly to transform real functions at the extent of only $n/2$ memory.   |           |
| <b>§4 - Arbitrary Precision Arithmetic</b>   | <b>12</b> |
| Application of the Fast Fourier Transform and the discrete Convolution Theorem to achieve an $O(n \log n)$ speed algorithm for integer multiplication. |           |
| <b>§5 - Optimization</b>   | <b>18</b> |
| Re-Implementation of the FFT in x86 Assembler. This yields a factor 2 speed improvement for all multiplications and divisions.                         |           |
| <b>§6 - Conclusion</b>   | <b>20</b> |
| <b>§7 - References</b>   | <b>20</b> |

## §1 – Discrete Fourier Transform

We will now introduce the basic terminology and the theoretical background for the further elaboration of the FFT algorithm. It is required that you have an understanding of basic calculus on  $\mathbb{C}$ . Note that it is necessary to investigate the Fourier transform of complex functions before we can turn towards  $\mathbb{R}$ .

**Definition 1.1.** We call a sequence  $f = (f_k)_{k \in \mathbb{Z}} \subset \mathbb{C}$  a discrete function of length  $n$  if and only if  $\forall k \in \mathbb{Z} : f_k = f_{k+n}$ . Thus, any index operation on  $f$  is performed modulo  $n$ . We will also write such a function as a tuple  $f \hat{=} (f_0, \dots, f_{n-1}) \in \mathbb{C}^n$  by which it is uniquely defined. We will also refer to these tuples as “functions of length  $n$ ”.

**Definition 1.2.** From now on and for the rest of this paper, let always  $\omega_n := e^{2i\pi/n}$ . We note that  $(\omega_n^k)_{k \in \mathbb{Z}}$  is a discrete function of length  $n$ , because  $\omega_n^n = e^{2\pi i} = 1$ . Also,  $\omega_{2m}^m = e^{\pi i} = -1$ .

**Definition 1.3.** Let  $f$  be a function of length  $n$ . We then define the discrete Fourier transform of  $f$  to be the function  $\mathcal{F}_n(f)$  of length  $n$  with

$$\mathcal{F}_n(f)_j := \frac{1}{n} \cdot \sum_{k=0}^{n-1} (f_k \cdot \omega_n^{-kj})$$

We note that the operator  $\mathcal{F}_n : \mathbb{C}^n \rightarrow \mathbb{C}^n$  is *linear*, which means we have  $\forall \lambda \in \mathbb{C} : \forall a, b \in \mathbb{C}^n :$

$\mathcal{F}_n(a + \lambda b) = \mathcal{F}_n(a) + \lambda \cdot \mathcal{F}_n(b)$ . We now also define the inverse Fourier transform of  $f$  to be the function  $\mathcal{F}_n^{-1}(f)$  with

$$\mathcal{F}_n^{-1}(f)_j := \sum_{k=0}^{n-1} (f_k \cdot \omega_n^{kj}).$$

**Lemma 1.4.** We have  $\forall f \in \mathbb{C}^n : \mathcal{F}_n^{-1}(\mathcal{F}_n(f)) = f$ .

**Proof.** We first write

$$\mathcal{F}_n^{-1}(\mathcal{F}_n(f))_j = \sum_{l=0}^{n-1} (\mathcal{F}_n(f)_l \cdot \omega_n^{lj}) = \frac{1}{n} \cdot \sum_{l=0}^{n-1} \sum_{k=0}^{n-1} f_k \omega_n^{-kl} \cdot \omega_n^{lj} = \frac{1}{n} \cdot \sum_{k=0}^{n-1} f_k \cdot \underbrace{\sum_{l=0}^{n-1} \omega_n^{(k-j)l}}_{\gamma(k,j)} \quad (*)$$

We note  $\gamma(k, k) = \sum_{l=0}^{n-1} \omega_n^{(k-k)l} = \sum_{l=0}^{n-1} \omega_n^0 = \sum_{l=0}^{n-1} 1 = n$ . On the other hand, let  $p := k - j \neq 0$  and we get

$$\gamma(k, j) = \sum_{l=0}^{n-1} \omega_n^{(k-j)l} = \sum_{l=0}^{n-1} (\omega_n^p)^l = \frac{1 - (\omega_n^p)^n}{1 - \omega_n^p} = \frac{1 - (\omega_n^n)^p}{1 - \omega_n^p} = \frac{1 - 1}{1 - \omega_n^p} = 0$$

Thus,  $\gamma(k, j) = n \cdot \delta_{kj}$  where  $\delta_{kj}$  is the Kronecker delta. This means that in  $(*)$ , we actually have

$$\mathcal{F}_n^{-1}(\mathcal{F}_n(f))_j = \frac{1}{n} \cdot \sum_{k=0}^{n-1} (f_k \cdot \delta_{kj} \cdot n) = \sum_{k=0}^{n-1} (f_k \cdot \delta_{kj}) = f_j \quad \square$$

**Definition 1.5.** We also define the discrete convolution of two functions  $f$  and  $g$  of length  $n$  to be the function  $f * g$  of length  $n$  with

$$(f * g)_j := \frac{1}{n} \cdot \sum_{k=0}^{n-1} f_k \cdot g_{j-k}$$

**Note 1.6.** The multiplication of two big integers, stored as tuples of  $p$ -bit words, will reduce to calculating a discrete convolution. We will later see that we can calculate the Fourier transform in  $O(N \cdot \log(N))$ . Thus, the following result is of great importance for the development of a fast multiplication algorithm.

**Lemma 1.7.** Let  $a, b \in \mathbb{C}[X]$  be two polynomials of degree  $\deg(a) = \deg(b) = n$ ,

$a(X) = \sum_{k=0}^n a_k X^k$  and  $b(X) = \sum_{k=0}^n b_k X^k$ , say. Then their product can be written as

$$(a \cdot b)(X) = \sum_{k=0}^n \sum_{l=0}^k a_l b_{k-l} X^k + \sum_{k=n+1}^{2n} \sum_{l=k-n}^n a_l b_{k-l} X^k$$

**Proof.** We know that  $(a \cdot b)(X) = a(X)b(X) = \sum_{k=0}^n \sum_{j+l=k} a_l b_j X^k$ . Now we can write

$$\sum_{j+l=k} a_l b_j X^k = \sum_{l=0}^k a_l b_{k-l} X^k \quad \text{for } k \leq n \text{ and}$$

$$\sum_{j+l=k} a_l b_j X^k = \sum_{l=k-n}^n a_l b_{k-l} X^k \quad \text{for } k > n,$$

which is just what we claimed. □

**Theorem 1.8 (Discrete Convolution Theorem).** Let  $f$  and  $g$  be two discrete functions of length  $n$ .

We can then say that  $\mathcal{F}_n(f * g) = \mathcal{F}_n(f) \cdot \mathcal{F}_n(g)$ , where the multiplication is to be understood component-wise.

**Proof.** We simply write  $N := 2n - 2$  and apply 1.7:

$$\begin{aligned} \mathcal{F}_n(f)_j \cdot \mathcal{F}_n(g)_j &= \frac{1}{n^2} \cdot \sum_{k=0}^{n-1} f_k \omega_n^{-jk} \cdot \sum_{k=0}^{n-1} g_k \omega_n^{-jk} \\ &= \frac{1}{n^2} \left( \sum_{k=0}^{n-1} \sum_{l=0}^k f_l g_{k-l} \omega_n^{-jk} + \sum_{k=n}^N \sum_{l=k-(n-1)}^{n-1} f_l g_{k-l} \omega_n^{-jk} \right) && \text{Polynomials in } X := \omega_n^{-j} \\ &= \frac{1}{n^2} \left( \sum_{k=0}^{n-1} \sum_{l=0}^k f_l g_{k-l} \omega_n^{-jk} + \sum_{k=0}^{n-2} \sum_{l=k+1}^{n-1} f_l g_{k-l+n} \omega_n^{-j(k+n)} \right) && \text{Shift index by } n \text{ (second sum)} \\ &= \frac{1}{n^2} \left( \sum_{k=0}^{n-1} \sum_{l=0}^k f_l g_{k-l} \omega_n^{-jk} + \sum_{k=0}^{n-2} \sum_{l=k+1}^{n-1} f_l g_{k-l} \omega_n^{-jk} \right) && \omega_n \text{ is discrete function} \\ &= \frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} f_l g_{k-l} \omega_n^{-jk} = \frac{1}{n} \sum_{k=0}^{n-1} (f * g)_k \omega_n^{-jk} = \mathcal{F}_n(f * g)_j \end{aligned}$$

□

## §2 – The Fast Fourier Transform

Judging by (1.3), one might assume that calculating the Fourier transform of a given function  $f$  is to be implemented in  $O(N^2)$  time. However, by deploying a divide-and-conquer-strategy, we can construct an Algorithm to compute the Fourier transform of a function  $f$  of length  $N = 2^p$  in  $O(N \cdot \log(N))$  steps. Implementing this algorithm will be the ultimate goal of this chapter.

**Lemma 2.1.** Let  $f$  be a discrete function of length  $n = 2 \cdot m$ . Given the two discrete functions of length  $m$   $f_k^0 := f_{2k}$  (even indices) and  $f_k^1 := f_{2k+1}$  (odd indices), we get

$$2 \cdot \mathcal{F}_n(f)_j = \mathcal{F}_m(f^0)_j + \omega_n^{-j} \cdot \mathcal{F}_m(f^1)_j$$

**Proof.** First note that  $\omega_n^{2k} = \exp\left(2k \cdot \frac{2i\pi}{n}\right) = \exp\left(\frac{k2i\pi}{m}\right) = \omega_m^k$ . Hence, we write

$$\begin{aligned} 2 \cdot \mathcal{F}_n(f)_j &= \frac{2}{n} \cdot \sum_{k=0}^{n-1} f_k \omega_n^{-kj} = \frac{2}{2m} \cdot \sum_{k=0}^{m-1} \left[ f_{2k} \omega_n^{-2kj} + f_{2k+1} \omega_n^{-(2k+1)j} \right] \\ &= \left[ \frac{1}{m} \cdot \sum_{k=0}^{m-1} f_k^0 \omega_m^{-kj} \right] + \left[ \frac{1}{m} \cdot \sum_{k=0}^{m-1} f_k^1 \omega_m^{-kj} \right] \omega_n^{-j} = \mathcal{F}_m(f^0)_j + \omega_n^{-j} \cdot \mathcal{F}_m(f^1)_j \end{aligned}$$

□

**Corollary 2.2.** From the proof, we observe that we have a similar identity for the inverse discrete Fourier transform:  $\mathcal{F}_n^{-1}(f)_j = \mathcal{F}_m^{-1}(f^0)_j + \omega_n^j \mathcal{F}_m^{-1}(f^1)_j$ . The proof works identically. Also:

- $2 \cdot \mathcal{F}_m^{-1}(f^0)_j = \mathcal{F}_n^{-1}(f)_j + \mathcal{F}_n^{-1}(f)_{j+m}$
- $2 \cdot \mathcal{F}_m^{-1}(f^1)_j = \omega_n^{-j} \left( \mathcal{F}_n^{-1}(f)_j - \mathcal{F}_n^{-1}(f)_{j+m} \right)$

**Proof.** First, write

$$\begin{aligned} \mathcal{F}_n^{-1}(f)_j &= \mathcal{F}_m^{-1}(f^0)_j + \omega_n^j \mathcal{F}_m^{-1}(f^1)_j \\ \Rightarrow \mathcal{F}_n^{-1}(f)_{j+m} &= \mathcal{F}_m^{-1}(f^0)_j - \omega_n^j \mathcal{F}_m^{-1}(f^1)_j \end{aligned}$$

Hence,  $\mathcal{F}_m^{-1}(f^0)_j = \mathcal{F}_n^{-1}(f)_j - \omega_n^j \mathcal{F}_m^{-1}(f^1)_j = \mathcal{F}_n^{-1}(f)_j + \mathcal{F}_n^{-1}(f)_{j+m} - \mathcal{F}_m^{-1}(f^0)_j$  which yields

$$2 \cdot \mathcal{F}_m^{-1}(f^0)_j = \mathcal{F}_n^{-1}(f)_j + \mathcal{F}_n^{-1}(f)_{j+m}$$

Then  $\mathcal{F}_m^{-1}(f^1)_j = \omega_n^{-j} \left( \mathcal{F}_n^{-1}(f)_j - \mathcal{F}_m^{-1}(f^0)_j \right) = \omega_n^{-j} \left( \mathcal{F}_n^{-1}(f)_j - \mathcal{F}_n^{-1}(f)_{j+m} - \omega_n^j \mathcal{F}_m^{-1}(f^1)_j \right)$ , which gives us

$$2 \cdot \mathcal{F}_m^{-1}(f^1)_j = \omega_n^{-j} \left( \mathcal{F}_n^{-1}(f)_j - \mathcal{F}_n^{-1}(f)_{j+m} \right)$$

□

**Corollary 2.3.** Of course, we can obtain a similar result for the forward transform:

- $\tilde{\mathcal{F}}_m(f^0)_j = \tilde{\mathcal{F}}_n(f)_j + \tilde{\mathcal{F}}_n(f)_{j+m}$
- $\tilde{\mathcal{F}}_m(f^1)_j = \omega_n^j \left( \tilde{\mathcal{F}}_n(f)_j - \tilde{\mathcal{F}}_n(f)_{j+m} \right)$

**Definition 2.4.** Let  $f$  be a discrete function of length  $2^p$ . Let also  $q \leq p \in \mathbb{N}$  and  $\kappa \in \mathbb{F}_2^q$ .<sup>1</sup>

We then recursively define the function  $f^\kappa$  of length  $2^{p-q}$  as follows:

- For any  $\lambda \in \mathbb{F}_2$ , define  $f^{(\lambda)}$  by  $f_k^\lambda := f_{2k+\lambda}$
- Recursively set  $f^{(\kappa_1, \dots, \kappa_k)} := \left( f^{(\kappa_1, \dots, \kappa_{k-1})} \right)^{(\kappa_k)}$

We also write  $f^{\kappa_1, \dots, \kappa_q}$  for  $f^{(\kappa_1, \dots, \kappa_q)}$ , omitting the brackets. Note that this is completely equivalent to the Notation used in **2.1**.

**Note 2.5.** If we have a given  $h \in \mathbb{C}^n$  and our goal is to calculate  $f := \tilde{\mathcal{F}}_n(h)$ , we can also write  $h = \tilde{\mathcal{F}}_n^{-1}(f)$ . We then split up  $f$  into its even and odd components  $f^0, f^1$  and calculate the inverse Fourier transforms of these functions by **2.2**. Proceeding recursively, we obtain the inverse Fourier transforms of  $f^\kappa$  for all  $\kappa \in \mathbb{F}_2^p$ . Since  $f^\kappa$  is a function of length 1, we simply have

$\tilde{\mathcal{F}}_1^{-1}(f^\kappa) = f_{\sigma(\kappa)}$ . Thus, all we need to do is figure out the index permutation  $\sigma$  and set  $f_i := f_{\sigma^{-1}(i)}$  for all  $i = 0, \dots, n-1$  to obtain the discrete Fourier transform of  $h$ .

Let from now on  $M_p := \{0, \dots, 2^p - 1\}$  the set of indices for our discrete function. There is a canonical Isomorphism  $\mathbb{F}_2^p \rightarrow M_p$  by  $\kappa = (\kappa_0, \kappa_1, \dots, \kappa_{p-1}) \mapsto \sum_{v=0}^{p-1} \kappa_v 2^v$ .

**Definition 2.6.** The permutation  $\sigma_p : \mathbb{F}_2^p \rightarrow M$ ,  $\kappa = (\kappa_0, \dots, \kappa_{p-1}) \mapsto \sigma_p(\kappa) := \sum_{k=1}^p \kappa_{p-k} 2^k$  is called the bit reversal of length  $p$ . Note that  $\sigma_p^{-1} = \sigma_p$ .

**Lemma 2.7.** Let  $n = 2^p$  with  $p \in \mathbb{N}$  and let  $f$  be a function of length  $n$ . If we apply corollary **2.2** to  $f$  recursively, we obtain  $\tilde{\mathcal{F}}_1^{-1}(f^\kappa) = f_{\sigma_p(\kappa)}$  for all  $\kappa \in \mathbb{F}_2^p$ , where  $\sigma_p$  is the bit reversal of length  $p$ .

**Proof.** Induction by  $p$ . The statement is obviously correct for  $p = 1$ . Hence, let  $p \geq 2$ . Let  $\kappa = (\lambda, \kappa_1, \dots, \kappa_{p-1}) \in \mathbb{F}_2^p$  and set  $\mathcal{G} := (\kappa_1, \dots, \kappa_{p-1}) \in \mathbb{F}_2^{p-1}$ . Due to the induction hypothesis, the lemma is correct for  $f^0$  and  $f^1$ . Thus, we have

$$\tilde{\mathcal{F}}_1^{-1}(f^\kappa) = \tilde{\mathcal{F}}_1^{-1}(f^{\lambda, \mathcal{G}}) = f_{\sigma_{p-1}(\mathcal{G})}^\lambda = f_{2 \cdot \sigma_{p-1}(\mathcal{G}) + \lambda} = f_{\sigma_p(\lambda, \mathcal{G})} = f_{\sigma_p(\kappa)}$$

which completes the proof. □

---

<sup>1</sup>  $\mathbb{F}_2 = \{0, 1\}$  is the smallest field.  $\mathbb{F}_2^p = \{(\kappa_0, \dots, \kappa_{p-1}) \mid \forall i: \kappa_i \in \mathbb{F}_2\}$  all  $p$ -tuples of Elements in  $\mathbb{F}_2$ .

**Algorithm 2.8 – Fast Fourier Transform.**

**Input:** A tuple  $f = (f_0, \dots, f_{n-1}) \in \mathbb{C}^n$  with  $n = 2^p$  for some  $p \in \mathbb{N}$  and  $s \in \{1, -1\}$ .

**Output:** If  $s = -1$ , returns the discrete Fourier transform  $\mathcal{F}_n(f)$ . If  $s = 1$ , return  $\mathcal{F}_n^{-1}(f)$ .

```

(1)   SET  $\omega := \exp(s \cdot 2\pi i/n)$ 
(2)   IF  $s < 0$  THEN
(3)       FOR  $j := 0, \dots, n-1$  DO SET  $f_j := f_j/n$ 
(4)   FOR  $l := p, \dots, 1$  DO
(5)       SET  $N := 2^l$ ;  $M := 2^{l-1}$ ;  $R := 2^{p-l}$ 
(6)       FOR  $g = 0, \dots, R-1$  DO
(7)           SET  $z := 1$ 
(8)           FOR  $k = 0, \dots, M-1$  DO
(9)               SET  $a := (f_{g \cdot N+k} - f_{g \cdot N+k+M}) \cdot z$ 
(10)              SET  $f_{g \cdot N+k} := f_{g \cdot N+k} + f_{g \cdot N+k+M}$ 
(11)              SET  $f_{g \cdot N+k+M} := a$ 
(12)              SET  $z := z \cdot \omega$ 
(13)       SET  $\omega := \omega^2$ 
(14)   FOR  $g = 0, \dots, n-1$  DO
(15)       IF  $g < \sigma_p(g) =: g'$  THEN EXCHANGE  $f_g \leftrightarrow f_{g'}$ 

```

**Theorem 2.9 (Fast Fourier Transform Algorithm).** The FFT Algorithm works correctly and terminates after  $O(n \cdot \log(n))$  steps.

**Proof.** In each iteration of **(4)**, the array  $f$  changes completely. For the purpose of this proof, we will treat them as *distinct* arrays  $f = F^{(p)}, F^{(p-1)}, \dots, F^{(1)}$ . Note that  $F^{(l)}$  is essentially an array of  $N = 2^l$  sub-arrays of length  $R$ . Hence, we have  $F^{(l)} \hat{=} (F^{(l,0)}, \dots, F^{(l,R-1)})$  where we can say that  $F_k^{(l,g)} = F_{g \cdot N+k}^{(l)} \hat{=} \mathcal{F}_N^{-1}(f^g)_k$ . The correctness of the algorithm then follows with **2.2** and **2.7**. Runtime is obvious since  $\sigma_p$  can be implemented in  $O(n \cdot \log(n))$ , as the following code illustrates.

□

---

```

typedef struct _COMPLEX {
    double real;
    double imag;
} COMPLEX, * PCOMPLEX;

COMPLEX* fft(COMPLEX* f, unsigned long n, double s) {

    unsigned long k,j,M,N=n;
    double tmpval;
    COMPLEX w,z,a;

    s /= fabs(s);
    tmpval = s * 6.2831853071795862 / n;
    w.real = cos(tmpval);
    w.imag = sin(tmpval);

    // scaling for the forward transform
    if (s < 0) for (j=0; j<n; j++) {
        f[j].imag /= n;
        f[j].real /= n; }

    while (N > 1) { M = N>>1;

        for (j=0; j<n; j+=N)
        {
            z.real = 1;
            z.imag = 0;

            for (k=0; k<M; k++)
            {
                a = f[j+k];
                a.real -= f[j+k+M].real;
                a.imag -= f[j+k+M].imag;

                a.real = (tmpval = a.real)*z.real - a.imag*z.imag;
                a.imag = a.imag*z.real + tmpval*z.imag;

                f[j+k].real += f[j+k+M].real;
                f[j+k].imag += f[j+k+M].imag;

                f[j+k+M] = a;

                z.real = (tmpval = z.real)*w.real - z.imag*w.imag;
                z.imag = z.imag*w.real + tmpval*w.imag;
            }
        }

        w.real = (tmpval=w.real)*w.real - w.imag*w.imag;
        w.imag *= tmpval; w.imag += w.imag;
        N = M;
    }

    for (j=k=0; k<n; k++) {
        if (j<k) {
            a = f[j];
            f[j] = f[k];
            f[k] = a;
        }
        for (N=n; j<N; j^=N)
            N >>= 1;
    }
    return f;
}

```

---

### §3 – Real Domain Transforms

Since we will only transform discrete functions  $f \in \mathbb{R}^n \subset \mathbb{C}^n$  during a big integer multiplication, we first aim to find a way to implement this transform more effectively than embedding  $f$  in  $\mathbb{C}^n$ .

**Lemma 3.1** Let  $f$  be a discrete function of length  $n = 2m$ . We then have

$$(i) \quad \overline{\mathcal{F}_n(f)_k} = \mathcal{F}_n(\bar{f})_{n-k} \text{ and hence also } \overline{\mathcal{F}_n(f)_{n-k}} = \mathcal{F}_n(\bar{f})_k.$$

(ii) Let also  $f \in \mathbb{R}^n$  and  $g := f^0 + f^1 i \in \mathbb{C}^m$ . Then:

$$4 \cdot \mathcal{F}_n(f)_k = \left( \mathcal{F}_m(g)_k + \overline{\mathcal{F}_m(g)_{m-k}} \right) - \left( \mathcal{F}_m(g)_k - \overline{\mathcal{F}_m(g)_{m-k}} \right) i \omega_n^{-k}$$

**Proof.**

$$(i) \quad n \cdot \overline{\mathcal{F}_n(f)_j} = \sum_{k=0}^{n-1} \overline{f_k \omega_n^{-kj}} = \sum_{k=0}^{n-1} \overline{f_k} \overline{\omega_n^{-kj}} = \sum_{k=0}^{n-1} \overline{f_k} \omega_n^{kj} = \sum_{k=0}^{n-1} \overline{f_k} \omega_n^{-nk+kj} = \sum_{k=0}^{n-1} \overline{f_k} \omega_n^{-(n-j)k} = n \cdot \mathcal{F}_n(\bar{f})_{n-j}$$

$$(ii) \quad \frac{\mathcal{F}_m(g)_k + \overline{\mathcal{F}_m(g)_{m-k}}}{4} + \frac{\mathcal{F}_m(g)_k - \overline{\mathcal{F}_m(g)_{m-k}}}{4i} \omega_n^{-k} = \frac{\mathcal{F}_m(g)_k + \mathcal{F}_m(\bar{g})_k}{4} + \frac{\mathcal{F}_m(g)_k - \mathcal{F}_m(\bar{g})_k}{4i} \omega_n^{-k}$$

$$= \frac{1}{2} \cdot \left( \mathcal{F}_m\left(\frac{1}{2}(g + \bar{g})\right)_k + \mathcal{F}_m\left(\frac{1}{2i}(g - \bar{g})\right)_k \cdot \omega_n^{-k} \right) = \frac{1}{2} \cdot \left( \mathcal{F}_m(\operatorname{Re} g)_k + \mathcal{F}_m(\operatorname{Im} g)_k \cdot \omega_n^{-k} \right)$$

$$= \frac{1}{2} \cdot \left( \mathcal{F}_m(f^0)_k + \mathcal{F}_m(f^1)_k \cdot \omega_n^{-k} \right) = \mathcal{F}_n(f)_k$$

by (i) and Lemma 2.1. □

**Note 3.2.** This lemma outlines how we can implement a real Fourier transform without additional memory usage: Due to  $\overline{\mathcal{F}_n(f)_k} = \mathcal{F}_n(\bar{f})_{n-k} = \mathcal{F}_n(f)_{n-k}$ , saving  $\mathcal{F}_n(f)_0, \dots, \mathcal{F}_n(f)_m$  is sufficient. The second statement also allows us to reuse Algorithm 2.8 for the real Fourier transform by applying it to  $\tilde{g} := (f_{2k} + i \cdot f_{2k+1})_{k=0}^{m-1} \in \mathbb{C}^m$  and then calculating the first  $m+1$  components of  $\mathcal{F}_n(f)$  from it. We will see that both  $\mathcal{F}_n(f)_m$  and  $\mathcal{F}_n(f)_0$  are real, hence we can store them in one complex variable.

For the following considerations, we agree on  $x^I := \operatorname{Im}(x)$  and  $x^R := \operatorname{Re}(x)$  for readability. Let  $g$  be the Fourier transform of  $\tilde{g}$ . Let  $\varepsilon_k = \omega_n^{-k}$ . We note that  $\varepsilon_m = \omega_{2m}^{-m} = e^{-2m\pi i/2m} = e^{-\pi i} = -1$ .

Particularly, we have  $\varepsilon_{m-k} = \omega_{2m}^{k-m} = -\omega_{2m}^k = -\overline{\omega_{2m}^{-k}} = -\overline{\varepsilon_k}$  which yields  $\varepsilon_{m-k}^I = \varepsilon_k^I$  and  $\varepsilon_{m-k}^R = -\varepsilon_k^R$ . Now, set

- $a_k := g_k^R + g_{m-k}^R$
- $b_k := g_k^R - g_{m-k}^R$
- $c_k := g_k^I + g_{m-k}^I$
- $d_k := g_k^I - g_{m-k}^I$

We then have  $a_{m-k} = a_k$ ,  $c_{m-k} = c_k$ ,  $b_{m-k} = -b_k$  and  $d_{m-k} = -d_k$ . In particular, we note that  $d_0 = b_0 = d_m = b_m = 0$ . Hence:

$$\begin{aligned}
4 \cdot f_k &= (g_k + \overline{g_{m-k}}) - (g_k - \overline{g_{m-k}}) i \varepsilon_k \\
&= (g_k^R + g_{m-k}^R) + (g_k^I - g_{m-k}^I) i - (g_k^R - g_{m-k}^R) i \varepsilon_k - (g_k^I + g_{m-k}^I) i^2 \varepsilon_k \\
&= (g_k^R + g_{m-k}^R) + (g_k^I - g_{m-k}^I) i - (g_k^R - g_{m-k}^R) (\varepsilon_k^R + i \varepsilon_k^I) i + (g_k^I + g_{m-k}^I) (\varepsilon_k^R + i \varepsilon_k^I) \\
&= (g_k^R + g_{m-k}^R) + (g_k^R - g_{m-k}^R) \varepsilon_k^I + (g_k^I + g_{m-k}^I) \varepsilon_k^R \\
&\quad + \left[ (g_k^I - g_{m-k}^I) - (g_k^R - g_{m-k}^R) \varepsilon_k^R + (g_k^I + g_{m-k}^I) \varepsilon_k^I \right] \cdot i
\end{aligned}$$

thus  $4 \cdot f_k = (a_k + b_k \varepsilon_k^I + c_k \varepsilon_k^R) + (d_k - b_k \varepsilon_k^R + c_k \varepsilon_k^I) i$

and  $4 \cdot f_{m-k} = (a_k - b_k \varepsilon_k^I - c_k \varepsilon_k^R) + (-d_k - b_k \varepsilon_k^R + c_k \varepsilon_k^I) i$ .

Consider  $m/2 \in \mathbb{N}$ . We have  $b_{m/2} = d_{m/2} = 0$ , and also  $\varepsilon_{m/2} = e^{-\pi i/2} = -i$  which gives us

$$2 \cdot f_{m/2} = \frac{1}{2} (a_{m/2} - c_{m/2} \cdot i) = \overline{g_{m/2}}. \text{ As a final note, we have}$$

$$\text{Im}(4f_0) = d_0 - b_0 \varepsilon_0^R + c_0 \varepsilon_0^I = d_0 - b_0 = 0 = -d_0 - b_0 = \text{Im}(4f_m)$$

Hence  $4f_0 = a_0 + b_0 \varepsilon_0^I + c_0 \varepsilon_0^R = 2g_0^R + 2g_0^I$  and  $4f_m = a_0 - b_0 \varepsilon_0^I - c_0 \varepsilon_0^R = 2g_0^R - 2g_0^I$ . We set  $F_0 := f_0 + f_m \cdot i$  and  $F_k := f_k$  for  $k > 0$  in each iteration. We can now implement a real Fourier transform as hoped, but we still need a convenient method for the inverse transform:

**Lemma 3.3.** Let conditions as in 3.1 (ii). Then the identity

$$\tilde{\mathcal{F}}_m(g)_k = \left( \tilde{\mathcal{F}}_n(f)_k + \overline{\tilde{\mathcal{F}}_n(f)_{m-k}} \right) + \left( \tilde{\mathcal{F}}_n(f)_k - \overline{\tilde{\mathcal{F}}_n(f)_{m-k}} \right) i \omega_n^k$$

holds for all  $k = 0, \dots, m-1$ .

**Proof.** By 2.3, we have

$$\begin{aligned}
\tilde{\mathcal{F}}_m(g)_k &= \tilde{\mathcal{F}}_m(f^0 + i \cdot f^1)_k = \tilde{\mathcal{F}}_m(f^0)_k + i \cdot \tilde{\mathcal{F}}_m(f^1)_k \\
&= \tilde{\mathcal{F}}_n(f)_k + \tilde{\mathcal{F}}_n(f)_{k+m} + i \omega_n^k (\tilde{\mathcal{F}}_n(f)_k - \tilde{\mathcal{F}}_n(f)_{k+m}) \\
&= \tilde{\mathcal{F}}_n(f)_k + \overline{\tilde{\mathcal{F}}_n(f)_{n-m-k}} + i \omega_n^k (\tilde{\mathcal{F}}_n(f)_k - \overline{\tilde{\mathcal{F}}_n(f)_{n-m-k}}) \\
&= \tilde{\mathcal{F}}_n(f)_k + \overline{\tilde{\mathcal{F}}_n(f)_{m-k}} + i \omega_n^k (\tilde{\mathcal{F}}_n(f)_k - \overline{\tilde{\mathcal{F}}_n(f)_{m-k}})
\end{aligned}$$

□

**Continuation of Note 3.2.** For the real backward transform, we hence get a very similar formula with  $\varepsilon_k := \omega_n^k$ , because all we have to do is flip signs in front of  $\varepsilon_k^R$  and  $\varepsilon_k^I$  in order to extract the backwards transform formula from the one for the forward transform we already have. In the definition of  $a_k, b_k, c_k, d_k$  we replace  $g$  with  $f$  and get:

- $g_k = (a_k - b_k \varepsilon_k^I - c_k \varepsilon_k^R) + (d_k + b_k \varepsilon_k^R - c_k \varepsilon_k^I) i$
- $g_{m-k} = (a_k + b_k \varepsilon_k^I + c_k \varepsilon_k^R) + (-d_k + b_k \varepsilon_k^R - c_k \varepsilon_k^I) i$

Since we now have  $\varepsilon_{m/2} = i$ , we get  $g_{m/2} = a_{m/2} - c_{m/2} i = 2 \overline{f_{m/2}}$  and because of

$\text{Im}(f_0) = \text{Im}(f_m) = 0$ , we have  $c_0 = d_0 = 0$ . This yields  $g_0 = a_0 + b_0 i = (f_0^R + f_m^R) + (f_0^R - f_m^R) i$  and thus:

- $g_0 = (F_0^R + F_0^I) + (F_0^R - F_0^I) i$

Because it is of little interest and relatively complex, we skip a formal description of this modification to the FFT algorithm and head directly to the implementation. Due to the detailed construction of the method above, it should be easy to understand nonetheless.

For  $s < 0$ , the function expects the input  $g = f^0 + i \cdot f^1$  and returns  $(F_0, \dots, F_{m-1}) \in \mathbb{C}^m$  where

$$F_i = \begin{cases} \mathcal{F}_n(f)_0 + i \cdot \mathcal{F}_n(f)_m & ; i=0 \\ \mathcal{F}_n(f)_i & ; i \neq 0 \end{cases}$$

For  $s > 0$ , it expects a Fourier transform of a real function in the same format and returns the original  $g = f^0 + i \cdot f^1 \in \mathbb{C}^m \cong \mathbb{R}^n$ .

---

```

COMPLEX* realfft(COMPLEX* g, unsigned long n, double s) {
    unsigned long k,N=n>>1;
    double tmpval;
    double a,b,c,d;
    COMPLEX w,e;

    s /= fabs(s);
    tmpval = s * 3.1415926535897931 / n;
    e.real = w.real = cos(tmpval);
    e.imag = w.imag = sin(tmpval);

    if (s < 0) {
        fft(g,n,s);

        g[0].real /= 2;
        g[0].imag /= 2;
        g[N].real /= 2;
        g[N].imag = -g[N].imag / 2;

        for (k=1;k<N;k++) {
            g[k].real /= 4;
            g[k].imag /= 4;
        }
        for (k=n-1;k>N;k--) {
            g[k].real /= 4;
            g[k].imag /= 4;
        }
    } else {
        g[N].real = 2*g[N].real;
        g[N].imag = -2*g[N].imag;
    }

    for (s=-s,k=1; k<N; k++)
    {
        a = (g[k].real + g[n-k].real);
        b = (g[k].real - g[n-k].real) * s;
        c = (g[k].imag + g[n-k].imag) * s;
        d = (g[k].imag - g[n-k].imag);

        g[k].real = a + b*e.imag + c*e.real;
        g[k].imag = d - b*e.real + c*e.imag;
        g[n-k].real = a - b*e.imag - c*e.real;
        g[n-k].imag = c*e.imag - b*e.real - d;

        e.real = (tmpval=e.real)*w.real - e.imag*w.imag;
        e.imag = e.imag *w.real + tmpval*w.imag;
    }

    g[0].real = (tmpval=g[0].real) + g[0].imag;
    g[0].imag = tmpval - g[0].imag;

    return (s<0) ? fft(g,n,1.) : g;
}

```

---

## §4 – Arbitrary Precision Arithmetic

We will treat integers as bitstrings, stored in arrays of machine words. For theoretical purposes, we will refer to  $D = 2^p$  as the base of representation, namely the largest integer number not representable in one unsigned machine word.

In code, integers are stored as arrays of 16-bit words. To represent signed numbers, we use the two's complement. However, both multiplication and division are **exclusively** unsigned, due to the used FFT method limits. As mentioned before, integer addition and subtraction are trivially implemented in linear time. For now, consider `iAdd`, `iNeg`, `iSub`, `iLShift`, `iRShift` to be already provided.

**Multiplication.** The multiplication of two big integers reduces to calculating a discrete convolution.

Given two integers  $a = \sum_{k=0}^{n-1} a_k D^k$  and  $b = \sum_{k=0}^{n-1} b_k D^k$ , we set  $N := 2n$  and  $a_k, b_k := 0$  for  $N > k \geq n$ . Calculating the product  $c := a \cdot b$  can be written as:

$$\begin{aligned}
 a \cdot b &= \sum_{k=0}^{n-1} a_k D^k \cdot \sum_{k=0}^{n-1} b_k D^k = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} a_k b_j D^{k+j} \\
 &= \sum_{k=0}^{n-1} \sum_{j=0}^k a_j b_{k-j} D^k + \sum_{k=n}^{N-1} \sum_{j=k-(n-1)}^{N-1} a_j b_{k-j} D^k && \text{Product of polynomials, see 1.7} \\
 &= \sum_{k=0}^{n-1} \sum_{j=0}^{N-1} a_j b_{k-j} D^k + \sum_{k=n}^{N-1} \sum_{j=0}^{N-1} a_j b_{k-j} D^k && \text{By definition, } a_k, b_k := 0 \text{ for } N > k \geq n \\
 &= \sum_{k=0}^{N-1} \left( \sum_{j=0}^{N-1} a_j b_{k-j} \right) D^k = \sum_{k=0}^{N-1} c_k D^k && \text{with } c := \left( \sum_{j=0}^{N-1} a_j b_{k-j} \right)_{k=0}^N = N \cdot (a * b)
 \end{aligned}$$

Calculating  $a * b$  can be done in  $O(n \cdot \log(n))$ , if  $n$  is an integer power of 2, by applying the discrete convolution theorem and using **3.4** to calculate the fourier transforms.

The Algorithm calculates the Product  $a \cdot b$  of two integer numbers stored as one  $n$ –length and one  $m$ –length array of 16 bit words. The result is returned in the first  $m+n$  places of a buffer which is required to have size  $4 \cdot 2^{\lceil \log(\max(n,m)) \rceil} \cdot \text{sizeof}(\text{double})$ . This size is required for the double precision float vectors used for the FFT.

---

```

void iMul(
    double *A,      /* [OUT] A buffer of length multsize, which will contain the
                    result of the multiplication. */
    ulong multsize, /* [IN] Required size in bytes for the output buffer:
                    This is always equal to  $4 \cdot 2^{\lceil \log(\max(\text{size}_a, \text{size}_b)) \rceil} \cdot \text{sizeof}(\text{double})$  */
    ushort *a,     /* [IN] The first integer as an array of shorts */
    ushort *b,     /* [IN] The second integer */
    ulong size_a,  /* [IN] Length of the first integer (size in words) */
    ulong size_b) /* [IN] Length of the second integer */
{
    double *B,T,C;

    /* The output buffer will be used as two double arrays to store the two
       big integers as double arrays respectively. */
    ulong k, fftsize = multsize/2;
    B = &A[multsize];

    /* Fill the two double arrays with the data from the big integer arrays */

    for (k=0;k<size_b;k++) {
        A[k] = (double) a[k];
        B[k] = (double) b[k];
    }
    for (k=size_b; k<size_a; k++) {
        A[k] = (double) a[k];
        B[k] = 0.;
    }
    for (; k<multsize; k++) A[k]=B[k]=0.;

    realfft((COMPLEX*)A,fftsize,-1); /* Transform the first vector */
    realfft((COMPLEX*)B,fftsize,-1); /* Transform the second vector */

    /* Perform a complex multiplication on the transformed arrays,
       component wise: */
    A[0] *= B[0]; A[1] *= B[1];
    for (k=2;k<multsize;k+=2) {
        A[k] = (T=A[k])*B[k]-A[k+1]*B[k+1];
        A[k+1] = T*B[k+1] + A[k+1]*B[k];
    }

    realfft((COMPLEX*)A,fftsize,1); /* Transform the result back */

    /* A final loop to fix the carry: */
    for (C=0.0,k=0; k<multsize; k++) {
        C = floor( (T = A[k]*multsize+C+.5) / 65536. );
        A[k] = T - C*65536.;
    }

    /* And now, cast the result double array down to an ushort-array: */
    for (k=0; k<multsize; k++)
        ((ushort*)A)[k] = (ushort) A[k];
}

```

---

**Division.** All we need, now, is an integer division algorithm, most preferably based on the fast multiplication we already have.

**Definition 4.2 (Newton Iteration).** Let  $f : ]a, b[ \rightarrow \mathbb{R}$  be a continuously differentiable mapping and  $x_0 \in ]a, b[$ . If it exists, the Newton Iteration of  $f$  with regards to  $x_0$  is recursively defined as

$$x_{k+1} := x_k - \frac{f(x_k)}{f'(x_k)}$$

where  $f'(x) = \frac{d}{dx} f(x)$  is the first order derivate of  $f$ .

We hope that the Newton Iteration of a function  $f$  converges to its zero  $x^* \in ]a, b[$ . Motivation for this is given by the linear approximation property of the derivate:

$$f(x + \alpha) = f(x) + f'(x) \cdot \alpha + \varphi(\alpha) \text{ with } \varphi = o(|\alpha|), \text{ meaning } \lim_{\alpha \rightarrow 0} \frac{\varphi(\alpha)}{|\alpha|} = 0.$$

In our case, it means  $0 = f(x^*) = f(x_k + \Delta_k) \approx f(x_k) + f'(x_k) \Delta_k \Rightarrow -\Delta_k = f'(x_k)^{-1} \cdot f(x_k)$ . Therefore, if we pick a start value  $x_0$  with a small enough  $\Delta_0$  from  $x^*$ , we can hope to get closer to  $x^*$  with each Iteration.

Now, consider the mapping  $f_b(x) := D^n (xb)^{-1} - 1$ . Obviously,  $f_b(D^n b^{-1}) = 0$ . Instead of actually performing an integer division  $a = q \cdot b + r$ , we will approximate the reciprocal value  $\beta := D^n b^{-1}$  and calculate  $a \cdot \beta \cdot D^{-n} = a \cdot 1/b \cdot D^n \cdot D^{-n} = a/b$ . Another Subtraction is required to extract the remainder. We derive  $f'_b(x) = -D^n b^{-1} \cdot x^{-2}$  and get the Iteration

$$x_{k+1} := x_k + f'_b(x_k)^{-1} f_b(x_k) = x_k + D^{-n} b x_k^2 (D^n x_k^{-1} b^{-1} - 1) = 2x_k - D^{-n} b x_k^2 = 2 \left( x_k - \frac{1}{2} b x_k^2 D^{-n} \right)$$

Where the last term is chosen because  $\frac{1}{2} b x_k^2 D^{-n}$  will fit into a single integer Variable.

By choosing a function  $f_b$  with  $f_b(D^n b^{-1}) = 0$ , we have reduced integer division to integer multiplication, as the Newton Iteration of  $f_b$  only involves multiplications. This happens at the cost of an additional  $O(\log \log n)$  factor for the runtime of the division algorithm.

---

The Algorithm calculates the most significant bits of the reciprocal value  $D^n b^{-1}$  and uses it to perform the division  $a = q \cdot b + r$  of two integer numbers  $a$  and  $b$ .

---

```

/* This example code is for MS VC++, but bLog2 should generally return the position
   of the highest bit of a which is set. */
ulong bLog2(ulong a) {
    __asm {
        mov eax,a
        bsr eax,eax
    } }

/* iInv calculates the reciprocal value x := 1/b << SIZE and requires an additional
   temporary array t. Here, n is the size of b and m the size of the array x. */
void iInv(ulong *A, ulong mulsize, ulong *x, ulong *t, ulong *b, ulong m, ulong n)
{
    double appr;
    ulong k,j;

    for (k=n-1;k<n;k--) if(b[k]) break; /* find the first nonzero word */
    if (!(k+1)) __asm int 0; /* zero division exception */

    memset (x,0,m*sizeof(ulong));

    appr = 1./b[k]; /* First approximation by using the inverse */
    for (k=m-k; appr; k--) { /* of the highest word of b. Loop until we */
        x[k] = (ulong) appr; /* obtain all places. */
        appr = (appr-x[k])*4294967296.;
    }

    /* After 32=0x20 rounds, the iteration will converge. This could be implemented in
       a smarter way - for the purpose of demonstration, we refrained from doing so. */
    for (j=0x21;j;j--) {
        iMul(A,x,b,m,n);
        memcpy(t,A,m*sizeof(ulong)); /* t := x*b */

        if (A[m]) {
            iRShift(t,m);
            t[m-1] |= (1<<31);
        } else iRShift(t,m); /* t := (x*b)/2 */
        iMul(A,mulsize,x,t,m,m);
        memcpy(t,A+m, m*sizeof(ulong)); /* t := (x*x*b)/2 >> m */
        iSub(x,t,m); /* x := x - t */
        iLShift(x,m); /* x := 2 * (x - t) */
    }
}

/* The actual division is done using iInv: */
void iDiv(ulong *A, ulong mulsize, ulong *a, ulong *b, ulong size, ulong size_b, ulong *q)
{
    ulong *x,*t;
    x = (ulong*) (A+(2*mulsize));
    t = x + size;

    if (!q) q = a;
    iInv(A,mulsize,x,t,b,size,size_b); /* x := 1/b */
    iMul(A,mulsize,a,x,size,size); /* t := a * x = a/b */
    memcpy(t,A+size,size*sizeof(ulong));
    iMul(A,mulsize,t,b,size,size_b); /* x := t * b = a/b*b = a */
    memcpy(x,A,size*sizeof(ulong)); /* Check for OffByOne */
    if (iCmp(x,a,size) > 0)
/* Call method to subtract one word (3rd argument) from a big integer (1st and 2nd argument).
   This method is easily implemented. */
        iShortSub(t,size,1);
    memcpy(q, t, size*sizeof(ulong)); /* copy final result to q */
}

```

---

For the sake of completeness, a detailed proof for the correctness and the runtime of this division algorithm has been added below. Some of the concepts used are beyond the original scope of this article, so if you are not interested, you can skip ahead to §5.

**Theorem 4.4.** Let  $f : ]a, b[ \rightarrow \mathbb{R}$  be a continuously differentiable mapping and  $a \in ]a, b[$  with  $f(a) = 0$ . Let also  $f'(x) \neq 0$  for all  $x \in ]a, b[$ . For a  $\omega > 0$ , the inequality

$$|f'(x + sh) - f'(x)| \leq \omega sh \cdot |f'(x)|$$

may hold for all  $x \in ]a, b[$  and  $s \in [0, 1]$ , where  $h \in \mathbb{R}$  with  $x + h \in ]a, b[$ . Write  $\delta := 2/\omega$ . Then, for any  $x_0 \in B_\delta(a) := ]a - \delta, a + \delta[$ , the Newton Iteration of  $f$  with regards to  $x_0$  converges to  $a$ . Furthermore, the convergence rate is quadratic:

$$|x_{k+1} - a| \leq \frac{\omega}{2} \cdot |x_k - a|^2$$

**Proof.** Let  $\varphi(x) := x - f'(x)^{-1} f(x)$  and  $x \in B_\delta(a)$ . We can write

$$\begin{aligned} f(x) + \int_0^1 f'(x + s(a-x))(a-x) ds &= f(x) + [f(x + s(a-x))]_0^1 \\ &= f(x) + f(a) - f(x) = f(a) = 0, \text{ and hence} \end{aligned}$$

$$\begin{aligned} |a - \varphi(x)| &= \left| a - x + f'(x)^{-1} f(x) \right| = \left| f'(x)^{-1} (f(x) + f'(x)(a-x)) \right| \\ &= \left| f'(x)^{-1} \cdot \left( f'(x)(a-x) - \int_0^1 f(x + h(a-x))(a-x) dh \right) \right| \\ &= \left| - \int_0^1 f'(x)^{-1} (f(x + h(a-x)) - f'(x))(a-x) dh \right| \\ &\leq \int_0^1 |f'(x)|^{-1} \cdot |f(x + h(a-x)) - f'(x)| \cdot |a-x| dh \\ &\leq \int_0^1 |f'(x)|^{-1} \cdot |f'(x)| \cdot \omega h \cdot |a-x|^2 dh = \int_0^1 h \cdot \omega |a-x|^2 dh = \frac{\omega}{2} \cdot |a-x|^2 \end{aligned}$$

Finally, write  $|a - x_{k+1}| = |a - \varphi(x_k)| \leq \frac{\omega}{2} |a - x_k|^2 < |a - x_k|$

□

**Corollary 4.5.** Let  $D, b \in \mathbb{R}$ . The Newton Iteration of  $f : B_r(b) \rightarrow \mathbb{R}$ ,  $x \mapsto D(xb)^{-1} - 1$  with  $r < \frac{1}{4}b$  converges to  $Db^{-1}$  at a quadratic rate for any start value  $x_0 \in B_r(b)$ .

**Proof.** Hence, we need to prove that the Conditions of 4.4 are met for  $f$ . We know that  $f'(x) := -Db^{-1}x^{-2}$  and  $f''(x) := 2Db^{-1}x^{-3}$ . Hence:

$$\exists \xi \in B_r(b) : |f'(x+sh) - f'(x)| = |f''(\xi)| \cdot sh = |2Db^{-1}\xi^{-3}| sh \leq 2Db^{-1}(b-r)^{-3} sh$$

Also,  $|f'(x)^{-1}| = D^{-1}bx^2 \leq D^{-1}b(b+r)^2$  and together

$$|f'(x)^{-1}| \cdot |f'(x+sh) - f'(x)| \leq 2Db^{-1}(b-r)^3 sh \cdot D^{-1}b(b+r)^2 = 2(b-r)^3(b+r)^2 \cdot sh$$

This means  $\omega = 2(b-r)^3(b+r)^2$  and thus we have to show that  $r < \delta := (b-r)^3(b+r)^{-2}$ . Due to  $b > 4r$ , we get:

$$\begin{aligned} (b+r)^2 r &< (b-r)^3 \\ \Leftrightarrow b^2 r + 2br^2 + r^3 &< b^3 - 3b^2 r + 3br^2 - r^3 \\ \Leftrightarrow 4b^2 r + 2r^3 - br^2 &< b^3 \\ \Leftrightarrow r(4b^2 + r(2r-b)) &< r(4b^2) < b^3 \end{aligned}$$

□

**Corollary 4.6.** The Algorithm 4.3 works correctly and terminates after  $O(n \cdot \log n \cdot \log \log n)$  steps.

**Proof.** Since we use a 53 bit approximation (64 bit double, minus 11 bit exponent and one sign bit) to  $D^n b^{-1}$ , we can assume  $r_0 < \varepsilon b$  with  $\varepsilon = 2^{-50} < 2^{-2} = \frac{1}{4}$ . Hence, we have quadratic convergence by

4.5. More precisely, we have  $r_k < \varepsilon^{2^k} b$ :

$$r_{k+1} := |x_{k+1} - b^{-1}| < (b-r_k)^{-3} (b+r_k)^2 r_k^2 < (b-r_k)^{-1} r_k^2 = \frac{1}{b/r_k^2 - r_k^{-1}} < \frac{1}{b/r_k^2} < (\varepsilon^{2^k} b)^2 b^{-1} = \varepsilon^{2^{k+1}} b$$

And thus,  $O(\log \log n)$  Multiplications are required before the sequence converges. (Consider

$$\varepsilon^{2^{\log \log n}} = \varepsilon^{\log n} = 2^{-50 \cdot \log(n)} = n^{-50} \approx 0)$$

□

## §5 – Optimization

*“Compatible code is slow.”*

We have seen that almost all computational effort lies in the efficient implementation of the FFT algorithm. Hence, a hand-tuned Assembler version of **2.10** has been devised. Benchmarks showed a factor 2 speedup for multiplications (and therefore, also divisions).

The main speedup is achieved by minimizing memory access times. In the original C version of the Algorithm, many complex variables had to be defined to store information that can as well remain on the FPU stack during the entire computation. Without further explanation, the thoroughly commented code is presented:

---

```
COMPLEX* fft(COMPLEX* f, unsigned long n, signed int s) {
  __asm {
    mov ebx, n          // ebx contains length of function
    mov edx, ebx       // so does edx
    finit              // initialize Floating Point Unit
    fld s               // load forward/backwards indicator
    fld st(0)          // push another copy of s on the stack
    fabs               // |s|, s on stack
    fdivp st(1),st(0)  // set ST(0) to s/|s| = sgn(s)
    fist s             // store the result in s, do not pop
    fld1               // load 1 onto stack
    fldpi              // load pi onto stack
    fscale             // scaling pi by 2
    fstp st(1)         // now, ST(0) = 2*pi
    fmulp st(1),st(0)  // ST(0) = 2*pi*sgn(s)
    fdiv n              // ST(0) = 2*pi*sgn(s)/n
    fsincos            // cos(ST(0)),sin(ST(0)) on stack.

    // We now have w on stack:
    // ST(0) = Re(w)
    // ST(1) = Im(w)

    mov esi, f         // esi = f
    cmp s, 1           // check,
    je START           // do not scale if s != 1
    fld n              // push n onto FPU stack
    mov ecx, ebx       // n values to be scaled
    shl ecx, 1         // each value consists of 2 doubles
SCALE: fld qword ptr [esi+8*ecx-8] // push double on the stack
    fdiv ST(0),ST(1)   // scale by n
    fstp qword ptr [esi+8*ecx-8] // store scaled double
    loop SCALE        // loop until ecx=0
    fcomp             // pop n off the FPU stack again.

    // In terms of the C version of the algorithm, we will use
    // the ebx register as N, the ecx register as j, the eax
    // register as k and the edx register as M.

START: cmp ebx,1      // Only continue to transform as
    jng END           // long as N > 1.
    shr edx,1         // Set M := N >> 1;
    xor ecx,ecx       // start new loop with j=0
INNER1: fldz          // push 0 onto FPU stack
    fld1             // push 1 onto FPU stack

    // we now have w and the initial z on the FPU stack:
    // ST(0) = Re(z)
    // ST(1) = Im(z)
    // ST(2) = Re(w)
    // ST(3) = Im(w)

    xor eax,eax       // start new loop with k=0
INNER2: add eax,ecx   // for indexing, we need k+j
    mov edi,eax       // and
    add edi,edx       // k+j+M
    shl eax,4         // we are indexing 16-byte COMPLEX
    shl edi,4         // values, scale indices

    fld qword ptr [esi+edi+8] // ST(2) = f[j+k+M].imag
    fld qword ptr [esi+eax+8] // ST(1) = f[j+k].imag
    fld ST(1)          // ST(0) = f[j+k+M].imag
    fadd ST(0),ST(1)   // ST(0) = f[j+k].imag + f[j+k+M].imag
    fstp qword ptr [esi+eax+8] // f[j+k] = f[j+k].imag + f[j+k+M].imag
    fsubrp ST(1),ST(0) // ST(0) = f[j+k].imag - f[j+k+M].imag

    // Now, the same procedure is performed for the real parts of
    // the two complex variables:

    fld qword ptr [esi+edi]
    fld qword ptr [esi+eax]
    fld ST(1)
    fadd ST(0),ST(1)
```

```

    fstp qword ptr [esi+eax]
    fsubrp ST(1),ST(0)

// the FPU stack now looks like this:
// ST(0) = real part of f[j+k]-f[j+k+M] := a
// ST(1) = imag part of a
// ST(2) = real of z
// ST(3) = imag of z
// ST(4) = real of w
// ST(5) = imag of w
// we want to calculate a*z and store it in f[j+k+M].

    fld ST(0) // push Re(a)
    fmul ST(0),ST(3) // ST(0) <- Re(z)*Re(a)
    fld ST(2) // push Im(a)
    fmul ST(0),ST(5) // ST(0) <- Im(a)*Im(z)
    fsubp ST(1),ST(0) // TOS <- ST(1)-ST(0)
    fstp qword ptr [esi+edi] // pop to f[j+k+M].real
// -----
    fmul ST(0),ST(3) // ST(0) <- Re(a)*Im(z)
    fxch ST(1) // ST(0) <-> ST(1)
    fmul ST(0),ST(2) // ST(1) <- Im(a)*Re(z)
    faddp ST(1),ST(0) // TOS <- ST(0)+ST(1)
    fstp qword ptr [esi+edi+8] // pop to f[j+k+M].imag

// We now have only z and w on the FPU stack and for the
// next iteration, we want to set z := z*w.

    fld ST(0) // push Re(z)
    fmul ST(0),ST(4) // ST(0) <- Re(z)*Im(w)
    fld ST(2) // push Im(z)
    fmul ST(0),ST(4) // ST(0) <- Im(z)*Re(w)
    faddp ST(1),ST(0) // TOS <- Im(z*w)
    fxch ST(2) // set new Im(z)
    fmul ST(0),ST(4) // ST(0) <- Im(z)*Im(w)
    fxch ST(1) // ST(0) <-> ST(1)
    fmul ST(0),ST(3) // ST(0) <- Re(z)*Re(w)
    fsubp ST(1),ST(0) // ST(0) <- -Re(z*w)
    fchs // ST(0) <- Re(z*w)

    shr eax,4 // unscale indices again

    sub eax,ecx // eax := (k+j)-j = k
    inc eax // increase by one
    cmp eax,edx // check whether k<M
    jl INNER2 // if yes, loop again
// the inner loop ends here:
    fcompp // pop z off the stack
    add ecx,ebx // j += N
    cmp ecx,n // check whether j < n
    jl INNER1 // if yes, loop again

// the outer loop ends here. We now want to calculate
// w := w*w and simply jump back to START where the
// N>1 condition is checked.

    fld ST(1) // Push Im(w) onto stack
    fmul ST(0),ST(1) // ST(0) := Re(w)*Im(w)
    fld ST(0) // Push ST(0) again
    faddp ST(1),ST(0) // ST(0) = 2*Re(w)*Im(w)
    fxch ST(2) // exchange it with Im(w)
    fmul ST(0),ST(0) // ST(0) := Im(w)*Im(w)
    fxch ST(1) // exchange it with Re(w)
    fmul ST(0),ST(0) // ST(0) := Re(w)*Re(w)
    fsubp ST(1),ST(0) // ST(0) = Im(w)*Im(w)-Re(w)*Re(w)
    fchs // invert sign, done

    mov ebx,edx // N := M
    jmp START // and again

// The transformation has now been performed completely. All
// that is left to be done is doing the bit reversal.

END: xor edx,edx // edx will be first index
    mov eax,edx // eax the second
    mov ebx,n // ebx stores n
BREU: cmp edx,eax // only swap values if
    jle NOSWAP // edx > eax
    shl edx,4
    shl eax,4
    fld qword ptr [esi+edx]
    fld qword ptr [esi+edx+8]
    fld qword ptr [esi+eax]
    fld qword ptr [esi+eax+8]
    fstp qword ptr [esi+edx+8]
    fstp qword ptr [esi+edx]
    fstp qword ptr [esi+eax+8]
    fstp qword ptr [esi+eax]
    shr edx,4
    shr eax,4
NOSWAP: mov ecx,ebx // calculate bitreversal:
CLOOP: shr ecx,1 // ecx is of the form 1<<k.
    xor edx,ecx // XOR ecx to edx until the result of
    cmp edx,ecx // that XOR is not 0, hence ecx<=edx
    jl CLOOP // otherwise we have a 'carry'.
    inc eax // increase normal index
    cmp eax,ebx // and loop until we're done
    jl BREU
    mov eax,esi // return f;
}}

```

## §6 – Conclusion

We have successfully implemented  $O(n \cdot \log n)$  - Multiplication and  $O(n \cdot \log n \cdot \log \log n)$  - Division for big integers. Addition and Subtraction are easily enough implemented in linear time. In practice, it has proven more convenient to use a naïve  $O(n^2)$  division method rather than the (asymptotically faster) one described above. However, the speed improvement of an FFT multiplication over a divide and conquer approach were already noticeable for integers of 960 bits size.

## §7 – References

- Deuffhard, P., Numerical Analysis, 3<sup>rd</sup> Edition. Walter de Gruyter, 2002.  
**ISBN 3110140314**
- William H. Press [et al.], Numerical recipes in C : the art of scientific computing, 2<sup>nd</sup> Edition. Cambridge University Press, 1997.  
**ISBN 0521431085**
- IA-32 Intel® Architecture Software Developer's Manual  
<http://www.intel.com/products/processor/manuals/index.htm> (10/15/07)
- First prototype Implementation of the .aware big integer library (08/29/08)  
<https://www.awarenetwork.org/home/.rants/07-09-2008.10.36/bint.rar>