



Master Degree Project

Generalizing the Number Theoretic Transform



June 1, 2026

Author: Jan Kristof Köglburger

Supervisors: Per-Anders Svensson, Marcus Nilsson

Semester: Spring 2026

Subject: Mathematics

Abstract

The Number Theoretic Transform (NTT) allows an especially fast implementation of polynomial multiplication. It is usually described separately for $\mathbb{Z}_q[x]/(x^n - 1)$ and $\mathbb{Z}_q[x]/(x^n + 1)$. We define a generalized NTT for any ring $\mathbb{Z}_q[x]/(x^n - \omega^{nk})$, where ω is a generator of \mathbb{Z}_q^* , n is a power of 2, q is a prime such that $n \mid (q-1)$ and k is an integer such that $0 \leq k < (q-1)/n$. To weaken the condition on the prime to $(n/2^\beta) \mid (q-1)$, [Zho+18] and [ZLP21] described the methods Pt-NTT and K-NTT for the rings $\mathbb{Z}_q[x]/(x^n \pm 1)$. We apply these methods to the generalized NTT. Our approach is an algebraic one, whence the underlying ring isomorphisms of NTT, generalized NTT, Pt-NTT and K-NTT are emphasized throughout the thesis. Finally we conduct experiments in Python which measure the average time it takes to multiply two polynomials, demonstrating the significant speedup that our generalized NTT can provide compared to ordinary schoolbook multiplication.

Acknowledgements

I would like to thank Marcus for his valuable support in finding and concretizing the topic as well as co-supervising my thesis work. I am very grateful for the guidance and feedback that Per-Anders provided, not to forget his great teaching throughout the master's program.

I am truly thankful to my family who has always supported me both personally and financially and encouraged me to follow my interests, all of which enabled me to obtain this degree. I would also like to express my gratitude to my partner Léanne for the emotional support she provided throughout the thesis process, and to my friends Cristian and Jorge, who made my time in Växjö a great experience.

Last but not least, I appreciate that the Kingdom of Sweden and the European Union offered me the possibility to study without paying fees.

Contents

1	Introduction	1
1.1	Background	1
1.2	Number Theoretic Transform	1
1.3	Generalizing the NTT	1
1.4	Structure of the thesis	1
1.5	My contribution	2
2	Preliminaries	3
2.1	Abstract Algebra	4
2.2	Notation	5
3	Classical Number Theoretic Transform	6
3.1	Cyclic convolution using NTT	6
3.1.1	NTT	7
3.1.2	INTT	8
3.2	Negacyclic convolution using NTT	8
3.2.1	NTT	10
3.2.2	INTT	10
4	Our generalized Number Theoretic Transform	12
4.1	Polynomials and their factorization	12
4.2	gNTT	16
4.3	gINTT	19
4.4	Using gNTT to multiply polynomials	21
4.5	Computational complexity	22
5	Methods to weaken condition on prime	24
5.1	Pt-NTT	25
5.2	K-NTT	26
6	Experiments	28
6.1	Implementation	28
6.2	Results	28
6.2.1	Parameters from Dilithium	28
6.2.2	Parameters from Kyber	29
7	Conclusion	30
	References	31
A	Proofs	33
B	Python code	36
B.1	Packages	36
B.2	Preliminary functions	36
B.3	Schoolbook Multiplication	38
B.4	Generalized NTT	38
B.5	Pt-NTT	40
B.6	K-NTT	40

B.7	Timing	41
B.8	Displaying results	43

1 Introduction

1.1 Background

The possible development of a full-scale quantum computer has made it necessary to develop quantum-safe cryptosystems. In August 2024 the National Institute of Standards and Technology (NIST) in the United States released its first Post-Quantum Cryptography (PQC) standards. Two of these standards (Kyber [Nat24b] and Dilithium [Nat24a]) use the Number Theoretic Transform to significantly speed up polynomial multiplications in the ring $\mathbb{Z}_q[x]/(x^n + 1)$.

1.2 Number Theoretic Transform

The Number Theoretic Transform (NTT) is typically described for the rings $\mathbb{Z}_q[x]/(x^n - 1)$ and $\mathbb{Z}_q[x]/(x^n + 1)$ (see [LZ22] and [Sat+23]). Mathematically it is an isomorphism from $\mathbb{Z}_q[x]/(x^n \pm 1)$ to a direct product of n rings, each of which is isomorphic to \mathbb{Z}_q . Multiplying two elements in this direct product thus amounts to n multiplications in \mathbb{Z}_q . In comparison, using schoolbook multiplication in $\mathbb{Z}_q[x]/(x^n \pm 1)$ requires n^2 multiplications modulo q . One can efficiently compute the NTT and its inverse step by step, making it significantly faster to multiply two polynomials in $\mathbb{Z}_q[x]/(x^n \pm 1)$ by first applying the NTT to both of them, then computing the product of their images and at last applying the inverse of the NTT to the result.

To use the Number Theoretic Transform, however, n has to be a power of 2 and q a prime such that $n \mid (q - 1)$ (in the case of $\mathbb{Z}_q[x]/(x^n - 1)$) or $2n \mid (q - 1)$ (in the case of $\mathbb{Z}_q[x]/(x^n + 1)$). One can weaken the condition on the prime q to $(n/2^\beta) \mid (q - 1)$ with the methods Pt-NTT and K-NTT described in Section 6.2 of [LZ22], [Zho+18] and [ZLP21]. Basically one computes the multiplication in a ring which is isomorphic to $\mathbb{Z}_q[x]/(x^n \pm 1)$ and where products are computed using multiplications of polynomials in $\mathbb{Z}_q[y]/(y^{n/2^\beta} \pm 1)$.

1.3 Generalizing the NTT

Instead of just defining the NTT for $\mathbb{Z}_q[x]/(x^n - 1)$ and $\mathbb{Z}_q[x]/(x^n + 1)$ separately, this thesis generalizes the NTT to any factor ring $\mathbb{Z}_q[x]/(x^n - \omega^{nk})$. Again n is a power of 2 and q a prime such that $n \mid (q - 1)$. Moreover, ω is a generator of \mathbb{Z}_q^* and k is an integer such that $0 \leq k < (q - 1)/n$.

The methods to weaken the condition on the prime to $(n/2^\beta) \mid (q - 1)$ can also be applied to the generalized NTT.

1.4 Structure of the thesis

Section 2 equips the reader with important definitions and theorems.

In Section 3 the classical NTT for cyclic convolution (multiplication in the ring $\mathbb{Z}_q[x]/(x^n - 1)$) and negacyclic convolution (multiplication in the ring $\mathbb{Z}_q[x]/(x^n + 1)$) is presented. This section aims to make the reader understand the concept rather than providing mathematically rigorous proofs.

In Section 4 the generalized NTT is presented with all necessary results and proofs in full mathematical rigor. This section also contains results about the computational complexity of polynomial multiplication using the generalized NTT in contrast to schoolbook multiplication.

Section 5 describes the methods Pt-NTT and K-NTT which weaken the condition on the prime compared to the generalized NTT.

Finally, Section 6 contains the experimental part of this thesis. Implementations in Python of the different methods for polynomial multiplication (schoolbook multiplication, generalized NTT, Pt-NTT, K-NTT) are presented. The time needed for computing 10,000 products of randomly generated polynomials is measured for each method and a comparison is provided.

1.5 My contribution

The generalization of the NTT to $\mathbb{Z}_q[x]/(x^n - \omega^{nk})$ is my own work. All lemmas and propositions in Section 4 are formulated and proven by me.

Proposition 5.1 was formulated for $\mathbb{Z}_q[x]/(x^n \pm 1)$ in [LZ22] but not proven there. I formulate and prove it for the general case of $\mathbb{Z}_q[x]/(x^n - c)$.

I implement the multiplication methods in Python and conduct the time measurements myself.

I believe that this thesis facilitates the understanding of both the classical and the generalized NTT as a ring isomorphism.

2 Preliminaries

Definition 2.1 (Cyclic and negacyclic convolution). By *cyclic convolution* we mean multiplication in the ring $\mathbb{Z}_q[x]/(x^n - 1)$.

By *negacyclic convolution* we mean multiplication in the ring $\mathbb{Z}_q[x]/(x^n + 1)$.

Definition 2.2 (Schoolbook multiplication). Let $c \in \mathbb{Z}_q$ and

$$f(x) = \sum_{i=0}^{n-1} f_i x^i, \quad g(x) = \sum_{i=0}^{n-1} g_i x^i \in \mathbb{Z}_q[x]/(x^n - c).$$

Let

$$h(x) = \sum_{i=0}^{n-1} h_i x^i \in \mathbb{Z}_q[x]/(x^n - c)$$

denote the product of $f(x)$ and $g(x)$. By *schoolbook multiplication* we mean to compute $h(x)$ using

$$h_i = \sum_{k=0}^i f_k g_{i-k} + c \sum_{k=i+1}^{n-1} f_k g_{i+n-k}.$$

Example 2.1 (Schoolbook multiplication). Let $f_0 + f_1 x, g_0 + g_1 x \in \mathbb{Z}_q[x]/(x^2 - c)$. We compute their product $h_0 + h_1 x \in \mathbb{Z}_q[x]/(x^2 - c)$ via schoolbook multiplication by

$$h_0 = f_0 g_0 + c f_1 g_1 \quad \text{and} \quad h_1 = f_0 g_1 + f_1 g_0.$$

Definition 2.3 (Bit reversal). Let $n = 2^r$ for some $r \in \mathbb{N}$ and let the binary expansion of any integer $0 \leq i < n$ be given by

$$(b_{r-1} b_{r-2} \dots b_2 b_1 b_0)_2,$$

where $b_j \in \{0, 1\}$ for every $j \in \{0, 1, 2, \dots, r-1, r-2\}$. The *bit reversal of i with respect to n* is defined as

$$\text{BitRev}_n(i) = (b_0 b_1 b_2 \dots b_{r-2} b_{r-1})_2,$$

the integer that one gets when reversing the binary expansion of i .

Example 2.2 (Bit reversal). We want to compute the bit reversal of 11 with respect to 16. In binary form we have $11 = (1011)_2$, whence

$$\text{BitRev}_{16}(11) = (1101)_2 = 13.$$

The following description of the Karatsuba trick is taken from [LZ22].

Definition 2.4 (Karatsuba trick). Let f_i, f_j, g_i and g_j be elements of some ring. If one needs to compute

$$f_i g_i, f_j g_j \quad \text{and} \quad f_i g_j + f_j g_i,$$

one can save one multiplication at the cost of extra additions and subtractions by computing

$$(f_i + f_j)(g_i + g_j) - f_i g_i - f_j g_j$$

instead of computing $f_i g_j + f_j g_i$ directly. This is called the *Karatsuba trick*.

2.1 Abstract Algebra

See [Kru26], [Hun74] and [Lan02] for the following definitions and theorems.

Definition 2.5 (Generator of \mathbb{Z}_q^*). Let q be a prime. An element $\omega \in \mathbb{Z}_q$ is called a *generator* of $\mathbb{Z}_q^* = \mathbb{Z}_q \setminus \{0\}$ if

$$\{\omega^k \mid k \in \mathbb{Z}, 0 \leq k < q - 1\} = \mathbb{Z}_q^*.$$

Example 2.3 (Generator of \mathbb{Z}_7^*). Consider the element $3 \in \mathbb{Z}_7^*$. We have

$$\{3^k \mid 0 \leq k < 6\} = \{1, 3, 2, 6, 4, 5\} = \mathbb{Z}_7^*,$$

whence 3 is a generator of \mathbb{Z}_7^* .

Theorem 2.1 (\mathbb{Z}_q^* is cyclic). Let q be a prime. The field \mathbb{Z}_q contains a generator of $\mathbb{Z}_q^* = \mathbb{Z}_q \setminus \{0\}$, i.e. there exists $\omega \in \mathbb{Z}_q$ such that

$$\{\omega^k \mid k \in \mathbb{Z}, 0 \leq k < q - 1\} = \mathbb{Z}_q^*.$$

Definition 2.6 (Primitive n -th root of unity). Let q be a prime and n be positive integer. An element $\zeta \in \mathbb{Z}_q$ is called a *primitive n -th root of unity* if $\zeta^n = 1$ and $\zeta^k \neq 1$ for any integer $1 \leq k < n$.

Example 2.4 (Primitive 4-th root of unity). Consider the element $13 \in \mathbb{Z}_{17}^*$. We have

$$\begin{aligned} 13^2 &= 169 = -1 \pmod{17}, & 13^3 &= -1 \cdot 13 = -13 \pmod{17} \\ \text{and } 13^4 &= (-1)^2 = 1 \pmod{17}, \end{aligned}$$

whence 13 is a primitive 4-th root of unity in \mathbb{Z}_{17}^* .

Definition 2.7 (Sum of two ideals). Let R be a commutative ring with unity and I and J be two ideals of R . The *sum* of I and J is defined as

$$I + J := \{i + j \mid i \in I, j \in J\}.$$

Definition 2.8 (Product of ideals). Let R be a commutative ring with unity and I_1, I_2, \dots, I_n be ideals of R . The *product* of these ideals is defined as

$$I_1 I_2 \cdots I_n := \left\{ \sum_{k=0}^m i_{1,k} i_{2,k} \cdots i_{n,k} \mid m \in \mathbb{N}, i_{1,k} \in I_1, i_{2,k} \in I_2, \dots, i_{n,k} \in I_n \right\}.$$

Definition 2.9 (Coprime ideals). Let R be a commutative ring with unity. Two ideals I and J of R are called *coprime* if

$$I + J = R.$$

The ideals I_1, I_2, \dots, I_n of R are called *pairwise coprime* if

$$I_k + I_l = R$$

for any $k, l \in \{1, 2, \dots, n\}$ with $k \neq l$.

Theorem 2.2 (Chinese Remainder Theorem for rings). Let R be a commutative ring with unity and I_1, I_2, \dots, I_n be ideals of R that are pairwise coprime. The mapping

$$\begin{aligned} \phi : R/(I_1 I_2 \cdots I_n) &\mapsto (R/I_1) \times (R/I_2) \times \dots \times (R/I_n) \\ r + (I_1 I_2 \cdots I_n) &\mapsto (r + I_1, r + I_2, \dots, r + I_n) \end{aligned}$$

is a ring isomorphism, whence

$$R/(I_1 I_2 \cdots I_n) \cong (R/I_1) \times (R/I_2) \times \dots \times (R/I_n).$$

2.2 Notation

We denote component-wise multiplication of vectors by \odot .

3 Classical Number Theoretic Transform

Let n be a power of 2 and q be a prime. The fundamental idea behind the Number Theoretic Transform is to repeatedly use the identity

$$x^2 - b^2 = (x - b) \cdot (x + b)$$

to factorize the polynomial $x^2 - b^2$, where $b \in \mathbb{Z}_q$.

3.1 Cyclic convolution using NTT

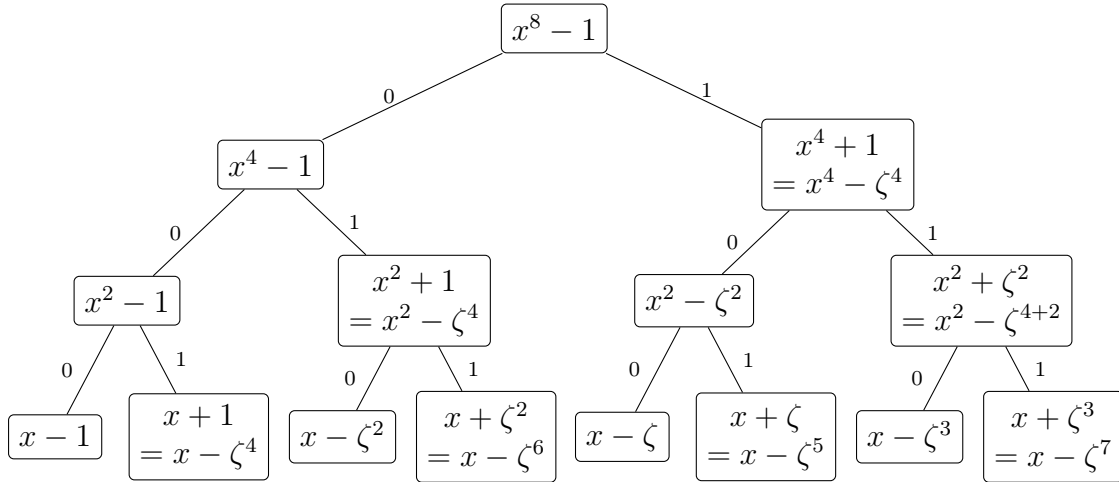


Figure 3.1: Illustration of the steps that split the polynomial $x^8 - 1$ into linear factors. The identity $\zeta^4 = -1 \in \mathbb{Z}_q$ is used. A value of 0 next to the edges indicates that the edge goes to the factor containing the minus, which we always write on the left side. On the other hand a value of 1 indicates that the edge goes to the factor containing the plus, which we always write on the right side. In case we go to the right side, we have to replace $+$ by $-\zeta^4$.

Let $q - 1$ be divisible by n . Then a primitive n -th root of unity in \mathbb{Z}_q exists. Let ζ be such a n -th root of unity. It holds that $\zeta^{n/2} = -1 \in \mathbb{Z}_q$. We can factorize $x^n - 1$ step by step until we have only linear factors. Assuming that $n \geq 4$, we first factorize

$$x^n - 1 = (x^{n/2} - 1) \cdot (x^{n/2} + 1)$$

and then

$$(x^{n/2} - 1) = (x^{n/4} - 1) \cdot (x^{n/4} + 1)$$

as well as

$$(x^{n/2} + 1) = (x^{n/2} - (-1)) = (x^{n/2} - \zeta^{n/2}) = (x^{n/4} - \zeta^{n/4}) \cdot (x^{n/4} + \zeta^{n/4}).$$

We can draw a tree that shows how in every step the polynomial is factorized into two polynomials of half the degree using the identity $(x^{2^\alpha} - \zeta^{2^j}) = (x^{2^{\alpha-1}} - \zeta^j) \cdot (x^{2^{\alpha-1}} + \zeta^j)$ (see Figure 3.1 for an illustration). Note that we always write the factor $(x^{2^{\alpha-1}} - \zeta^j)$ on the left side (indicated by a 0 next to the edge) and the factor $(x^{2^{\alpha-1}} + \zeta^j)$ on the right side (indicated by a 1 next to the edge).

We rewrite $(x^{2^{\alpha-1}} + \zeta^j)$ as $(x^{2^{\alpha-1}} - \zeta^{n/2+j})$ and can continue the factorization in case $\alpha > 1$. A value of 1 next to the edge encodes the fact that we add the term $n/2$ to the exponent of ζ in that particular step. The term $n/2$ in the exponent of ζ will be divided by 2 whenever we go one layer down (either to the left or to the right). The later the term $n/2$ gets added to the exponent of ζ , the more significant it is in the end. If it gets added in the last layer (the layer where the polynomials are linear), it does not get divided by 2 and is simply $n/2$. If it gets added at the very beginning of the factorization, it gets divided by 2 exactly $\log_2(n) - 1$ times which means that in the last layer it is reduced to 1.

This leads us to a simple formula for the exponent i' of ζ of any polynomial $x - \zeta^{i'}$ in the last layer. We get i' in binary form by concatenating the bits next to the edges that we need to follow to get from $x - \zeta^{i'}$ to the original polynomial $x^n - 1$. For example, in Figure 3.1 we find the value 1 next to the edge from the polynomial $x + \zeta^2 = x - \zeta^6$ towards the polynomial $x^2 + 1 = x^2 - \zeta^4$, the value 1 next to the edge from that polynomial to the polynomial $x^4 - 1$ and the value 0 from that polynomial towards the polynomial $x^8 - 1$. We have $(110)_2 = 6$, which is the exponent of ζ in $x - \zeta^6$.

If we follow the edges in the other direction (from $x^n - 1$ to the linear polynomial) we simply get the horizontal position in which we find the linear polynomial (starting from 0 and ending with $n - 1$). For example, in Figure 3.1 the polynomial $x + \zeta^2 = x - \zeta^6$ is at position $3 = (011)_2$ from the left. Note that the polynomial $x - 1$ is at position $0 = (000)_2$ and the polynomial $x + \zeta^3 = x - \zeta^7$ is at position $7 = (111)_2$.

Hence the exponent of ζ of any polynomial in the last layer is given by the bit reversal of its position with respect to n . We can write

$$x^n - 1 = \prod_{i=0}^{n-1} (x - \zeta^{\text{BitRev}_n(i)}), \quad (1)$$

The order of the factors on the right-hand side of Equation (1) is the same as the order of the factors from left to right in the last layer of the tree.

3.1.1 NTT

Since the ideals $(x - \zeta^{\text{BitRev}_n(0)}), (x - \zeta^{\text{BitRev}_n(1)}), \dots, (x - \zeta^{\text{BitRev}_n(n-1)})$ are pairwise coprime, it follows by the Chinese Remainder Theorem for rings (Theorem 2.2) that

$$\mathbb{Z}_q[x]/(x^n - 1) \cong \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \zeta^{\text{BitRev}_n(i)}).$$

We can compute this isomorphism step by step using the tree structure we described for the factorization of $x^n - 1$. Given a polynomial $f \in \mathbb{Z}_q[x]/(x^n - 1)$ we compute in the first step $f_0 \in \mathbb{Z}_q[x]/(x^{n/2} - 1)$ by substituting $x^{n/2} = 1$ and $f_1 \in \mathbb{Z}_q[x]/(x^{n/2} + 1)$ by substituting $x^{n/2} = -1$. In the second step we compute $f_{00} \in \mathbb{Z}_q[x]/(x^{n/4} - 1)$ and $f_{01} \in \mathbb{Z}_q[x]/(x^{n/4} + 1)$ by substituting $x^{n/4} = \pm 1$ in f_0 as well as $f_{10} \in \mathbb{Z}_q[x]/(x^{n/4} - \zeta^{n/4})$ and $f_{11} \in \mathbb{Z}_q[x]/(x^{n/4} + \zeta^{n/4})$ by substituting $x^{n/4} = \pm \zeta^{n/4}$. Repeating this procedure eventually yields the desired constant polynomials in $\mathbb{Z}_q[x]/(x - \zeta^{\text{BitRev}_n(i)})$. The isomorphism we described is the Number Theoretic Transform (NTT) for cyclic convolution.

3.1.2 INTT

The computation of the Inverse Number Theoretic Transform (INTT) can be carried out by traversing the tree in the opposite direction. Given a vector

$$(a_0, a_1, \dots, a_{n-1}) \in \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \zeta^{\text{BitRev}_n(i)})$$

we remember that in the NTT we would compute a_{2i} and a_{2i+1} from a polynomial

$$(b_{2i} + b_{2i+1}x) \in \mathbb{Z}_q[x]/(x^2 - \zeta^{2 \cdot \text{BitRev}_n(2i)})$$

by

$$a_{2i} = b_{2i} + \zeta^{\text{BitRev}_n(2i)} b_{2i+1}, \quad a_{2i+1} = b_{2i} - \zeta^{\text{BitRev}_n(2i)} b_{2i+1}.$$

The equations

$$\begin{aligned} 2^{-1}(a_{2i} + a_{2i+1}) &= 2^{-1} \cdot 2 \cdot b_{2i} = b_{2i}, \\ 2^{-1}(\zeta^{\text{BitRev}_n(2i)})^{-1}(a_{2i} - a_{2i+1}) &= 2^{-1}(\zeta^{\text{BitRev}_n(2i)})^{-1} \cdot 2 \cdot \zeta^{\text{BitRev}_n(2i)} \cdot b_{2i+1} = b_{2i+1} \end{aligned}$$

show how to compute the polynomial $b_{2i} + b_{2i+1}x$ from a_{2i} and a_{2i+1} for any $i \in \{0, 1, 2, \dots, \frac{n}{2} - 1\}$. For the next step we remember that in the NTT we would compute the polynomials

$$\begin{aligned} (b_{4i} + b_{4i+1}x) &\in \mathbb{Z}_q[x]/(x^2 - \zeta^{2 \cdot \text{BitRev}_n(4i)}), \\ (b_{4i+2} + b_{4i+3}x) &\in \mathbb{Z}_q[x]/(x^2 - \zeta^{2 \cdot \text{BitRev}_n(4i+2)}) = \mathbb{Z}_q[x]/(x^2 + \zeta^{2 \cdot \text{BitRev}_n(4i)}) \end{aligned}$$

from a polynomial

$$(c_{4i} + c_{4i+1}x + c_{4i+2}x^2 + c_{4i+3}x^3) \in \mathbb{Z}_q[x]/(x^4 - \zeta^{4 \cdot \text{BitRev}_n(4i)})$$

by

$$\begin{aligned} \tilde{b}_{4i}(x) &:= b_{4i} + b_{4i+1}x = (c_{4i} + c_{4i+1}x) + \zeta^{2 \cdot \text{BitRev}_n(4i)} \cdot (c_{4i+2} + c_{4i+3}x), \\ \hat{b}_{4i}(x) &:= b_{4i+2} + b_{4i+3}x = (c_{4i} + c_{4i+1}x) - \zeta^{2 \cdot \text{BitRev}_n(4i)} \cdot (c_{4i+2} + c_{4i+3}x). \end{aligned}$$

Hence we get the equations

$$\begin{aligned} 2^{-1} \cdot (\tilde{b}_{4i}(x) + \hat{b}_{4i}(x)) &= c_{4i} + c_{4i+1}x, \\ 2^{-1} \cdot (\zeta^{2 \cdot \text{BitRev}_n(4i)})^{-1} \cdot (\tilde{b}_{4i}(x) - \hat{b}_{4i}(x)) &= c_{4i+2} + c_{4i+3}x \end{aligned}$$

for computing the polynomial $c_{4i} + c_{4i+1}x + c_{4i+2}x^2 + c_{4i+3}x^3$ from $\tilde{b}_{4i}(x)$ and $\hat{b}_{4i}(x)$.

Repeating the procedure layer by layer eventually yields the desired polynomial in $\mathbb{Z}_q[x]/(x^n - 1)$. Note that instead of multiplying by 2^{-1} in every layer, one could just multiply by $n^{-1} = (2^{-1})^{\log_2(n)}$ at the end.

3.2 Negacyclic convolution using NTT

Let $q - 1$ be divisible by $2n$. We know that a primitive $2n$ -th root of unity exists in \mathbb{Z}_q . Let ξ be such a primitive $2n$ -th root of unity. It holds that $\xi^n = -1 \in \mathbb{Z}_q$.

Similarly to the cyclic convolution, we can factor $x^n + 1$ into linear factors, using the identity $\xi^n = -1$, whenever necessary. Figure 3.2 shows the factorization for $n = 8$.

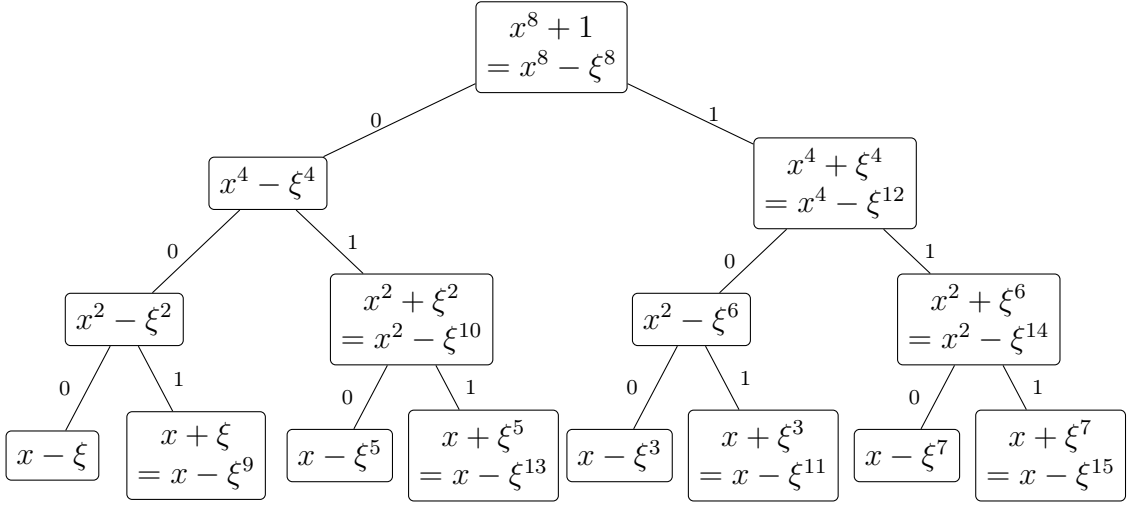


Figure 3.2: Illustration of the steps that split the polynomial $x^8 + 1$ into linear factors. The identity $\xi^8 = -1 \in \mathbb{Z}_q$ is used. A value of 0 next to the edges indicates that the edge goes to the factor containing the minus, which we always write on the left side. On the other hand a value of 1 indicates that the edge goes to the factor containing the plus, which we always write on the right side. In case we go to the right side, we have to replace $+$ by $-\xi^8$.

We can derive the exponents of ξ in a similar manner like in Section 3.1. Here, however, we add the term n to the exponent of ξ , whenever we go to the right side. Moreover, we start with the exponent n , since $x^n + 1 = x^n - \xi^n$. Exactly as in the case of cyclic convolution, the exponent of ξ gets divided by 2 whenever we go one layer down, no matter if we go to the left or to the right. Consider the exponent i' of any polynomial $x - \xi^{i'}$ in the last layer. It is given in binary form by concatenating all the bits next to the edges that we need to follow to get from $x - \xi^{i'}$ to the original polynomial $x^n + 1$ and the bit 1 (which corresponds to the exponent n of ξ in $x^n - \xi^n$). For example, in Figure 3.2 we find the value 1 next to the edge from the polynomial $x - \xi^{11}$ towards the polynomial $x^2 - \xi^6$, the value 0 next to the edge from that polynomial to the polynomial $x^4 - \xi^{12}$ and the value 1 next to the edge from that polynomial towards the original polynomial $x^8 - \xi^8$. Concatenating those three bits and the bit 1 (which corresponds to the exponent of ξ in the original polynomial) gives the integer $(1011)_2 = 11$, which is the exponent of ξ in $x - \xi^{11}$. Reversing the order of the bitstring yields the horizontal position in which we find the linear polynomial (starting from n and ending with $2n - 1$ since the most significant bit is always 1). For example, in Figure 3.2, the polynomial $x - \xi^{11}$ is at position

$$\text{BitRev}_{16}((1011)_2) = (1101)_2 = 13 = 8 + 5,$$

which is the sixth position from the left since the first position is 8. Due to the reasoning above we can factorize

$$x^n + 1 = \prod_{i=n}^{2n-1} (x - \xi^{\text{BitRev}_{2n}(i)}). \quad (2)$$

Since the most significant bit in the binary form of i is always 1, i.e.

$$i = (1b_{\log_2(n)-1} \dots b_1 b_0)_2 \quad \text{for some} \quad b_{\log_2(n)-1}, \dots, b_1, b_0 \in \{0, 1\},$$

we can rewrite

$$\begin{aligned}\text{BitRev}_{2n}(i) &= (b_0 b_1 \dots b_{\log_2(n)-1} 1)_2 = 2 \cdot (b_0 b_1 \dots b_{\log_2(n)-1})_2 + 1 \\ &= 2 \cdot \text{BitRev}_n((b_{\log_2(n)-1} \dots b_1 b_0)_2) + 1 = 2 \cdot \text{BitRev}_n(i - n) + 1.\end{aligned}$$

Hence Equation (2) becomes

$$x^n + 1 = \prod_{i=0}^{n-1} (x - \xi^{2 \cdot \text{BitRev}_n(i)+1}). \quad (3)$$

3.2.1 NTT

Since the ideals $(x - \xi^{2 \cdot \text{BitRev}_n(0)+1}), (x - \xi^{2 \cdot \text{BitRev}_n(1)+1}), \dots, (x - \xi^{2 \cdot \text{BitRev}_n(n-1)+1})$ are pairwise coprime, it follows by the Chinese Remainder Theorem for rings (Theorem 2.2) that

$$\mathbb{Z}_q[x]/(x^n + 1) \cong \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \xi^{2 \cdot \text{BitRev}_n(i)+1}).$$

This isomorphism is the Number Theoretic Transform for negacyclic convolution. Analogously to the case of cyclic convolution, we can compute it step by step using the tree structure of the factorization of $x^n + 1$.

3.2.2 INTT

The computation of the Inverse Number Theoretic Transform can be carried out by traversing the tree step by step in the opposite direction. It works analogously to the case of cyclic convolution but allows a simpler implementation. One can use the same powers of ξ that one needs (and has usually precomputed) for the NTT, in the computation of the Inverse NTT. In Figure 3.2 this is easy to see. In the last layer we have $\xi \cdot (-\xi^7) = -\xi^8 = 1$ and $\xi^5 \cdot (-\xi^3) = -\xi^8 = 1$. In general, given a vector

$$(f_0, f_1, \dots, f_{n-1}) \in \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \xi^{2 \cdot \text{BitRev}_n(i)+1})$$

we note that in the NTT we would compute f_{2i} and f_{2i+1} from a polynomial

$$(g_{2i} + g_{2i+1}x) \in \mathbb{Z}_q[x]/(x^2 - \xi^{4 \cdot \text{BitRev}_n(2i)+2})$$

by

$$f_{2i} = g_{2i} + \xi^{2 \cdot \text{BitRev}_n(2i)+1} g_{2i+1}, \quad f_{2i+1} = g_{2i} - \xi^{2 \cdot \text{BitRev}_n(2i)+1} g_{2i+1}.$$

The equations

$$2^{-1}(f_{2i} + f_{2i+1}) = g_{2i} \quad \text{and} \quad 2^{-1}(\xi^{2 \cdot \text{BitRev}_n(2i)+1})^{-1}(f_{2i} - f_{2i+1}) = g_{2i+1}$$

show how to compute the polynomial $g_{2i} + g_{2i+1}x$ from f_{2i} and f_{2i+1} for any $i \in \{0, 1, 2, \dots, \frac{n}{2} - 1\}$.

We take a closer look at the term $(\xi^{2 \cdot \text{BitRev}_n(2i)+1})^{-1}$. Since i is any integer between 0 and $n/2 - 1$, it follows that $0 \leq 2i \leq n - 2$ and

$$0 \leq n - 2 - 2i \leq n - 2.$$

Hence there exists an integer $0 \leq j \leq n/2 - 1$ such that $2j = n - 2 - 2i$. We claim that

$$(\xi^{2 \cdot \text{BitRev}_n(2i)+1})^{-1} = -\xi^{2 \cdot \text{BitRev}_n(2j)+1}. \quad (4)$$

Note that in binary form we have

$$2i = (b_{\log_2(n)-1} \dots b_1 b_0 0)_2$$

for some $b_{\log_2(n)-1}, \dots, b_1, b_0 \in \{0, 1\}$ and

$$n - 2 - 2i = (\hat{b}_{\log_2(n)-1} \dots \hat{b}_1 \hat{b}_0 0)_2,$$

where $\hat{b}_s = 1 - b_s$. Therefore we get

$$2 \cdot \text{BitRev}_n(2i) + 2 \cdot \text{BitRev}_n(n - 2 - 2i) = (1 \dots 10)_2 = n - 2,$$

whence

$$\xi^{2 \cdot \text{BitRev}_n(2i)+1} \cdot \xi^{2 \cdot \text{BitRev}_n(2j)+1} = \xi^n.$$

Since $\xi^n = -1$, Equation (4) follows.

We have shown that the inverse of the power of ξ which we need in the computation of the Inverse NTT is (apart from a minus) just another power of ξ which we also need for the computation of the NTT. This is the reason why the NTT in [Nat24a] is implemented with only one array of precomputed powers of the $2n$ -th root of unity. The same array is used for both the computation of the NTT and the Inverse NTT.

4 Our generalized Number Theoretic Transform

Let n be a power of 2 and q be a prime such that $n \mid (q-1)$. Due to Theorem 2.1 there exists a generator $w \in \mathbb{Z}_q^*$. Let k be an integer such that $0 \leq k < (q-1)/n$. Throughout this section we assume that n, q, ω and k fulfill those conditions, if not stated otherwise.

4.1 Polynomials and their factorization

One might think that the polynomials $x^n - \omega^{nk}$ which permit our generalized NTT depend on the choice of the generator w . Proposition 4.1 shows that no matter which generator we choose, we always get the same set of polynomials.

Proposition 4.1. *Let ω and $\hat{\omega}$ be generators of \mathbb{Z}_q^* . It holds that*

$$\Omega := \{\omega^{nk} \mid k \in \mathbb{Z}, 0 \leq k < (q-1)/n\} = \{\hat{\omega}^{nk} \mid k \in \mathbb{Z}, 0 \leq k < (q-1)/n\} =: \hat{\Omega}.$$

Proof. Let $a = \omega^{nk} \in \Omega$. Since $\hat{\omega}$ is a generator of \mathbb{Z}_q^* and $\omega \in \mathbb{Z}_q^*$, there exists an integer $0 \leq j < q-1$ such that $\hat{\omega}^j = \omega$. We have $a = \omega^{nk} = (\hat{\omega}^j)^{nk} = \hat{\omega}^{jnk}$. In case $jk < (q-1)/n$, we get $a = \hat{\omega}^{jnk} \in \hat{\Omega}$.

Suppose that $jk \geq (q-1)/n$. This obviously implies $njk \geq q-1$. We can find positive integers m and $0 \leq l < q-1$ such that $njk = m(q-1) + l$. Since $n \mid (q-1)$ there exists a positive integer s such that

$$njk = mns + l \iff n(jk - ms) = l.$$

Since $0 \leq l < q-1$, we have $0 \leq jk - ms < (q-1)/n$ and thus $\hat{\omega}^l \in \hat{\Omega}$. Due to Fermat's little theorem it holds that

$$a = \hat{\omega}^{njk} = \hat{\omega}^{m(q-1)+l} = (\hat{\omega}^{q-1})^m \hat{\omega}^l = \hat{\omega}^l \in \hat{\Omega}.$$

We have shown that $\Omega \subseteq \hat{\Omega}$. Interchanging the roles of ω and $\hat{\omega}$ in the argumentation above yields $\hat{\Omega} \subseteq \Omega$. We therefore get $\Omega = \hat{\Omega}$. \square

We describe how the polynomial $x^n - \omega^{nk}$ splits into linear factors over \mathbb{Z}_q . First note that $\omega^{(q-1)/2} = -1$ in \mathbb{Z}_q . Similar to the classical NTT we repeatedly use the identity

$$x^{2^\alpha} - \omega^{2^j} = (x^\alpha - \omega^j)(x^\alpha + \omega^j) = (x^\alpha - \omega^j)(x^\alpha - \omega^{(q-1)/2+j}). \quad (5)$$

In the first step we have

$$x^n - \omega^{nk} = (x^{n/2} - \omega^{n/2 \cdot k})(x^{n/2} + \omega^{n/2 \cdot k}) = (x^{n/2} - \omega^{n/2 \cdot k})(x^{n/2} - \omega^{(q-1)/2+n/2 \cdot k}).$$

In the second step we get

$$(x^{n/2} - \omega^{n/2 \cdot k}) = (x^{n/4} - \omega^{n/4 \cdot k})(x^{n/4} + \omega^{n/4 \cdot k}) = (x^{n/4} - \omega^{n/4 \cdot k})(x^{n/4} - \omega^{(q-1)/2+n/4 \cdot k})$$

and

$$(x^{n/2} - \omega^{(q-1)/2+n/2 \cdot k}) = (x^{n/4} - \omega^{(q-1)/4+n/4 \cdot k})(x^{n/4} + \omega^{(q-1)/4+n/4 \cdot k}).$$

The following two lemmas contain formulas for the factorization of $x^n - \omega^{nk}$ in every step.

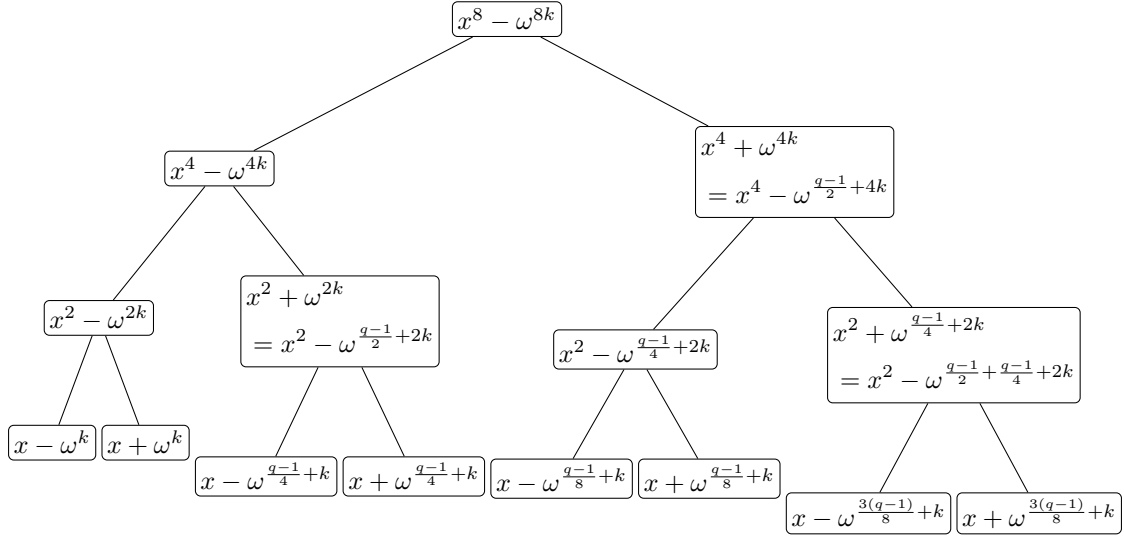


Figure 4.1: Illustration of the steps that split the polynomial $x^8 - \omega^{8k}$ into linear factors. The identity $\omega^{(q-1)/2} = -1 \in \mathbb{Z}_q$ is used.

Lemma 4.1. *The polynomial $x^n - \omega^{nk}$ splits into linear factors over \mathbb{Z}_q . For any integer $0 \leq \alpha \leq \log_2(n)$ it holds that*

$$x^n - \omega^{nk} = \prod_{i=0}^{2^\alpha - 1} \left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n + n/2^\alpha \cdot k} \right) \quad (6)$$

In particular, for any integer $0 \leq \alpha \leq \log_2(n) - 1$ it holds that

$$\begin{aligned} & x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n + n/2^\alpha \cdot k} \\ &= \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i)(q-1)/n + n/2^{\alpha+1} \cdot k} \right) \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i+1)(q-1)/n + n/2^{\alpha+1} \cdot k} \right), \end{aligned} \quad (7)$$

where $0 \leq i \leq 2^\alpha - 1$.

Proof. We prove Equation (6) by induction over α . Obviously it holds for $\alpha = 0$. Suppose that Equation (6) holds for α , where $0 \leq \alpha \leq \log_2(n) - 1$. Consider any factor

$$x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n + n/2^\alpha \cdot k},$$

where $0 \leq i \leq 2^\alpha - 1$. Clearly $n/2^\alpha$ is divisible by two since n is a power of 2 and $\alpha < \log_2(n)$. Let the binary expansion of i be given by

$$(b_{\log_2(n)-1} b_{\log_2(n)-2} \dots b_2 b_1 b_0)_2,$$

where $b_j \in \{0, 1\}$. We show that $b_{\log_2(n)-1} = 0$. Suppose that $b_{\log_2(n)-1} = 1$. Then

$$i \geq 2^{\log_2(n)-1} \geq 2^\alpha > 2^\alpha - 1,$$

which contradicts $i \leq 2^\alpha - 1$. Hence $b_{\log_2(n)-1} = 0$ and

$$\text{BitRev}_n(i) = (b_0 b_1 b_2 \dots b_{\log_2(n)-2} 0)_2,$$

which is divisible by two. Precisely, we have

$$\begin{aligned} \text{BitRev}_n(i)/2 &= (0b_0b_1b_2 \dots b_{\log_2(n)-2})_2 \\ &= \text{BitRev}_n((b_{\log_2(n)-2} \dots b_2b_1b_00)_2) = \text{BitRev}_n(2i). \end{aligned}$$

We can therefore factorize

$$x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \quad (8)$$

as

$$\left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k}\right) \left(x^{n/2^{\alpha+1}} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k}\right) \quad (9)$$

for every $0 \leq i \leq 2^\alpha - 1$. Note that

$$n/2 = (\underbrace{10 \dots 0}_n)_{\text{digits}} = \text{BitRev}_n((0 \dots 01)_2) = \text{BitRev}_n(1)$$

and thus

$$\frac{q-1}{2} = \frac{n}{2} \cdot \frac{q-1}{n} = \text{BitRev}_n(1) \cdot \frac{q-1}{n}.$$

Using the identity $\omega^{(q-1)/2} = -1$ we can rewrite

$$\begin{aligned} x^{n/2^{\alpha+1}} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} &= x^{n/2^{\alpha+1}} - \omega^{(\text{BitRev}_n(1)+\text{BitRev}_n(2i))(q-1)/n+n/2^{\alpha+1} \cdot k} \\ &= x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i+1)(q-1)/n+n/2^{\alpha+1} \cdot k}, \end{aligned}$$

which proves Equation (7). Using the induction hypothesis, we also get

$$\begin{aligned} x^n - \omega^{nk} &= \prod_{i=0}^{2^\alpha-1} \left(\left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) \right. \\ &\quad \left. \cdot \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i+1)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) \right) \\ &= \prod_{i=0}^{2^{\alpha+1}-1} \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right), \end{aligned} \quad (10)$$

which shows that Equation (6) holds for $\alpha + 1$. By the principle of induction, Equation (6) holds for every integer $0 \leq \alpha \leq \log_2(n)$. For $\alpha = \log_2(n)$ the exponent of x on the right-hand side of Equation (6) is given by $n/2^\alpha = n/2^{\log_2(n)} = 1$, which means that $x^n - \omega^{nk}$ splits into linear factors. \square

In the proof of Lemma 4.1 we first rewrote Expression (8) as Expression (9), where the two factors only differ by a minus in front of the constant term of the polynomial. We will see later that writing down the factorization in this form is useful for an efficient computation of the NTT. Hence we state it as Lemma 4.2.

Lemma 4.2. *For any integer $1 \leq \alpha \leq \log_2(n)$ it holds that*

$$\begin{aligned} x^n - \omega^{nk} &= \prod_{\substack{i=0 \\ i \text{ even}}}^{2^\alpha-2} \left(\left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right) \right. \\ &\quad \left. \cdot \left(x^{n/2^\alpha} + \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right) \right). \end{aligned}$$

In particular, for any integer $0 \leq \alpha \leq \log_2(n) - 1$ it holds that

$$\begin{aligned} & x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \\ &= \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) \left(x^{n/2^{\alpha+1}} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right), \end{aligned} \quad (11)$$

where $0 \leq i \leq 2^\alpha - 1$.

Proof. Following the proof of Lemma 4.1, we can for $1 \leq \alpha \leq \log_2(n)$ rewrite

$$x^{n/2^{\alpha-1}} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^{\alpha-1} \cdot k} \quad (12)$$

as

$$\left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^\alpha \cdot k} \right) \left(x^{n/2^\alpha} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^\alpha \cdot k} \right) \quad (13)$$

for every $0 \leq i \leq 2^{\alpha-1} - 1$. Hence Equation (11) holds. Using Equation (6) we get

$$\begin{aligned} x^n - \omega^{nk} &= \prod_{i=0}^{2^{\alpha-1}-1} \left(\left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^\alpha \cdot k} \right) \right. \\ &\quad \left. \cdot \left(x^{n/2^\alpha} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^\alpha \cdot k} \right) \right) \\ &= \prod_{\substack{i=0 \\ i \text{ even}}}^{2^\alpha-2} \left(\left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right) \right. \\ &\quad \left. \cdot \left(x^{n/2^\alpha} + \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right) \right). \quad \square \end{aligned}$$

As in the case of the classical NTT, we will use the Chinese Remainder Theorem for rings. In order to do that, we first need to verify that the ideals generated by the factors given in Lemmas 4.1 and 4.2 are pairwise coprime.

Lemma 4.3. *For any integer $0 \leq \alpha \leq \log_2(n)$ and any integer $0 \leq i \leq 2^\alpha - 1$ let $I_{\alpha,i}$ be the ideal of $\mathbb{Z}_q[x]$ that is generated by the polynomial*

$$x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k}.$$

The ideals $I_{\alpha,0}, I_{\alpha,1}, \dots, I_{\alpha,2^\alpha-1}$ are pairwise coprime.

Proof. Let $i, j \in \{0, 1, \dots, 2^\alpha - 1\}$ and $i \neq j$. By the definition of an ideal we have

$$x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \in I_{\alpha,i} = \left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right)$$

and

$$-x^{n/2^\alpha} + \omega^{\text{BitRev}_n(j)(q-1)/n+n/2^\alpha \cdot k} \in I_{\alpha,j} = \left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(j)(q-1)/n+n/2^\alpha \cdot k} \right).$$

Therefore the sum

$$\begin{aligned} & x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} - x^{n/2^\alpha} + \omega^{\text{BitRev}_n(j)(q-1)/n+n/2^\alpha \cdot k} \\ &= \omega^{\text{BitRev}_n(j)(q-1)/n+n/2^\alpha \cdot k} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \end{aligned} \quad (14)$$

is contained in $I_{\alpha,i} + I_{\alpha,j}$. If Expression (14) is non-zero, it is a unit of $\mathbb{Z}_q[x]$ and therefore $I_{\alpha,i} + I_{\alpha,j} = \mathbb{Z}_q[x]$.

Since $i \leq 2^\alpha - 1$, the first $\log_2(n) - \alpha$ digits of $\log_2(n)$ digits in the binary expansion of i are 0, i.e.

$$i = (0 \dots 0 b_{\alpha-1} b_{\alpha-2} \dots b_2 b_1 b_0)_2,$$

where $b_{\alpha-1}, \dots, b_0 \in \{0, 1\}$. Consequently the binary expansion of $\text{BitRev}_n(i)$ is largest when its first α digits are 1 and its remaining $\log_2(n) - \alpha$ digits are 0, i.e.

$$\text{BitRev}_n(i) \leq (\underbrace{1 \dots 1}_\alpha 0 \dots 0)_2 = n - 1 - (2^{\log_2(n) - \alpha} - 1) = n - 2^{\log_2(n) - \alpha}.$$

Note that $k < (q-1)/n$ and thus $n/2^\alpha \cdot k < (q-1)/2^\alpha$. It follows that

$$\text{BitRev}_n(i)(q-1)/n + n/2^\alpha \cdot k < (n - 2^{\log_2(n) - \alpha})(q-1)/n + (q-1)/2^\alpha = q - 1.$$

Since also $j \leq 2^\alpha - 1$, we get with the same reasoning that

$$\text{BitRev}_n(j)(q-1)/n + n/2^\alpha \cdot k < q - 1.$$

It is easy to see that the bit reversal is a bijective function on the set of integers between 0 and $n-1$, whence $i \neq j$ implies $\text{BitRev}_n(i) \neq \text{BitRev}_n(j)$. Moreover, because $(q-1)/n \neq 0$, we conclude that

$$\text{BitRev}_n(i)(q-1)/n + n/2^\alpha \cdot k \neq \text{BitRev}_n(j)(q-1)/n + n/2^\alpha \cdot k.$$

Since ω is a generator of \mathbb{Z}_q^* and the exponents are distinct and both smaller than $q-1$, we have

$$\omega^{\text{BitRev}_n(j)(q-1)/n + n/2^\alpha \cdot k} \neq \omega^{\text{BitRev}_n(i)(q-1)/n + n/2^\alpha \cdot k}.$$

This means that Expression (14) is non-zero, whence $I_{\alpha,i}$ and $I_{\alpha,j}$ are coprime. \square

4.2 gNTT

We are now ready to describe the ring isomorphisms we use in the generalized NTT.

Proposition 4.2. *For any integer $0 \leq \alpha \leq \log_2(n)$ the mapping*

$$\begin{aligned} \psi_\alpha : \mathbb{Z}_q[x]/(x^n - \omega^{nk}) &\mapsto \prod_{i=0}^{2^\alpha-1} \mathbb{Z}_q[x]/(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n + n/2^\alpha \cdot k}), \\ \psi_\alpha \left(\sum_{j=0}^{n-1} f_j x^j \right) &= \left(\sum_{j=0}^{n-1} f_j x^j + (x^{n/2^\alpha} - \omega^{\text{BitRev}_n(0)(q-1)/n + n/2^\alpha \cdot k}), \dots, \right. \\ &\quad \left. \dots, \sum_{j=0}^{n-1} f_j x^j + (x^{n/2^\alpha} - \omega^{\text{BitRev}_n(2^\alpha-1)(q-1)/n + n/2^\alpha \cdot k}) \right), \end{aligned}$$

where \prod denotes the direct product, is a ring isomorphism.

Proof. Lemma 4.1 shows that

$$x^n - \omega^{nk} = \prod_{i=0}^{2^\alpha-1} (x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k})$$

and Lemma 4.3 that the ideals generated by the factors are pairwise coprime. We can therefore apply the Chinese Remainder Theorem for rings (Theorem 2.2), which proves the statement. \square

Consider the isomorphism in Proposition 4.2 for $\alpha = \log_2(n)$. It is given by

$$\begin{aligned} \psi_{\log_2(n)} : \mathbb{Z}_q[x]/(x^n - \omega^{nk}) &\mapsto \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \omega^{\text{BitRev}_n(i)(q-1)/n+k}), \\ \psi_{\log_2(n)} \left(\sum_{j=0}^{n-1} f_j x^j \right) &= \left(\sum_{j=0}^{n-1} f_j \omega^{j \cdot (\text{BitRev}_n(0)(q-1)/n+k)}, \dots, \right. \\ &\quad \left. \dots, \sum_{j=0}^{n-1} f_j \omega^{j \cdot (\text{BitRev}_n(n-1)(q-1)/n+k)} \right). \end{aligned}$$

and we will call it the generalized NTT.

Definition 4.1 (Generalized NTT). We define the *generalized NTT* as the isomorphism $\psi_{\log_2(n)}$ from

$$\mathbb{Z}_q[x]/(x^n - \omega^{nk}) \quad \text{to} \quad \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \omega^{\text{BitRev}_n(i)(q-1)/n+k}),$$

which is described in Proposition 4.2. We denote $\text{gNTT} := \psi_{\log_2(n)}$.

The generalized NTT can be efficiently computed using the factorization given in Lemma 4.2.

Lemma 4.4. *Let $0 \leq \alpha \leq \log_2(n) - 1$ and $0 \leq i \leq 2^\alpha - 1$ be integers. The mapping*

$$\begin{aligned} \theta_{\alpha,i} : \mathbb{Z}_q[x]/(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k}) \\ \mapsto \left(\mathbb{Z}_q[x]/(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k}) \right. \\ \left. \times \mathbb{Z}_q[x]/(x^{n/2^{\alpha+1}} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k}) \right) \end{aligned}$$

given by

$$\begin{aligned} \theta_{\alpha,i} \left(\sum_{j=0}^{n/2^\alpha-1} f_j x^j \right) &= \left(\sum_{j=0}^{n/2^{\alpha+1}-1} \left(f_j + f_{j+n/2^{\alpha+1}} \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) x^j, \right. \\ &\quad \left. \sum_{j=0}^{n/2^{\alpha+1}-1} \left(f_j - f_{j+n/2^{\alpha+1}} \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) x^j \right) \end{aligned}$$

is a ring isomorphism.

Proof. Due to Equation (11) from Lemma 4.2 and Lemma 4.3 we can apply the Chinese Remainder Theorem for rings (Theorem 2.2) to obtain the desired result. \square

Proposition 4.3. *Let $0 \leq \alpha \leq \log_2(n) - 1$ be an integer. Let ψ_α and $\theta_{\alpha,i}$ be defined as in Proposition 4.2 and Lemma 4.4. Define θ_α as the mapping from the direct product*

$$\prod_{i=0}^{2^\alpha-1} \mathbb{Z}_q[x] / \left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right)$$

to the direct product

$$\prod_{i=0}^{2^{\alpha+1}-1} \mathbb{Z}_q[x] / \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right)$$

given by

$$\begin{aligned} \theta_\alpha & \left(\sum_{j=0}^{n/2^\alpha-1} f_{j,0} x^j, \dots, \sum_{j=0}^{n/2^\alpha-1} f_{j,2^\alpha-1} x^j \right) \\ & = \left(\theta_{\alpha,0} \left(\sum_{j=0}^{n/2^\alpha-1} f_{j,0} x^j \right), \dots, \theta_{\alpha,2^\alpha-1} \left(\sum_{j=0}^{n/2^\alpha-1} f_{j,2^\alpha-1} x^j \right) \right) \end{aligned}$$

This mapping is an isomorphism. Moreover, we have

$$\psi_{\alpha+1} = \theta_\alpha \circ \psi_\alpha.$$

It follows by induction that

$$\psi_{\alpha+1} = \theta_\alpha \circ \theta_{\alpha-1} \circ \dots \circ \theta_1 \circ \theta_0 \quad \text{and} \quad \text{gNTT} = \theta_{\log_2(n)-1} \circ \theta_{\log_2(n)-2} \circ \dots \circ \theta_1 \circ \theta_0.$$

Proof. The mapping θ_α is an isomorphism since all the mappings $\theta_{\alpha,i}$ from the components of the direct product are isomorphisms due to Lemma 4.4. Moreover, let $0 \leq i \leq 2^\alpha - 1$ be an integer. The isomorphism $\theta_{\alpha,i}$ is mapping every element

$$\sum_{j=0}^{n-1} f_j x^j + \left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right)$$

to

$$\left(\sum_{j=0}^{n-1} f_j x^j + \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right), \right. \\ \left. \sum_{j=0}^{n-1} f_j x^j + \left(x^{n/2^{\alpha+1}} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) \right).$$

We know from the proof of Lemma 4.1 that

$$\omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} = -\omega^{\text{BitRev}_n(2i+1)(q-1)/n+n/2^{\alpha+1} \cdot k}.$$

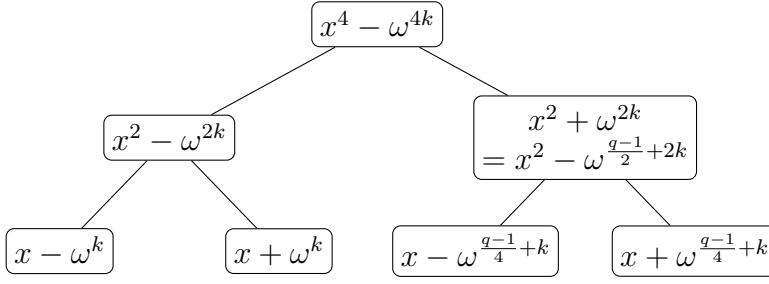


Figure 4.2: Illustration of the steps that split the polynomial $x^4 - \omega^{4k}$ into linear factors. The identity $\omega^{(q-1)/2} = -1 \in \mathbb{Z}_q$ is used.

Hence

$$\begin{aligned}
 (\theta_\alpha \circ \psi_\alpha) \left(\sum_{j=0}^{n-1} f_j x^j \right) &= \left(\sum_{j=0}^{n-1} f_j x^j + \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(0)(q-1)/n+n/2^{\alpha+1} \cdot k} \right), \dots, \right. \\
 &\quad \left. \dots, \sum_{j=0}^{n-1} f_j x^j + \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2^{\alpha+1}-1)(q-1)/n+n/2^{\alpha} \cdot k} \right) \right)
 \end{aligned}$$

which is equal to

$$\psi_{\alpha+1} \left(\sum_{j=0}^{n-1} f_j x^j \right).$$

Using induction and the definition of gNTT yields the remaining statements. \square

Example 4.1. Let $n = 4$ and $q = 29$. Then $n \mid (q - 1)$. We can choose $k = 3$ since $3 < (q - 1)/n = 7$. The element $\omega = 2$ is a generator of \mathbb{Z}_{29}^* . Consider the polynomial

$$f(x) = 3 + 23x + 18x^2 + 7x^3 \in \mathbb{Z}_{29}[x]/(x^4 - 2^{4 \cdot 3}) = \mathbb{Z}_{29}[x]/(x^4 - 7).$$

We will compute $\text{gNTT}(f(x))$. Figure 4.2 shows how the polynomial $x^4 - 2^{4 \cdot 3}$ splits. Note that $2^{2 \cdot 3} = 6 \pmod{29}$ and $18 \cdot 6 = 21 \pmod{29}$ as well as $7 \cdot 6 = 13 \pmod{29}$. Hence we get in the first step

$$\begin{aligned}
 \theta_0(f(x)) &= ((3 + 21) + (23 + 13) \cdot x, (3 - 21) + (23 - 13) \cdot x) \\
 &= (24 + 7x, 11 + 10x).
 \end{aligned}$$

Since $2^3 = 8 \pmod{29}$ and $2^{28/4+3} = 2^{10} = 9 \pmod{29}$, we compute $7 \cdot 8 = 27 \pmod{29}$ and $10 \cdot 9 = 3 \pmod{29}$. In the second and last step we therefore get

$$\begin{aligned}
 \text{gNTT}(f(x)) &= \theta_1(\theta_0(f(x))) = \theta_1(24 + 7x, 11 + 10x) \\
 &= (24 + 27, 24 - 27, 11 + 3, 11 - 3) \\
 &= (22, 26, 14, 8).
 \end{aligned}$$

4.3 gINTT

Definition 4.2 (Generalized INTT). The *generalized INTT* is the inverse of the generalized NTT. We denote it by gINTT.

Due to Proposition 4.3 we have

$$\begin{aligned} \text{gINTT} &= \text{gNTT}^{-1} = (\theta_{\log_2(n)-1} \circ \theta_{\log_2(n)-2} \circ \dots \circ \theta_1 \circ \theta_0)^{-1} \\ &= \theta_0^{-1} \circ \theta_1^{-1} \circ \dots \circ \theta_{\log_2(n)-2}^{-1} \circ \theta_{\log_2(n)-1}^{-1}. \end{aligned}$$

This shows that the generalized INTT can be computed step by step, like the generalized NTT. To invert θ_α , one needs to invert $\theta_{\alpha,i}$ for every $0 \leq i \leq 2^\alpha - 1$. Proposition 4.4 shows how to do that.

Proposition 4.4. *Let $0 \leq \alpha \leq \log_2(n) - 1$ and $0 \leq i \leq 2^\alpha - 1$ be integers. The inverse of $\theta_{\alpha,i}$ as defined in Lemma 4.4 is given by*

$$\begin{aligned} \theta_{\alpha,i}^{-1} &: \left(\mathbb{Z}_q[x] / \left(x^{n/2^{\alpha+1}} - \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) \right. \\ &\quad \times \left. \mathbb{Z}_q[x] / \left(x^{n/2^{\alpha+1}} + \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} \right) \right) \\ &\mapsto \mathbb{Z}_q[x] / \left(x^{n/2^\alpha} - \omega^{\text{BitRev}_n(i)(q-1)/n+n/2^\alpha \cdot k} \right), \\ \theta_{\alpha,i}^{-1} &\left(\sum_{j=0}^{n/2^{\alpha+1}-1} g_j x^j, \sum_{j=0}^{n/2^{\alpha+1}-1} h_j x^j \right) \\ &= 2^{-1} \left(\sum_{j=0}^{n/2^{\alpha+1}-1} (g_j + h_j) x^j \right. \\ &\quad \left. + \omega^{q-1-\text{BitRev}_n(2i)(q-1)/n-n/2^{\alpha+1} \cdot k} \sum_{j=0}^{n/2^{\alpha+1}-1} (g_j - h_j) x^{j+n/2^{\alpha+1}} \right). \end{aligned}$$

Proof. We regard

$$\left(\sum_{j=0}^{n/2^{\alpha+1}-1} g_j x^j, \sum_{j=0}^{n/2^{\alpha+1}-1} h_j x^j \right)$$

as the image under $\theta_{\alpha,i}$ of some element $\sum_{j=0}^{n/2^{\alpha+1}-1} f_j x^j$. For every $0 \leq j \leq n/2^{\alpha+1} - 1$ we have

$$g_j = f_j + f_{j+n/2^{\alpha+1}} \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k}$$

and

$$h_j = f_j - f_{j+n/2^{\alpha+1}} \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k}.$$

Therefore $g_j + h_j = 2 \cdot f_j$. Moreover,

$$\omega^{q-1-\text{BitRev}_n(2i)(q-1)/n-n/2^{\alpha+1} \cdot k} \cdot \omega^{\text{BitRev}_n(2i)(q-1)/n+n/2^{\alpha+1} \cdot k} = \omega^{q-1} = 1,$$

whence

$$\omega^{q-1-\text{BitRev}_n(2i)(q-1)/n-n/2^{\alpha+1} \cdot k} \cdot (g_j - h_j) = 2 \cdot f_{j+n/2^{\alpha+1}}$$

and

$$\theta_{\alpha,i}^{-1} \left(\sum_{j=0}^{n/2^{\alpha+1}-1} g_j x^j, \sum_{j=0}^{n/2^{\alpha+1}-1} h_j x^j \right) = \sum_{j=0}^{n/2^\alpha-1} f_j x^j. \quad \square$$

Remark 4.1. All θ_α^{-1} are ring isomorphisms, thus for any element z_α in the domain of θ_α it holds that $\theta_\alpha(2 \cdot z_\alpha) = 2 \cdot \theta_\alpha(z_\alpha)$. Instead of multiplying by 2^{-1} in every step, one can therefore just multiply by $(2^{\log_2(n)})^{-1} = n^{-1}$ in the last step. This speeds up the computation of gINTT by avoiding multiplications.

Example 4.2. *As in Example 4.1 let $n = 4$, $q = 29$, $\omega = 2$ and $k = 3$. Consider the element*

$$\begin{aligned} \hat{f} &= (22, 26, 14, 8) \in \left(\mathbb{Z}_{29}[x]/(x - \omega^k) \times \mathbb{Z}_{29}[x]/(x + \omega^k) \right. \\ &\quad \left. \times \mathbb{Z}_{29}[x]/(x - \omega^{(q-1)/4+k}) \times \mathbb{Z}_{29}[x]/(x + \omega^{(q-1)/4+k}) \right) \\ &= \left(\mathbb{Z}_{29}[x]/(x - 8) \times \mathbb{Z}_{29}[x]/(x + 8) \right. \\ &\quad \left. \times \mathbb{Z}_{29}[x]/(x - 9) \times \mathbb{Z}_{29}[x]/(x + 9) \right). \end{aligned}$$

We will compute $\text{gINTT}(\hat{f})$. Following Proposition 4.4, we first compute $\theta_{1,0}^{-1}(22, 26)$. Note that $\omega^{q-1-k} = \omega^{25} = 11 \pmod{29}$. Hence

$$\theta_{1,0}^{-1}(22, 26) = 2^{-1}(22 + 26 + 11(22 - 26)x) = 2^{-1}(19 + 14x).$$

Similarly we have $\omega^{q-1-(q-1)/4-k} = \omega^{18} = 13 \pmod{29}$ and

$$\theta_{1,1}^{-1}(14, 8) = 2^{-1}(14 + 8 + 13(14 - 8)x) = 2^{-1}(22 + 20x).$$

For the next step note that $\omega^{q-1-2k} = \omega^{22} = 5 \pmod{29}$. Therefore

$$\begin{aligned} \theta_0^{-1}(2^{-1}(19 + 14x), 2^{-1}(22 + 20x)) &= 4^{-1} \left(19 + 22 + (14 + 20)x \right. \\ &\quad \left. + 5((19 - 22)x^2 + (14 - 20)x^3) \right) \\ &= 4^{-1} (12 + 5x + 14x^2 + 28x^3) \\ &= 22 (12 + 5x + 14x^2 + 28x^3) \\ &= 3 + 23x + 18x^2 + 7x^3, \end{aligned}$$

which is the polynomial $f(x)$ from Example 4.1, i.e. $\text{gINTT}(\text{gNTT}(f(x))) = f(x)$.

4.4 Using gNTT to multiply polynomials

As in the case of the classical NTT, the generalized NTT is used to speed up the multiplication of polynomials in rings of the form $\mathbb{Z}_q[x]/(x^n - \omega^{nk})$. This is done by computing the actual multiplication in the isomorphic ring

$$\prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \omega^{\text{BitRev}_n(i)(q-1)/n+k}).$$

In this ring every element is a tuple of integers modulo q , whence the multiplication is done component-wise modulo q .

Example 4.3. As in Examples 4.1 and 4.2, let $n = 4$, $q = 29$, $\omega = 2$ and $k = 3$. We want to compute the product of

$$f(x) = 3 + 23x + 18x^2 + 7x^3 \quad \text{and} \quad g(x) = 16 + 2x + 25x^2 + 6x^3$$

in $\mathbb{Z}_{29}[x]/(x^4 - 7)$. We have seen in Example 4.1 that $\text{gNTT}(f(x)) = (22, 26, 14, 8)$. Computing the generalized NTT of $g(x)$ in the same way yields

$$\text{gNTT}(g(x)) = (6, 7, 24, 27).$$

Denoting component-wise multiplication by \odot , we get

$$\text{gNTT}(f(x)) \odot \text{gNTT}(g(x)) = (22 \cdot 6, 26 \cdot 7, 14 \cdot 24, 8 \cdot 27) = (16, 8, 17, 13).$$

Applying the generalized INTT yields

$$\text{gINTT}(16, 8, 17, 13) = (28, 6, 7, 16),$$

the product of $f(x)$ and $g(x)$.

4.5 Computational complexity

The following results show the numbers of multiplications modulo q that are required to compute the product of two polynomials using the generalized NTT and schoolbook multiplication, respectively. We assume that the powers of ω and n^{-1} , which we need for for gNTT and gINTT have been precomputed before.

Proposition 4.5. Given any polynomial $f(x) \in \mathbb{Z}_q[x]/(x^n - \omega^{nk})$, we can compute $\text{gNTT}(f(x))$ using

$$\frac{n}{2} \log_2(n) \quad \text{multiplications modulo } q. \quad (15)$$

Given any element $\hat{f} \in \prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \omega^{\text{BitRev}_n(i)(q-1)/n+k})$, we can compute $\text{gINTT}(\hat{f})$ using

$$\frac{n}{2} \log_2(n) + n \quad \text{multiplications modulo } q. \quad (16)$$

Given two polynomials $f(x), g(x) \in \mathbb{Z}_q[x]/(x^n - \omega^{nk})$, we can compute their product using

$$\frac{3}{2} n \log_2(n) + 2n \quad \text{multiplications modulo } q. \quad (17)$$

Proof. Consider the definitions of $\theta_{\alpha,i}$ and θ_α in Lemma 4.4 and Proposition 4.3. We need $n/2^{\alpha+1}$ multiplications modulo q to compute $\theta_{\alpha,i}$. Computing θ_α thus requires $2^\alpha \cdot n/2^{\alpha+1} = n/2$ multiplications modulo q . Statement (15) follows since gNTT is the composition of $\log_2(n) - 1$ mappings θ_α according to Proposition 4.3.

Looking at Proposition 4.4, we see that computing $\theta_{\alpha,i}$ requires $n/2^{\alpha+1}$ multiplications modulo q if we omit the multiplication by 2^{-1} according to Remark 4.1. Multiplying the polynomial by n^{-1} in the last step obviously requires n multiplications modulo q . It follows with the same reasoning as in the case of gNTT that gINTT can be computed using $n/2 \cdot \log_2(n) + n$ multiplications modulo q .

To compute the product of $f(x)$ and $g(x)$ we first compute $\text{gNTT}(f(x))$ and $\text{gNTT}(g(x))$, which requires $n \log_2(n)$ multiplications modulo q . Computing the product in $\prod_{i=0}^{n-1} \mathbb{Z}_q[x]/(x - \omega^{\text{BitRev}_n(i)(q-1)/n+k})$ requires n multiplications and computing gINTT of that product requires another $n/2 \cdot \log_2(n) + n$ multiplications modulo q . Hence in total we need $3/2 \cdot n \log_2(n) + 2n$ multiplications modulo q . \square

Proposition 4.6. *Let q be a prime and $c \in \mathbb{Z}_q$. Computing the product of two polynomials $f(x), g(x) \in \mathbb{Z}_q[x]/(x^n - c)$ using schoolbook multiplication generally requires $n^2 + n - 1$ multiplications modulo q . If $c \in \{-1, 0, 1\}$, then it requires n^2 multiplications modulo q .*

Proof. We want to compute the product of

$$f(x) = \sum_{i=0}^{n-1} f_i x^i \quad \text{and} \quad g(x) = \sum_{i=0}^{n-1} g_i x^i$$

via schoolbook multiplication. We have to multiply every coefficient f_i by every coefficient g_j to get a summand of the coefficient of x^{i+j} . Since both polynomials have n coefficients, this makes a total of n^2 multiplications modulo q . Moreover, for $i + j \geq n$, one has to multiply the coefficient of x^{i+j} by c . Since $i + j \leq 2n - 2$, this requires another $n - 1$ multiplications modulo q . If, however, $c \in \{-1, 0, 1\}$, one can skip those multiplications and simply replace x^n by $-1, 0$ or 1 , respectively. \square

5 Methods to weaken condition on prime

In the case of the classical NTT for cyclic convolution, we need to have $n \mid (q - 1)$, i.e. $q \equiv 1 \pmod{n}$. For negacyclic convolution, the even stronger condition $q \equiv 1 \pmod{2n}$ is required. In Section 6.2 of [LZ22] a method called Pt-NTT is described which weakens this condition. Using the Karatsuba trick leads to an improvement of this method called K-NTT.

In the case of our generalized NTT, we also require $n \mid (q - 1) \iff q \equiv 1 \pmod{n}$. It is straightforward to use Pt-NTT and K-NTT with our generalized NTT.

The central part of both these methods is the isomorphism described in the following proposition.

Proposition 5.1. *Let q be a prime, $c \in \mathbb{Z}_q^*$, n be a positive integer and β a non-negative integer such that n is divisible by 2^β . The mapping*

$$\phi_\beta : \mathbb{Z}_q[x]/(x^n - c) \mapsto (\mathbb{Z}_q[y]/(y^{n/2^\beta} - c))[x]/(x^{2^\beta} - y),$$

$$\phi_\beta \left(\sum_{i=0}^{n-1} f_i x^i \right) = \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j \right) x^i$$

is a ring isomorphism. Its inverse is given by

$$\phi_\beta^{-1} \left(\sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} \hat{f}_{i,j} y^j \right) x^i \right) = \sum_{j=0}^{n/2^\beta-1} \sum_{i=0}^{2^\beta-1} \hat{f}_{i,j} x^{2^\beta j+i}.$$

Proof. See Appendix A. □

Example 5.1 illustrates how ϕ_β reorders the coefficients of the polynomial using the identity $x^{2^\beta} = y$.

Example 5.1. *Let $\beta = 1$, $n = 8$, q be a prime and $c \in \mathbb{Z}_q^*$. We describe how to map the polynomial $f = \sum_{i=0}^7 f_i x^i \in \mathbb{Z}_q[x]/(x^8 - c)$ to $(\mathbb{Z}_q[y]/(y^4 - c))[x]/(x^2 - y)$ via the ring isomorphism ϕ_1 from Proposition 5.1.*

In the first step we replace x^2 by y in f whenever it is possible. We have

$$\begin{array}{cccccccc} f = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + f_4 x^4 + f_5 x^5 + f_6 x^6 + f_7 x^7 \\ \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \\ f_0 + f_1 x + f_2 y + f_3 y x + f_4 y^2 + f_5 y^2 x + f_6 y^3 + f_7 y^3 x. \end{array}$$

In the second step we rearrange those terms by the powers of x and obtain

$$(f_0 + f_2 y + f_4 y^2 + f_6 y^3) x^0 + (f_1 + f_3 y + f_5 y^2 + f_7 y^3) x^1 = \phi_1(f),$$

the corresponding element in $(\mathbb{Z}_q[y]/(y^4 - c))[x]/(x^2 - y)$.

Let n be a power of 2, β an integer $0 \leq \beta \leq \log_2(n)$ and q a prime such that $(n/2^\beta) \mid (q - 1)$. Moreover, let k be an integer such that $0 \leq k < (q - 1)/(n/2^\beta)$.

5.1 Pt-NTT

The following method is called Preprocess-then-NTT (Pt-NTT) and was proposed in [Zho+18]. Let ϕ_β be the ring isomorphism of Proposition 5.1. Given two polynomials

$$f = \sum_{i=0}^{n-1} f_i x^i, g = \sum_{i=0}^{n-1} g_i x^i \in \mathbb{Z}_q[x]/(x^n - \omega^{n/2^\beta \cdot k}),$$

we will compute their product in the isomorphic ring

$$(\mathbb{Z}_q[y]/(y^{n/2^\beta} - \omega^{n/2^\beta \cdot k}))[x]/(x^{2^\beta} - y).$$

For any integer $0 \leq i \leq 2^\beta - 1$, we have

$$f'_i := (\phi_\beta(f))_i = \sum_{j=0}^{n/2^\beta - 1} f_{2^\beta j + i} y^j.$$

Since $x^{2^\beta} = y$, the product of $\phi_\beta(f)$ and $\phi_\beta(g)$ is given by

$$\phi_\beta(f) \cdot \phi_\beta(g) = \sum_{i=0}^{2^\beta - 1} h'_i x^i, \text{ where } h'_i = \sum_{k=0}^i f'_k g'_{i-k} + \sum_{k=i+1}^{2^\beta - 1} y f'_k g'_{i+2^\beta - k}.$$

The multiplications and summations to compute h'_i are done in the ring

$$\mathbb{Z}_q[y]/(y^{n/2^\beta} - \omega^{n/2^\beta \cdot k}),$$

which is isomorphic to

$$\prod_{i=0}^{n/2^\beta - 1} \mathbb{Z}_q[x]/\left(x - \omega^{\text{BitRev}_{n/2^\beta}(i)(q-1)/(n/2^\beta) + k}\right)$$

via our generalized NTT. We can therefore compute

$$h'_i = \text{gINTT} \left(\sum_{k=0}^i \text{gNTT}(f'_k) \odot \text{gNTT}(g'_{i-k}) + \sum_{k=i+1}^{2^\beta - 1} \text{gNTT}(y f'_k) \odot \text{gNTT}(g'_{i+2^\beta - k}) \right). \quad (18)$$

Since $y^{n/2^\beta} = \omega^{n/2^\beta \cdot k}$, we have

$$y f'_k = y \sum_{j=0}^{n/2^\beta - 1} f_{2^\beta j + k} y^j = \sum_{j=0}^{n/2^\beta - 1} f_{2^\beta j + k} y^{j+1} = \omega^{n/2^\beta \cdot k} f_{n-2^\beta + k} + \sum_{j=0}^{n/2^\beta - 2} f_{2^\beta j + k} y^{j+1}.$$

The final step to get the product of f and g is to compute

$$\phi_\beta^{-1} \left(\sum_{i=0}^{2^\beta - 1} h'_i x^i \right),$$

which means to replace y by x^{2^β} .

Example 5.2. Let $n = 8$ and $\beta = 1$, whence $n/2^\beta = 4$. Moreover, let q be a prime such that $4 \mid (q - 1)$, ω be a generator of \mathbb{Z}_q^* and $0 \leq k \leq (q - 1)/4$. We compute the product of two polynomials

$$f = \sum_{i=0}^7 f_i x^i, \quad g = \sum_{i=0}^7 g_i x^i \in \mathbb{Z}_q[x]/(x^8 - \omega^{4k}).$$

Following the computation of ϕ_1 in Example 5.1 we get

$$\begin{aligned} f'_0 &= f_0 + f_2 y + f_4 y^2 + f_6 y^3, & f'_1 &= f_1 + f_3 y + f_5 y^2 + f_7 y^3 \\ \text{and } g'_0 &= g_0 + g_2 y + g_4 y^2 + g_6 y^3, & g'_1 &= g_1 + g_3 y + g_5 y^2 + g_7 y^3. \end{aligned}$$

Moreover, we have

$$h'_0 = f'_0 g'_0 + y f'_1 g'_1 \quad \text{and} \quad h'_1 = f'_0 g'_1 + f'_1 g'_0.$$

Since $4 \mid (q - 1)$, we can apply the generalized NTT to the polynomials

$$f'_i, g'_i \in \mathbb{Z}_q[y]/(y^4 - \omega^{4k}),$$

such that h'_0 is given by

$$\begin{aligned} &\text{gINTT} \left(\text{gNTT} (f_0 + f_2 y + f_4 y^2 + f_6 y^3) \odot \text{gNTT} (g_0 + g_2 y + g_4 y^2 + g_6 y^3) \right. \\ &\quad \left. + \text{gNTT} (\omega^{4k} f_7 + f_1 y + f_3 y^2 + f_5 y^3) \odot \text{gNTT} (g_1 + g_3 y + g_5 y^2 + g_7 y^3) \right) \end{aligned}$$

and h'_1 is given by

$$\begin{aligned} &\text{gINTT} \left(\text{gNTT} (f_0 + f_2 y + f_4 y^2 + f_6 y^3) \odot \text{gNTT} (g_1 + g_3 y + g_5 y^2 + g_7 y^3) \right. \\ &\quad \left. + \text{gNTT} (f_1 + f_3 y + f_5 y^2 + f_7 y^3) \odot \text{gNTT} (g_0 + g_2 y + g_4 y^2 + g_6 y^3) \right). \end{aligned}$$

Once we have computed

$$h'_0 = h'_{0,0} + h'_{0,1} y + h'_{0,2} y^2 + h'_{0,3} y^3 \quad \text{and} \quad h'_1 = h'_{1,0} + h'_{1,1} y + h'_{1,2} y^2 + h'_{1,3} y^3,$$

we just need to replace y by x^2 in $h'_0 + h'_1 x$ to get the product of f and g , i.e.

$$f \cdot g = h'_{0,0} + h'_{1,0} x + h'_{0,1} x^2 + h'_{1,1} x^3 + h'_{0,2} x^4 + h'_{1,2} x^5 + h'_{0,3} x^6 + h'_{1,3} x^7.$$

5.2 K-NTT

An improvement of Pt-NTT called K-NTT was proposed in [ZLP21]. It uses the Karatsuba trick (Definition 2.4) to avoid some component-wise multiplications in the computation of h'_i that we described in Equation (18). Moreover, $\text{gNTT}(y)$ is precomputed.

Note that if i is odd, then h'_i is given by

$$\text{gINTT} \left(\sum_{k=0}^{\lfloor i/2 \rfloor} \left(\text{gNTT}(f'_k) \odot \text{gNTT}(g'_{i-k}) + \text{gNTT}(f'_{i-k}) \odot \text{gNTT}(g'_k) \right) \right) \quad (19)$$

$$+ \text{gNTT}(y) \odot \sum_{k=i+1}^{\lfloor (2^\beta+i)/2 \rfloor} \left(\text{gNTT}(f'_k) \odot \text{gNTT}(g'_{i+2^\beta-k}) \right) \quad (20)$$

$$+ \text{gNTT}(f'_{i+2^\beta-k}) \odot \text{gNTT}(g'_k) \Big). \quad (21)$$

In the case of even i the formula for h'_i is slightly different. The upper limit of the sum in Row (19) is then given by $i/2 - 1$ and the term $\text{gNTT}(f'_{i/2}) \odot \text{gNTT}(g'_{i/2})$ is added to the sum in this row. Furthermore the upper limit of the sum in Row (20) is given by $(2^\beta + i)/2 - 1$ and the term $\text{gNTT}(f'_{(2^\beta+i)/2}) \odot \text{gNTT}(g'_{(2^\beta+i)/2})$ is added to the sum which gets multiplied by $\text{gNTT}(y)$.

One first computes $\text{gNTT}(f'_i) \odot \text{gNTT}(g'_i)$ for all integers $0 \leq i \leq 2^\beta - 1$. The Karatsuba trick is then used to compute the term

$$\text{gNTT}(f'_k) \odot \text{gNTT}(g'_{i-k}) + \text{gNTT}(f'_{i-k}) \odot \text{gNTT}(g'_k)$$

in Row (19) as

$$\begin{aligned} & (\text{gNTT}(f'_k) + \text{gNTT}(f'_{i-k})) \odot (\text{gNTT}(g'_k) + \text{gNTT}(g'_{i-k})) \\ & - \text{gNTT}(f'_k) \odot \text{gNTT}(g'_k) - \text{gNTT}(f'_{i-k}) \odot \text{gNTT}(g'_{i-k}) \end{aligned}$$

and the term

$$\text{gNTT}(f'_k) \odot \text{gNTT}(g'_{i+2^\beta-k}) + \text{gNTT}(f'_{i+2^\beta-k}) \odot \text{gNTT}(g'_k)$$

in Rows (20) and (21) as

$$\begin{aligned} & (\text{gNTT}(f'_k) + \text{gNTT}(f'_{i+2^\beta-k})) \odot (\text{gNTT}(g'_k) + \text{gNTT}(g'_{i+2^\beta-k})) \\ & - \text{gNTT}(f'_k) \odot \text{gNTT}(g'_k) - \text{gNTT}(f'_{i+2^\beta-k}) \odot \text{gNTT}(g'_{i+2^\beta-k}). \end{aligned}$$

Apart from the changes we just mentioned, K-NTT works exactly like Pt-NTT.

Example 5.3. *Consider Example 5.2 again. We demonstrate what changes if we use K-NTT instead of Pt-NTT.*

First of all, $\text{gNTT}(y)$ is precomputed when implementing K-NTT. Remember that

$$h'_0 = f'_0 g'_0 + y f'_1 g'_1 \quad \text{and} \quad h'_1 = f'_0 g'_1 + f'_1 g'_0.$$

We compute and store

$$\text{gNTT}(f'_0) \odot \text{gNTT}(g'_0) \quad \text{and} \quad \text{gNTT}(f'_1) \odot \text{gNTT}(g'_1).$$

Finally, h'_0 is computed as

$$\text{gINTT}\left(\text{gNTT}(f'_0) \odot \text{gNTT}(g'_0) + \text{gNTT}(y) \odot (\text{gNTT}(f'_1) \odot \text{gNTT}(g'_1))\right)$$

and h'_1 is computed as

$$\begin{aligned} & \text{gINTT}\left((\text{gNTT}(f'_0) + \text{gNTT}(f'_1)) \odot (\text{gNTT}(g'_0) + \text{gNTT}(g'_1)) \right. \\ & \left. - \text{gNTT}(f'_0) \odot \text{gNTT}(g'_0) - \text{gNTT}(f'_1) \odot \text{gNTT}(g'_1)\right). \end{aligned}$$

6 Experiments

In Section 4.5 we established some theoretical results about the computational complexity of schoolbook multiplication and polynomial multiplication using our generalized NTT. In this section we present an empirical comparison of the speed of different multiplication methods in a Python implementation.

6.1 Implementation

All the code needed for the experiments we conducted in Python can be found in Appendix B. We implemented schoolbook multiplication, our generalized NTT, Pt-NTT (using our generalized NTT) and K-NTT (using our generalized NTT) in Python. We represent the polynomials as `numpy` arrays and do the computations on them. The package `time` is used to measure the time the different multiplication methods take. To display the results nicely, we use `tabulate`.

We generalized Algorithms 41, 42 and 43 of [Nat24a] to any n that is a power of 2. The main difference between the implementation of the NTT as described in [Nat24a] and the implementation of our generalized NTT is that we need to precompute two arrays of powers of the generator instead of one. As described in Section 3.2.2, in the case of negacyclic convolution, one can use the same array of powers of the generator for both the NTT and the INTT. In our case we need to precompute one array for the gNTT and one array for the gINTT. Moreover, we precompute the multiplicative inverse of n modulo the prime q .

The methods Pt-NTT and K-NTT call our implementation of gNTT and gINTT as a subroutine. For K-NTT we also precompute $\text{gNTT}(y)$.

For the timing, we first generate m random pairs of polynomials in $\mathbb{Z}_q[x]/(x^n - \omega^{nk})$, and then measure for each multiplication method how long it takes to compute the m multiplications. We use `time.process_time_ns`, which returns the value in nanoseconds of the sum of the system and user CPU time of the current process (see [Fou]).

6.2 Results

The parameters n and q for which we compare the speed of the different methods are taken from [Nat24b] and [Nat24a]. Remember that Pt-NTT and K-NTT can be used even when $n \nmid (q - 1)$. In our experiments, however, we only investigate cases where $n \mid (q - 1)$. This enables us to use the generalized NTT and directly compare the speed of polynomial multiplication via the generalized NTT, Pt-NTT and K-NTT.

6.2.1 Parameters from Dilithium

In [Nat24a] the parameters are $q = 8380417$ and $n = 256$. We select the generator $\omega = 10$. The randomly chosen $k = 8317$ yields then $\omega^{nk} = 3812918$, so we multiply polynomials in $\mathbb{Z}_q[x]/(x^n - 3812918)$. For Pt-NTT and K-NTT we set $\beta = 1$. Figure 6.1 shows the experimental results for 10,000 multiplications. We see that schoolbook multiplication is by far the slowest method, taking about eight times as long as the other methods. The differences between the generalized NTT, Pt-NTT and K-NTT are marginal and might not be of statistical significance.

Mult. method	Total time in ns	Time per mult. in ns	Methods ratio
Schoolbook Mult.	262718750000	2.62719e+07	1
Generalized NTT	32718750000	3.27188e+06	0.124539
Pt-NTT	33890625000	3.38906e+06	0.129
K-NTT	32890625000	3.28906e+06	0.125193

Figure 6.1: Total time for 10,000 multiplications in nanoseconds, average time per multiplication in nanoseconds and the ratio between the multiplication method and the slowest multiplication method for schoolbook multiplication, our generalized NTT, Pt-NTT and K-NTT. The parameters are $q = 8380417$, $n = 256$, $\omega = 10$, $k = 8317$ and $\beta = 1$ (for Pt-NTT and K-NTT).

6.2.2 Parameters from Kyber

In [Nat24b] the parameters are $q = 3329$ and $n = 256$. We select the generator $\omega = 3$. The randomly chosen $k = 7$ yields then $\omega^{nk} = 2764$, so we multiply polynomials in $\mathbb{Z}_q[x]/(x^n - 2764)$. For the Pt-NTT and K-NTT we set $\beta = 1$. Figure 6.2 shows the experimental results for 10,000 multiplications. We see that schoolbook multiplication is by far the slowest method, taking about eight times as long as the other methods. The Pt-NTT is slightly slower than the generalized NTT, while the K-NTT is slightly faster.

Mult. method	Total time in ns	Time per mult. in ns	Methods ratio
Schoolbook Mult.	267781250000	2.67781e+07	1
Generalized NTT	33593750000	3.35938e+06	0.125452
Pt-NTT	34875000000	3.4875e+06	0.130237
K-NTT	30531250000	3.05312e+06	0.114016

Figure 6.2: Total time for 10,000 multiplications in nanoseconds, average time per multiplication in nanoseconds and the ratio between the multiplication method and the slowest multiplication method for schoolbook multiplication, our generalized NTT, Pt-NTT and K-NTT. The parameters are $q = 3329$, $n = 256$, $\omega = 3$, $k = 7$ and $\beta = 1$ (for Pt-NTT and K-NTT).

7 Conclusion

We have shown how the Number Theoretic Transform can be generalized from the rings $\mathbb{Z}_q[x]/(x^n \pm 1)$ to any ring $\mathbb{Z}_q[x]/(x^n - \omega^{nk})$, where ω is a generator of \mathbb{Z}_q and k an integer such that $0 \leq k < (q-1)/n$.

Moreover, we have shown how the methods Pt-NTT and K-NTT can be applied to the generalized NTT to weaken the condition on the prime q to $(n/2^\beta) \mid (q-1)$.

Finally the experiments in Section 6 confirm a significant speedup of using the generalized NTT to multiply polynomials in $\mathbb{Z}_q[x]/(x^{256} - \omega^{256 \cdot k})$ compared to school-book multiplication. They further verify that Pt-NTT and K-NTT with $\beta = 1$ perform similarly to the generalized NTT.

References

- [AB75] R.C. Agarwal and C.S. Burrus. “Number theoretic transforms to implement fast digital convolution”. In: *Proceedings of the IEEE* 63.4 (1975), pp. 550–560. DOI: 10.1109/PROC.1975.9791.
- [CT65] James W. Cooley and John W. Tukey. “An Algorithm for the Machine Calculation of Complex Fourier Series”. In: *Mathematics of Computation* 19 (1965), pp. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1.
- [Fou] Python Software Foundation. *time — Time access and conversions*. URL: <https://docs.python.org/3/library/time.html> (visited on Apr. 29, 2026).
- [GS66] W. M. Gentleman and G. Sande. “Fast Fourier Transforms: for fun and profit”. In: *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference. AFIPS '66 (Fall)*. San Francisco, California: Association for Computing Machinery, 1966, pp. 563–578. ISBN: 9781450378932. DOI: 10.1145/1464291.1464352.
- [Hun74] Thomas W. Hungerford. *Algebra*. Graduate texts in mathematics, 73. New York: Springer, 1974. ISBN: 0387905189.
- [Kru26] David Krumm. *A Course in Ring Theory*. 2026. arXiv: 2512.22133 [math.RA]. URL: <https://arxiv.org/abs/2512.22133> (visited on Apr. 29, 2026).
- [Lan02] Serge Lang. *Algebra*. Rev. 3. ed. Graduate texts in mathematics 211. New York: Springer, 2002. ISBN: 9780387953854.
- [LPR12] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *On Ideal Lattices and Learning with Errors Over Rings*. Cryptology ePrint Archive, Paper 2012/230. 2012. URL: <https://eprint.iacr.org/2012/230> (visited on Apr. 29, 2026).
- [LZ22] Zhichuang Liang and Yunlei Zhao. *Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey*. 2022. arXiv: 2211.13546 [cs.CR]. URL: <https://arxiv.org/abs/2211.13546> (visited on Apr. 29, 2026).
- [Nat24a] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard*. Federal Information Processing Standards Publication (FIPS) 204. Washington, D.C.: U.S. Department of Commerce, Aug. 2024. URL: <https://doi.org/10.6028/NIST.FIPS.204> (visited on Dec. 6, 2025).
- [Nat24b] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Federal Information Processing Standards Publication (FIPS) 203. Washington, D.C.: U.S. Department of Commerce, Aug. 2024. URL: <https://doi.org/10.6028/NIST.FIPS.203> (visited on Dec. 6, 2025).
- [Sat+23] Ardianto Satriawan et al. “Conceptual Review on Number Theoretic Transform and Comprehensive Review on Its Implementations”. In: *IEEE Access* 11 (2023), pp. 70288–70316. DOI: 10.1109/ACCESS.2023.3294446.

- [SML24] Ardianto Satriawan, Rella Mareta, and Hanho Lee. *A Complete Beginner Guide to the Number Theoretic Transform (NTT)*. Cryptology ePrint Archive, Paper 2024/585. 2024. URL: <https://eprint.iacr.org/2024/585> (visited on Apr. 29, 2026).
- [Zho+18] Shuai Zhou et al. *Preprocess-then-NTT Technique and Its Applications to KYBER and NEWHOPE*. Cryptology ePrint Archive, Paper 2018/995. 2018. URL: <https://eprint.iacr.org/2018/995> (visited on Apr. 29, 2026).
- [ZLP21] Yiming Zhu, Zhen Liu, and Yanbin Pan. “When NTT Meets Karatsuba: Preprocess-then-NTT Technique Revisited”. In: *Information and Communications Security*. Ed. by Debin Gao et al. Cham: Springer International Publishing, 2021, pp. 249–264. ISBN: 978-3-030-88052-1.

A Proofs

Proposition 5.1. *Let q be a prime, $c \in \mathbb{Z}_q^*$, n be a positive integer and β a non-negative integer such that n is divisible by 2^β . The mapping*

$$\phi_\beta : \mathbb{Z}_q[x]/(x^n - c) \mapsto (\mathbb{Z}_q[y]/(y^{n/2^\beta} - c))[x]/(x^{2^\beta} - y),$$

$$\phi_\beta \left(\sum_{i=0}^{n-1} f_i x^i \right) = \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j \right) x^i$$

is a ring isomorphism. Its inverse is given by

$$\phi_\beta^{-1} \left(\sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} \hat{f}_{i,j} y^j \right) x^i \right) = \sum_{j=0}^{n/2^\beta-1} \sum_{i=0}^{2^\beta-1} \hat{f}_{i,j} x^{2^\beta j+i}.$$

Proof. Clearly $\phi_\beta(1) = 1$. Let $f = \sum_{i=0}^{n-1} f_i x^i$ and $g = \sum_{i=0}^{n-1} g_i x^i$ be elements of $\mathbb{Z}_q[x]/(x^n - c)$. We define

$$\tilde{f}_i := \sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j, \quad \tilde{g}_i := \sum_{j=0}^{n/2^\beta-1} g_{2^\beta j+i} y^j.$$

We first show that $\phi_\beta(f) + \phi_\beta(g) = \phi_\beta(f + g)$. Adding the coefficients of the polynomial in x and then the coefficients of the polynomials in y yields

$$\begin{aligned} \phi_\beta(f) + \phi_\beta(g) &= \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j \right) x^i + \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} g_{2^\beta j+i} y^j \right) x^i \\ &= \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j + \sum_{j=0}^{n/2^\beta-1} g_{2^\beta j+i} y^j \right) x^i \\ &= \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} (f_{2^\beta j+i} + g_{2^\beta j+i}) y^j \right) x^i \\ &= \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} (f + g)_{2^\beta j+i} y^j \right) x^i = \phi_\beta(f + g). \end{aligned}$$

Next we show that $\phi_\beta(f \cdot g) = \phi_\beta(f) \cdot \phi_\beta(g)$. In $\mathbb{Z}_q[x]/(x^n - c)$ we get

$$f \cdot g = \sum_{i=0}^{n-1} \left(\sum_{l=0}^i f_l g_{i-l} + c \sum_{l=i+1}^{n-1} f_l g_{n+i-l} \right) x^i$$

and it follows by the definition of ϕ_β that

$$\phi_\beta(f \cdot g) = \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} \left(\sum_{l=0}^{2^\beta j+i} f_l g_{2^\beta j+i-l} + c \sum_{l=2^\beta j+i+1}^{n-1} f_l g_{2^\beta j+n+i-l} \right) y^j \right) x^i.$$

Denoting the coefficient of y^j of x^i of $\phi_\beta(f \cdot g)$ by $(\phi_\beta(f \cdot g))_{i,j}$ yields

$$(\phi_\beta(f \cdot g))_{i,j} = \sum_{l=0}^{2^\beta j+i} f_l g_{2^\beta j+i-l} + c \sum_{l=2^\beta j+i+1}^{n-1} f_l g_{2^\beta j+n+i-l}. \quad (22)$$

We use the fact that $x^{2^\beta} = y$ in $\mathbb{Z}_q[y]/(y^{n/2^\beta} - c)[x]/(x^{2^\beta} - y)$ to write the coefficient of x^i of $\phi_\beta(f) \cdot \phi_\beta(g)$ as

$$\begin{aligned} (\phi_\beta(f) \cdot \phi_\beta(g))_i &= \left(\left(\sum_{i=0}^{2^\beta-1} \tilde{f}_i x^i \right) \left(\sum_{i=0}^{2^\beta-1} \tilde{g}_i x^i \right) \right)_i \\ &= \sum_{l=0}^i \tilde{f}_l \tilde{g}_{i-l} + y \sum_{l=i+1}^{2^\beta-1} \tilde{f}_l \tilde{g}_{i+2^\beta-l} \\ &= \sum_{l=0}^i \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+l} y^j \right) \left(\sum_{j=0}^{n/2^\beta-1} g_{2^\beta j+i-l} y^j \right) \end{aligned} \quad (23)$$

$$+ y \sum_{l=i+1}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+l} y^j \right) \left(\sum_{j=0}^{n/2^\beta-1} g_{2^\beta j+i+2^\beta-l} y^j \right). \quad (24)$$

Using the identity $y^{n/2^\beta} = c$ we can rewrite Expression (23) as

$$\sum_{l=0}^i \sum_{j=0}^{n/2^\beta-1} \left(\sum_{h=0}^j f_{2^\beta h+l} g_{2^\beta(j-h)+i-l} + c \sum_{h=j+1}^{n/2^\beta-1} f_{2^\beta h+l} g_{2^\beta(j+n/2^\beta-h)+i-l} \right) y^j \quad (25)$$

and Expression (24) as

$$\sum_{l=i+1}^{2^\beta-1} \sum_{j=0}^{n/2^\beta-1} \left(\sum_{h=0}^j f_{2^\beta h+l} g_{2^\beta(j-h)+i+2^\beta-l} + c \sum_{h=j+1}^{n/2^\beta-1} f_{2^\beta h+l} g_{2^\beta(j+n/2^\beta-h)+i+2^\beta-l} \right) y^{j+1}. \quad (26)$$

We denote the constant term of $(\phi_\beta(f) \cdot \phi_\beta(g))_i$ by $(\phi_\beta(f) \cdot \phi_\beta(g))_{i,0}$. To compute it we have to sum up the coefficients of y^0 in the Expressions (25) and (26). Using the identity $y^{n/2^\beta} = c$ again we get

$$\begin{aligned} (\phi_\beta(f) \cdot \phi_\beta(g))_{i,0} &= \sum_{l=0}^i \left(f_l g_{i-l} + c \sum_{h=1}^{n/2^\beta-1} f_{2^\beta h+l} g_{n+i-(2^\beta h+l)} \right) \\ &\quad + c \sum_{l=i+1}^{2^\beta-1} \sum_{h=0}^{n/2^\beta-1} f_{2^\beta h+l} g_{n+i-(2^\beta h+l)} \\ &= \sum_{l=0}^i f_l g_{i-l} + c \sum_{h=1}^{n/2^\beta-1} \sum_{l=0}^{2^\beta-1} f_{2^\beta h+l} g_{n+i-(2^\beta h+l)} + c \sum_{l=i+1}^{2^\beta-1} f_l g_{n+i-l} \\ &= \sum_{l=0}^i f_l g_{i-l} + c \sum_{k=2^\beta}^{n-1} f_k g_{n+i-k} + c \sum_{l=i+1}^{2^\beta-1} f_l g_{n+i-l} \\ &= \sum_{l=0}^i f_l g_{i-l} + c \sum_{l=i+1}^{n-1} f_l g_{n+i-l}. \end{aligned} \quad (27)$$

For $j = 0$ the right-hand side of Equation (22) is equal to Expression (27), whence

$$(\phi_\beta(f \cdot g))_{i,0} = (\phi_\beta(f) \cdot \phi_\beta(g))_{i,0}. \quad (28)$$

Let now $j \geq 1$. To compute the coefficient of y^j of x^i of $\phi_\beta(f) \cdot \phi_\beta(g)$ we have to sum up the coefficients of y^j in Expressions (25) and (26). We have

$$\begin{aligned} & (\phi_\beta(f) \cdot \phi_\beta(g))_{i,j} \\ &= \sum_{l=0}^i \left(\sum_{h=0}^j f_{2^\beta h+l} g_{2^\beta j+i-(2^\beta h+l)} + c \sum_{h=j+1}^{n/2^\beta-1} f_{2^\beta h+l} g_{2^\beta j+n+i-(2^\beta h+l)} \right) \\ & \quad + \sum_{l=i+1}^{2^\beta-1} \left(\sum_{h=0}^{j-1} f_{2^\beta h+l} g_{2^\beta j+i-(2^\beta h+l)} + c \sum_{h=j}^{n/2^\beta-1} f_{2^\beta h+l} g_{2^\beta j+n+i-(2^\beta h+l)} \right) \\ &= \sum_{h=0}^{j-1} \sum_{l=0}^{2^\beta-1} f_{2^\beta h+l} g_{2^\beta j+i-(2^\beta h+l)} + \sum_{l=0}^i f_{2^\beta j+l} g_{2^\beta j+i-(2^\beta j+l)} \\ & \quad + c \left(\sum_{h=j+1}^{n/2^\beta-1} \sum_{l=0}^{2^\beta-1} f_{2^\beta h+l} g_{2^\beta j+n+i-(2^\beta h+l)} + \sum_{l=i+1}^{2^\beta-1} f_{2^\beta j+l} g_{2^\beta j+n+i-(2^\beta j+l)} \right) \\ &= \sum_{k=0}^{2^\beta j-1} f_k g_{2^\beta j+i-k} + \sum_{k=2^\beta j}^{2^\beta j+i} f_k g_{2^\beta j+i-k} \\ & \quad + c \left(\sum_{k=2^\beta j+2^\beta}^{n-1} f_k g_{2^\beta j+n+i-k} + \sum_{k=2^\beta j+i+1}^{2^\beta j+2^\beta-1} f_k g_{2^\beta j+n+i-k} \right) \\ &= \sum_{l=0}^{2^\beta j+i} f_l g_{2^\beta j+i-l} + c \sum_{l=2^\beta j+i+1}^{n-1} f_l g_{2^\beta j+n+i-l}. \quad (29) \end{aligned}$$

Expression (29) is equal to the right-hand side of Equation (22), whence

$$(\phi_\beta(f \cdot g))_{i,j} = (\phi_\beta(f) \cdot \phi_\beta(g))_{i,j} \quad (30)$$

for $j \geq 1$. Since i was arbitrary, it follows from Equations (28) and (30) that

$$\phi_\beta(f \cdot g) = \phi_\beta(f) \cdot \phi_\beta(g).$$

This establishes that ϕ_β is a ring homomorphism. It remains to show that ϕ_β is bijective.

Assume that

$$\phi_\beta(f) = \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j \right) x^i = 0$$

in $(\mathbb{Z}_q[y]/(y^{n/2^\beta} - c))[x]/(x^{2^\beta} - y)$. It follows that

$$\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j = 0$$

for all $i \in \{0, \dots, 2^\beta - 1\}$. This implies that $f_{2^\beta j+i} = 0$ for all $j \in \{0, \dots, n/2^\beta - 1\}$ and all $i \in \{0, \dots, 2^\beta - 1\}$. Consequently $f_k = 0$ for all $k \in \{0, \dots, n - 1\}$ which means that $f = 0$. Hence $\ker \phi_\beta = 0$ and therefore ϕ_β is injective.

Let now

$$\hat{s} = \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} \hat{s}_{i,j} y^j \right) x^i$$

be any element of $(\mathbb{Z}_q[y]/(y^{n/2^\beta} - c))[x]/(x^{2^\beta} - y)$. We define

$$\mathbb{Z}_q[x]/(x^n - c) \ni s := \sum_{j=0}^{n/2^\beta-1} \sum_{i=0}^{2^\beta-1} \hat{s}_{i,j} x^{2^\beta j+i}$$

and get

$$\phi_\beta(s) = \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} s_{2^\beta j+i} y^j \right) x^i = \sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} \hat{s}_{i,j} y^j \right) x^i = \hat{s},$$

where $s_{2^\beta j+i}$ is the coefficient of $x^{2^\beta j+i}$ of s . This shows the surjectivity and finishes the proof that ϕ_β is a ring isomorphism.

For any polynomial $f = \sum_{i=0}^{n-1} f_i x^i \in \mathbb{Z}_q[x]/(x^n - c)$ we get

$$\begin{aligned} \phi_\beta^{-1}(\phi_\beta(f)) &= \phi_\beta^{-1} \left(\sum_{i=0}^{2^\beta-1} \left(\sum_{j=0}^{n/2^\beta-1} f_{2^\beta j+i} y^j \right) x^i \right) \\ &= \sum_{j=0}^{n/2^\beta-1} \sum_{i=0}^{2^\beta-1} f_{2^\beta j+i} x^{2^\beta j+i} = \sum_{i=0}^{n-1} f_i x^i = f, \end{aligned}$$

which shows that ϕ_β^{-1} is the inverse of ϕ_β . □

B Python code

B.1 Packages

```
import numpy as np
import time
from tabulate import tabulate
```

B.2 Preliminary functions

```
# Compute the power of a^n modulo m
def fast2Power(a,n,m):
    res = 1
    while n>0:
        if n%2 == 1: # If the bit is 1 multiply by the corresponding square
            res = (res*a) % m
        a = (a*a) % m
        n = n//2
    return res
```

```

# Output [r,s,t] satisfying s*a+t*b=r=gcd(a,b)
def extendedGCD(a,b):
    r0,r = a,b
    s0,s = 1,0
    t0,t = 0,1
    while (r>0):
        tempr,temps,tempt = r,s,t
        q = r0 // r
        r,s,t = r0-q*r,s0-q*s,t0-q*t
        r0,s0,t0 = tempr,temps,tempt
    return [r0,s0,t0]

```

```

# Output the multiplicative inverse of a modulo p
def multInverse(a,p):
    result = extendedGCD(a,p)
    if result [0]!=1: # Error message if a and p are not relatively prime
        s = " Numbers need to be relatively prime "
        return s
    inv = result [1]% p
    return inv

```

```

# finds a generator w of  $\mathbb{Z}_q^*$  given the prime factors of q-1 as a list
def findGenerator(q,primeList):
    for w in range(2,q-1): # possible generators
        flag = False
        for p in primeList:
            if fast2Power(w,(q-1)//p,q) == 1:
                flag = True
                break
        if flag == False:
            return w

```

```

#takes as input the integer x and the power alpha
#returns a list of length alpha corresponding to the bit representation of x mod  $2^\alpha$ 
def integerToBits(x,alpha):
    result_list = []
    for i in range(alpha):
        result_list.append(x % 2)
        x = x // 2
    return result_list

```

```

# takes as input a list of bits and the alpha
# returns the corresponding integer
def bitsToInteger(bitList,alpha):
    x = 0
    for i in range(1,alpha+1):
        x = 2*x + bitList[alpha-i]
    return x

```

```

# outputs the number (in decimal base) that one gets when reversing the  $\log_2(n)$ -bit binary
expansion of m
def bitRev(n,m):
    alpha = n.bit_length() - 1 #  $2^\alpha = n$ ;  $\alpha = \log_2(n)$ 
    bitList = integerToBits(m,alpha)
    bitReversed = [0]*(alpha)
    for i in range(alpha):
        bitReversed[i] = bitList[(alpha-1)-i]
    r = bitsToInteger(bitReversed,alpha)
    return r

```

```

# generates a random polynomial of degree at most n-1 with coefficients in  $\mathbb{Z}_q$ 
def genRandPoly(n,q):
    rng = np.random.default_rng()
    f = rng.integers(low=0,high=q,size=n)
    return f

```

```

# input is the number m of random polynomials to be generated, n the degree of the polynomials and
q the prime

```

```

# return a numpy array that contains m pairs of randomly generated polynomials of length n
def randPolyArray(m,n,q):
    arr = np.zeros(shape=(m,2,n),dtype=int)

    for instance in arr:
        instance[0] = genRandPoly(n,q)
        instance[1] = genRandPoly(n,q)

    return arr

```

B.3 Schoolbook Multiplication

```

# input are two polynomials f and g of degree at most n-1 with coefficients in Z_q
# the algorithm calculates the product c = f*g in Z_q[x]/(x^n-a)
# f and g are numpy arrays and the resulting output is also a numpy array

def schoolMult(f,g,n,q,a):
    c = np.zeros(n,dtype=int)
    for k in range(n):
        c_k = 0
        c_k_temp = 0 # the sum that is multiplied by a since the power of x is at least n-1
        for i in range(0,k+1):
            c_k = (c_k + f[i]*g[k-i]) % q
        for i in range(k+1,n):
            c_k_temp = (c_k_temp + ((f[i]*g[k+n-i])%q)) % q
        c_k_temp = (a*c_k_temp) % q
        c[k] = (c_k + c_k_temp) % q
    return c

```

B.4 Generalized NTT

```

# computes the array with the powers of w and the array with the powers of w for the INTT
# notice the ordering of the array for the INTT
# w is a generator of Z_q*
# n is power of 2, q is a prime with n|(q-1) and k is the exponent such that w^(n*k) = a
# output is a tuple NTT_array,INTTarray

def compute_arrays(w,n,q,k):
    # the fraction we will always use in the exponents
    fraction = (q-1) // n

    # arrays with the powers of w; length n because 0 at position 0 for NTT algorithm
    NTTarray = np.zeros(n,dtype=int)
    INTTarray = np.zeros(n,dtype=int)

    # the last vertical layer from top, it is log2(n) since we start with 1
    last_layer = n.bit_length() - 1

    # indicates the position in the NTTarray and INTTarray which have to be filled next
    array_position = 1
    INTTarray_position = 1

    for layer_from_top in range(1,last_layer+1):
        #horizontal length of current layer
        layer_length = 2**(layer_from_top-1)
        # k is multiplied with the same integers in the whole vertical layer
        k_value = bitRev(n,2**(layer_from_top-1))*k
        # setting the INTTarray_position to the rightmost position in the current vertical layer
        INTTarray_position = array_position + layer_length - 1

        for horizontal_position in range(2**(layer_from_top-1)):
            # the exponents of w for the NTT_array
            exp_NTT = (bitRev(n,2*horizontal_position)*fraction + k_value) % (q-1)
            # adding (q-1)//2 in the exponent of w leads to a multiplication by -1
            exp_INTT = ((q-1)//2 - exp_NTT) % (q-1)

            NTTarray[array_position] = fast2Power(w,exp_NTT,q)
            INTTarray[INTTarray_position] = fast2Power(w,exp_INTT,q)
            array_position += 1

```

```

        # moving towards the left in the vertical layer
        INTTarray_position -= 1

    return NTTarray,INTTarray

```

```

# n is as usual the dimension (a power of 2), q the prime,
# f is the polynomial in  $Z_q[x]/(x^n-w^{(n*k)})$  represented by a list or array of length n
# w_array is the array created by w_array(w,n,q,k)
# computes the NTT transform defined by the w_array

```

```

def NTT(f,n,q,NTT_array):
    m = 0
    length = n // 2
    while length >= 1:
        start = 0
        while start < n:
            m += 1
            z = NTT_array[m]
            for j in range(start,start+length):
                t = (z*f[j+length]) % q
                f[j+length] = (f[j]-t) % q
                f[j] = (f[j]+t) % q
            start = start + 2*length
        length //= 2
    return f

```

```

# v is a numpy array
# n is as usual the dimension (a power of 2), q the prime
# INTTarray is the array with the powers of w that we need for the inverseNTT (created by
    compute_arrays(w,n,q,k))
# n_inv is the multiplicative inverse of n modulo q

```

```

def INTT(v,n,q,n_inv,INTTarray):
    m = n
    length = 1
    while length < n:
        start = 0
        while start < n:
            m -= 1
            z = -INTTarray[m]
            for j in range(start,start+length):
                t = v[j]
                v[j] = (t+v[j+length]) % q
                v[j+length] = (t-v[j+length]) % q
                v[j+length] = (z*v[j+length]) % q
            start = start + 2*length
        length *= 2
    v = (n_inv*v) % q
    return v

```

```

# computes the sum of v and u
# v and u are given in NTT form (as numpy arrays)
# the output is their sum in NTT form (as a numpy array)
# q is the prime as usual

```

```

def addNTT(v,u,q):
    return (v+u) % q

```

```

# computes the component-wise product of v and u
# v and u are given in NTT form (as numpy arrays)
# the output is their product in NTT form (as a numpy array)
# q is the prime as usual

```

```

def multiplyNTT(v,u,q):
    return (v*u) % q

```

```

# polynomial multiplication using NTT

```

```

def polyMultbyNTT(f,g,n,q,n_inv,NTTarray,INTTarray):
    return INTT(multiplyNTT(NTT(f,n,q,NTTarray),NTT(g,n,q,NTTarray),q),n,q,n_inv,INTTarray)

```

B.5 Pt-NTT

```

# q is a prime and n is a power of 2
# f and g are polynomials in  $Z_q[x]/(x^n-a)$ 
# alpha is a non-negative integer such that full NTT is possible in  $(Z_q[y]/(y^{(n/(2^\alpha))-a}))$ ,
# i.e.  $a = w^{(n/(2^\alpha)*k)}$  for some generator w of  $Z_q^*$  and  $(n/(2^\alpha))|(q-1)$ 
# columns_inv is the multiplicative inverse of  $(n/(2^\alpha))$  modulo q
# NTTarray and INTTarray are the arrays created by the function compute_arrays(w,n/(2^\alpha),q,k)

def polyMultbyPtNTT(f,g,n,q,a,alpha,columns_inv,NTTarray,INTTarray):
    # the rows are representing polynomials in  $Z_q[y]/(y^{(n/(2^\alpha))-c})$ 
    rows = 2**alpha
    columns = n//rows
    f = np.reshape(f,shape=(rows,columns),order='F')
    g = np.reshape(g,shape=(rows,columns),order='F')

    # variable for computing the product
    prod = np.zeros(shape=(rows,columns),dtype=int)

    # computing  $y*f_l$ ; it is only needed for  $l \geq 1$ 
    f_hat = np.roll(f,shift=1,axis=1)
    f_hat[1:,0] = (f_hat[1:,0]* a) % q
    #print(f_hat)

    # computing all the necessary NTTs and storing them in the same variables
    # the NTTs are computed in the ring  $Z_q[y]/(y^{(n/(2^\alpha))-c})$ 
    f = np.array([NTT(matrix_row,columns,q,NTTarray) for matrix_row in f])
    g = np.array([NTT(matrix_row,columns,q,NTTarray) for matrix_row in g])

    if alpha > 0: # otherwise f_hat has only one row
        f_hat[1:] = np.array([NTT(matrix_row,columns,q,NTTarray) for matrix_row in f_hat[1:]])

    # computing the entries of prod in  $Z_q[y]/(y^{(n/(2^\alpha))-c})$ 
    for i in range(rows):
        for l in range(i+1):
            prod[i] = addNTT(prod[i],multiplyNTT(f[l],g[i-l],q),q)
        for l in range(i+1,rows):
            prod[i] = addNTT(prod[i],multiplyNTT(f_hat[l],g[rows+i-l],q),q)

        prod[i] = INTT(prod[i],columns,q,columns_inv,INTTarray)

    # rewriting the result as a polynomial in x (vector instead of array)
    prod = np.reshape(prod,shape=-1,order='F')

    return prod

```

B.6 K-NTT

```

# q is a prime and n is a power of 2
# f and g are polynomials in  $Z_q[x]/(x^n-a)$ 
# alpha is a non-negative integer such that full NTT is possible in  $Z_q[y]/(y^{(n/(2^\alpha))-a})$ ,
# i.e.  $a = w^{(n/(2^\alpha)*k)}$  for some generator w of  $Z_q^*$  and  $(n/(2^\alpha))|(q-1)$ 
# columns_inv is the multiplicative inverse of  $(n/(2^\alpha))$  modulo q
# NTTarray and INTTarray are the arrays created by the function compute_arrays(w,n/(2^\alpha),q,k)
# NTTy is the precomputed NTT of y, where y is in  $Z_q[y]/(y^{(n/(2^\alpha))-a})$  (using the same
# NTTarray)

def polyMultbykNTT(f,g,n,q,a,alpha,columns_inv,NTTarray,INTTarray,NTTy):
    # the rows are representing polynomials in  $Z_q[y]/(y^{(n/(2^\alpha))-a})$ 
    rows = 2**alpha
    columns = n//rows
    f = np.reshape(f,shape=(rows,columns),order='F')
    g = np.reshape(g,shape=(rows,columns),order='F')

    # variable for computing the product
    prod = np.zeros(shape=(rows,columns),dtype=int)

    # computing all the necessary NTTs and storing them in the same variables
    # the NTTs are computed in the ring  $Z_q[y]/(y^{(n/(2^\alpha))-c})$ 
    f = np.array([NTT(matrix_row,columns,q,NTTarray) for matrix_row in f])
    g = np.array([NTT(matrix_row,columns,q,NTTarray) for matrix_row in g])

```

```

# precomputing the products of the NTT; pointwise product of the i-th row of f with
# the i-th row of g for all rows i
precomp = (f*g) % q

# computing the entries of prod in  $Z_q[y]/(y^{n/(2^\alpha)}-a)$ 
# we first compute the entries of the product for rows with even indices
for i in range(0,rows,2):
    # l is lower index, i-l is higher index, we want to compute  $(f[l]*g[i-l]+f[i-l]*g[l])$ 
    # in one go using Karatsuba trick (the f[j] and g[j] are already in NTT form)
    for l in range(i//2):
        prod[i] = (addNTT(prod[i],multiplyNTT(addNTT(f[l],f[i-l],q),
            addNTT(g[l],g[i-l],q),q)-precomp[l]-precomp[i-l],q))
    # adding the term  $f[i/2]*g[i/2]$ 
    prod[i] = addNTT(prod[i],precomp[i//2],q)
    # temp_sum is the sum that is multiplied by NTT before adding it to prod[i]
    # adding the term  $f[l]*g[l]$  where  $l = 2^\alpha+i-l$  (which is  $(2^\alpha+i)/2$ ) to temp_sum
    # have to distinguish the case where alpha=0; then this sum is empty
    if alpha>0:
        temp_sum = precomp[(rows+i)//2]
    else:
        temp_sum = np.zeros(columns,dtype=int)
    for l in range(i+1,(rows+i)//2):
        temp_sum = (addNTT(temp_sum,multiplyNTT(addNTT(f[l],f[rows+i-l],q),
            addNTT(g[l],g[rows+i-l],q),q)-precomp[l]-precomp[rows+i-l],q))
    # adding NTT*temp_sum to prod[i] and applying the inverse NTT
    prod[i] = addNTT(prod[i],multiplyNTT(NTT,temp_sum,q),q)
    prod[i] = INTT(prod[i],columns,q,columns_inv,INTTarray)

# computing the entries of the product for rows with odd indices
for i in range(1,rows,2):
    # l is lower index, i-l is higher index, we want to compute  $(f[l]*g[i-l]+f[i-l]*g[l])$ 
    # in one go using Karatsuba trick (the f[j] and g[j] are already in NTT form)
    for l in range(i//2+1): # including l=i//2
        prod[i] = (addNTT(prod[i],multiplyNTT(addNTT(f[l],f[i-l],q),
            addNTT(g[l],g[i-l],q),q)-precomp[l]-precomp[i-l],q))
    # temp_sum is the sum that is multiplied by NTT before adding it to prod[i]
    temp_sum = np.zeros(columns,dtype=int)
    for l in range(i+1,(rows+i)//2+1):
        temp_sum = (addNTT(temp_sum,multiplyNTT(addNTT(f[l],f[rows+i-l],q),
            addNTT(g[l],g[rows+i-l],q),q)-precomp[l]-precomp[rows+i-l],q))
    # adding NTT*temp_sum to prod[i] and applying the inverse NTT
    prod[i] = addNTT(prod[i],multiplyNTT(NTT,temp_sum,q),q)
    prod[i] = INTT(prod[i],columns,q,columns_inv,INTTarray)

# rewriting the result as a polynomial in x (vector instead of array)
prod = np.reshape(prod,shape=-1,order='F')

return prod

```

B.7 Timing

```

# writing a function that should take as an input all multiplication methods that are to be
# compared in the form of
# a dictionary,
# the number m of polynomial multiplications, n and q,
# all necessary parameters for the multiplication methods that are compared in **kwargs

# the necessary **kwargs are:
# - a for schoolbook multiplication
# - w,full_k for generalized NTT
# - w,alpha,split_k for Pt-NTT
# - w,alpha,split_k for K-NTT

# the arrays for the NTT and the INTT are created inside the timing function

# if both generalized NTT and either Pt-NTT or K-NTT are computed, then split_k = full_k *
# (2^alpha) such that
# a = w^(n*full_k) = w^(n/(2^alpha)*split_k)

# output is a dictionary with the times

def timing(to_time,m,n,q,**kwargs):

```

```

# the array containing the m pairs of random polynomials and a hard copy of it for every
multiplication method
originalArr = randPolyArray(m,n,q)
copiedArr = {name:np.copy(originalArr) for name in to_time}

# an array to store the results
resultArr = {name:np.zeros(shape=(m,n),dtype=int) for name in to_time}

# the dictionary to store the resulting times
times = {}

for method in to_time:
    # doing the precomputations and creating the lambda function which will be used to measure
    the time
    if method == "Schoolbook Multiplication":
        function = lambda f,g: to_time[method](f,g,n,q,kwargs.get('a'))

    elif method == "Generalized NTT":
        # computing arrays and the parameter n_inv
        full_NT TArray,full_INT TArray = compute_arrays(kwargs.get('w'),n,q,kwargs.get('full_k'))
        n_inv = multInverse(n,q)

        function = lambda f,g: to_time[method](f,g,n,q,n_inv,full_NT TArray,full_INT TArray)

    elif method == "Pt-NTT":
        # unpacking kwargs
        w = kwargs.get('w')
        split_k = kwargs.get('split_k')
        alpha = kwargs.get('alpha')

        # computing parameters and arrays
        columns = n//(2**alpha)
        columns_inv = multInverse(columns,q)
        a = fast2Power(w,columns*split_k,q)
        split_NT TArray,split_INT TArray = compute_arrays(w,columns,q,split_k)

        function = lambda f,g:
to_time[method](f,g,n,q,a,alpha,columns_inv,split_NT TArray,split_INT TArray)

    elif method == "K-NTT":
        # unpacking kwargs
        w = kwargs.get('w')
        split_k = kwargs.get('split_k')
        alpha = kwargs.get('alpha')

        # computing parameters and arrays
        columns = n//(2**alpha)
        columns_inv = multInverse(columns,q)
        a = fast2Power(w,columns*split_k,q)
        split_NT TArray,split_INT TArray = compute_arrays(w,columns,q,split_k)

        # precomputing the NTT of y
        y = np.zeros(columns,dtype=int)
        y[1]=1
        NTTy = NTT(y,columns,q,split_NT TArray)

        function = lambda f,g:
to_time[method](f,g,n,q,a,alpha,columns_inv,split_NT TArray,split_INT TArray,NTTy)

    # time.process_time_ns() or time.perf_counter_ns()
    start_time = time.process_time_ns()

    for element in copiedArr[method]:
        element[0] = function(element[0],element[1])

    # same as in start_time
    end_time = time.process_time_ns()

    # computing results and adding to times dictionary
    total_time = end_time-start_time
    timePerMult = total_time / m

    times[method] = {"Total time in ns": total_time,"Time per multiplication in ns":
timePerMult}

```

```

    # filling the resultArr
    for i in range(m):
        resultArr[method][i] = copiedArr[method][i][0]

    # checking if all methods that we used yield the same results
    check = all(np.array_equal(val, list(resultArr.values())[0]) for val in resultArr.values())
    print("All methods yield the same result: ",check)

return times

```

B.8 Displaying results

```

# takes as input a dictionary which contains the results of the timing(...) function
# outputs a table (for printing in Python or Latex)
# tableformat can be changed to "latex", "latex_raw", "latex_booktabs" or "latex_longtable"

def print_results(results,tableformat="fancy_outline"):
    headers = ["Multiplication method","Total time in ns","Time per multiplication in ns","Method /
slowest method"]

    # reformatting the dictionary to a list
    table = []
    for element in results:
        table.append([element,results[element]["Total time in ns"],results[element]["Time per
multiplication in ns"]])

    # computing "Slowest method divided by current method"
    total_times = []
    for row in table:
        total_times.append(row[1])
    largest_time = max(total_times)
    for row in table:
        row.append(row[1]/largest_time)
    print(tabulate(table,headers,tablefmt=tableformat))

```