FASTER INTEGER MULTIPLICATION

MARTIN FÜRER*

Abstract. For more than 35 years, the fastest known method for integer multiplication has been the Schönhage-Strassen algorithm running in time $O(n \log n \log\log n)$. Under certain restrictive conditions, there is a corresponding $\Omega(n \log n)$ lower bound. All this time, the prevailing conjecture has been that the complexity of an optimal integer multiplication algorithm is $\Theta(n \log n)$. We present a major step towards closing the gap from above by presenting an algorithm running in time $n \log n\, 2^{O(\log^* n)}$. The running time bound holds for multitape Turing machines. The same bound is valid for the size of boolean circuits.

1. Introduction. All known methods for integer multiplication (except the trivial school method) are based on some version of the Chinese Remainder Theorem. Schönhage [Sch66] computes modulo numbers of the form $2^k + 1$. Most methods can be interpreted as schemes for the evaluation of polynomials, multiplication of their values, followed by interpolation. The classical method of Karatsuba [KO62] can be viewed as selecting the values of homogeneous linear forms at $(0, 1)$, $(1, 0)$, and $(1, 1)$ to achieve time $T(n) = O(n^{\lg 3})$[1]. Toom [Too63] evaluates at small consecutive integer values to improve the circuit complexity to $T(n) = O(n^{1+\epsilon})$ for every $\epsilon > 0$. Cook [Coo66] presents a corresponding Turing machine implementation. Finally Schönhage and Strassen [SS71] use the usual fast Fourier transform (FFT) (i.e., evaluation and interpolation at 2^mth roots of unity) to compute integer products in time $O(n \log n \log\log n)$. They conjecture the optimal upper bound (for a yet unknown algorithm) to be $O(n \log n)$, but their result has remained unchallenged.

Schönhage and Strassen [SS71] really propose two distinct methods. The first one uses numerical approximation to complex arithmetic, and reduces multiplication of length n to that of length $O(\log n)$. The complexity of this method is slightly higher. Even as a one level recursive approach, with the next level of multiplications done by a trivial algorithm, it is already very fast. The second method employs arithmetic in rings of integers modulo numbers of the form $F_m = 2^{2^m} + 1$ (Fermat numbers), and reduces the length of the factors from n to $O(\sqrt{n})$. This second method is used recursively with $O(\log\log n)$ nested calls. In the ring $\mathbb{Z}_{F_m}$ of integers modulo F_m, the integer 2 is a particularly convenient root of unity for the FFT computation, because all multiplications with this root of unity are just modified cyclic shifts.

On the other hand, the first method has the advantage of a significant length reduction from n to $O(\log n)$ in one level of recursive calls. If this method is applied recursively with $\lg^* n - O(1)$[2] nested calls (i.e., until the factors are of constant length), then the running time is of order $n \log n \log\log n \cdots 2^{O(\lg^* n)}$, because at level 0, time $O(n \log n)$ is spent, and during the kth of the $\lg^* n - O(1)$ recursion levels, the amount of time increases by a factor of $O(\log\log \ldots \log n)$ (with the log iterated $k+1$ times) compared to the amount of time spent at the previous level k. The time spent at level k refers to the time spent during k-fold nested recursive calls, excluding the time spent during the deeper nested recursive calls.

Note that for their second method, Schönhage and Strassen have succeeded with

*Research supported in part by the Penn State Grove Award and the National Science Foundation under Grant CCF-0728921, Department of Computer Science and Engineering, Pennsylvania State University, Visiting: Laboratoire d'Algorithmique, EPFL, Lausanne, furer@cse.psu.edu,

[1] lg refers to the logarithm to the base 2

[2] $\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$ with $\lg^{(0)} n = n$ and $\lg^{(i+1)} n = \lg \lg^{(i)} n$

the difficult task of keeping the time at each level basically fixed. Even just a constant factor increase per level would have resulted in a factor of $2^{O(\log\log n)} = (\log n)^{O(1)}$ instead of $O(\log\log n)$.

Our novel version of the FFT allows us to combine the main advantages of both methods of Schönhage and Strassen. The reduction is from length n to length $O(\log^2 n)$, and still most multiplications with roots of unity are just modified cyclic shifts. Unfortunately, we are not able to avoid the geometric increase over the $\lg^* n$ levels.

Relative to the conjectured optimal time of $\Theta(n \log n)$, the first Schönhage and Strassen method had an overhead factor of $\log\log n \, \log\log\log n \ldots 2^{O(\lg^* n)}$, representing a doubly exponential improvement compared to previous methods. Their second method with an overhead of $O(\log\log n)$ constitutes another poly-logarithmic decrease. Our new method reduces the overhead to $2^{O(\lg^* n)}$, and thus represents a more than multiple exponential improvement of the overhead factor. Naturally, we have to admit that for practical values of n, say in the millions or billions, it is not immediately obvious how to benefit from this last improvement.

We use a somewhat unusual divide-and-conquer approach to the N-point FFT, where N is a power of 2. Throughout this paper, integers denoted by capital letters are usually powers of 2. It is well known and obvious that the JK-point FFT graph (butterfly graph, Figure 1.1) can be composed of two stages, one containing K copies

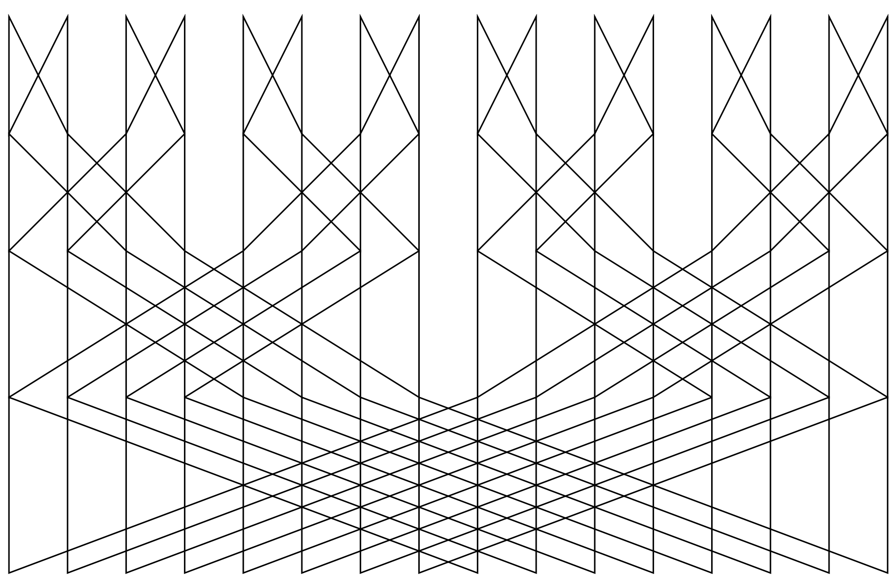

Fig. 1.1. *The butterfly graph of a 16 point FFT*

of a J-point FFT graph, and the other containing J copies of a K-point FFT graph. Clearly $N = JK$ could be factored differently into $N = J'K'$ and the same N-point FFT graph could be viewed as being composed of J'-point and K'-point FFT graphs. The astonishing fact is that this is just true for the FFT graph and not for the FFT computation. Every way of (recursively) partitioning N produces another FFT algorithm. Multiplications with other powers of ω (twiddle factors) appear when another recursive decomposition is used. Here ω is the principal Nth root of unity used in the FFT.

It seems that this fact has not been widely noticed within the algorithms community. It has been successfully used in several papers to improve the constant factor in the complexity of the FFT over $\mathbb{C}$. The number of real multiplications is decreased by increasing the use of the roots of unity $\pm i$ and to a lesser extent $\pm(1 \pm i)/\sqrt{2}$.

There seems to be only one previous attempt to decrease the asymptotic complexity of any algorithm based on a different decomposition of a 2^m-point FFT. It was an earlier attempt to obtain a faster integer multiplication algorithm [Für89]. In that paper, the following result has been shown. If there is an integer $k > 0$ such that for every m, there is a prime number in the sequence $F_{m+1}, F_{m+2}, \ldots, F_{2^{m+k}}$ of Fermat numbers ($F_m = 2^{2^m} + 1$), then multiplication of binary integers of length n can be done in time $n \log n \, 2^{O(\lg^* n)}$. Hence, the Fermat primes could be extremely sparse and would still be sufficient for a faster integer multiplication algorithm. It turns out that the Fermat prime paper [Für89] provides some of the key ingredients of the current faster integer multiplication algorithm. Nevertheless, that paper by itself was not so exciting, because it is well known—based on probabilistic assumptions—that the number of Fermat primes is conjectured to be finite, and even that F_4 is the largest of them.

It has long become standard to view the FFT as an iterative process (see e.g., [SS71, AHU74]). Even though the description of the algorithm gets more complicated, it results in less computational overhead. A vector of coefficients at level 0 is transformed level by level, until the Fourier transformed vector at level $\lg N$ is reached. The operations at each level are additions, subtractions, and multiplications with powers of an Nth root of unity ω. They are done as if the N-point FFT were recursively decomposed into two $N/2$-point FFTs followed by $N/2$ two-point FFTs. The current author has seen Schönhage present the other natural recursive decomposition into $N/2$ two-point FFTs followed by two $N/2$-point FFTs. It results in another distribution of the powers of ω, even though, each power of ω appears as a coefficient in both of these iterative methods with the same frequency. But other decompositions produce completely different frequencies. The standard fast algorithm design principle, divide-and-conquer, calls for a balanced partition, but in this case it is not at all obvious that it will provide any benefit.

A balanced approach uses two stages. K J-point FFTs are followed by J K-point FFTs, where J and K are roughly $\sqrt{N}$. If $N = 2^m$, then one can choose $J = 2^{\lceil m/2 \rceil}$ and $K = 2^{\lfloor m/2 \rfloor}$. This can improve the FFT computation, because it turns out that "odd" powers of ω are then used very seldom. Most multiplications with powers of ω are then actually multiplications with powers of ω^K. This key observation alone is not sufficiently powerful to obtain a better asymptotic running time, because usually 1, -1, i, $-i$ and to a lesser extent $\pm(1 \pm i)/\sqrt{2}$ are the only powers of ω that are easier to handle. We will achieve the desired speed-up by working over a ring with many "easy" powers of ω. Hence, the new faster integer multiplication algorithm is based on two key ideas.

- An unconventional FFT algorithm is used with the property that most occurring roots of unity are of low order.
- The computation is done over a ring where multiplications with many low order roots of unity are very simple and can be implemented as a kind of cyclic shifts. At the same time, this ring also contains high order roots of unity.

It is immediately clear that integers modulo a Fermat prime F_m form such a ring. For

$N = F_m - 1 = 2^{2^m}$, the number 2 is a nice low order $(2 \lg N)$th root of unity, while 3 is an Nth root of unity. Instead of 3, it is computationally easy to find a number $\omega \in \mathbb{Z}_{F_m}$ such that $\omega^{N/(2 \lg N)} = 2$. These properties form the basis of the Fermat prime result [Für89] mentioned above. The additional main accomplishment of the current paper is to provide a ring having similar properties as the field $\mathbb{Z}_{F_m}$.

The question remains whether the optimal running time for integer multiplication is indeed of the form $n \log n \, 2^{O(\lg^* n)}$. Already, Schönhage and Strassen [SS71] have conjectured that the more elegant expression $O(n \log n)$ was optimal as we mentioned before. It would indeed be strange if such a natural operation as integer multiplication had such a complicated expression for its running time. But even for $O(n \log n)$ there is no unconditional corresponding lower bound. Still, long ago there have been some remarkable attempts. In the algebraic model, Morgenstern [Mor73] has shown that every N-point Fourier transform—done by just using linear combinations $ax + by$ with $|a| + |b| \leq c$ for inputs or previously computed values x and y—requires at least $(n \lg n)/(2(1 + \lg c))$ operations. More recently, Bürgisser and Lotz [BL04] have extended this result to multiplying complex polynomials.

Under different assumptions on the computation graph, Papadimitriou [Pap79] and Pan [Pan86] have shown conditional lower bounds of $\Omega(n \log n)$ for the FFT. Both are for the interesting case of n being a power of 2. Cook and Anderaa [CA69] have developed a method for proving non-linear lower bounds for on-line computations of integer products and related functions. Based on this method, Paterson, Fischer and Meyer [PFM74] have improved the lower bound for on-line integer multiplication to $\Omega(n \log n)$. Naturally, one would like to see unconditional lower bounds, as the on-line requirement is a very severe restriction. On-line means that starting with the least significant bit, the kth bit of the product is written before the $k + 1$st bits of the factors are read.

With the lack of tight lower bounds, it may be tempting to experiment with variations of our new algorithm with the goal of improving the running time or better understand the difficulties in trying to improve it. Indeed, after the publication of the conference version of our algorithm [Für07], a more discrete algorithm based on the same ideas has been obtained by De, Kurur, Saha, and Saptharishi [DKSS08]. Its running time is still $n \log n \, 2^{O(\lg^* n)}$, but it is based on p-adic numbers instead of complex numbers. This might be a useful ingredient in the quest to decrease the constant factor blow-up from one recursion level to the next. The ambitious goal is to obtain a blow-up factor of $1 + o(1)$.

In Section 2, we present the basics about roots of unity in rings, the Chinese Remainder Theorem for rings, and the discrete Fourier transform. In Section 3, we review the FFT in a way that shows which twiddle factors (powers of ω) are used for any given recursive decomposition of the algorithm. In Section 4, we present a ring with many nice roots of unity allowing our faster FFT computation. In Section 5, we describe the new method of using this FFT for integer multiplication. It would be helpful for the reader to be familiar with the Schönhage-Strassen integer multiplication algorithm, e.g., as described in the original paper, or in Aho, Hopcroft and Ullman [AHU74], but this is not a prerequisite. In Section 6, we study the precision requirements for the numerical approximations used in the Fourier transforms. Finally, in Section 7, we state the complexity results, followed by open problems in Section 8.

2. The Discrete Fourier Transform. Throughout this paper all rings are commutative rings with 1. Primitve roots of unity are well known objects in algebra.

An element ω in a ring $\mathcal{R}$ is a *primitive* Nth root of unity if it has the following properties.

1. $\omega^N = 1$
2. $\omega^k \neq 1$ for $1 \leq k < N$

Closely related is the notion of a principal root of unity. An element ω in a ring $\mathcal{R}$ is a *principal* Nth root of unity if it has the following properties.

1. $\omega^N = 1$
2. $\sum_{j=0}^{N-1} \omega^{jk} = 0$ for $1 \leq k < N$

The two notions coincide in fields of characteristic 0, but for the discrete Fourier transform over rings, we need the stronger principal roots of unity. A principal Nth root of unity is also a primitive Nth root of unity unless the characteristic of the ring $\mathcal{R}$ is a divisor of N. For integral domains (commutative rings with 1 and without zero divisors), every primitive root of unity is also a principal root of unity.

The definition of principal roots of unity immediately implies the following result. If ω is a prinicpal Nth root of unity, then ω^J is a principal $(N/\gcd(N,J))$th root of unity.

Example 1. *In $\mathbb{C} \times \mathbb{C}$ the element $(1, i)$ is a primitive 4th root of unity, but not a principle 4th root of unity.*

Lemma 2.1. *If N is a power of 2, and $\omega^{N/2} = -1$ in an arbitrary ring, then ω is a principal Nth root of unity.*

Proof. Property 1 of principal roots of unity is trivial. Property 2 is shown as follows.

Let $0 < k = (2u+1)2^v < N$, trivially implying that $k/2^v = 2u+1$ is an odd integer and $2^{v+1} \leq N$.

$$\begin{aligned}
\sum_{j=0}^{N-1} \omega^{jk} &= \sum_{i=0}^{2^{v+1}-1} \sum_{j=0}^{N/2^{v+1}-1} \omega^{(iN/2^{v+1}+j)k} \\
&= \sum_{j=0}^{N/2^{v+1}-1} \omega^{jk} \underbrace{\sum_{i=0}^{2^{v+1}-1} \underbrace{\omega^{(iN/2^{v+1})(2u+1)2^v}}_{(-1)^i}}_{0} \\
&= 0
\end{aligned}$$

□

Definition 2.2. *The N-point discrete Fourier transform (DFT) over a ring $\mathcal{R}$ is the linear function, mapping the vector $\boldsymbol{a} = (a_0, \ldots, a_{N-1})^\mathsf{T}$ to $\boldsymbol{b} = (b_0, \ldots, b_{N-1})^\mathsf{T}$ by*

$$\boldsymbol{b} = \Omega \boldsymbol{a}, \text{ where } \Omega = (\omega^{jk})_{0 \leq j,\, k \leq N-1}$$

for a given principal Nth root of unity ω.

In other words,

$$b_j = \sum_{k=0}^{N-1} \omega^{jk} a_k \tag{2.1}$$

Hence, the discrete Fourier transform maps the vector of coefficients $(a_0, \ldots, a_{N-1})^\mathsf{T}$ of a polynomial

$$p(x) = \sum_{k=0}^{N-1} a_k x^k$$

of degree $N-1$ to the vector

$$(b_0, \ldots, b_{N-1})^\mathsf{T} = (p(1), p(\omega), p(\omega^2), \ldots, p(\omega^{N-1}))^\mathsf{T}$$

of values at the N powers of ω.

Every Nth root of unity ω has an inverse $\omega^{-1} = \omega^{N-1}$. If N also has an inverse in the ring $\mathcal{R}$, and ω is a principal root of unity, then ω^{-1} is also a principal root of unity, and the matrix Ω has an inverse Ω^{-1}. The former is shown as follows

$$\sum_{j=0}^{N-1} \omega^{-jk} = \sum_{j=0}^{N-1} \omega^{Nk}\omega^{-jk} = \sum_{j=0}^{N-1} \omega^{(N-j)k} = \sum_{j=1}^{N} \omega^{jk} = 0$$

for $0 < k < N$. The inverse of the discrete Fourier transform is $1/N$ times the discrete Fourier transform with the principal Nth root of unity ω^{-1}, because

$$\sum_{j=0}^{N-1} \omega^{-ij}\omega^{jk} = \sum_{j=0}^{N-1} \omega^{(k-i)j} = \begin{cases} N & \text{if } i = k \\ 0 & \text{if } i \neq k \end{cases}$$

If the Fourier transform operates over the field $\mathbb{C}$, then $\omega^{-1} = \bar{\omega}$, the complex conjugate of ω. Therefore, the discrete Fourier transform scaled by $1/\sqrt{N}$ is a unitary transformation, and $\frac{1}{\sqrt{N}}\Omega$ is a unitary matrix, i.e., $\bar{\Omega}^\mathsf{T}\Omega = N\mathbf{I}$, where $\mathbf{I}$ is the unit matrix.

Closely related to the N-point DFT, we also consider what we call the N-point half discrete Fourier transform (Half-DFT) in the case of N being a power of 2. Here the evaluations are done at the N odd powers of ζ, where ζ is a principal $2N$th root of unity. The Half-DFT could be computed by extending $\boldsymbol{a}$ with $a_N = \cdots = a_{2N-1} = 0$ and doing a $2N$-point DFT, but actually only about half the work is needed.

Definition 2.3. *The N-point half discrete Fourier transform (Half-DFT) is the linear function, mapping the vector $\boldsymbol{a} = (a_0, \ldots, a_{N-1})^\mathsf{T}$ to $\boldsymbol{b} = (b_0, \ldots, b_{N-1})^\mathsf{T}$ by*

$$\boldsymbol{b} = Z\boldsymbol{a}, \text{ where } Z = (\zeta^{j(2k+1)})_{0 \le j,\, k \le N-1}$$

for a given principal $2N$th root of unity ζ.

Hence, the Half-DFT maps the vector of coefficients $(a_0, \ldots, a_{N-1})^\mathsf{T}$ of a polynomial

$$p(x) = \sum_{j=0}^{N-1} a_j x^j$$

of degree $N-1$ to the vector

$$\boldsymbol{b} = (p(\zeta), p(\zeta^3), p(\zeta^5), \ldots, p(\zeta^{2N-1}))^\mathsf{T}$$

of values at the N odd powers of ζ. Thus

$$b_j = \sum_{k=0}^{N-1} \zeta^{(2j+1)k} a_k = \sum_{k=0}^{N-1} \omega^{jk}\zeta^k a_k \qquad (0 \le j < N)$$

for $\omega = \zeta^2$.

As usual, let $\Omega = (\omega^{jk})_{0 \le j,\, k \le N-1}$. Define $\mathrm{diag}(\boldsymbol{z})$, for $\boldsymbol{z} = (\zeta^0, \zeta^1, \ldots, \zeta^{N-1})^\mathsf{T}$ as the diagonal matrix with $\boldsymbol{z}$ in its diagonal. Then we have $Z = \Omega\ \mathrm{diag}(z)$ or

$$Z = \begin{pmatrix} \zeta^{1\cdot 0} & \zeta^{1\cdot 1} & \cdots & \zeta^{1(N-1)} \\ \zeta^{3\cdot 0} & \zeta^{3\cdot 1} & \cdots & \zeta^{3(N-1)} \\ \vdots & \vdots & & \vdots \\ \zeta^{(2N-1)0} & \zeta^{(2N-1)1} & \cdots & \zeta^{(2N-1)(N-1)} \end{pmatrix}$$

$$= \begin{pmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \cdots & \omega^{0(N-1)} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \cdots & \omega^{1(N-1)} \\ \vdots & \vdots & & \vdots \\ \omega^{(N-1)0} & \omega^{(N-1)1} & \cdots & \omega^{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} \zeta^0 & 0 & \cdots & 0 \\ 0 & \zeta^1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \zeta^{N-1} \end{pmatrix}$$

Thus for N a power of 2, and ζ a principal $2N$th root of unity in a ring, an N-point half discrete Fourier transform (Half-DFT) is a scaling operation followed by a standard N-point DFT with $\omega = \zeta^2$. The Half-DFT is invertible if and only if the corresponding DFT is invertible, which is the case if and only if 2 has an inverse in the ring.

Over the field $\mathbb{C}$, the DFT as well as the Half-DFT are scaled unitary transformations, as the matrices $\frac{1}{\sqrt{N}}\Omega$ and $\mathrm{diag}(\boldsymbol{z})$ are both unitary.

The DFT and the Half-DFT perform a ring isomorphism as determined by the Chinese Remainder Theorem, which we copy from Bürgisser et al. [BCS97, p. 75].

Theorem 2.4. *(Chinese Remainder Theorem) Let $\mathcal{R}$ be a commutative ring, $I_1, \ldots, I_N$ ideals in $\mathcal{R}$ which are pairwise coprime, i.e., $I_i + I_j = \mathcal{R}$ for all $i \neq j$. Then the ring homomorphism*

$$\mathcal{R} \ni z \mapsto (z + I_1, \ldots, z + I_N) \in \prod_j \mathcal{R}/I_j$$

is surjective with kernel $I = \bigcap_{j=1}^N I_j$. This induces a ring isomorphism

$$\mathcal{R}/I \to \mathcal{R}/I_1 \times \cdots \times \mathcal{R}/I_N$$

Consider a polynomial ring $\mathcal{R} = \mathcal{R}'[x]$ over a ring $\mathcal{R}'$. We want to apply the Chinese Remainder Theorem to two cases. In the first case, the ideal I_j is the ideal $(x - \omega^j)$ generated by $x - \omega^j$ for a principle Nth root of unity ω. In this case, the induced ring isomorphism is the DFT. In the second case, the ideal I_j is $(x - \zeta^{2j+1})$ for a principle $2N$th root of unity ζ. In this case, the induced ring isomorphism is the half-DFT. In the former case, we want to prove the intersection of ideals $I = \bigcap_{j=1}^N I_j$ to be $(x^N - 1)$ and in the latter case, we want to show $I = (x^N + 1)$. These results are easily obtained if $\mathcal{R}$ is the ring of polynomials $F[x]$ over some field F. We will need much more effort to prove these results for some cases where $\mathcal{R}$ is just a ring.

In order to apply the Chinese Remainder Theorem, we have to show that the ideals $(x - \zeta^i)$ and $(x - \zeta^j)$ are coprime for $i \neq j$, or equivalently that $\zeta^i - \zeta^j$ is a unit (i.e., an invertible element) in $\mathcal{R}$. For this purpose, we first show an auxiliary result.

Lemma 2.5. *Let N be a unit, and let ζ be a principal $2N$th root of unity in a ring. Then $\zeta^N = -1$.*

Proof.

$$0 = \frac{1}{N}\sum_{j=0}^{2N-1}\zeta^{jN} = \frac{1}{N}\sum_{j=0}^{2N-1}\zeta^{2(j/2-\lfloor j/2\rfloor)N} = 1+\zeta^N$$

Here, the first term (1) is obtained as the sum over all even values of j, while the second term (ζ^N) is obtained as the sum over all odd values of j. □

LEMMA 2.6. *For N a power of 2, let ω be a principal Nth root of unity in a ring $\mathcal{R}$. Let 2 be a unit. Then $1-\omega^k$ is a unit for $1 \le k < N$, and the ideals $(x-\omega^i)$ and $(x-\omega^j)$ are coprime for $i \not\equiv j \pmod N$.*

Proof. Let $k = (2u+1)2^v$. Then

$$(1-\omega^k)\sum_{j=0}^{N2^{-v-1}-1}\omega^{jk} = 1-\omega^{N2^{-v-1}k} = 1-\omega^{(N/2)(2u+1)} = 1-(-1)^{2u+1} = 2$$

Let $0 \le i < j < N$. Then $\omega^i - \omega^j = \omega^i(1-\omega^{j-i})$ is in the ideal $(x-\omega^i, x-\omega^j)$. As both, ω^i and $1-\omega^{j-i}$ are units, so is $\omega^i-\omega^j$, implying $(x-\omega^i, x-\omega^j) = \mathcal{R}[x]$, i.e., $(x-\omega^i)$ and $(x-\omega^j)$ are coprime. □

3. The Fast Fourier Transform. Now let $N = JK$. Then $\omega^{JK} = 1$. We want to represent the N-point DFT as a set of K parallel J-point DFTs (inner DFTs), followed by scalar multiplications and a set of J parallel K-point DFTs (outer DFTs). The inner DFTs employ the principal Jth root of unity ω^K, while the outer DFTs work with the principal Kth root of unity ω^J. Hence, most powers of ω (twiddle factors) used during the transformation are powers of ω^J or ω^K. Only the scalar multiplications in the middle are by "odd" powers of ω. This general recursive decomposition of the DFT has in fact been presented in the original paper of Cooley and Tukey [CT65]. Any such recursive decomposition (even for $J = 2$ or $K = 2$) results in a fast algorithm for the DFT and is called *Fast Fourier Transform (FFT)*. At one time, the FFT has been fully credited to Cooley and Tukey, but the FFT has appeared earlier. For the older history of the FFT back to Gauss, the reader is referred to [HJB84].

Here, we are only interested in the usual case of N being a power of 2. Instead of using j and k ranging from 0 to $N-1$, we use $j'J+j$ and $k'K+k$ with $0 \le j, k' \le J-1$ and $0 \le j', k \le K-1$. Almost any textbook presenting the Fourier transformation recursively would use either $K = 2$ or $J = 2$.

For $0 \le j \le J-1$ and $0 \le j' \le K-1$, Equation 2.1 transforms into the following equation, which after minor manipulations exhibits an arbitrary recursive decomposition of the Fourier transform.

$$\begin{aligned} b_{j'J+j} &= \sum_{k=0}^{K-1}\sum_{k'=0}^{J-1}\omega^{(j'J+j)(k'K+k)}a_{k'K+k} \\ &= \underbrace{\sum_{k=0}^{K-1}\omega^{Jj'k}\underbrace{\omega^{jk}\underbrace{\sum_{k'=0}^{J-1}\omega^{Kjk'}a_{k'K+k}}_{\text{inner (first) DFTs}}}_{\text{coefficients of outer DFTs}}}_{\text{outer (second) DFTs}} \end{aligned}$$

For N being a power of 2, the fast Fourier transforms (FFTs) are obtained by recursive application of this method until $N = 2$.

We could apply a *balanced* FFT with $J = K = \sqrt{N}$ or $J = 2K = \sqrt{2N}$ depending on N being an even or odd power of 2. But actually, we just require the partition not to be extremely unbalanced.

4. The Ring $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$. We consider the ring of polynomials $\mathcal{R}[y]$ over the ring $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$. In all applications, we will assume P to be a power of 2. For a primitive $2P$th root of unity η in $\mathbb{C}$, e.g., $\eta = e^{i\pi/P}$, we have

$$\begin{aligned}\mathcal{R} &= \mathbb{C}[x]/(x^P + 1)\\ &= \mathbb{C}[x]/\prod_{j=0}^{P-1}(x - \eta^{2j+1})\\ &\cong \prod_{j=0}^{P-1}\mathbb{C}[x]/(x - \eta^{2j+1})\\ &\cong \mathbb{C}^P\end{aligned}$$

The first isomorphism is provided by an easy version of the Chinese Remainder Theorem for fields.

We want to do Half-DFTs over the ring $\mathcal{R}$. We notice that polynomials over $\mathcal{R}$ decompose into products of polynomials over $\mathbb{C}$.

$$\mathcal{R}[y] = \mathbb{C}[x]/(x^P + 1)[y] \cong \mathbb{C}^P[y] \cong \mathbb{C}[y]^P$$

Each component $\mathbb{C}[y]$ is a principal ideal domain and therefore it is factorial, i.e., it has unique factorization. The isomorphic image $(\zeta_0, \zeta_1, \ldots, \zeta_{P-1})^\mathsf{T}$ in $\mathbb{C}^P$ of a principal $2N$th root of unity ζ in $\mathcal{R}$ is a principal $2N$th root of unity. Thus each of its components ζ_k is a principal $2N$th root of unity in the field $\mathbb{C}$, implying that $(y - \zeta_k^{2j+1})$ divides $(y^N + 1)$ and $\gcd(y - \zeta_k^{2i+1}, y - \zeta_k^{2j+1}) = \zeta_k^{2i+1} - \zeta_k^{2j+1}$ are units in $\mathbb{C}$ for all $i \neq j$ with $0 \leq i, j < N$. As a consequence of unique factorization in $\mathbb{C}[y]$, not only each $(y - \zeta_k^{2j+1})$, but also $\prod_{j=0}^{N-1}(y - \zeta_k^{2j+1})$ divides $y^N - 1$. Just looking at the coefficient of y^N, we see that $\prod_{j=0}^{N-1}(y - \zeta_k^{2j+1}) = y^N + 1$.

We apply the Chinese Remainder Theorem for $I_j = (y - \zeta_k^{2j+1})$. We have seen that $I_j = (y - \zeta_k^{2j+1})$ divides $(y^N + 1)$ for all j, implying $(y^N + 1) \subseteq \bigcap_{j=0}^{N-1} I_j = I$. The opposite containment is nontrivial. Every element of I is not only a multiple of $y - \zeta_k^{2j+1}$ for each j, but because of unique factorization also a multiple of their product $\prod_{j=0}^{N-1}(y - \zeta_k^{2j+1})$, i.e., $I \subseteq (y^N + 1)$. Thus we obtain the following result.

LEMMA 4.1. $\mathcal{R}[y]/(y^N + 1) \cong \prod_{j=0}^{N-1} \mathcal{R}[y]/(y - \zeta^{2j+1})$ *and the Half-DFT produces this isomorphism.*

$\mathcal{R}$ contains an interesting principal $2P$th root of unity, namely x. This follows from Lemma 2.1, because $x^P = -1$ in $\mathcal{R}$. Alternatively, because $\mathcal{R}$ is isomorphic to $\mathbb{C}^P$, a $\zeta \in \mathcal{R}$ is a principal mth root of unity, if and only if it is a principal mth root of unity in every factor $\mathbb{C}[x]/(x - \eta^{2j+1})$ of $\mathcal{R}$. But $x \bmod (x - \eta^{2j+1})$ is just η^{2j+1}, which is a principal $2P$th root of unity in $\mathbb{C}$.

LEMMA 4.2. *For P a power of 2, the variable x is a principal $2P$th root of unity in the ring $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$.*

x is a very desirable root of unity, because multiplication by x can be done very efficiently. As $x^P = -1$, multiplication by x is just a cyclic shift of the polynomial coefficients with a sign change on wrap around.

There are many principal $2N$th roots of unity in

$$\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$$

One can choose an arbitrary primitive $2N$th root of unity in every factor

$$\mathbb{C}[x]/(x - \eta^{2j+1})$$

independently. We want to pick one such $2N$th root of unity

$$\rho \in \mathcal{R} = \mathbb{C}[x]/(x^P + 1)$$

with the convenient property

$$\rho^{N/P} = x$$

We write $\rho(x)$ for ρ to emphasize that it is represented by a polynomial in x. Let σ be a primitive $2N$th root of unity in $\mathbb{C}$, with

$$\sigma^{N/P} = \eta$$

e.g.,

$$\sigma = e^{i\pi/N}$$

Now we select the polynomial

$$\rho(x) = \sum_{j=0}^{P-1} r_j x^j$$

such that

$$\rho(x) \equiv \sigma^{2k+1} \pmod{x - \eta^{2k+1}} \quad \text{for } k = 0, 1, \ldots, P-1$$

i.e., σ^{2k+1} is the value of the polynomial $\rho(x)$ at η^{2k+1}. Then

$$\rho(x)^{N/P} \equiv \sigma^{(2k+1)N/P} = \eta^{2k+1} \equiv x \pmod{x - \eta^{2k+1}}$$

for $0 \leq k < P$, implying

$$\rho(x)^{N/P} \equiv x \pmod{x^P + 1}$$

For the FFT algorithm, the coefficients of $\rho(x)$ could be computed from Lagrange's interpolation formula

$$\rho(x) = \sum_{k=0}^{P-1} \sigma^{2k+1} \frac{\prod_{j \neq k}(x - \eta^{2j+1})}{\prod_{j \neq k}(\eta^{2k+1} - \eta^{2j+1})}$$

without affecting the asymptotic running time, because $P = O(\log N)$ is small.

In both products, j ranges over $\{0, \ldots, P-1\} \setminus \{k\}$. The numerator in the previous expression is

$$\frac{x^P + 1}{x - \eta^{2k+1}} = -\sum_{j=0}^{P-1} \eta^{-(j+1)(2k+1)} x^j$$

This implies that in our case, all coefficients of each of the additive terms in Lagrange's formula have the same absolute value. We also want to show that all coefficients of $\rho(x)$ have an absolute value of at most 1.

DEFINITION 4.3. *The* l_2-norm *of a polynomial* $p(x) = \sum a_k x^k$ *is* $||p(x)|| = \sqrt{\sum |a_k|^2}$.

Our FFT will be done with the principal root of unity $\rho(x)$ defined above. In order to control the required numerical accuracy of our computations, we need a bound on the absolute value of the coefficients of $\rho(x)$. Such a bound is provided by the l_2-norm $||\rho(x)||$ of $\rho(x)$.

LEMMA 4.4. *The* l_2*-norm of* $\rho(x)$ *is* $||\rho(x)|| = 1$.

Proof. Note that the values of the polynomial $\rho(x)$ at all the primitive $2P$th roots of unity are also roots of unity, in particular complex numbers with absolute value 1. Thus the vector $\boldsymbol{b}$ of these values has l_2-norm $\sqrt{P}$. The coefficients of $\rho(x)$ are obtained by an inverse half discrete Fourier transform $Z'^{-1}\boldsymbol{b}$. As $\sqrt{P}Z'^{-1}$ is a unitary matrix, the vector of coefficients has norm 1. □

COROLLARY 4.5. *The absolute value of every coefficient of* $\rho(x)$ *is at most 1.*

5. The Algorithm. In order to multiply two non-negative integers modulo 2^n+1, we encode them as polynomials of $\mathcal{R}[y]$, where $\mathcal{R} = \mathbb{C}[x]/(x^P+1)$, and we multiply these polynomials with the help of the Fourier transform as follows. Let $P = \Theta(\log n)$ be rounded to a power of 2. The binary integers to be multiplied are decomposed into (large) pieces of length $P^2/2$. Again, each such piece is decomposed into small pieces of length P. If $a_{i\,P/2-1}, \ldots, a_{i0}$ are the small pieces belonging to a common big piece a_i, then they are encoded as

$$\tilde{a}_i = \sum_{j=0}^{P-1} a_{ij} x^j \in \mathcal{R} = \mathbb{C}[x]/(x^P+1)$$

with

$$a_{i\,P-1} = a_{i\,P-2} = \cdots = a_{i\,P/2} = 0$$

Thus each large piece is encoded as an element of $\mathcal{R}$, which is a coefficient of a polynomial in y.

These elements of $\mathcal{R}$ are themselves polynomials in x. Their coefficients are integers at the beginning and at the end of the algorithm. The intermediate results, as well as the roots of unity are polynomials with complex coefficients, which themselves are represented by pairs of reals that have to be approximated numerically. In Section 6, we will show that it is sufficient to use fixed-point arithmetic with $O(P) = O(\log n)$ bits in the integer and fraction part.

Now every factor $\sum_{i=0}^{N-1} \tilde{a}_i 2^{iP^2/2}$ is represented by a polynomial $\sum_{i=0}^{N-1} a_i y^i \in \mathcal{R}[y]$. A Half-FFT computes the values of such a polynomial at those roots of unity which are odd powers of the $2N$th root of unity $\rho(x) \in \mathcal{R}$, defined in the previous section. The values are multiplied and an inverse Half-FFT produces another polynomial of $\mathcal{R}[y]$. From this polynomial the resulting integer product can be recovered by just doing some simple additions. The relevant parts of the coefficients have now grown to some length $O(P)$ from the initial length of P. (The constant factor growth could actually be decreased to a factor $2 + o(1)$ by increasing the parameter P from $\Theta(\log n)$ to $\Theta(\log^2 n)$, but this would only affect the constant factor in front of $\lg^*$ in the exponent of the running time.)

Thus the algorithm runs pretty much like that of Schönhage and Strassen [SS71] except that the field $\mathbb{C}$ or the ring of integers modulo the mth Fermat prime F_m has been replaced by the ring $\mathcal{R} = \mathbb{C}[x]/(x^P+1)$, and the FFT is decomposed more evenly. The standard decomposition of the N-point FFT into two $N/2$-point FFTs and many 2-point FFTs would not allow such an improvement. Nevertheless, there is no need for balancing completely. Instead of recursively decomposing the $N = \Theta(\frac{n}{\log^2 n})$-point FFT in the middle (in a divide-and-conquer fashion), we decompose into $2P$-point FFTs and $N/(2P)$-point FFTs. This is mainly done for simplicity. Both versions are efficient (even though with different constant factors in front of $\lg^*$ in the exponent), as only about every $\log P$th level of the overall FFT requires complicated multiplications with difficult roots of unity (twiddle factors). At all the other levels, the twiddle factors are powers of x. Multiplications with these twiddle factors are just cyclic rotations of the P-tuple of coefficients of elements of $\mathcal{R}$, with a sign change on wrap around.

We use the auxiliary functions Decompose (Figure 5.1) and Compose (Figure 5.2).

Procedure Decompose:

Input: Integer a of length at most $n = NP^2/2$ in binary; N, P (powers of 2)
Output: $\boldsymbol{a} \in \mathcal{R}^N$ (or $\alpha \in \mathcal{R}[y]$) encoding the integer a
Comment: The integer a is the concatenation of the a_{ij} for $0 \leq i < N$ and $0 \leq j < P/2$ as binary integers of length P defined by Equations (5.1), (5.2) and (5.4), and Inequalities (5.3). $a_{i0}, a_{i1}, \ldots, a_{i\,P-1}$ are the coefficients of $\alpha_i \in \mathcal{R}$. $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$ are the components of $\boldsymbol{a} \in \mathcal{R}^N$ as defined by Equation (5.5).

for $i = 0$ **to** $N-1$ **do**
  for $j = 0$ **to** $P/2-1$ **do**
    $a_{ij} = a \bmod 2^P$
    $a = \lfloor a/2^P \rfloor$
  for $j = P/2$ **to** $P-1$ **do**
    $a_{ij} = 0$
  $\alpha_i = a_{i0} + a_{i1}x + a_{i2}x^2 + \cdots + a_{i\,P-1}x^{P-1}$
Return $\boldsymbol{a} = (\alpha_0, \ldots, \alpha_{N-1})$

FIG. 5.1. *The procedure Decompose*

"Decompose" takes a binary number a of length $n = NP^2/2$. First a is decomposed into N pieces $a_{N-1}, a_{N-2}, \ldots, a_0$ of length $P^2/2$ each. Then every a_i is decomposed into $P/2$ pieces $a_{i\,P/2-1}, a_{i,\,P/2-2}, \ldots, a_{i\,0}$ of length P each. The remaining a_{ij} (for $0 \leq i < N$ and $P/2 \leq j < P$) are defined to be 0. This padding allows to properly recover the integer product from the product of the polynomials. In other words, we have

$$a = \sum_{i=0}^{N-1} a_i 2^{iP^2/2} \quad \text{and} \quad a_i = \sum_{j=0}^{P-1} a_{ij} 2^{jP} \tag{5.1}$$

implying

$$a = \sum_{i=0}^{N-1} \sum_{j=0}^{P-1} a_{ij} 2^{i(P^2/2)+jP} \tag{5.2}$$

Procedure Compose:

Input: $\boldsymbol{a} \in \mathcal{R}^N$, N, P (powers of 2)
Output: Integer a encoded by $\boldsymbol{a}$
Comment: $\alpha_0, \alpha_1, \ldots, \alpha_{N-1}$ are the components of a vector $\boldsymbol{a} \in \mathcal{R}^N$. For all i, j, a_{ij} is the coefficient of x^j in α_i. The integer a is obtained from the rounded a_{ij} as defined in Equation (5.2).

```
round all a_ij to the nearest integer
a = 0
for j = P − 1 downto P/2 do
    a = a · 2^P + a_{N−1 j}
for i = N − 1 downto 1 do
    for j = P/2 − 1 downto 0 do
        a = a · 2^P + a_ij + a_{i−1 j+P/2}
for j = P/2 − 1 downto 0 do
    a = a · 2^P + a_0j
Return a mod (2^n + 1)
```

Fig. 5.2. *The procedure Compose*

with

$$0 \leq a_{ij} < 2^P \quad \text{for all } i, j \tag{5.3}$$

and

$$a_{ij} = 0 \quad \text{for } 0 \leq i < N \text{ and } P/2 \leq j < P \tag{5.4}$$

We use Greek letters to denote elements of the ring $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$. The number a_i is encoded as an element $\alpha_i \in \mathcal{R}$ and a is encoded as a polynomial

$$\alpha = \sum_{i=0}^{N-1} \alpha_i y^i \in \mathcal{R}[y]$$

represented by its vector of coefficients $\boldsymbol{a} = (\alpha_0, \ldots, \alpha_{N-1})^\mathsf{T} \in \mathcal{R}^N$, with

$$\alpha_i = \sum_{j=0}^{P-1} a_{ij} x^j = a_{i0} + a_{i1}x + a_{i2}x^2 + \cdots + a_{i\,P-1}x^{P-1} \tag{5.5}$$

We say that $\boldsymbol{a}$ *represents* the integer a, when (5.1) and (5.2) hold. Typically an integer a ($0 \leq a < 2^{NP^2/2}$) has many representations. “Decompose” selects the unique representation in normal form, defined by (5.3) and (5.4).

In normal form, the padding (defined by (5.4)) is designed to avoid any wrap around modulo $x^P + 1$ when doing multiplication in $\mathcal{R}$. “Compose” not only reverses the effect of “Decompose”, but it works just as well for arbitrary representations not in normal form, which are produced during the computation.

To be precise, no wrap around modulo $x^P + 1$ would occur, if the product of polynomials were computed with the standard school multiplication. During the actual computation of the product using FFT, plenty of wrap arounds happen, but

Procedure Select:

Input: $N \geq 4$ (a power of 2), P (a power of 2)
Output: $J \geq 2$ (a power of 2 dividing $N/2$)
Comment: The procedure selects J such that the N-point FFT is decomposed into J point FFTs followed by $K = N/J$-point FFTs.

if $N \leq 2P$ **then** Return 2 **else** Return $2P$

FIG. 5.3. *The procedure Select determining the recursive decomposition*

naturally the final result is just the product of the polynomials, i.e., the same as if it were computed with school multiplication.

The procedure Select(N) (Figure 5.3) determines how the FFT is broken down recursively, corresponding to a factorization $N = JK$. Schönhage has used Select(N) = 2, Aho, Hopcroft and Ullman use Select(N) = $N/2$, a balanced approach corresponds to Select(N) = $2^{\lfloor (\lg N)/2 \rfloor}$. We choose Select($N$) = $2P$, which is slightly better than a balanced solution (only by a constant factor in front of $\lg^* n$), because with our choice of Select(N) only every $2P$th level (instead of every ith for some $P < i \leq 2P$) requires expensive multiplications with twiddle factors.

The Structure of the Complete Algorithm:

Comment: Lower level (indented) algorithms are called from the higher level algorithms. In addition, the algorithm FFT calls itself recursively, and it calls the auxiliary procedure Select. Furthermore, in FFT and Componentwise-Multiplication, the Operation $*$ calls Modular-Integer-Multiplication recursively.

- Integer-Multiplication
  - Modular-Integer-Multiplication
    - Decompose
    - Half-FFT
      - FFT
    - Componentwise-Multiplication
    - Half-Inverse-FFT
      - FFT
    - Compose

FIG. 5.4. *Overall structure of the multiplication algorithm*

With the help of these auxiliary procedures, we can now give an overview of the whole integer multiplication algorithm in Figure 5.4. A more detailed descrip-

Various functions:

lg: the log to the base 2
length: the length in binary
round: rounded up to the next power of 2

FIG. 5.5. *Various functions*

tion of the various procedures follows. Integer-Multiplication and Modular-Integer-Multiplication also use the auxiliary functions of Figure 5.5.

The previously presented procedures Decompose and Compose are simple format conversions. The three major parts of Modular-Integer-Multiplication are Half-FFT (Figure 5.8) for both factors, Componentwise-Multiplication (Figure 5.10), and Inverse-Half-FFT (Figure 5.11). The crucial part, FFT, is presented as a recursive algorithm for simplicity and clarity. It uses the auxiliary procedure Select (Figure 5.3). The algorithms FFT and Componentwise-Multiplication, use the operation $*$ (Figure 5.12), which is the multiplication in the ring $\mathcal{R}$. This operation is implemented by modular integer multiplication, which is executed by recursive calls to the procedure Modular-Integer-Multiplication (Figure 5.7).

We compute the product of two complex polynomials (elements of $\mathcal{R}$) by first writing each as a sum of a real and an imaginary polynomial, and then computing four products of real polynomials. Alternatively, we could achieve the same result with three real polynomial multiplications based on the basic idea of [KO62]. Real polynomials (with coefficients given in fixed point representation) are multiplied by multiplying their values at a good power of 2 as proposed by Schönhage [Sch82]. A good power of 2 makes the values not too big, but still allows the coefficients of the product polynomial to be easily recovered from the binary representation of the integer product. The case of positive integer coefficients is particularly intuitive. A binary integer is formed by concatenating the coefficients after padding them with leading zeros such that all have equal length and the non-zero parts are well separated.

Schönhage [Sch82] has shown that such an encoding can easily be achieved with a constant factor blow-up. Actually, he proposes an even better method for handling complex polynomials. He does integer multiplication modulo $2^N + 1$ and notices that $2^{N/2}$ can serve as the imaginary unit i. We don't further elaborate on this method, as it only affects the constant factor in front of $\lg^* n$ in the exponent of the running time. An even better method has been suggested by Bernstein [Ber]. In the definition of the ring $\mathcal{R}$, the field $\mathbb{C}$ could be replaced by $\mathbb{R}$, as $x^{P/2}$ could be viewed as the imaginary unit i.

Algorithm Integer-Multiplication:

Input: Integers a and b in binary
Output: Product $d = ab$
Comment: The product $d = ab$ is computed with Modular-Integer-Multiplication. The length n is chosen to be a power of 2 sufficiently large to avoid any warp around.

$n =$ round(length(a) + length(b))
Return Modular-Integer-Multiplication(n, a, b)

FIG. 5.6. *The Algorithm Integer-Multiplication*

6. Precision for the FFT over $\mathcal{R}$. We compute Fourier transforms over the ring $\mathcal{R}$. Elements of $\mathcal{R}$ are polynomials over $\mathbb{C}$ modulo $x^P + 1$. The coefficients are represented by pairs of reals with fixed precision for the real and imaginary part. We want to know the numerical precision needed for the coefficients of these polynomials.

Algorithm Modular-Integer-Multiplication:

Input: Integer n, Integers a and b modulo $2^n + 1$ in binary
Output: Product $d = ab \bmod 2^n + 1$
Comment: n is a power of 2. The product $d = ab \bmod 2^n + 1$ is computed with half Fourier transforms over $\mathcal{R}$. Let ζ be the $2N$th root of unity in $\mathcal{R}$ with value $e^{i\pi(2k+1)/N}$ at $e^{i\pi(2k+1)/P}$ (for $k = 0, \ldots, P-1$). Let $n_0 \geq 16$ be some constant. For $n \leq n_0$, a trivial multiplication algorithm is used.

if $n \leq n_0$ **then** Return $ab \bmod 2^n + 1$
$P = \mathrm{round}(\lg n)$
$N = 2n/P^2$
$\boldsymbol{a} =$ Half-FFT$(\mathit{Decompose}(a), \zeta, N, P)$
$\boldsymbol{b} =$ Half-FFT$(\mathit{Decompose}(b), \zeta, N, P)$
$\boldsymbol{c} =$ Componentwise-Multiplication$(\boldsymbol{a}, \boldsymbol{b}, N, P)$
$\boldsymbol{d} =$ Inverse-Half-FFT$(\boldsymbol{c}, \zeta, N, P)$
Return $\mathit{Compose}(\boldsymbol{d})$

Fig. 5.7. *The Algorithm Modular-Integer-Multiplication*

Algorithm Half-FFT:

Input: $\boldsymbol{a} = (\alpha_0, \ldots, \alpha_{N-1})^{\mathsf{T}} \in \mathcal{R}^N$, $\zeta \in \mathcal{R} = \mathbb{C}[x]/(x^P + 1)$ (ζ is a principal $2N$th root of unity in $\mathcal{R}$ with $\zeta^{N/P} = x$), N, P (powers of 2),
Output: $\boldsymbol{b} \in \mathcal{R}^N$ the N-point half DFT of the input
Comment: The N-point half DFT is the evaluation of the polynomial with coefficient vector $\boldsymbol{a}$ at the odd powers of ζ, i.e., in those powers of ζ that are principal $2N$th roots of unity in $\mathcal{R}$.

for $k = 0$ **to** $N - 1$ **do**
  $\alpha_k = \alpha_k \zeta^k$
$\omega = \zeta^2$
Return FFT$(\boldsymbol{a}, \omega, N, P)$

Fig. 5.8. *The algorithm half-FFT*

We start with integer coefficients. After doing two Half-FFTs in parallel, and multiplying the corresponding values followed by an inverse Half-FFT, we know that the result has again integer coefficients. Therefore, the precision has to be such that at the end the absolute errors are less than $\frac{1}{2}$. Hence, a set of final rounding operations provably produces the correct result.

We do all computations with at least S bits of precision, where S is fixed (as a function of n) and will be determined later. We use at least S bits for the real as well as the imaginary part of each complex number occurring in the FFT algorithms. In addition, there is a sign to be stored with each number. Of the S bits, we use at least V bits before the binary point and at least $S - V$ bits after the binary point. V varies throughout the algorithm. We are very generous with V and S, meaning that the bits in the integer part might include many leading zeros and the number of bits in the fractional part might be unnecessarily high. The main purpose is to prove correctness. Tighter bounds would improve the constant factor hidden by the O-notation in the running time.

In a practical implementation, one can either use floating point arithmetic with

Algorithm FFT:

Input: $\boldsymbol{a} = (\alpha_0, \ldots, \alpha_{N-1})^\mathsf{T} \in \mathcal{R}^N$, $\omega \in \mathcal{R} = \mathbb{C}[x]/(x^P + 1)$ (ω is an Nth root of unity in $\mathcal{R}$ with $\omega^{N/2P} = x$; $\omega = x^{2P/N}$ for $N < 2P$), N, P (powers of 2),

Output: $\boldsymbol{b} \in \mathcal{R}^N$ the N-point DFT of the input

Comment: The N-point FFT is the composition of J-point inner FFTs and K-point outer FFTs. We use the vectors $\boldsymbol{a}$, $\boldsymbol{b} \in \mathcal{R}^N$, $\boldsymbol{c}^k = (\gamma_0^k, \ldots, \gamma_{J-1}^k)^\mathsf{T} \in \mathcal{R}^J$ ($k = 0, \ldots, K-1$), and $\boldsymbol{d}^j = (\delta_0^j, \ldots, \delta_{K-1}^j)^\mathsf{T} \in \mathcal{R}^K$ ($j = 0, \ldots, J-1$).

```
if N = 1 then Return a
if N = 2 then {β_0 = α_0 + α_1; β_1 = α_0 − α_1; Return b = (β_0, ..., β_{N−1})^T}
J = Select(N, P); K = N/J
for k = 0 to K − 1 do
    for k' = 0 to J − 1 do
        γ^k_{k'} = α_{k'K+k}
    c^k = FFT(c^k, ω^K, J)  //inner FFTs
for j = 0 to J − 1 do
    for k = 0 to K − 1 do
        δ^j_k = γ^k_j * ω^{jk}
    d^j = FFT(d^j, ω^J, K)  //outer FFTs
    for j' = 0 to K − 1 do
        β_{j'J+j} = δ^j_{j'}
Return b = (β_0, ..., β_{N−1})^T
```

FIG. 5.9. *The algorithm FFT*

Algorithm Componentwise-Multiplication:

Input: $\boldsymbol{a} = (\alpha_0, \ldots, \alpha_{N-1})^\mathsf{T}, \boldsymbol{b} = (\beta_0, \ldots, \beta_{N-1})^\mathsf{T} \in \mathcal{R}^N$, N, P (powers of 2)

Output: $\boldsymbol{c} \in \mathcal{R}^N$ (the componentwise product of $\boldsymbol{a}$ and $\boldsymbol{b}$)

```
for j = 0 to N − 1 do
    γ_j = α_j * β_j
Return c = (γ_0. ..., γ_{N−1})^T
```

FIG. 5.10. *The algorithm Componentwise-Multiplication*

Algorithm Inverse-Half-FFT:

Input: $\boldsymbol{a} = (\alpha_0, \ldots, \alpha_{N-1})^\mathsf{T} \in \mathcal{R}^N$, $\zeta \in \mathcal{R}$ (a principal $2N$th root of unity in $\mathcal{R}$), N, P (powers of 2)

Output: $\boldsymbol{b} \in \mathcal{R}^N$ (the inverse of the half N-point DFT applied to the input)

```
ω = ζ^2
b = (1/N) FFT(a, ω^{−1}, N, P)
for k = 0 to N − 1 do
    β_k = β_k ζ^{−k}
Return b = (β_0, ..., β_{N−1})^T
```

FIG. 5.11. *The algorithm Inverse-FFT*

at least S bits in the mantissa, or one could always scale with the known appropriate power of 2 and use integer arithmetic.

For the initial call to Half-FFT, the fractional part after the binary point is 0, Operation * (= Multiplication in $\mathcal{R}$):

Input: $\alpha, \beta \in \mathcal{R}$, P (a power of 2)

Output: γ (the product $\alpha \cdot \beta \in \mathcal{R}$)

Comment: First write each of the two polynomials as a sum of a real and an imaginary polynomial. Then compute the 4 products of real polynomials by multiplying their values at a good power of 2, which pads the space between the coefficients nicely such that the coefficients of the product polynomial can easily be recovered from the binary representation of the integer product.
The details can easily be filled in, as the coefficients are presented in fixed-point arithmetic. Precise bounds on the lengths of the integer and fractional parts are given in Section 6.

FIG. 5.12. *The multiplication in $\mathcal{R}$ (= operation *)*

and we use $V = P$ bits in the integer part in front of the binary point. In each level of the FFT, we do everywhere an addition or subtraction and a multiplication with a twiddle factor (which might be 1). We generously increase V by 1 for each addition or subtraction. Most multiplications with twiddle factors are handled by cyclic shifts producing no errors. At every level divisible by $\lg P + 1$, the multiplications by twiddle factors are general multiplications in $\mathcal{R}$.

We first investigate the growth of the value and error bounds during these multiplications. Elements of $\mathcal{R}$ are represented by polynomials in x of degree $P-1$.

NOTATION 1. *We refer to the real or imaginary part of any coefficient of an element of $\mathcal{R}$ simply as a* part.

Let r be a part of any element of $\mathcal{R}$ occurring in an idealized infinite precision algorithm. In reality, a finite precision algorithm uses an approximation $r + \varepsilon_r$ instead of r.

We say that at some stage of an algorithm, we have a *value bound* v and an *error bound* e, if we have the following bounds for all parts r.

$$|r| \leq v, \qquad |\varepsilon_r| \leq e$$

Our bounds are always powers of 2. Whenever we have a value bound v, we do all computations with $V = \lg v$ bits before the binary point. For twiddle factors (which are also elements of $\mathcal{R}$), we have the following stricter requirements for all its parts t.

$$|t + \varepsilon_t| \leq 1, \qquad |\varepsilon_t| \leq 2^{-S}$$

LEMMA 6.1. *Let $v_{\boldsymbol{c}}$ be a value bound and $e_{\boldsymbol{c}}$ an error bound on the parts of an element of $\mathcal{R}$ before a multiplication with a twiddle factor. Then*

$$v_{\boldsymbol{d}} = 2Pv_{\boldsymbol{c}}$$

is a value bound and

$$\begin{aligned} e_{\boldsymbol{d}} &= 2Pe_{\boldsymbol{c}} + v_{\boldsymbol{d}} 2^{-S+1} \\ &= 2P(e_{\boldsymbol{c}} + v_{\boldsymbol{c}} 2^{-S+1}) \end{aligned}$$

is an error bound after the multiplication.

Proof. All parts (real and imaginary parts of coefficients) t of twiddle factors have an absolute value of at most 1. Therefore, the value bound on $|rt|$ is the same as the bound on $|r|$. All multiplications of parts are of the form $(r+\varepsilon_r)(t+\varepsilon_t)$, where $r+\varepsilon_r$ is the current approximation to a part r, and $t+\varepsilon_t$ is the approximation to a part t of a twiddle factor. Using the bounds $|r| \leq v_{\boldsymbol{c}}$, $|\varepsilon_r| \leq e_{\boldsymbol{c}}$, $|t+\varepsilon_t| \leq 1$, and $|\varepsilon_t| \leq 2^{-S}$, where $v_{\boldsymbol{c}}$ and $e_{\boldsymbol{c}}$ are the current value and error bounds respectively, associated with parts r, we see that the error after an exact multiplication of the approximated parts is

$$|\varepsilon_{rt}| = |(t+\varepsilon_t)\varepsilon_r + r\varepsilon_t| \leq e_{\boldsymbol{c}} + v_{\boldsymbol{c}} 2^{-S}$$

Thus the absolute value of the old error $|\varepsilon_r| \leq e_{\boldsymbol{c}}$ does not increase during this multiplication of parts, but due to the error ε_t in a part of the twiddle factor, a new error of at most $v_{\boldsymbol{c}} 2^{-S}$ is created.

Every coefficient of the product in $\mathcal{R}$ is the sum of P products of coefficients of the two factors. As these coefficients are complex numbers, each part of the product of two coefficients involves two products of real numbers. Thus, we obtain a trivial upper bound $v_{\boldsymbol{d}}$ on the absolute values of the parts of the product if we multiply the upper bound on products of parts $v_{\boldsymbol{c}}$ by $2P$.

Similarly, the error bound of $e_{\boldsymbol{c}} + v_{\boldsymbol{c}} 2^{-S}$ for the product of two parts is multiplied by $2P$ to obtain the error bound $e_{\boldsymbol{d}}$ for the parts of the product. Note that non-trivial multiplication in $\mathcal{R}$ is done by reduction to integer multiplication. Thus, starting from the representations with S bits per part, initially all multiplications and additions are done exactly, i.e., without any rounding in between. But finally all parts are rounded, creating an additional new error of at most $v_{\boldsymbol{d}} 2^{-S}$. Thus the total new error for multiplication with roots of unity in $\mathcal{R}$ is at most $4Pv_{\boldsymbol{c}} 2^{-S}$, resulting in a total error bound as claimed by the lemma. □

The proof of the following lemma shows value and error bounds by induction based on the recursive structure of the FFT algorithm. For any vector $\boldsymbol{a}$ over $\mathcal{R}$, let $v_{\boldsymbol{a}}$ be a bound on the absolute values of the parts (real and imaginary parts of any coefficient) of any component of $\boldsymbol{a}$, and let $e_{\boldsymbol{a}}$ be a bound on the absolute values of the error in the parts of any component of $\boldsymbol{a}$. Note that the following lemma considers an arbitrary error bound at the start, because FFT as a recursive procedure is also called in the middle of other FFT computations.

LEMMA 6.2. *Let N, P be powers of 2 with $N \geq 2$ and $P \geq 1$. Let $L = \lceil (\lg N)/\lg(2P) \rceil - 1$ be the number of levels with computationally intensive twiddle factors. If the input $\boldsymbol{a}$ of an N-point FFT has a value bound $v_{\boldsymbol{a}}$ and an error bound $e_{\boldsymbol{a}}$, then the output $\boldsymbol{b}$ has a value bound*

$$v_{\boldsymbol{b}} = N(2P)^L v_{\boldsymbol{a}} \leq N^2 v_{\boldsymbol{a}}$$

and an error bound

$$\begin{aligned} e_{\boldsymbol{b}} &= N(2P)^L e_{\boldsymbol{a}} + v_{\boldsymbol{b}} 2^{-S} (\lg N + 2L) \\ &= N(2P)^L \left(e_{\boldsymbol{a}} + v_{\boldsymbol{a}} 2^{-S} (\lg N + 2L) \right) \\ &\leq N^2 (e_{\boldsymbol{a}} + v_{\boldsymbol{a}} 2^{-S+1} \lg N) \end{aligned}$$

Proof. $(2P)^L \leq N$ immediately follows from the definition of L implying both inequalities. We show the other bounds by induction on N. We note that $L = 0$ for $N \leq 2P$.

For $N = 2$, the algorithm does one addition or subtraction at most doubling the previous values (bounded by $v_{\boldsymbol{a}}$) and the previous errors (bounded by $e_{\boldsymbol{a}}$). Furthermore, the result is rounded in the last position, creating a new error of at most $v_{\boldsymbol{b}} 2^{-S}$. Thus $v_{\boldsymbol{b}} = 2v_{\boldsymbol{a}}$ is a value bound and $e_{\boldsymbol{b}} = 2e_{\boldsymbol{a}} + v_{\boldsymbol{b}} 2^{-S}$ is an error bound.

For $N > 2$, the FFT is a composition of inner FFTs (computing $\boldsymbol{c}$ from $\boldsymbol{a}$), multiplications with twiddle factors (computing $\boldsymbol{d}$ from $\boldsymbol{c}$), and outer FFTs (computing $\boldsymbol{b}$ from $\boldsymbol{d}$). We use the inductive hypotheses for the inner and outer FFTs.

For $2 < N \leq 2P$, after the inner 2-point FFTs, we have a value bound of $v_{\boldsymbol{c}} = 2v_{\boldsymbol{a}}$ and an error bound $e_{\boldsymbol{c}} = 2e_{\boldsymbol{a}} + v_{\boldsymbol{c}} 2^{-S}$. The multiplications with twiddle factors are just cyclic shifts without any new errors or bound increases, i.e., $v_{\boldsymbol{d}} = v_{\boldsymbol{c}}$ and $e_{\boldsymbol{d}} = e_{\boldsymbol{c}}$ are value and error bounds respectively. After the outer $N/2$-point FFTs, the inductive hypothesis implies a value bound of

$$v_{\boldsymbol{b}} = (N/2)v_{\boldsymbol{d}} = N v_{\boldsymbol{a}}$$

and the error bound of

$$\begin{aligned}
e_{\boldsymbol{b}} &= (N/2)e_{\boldsymbol{d}} + v_{\boldsymbol{b}} 2^{-S} \lg(N/2) \\
&= (N/2)(2e_{\boldsymbol{a}} + v_{\boldsymbol{d}} 2^{-S}) + v_{\boldsymbol{b}} 2^{-S} \lg(N/2) \\
&= N e_{\boldsymbol{a}} + v_{\boldsymbol{b}} 2^{-S} \lg N \\
&= N(e_{\boldsymbol{a}} + v_{\boldsymbol{a}} 2^{-S} \lg N)
\end{aligned}$$

For $N > 2P$, after the inner $2P$-point FFT,

$$v_{\boldsymbol{c}} = 2P v_{\boldsymbol{a}}$$

is a value bound, and

$$e_{\boldsymbol{c}} = 2P e_{\boldsymbol{a}} + v_{\boldsymbol{c}} 2^{-S} \lg(2P) = 2P(e_{\boldsymbol{a}} + v_{\boldsymbol{a}} 2^{-S} \lg(2P))$$

is an error bound.

Multiplication with the twiddle factors increases the value and the previous error by at most a factor of $2P$ and introduces a new rounding error of at most $v_{\boldsymbol{d}} 2^{-S}$ as well as an error with the same bound due to the inaccuracy in the twiddle factor as shown in Lemma 6.1. Thus

$$v_{\boldsymbol{d}} = 2P v_{\boldsymbol{c}} = (2P)^2 v_{\boldsymbol{a}}$$

is a value bound and

$$\begin{aligned}
e_{\boldsymbol{d}} &= 2P e_{\boldsymbol{c}} + v_{\boldsymbol{d}} 2^{-S+1} \\
&= (2P)^2 (e_{\boldsymbol{a}} + v_{\boldsymbol{a}} 2^{-S} \lg(2P)) + (2P)^2 v_{\boldsymbol{a}} 2^{-S+1} \\
&= (2P)^2 (e_{\boldsymbol{a}} + v_{\boldsymbol{a}} 2^{-S} (\lg(2P) + 2))
\end{aligned}$$

is an error bound.

Finally, after the outer $(\lg N - \lg(2P))$-point FFT, the inductive hypothesis provides a value bound of

$$v_{\boldsymbol{b}} = \frac{N}{2P} (2P)^{L-1} v_{\boldsymbol{d}} = N(2P)^L v_{\boldsymbol{a}}$$

and an error bound of

$$
\begin{aligned}
e_{\boldsymbol{b}} &= \frac{N}{2P}(2P)^{L-1}e_{\boldsymbol{d}} + v_{\boldsymbol{b}}2^{-S}\left(\lg\frac{N}{2P} + 2(L-1)\right)\\
&= N(2P)^{L-2}(2P)^2(e_{\boldsymbol{a}} + v_{\boldsymbol{a}}2^{-S}(\lg(2P)+2)) + v_{\boldsymbol{b}}2^{-S}\left(\lg\frac{N}{2P} + 2(L-1)\right)\\
&= N(2P)^{L}e_{\boldsymbol{a}} + v_{\boldsymbol{b}}2^{-S}(\lg(2P)+2) + v_{\boldsymbol{b}}2^{-S}\left(\lg\frac{N}{2P} + 2(L-1)\right)\\
&= N(2P)^{L}e_{\boldsymbol{a}} + v_{\boldsymbol{b}}2^{-S}(\lg N + 2L)\\
&= N(2P)^{L}(e_{\boldsymbol{a}} + v_{\boldsymbol{a}}2^{-S}(\lg N + 2L))
\end{aligned}
$$

□

After having computed the FFTs of both factors, we compute products of corresponding values. These are elements of $\mathcal{R}$. The following lemma controls the value and error bound for these multiplications. Let $v_{\boldsymbol{c}}$ and $e_{\boldsymbol{c}}$ refer to the value and error bounds of parts, i.e., real or imaginary parts of coefficients of the vectors $\boldsymbol{c}$ and $\boldsymbol{c}'$ representing the two factors.

Lemma 6.3. *If $v_{\boldsymbol{c}}$ is a value bound and $e_{\boldsymbol{c}}$ with $2^{-S} \le e_{\boldsymbol{c}} \le v_{\boldsymbol{c}}$ is an error bound for $\boldsymbol{c}$ and $\boldsymbol{c}'$ before the multiplications of the values, then $v_{\boldsymbol{d}} = 2Pv_{\boldsymbol{c}}^2$ is a value bound and $e_{\boldsymbol{d}} = 8Pv_{\boldsymbol{c}}e_{\boldsymbol{c}}$ is an error bound afterwards.*

Proof. As $v_{\boldsymbol{c}}$ is a value bound on the parts of the factors in $\boldsymbol{c}$ and $\boldsymbol{c}'$, obviously $v_{\boldsymbol{c}}^2$ is a value bound for their products. Because every part of a product in $\mathcal{R}$ is the sum of $2P$ products of parts, $v_{\boldsymbol{d}} = 2Pv_{\boldsymbol{c}}^2$ is a value bound for the result. The multiplications are of the form $(r+\varepsilon_r)(r'+\varepsilon_{r'})$, where $r+\varepsilon_r$ and $r'+\varepsilon_{r'}$ are the current approximations to parts of $\boldsymbol{c}$ and $\boldsymbol{c}'$. Using the fact that $|r|, |r'| \le v_{\boldsymbol{c}}$ and $|\varepsilon_r|, |\varepsilon_r'| \le e_{\boldsymbol{c}}$, where $e_{\boldsymbol{c}}$ is the current error bound, we see that the error after an exact multiplication of the approximated parts has a bound of $|r\varepsilon_{r'} + r'\varepsilon_r + \varepsilon_r\varepsilon_{r'}|$, which is generously bounded by $3v_{\boldsymbol{c}}e_{\boldsymbol{c}}$. During the additions, the error bound increases by a factor $2P$, and an additional error of at most $v_{\boldsymbol{d}}2^{-S} = 2Pv_{\boldsymbol{c}}^2 2^{-S}$ occurs due to rounding. Therefore, using the condition that $v_{\boldsymbol{c}}2^{-S} \le e_{\boldsymbol{c}}$ the rounded product has an error bound of

$$
e_{\boldsymbol{d}} \le 2P3v_{\boldsymbol{c}}e_{\boldsymbol{c}} + v_{\boldsymbol{d}}2^{-S} \le 6Pv_{\boldsymbol{c}}e_{\boldsymbol{c}} + 2Pv_{\boldsymbol{c}}^2 2^{-S} \le 8Pv_{\boldsymbol{c}}e_{\boldsymbol{c}}
$$

□

Lemma 6.4. *For $P = \mathrm{round}(\lg n) \ge 2$ and $N = \mathrm{round}(2n/P^2)$, where round is rounding to the next power of 2, precision $S \ge 5\lg N + \lg\lg(2N) + 2P + 4\lg P + 9$ is sufficient for the multiplication of integers of length n.*

Proof. We start with a value bound of 2^P and an error bound of 0. The initial multiplication of the Half-FFT with twiddle factors is analyzed in Lemma 6.1. Thus for the subsequent FFT, the value bound is $v_{\boldsymbol{a}} = 2P\,2^P$ and the error bound is $e_{\boldsymbol{a}} = v_{\boldsymbol{a}}2^{-S+1} = 2P\,2^P 2^{-S+1}$. By Lemma 6.2, the bounds after the N-point FFT are

$$
v_{\boldsymbol{c}} \le N^2 v_{\boldsymbol{a}} = 2P2^P N^2
$$

and

$$
e_{\boldsymbol{c}} \le N^2(e_{\boldsymbol{a}} + v_{\boldsymbol{a}}2^{-S+1}\lg N) = N^2 v_{\boldsymbol{a}}2^{-S+1}\lg(2N) = 2P\,2^P N^2\lg(2N)\,2^{-S+1}
$$

By Lemma 6.3, the bounds after the multiplication of values stage are

$$
v_{\boldsymbol{d}} = 2Pv_{\boldsymbol{c}}^2 \le (2P)^3 2^{2P} N^4
$$

and

$$e_{\boldsymbol{d}} = 8Pv_{\boldsymbol{c}}e_{\boldsymbol{c}} \le 8P(2P)^2 2^{2P} N^4 \lg(2N)\, 2^{-S+1} = 32P^3 2^{2P} N^4 \lg(2N)\, 2^{-S+1}$$

The inverse N-point FFT obeys the bounds of Lemma 6.2 followed by a scaling by $1/N$ of the value and error bounds. Thus after the inverse FFT, we have the following bounds.

$$\begin{aligned} v_{\boldsymbol{b}} &\le \frac{1}{N} N^2 v_{\boldsymbol{d}} \le (2P)^3 2^{2P} N^5 \\ e_{\boldsymbol{b}} &\le \frac{1}{N} N^2 (e_{\boldsymbol{d}} + v_{\boldsymbol{d}}\, 2^{-S+1} \lg N) \\ &= N(32P^3 2^{2P} N^4 \lg(2N)\, 2^{-S+1} + 8P^3 2^{2P} N^4 \lg N\, 2^{-S+1}) \\ &\le 40P^3 2^{2P} N^5 \lg(2N)\, 2^{-S+1} \end{aligned}$$

The final part of Inverse-Half-FFT consists of multiplications with twiddle factors. It results in a value bound of

$$2Pv_{\boldsymbol{b}} \le (2P)^4 2^{2P} N^5$$

and an error bound of

$$2P(e_{\boldsymbol{b}} + v_{\boldsymbol{b}} 2^{-S+1}) < 96P^4 2^{2P} N^5 \lg(2N)\, 2^{-S+1}$$

which is less than $1/2$ if

$$7 + 4\lg P + 5\lg N + 2P + \lg\lg(2N) - S + 1 \le -1$$

proving the claim. □

The bounds of the previous proof are summarized in Table 6.1. As an immediate

Position in Algorithm	Value bound	Absolute error bound
Start	2^P	0
After first level of Half-FFT	$2P\, 2^P$	$2P\, 2^P\, 2^{-S+1}$
After N-point FFT	$2P\, 2^P N^2$	$2P\, 2^P N^2 \lg(2N)\, 2^{-S+1}$
After multiplication of values	$(2P)^3\, 2^{2P} N^4$	$32P^3\, 2^{2P} N^4 \lg(2N)\, 2^{-S+1}$
After inverse FFT	$(2P)^3\, 2^{2P} N^5$	$40P^3\, 2^{2P} N^5 \lg(2N)\, 2^{-S+1}$
After last level of inverse Half-FFT	$(2P)^4\, 2^{2P} N^5$	$96P^4\, 2^{2P} N^5 \lg(2N)\, 2^{-S+1}$

TABLE 6.1
Bounds on absolute values and errors

implication of Lemma 6.4, we obtain the following result.

THEOREM 6.5. *For some $S = \Theta(\lg n)$, doing Half-FFT with precision S is sufficient for the Algorithm Modular-Integer-Multiplication.*

7. Complexity. Independently of how an N-point Fourier transform is recursively decomposed, the computation can always be visualized by the well known butterfly graph with $\lg N + 1$ rows. Every row represents N elements of the ring $\mathcal{R}$. Row 0 represents the input, row N represents the output, and every entry of row $j+1$ is obtained from row j ($0 \le j < N$) by an addition or subtraction and possibly a multiplication with a power of ω. When investigating the complexity of performing the multiplications in $\mathcal{R}$ recursively, it is best to still think in terms of the same

$\lg N + 1$ rows. At the next level of recursion, N multiplications are done per row. It is important to observe that the sum of the lengths of the representations of all entries in one row grows just by a constant factor from each level of recursion to the next. The blow-up by a constant factor is due to the padding with 0's, and due to the precision needed to represent numerical approximations of complex roots of unity. Padding with 0's occurs when reducing multiplication in $\mathcal{R}$ to modular integer multiplication and during the procedure Decompose.

We do $O(\lg^* n)$ levels of recursive calls to Modular-Integer-Multiplication. As the total length of a row grows by a constant factor from level to level, we obtain the factor $2^{O(\lg^* n)}$ in the running time. From a practical point of view, one should not worry too much about this factor. The function $\lg^* n$ in the exponent of the running time actually represents $\lg^* n - 4$ or $\lg^* n - 3$, which for all practical purposes could be thought as being 1 or 2, because at a low recursion level, one would switch to a more traditional multiplication method.

The crucial advantage of our new FFT algorithm is the fact that most multiplications with twiddle factors can be done in linear time, as each of them only involves a cyclic shift (with sign change on wrap around) of a vector of coefficients representing an element of $\mathcal{R}$. Indeed, only every $O(\log\log N)$th row of the FFT requires recursive calls for non-trivial multiplications with roots of unity. We recall that our Fourier transform is over the ring $\mathcal{R}$, whose elements are represented by polynomials of degree $P-1$ with coefficients of length $O(P) = O(\log N)$.

Based on these arguments, one obtains the following recurrence equations for the boolean circuit complexity $T(n)$ of Modular-Integer-Multiplication and $T'(N)$ of FFT.

$$
\begin{aligned}
T(n) &= O(T'(n/\log^2 n)) \\
T'(N) &= O\left(N \log^3 N + \frac{N \log N}{\log\log N} T(O(\log^2 N))\right)
\end{aligned}
$$

These recurrence equations have the following solutions.

$$
\begin{aligned}
T(n) &= n \log n \, 2^{O(\lg^* n)} \\
T'(N) &= N \log^3 N \, 2^{O(\lg^* N)}
\end{aligned}
$$

A reader convinced by these intuitive arguments may jump directly to Theorem 7.5. We are more formal here, producing the recurrence equations step by step based on the recursive structure of the algorithms.

First we count the number of additions $\text{Add}(N)$ and the number of multiplications $\text{Mult}(N)$ of the N-point FFT. The counts refer to operations in $\mathcal{R}$. As always, we assume J, K and N to be powers of 2 with $JK = N$.

$$
\text{Add}(N) = \begin{cases} 0 & \text{if } N = 1 \\ 2 & \text{if } N = 2 \\ K \,\text{Add}(J) + J \,\text{Add}(K) & \text{otherwise} \end{cases}
$$

The solution $\text{Add}(N) = N \lg N$ is immediate.

$$
\text{Mult}(N) = \begin{cases} 0 & \text{if } N \le 2 \\ K \,\text{Mult}(J) + J \,\text{Mult}(K) + KJ & \text{otherwise} \end{cases}
$$

Induction on N verifies the solution

$$\text{Mult}(N) = \begin{cases} 0 & \text{if } N = 1 \\ N(\lg N - 1) & \text{if } N \geq 2 \end{cases}$$

One should note that more than half of the multiplications counted by Mult(N) are actually multiplications by 1. For the sake of simplicity of the presentation, we did not do the corresponding obvious optimization (for $j = 0$ and $k = 0$) in the algorithm FFT.

More interesting than the total number of multiplications Mult(N), is the number of expensive multiplications EMult(N). Multiplications with (low order) $2P$th roots of unity are inexpensive, as they are done by cyclic shifts (with sign changes on wrap around). The recurrence equations for Mult and EMult differ in the start conditions.

$$\text{EMult}(N) = \begin{cases} 0 & \text{if } N \leq 2P \\ K\,\text{EMult}(J) + J\,\text{EMult}(K) + KJ & \text{otherwise} \end{cases}$$

Note that for $N > 2P$, the procedure Select chooses $J = 2P$ and $K = N/(2P)$, implying EMult(J) $= 0$ simplifying the recurrence equation.

$$\text{EMult}(N) = \begin{cases} 0 & \text{if } N \leq 2P \\ 2P\,\text{EMult}(N/(2P)) + N & \text{otherwise} \end{cases}$$

Lemma 7.1. *This recurrence equation has the sulution*

$$\textit{EMult}(N) = N(\lceil \log_{2P} N \rceil - 1) \leq N \frac{\lg N}{\lg(2P)}$$

Proof. Only the case $N > 2P$ is non-trivial.

$$\begin{aligned} \text{EMult}(N) &= 2P\text{EMult}(N/(2P)) + N \\ &= 2P\frac{N}{2P}(\lceil \log_{2P}(N/(2P)) \rceil - 1) + N \\ &= N(\lceil \log_{2P} N \rceil - 2) + N \\ &= N(\lceil \log_{2P} N \rceil - 1) \end{aligned}$$

□

Lemma 7.2. *The number of expensive multiplications is*
(a) $N(\lceil \log_{2P} N \rceil - 1)$ *for FFT*
(b) $N\lceil \log_{2P} N \rceil$ *for Half-FFT*
(c) $N(3\lceil \log_{2P} N \rceil + 1)$ *for Modular-Integer-Multiplication*

Proof. The number of expensive multiplications for FFT is $EMult(N)$. The other results immediately follow from the definitions of the algorithms. □

Let $T(n)$ be the boolean circuit complexity of Modular-Intege-Multiplication. $T(n) + O(n)$ is then also the circuit complexity of Integer Multiplication with $0 \leq$ product $\leq 2^n$.

Lemma 7.3. *Let n_0 and N be the positive integers from the algorithm Modular-Integer-Multiplication. $n_0 \geq 16$ is a constant, and $\frac{1}{2}n/\lg^2 n \leq N \leq 2n/\lg^2 n \leq n$ for all $n \geq n_0$. For some real constants $c, c' > 1$, $T(n)$ satisfies the following recurrence*

$$T(n) \leq N(3\lceil \log_{2P} N \rceil + 1) \cdot T(c \lg^2 N) + c' N \lg N \cdot \lg^2 N \quad \textit{for } n \geq n_0$$

Proof. This recurrence is based on the counts of additions, multiplications and expensive multiplications, and on the following facts. Binary integers are chopped into $N = O(n/\log^2 n)$ pieces, which are represented by elements of $\mathcal{R}$ encoded by strings of length $O(\log^2 N)$. In this encoding, additions, easy multiplications and all bookkeeping operations are done in linear time. Expensive multiplications in $\mathcal{R}$ are done recursively after encoding the elements of $\mathcal{R}$ as modular integers, which causes a constant factor blow-up . □

Now we can claim

$$T(n) \le n \lg n \, 2^{O(\lg^* n)}$$

but such a claim resists a direct induction proof, because $\lg^* n$ is not continuous. Even though there are only $O(\lg^* n)$ recursion levels, $\lg^* n$ does not decrease at each level due to the reduction from n to $O(\log^2 n)$ not $\lg n$. As a trick, we use the fact that $\lg^* \sqrt[4]{n}$ decreases at each level.

LEMMA 7.4.

$$T(n) \le n \lg n \, (2^{d \lg^* \sqrt[4]{n}} - d')$$

for some $d, d' > 0$ and all $n \ge 2$.

Proof. From the algorithm Modular-Integer-Multiplication, recall the definitions, $P = \mathrm{round}(\lg n)$ and $N = 2n/P^2$. The implications $N \le \min(n, 2n/\lg^2 n)$ for $n \ge 2$, and $\lceil \log_{2P} N \rceil < \log_{2P} n$ for $n \ge 16$ are used in Ineq. 7.3 below.

First we do the inductive step. d, d' and a constant n_0' will be determined later. Let $n \ge n_0' \ge n_0 \ge 16$. Assume the claim of the lemma holds for all n' with $2 \le n' < n$.

$$\begin{aligned}
T(n) &\le N(3\lceil \log_{2P} N \rceil + 1)\, T(c \lg^2 N) + c' N \lg^3 N && (7.1)\\
&\le 4N \lceil \log_{2P} N \rceil \, c \lg^2 N \, \lg(c \lg^2 N)\, (2^{d \lg^* \sqrt[4]{c \lg^2 N}} - d') + c' N \lg^3 N && (7.2)\\
&\le 8 \frac{n}{\lg^2 n} \log_{2P} n \; c \lg^2 n \; 2 \lg\lg n \, (2^{d \lg^* (\frac{1}{4} \lg n)} - d') + 2c' n \lg n && (7.3)\\
&= 16 c n \lg n \frac{\lg \lg n}{\lg 2P} \, (2^{d(\lg^* \sqrt[4]{n} - 1)} - d') + 2c' n \lg n && (7.4)\\
&\le n \lg n (2^{d \lg^* \sqrt[4]{n}} - d') && (7.5)
\end{aligned}$$

Ineq. 7.1 is the recurrence from Lemma 7.3. The inductive hypothesis is used in Ineq. 7.2. For Ineq. 7.3, we use the definitions of P and N, and we select n_0' sufficiently big such that $\sqrt[4]{c \lg^2 N} \le \frac{1}{4} \lg n$ for all $n \ge n_0'$. Finally, for Ineq. 7.5, we use $\lg \lg n \le \lg 2P \le \lg \lg n + 2 \le 2 \lg \lg n$, and we just choose d and d' sufficiently big such that $16c \le 2^d$ and $-8cd' + 2c' \le -d'$. Furthermore, we make sure d is big enough that the claim of the lemma holds for all n with $2 \le n < n_0'$. □

Lemma 7.4 implies our main results for circuit complexity and (except for the organizational details) for multitape Turing machines.

THEOREM 7.5. *Multiplication of binary integers of length n can be done by a boolean circuit of size $n \log n \, 2^{O(\lg^* n)}$.*

THEOREM 7.6. *Multiplication of binary integers of length n can be done in time $n \log n \, 2^{O(\lg^* n)}$ on a 2-tape Turing machine.*

A detailed proof of Theorem 7.6 would be quite tedious. Nevertheless, it should be obvious that due to the relatively simple structure of the algorithms, there is no principle problem to implement them on Turing machines.

As an important application of integer multiplication, we obtain corresponding bounds for the multiplication of polynomials by boolean circuits or Turing machines. We are looking at bit complexity, not assuming that products of coefficients can be obtained in one step.

COROLLARY 7.7. *Products of polynomials of degree less than n, with a $2^{O(m)}$ upper bound on the absolute values of their real or complex coefficients, can be approximated in time $mn \log mn\, 2^{O(\log^* mn)}$ with an absolute error bound of 2^{-m}, for a given $m = \Omega(\log n)$.*

Proof. Schönhage [Sch82] has shown how to reduce the multiplication of polynomials with complex coefficients to integer multiplication with only a constant factor in time increase. □

Indeed, multiplying polynomials with real or complex coefficients is a major area where long integer multiplication is very useful. Long integer multiplication is used extensively for finding large prime numbers. Another application is the computation of billions of digits of π to study patterns. A very practical application is the testing computational hardware.

8. Open Problem. Besides the obvious question whether integer multiplication is in $O(n \log n)$, a multiplication algorithm running in time $O(n \log n \lg^* n)$ would also be very desirable. It could be achieved, if one could avoid the constant factor cost increase from one recursion level to the next. Furthermore, it would be nice to have an implementation that compares favorably with current implementations of the algorithm of Schönhage and Strassen. The asymptotic improvement from $O(n \log n \log\log n)$ to $n \log n\, 2^{O(\log^* n)}$ might suggest that an actual speed-up only shows up for astronomically large numbers. Indeed, the expressions are not very helpful to judge the performance for reasonable values of n. But one should notice that $\lg^* n$ in the exponent really just represents an upper bound on the nesting of recursive calls to integer multiplication. For any practical purposes, one would nest these calls at most twofold.

REFERENCES

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[BCS97] Peter Bürgisser, Michael Clausen, and M. Amin Shokrollahi, *Algebraic complexity theory*, Springer-Verlag, Berlin, 1997.

[Ber] Daniel J. Bernstein, *Personal communication.*

[BL04] Peter Bürgisser and Martin Lotz, *Lower bounds on the bounded coefficient complexity of bilinear maps*, Journal of the ACM **51** (2004), no. 3, 464–482.

[CA69] S. A. Cook and S. O. Aanderaa, *On the minimum computation time of functions*, Transactions of the AMS **142** (1969), 291–314.

[Coo66] Stephen A. Cook, *On the minimum computation time of functions*, Ph.D. thesis, Harvard University, 1966.

[CT65] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Math. of Comput. **19** (1965), 297–301.

[DKSS08] Anindya De, Piyush Kurur, Chandan Saha, and Ramprasad Saptharishi, *Fast integer multiplication using modular arithmetic*, STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 2008, pp. 499–506.

[Für89] Martin Fürer, *On the complexity of integer multiplication, (extended abstract)*, Tech. Report Technical report CS-89-17, Department of Computer Science, The Pennsylvania State University, 1989.

[Für07] ———, *Faster integer multiplication*, STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 2007, pp. 57–66.

[HJB84] M. T. Heideman, D. H. Johnson, and C. S. Burrus, *Gauss and the history of the FFT*, IEEE Acoustics, Speech, and Signal Processing **1** (1984), 14–21.

[KO62] Anatolii Karatsuba and Yu Ofman, *Multiplication of multidigit numbers on automata*, Doklady Akademii Nauk SSSR **145** (1962), no. 2, 293–294, (in Russian). English translation in Soviet Physics-Doklady 7, 595-596,1963.

[Mor73] Jacques Morgenstern, *Note on a lower bound on the linear complexity of the fast Fourier transform*, Journal of the ACM **20** (1973), no. 2, 305–306.

[Pan86] Victor Ya. Pan, *The trade-off between the additive complexity and the asynchronicity of linear and bilinear algorithms*, Information Processing Letters **22** (1986), no. 1, 11–14.

[Pap79] Christos H. Papadimitriou, *Optimality of the fast Fourier transform*, Journal of the ACM **26** (1979), no. 1, 95–102.

[PFM74] Michael S. Paterson, Michael J. Fischer, and Albert R. Meyer, *An improved overlap argument for on-line multiplication*, Tech. Report 40, Project MAC, MIT, January 1974.

[Sch66] Arnold Schönhage, *Multiplikation großer Zahlen*, Computing **1** (1966), no. 3, 182–196 (German).

[Sch82] ———, *Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coeficients*, Computer Algebra, EUROCAM '82, European Computer Algebra Conference, Marseille, France, 5-7 April, 1982, Proceedings, Lecture Notes in Computer Science, vol. 144, Springer, 1982, pp. 3–15.

[SS71] Arnold Schönhage and Volker Strassen, *Schnelle Multiplikation grosser Zahlen*, Computing **7** (1971), 281–292.

[Too63] Andre L. Toom, *The complexity of a scheme of functional elements simulating the multiplication of integers*, Dokl. Akad. Nauk SSSR **150** (1963), 496–498, (in Russian). English translation in Soviet Mathematics 3, 714-716, 1963.