Fast Integer Multiplication Using Modular Arithmetic *

Anindya De†
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720, USA
anindya@eecs.berkeley.edu

Piyush P Kurur‡
Dept. of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, UP, India, 208016
ppk@cse.iitk.ac.in

Chandan Saha
Dept. of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, UP, India, 208016
csaha@cse.iitk.ac.in

Ramprasad Saptharishi§
Chennai Mathematical Institute
Plot H1, SIPCOT IT Park
Padur PO, Siruseri, India, 603103
ramprasad@cmi.ac.in

Abstract

We give an $N \cdot \log N \cdot 2^{O(\log^* N)}$ time algorithm to multiply two N-bit integers that uses modular arithmetic for intermediate computations instead of arithmetic over complex numbers as in Fürer's algorithm, which also has the same and so far the best known complexity. The previous best algorithm using modular arithmetic (by Schönhage and Strassen) has complexity $O(N \cdot \log N \cdot \log\log N)$. The advantage of using modular arithmetic as opposed to complex number arithmetic is that we can completely evade the task of bounding the truncation error due to finite approximations of complex numbers, which makes the analysis relatively simple. Our algorithm is based upon Fürer's algorithm, but uses FFT over multivariate polynomials along with an estimate of the least prime in an arithmetic progression to achieve this improvement in the modular setting. It can also be viewed as a p-adic version of Fürer's algorithm.

1 Introduction

Computing the product of two N-bit integers is nearly a ubiquitous operation in algorithm design. Being a basic arithmetic operation, it is no surprise that multiplications of integers occur as intermediate steps of computation in algorithms from every possible domain of computer science. But seldom do the complexity of such multiplications influence the overall efficiency of the algorithm as the integers involved are relatively small in size and the multiplications can often be implemented as fast hardware operations. However, with the advent of modern cryptosystems, the study of the bit complexity of integer multiplication received a significant impetus. Indeed, large integer multiplication forms the foundation of many modern day public-key crystosytems, like RSA, El-Gamal and Elliptic Curve crytosystems. One of the most notable applications is the RSA cryptosystem,

*A preliminary version appeared in the proceedings of the 40th ACM Symposium on Theory of Computing, 2008.
†Research done while the author was at the Dept of Computer Science and Engineering, IIT Kanpur
‡Research supported through Research I Foundation project NRNM/CS/20030163
§Research done while visiting IIT Kanpur under Project FLW/DST/CS/20060225

where it is required to multiply two primes that are hundreds or thousands of bits long. The larger these primes the harder it is to factor their product, which in turn makes the RSA extremely secure in practice.

In this paper, our focus is more on the theoretical aspects of integer multiplication, it being a fundamental problem in its own right. This is to say, we will be concerned with the asymptotic bit complexity of mutiplying two N-bit integers with little emphasis on optimality in practice. We begin with a brief account of earlier work on integer multiplication algorithms.

1.1 Previous Work

The naive approach to multiply two N-bit integers leads to an algorithm that uses $O(N^2)$ bit operations. Karatsuba [KO63] showed that some multiplication operations of such an algorithm can be replaced by less costly addition operations which reduces the overall running time of the algorithm to $O(N^{\log_2 3})$ bit operations. Shortly afterwards, this result was improved by Toom [Too63] who showed that for any $\varepsilon > 0$, integer multiplication can be done in $O(N^{1+\varepsilon})$ time. This led to the question as to whether the time complexity can be improved further by replacing the term $O(N^{\epsilon})$ by a poly-logarithmic factor. In a major breakthrough, Schönhage and Strassen [SS71] gave two efficient algorithms for multiplying integers using fast polynomial multiplication. One of the algorithms achieved a running time of $O(N \cdot \log N \cdot \log\log N \ldots 2^{O(\log^* N)})$ using arithmetic over complex numbers (approximated to suitable precisions), while the other used arithmetic modulo carefully chosen integers to improve the complexity further to $O(N \cdot \log N \cdot \log\log N)$ bit operations. The modular algorithm remained the best for a long period of time until a recent remarkable result by Fürer [Für07] (see also [Für09]). Fürer gave an algorithm that uses arithmetic over complex numbers and runs in $N \cdot \log N \cdot 2^{O(\log^* N)}$ time. Till date this is the best time complexity known for integer multiplication and indeed our result is inspired by Fürer's algorithm.

Further details on other approaches and enhancements to previous integer multiplication algorithms can be found in [Für09].

1.2 The Motivation

Schönhage and Strassen introduced two seemingly different approaches to integer multiplication – using complex and modular arithmetic. Fürer's algorithm improves the time complexity in the complex arithmetic setting by cleverly reducing some costly multiplications to simple shift operations. However, the algorithm needs to approximate the complex numbers to certain precisions during computation. This introduces the added task of bounding the total truncation errors in the analysis of the algorithm. On the contrary, in the modular setting the error analysis is virtually absent or rather more implicit, which in turn simplifies the overall analysis. In addition, modular arithmetic gives a discrete approach to a discrete problem like integer multiplication. Therefore, it seems natural to ask whether we can achieve a similar improvement in time complexity of this problem in the modular arithmetic setting. In this work, we answer this question affirmatively. We give an $N \cdot \log N \cdot 2^{O(\log^* N)}$ time algorithm for integer multiplication using only modular arithmetic, thus matching the improvement made by Fürer.

Overview of our result

As is the case in both Schönhage-Strassen's and Fürer's algorithms, we start by reducing the problem to polynomial multiplication over a ring $\mathcal{R}$ by properly encoding the given integers. Polynomials can be multiplied efficiently using Discrete Fourier Transforms (DFT). However, in order that we are able to use Fast Fourier Transform (FFT), the ring $\mathcal{R}$ should have some special roots of unity. For instance, to multiply two polynomials of degree less than M using FFT, we require a *principal* $2M$-th root of unity (see Definition 2.2 for principal roots). One way to construct such a ring in the modular setting is to consider rings of the form $\mathcal{R} = \mathbb{Z}/(2^M + 1)\mathbb{Z}$ as in Schönhage and Strassen's work [SS71]. In this case, the element 2 is a $2M$-th principal root of unity in $\mathcal{R}$. This approach can be equivalently viewed as attaching an 'artificial' root to the ring of integers. However, this makes the size of $\mathcal{R}$ equal to 2^M and thus a representation of an arbitrary element in $\mathcal{R}$ takes M bits. This means an N-bit integer is encoded as a polynomial of degree M with every coefficient about M bits long, thereby making $M \approx \sqrt{N}$ as the optimal choice. Indeed, the choice of such an $\mathcal{R}$ is the basis of Schönhage and Strassen's modular algorithm in which they reduce multiplication of N-bit integers to multiplication of $\sqrt{N}$-bit integers and achieve a complexity of $O(N \cdot \log N \cdot \log\log N)$ bit operations.

Naturally, such rings are a little too expensive in our setting. We would rather like to find a ring whose size is bounded by some polynomial in M and which still contains a principal $2M$-th root of unity. In fact, it is this task of choosing a suitable ring that poses the primary challenge in adapting Fürer's algorithm and making it work in the discrete setting.

We choose the ring to be $\mathcal{R} = \mathbb{Z}/p^c\mathbb{Z}$, for a prime p and a constant c such that $p^c = \mathsf{poly}(M)$. The ring $\mathbb{Z}/p^c\mathbb{Z}$, has a principal $2M$-th root of unity if and only if $2M$ divides $p-1$, which means that we need to find a prime p from the arithmetic progression $\{1 + i \cdot 2M\}_{i>0}$. To make this search computationally efficient, we also need the degree of the polynomials, M to be sufficiently small. This we can achieve by encoding the integers as multivariate polynomials instead of univariate ones. It turns out that the choice of the ring as $\mathcal{R} = \mathbb{Z}/p^c\mathbb{Z}$ is still not quite sufficient and needs a little more refinement. This is explained in Section 2.1.

The use of multivariate polynomial multiplications along with a small base ring are the main steps where our algorithm differs from earlier algorithms by Schönhage-Strassen and Fürer. Towards understanding the notion of *inner* and *outer* DFT in the context of multivariate polynomials, we also present a group theoretic interpretation of DFT. The use of inner and outer DFT plays a central role in both Fürer's as well as our algorithm. Arguing along the line of Fürer [Für07], we show that repeated use of efficient computation of inner DFT's using some special roots of unity in $\mathcal{R}$ reduces the number of 'bad multiplications' (in comparison to Schönhage-Strassen's algorithm) and makes the overall process efficient, thereby leading to an $N \cdot \log N \cdot 2^{O(\log^* N)}$ time algorithm.

2 The Basic Setup

2.1 The Underlying Ring

Rings of the form $\mathcal{R} = \mathbb{Z}/(2^M + 1)\mathbb{Z}$ have the nice property that multiplications by powers of 2, the $2M$-th principal root of unity, are mere shift operations and are therefore very efficient. Although by choosing the ring $\mathcal{R} = \mathbb{Z}/p^c\mathbb{Z}$ we ensure that the ring size is small, it comes with a price: multiplications by principal roots of unity are no longer just shift operations. Fortunately, this can be redeemed by working with rings of the form $\mathcal{R} = \mathbb{Z}[\alpha]/(p^c, \alpha^m + 1)$ for some m whose value will

be made precise later. Elements of $\mathcal{R}$ are thus $m-1$ degree polynomials over α with coefficients from $\mathbb{Z}/p^c\mathbb{Z}$. By construction, α is a $2m$-th root of unity and multiplication of any element in $\mathcal{R}$ by any power of α can be achieved by shift operations — this property is crucial in making some multiplications in the FFT less costly (see Section 3.2).

Given an N-bit number a, we encode it as a k-variate polynomial over $\mathcal{R}$ with degree in each variable less than M. The parameters M and m are powers of two such that M^k is roughly $\frac{N}{\log^2 N}$ and m is roughly $\log N$. The parameter k will ultimately be chosen a constant (see Section 4.2). We now explain the details of this encoding process.

2.2 Encoding Integers into k-variate Polynomials

Given an N-bit integer a, we first break these N bits into M^k blocks of roughly $\frac{N}{M^k}$ bits each. This corresponds to representing a in base $q = 2^{\frac{N}{M^k}}$. Let $a = a_0 + \ldots + a_{M^k-1}q^{M^k-1}$, where every $a_i < q$. The number a is converted into a polynomial as follows:

1. Express i in base M as $i = i_1 + i_2 M + \cdots + i_k M^{k-1}$.
2. Encode each term $a_i q^i$ as the monomial $a_i \cdot X_1^{i_1} X_2^{i_1} \cdots X_k^{i_k}$. As a result, the number a gets converted to the polynomial $\sum_{i=0}^{M^k-1} a_i \cdot X_1^{i_1} \cdots X_k^{i_k}$.

Further, we break each a_i into $\frac{m}{2}$ equal sized blocks where the number of bits in each block is $u = \frac{2N}{M^k \cdot m}$. Each coefficient a_i is then encoded as a polynomial in α of degree less than $\frac{m}{2}$. The polynomials are then padded with zeroes to stretch their degrees to m. Thus, the N-bit number a is converted to a k-variate polynomial $a(X)$ over $\mathbb{Z}[\alpha]/(\alpha^m + 1)$.

Given integers a and b, each of N bits, we encode them as polynomials $a(X)$ and $b(X)$ and compute the product polynomial. The product $a \cdot b$ can be recovered by substituting $X_s = q^{M^{s-1}}$, for $1 \leq s \leq k$, and $\alpha = 2^u$ in the polynomial $a(X) \cdot b(X)$. The coefficients in the product polynomial could be as large as $M^k \cdot m \cdot 2^{2u}$ and hence it is sufficient to do arithmetic modulo p^c where $p^c > 2M^k \cdot m \cdot 2^{2u}$. Our choice of the prime p ensures that c is in fact a constant (see Section 4.2). We summarize this discussion as a lemma.

Lemma 2.1. *Multiplication of two N-bit integers reduces to multiplication of two k-variate polynomials, with degree in each variable bounded by M, over the ring $\mathbb{Z}[\alpha]/(p^c, \alpha^m + 1)$ for a prime p satisfying $p^c > 2M^k \cdot m \cdot 2^{2u}$, where $u = \frac{2N}{M^k m}$. Furthermore, the reduction can be performed in $O(N)$ time.*

2.3 Choosing the Prime

The prime p should be chosen such that the ring $\mathbb{Z}/p^c\mathbb{Z}$ has a *principal* $2M$-th root of unity, which is required for polynomial multiplication using FFT. A principal root of unity is defined as follows.

Definition 2.2. (Principal root of unity) *An n-th root of unity $\zeta \in \mathcal{R}$ is said to be primitive if it generates a cyclic group of order n under multiplication. Furthermore, it is said to be principal if n is coprime to the characteristic of $\mathcal{R}$ and ζ satisfies $\sum_{i=0}^{n-1} \zeta^{ij} = 0$ for all $0 < j < n$.*

In $\mathbb{Z}/p^c\mathbb{Z}$, a $2M$-th root of unity is principal if and only if $2M \mid p-1$ (see also Section 5). As a result, we need to choose the prime p from the arithmetic progression $\{1 + i \cdot 2M\}_{i>0}$, which is

potentially the main bottleneck of our approach. We now explain how to circumvent this problem.

An upper bound for the least prime in an arithmetic progression is given by the following theorem by Linnik [Lin44]:

Theorem 2.3. (Linnik) *There exist absolute constants ℓ and L such that for any pair of coprime integers d and n, the least prime p such that $p \equiv d \bmod n$ is less than ℓn^L.*

Heath-Brown [HB92] showed that the *Linnik constant* $L \leq 5.5$ (a recent work by Xylouris [Xyl10] showed that $L \leq 5.2$). Recall that M is chosen such that M^k is $O\left(\frac{N}{\log^2 N}\right)$. If we choose $k = 1$, that is if we use univariate polynomials to encode integers, then the parameter $M = O\left(\frac{N}{\log^2 N}\right)$. Hence the least prime $p \equiv 1 \pmod{2M}$ could be as large as N^L. Since all known deterministic sieving procedures take at least N^L time this is clearly infeasible (for a randomized approach see Section 2.3). However, by choosing a larger k we can ensure that the least prime $p \equiv 1 \pmod{2M}$ is $O(N^\varepsilon)$ for some constant $\varepsilon < 1$. Since primality testing is in deterministic polynomial[1] time [AKS04], we can find the least prime $p \equiv 1 \pmod{2M}$ in $o(N)$ time.

Lemma 2.4. *If k is any integer greater than $L + 1$, then $M^L = O\left(N^{\frac{L}{L+1}}\right)$ and hence the least prime $p \equiv 1 \pmod{2M}$ can be found in $o(N)$ time.*

Choosing the Prime Randomly

To ensure that the search for a prime $p \equiv 1 \pmod{2M}$ does not affect the overall time complexity of the algorithm, we considered multivariate polynomials to restrict the value of M; an alternative is to use randomization.

Proposition 2.5. *Assuming ERH, a prime $p \equiv 1 \pmod{2M}$ can be computed by a randomized algorithm with expected running time $\tilde{O}(\log^3 M)$.*

Proof. Titchmarsh [Tit30] (see also Tianxin [Tia90]) showed, assuming ERH, that the number of primes less than x in the arithmetic progression $\{1 + i \cdot 2M\}_{i>0}$ is given by,

$$\pi(x, 2M) = \frac{Li(x)}{\varphi(2M)} + O(\sqrt{x} \log x)$$

for $2M \leq \sqrt{x} \cdot (\log x)^{-2}$, where $Li(x) = \Theta(\frac{x}{\log x})$ and φ is the Euler totient function. In our case, since M is a power of two, $\varphi(2M) = M$, and hence for $x \geq 4M^2 \cdot \log^6 M$, we have $\pi(x, 2M) = \Omega\left(\frac{x}{M \log x}\right)$. Therefore, for an i chosen uniformly randomly in the range $1 \leq i \leq 2M \cdot \log^6 M$, the probability that $i \cdot 2M + 1$ is a prime is at least $\frac{d}{\log x}$ for a constant d. Furthermore, primality test of an $O(\log M)$ bit number can be done in $\tilde{O}(\log^2 M)$ time using Rabin-Miller primality test [Mil76, Rab80]. Hence, with $x = 4M^2 \cdot \log^6 M$, a suitable prime for our algorithm can be found in expected $\tilde{O}(\log^3 M)$ time. □

Remark - We prefer to use Linnik's theorem (Theorem 2.3) instead of Proposition 2.5 while choosing the prime in the progression $\{1 + i \cdot 2M\}_{i>0}$ so as to make the results in this paper independent of the ERH and the usage of random bits.

[1] a subexponential algorithm would suffice

2.4 Finding the Root of Unity

We require a principal $2M$-th root of unity $\rho(\alpha)$ in $\mathcal{R}$ to compute the Fourier transforms. This root $\rho(\alpha)$ should also have the property that its $(\frac{M}{m})$-th power is α, so as to make some multiplications in the FFT efficient (see Section 3.2). The root $\rho(\alpha)$ can be computed by interpolation in a way similar to that in Fürer's algorithm [Für07, Section 3], except that we need a principal $2M$-th root of unity ω in $\mathbb{Z}/p^c\mathbb{Z}$ to start with.

To obtain such a root, we first obtain a $2M$-th root of unity ω_1 in $\mathbb{Z}/p\mathbb{Z}$. A generator ζ of $\mathbb{F}_p^\times$ can be computed by brute force, as p is sufficiently small, and $\omega_1 = \zeta^{(p-1)/2M}$ is a principal $2M$-th root of unity in $\mathbb{Z}/p\mathbb{Z}$. A principal $2M$-root of unity ω_1 must be a root of the polynomial $f(x) = x^M + 1$ in $\mathbb{Z}/p\mathbb{Z}$. Having obtained ω_1, we use Hensel Lifting [NZM91, Theorem 2.23].

Lemma 2.6 (Hensel Lifting)**.** *Let ω_s be a root of $f(x) = x^M + 1$ in $\mathbb{Z}/p^s\mathbb{Z}$. Then there exists a unique root ω_{s+1} in $\mathbb{Z}/p^{s+1}\mathbb{Z}$ such that $\omega_{s+1} \equiv \omega_s \pmod{p^s}$ and $f(\omega_{s+1}) = 0 \pmod{p^{s+1}}$. This unique root is given by $\omega_{s+1} = \omega_s - \frac{f(\omega_s)}{f'(\omega_s)}$.*

It is clear from the above lemma that we can compute a $2M$-th root of unity $\omega = \omega_c$ in $\mathbb{Z}/p^c\mathbb{Z}$. We will need the following well-known and useful fact about principal roots of unity in any ring.

Lemma 2.7. *[Für09, Lemma 2.1] If M is a power of 2 and $\omega^M = -1$ in an arbitrary ring $\mathcal{R}$. If M is relatively prime to the characteristic of $\mathcal{R}$, then ω is a principal $2M$-th root of unity in $\mathcal{R}$.*

Hence it follows that the root ω of $f(x) = x^M + 1$ in $\mathbb{Z}/p^c\mathbb{Z}$ is a principal $2M$-th root of unity in $\mathbb{Z}/p^c\mathbb{Z}$. Furthermore, $\zeta^{(p-1)/2M} = \omega_1 \equiv \omega \bmod p$. Since ζ is a generator of $\mathbb{F}_p^\times$, different powers of ζ must generate the group $\mathbb{F}_p^\times$ and hence must be distinct modulo p. Hence it follows that different powers of ω must be distinct modulo p as well. Therefore, the difference between any two of them is a unit in $\mathbb{Z}/p^c\mathbb{Z}$ and this makes the following interpolation feasible in our setting.

Finding $\rho(\alpha)$ from ω: Since ω is a principal $2M$-th root of unity, $\gamma = \omega^{\frac{2M}{2m}}$ is a principal $2m$-th root of unity in $\mathbb{Z}/p^c\mathbb{Z}$. Notice that, $\alpha^m + 1$ uniquely factorizes as, $\alpha^m + 1 = (\alpha - \gamma)(\alpha - \gamma^3)\dots(\alpha - \gamma^{2m-1})$. The ideals generated by $(\alpha - \gamma^i)$ and $(\alpha - \gamma^j)$ are mutually coprime as $\gamma^i - \gamma^j$ is a unit for $i \neq j$ and is contained in the ideal generated by $(\alpha - \gamma^i)$ and $(\alpha - \gamma^j)$. Therefore, using Chinese Remaindering, α has the direct sum representation $(\gamma, \gamma^3, \dots, \gamma^{2m-1})$ in $\mathcal{R}$. Since we require $\rho(\alpha)^{\frac{2M}{2m}} = \alpha$, it is sufficient to choose a $\rho(\alpha)$ whose direct sum representation is $(\omega, \omega^3, \dots, \omega^{2m-1})$. Now use Lagrange's formula to interpolate $\rho(\alpha)$ as,

$$\rho(\alpha) = \sum_{i=1,\ i \text{ odd}}^{2m-1} \omega^i \cdot \prod_{j=1,\ j\neq i,\ j \text{ odd}}^{2m-1} \frac{\alpha - \gamma^j}{\gamma^i - \gamma^j}$$

The inverses of the elements $\gamma^i - \gamma^j$ in $\mathbb{Z}/p^c\mathbb{Z}$ can be easily computed in $\mathrm{poly}(\log p)$ time. It is also clear that $\rho(\alpha)^M$, in the direct-sum representation, is $(\omega^M, \omega^{3M}, \cdots, \omega^{(2m-1)M})$ which is the element -1 in $\mathcal{R}$. Hence $\rho(\alpha)$ is a principal $2M$-th root of unity (by Lemma 2.7).

Besides finding a generator ζ of $\mathbb{Z}/p\mathbb{Z}$ by brute-force (which can be performed in $O(p)$ time), all other computations can be done in $\mathrm{poly}(\log p)$ time. We summarize this as a lemma.

Lemma 2.8. *A principal $2M$-th root of unity $\rho(\alpha) \in \mathcal{R}$ such that $\rho(\alpha)^{2M/2m} = \alpha$ can be computed in deterministic time $O(p \cdot \mathrm{poly}(\log(p)))$ (which is $o(N)$ if $p = o(N)$).*

3 Fourier Transform

3.1 Inner and Outer DFT

Suppose that $a(x) \in \mathcal{S}[x]$ is a polynomial of degree less than $2M$, where $\mathcal{S}$ is a ring containing a $2M$-th principal root of unity ρ. Let us say that we want to compute the $2M$-point DFT of $a(x)$ using ρ as the root of unity. In other words, we want to compute the elements $a(1), a(\rho), \ldots, a(\rho^{2M-1})$ is $\mathcal{S}$. This can be done in two steps.

Step 1: Compute the following polynomials using $\alpha = \rho^{2M/2m}$.

$$\begin{aligned} a_0(x) &= a(x) \mod (x^{2M/2m} - 1) \\ a_1(x) &= a(x) \mod (x^{2M/2m} - \alpha) \\ &\vdots \\ a_{2m-1}(x) &= a(x) \mod (x^{2M/2m} - \alpha^{2m-1}), \end{aligned}$$

where $\deg(a_j(x)) < \frac{2M}{2m}$ for all $0 \leq j < 2m$.

Step 2: Note that, $a_j(\rho^{k \cdot 2m+j}) = a(\rho^{k \cdot 2m+j})$ for every $0 \leq j < 2m$ and $0 \leq k < \frac{2M}{2m}$. Therefore, all we need to do to compute the DFT of $a(x)$ is to evaluate the polynomials $a_j(x)$ at appropriate powers of ρ.

The idea is to show that both Step 1 and Step 2 can be performed by computation of some 'smaller' DFTs. Let us see how.

Performing Step 1: The crucial observation here is the following. Fix an integer ℓ in the range $[0, \frac{2M}{2m} - 1]$. Then the ℓ^{th} coefficients of $a_0(x), a_1(x), \ldots, a_{2m-1}(x)$ are exactly $e_\ell(1), e_\ell(\alpha), \ldots, e_\ell(\alpha^{2m-1})$, respectively, where $e_\ell(y)$ is the polynomial,

$$e_\ell(y) = \sum_{j=0}^{2m-1} a_{j \cdot \frac{2M}{2m} + \ell} \cdot y^j.$$

But then, finding $e_\ell(1), e_\ell(\alpha), \ldots, e_\ell(\alpha^{2m-1})$ is essentially computing the $2m$-point DFT of $e_\ell(y)$ using α as the $2m^{th}$ root of unity. Therefore, all we need to do to find $a_0(x), \ldots, a_{2m-1}(x)$ is to compute the DFTs of $e_\ell(y)$ for all $0 \leq \ell < \frac{2M}{2m}$. These $\frac{2M}{2m}$ many $2m$-point DFTs are called the *inner* DFTs.

Performing Step 2: In order to find $a_j(\rho^{k \cdot 2m+j})$, for $0 \leq k < \frac{2M}{2m}$ and a fixed j, we first compute the polynomial $\tilde{a}_j(x) = a_j(x \cdot \rho^j)$ followed by a $\frac{2M}{2m}$-point DFT of $\tilde{a}_j(x)$ using ρ^{2m} as the root of unity. These $2m$ many $\frac{2M}{2m}$-point DFTs (j running from 0 to $2m-1$) are called the *outer* DFTs. The polynomials $\tilde{a}_j(x)$ can be computed by multiplying the coefficients of $a_j(x)$ by suitable powers of ρ. Such multiplications are termed as *bad* multiplications (as they would result in recursive calls to integer multiplication).

The above discussion is summarized in the following lemma.

Lemma 3.1. (DFT time = Inner DFTs + Bad multiplications + Outer DFTs)
Time taken to compute a $2M$-point DFT over $\mathcal{S}$ is sum of:

1. *Time taken to compute $\frac{2M}{2m}$ many $2m$-point inner DFTs over $\mathcal{S}$ using α as the $2m$-th root of unity.*
2. *Time to do $2M$ multiplications in $\mathcal{S}$ by powers of ρ (bad multiplications).*
3. *Time taken to compute $2m$ many $\frac{2M}{2m}$-point outer DFTs over $\mathcal{S}$ using ρ^{2m} as the $\frac{2M}{2m}$-th root of unity.*

3.2 Analysis of the FFT

We are now ready to analyse the complexity of multiplying the two k-variate polynomials $a(X)$ and $b(X)$ (see Section 2.2) using Fast Fourier Transform. Treat $a(X)$ and $b(X)$ as univariate polynomials in variable X_k over the ring $\mathcal{S} = \mathcal{R}[X_1, \ldots, X_{k-1}]$. We write $a(X)$ and $b(X)$ as $a(X_k)$ and $b(X_k)$, respectively, where $\deg(a(X_k))$ and $\deg(b(X_k))$ are less than M. Multiplication of $a(X)$ and $b(X)$ can be thought of as multiplication of the univariates $a(X_k)$ and $b(X_k)$ over $\mathcal{S}$. Also note that, the root $\rho(\alpha)$ (constructed in Section 2.4) is a primitive $2M$-th root of unity in $\mathcal{S} \supset \mathcal{R}$. Denote the multiplication complexity of $a(X_k)$ and $b(X_k)$ by $\mathcal{F}(2M, k)$.

Multiplication of $a(X_k)$ and $b(X_k)$ using FFT involves computation of three $2M$-point DFTs over $\mathcal{S}$ and $2M$ pointwise (or componentwise) multiplications in $\mathcal{S}$. Let $\mathcal{D}(2M, k)$ be the time taken to compute a $2M$-point DFT over $\mathcal{S}$. By Lemma 3.1, the time to compute a DFT is the sum of the time for the inner DFTs, the bad multiplications and the outer DFTs. Let us analyse these three terms separately. We will go by the notation in Section 3.1, using $\mathcal{S} = \mathcal{R}[X_1, \ldots, X_{k-1}]$ and $\rho = \rho(\alpha)$.

Inner DFT time: Computing a $2m$-point DFT requires $2m \log(2m)$ additions in $\mathcal{S}$ and $m \log(2m)$ multiplications by powers of α. The important observation here is: since $\mathcal{R} = \mathbb{Z}[\alpha]/(p^c, \alpha^m + 1)$, multiplication by a power of α with an element in $\mathcal{R}$ can be readily computed by simple cyclic shifts (with possible negations), which takes only $O(m \cdot \log p)$ bit operations. An element in $\mathcal{S}$ is just a polynomial over $\mathcal{R}$ in variables $X_1, \ldots, X_{k-1}$, with degree in each variable bounded by M. Hence, multiplication by a power of α with an element of $\mathcal{S}$ can be done using $\mathcal{N}_{\mathcal{S}} = O(M^{k-1} \cdot m \cdot \log p)$ bit operations. A total of $m \log(2m)$ multiplications takes $O(m \log m \cdot \mathcal{N}_{\mathcal{S}})$ bit operations. It is easy to see that $2m \log(2m)$ additions in $\mathcal{S}$ also require the same order of time.

Since there are $\frac{2M}{2m}$ many $2m$-point DFTs, the total time spent in the inner DFTs is $O(2M \cdot \log m \cdot \mathcal{N}_{\mathcal{S}})$ bit operations.

Bad multiplication time: Suppose that two arbitrary elements in $\mathcal{R}$ can be multiplied using $\mathcal{M}_{\mathcal{R}}$ bit operations. Mulitplication in $\mathcal{S}$ by a power of ρ amounts to $c_{\mathcal{S}} = M^{k-1}$ multiplications in $\mathcal{R}$. Since there are $2M$ such bad multiplications, the total time is bounded by $O(2M \cdot c_{\mathcal{S}} \cdot \mathcal{M}_{\mathcal{R}})$.

Outer DFT time: By Lemma 3.1, the total outer DFT time is $2m \cdot \mathcal{D}\left(\frac{2M}{2m}, k\right)$.

Total DFT time: Therefore, the net DFT time is bounded as,

$$\begin{aligned}\mathcal{D}(2M,k) &= O\left(2M\cdot\log m\cdot\mathcal{N}_{\mathcal{S}}+2M\cdot c_{\mathcal{S}}\cdot\mathcal{M}_{\mathcal{R}}\right)+2m\cdot\mathcal{D}\left(\frac{2M}{2m},k\right)\\ &= O\left(2M\cdot\log m\cdot\mathcal{N}_{\mathcal{S}}+2M\cdot c_{\mathcal{S}}\cdot\mathcal{M}_{\mathcal{R}}\right)\cdot\frac{\log 2M}{\log 2m}\\ &= O\left(M^k\log M\cdot m\log p+\frac{M^k\log M}{\log m}\cdot\mathcal{M}_{\mathcal{R}}\right),\end{aligned}$$

putting the values of $\mathcal{N}_{\mathcal{S}}$ and $c_{\mathcal{S}}$.

Pointwise multiplications: Finally, FFT does $2M$ pointwise multiplications in $\mathcal{S}$. Since elements of $\mathcal{S}$ are $(k-1)$-variate polynomials over $\mathcal{R}$, with degree in every variable bounded by M, the total time taken for pointwise multiplications is $2M\cdot\mathcal{F}(2M,k-1)$ bit operations.

Total polynomial multiplication time: This can be expressed as,

$$\begin{aligned}\mathcal{F}(2M,k) &= O\left(M^k\log M\cdot m\log p+\frac{M^k\log M}{\log m}\cdot\mathcal{M}_{\mathcal{R}}\right)+2M\cdot\mathcal{F}(2M,k-1)\\ &= O\left(M^k\log M\cdot m\log p+\frac{M^k\log M}{\log m}\cdot\mathcal{M}_{\mathcal{R}}\right),\end{aligned}\tag{1}$$

as k is a constant.

We now present an equivalent group theoretic interpretation of the above process of polynomial multiplication, which is a subject of interest in itself.

3.3 A Group Theoretic Interpretation

A convenient way to study polynomial multiplication is to interpret it as multiplication in a *group algebra*.

Definition 3.2. (Group Algebra) *Let G be any group. The* group algebra *of G over a ring R is the set of formal sums $\sum_{g\in G}\alpha_g g$ where $\alpha_g\in R$ with addition defined point-wise and multiplication defined via convolution as follows*

$$\left(\sum_g\alpha_g g\right)\left(\sum_h\beta_h h\right)=\sum_u\left(\sum_{gh=u}\alpha_g\beta_h\right)u$$

In this section, we study the Fourier transform over the group algebra $R[E]$ where E is an *additive abelian group*. Most of this, albeit in a different form, is well known but is provided here for completeness [Sha99, Chapter 17].

In order to simplify our presentation, we will fix the base ring to be $\mathbb{C}$, the field of complex numbers. Let n be the *exponent* of E, that is the maximum order of any element in E. A similar approach can be followed for any other base ring as long as it has a principal n-th root of unity.

We consider $\mathbb{C}[E]$ as a vector space with basis $\{x\}_{x\in E}$ and use the Dirac notation to represent elements of $\mathbb{C}[E]$ — the vector $|x\rangle$, x in E, denotes the element $1.x$ of $\mathbb{C}[E]$.

Multiplying univariate polynomials over R of degree less than n can be seen as multiplication in the group algebra $R[G]$ where G is the cyclic group of order $2n$. Say $a(x) = a_0 + a_1x + \cdots + a_dx^d$ and $b(x) = b_0 + b_1x \cdots + b_dx^d$ (with $d < n$) are the polynomials we wish to multiply, they can be embedded in $\mathbb{C}[\mathbb{Z}/2n\mathbb{Z}]$ as $|a\rangle = \sum_{i=0}^{d} a_i |i\rangle$ and $|b\rangle = \sum_{i=0}^{d} b_i |i\rangle$. It is trivial to see that their product in the group algebra is the embedding of the product of the polynomials. Similarly, multiplying k-variate polynomials of degree less than n in each variable can be seen as multiplying in the group algebra $R[G^k]$, where G^k denotes the k-fold product group $G \times \ldots \times G$.

Definition 3.3. (Characters) *Let E be an additive abelian group. A* character *of E is a homomorphism from E to $\mathbb{C}^*$.*

An example of a character of E is the trivial character, which we will denote by 1, that assigns to every element of E the complex number 1. If χ_1 and χ_2 are two characters of E then their product $\chi_1.\chi_2$ is defined as $\chi_1.\chi_2(x) = \chi_1(x)\chi_2(x)$.

Proposition 3.4. *[Sha99, Chapter 17, Theorem 1] Let E be an additive abelian group of exponent n. Then the values taken by any character of E are n-th roots of unity. Furthermore, the characters form a* multiplicative abelian group *$\hat{E}$ which is isomorphic to E.*

An important property that the characters satisfy is the following [Isa94, Corollary 2.14].

Proposition 3.5. (Schur's Orthogonality) *Let E be an additive abelian group. Then*

$$\sum_{x\in E} \chi(x) = \begin{cases} 0 & \text{if } \chi \neq 1, \\ \#E & \text{otherwise} \end{cases} \quad \text{and} \quad \sum_{\chi\in\hat{E}} \chi(x) = \begin{cases} 0 & \text{if } x \neq 0, \\ \#E & \text{otherwise.} \end{cases}$$

It follows from Schur's orthogonality that the collection of vectors $|\chi\rangle = \sum_x \chi(x) |x\rangle$ forms a basis of $\mathbb{C}[E]$. We will call this basis the *Fourier basis* of $\mathbb{C}[E]$.

Definition 3.6. (Fourier Transform) *Let E be an additive abelian group and let $x \mapsto \chi_x$ be an isomorphism between E and $\hat{E}$. The* Fourier transform *over E is the linear map from $\mathbb{C}[E]$ to $\mathbb{C}[E]$ that sends $|x\rangle$ to $|\chi_x\rangle$.*

Thus, the Fourier transform is a change of basis from the point basis $\{|x\rangle\}_{x\in E}$ to the Fourier basis $\{|\chi_x\rangle\}_{x\in E}$. The Fourier transform is unique only up to the choice of the isomorphism $x \mapsto \chi_x$. This isomorphism is determined by the choice of the principal root of unity.

It is a standard fact in representation theory that any character χ of an abelian group satisfies $\chi_y(x) = \chi_x(y)$ for every x, y. Using this and Proposition 3.5, it is easy to see that this transform can be inverted by the map $|x\rangle \mapsto \frac{1}{n} |\overline{\chi_x}\rangle$. Hence the *Inverse Fourier Transform* is essentially just a Fourier transform using $x \mapsto \overline{\chi_x}$ as the isomorphism between E and $\hat{E}$.

Remark 3.7. Given an element $|f\rangle \in \mathbb{C}[E]$, to compute its Fourier transform (or Inverse Fourier transform) it is sufficient to compute the *Fourier coefficients* $\{\langle\chi|f\rangle\}_{\chi\in\hat{E}}$.

Fast Fourier Transform

We now describe the Fast Fourier Transform for general abelian groups in the character theoretic setting. For the rest of the section fix an additive abelian group E over which we would like to compute the Fourier transform. Let A be any subgroup of E and let $B = E/A$. For any such pair of abelian groups A and B, we have an appropriate Fast Fourier transformation, which we describe in the rest of the section.

Proposition 3.8. *1. Every character λ of B can be "lifted" to a character of E (which will be denoted by $\tilde{\lambda}$ defined as follows $\tilde{\lambda}(x) = \lambda(x + A)$.*

2. Let χ_1 and χ_2 be two characters of E that when restricted to A are identical. Then $\chi_1 = \chi_2\tilde{\lambda}$ for some character λ of B.

3. The group $\hat{B}$ is (isomorphic to) a subgroup of $\hat{E}$ with the quotient group $\hat{E}/\hat{B}$ being (isomorphic to) $\hat{A}$.

Proof. It is very easy to check that $\tilde{\lambda}(x) = \lambda(x + A)$ is indeed a homomorphism from E to $\mathbb{C}$. This therefore establishes that $\hat{B}$ is a subgroup of $\hat{E}$.

As for the second, define the map $\tilde{\lambda}(x) = \frac{\chi_1(x)}{\chi_2(x)}$. It is easy to check that this is a homomorphism from E to $\mathbb{C}$. Then, for $x \in E$ and $a \in A$

$$\tilde{\lambda}(x + a) = \frac{\chi_1(x)\chi_1(a)}{\chi_2(x)\chi_2(a)} = \frac{\chi_1(x)}{\chi_2(x)} = \tilde{\lambda}(x)$$

And hence $\tilde{\lambda}$ is equal over cosets over A in E and hence χ is indeed a homomorphism from B to $\mathbb{C}$.

The third part follows from Proposition 3.4 and the fact that any quotient group of a finite abelian group is isomorphic to a subgroup. □

We now consider the task of computing the Fourier transform of an element $|f\rangle = \sum f_x |x\rangle$ presented as a list of coefficients $\{f_x\}$ in the point basis. For this, it is sufficient to compute the Fourier coefficients $\{\langle\chi|f\rangle\}$ for each character χ of E (Remark 3.7). To describe the Fast Fourier transform we fix two sets of cosets representatives, one of A in E and one of $\hat{B}$ in $\hat{E}$ as follows.

1. For each $b \in B$, b being a coset of A, fix a coset representative $x_b \in E$ such $b = x_b + A$.
2. For each character φ of A, fix a character χ_φ of E such that χ_φ restricted to A is the character φ. The characters $\{\chi_\varphi\}$ form (can be thought of as) a set of coset representatives of $\hat{B}$ in $\hat{E}$.

Since $\{x_b\}_{b\in B}$ forms a set of coset representatives, any $|f\rangle \in \mathbb{C}[E]$ can be written uniquely as $|f\rangle = \sum f_{b,a} |x_b + a\rangle$.

Proposition 3.9. *Let $|f\rangle = \sum f_{b,a} |x_b + a\rangle$ be an element of $\mathbb{C}[E]$. For each $b \in B$ and $\varphi \in \hat{A}$ let $|f_b\rangle \in \mathbb{C}[A]$ and $|f_\varphi\rangle \in \mathbb{C}[B]$ be defined as follows.*

$$\begin{aligned} |f_b\rangle &= \sum_{a\in A} f_{b,a} |a\rangle \\ |f_\varphi\rangle &= \sum_{b\in B} \overline{\chi}_\varphi(x_b)\langle\varphi|f_b\rangle |b\rangle \end{aligned}$$

Then for any character $\chi = \chi_\varphi\tilde{\lambda}$ of E the Fourier coefficient $\langle\chi|f\rangle = \langle\lambda|f_\varphi\rangle$.

Proof.

$$\langle \chi | f \rangle \quad = \quad \sum_{b \in B, a \in A} \overline{\chi_\varphi \tilde{\lambda}(x_b + a)} \cdot f_{b,a}$$

Recall that for any $\tilde{\lambda}$, that is a lift of a character λ of B, acts identically inside cosets of A and hence $\tilde{\lambda}(x_b + a) = \lambda(b)$. Therefore, the above sum can be rewritten as follows:

$$\begin{aligned} \sum_{b \in B, a \in A} \overline{\chi_\varphi \tilde{\lambda}(x_b + a)} \cdot f_{b,a} &= \sum_{b \in B} \sum_{a \in A} \overline{\chi_\varphi(x_b + a)\lambda(b)} \cdot f_{b,a} \\ &= \sum_{b \in B} \overline{\lambda(b)} \cdot \overline{\chi_\varphi(x_b)} \sum_{a \in A} \overline{\varphi(a)} f_{b,a} \end{aligned}$$

The inner sum over a is precisely $\langle \varphi | f_b \rangle$ and therefore we have:

$$\langle \chi | f \rangle \quad = \quad \sum_{b \in B} \overline{\lambda(x_b)} \cdot \overline{\chi_\varphi(x_b)} \langle \varphi | f_b \rangle$$

which can be rewritten as $\langle \lambda | f_\varphi \rangle$ as claimed. □

We are now ready to describe the Fast Fourier transform given an element $|f\rangle = \sum f_x |x\rangle$.

1. For each $b \in B$ compute the Fourier transforms of $|f_b\rangle$. This requires $\#B$ many Fourier transforms over A.
2. As a result of the previous step we have for each $b \in B$ and $\varphi \in \hat{A}$ the Fourier coefficients $\langle \varphi | f_b \rangle$. Compute for each φ the vectors $|f_\varphi\rangle = \sum_{b \in B} \overline{\chi}_\varphi(x_b) \langle \varphi | f_b \rangle |b\rangle$. This requires $\#\hat{A}.\#B = \#E$ many multiplications by roots of unity.
3. For each $\varphi \in \hat{A}$ compute the Fourier transform of $|f_\varphi\rangle$. This requires $\#\hat{A} = \#A$ many Fourier transforms over B.
4. Any character χ of E is of the form $\chi_\varphi \lambda$ for some $\varphi \in \hat{A}$ and $\lambda \in \hat{B}$. Using Proposition 3.9 we have at the end of Step 3 all the Fourier coefficients $\langle \chi | f \rangle = \langle \lambda | f_\varphi \rangle$.

If the quotient group B itself has a subgroup that is isomorphic to A then we can apply this process recursively on B to obtain a divide and conquer procedure to compute Fourier transform. In the standard FFT we use $E = \mathbb{Z}/2^n\mathbb{Z}$. The subgroup A is $2^{n-1}E$ which is isomorphic to $\mathbb{Z}/2\mathbb{Z}$ and the quotient group B is $\mathbb{Z}/2^{n-1}\mathbb{Z}$.

Analysis of the Fourier Transform

Our goal is to multiply k-variate polynomials over $\mathcal{R}$, with the degree in each variable less than M. This can be achieved by embedding the polynomials into the algebra of the product group $E = \left(\frac{\mathbb{Z}}{2M \cdot \mathbb{Z}}\right)^k$ and multiplying them as elements of the algebra. Since the exponent of E is $2M$, we require a principal $2M$-th root of unity in the ring $\mathcal{R}$. We shall use the root $\rho(\alpha)$ (as defined in Section 2.4) for the Fourier transform over E.

For every subgroup A of E, we have a corresponding FFT. We choose the subgroup A as $\left(\frac{\mathbb{Z}}{2m \cdot \mathbb{Z}}\right)^k$ and let B be the quotient group E/A. The group A has exponent $2m$ and α is a principal $2m$-th

root of unity. Since α is a power of $\rho(\alpha)$, we can use it for the Fourier transform over A. As multiplications by powers of α are just shifts, this makes Fourier transform over A efficient.

Let $\mathcal{F}(M,k)$ denote the complexity of computing the Fourier transform over $\left(\frac{\mathbb{Z}}{2M\cdot\mathbb{Z}}\right)^k$. We have

$$\mathcal{F}(M,k) = \left(\frac{M}{m}\right)^k \mathcal{F}(m,k) + (2M)^k \mathcal{M}_{\mathcal{R}} + (2m)^k \mathcal{F}\left(\frac{M}{2m}, k\right) \tag{2}$$

where $\mathcal{M}_{\mathcal{R}}$ denotes the complexity of multiplications in $\mathcal{R}$. The first term comes from the $\#B$ many Fourier transforms over A (Step 1 of FFT), the second term corresponds to the multiplications by roots of unity (Step 2) and the last term comes from the $\#A$ many Fourier transforms over B (Step 3).

Since A is a subgroup of B as well, Fourier transforms over B can be recursively computed in a similar way, with B playing the role of E. Therefore, by simplifying the recurrence in Equation 2 we get:

$$\mathcal{F}(M,k) = O\left(\frac{M^k \log M}{m^k \log m}\mathcal{F}(m,k) + \frac{M^k \log M}{\log m}\mathcal{M}_{\mathcal{R}}\right) \tag{3}$$

Lemma 3.10. $\mathcal{F}(m,k) = O(m^{k+1} \log m \cdot \log p)$

Proof. The FFT over a group of size n is usually done by taking 2-point FFT's followed by $\frac{n}{2}$-point FFT's. This involves $O(n \log n)$ multiplications by roots of unity and additions in base ring. Using this method, Fourier transforms over A can be computed with $O(m^k \log m)$ multiplications and additions in $\mathcal{R}$. Since each multiplication is between an element of $\mathcal{R}$ and a power of α, this can be efficiently achieved through shifting operations. This is dominated by the addition operation, which takes $O(m \log p)$ time, since this involves adding m coefficients from $\mathbb{Z}/p^c\mathbb{Z}$. □

Therefore, from Equation 3,

$$\mathcal{F}(M,k) = O\left(M^k \log M \cdot m \cdot \log p + \frac{M^k \log M}{\log m}\mathcal{M}_{\mathcal{R}}\right).$$

4 Algorithm and Analysis

4.1 Integer Multiplication Algorithm

We are given two integers $a, b < 2^N$ to multiply. We fix constants k and c whose values are given in Section 4.2. The algorithm is as follows:

1. Choose M and m as powers of two such that $M^k \approx \frac{N}{\log^2 N}$ and $m \approx \log N$. Find the least prime $p \equiv 1 \pmod{2M}$ (Lemma 2.4).
2. Encode the integers a and b as k-variate polynomials $a(X)$ and $b(X)$, respectively, over the ring $\mathcal{R} = \mathbb{Z}[\alpha]/(p^c, \alpha^m + 1)$ (Section 2.2).
3. Compute the root $\rho(\alpha)$ (Section 2.4).
4. Use $\rho(\alpha)$ as the principal $2M$-th root of unity to compute the Fourier transforms of the k-variate polynomials $a(X)$ and $b(X)$. Multiply component-wise and take the inverse Fourier transform to obtain the product polynomial. (Sections 3.1 and 3.2)
5. Evaluate the product polynomial at appropriate powers of two to recover the integer product and return it (Section 2.2).

4.2 Complexity Analysis

The choice of parameters should ensure that the following constraints are satisfied:

1. $M^k = O\left(\frac{N}{\log^2 N}\right)$ and $m = O(\log N)$.
2. $M^L = O(N^\varepsilon)$, where L is the Linnik constant (Theorem 2.3) and ε is any constant less than 1. Recall that this makes picking the prime by brute force feasible (see Lemma 2.4).
3. $p^c > 2M^k \cdot m \cdot 2^{2u}$ where $u = \frac{2N}{M^k m}$. This is to prevent overflows during modular arithmetic (see Section 2.2).

It is straightforward to check that $k > L + 1$ and $c > 5(k + 1)$ satisfy the above constraints. Since $L \leq 5.2$, it is sufficient to choose $k = 7$ and $c = 42$.

Let $T(N)$ denote the time complexity of multiplying two N bit integers. This consists of:

- Time required to pick a suitable prime p,
- Computing the root $\rho(\alpha)$,
- Encoding the input integers as polynomials,
- Multiplying the encoded polynomials,
- Evaluating the product polynomial.

As argued before, the prime p can be chosen in $o(N)$ time. To compute $\rho(\alpha)$, we need to lift a generator of $\mathbb{F}_p^\times$ to $\mathbb{Z}/p^c\mathbb{Z}$ followed by an interpolation. Since c is a constant and p is a prime of $O(\log N)$ bits, the time required for Hensel Lifting and interpolation is $o(N)$.

The encoding involves dividing bits into smaller blocks, and expressing the exponents of q in base M (Section 2.2) and all these take $O(N)$ time since M is a power of 2. Similarly, evaluation of the product polynomial takes linear time as well. Therefore, the time complexity is dominated by the time taken for polynomial multiplication.

Time complexity of Polynomial Multiplication

From Equation 1, the complexity of polynomial multiplication is given by,

$$\mathcal{F}(2M, k) = O\left(M^k \log M \cdot m \cdot \log p + \frac{M^k \log M}{\log m} \cdot \mathcal{M}_\mathcal{R}\right).$$

Proposition 4.1. *[Sch82] If $\mathcal{M}_\mathcal{R}$ denotes the complexity of multiplication in $\mathcal{R}$, then $\mathcal{M}_\mathcal{R} = T\left(O(\log^2 N)\right)$ where $T(x)$ denotes the complexity of multiplying two x-bit integers.*

Proof. Elements of $\mathcal{R}$ can be viewed as polynomials in α over $\mathbb{Z}/p^c\mathbb{Z}$ with degree at most m. Given two such polynomials $f(\alpha)$ and $g(\alpha)$, encode them as follows: Replace α by 2^d, transforming the polynomials $f(\alpha)$ and $g(\alpha)$ to the integers $f(2^d)$ and $g(2^d)$ respectively. The parameter d is chosen such that the coefficients of the product $h(\alpha) = f(\alpha)g(\alpha)$ can be recovered from the product $f(2^d) \cdot g(2^d)$. For this, it is sufficient to ensure that the maximum coefficient of $h(\alpha)$

is less than 2^d. Since f and g are polynomials of degree m, we would want 2^d to be greater than $m \cdot p^{2c}$, which can be ensured by choosing $d = O(\log N)$. The integers $f(2^d)$ and $g(2^d)$ are bounded by 2^{md}, which is of $O(\log^2 N)$ bits. The product $f(\alpha) \cdot g(\alpha)$ can be decoded from the integer product $f(2^d) \cdot g(2^d)$ by splitting the bits into $(2m-1)$ blocks of d bits each (one for each coefficient of α^i) to obtain a polynomial in $\mathbb{Z}[\alpha]$ and reducing it modulo p^c and $\alpha^m + 1$. Reducing modulo $(\alpha^m + 1)$ can be performed in $O(md) = O(\log^2 N)$ time. Dividing by p^c, which has $O(\log N)$ bits, can be performed in the same time as multiplying $O(\log N)$ bit integers using standard techniques (see for example [Knu98, Chapter 4]). Since $T(N) = \Omega(N)$, we have that $\mathcal{M}_{\mathcal{R}} = T(O(\log^2 N)) + O(\log N \cdot T(O(\log N))) = T(O(\log^2 N))$ bit operations. □

Therefore, the complexity of our integer multiplication algorithm $T(N)$ is given by,

$$\begin{aligned} T(N) &= O(\mathcal{F}(2M,k)) = O\left(M^k \log M \cdot m \cdot \log p + \frac{M^k \log M}{\log m} \cdot \mathcal{M}_{\mathcal{R}}\right) \\ &= O\left(N \log N + \frac{N}{\log N \cdot \log\log N} \cdot T(O(\log^2 N))\right) \end{aligned}$$

Solving the above recurrence leads to the following theorem.

Theorem 4.2. *Given two N bit integers, their product can be computed using $N \cdot \log N \cdot 2^{O(\log^* N)}$ bit operations.*

Performing on multi-tape turing machines

The upper-bound presented in Theorem 4.2 holds for multi-tape turing machines. The only part of the algorithm that warrants an explanation is regrouping of the terms in preparation for the inner and outer DFTs. For the inner DFT, we are given $|f\rangle = \sum_{a,b} f_{b,a} |x_b + a\rangle$ and we wish to write down $|f_b\rangle = \sum_a f_{b,a} |a\rangle$ for each $b \in B$. The following discussion essentially outlines how this can be performed on a multi-tape turing machine using $O(N \log m)$ bit operations. (Recall that the group $E = (\mathbb{Z}/2M\mathbb{Z})^k$ and $A = (\mathbb{Z}/2m\mathbb{Z})^k$ and both M and m are powers of 2).

We may assume that the coefficients are listed according to the natural lexicographic order of $(\mathbb{Z}/2M\mathbb{Z})^k$. To ease the presentation, we first present the approach for $k = 1$. It would be straightforward to generalize this to larger k by repeated applications of this approach.

Given as input is a sequence of coefficients f_g for each $g \in \tilde{E} = \mathbb{Z}/2M\mathbb{Z}$ in the natural lexicographic order on one of the tapes of the turing machine. We can then "shuffle" the input tape using two passes to order the coefficients according to $\{0, M, 1, M+1, \ldots, M-1, 2M-1\}$ (by copying the first half with appropriate blanks, and copying the second half on the second pass). This results in regrouping the inputs as various cosets of $\mathbb{Z}/2\mathbb{Z}$. By considering two successive elements as a block and repeating the shuffling, we obtain various cosets of $Z/4\mathbb{Z}$. Repeating this process $\log(2m)$ times regroups the elements according to various cosets of $\mathbb{Z}/2m\mathbb{Z}$.

Suppose that $E = (\mathbb{Z}/2M\mathbb{Z})^k$ and that the coefficients are ordered according to the natural lexicographic order. By repeating the same shuffling process for $\log(2m)$ steps, the coefficients are regrouped according to the cosets of $\{0\}^{k-1} \times (\mathbb{Z}/2m\mathbb{Z})$ and hence there are M/m many groups. By considering every $2m$ successive coefficients as one block, each of the M/m groups can be thought

of as coefficients ordered according to the natural lexicographic order of $(\mathbb{Z}/2M\mathbb{Z})^{k-1}$. By repeating this for $(k-1)$ more steps, we can reorder the coefficients according to the cosets of $(\mathbb{Z}/2m\mathbb{Z}^k)$.

The procedure totally requires $k\log(2m)$ passes over the tapes and hence can be performed in $O(N\log m)$ time on a 3-tape[2] turing machine. With the above discussion, it can be easily seen that the upper-bound holds for multi-tape turing machines.

Theorem 4.3. *Given two N bit integers, their product can be computed using $N\cdot\log N\cdot 2^{O(\log^* N)}$ bit operations on a multi-tape turing machine.*

5 A Comparison with Fürer's Algorithm

Our algorithm can be seen as a p-adic version of Fürer's integer multiplication algorithm, where the field $\mathbb{C}$ is replaced by $\mathbb{Q}_p$, the field of p-adic numbers (for a quick introduction, see Baker's online notes [Bak07]). Much like $\mathbb{C}$, where representing a general element (say in base 2) takes infinitely many bits, representing an element in $\mathbb{Q}_p$ takes infinitely many p-adic digits. Since we cannot work with infinitely many digits, all arithmetic has to be done with finite precision. Modular arithmetic in the base ring $\mathbb{Z}[\alpha]/(p^c,\alpha^m+1)$, can be viewed as arithmetic in the ring $\mathbb{Q}_p[\alpha]/(\alpha^m+1)$ keeping a precision of $\varepsilon=p^{-c}$.

Arithmetic with finite precision naturally introduces some errors in computation. However, the nature of $\mathbb{Q}_p$ makes the error analysis simpler. The field $\mathbb{Q}_p$ comes with a norm $|\cdot|_p$ called the p-adic norm, which satisfies the stronger triangle inequality $|x+y|_p\le\max\left(|x|_p,|y|_p\right)$ [Bak07, Proposition 2.6]. As a result, unlike in $\mathbb{C}$, the errors in computation do not compound.

Recall that the efficiency of FFT crucially depends on a special principal $2M$-th root of unity in $\mathbb{Q}_p[\alpha]/(\alpha^m+1)$. Such a root is constructed with the help of a primitive $2M$-th root of unity in $\mathbb{Q}_p$. The field $\mathbb{Q}_p$ has a primitive $2M$-th root of unity if and only if $2M$ divides $p-1$ [Bak07, Theorem 5.12]. Also, if $2M$ divides $p-1$, a $2M$-th root can be obtained from a $(p-1)$-th root of unity by taking a suitable power. A primitive $(p-1)$-th root of unity in $\mathbb{Q}_p$ can be constructed, to sufficient precision, using Hensel Lifting starting from a generator of $\mathbb{F}_p^\times$.

6 Conclusion

As mentioned earlier, there has been two approaches to multiplying integers - one using arithmetic over complex numbers and the other using modular arithmetic. Using complex numbers, Schönhage and Strassen [SS71] gave an $O(N\cdot\log N\cdot\log\log N\ldots 2^{O(\log^* N)})$ algorithm. Fürer [Für07] improved this complexity to $N\cdot\log N\cdot 2^{O(\log^* N)}$ using some special roots of unity. The other approach, that is modular arithmetic, can be seen as arithmetic in $\mathbb{Q}_p$ with certain precision. A direct adaptation of the Schönhage-Strassen's algorithm in the modular setting leads to an $O(N\cdot\log N\cdot\log\log N\ldots 2^{O(\log^* N)})$ time algorithm. In this work, we show that by choosing an appropriate prime and a special root of unity, a running time of $N\cdot\log N\cdot 2^{O(\log^* N)}$ can be achieved through modular arithmetic as well. Therefore, in a way, we have unified the two paradigms. The important question that remains open is:

- Can N-bit integers be multiplied using $O(N\cdot\log N)$ bit operations?

[2] an input tape, an output tape, and a third tape for a counter

Even an improvement of the complexity to $O(N \cdot \log N \cdot \log^* N)$ operations will be a significant step forward towards answering this question.

Acknowledgements

We are greatly thankful to the anonymous reviewers for their detailed comments that have improved the presentation of the paper significantly.

References

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[Bak07] Alan J. Baker. An introduction to p-adic numbers and p-adic analysis. *Online Notes*, 2007. http://www.maths.gla.ac.uk/∼ajb/dvi-ps/padicnotes.pdf.

[Für07] Martin Fürer. Faster Integer Multiplication. In *39th ACM Symposium on Theory of Computation (STOC 2007)*, pages 57–66, 2007.

[Für09] Martin Fürer. Faster Integer Multiplication. *SIAM Journal of Computing*, 39(3):979–1005, 2009.

[HB92] D. R. Heath-Brown. Zero-free regions for Dirichlet L-functions, and the least prime in an arithmetic progression. In *Proceedings of the London Mathematical Society, 64(3)*, pages 265–338, 1992.

[Isa94] I. Martin Isaacs. *Character theory of finite groups*. Dover publications Inc., New York, 1994.

[Knu98] Donald E. Knuth. *Art of Computer Programming*, volume 2. Addison-Wesley Professional, 1998.

[KO63] A Karatsuba and Y Ofman. Multipication of multidigit numbers on automata. *English Translation in Soviet Physics Doklady*, 7:595–596, 1963.

[Lin44] Yuri V. Linnik. On the least prime in an arithmetic progression, I. The basic theorem, II. The Deuring-Heilbronn's phenomenon. *Rec. Math. (Mat. Sbornik)*, 15:139–178 and 347–368, 1944.

[Mil76] Gary L. Miller. Riemann's Hypothesis and Tests for Primality. *J. Comput. Syst. Sci.*, 13(3):300–317, 1976.

[NZM91] Ivan Niven, Herbert S. Zuckerman, and Hugh L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley and Sons, Singapore, 1991.

[Rab80] Michael O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, 1980.

[Sch82] Arnold Schönhage. Asymptotically Fast Algorithms for the Numerical Multiplication and Division of Polynomials with Complex Coefficients. In *Computer Algebra, EUROCAM*, volume 144 of *Lecture Notes in Computer Science*, pages 3–15, 1982.

[Sha99] Igor R. Shafarevich. *Basic Notions of Algebra*. Springer Verlag, USA, 1999.

[SS71] A Schönhage and V Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[Tia90] Cai Tianxin. Primes Representable by Polynomials and the Lower Bound of the Least Primes in Arithmetic Progressions. *Acta Mathematica Sinica, New Series*, 6:289–296, 1990.

[Tit30] E. C. Titchmarsh. A Divisor Problem. *Rend. Circ. Mat. Palerme*, 54:414–429, 1930.

[Too63] A L. Toom. The Complexity of a Scheme of Functional Elements Simulating the Multiplication of Integers. *English Translation in Soviet Mathematics*, 3:714–716, 1963.

[Xyl10] Triantafyllos Xylouris. On Linnik's constant. arXiv:abs/0906.2749 [math.NT], 2010.