# Chapter 2

# Preliminaries

This chapter presents the preliminaries that are common among the following chapters of this thesis. Section 2.1 introduces the basic terminology of key-establishment and digital signature schemes. Section 2.2 presents polynomial-multiplication techniques with a focus on fast software implementations. Section 2.3 introduces algorithms for fast modular arithmetic, which is used as a building block of polynomial multiplication. Lastly, Section 2.4 presents the primary optimization platforms throughout this thesis: the Arm Cortex-M4 and Arm Cortex-M3.

## 2.1 Cryptographic Schemes

The NISTPQC competition seeks post-quantum replacements for the NIST standards for key-establishment schemes and digital signature schemes. The following sections introduce the terminology that is commonly used when describing those schemes. It discusses the security properties that are generally desired for such schemes. While these definitions are not limited to post-quantum cryptography, a scheme is called post-quantum in case the security properties still hold in case the adversary has a large-scale quantum computer.

### 2.1.1 Key-Establishment Schemes

The objective of key establishment is for two (or more) parties to agree on a shared secret key. Cryptographically this can be achieved in multiple different ways including either a PKE, a KEM, or a non-interactive key exchange (NIKE). Up until recently, post-quantum schemes were limited to either PKE or KEM. In 2018, after the deadline of the NIST project has passed, also post-quantum NIKE based on isogenies has been proposed [CLM+18].

However, as this came long after NIST called for proposals, NIST only allowed PKE and KEM submissions. As shown in the following, it is easy to construct a KEM from a PKE and vice versa.

A PKE scheme consists of three algorithms: `KeyGen`, `Encrypt`, and `Decrypt`:

**`KeyGen()`** takes no inputs and outputs a key pair consisting of a public key `pk` and a secret key `sk`.

**`Encrypt(pk, m)`** takes as inputs a public key `pk` and a message `m`, which is usually a bitstring. It outputs the encrypted message as ciphertext `c`.

**`Decrypt(sk, c)`** takes as inputs a secret key `sk` and a ciphertext `c`. It outputs the decrypted message `m'`.

A PKE scheme is correct if `Decrypt(sk,Encrypt(pk,m)) = m` for any key pair (`pk`, `sk`) produced by `KeyGen`() and any message `m`. It is called a deterministic PKE (DPKE) if `Encrypt` outputs the same ciphertext if called with the same inputs multiple times. Any non-deterministic PKE can be turned into a DPKE by slightly changing the API and making the randomness an explicit argument, e.g., by adding an argument `seed` that is used to derive all randomness pseudo-randomly.

A key-encapsulation mechanism (KEM) consists of three algorithms: `KeyGen`, `Encaps`, and `Decaps`:

**`KeyGen()`** takes no inputs and outputs a key pair consisting of a public key `pk` and a secret key `sk`.

**`Encaps(pk)`** takes as input a public key `pk` and returns a ciphertext `c` and a session key `ss`.

**`Decaps(sk, c)`** takes as input a secret key `sk` and a ciphertext `c`. It outputs a session key `ss'`.

A KEM is correct if given `c,ss` $\leftarrow$ `Encaps(pk)`, `Decaps(sk,c) = ss` for any key pair (`pk`, `sk`) produced by `KeyGen`(). To formalize the security of cryptographic schemes one usually describes the capabilities of the most powerful attack that a cryptographic scheme can resist. The most common security notion of PKE schemes and KEMs is indistinguishability under chosen-plaintext attacks (IND-CPA; CPA for short) and indistinguishability under chosen-ciphertext attacks (IND-CCA; CCA for short). Informally, these notations can be explained as follows. Indistinguishability refers to the inability of an attacker to reliably distinguish which of two ciphertexts corresponds to a given message. In the CPA setting, the adversary can encrypt as many messages as needed to obtain information from the ciphertexts that help to distinguish the given setting. This implies that a DPKE can never be CPA secure as the adversary can simply encrypt the given message. In

the CCA setting, the adversary is more powerful: He can actively choose ciphertexts (different from the target ciphertexts) and ask a decryption oracle to decrypt those. The information obtained from those queries must not help him to distinguish the ciphertexts. CPA attacks are also called passive attacks, while CCA attacks are called active attacks referring to the adversary actively tampering with some ciphertexts to obtain information about the secret key.

In practice, in most settings, the desired security notion is CCA security as it can often not be guaranteed that the adversary is unable to mount active attacks. For example, consider a public web server responding to requests. An adversary can send crafted ciphertexts to the web server that it will decrypt; the behavior of the web server often can be used to obtain some information about the secret key involved. Hence, it is usually only acceptable to use a CPA-secure scheme if each public key is only used once.

Luckily, there exist constructions that transform CPA-secure schemes into CCA-secure schemes. The most common one is the Fujisaki–Okamoto (FO) [FO99] transform which allows transforming a CPA-secure DPKE into a CCA-secure KEM. This approach is used by a large number of NISTPQC schemes including most of the ones covered in this thesis. Informally, the FO transform allows the decrypting party to verify that the ciphertext was honestly generated or if it was crafted. In case a dishonest ciphertext is detected, a random key is returned such that no information about the secret key is revealed. For a more formal description of the construction refer to [HHK17].

### 2.1.2 Digital Signature Schemes

A digital signature scheme consists of three algorithms: `KeyGen`, `Sign`, and `Open`:

**`KeyGen()`** takes no inputs and outputs a key pair consisting of a public key `pk` and a secret key `sk`.

**`Sign(sk, m)`** takes as inputs a secret key `sk` and a message `m` which is usually a bitstring of arbitrary length. It outputs a signed message `sm` commonly consisting of the concatenation of the message itself and a signature.

**`Open(pk, sm)`** takes as inputs a public key `pk` and a signed message `sm`. If the signature is valid under `pk`, it returns the message `m`. Otherwise, it returns an error.

A digital signature scheme is called correct if $\texttt{Open}(\texttt{pk}, \texttt{Sign}(\texttt{sk}, \texttt{m})) = \texttt{m}$ for any (`pk`, `sk`) produced by `KeyGen`() and any message `m`. There is an alternative definition of signature schemes where `Sign` returns a signature rather than a signed message, which is then verified using an algorithm

`Verify`. In that case, it is left to the user of the cryptographic scheme to decide what to do in case a signature is invalid. This can have catastrophic consequences as users (e.g., implementers using a cryptographic library) will often ignore such failure and proceed as usual. In the vast majority of cases, the sensible action to do is to discard the message and trigger some kind of error handling as the message cannot be trusted. This is the behavior enforced by the `Sign`/`Open` definition of signature schemes. NIST is following the `Sign`/`Open` definitions and requires the submitted software to adhere to the APIs reflecting them.

For a digital signature scheme to be secure, we require that it is impossible to produce a valid signed message `sm` without the knowledge of the secret key `sk`. The most common security notion for signature schemes is existential unforgeability under chosen message attacks (EU-CMA). Informally, it implies that an adversary is incapable to produce a signature for any message (existential forgery) even if he is allowed to obtain signatures for other messages, i.e., has access to a signing oracle.

## 2.2 Polynomial Multiplication for Computer Scientists

For thousands of years, humankind has studied how to efficiently multiply, either in anticipation of intriguing applications like cryptography or for its mere mathematical beauty. Hence, we can now choose from a plethora of multiplication methods that may or may not be useful for the problem at hand. Polynomial multiplication is a core building block for cryptography of all kinds. In particular, the vast majority of post-quantum schemes use polynomial multiplication. A closely related task is the multiplication of large integers which is used in yet another large number of cryptographic schemes.

Even more, multiplication methods are constantly adapted and optimized for a certain instance, such that any given implementation of polynomial multiplication contains dozens of tricks that have been discovered in decades or even centuries of research. This makes it incredibly hard to understand a given implementation without reading a huge number of papers. This is amplified by the fact that most tricks are simply part of all modern implementations, but rarely fully explained or attributed to the corresponding publications.

To date, I am not aware of any good comprehensive introduction to all polynomial multiplication methods used in state-of-the-art implementations of post-quantum cryptographic schemes. The renowned work by Bernstein [Ber01] is an excellent survey of numerous tricks and concisely presents them for a reader with experience in the underlying mathematical concepts. However, coming from a computer science background and lacking the nec-

essary mathematical background, it appears to be hardly decipherable. As I struggled for many years to conceive the tricks and I have seen many students struggle with it in the same way, I have decided to try to bridge this gap and write an introduction to polynomial multiplication for computer scientists.

In the following sections, I present every single way of multiplying polynomials that I have encountered in actual implementations of post-quantum cryptographic schemes over the years. Alongside the descriptions in this thesis, each section comes with sample implementations in Python and C. I aimed for this chapter to be brief, self-contained, and easy to follow. In parts, I skip over a lot background and formal definitions and instead provide examples that I hope provide a good intuition and serve as a starting point. For a more formal treatment, I recommend reading [vzGG13], [Nus82], and of course [Ber01].

The code is available at `https://github.com/mkannwischer/polymul` and can be freely used under a CC0 copyright waiver. All source code related to this thesis is also available in a single archive. See Appendix A.

**Notation.** Let $\mathbb{Z}$ be the ring of integers, and $\mathbb{Z}_q$ be the integer ring containing $\{0, ..., q-1\}$ with $q$ being a positive integer and arithmetic being performed modulo $q$. We write $\mathbb{Z}[x]$ to denote the polynomial ring in the variable $x$ with integer coefficients, and $\mathbb{Z}_q[x]$ to denote the polynomial ring with coefficients in $\mathbb{Z}_q$. Given a polynomial $a$ in some polynomial ring, we write $a_i$ to denote the coefficient corresponding to $x^i$, i.e, $a = \sum_{i=0}^{n-1} a_i x^i$. As these polynomial rings have an infinite number of elements, they are not particularly useful for cryptography. Hence, one uses a finite polynomial ring instead by computing modulo a certain polynomial $f(x)$ with degree $n$, such that polynomials remain at degree at most $n-1$, i.e., $n$ coefficients. We write $\mathbb{Z}_q[x]/(f(x))$ to denote the ring with all operations modulo $f(x)$ and $q$. Sometimes we require to split a polynomial $a$ into multiple parts by setting $y = x^k$ for some $k < n$. To denote the part $i$, we write $a^{(i)}$, such that $a = \sum_{i=0}^{\lceil n/k \rceil} a^{(i)} y^i$.

**Example 1:** Consider the polynomial ring $\mathbb{Z}_2[x]/(x^2+1)$. This polynomial ring has four elements: $\{0, 1, x, x+1\}$. We can now multiply two polynomials, e.g., $x \cdot (x+1) = x^2 + x \equiv x + 1 \pmod{q, x^2+1}$ as $x^2 \equiv -1 \pmod{q, x^2+1}$.

For cryptographic purposes, one uses polynomial rings with many more elements. In the following we are mostly considering rings that are used in cryptographic schemes based upon structured lattices, but all of the algorithms find application far beyond. A common choice for $q$ is either a power of two or a prime, commonly below 32 or 16 bits, such that one or two coefficients fit neatly into a processor word on all popular platforms. A common choice for $n$ is between 256 and 1024. For some polynomial multiplication

algorithms, $n$ is ideally a power of two as well, but that is not always the case in cryptographic schemes. A very common choice for $f(x)$ is $x^n + 1$ or $x^n - 1$ as the reduction is simple to implement and can often be done on the fly.

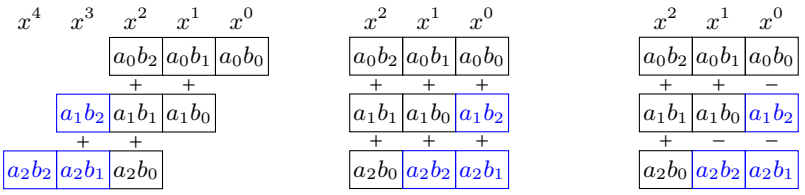
**Application:** Kyber [ABD+17] uses the ring $\mathbb{Z}_{3329}[x]/(x^{256} + 1)$, Saber [DKRV17] uses $\mathbb{Z}_{8192}[x]/(x^{256} + 1)$, NTRU [ZCH+19] uses (among others) $\mathbb{Z}_{8192}[x]/(x^{701} - 1)$, and Dilithium [LDK+17] uses $\mathbb{Z}_{8380417}[x]/(x^{256} + 1)$.

**Coefficient Multiplication.** Every polynomial-multiplication algorithm requires multiplying elements in $\mathbb{Z}_q$. Since $q$ is chosen such that coefficients fit into registers, multiplication can in most cases use the available multiplication instructions which multiply mod $2^k$ with $k \in 16, 32, 64$. If $q$ is a power of two, one can simply use the instruction $2^k > q$ and obtain the standard representative $(0, ..., q-1)$ using a logical AND with $q-1$. For intermediate values, the standard representative is usually not required, and one can omit the reductions. However, if $q$ is not a power of two, the multiplication needs to be performed mod $2^k > q^2$ and needs to be followed by an explicit reduction modulo $q$ to bring coefficients back to a single word. Furthermore, additions and subtractions of coefficients cause them to grow and one needs to be careful to reduce them before they overflow the word size. For reductions after multiplications, one commonly uses Montgomery reductions [Mon85], while for reductions after additions, one can use Montgomery reductions, Barrett reductions [Bar86], or specialized reductions for special primes, e.g., Solinas primes [Sol99]. For describing the polynomial-multiplication algorithms, we assume that modular multiplications in $\mathbb{Z}_q$ can be performed efficiently and cover the concrete algorithms separately in Section 2.3.

**Convolution.** In literature about polynomial multiplication one often also reads about convolution, positively wrapped (or cyclic) convolution, and negatively wrapped (or negacyclic) convolution. In general, convolution (written as $*$) of two functions $f(x)$ and $g(x)$ is defined as [Nus82, Sec 2.2.4]

$$[f * g](x) = \int f(\tau)g(x - \tau)\, d\tau.$$

However, if $f$ and $g$ are polynomials in $\mathbb{Z}[x]$ (or $\mathbb{Z}_q[x]$), the convolution of $f$ and $g$ is equivalent to polynomial multiplication. Similarly, cyclic convolution is equivalent to multiplication in $\mathbb{Z}[x]/(x^n - 1)$ (or $\mathbb{Z}_q[x]/(x^n - 1)$), and negacyclic convolution is equivalent to multiplication in $\mathbb{Z}[x]/(x^n + 1)$ (or $\mathbb{Z}_q[x]/(x^n + 1)$). These terms are often used interchangeably.




(a) $\mathbb{Z}[x]$ (b) $\mathbb{Z}[x]/(x^3-1)$ (c) $\mathbb{Z}[x]/(x^3+1)$

Figure 2.1: Schoolbook multiplication of polynomials $a = a_2x^2 + a_1x + a_0$ and $b = b_2x^2 + b_1x + b_0$

### 2.2.1 Schoolbook Multiplication

Before diving into the different algorithms available for polynomial multiplication, it makes sense to revisit the problem at hand and its straightforward solution: Given two $n$-coefficient polynomials $a, b$ in some polynomial ring, we want to compute the product $a \cdot b$. In case the polynomial ring is $\mathbb{Z}[x]$ or $\mathbb{Z}_q[x]$, the multiplication is defined as

$$a \cdot b = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \cdot b_j \cdot x^{i+j}.$$

In case the polynomial ring is $\mathbb{Z}[x]/(x^n-1)$, the multiplication becomes

$$a \cdot b \equiv \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} a_i \cdot b_j \cdot x^{i+j} + \sum_{j=1}^{n-1} \sum_{i=n-j}^{n-1} a_i \cdot b_j \cdot x^{i+j-n} \pmod{x^n-1}.$$

Similarly, for $\mathbb{Z}[x]/(x^n+1)$ the multiplication is defined as

$$a \cdot b \equiv \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} a_i \cdot b_j \cdot x^{i+j} - \sum_{j=1}^{n-1} \sum_{i=n-j}^{n-1} a_i \cdot b_j \cdot x^{i+j-n} \pmod{x^n+1}.$$

As we usually work with polynomial rings over $\mathbb{Z}_q$ rather than over $\mathbb{Z}$, the intermediate addition and multiplication of coefficients can be performed modulo $q$.

**Example 2:** Let $a = x^2 + 2x + 3$, $b = x^2 + x$.
In $\mathbb{Z}[x]$, the product is $x^4 + 3x^3 + 5x^2 + 3x$.
In $\mathbb{Z}[x]/(x^3-1)$, the product is $(5x^2 + 3x) + (x + 3) = 5x^2 + 4x + 6$.
In $\mathbb{Z}[x]/(x^3+1)$, the product is $(5x^2 + 3x) - (x + 3) = 5x^2 + 2x$.

These three multiplications are illustrated for 3-coefficient polynomials in Figure 2.1. When implementing schoolbook multiplication, there exist two

different approaches: Either one fixes a coefficient $a_i$ and iterates through all coefficients of $b$. This corresponds to computing one row in Figure 2.1 and is called operand scanning. Alternatively, one uses product scanning, where one column in Figure 2.1 is computed at a time.

Note that the number of multiplications of coefficients required for all these algorithms is $n^2$, while the number of additions is $(n-1)^2$. For small $n$ it is often the case that schoolbook multiplication is actually the fastest approach available as most other approaches work by breaking down a larger multiplication into multiple smaller ones which incur some overhead that may outweigh the gain. The actual cut-off point for each method not only depends on the polynomial ring, but also on the available multipliers and adders on the target platform and their respective performance characteristics. Hence, it is important to understand the optimal approach to implement schoolbook multiplication.

**Application:** Schoolbook multiplication of small polynomials is often used as a building block in other multiplications methods. Chapter 5 describes extensive optimization on the Arm Cortex-M4 of schoolbook multiplications for polynomials with eight to 16 coefficients applicable, for example, to Saber [DKRV17] and NTRU [ZCH+19].

### 2.2.2 Karatsuba Multiplication

Karatsuba's multiplication method [KO63] allows breaking down a large polynomial multiplication ($n$-coefficient multiplicands) into three smaller polynomial multiplications with $n/2$-coefficient multiplicands. The following example illustrates Karatsuba's idea.

**Example 3:** Consider the most straightforward example with 2-coefficient inputs $a = a_1x + a_0$ and $b = b_1x + b_0$. The schoolbook method from the previous section would compute the product $a \cdot b$ as

$$a \cdot b = (a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + (a_0b_1 + a_1b_0)x + a_0b_0.$$

This requires four coefficient multiplications and one coefficient addition. The Karatsuba algorithm exploits the fact that

$$(a_0b_1 + a_1b_0) = (a_0 + a_1)(b_0 + b_1) - a_1b_1 - a_0b_0.$$

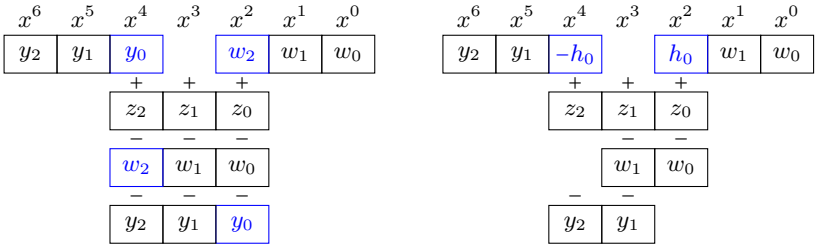
It hence computes the product as

$$a \cdot b = (a_1x + a_0)(b_1x + b_0) = a_1b_1x^2 + ((a_0 + a_1)(b_0 + b_1) - a_1b_1 - a_0b_0)x + a_0b_0.$$

This now requires five coefficient products. However, $a_0b_0$ and $a_1b_1$ are used twice, consequently, we only need to compute three coefficient multiplications. Despite the savings in multiplications, we now also require more additions. In total, four additions are needed.

This approach can be easily generalized to arbitrary-degree polynomials. Consider $a = \sum_{i=0}^{n-1} a_i x^i$, $b = \sum_{i=0}^{n-1} b_i x^i$. We now set $t = x^{\lfloor \frac{n}{2} \rfloor}$, and rewrite $a = a^{(1)}t + a^{(0)}, b = b^{(1)}t + b^{(0)}$ where $a^{(0)}$ and $b^{(0)}$ are $\lfloor \frac{n}{2} \rfloor$-coefficient polynomials consisting of the lower half of the coefficients of $a$ and $b$. Conversely, $a^{(1)}$ and $b^{(1)}$ are $\lceil \frac{n}{2} \rceil$-coefficient polynomials consisting of the upper half of the coefficients of $a$ and $b$. We can then use Karatsuba, to compute $a \cdot b$ using $a^{(0)} \cdot b^{(0)}$, $a^{(1)} \cdot b^{(1)}$, and $(a^{(0)} + a^{(1)}) \cdot (b^{(0)} + b^{(1)})$. Note that the smaller multiplications are now $\lfloor \frac{n}{2} \rfloor$- or $\lceil \frac{n}{2} \rceil$-coefficient polynomial multiplications. We illustrate the Karatsuba trick for $n = 4$ in Figure 2.2a.

**Recursive Karatsuba.** For large-degree inputs, this approach can be applied recursively. We can break down an $n$-coefficient multiplication into three $n/2$-coefficient multiplications each of which gets broken down into three $n/4$-coefficient multiplications. This is called applying multiple layers of Karatsuba. Each additional layer introduces more additions while saving some multiplications. It is likely that it is not optimal to do this all the way down to 1-coefficient polynomials at which polynomial multiplication becomes trivial, but rather one wants to stop earlier and perform a schoolbook multiplication of small-degree polynomials.




(a) Karatsuba (b) Refined Karatsuba

Figure 2.2: Karatsuba multiplication and refined Karatsuba multiplication for polynomials $a = a_0 + a_1x + a_2x^2 + a_3x^3, b = b_0 + b_1x + b_2x^2 + b_3x^3$. Let $w = (a_0 + a_1x)(b_0 + b_1x), y = (a_2 + a_3x)(b_2 + b_3x), z = ((a_0 + a_2) + (a_1 + a_3)x)((b_0 + b_2) + (b_1 + b_3)x)$. For refined Karatsuba, let $h = w_2 - y_0$.

**Example 4:** A common approach for multiplying 256-coefficient polynomials is to use four layers of Karatsuba and then switch to schoolbook multiplication. The schoolbook multiplications handle polynomials with $256/16 = 16$ coefficients and we need a total of $3^4 = 81$ of them.

**Refined Karatsuba.** Looking at Figure 2.2a, one can see that the term $x_2 - y_0$ appears twice in our product. Once for coefficient $c^{(2)}$, and once for coefficient $c^{(4)}$ (as $y_0 - x_2$). We can exploit this fact and simply compute the subtraction once as $h = x_2 - y_0$. The result is what is referred to as refined Karatsuba [Ber01] and is shown in Figure 2.2b. Assuming inputs of $n$ coefficients, this trick can save $(n/2) \cdot 2 - 1 = n - 1$ subtractions. Note, however, that this will only work if the negation of $h$ can be computed for free. This can be achieved by computing $z_2 - h$ rather than $-h + z_2$.

**Application:** Chapter 5 describes how to use recursive refined Karatsuba to implement efficient multiplications of polynomials of degree 16 to about 256 on the Arm Cortex-M4. Below degree 16 schoolbook multiplication is superior, above 256 Toom–Cook outperforms Karatsuba.

### 2.2.3 Toom–Cook Multiplication

The idea of splitting up input polynomials into smaller polynomials as done by Karatsuba's multiplication algorithm can be generalized to split into a larger number of polynomials. One such generalization is Toom–Cook multiplication [Too63, Coo66]. $N$-way Toom–Cook (or Toom-$N$ for short) is splitting polynomials with $n$ coefficients into $N$ parts of $n/N$ coefficients each. The underlying smaller polynomial multiplications will then process polynomials of $n/N$ coefficients.

For example, Toom-3 splits inputs into three parts of $n/3$ coefficients. This is done in the following way: We substitute $y = x^{n/3}$ and write the polynomial $a$ as $y^2a^{(2)} + ya^{(1)} + a^{(0)}$ and proceed similarly for the input $b$. The goal is now to compute the product

$$c = a \cdot b = y^4c^{(4)} + y^3c^{(3)} + y^2c^{(2)} + yc^{(1)} + c^{(0)}.$$

Toom–Cook does so by evaluating the polynomial $a$ and $b$ at certain values of $y$, then multiplying the smaller $n/N$ polynomials, such that $c$ can be recovered from the smaller products using interpolation. For Toom-3, this requires five values for $y$. A common choice is $y = \{0, 1, -1, -2, \infty\}$ with $a(\infty) = a^{(2)}$ Evaluating $a$ and $b$ yields

$$\begin{bmatrix} a(0) \\ a(1) \\ a(-1) \\ a(-2) \\ a(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a^{(0)} \\ a^{(1)} \\ a^{(2)} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} b(0) \\ b(1) \\ b(-1) \\ b(-2) \\ b(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b^{(0)} \\ b^{(1)} \\ b^{(2)} \end{bmatrix}.$$

By *pointwise* multiplication, we obtain five points of $c = a \cdot b$ which is sufficient to recover the polynomial $y^4c^{(4)} + y^3c^{(3)} + y^2c^{(2)} + yc^{(1)} + c^{(0)}$.

Given that

$$\begin{bmatrix} a(0) \cdot b(0) \\ a(1) \cdot b(1) \\ a(-1) \cdot b(-1) \\ a(-2) \cdot b(-2) \\ a(\infty) \cdot b(\infty) \end{bmatrix} = \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(-2) \\ c(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \end{bmatrix}$$

we can obtain $c^{(0)}, c^{1)}, c^{(2)}, c^{(3)}$, and $c^{(4)}$ by inverting the matrix as

$$\begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1/2 & 1/3 & -1 & 1/6 & -2 \\ -1 & 1/2 & 1/2 & 0 & -1 \\ -1/2 & 1/6 & 1/2 & -1/6 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(-2) \\ c(\infty) \end{bmatrix}.$$

This immediately reveals a disadvantage of Toom multiplication. In the case of Toom-3, one needs to be able to divide by 3 and 2. As we are commonly working in a finite ring $\mathbb{Z}_q$, this presents a challenge. In general, there are two ways this can be achieved: If working in $\mathbb{Z}_q$ with $q$ co-prime to 2 and 3, one can simply multiply by the inverses of 2 and 3. However, if $q$ is not co-prime to either 2 or 3, this is no longer possible. To counter this issue, we instead need to do all computations modulo a larger $q'$, such that we can be sure that whenever we need to do divisions, the remainder will always

**Algorithm 1** Toom-3 evaluation and interpolation sequence

**Input:** $a = a^{(0)} + a^{(1)}y + a^{(2)}y^2, \quad b = b^{(0)} + b^{(1)}y + b^{(2)}y^2$
**Output:** $c = a \cdot b = c^{(0)} + c^{(1)}y + c^{(2)}y^2 + c^{(3)}y^3 + c^{(4)}y^4$

1: $a(0) \leftarrow a^{(0)} \quad b(0) \leftarrow b^{(0)} \quad \triangleright$ Evaluate $a$ and $b$ at $y_i = \{0, 1, -1, -2, \infty\}$
2: $t \leftarrow a^{(0)} + a^{(2)} \quad a(1) \leftarrow t + a^{(1)} \quad a(-1) \leftarrow t - a^{(1)}$
3: $t \leftarrow b^{(0)} + b^{(2)} \quad b(1) \leftarrow t + b^{(1)} \quad b(-1) \leftarrow t - b^{(1)}$
4: $a(-2) \leftarrow a^{(0)} - 2a^{(1)} + 4a^{(2)} \quad b(-2) \leftarrow b^{(0)} - 2b^{(1)} + 4b^{(2)}$
5: $a(\infty) \leftarrow b^{(2)} \quad b(\infty) \leftarrow b^{(2)}$
6: ... $\triangleright$ Perform small mults to obtain $c(0)$, $c(\infty)$, $c(1)$, $c(-1)$, $c(-2)$
7: $c^{(0)} \leftarrow c(0) \quad c^{(4)} \leftarrow c(\infty)$ $\triangleright$ Interpolate c
8: $t_1 \leftarrow (c(-2) - c(1))/3$ $\triangleright$ $-c^{(1)} + c^{(2)} - 3c^{(3)} + 5c^{(4)}$
9: $t_2 \leftarrow (c(1) - c(-1))/2$ $\triangleright$ $c^{(1)} + c^{(3)}$
10: $t_3 \leftarrow c(-1) - c(0)$ $\triangleright$ $-c^{(1)} + c^{(2)} - c^{(3)} + c^{(4)}$
11: $c^{(3)} \leftarrow (t_3 - t_1)/2 + 2c^{(4)}$
12: $c^{(2)} \leftarrow t_3 + t_2 - c^{(4)}$
13: $c^{(1)} \leftarrow t_2 - c^{(3)}$

be 0. If $2 \mid q$, we need $q' \geq 2q$; if $3 \mid q$, we need $q' \geq 3q$; if $6 \mid q$, we need $q' \geq 6q$. Note that in the common case where $q$ is a power of two, the inverse of 2 does not exist, and consequently, we need $q' = 2q$, i.e., an additional bit is needed. In the literature, this is sometimes referenced to Toom–Cook losing bits of precision. In practice, this places constraints on the moduli that can be supported by our multiplication routine. For example, if we would like to use 16-bit coefficients, our Toom-3 implementation only supports $q \leq 2^{15}$.

Note that the performance of Toom–Cook is in large parts determined by the efficiency of the evaluation and interpolation stages. A straightforward implementation of each row of the Toom evaluation matrix and interpolation matrix will not yield a competitive implementation. Algorithm 1 outlines a more efficient evaluation and interpolation sequence.

**Toom-4.** Similarly, the Toom–Cook can be used to split into more and smaller parts. For example, Toom-4 uses

$$a = y^3a^{(3)} + y^2a^{(2)} + ya^{(1)} + a^{(0)}.$$

It is not hard to see that this requires to evaluate both arguments at seven points as the resulting product contains terms up to $y^6$:

$$a \cdot b = c = y^6a^{(6)} + y^5a^{(5)} + y^4a^{(4)} + y^3a^{(3)} + +y^2a^{(2)} + ya^{(1)} + c^{(0)}.$$

Common evaluation points for Toom-4 are $y = \{0, 1, -1, 2, -2, 3, \infty\}$, i.e.,

$$\begin{bmatrix} a(0) \\ a(1) \\ a(-1) \\ a(2) \\ a(-2) \\ a(3) \\ a(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 2 & 4 & 8 \\ 1 & -2 & 4 & -8 \\ 1 & 3 & 9 & 27 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a^{(0)} \\ a^{(1)} \\ a^{(2)} \\ a^{(3)} \end{bmatrix}.$$

After multiplying $a(y_i)$ with $b(y_i)$, we have

$$\begin{bmatrix} a(0) \cdot b(0) \\ a(1) \cdot b(1) \\ a(-1) \cdot b(-1) \\ a(2) \cdot b(2) \\ a(-2) \cdot b(-2) \\ a(3) \cdot b(3) \\ a(\infty) \cdot b(\infty) \end{bmatrix} = \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(2) \\ c(-2) \\ c(3) \\ c(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 \\ 1 & 3 & 9 & 27 & 81 & 243 & 729 \\ 0 & 0 & 0 & 0 & & 0 & 1 \end{bmatrix} \begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \\ c^{(5)} \\ c^{(6)} \end{bmatrix}.$$

By inverting the matrix, we obtain the following for interpolation:

$$\begin{bmatrix} c^{(0)} \\ c^{(1)} \\ c^{(2)} \\ c^{(3)} \\ c^{(4)} \\ c^{(5)} \\ c^{(6)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1/3 & 1 & -1/2 & -1/4 & 1/20 & 1/30 & -12 \\ -5/4 & 2/3 & 2/3 & -1/24 & -1/24 & 0 & 4 \\ 5/12 & -7/12 & -1/24 & 7/24 & -1/24 & -1/24 & 15 \\ 1/4 & -1/6 & -1/6 & 1/24 & 1/24 & 0 & -5 \\ -1/12 & 1/12 & 1/24 & -1/24 & -1/120 & 1/120 & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c(0) \\ c(1) \\ c(-1) \\ c(2) \\ c(-2) \\ c(3) \\ c(\infty) \end{bmatrix},$$

which requires a much longer evaluation and interpolation sequence. One which proved to yield good performance results in practice is shown in Algorithm 2. Note that we need to divide by $120 = 5 \cdot 3 \cdot 2^3$ and, consequently, either $q$ needs to be co-prime to 120, such that we can multiply by the inverses, or that we need to perform the smaller multiplications modulo $120q$ such that no wrap-around happens. If $q$ is a power of two, it suffices to use $q' = 8 \cdot q$, i.e., use three extra bits.

**Picking evaluation points.** In previous sections, we have picked points $y$ at which to evaluate the polynomials. Those points seemingly came out of thin air. Indeed, it is possible to pick other points and it is not immediately clear that these points are optimal. It appears natural to always use 0 and $\infty$ as those are cheap to evaluate at and interpolate from. The other points are arbitrary, and one picks the ones that give the best performance. The points presented in this section are the ones most commonly used in the

**Algorithm 2** Toom-4 interpolation sequence

**Input:** $a = a^{(0)} + a^{(1)}y + a^{(2)}y^2 + a^{(3)}y^3, \quad b = b^{(0)} + b^{(1)}y + b^{(2)}y^2 + b^{(3)}y^3$
**Output:** $c = a \cdot b = c^{(0)} + c^{(1)}y + c^{(2)}y^2 + c^{(3)}y^3 + c^{(4)}y^4 + c^{(5)}y^5 + c^{(6)}y^6$

1: ... ▷ Evaluate $a$ and $b$ at $y = \{0, 1, -1, 2, -2, 3, \infty\}$
2: ... ▷ Small multiplications → $c(0)$, $c(\infty)$, $c(1)$, $c(-1)$, $c(2)$, $c(-2)$, $c(3)$
3: $c^{(0)} \leftarrow c(0) \quad c^{(6)} \leftarrow c(\infty)$ ▷ Interpolate c
4: $t_0 \leftarrow (c(1) + c(-1))/2 - c^{(0)} - c^{(6)}$ ▷ $c^{(2)} + c^{(4)}$
5: $t_1 \leftarrow (c(2) + c(-2) - 2c^{(0)} - 128c^{(6)})/8$ ▷ $c^{(2)} + 4c^{(4)}$
6: $c^{(4)} \leftarrow (t_1 - t_0)/3 \quad c^{(2)} \leftarrow (t_0 - c^{(4)})$
7: $t_0 \leftarrow (c(1) - c(-1))/2$ ▷ $c^{(1)} + c^{(3)} + c^{(5)}$
8: $t_1 \leftarrow ((c(2) - c(-2))/4 - t_0)/3$ ▷ $c^{(3)} + 5c^{(5)}$
9: $t_2 \leftarrow (c(3) - c^{(0)} - 9c^{(2)} - 81c^{(4)} - 729c^{(6)})/3$ ▷ $c^{(1)} + 9c^{(3)} + 81c^{(5)}$
10: $t_2 \leftarrow (t_2 - t_0)/8 - t_1$ ▷ $5c^{(5)}$
11: $c^{(5)} \leftarrow t_2/5 \quad c^{(3)} \leftarrow t_1 - t_2 \quad c^{(1)} \leftarrow t_0 - c^{(3)} - c^{(5)}$

literature in the context of lattice-based cryptography, but other choices are possible.

**Finding the best combination of Toom-N and Karatsuba.** Since Toom–Cook, as well as Karatsuba, can be applied recursively, there are a large number of combinations that can be constructed. The actual optimal choice depends on the target platform and polynomials to be multiplied. The modulus $q$ introduces the largest restrictions. In case it is prime, many more combinations are possible, while for even moduli, higher order Toom–Cook and many combinations are not possible due to the requirements of using a larger $q$ in the smaller multiplications. For example, if smaller multiplications are done using 16-bit multiplications, $q$ must be at most $2^{15}$ ($2 \cdot q \leq 2^{16}$) for Toom-3, and at most $2^{13}$ ($8 \cdot q \leq 2^{16}$) for Toom-4. Applying a combination of Toom-4 and Toom-3 would require $8 \cdot 2 \cdot q = 16 \cdot q \leq 2^{16}$, i.e., $q \leq 2^{12}$.

**Application:** In Chapter 5, Toom-3 and Toom-4 are used in combination with Karatsuba to multiply polynomials of degree 256 and above on the Arm Cortex-M4.

### 2.2.4 Number-Theoretic Transform (NTT)

The number-theoretic transform can be seen as a discrete Fourier transform in a finite ring. The general idea is to perform (polynomial) multiplications by transforming both factors to a different domain in which multiplications are cheap. In the case of discrete Fourier transforms, this domain is called the frequency domain, while for NTTs, we refer to it as the *NTT domain*. Elements in the *NTT domain* are commonly identified by a hat, e.g., $\hat{a}$. The

product is then transformed back to the *normal domain* which is also called the *time domain* using the inverse NTT ($\text{NTT}^{-1}$).

Given two polynomials $a, b$, one computes their product as

$$c = a \cdot b = \text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$$

with $\circ$ referring to the multiplication in the NTT domain. In the most straightforward instantiation of NTTs, $\circ$ simply means coefficient-wise multiplication modulo $q$.

For simplicity, we assume that $q$ is a prime number as this makes the explanation easier. However, composite-modulus NTTs are possible and are being used. For understanding NTTs, there is an essential mathematical entity that we need to understand: primitive roots of unity in $\mathbb{Z}_q$ [Nus82, Sec. 2.1.3]. An element $a \in \mathbb{Z}_q$ is a $k$-th (with $k \in \mathbb{N}$) root of unity if $a^k \equiv 1 \mod q$. It is a primitive root of unity, if there is no $k' < k$, s.t. $a^{k'} \equiv 1 \mod q$. A primitive $k$-th root of unity exists only if $k \mid (q-1)$ (or more generally, it only exists if $k$ divides the order of $\mathbb{Z}_q^*$.) Throughout this section, $\omega_k$ denotes a primitive $k$-th root of unity.

**Example 5:** 1 and $-1 \equiv q-1 \mod q$ are 2-nd roots of unity. However, only $-1$ is a primitive 2-nd root of unity.
For $\mathbb{Z}_7$, there are six 6-th roots of unity ($\{1,2,3,4,5,6\}$), but only $\{3,5\}$ are primitive 6-th roots of unity.

When computing an NTT, one is evaluating a polynomial at powers of a primitive root of unity $\omega_k$: $\omega_k^0, \omega_k^1, \ldots, \omega_k^{n-1}$.

The NTT [Nus82, Sec. 8.1] is defined as

$$\hat{a} = \sum_{i=0}^{n-1} \hat{a}_i x^i \quad \text{with} \quad \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega_k^{i \cdot j}.$$

Transforming $\hat{a}$ to the normal domain is achieved by computing $\text{NTT}^{-1}$ as

$$a = \sum_{i=0}^{n-1} a_i x^i \quad \text{with} \quad a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega_k^{-i \cdot j}.$$

**Example 6:** From the definition of the NTT, it is not immediately obvious why it works. Consider the following example: We want to multiply $a = a_1x + a_0$ by $b = b_1x + b_0$ with coefficients in $\mathbb{Z}_7$. We know that the result will be of degree at most 2, therefore, we have to evaluate at 3 points. To compute the NTT, we require a 3-rd root of unity, e.g., $\omega_3 = 2$. We now evaluate $a(x)$ and $b(x)$ at $x = \{\omega_3^0, \omega_3^1, \omega_3^2\}$:

$$\hat{a} = \begin{bmatrix} a(\omega_3^0) \\ a(\omega_3^1) \\ a(\omega_3^2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_3 \\ 1 & \omega_3^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad \text{and} \quad \hat{b} = \begin{bmatrix} b(\omega_3^0) \\ b(\omega_3^1) \\ b(\omega_3^2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_3 \\ 1 & \omega_3^2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}.$$

A pointwise multiplication yields

$$\hat{a} \circ \hat{b} = \begin{bmatrix} c(\omega_3^0) \\ c(\omega_3^1) \\ c(\omega_3^2) \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_0b_1 + a_1b_0 + a_1b_1 \\ a_0b_0 + \omega_3 a_0b_1 + \omega_3 a_1b_0 + \omega_3^2 a_1b_1 \\ a_0b_0 + \omega_3^2 a_0b_1 + \omega_3^2 a_1b_0 + \omega_3 a_1b_1 \end{bmatrix}.$$

Computing $\text{NTT}^{-1}$ corresponds to the interpolation of $c(x)$ from $c(\omega_3^0)$, $c(\omega_3^1)$, and $c(\omega_3^2)$. For that we require the inverse of $\omega_3$ which in our example is $\omega_3^{-1} = 4$.
It now holds that,

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = 3^{-1} \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3^{-1} & \omega_3^{-2} \\ 1 & \omega_3^{-2} & \omega_3^{-1} \end{bmatrix} \begin{bmatrix} c(\omega_3^0) \\ c(\omega_3^1) \\ c(\omega_3^2) \end{bmatrix} = \begin{bmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{bmatrix}.$$

It is important to note here that this all works out because $1 + \omega_3 + \omega_3^2 \equiv 0 \mod q$. It is not hard to see that this can be generalized to any size of polynomials as long as the appropriate primitive root of unit exists.

In a normal polynomial multiplication for $n$-coefficient factors, it is sufficient to evaluate the polynomials at $2n - 1$ points. However, in real implementations, this is never how it is done because working with polynomials of odd length is very disadvantageous for fast implementations. Hence, one would use $2n$ points or round up to the closest power of two.

**Finding a root of unity.** In the previous examples, we have pulled the primitive root of unity out of thin air. The easiest way to find one is to try all field elements and check if they are indeed a $k$-th primitive root of unity.

**Example 7:** Considering $\mathbb{Z}_{17}$, we can check the multiplicative order of each element, and find that 16 is a primitive 2-nd root of unity, $\{4, 13\}$ are 4-th roots of unity, $\{2, 8, 9, 15\}$ are 8-th roots of unity, and $\{3, 5, 6, 7, 10, 11, 12, 14\}$ are 16-th roots of unity. If one needs a 16-th root of unity, any of the eight possible values will work and result in a correct NTT. Usually, the root is arbitrarily picked as the smallest available.

### Cyclic NTT

One big advantage of NTT-based multiplication is that it naturally supports convolutions. In the case of cyclic convolutions (modulo $x^n - 1$), this is possible using a cyclic NTT which is exactly the NTT that was introduced in the previous section. However, one only needs to evaluate at $n$ roots of unity. Why this works can be easily seen in the following example. For a more formal proof of the correctness, see [Nus82, Theorem 8.1].

**Example 8:** Consider the polynomial ring $\mathbb{Z}_q[x]/(x^2 - 1)$, and assume there is a 2-nd root of unity $\omega_2$, i.e., $\omega_2^2 \equiv 1 \mod q$. The polynomials $a = a_1x + b_0$, $b = b_1x + b_0$, can be transformed to the NTT domain by evaluating the polynomials at $\omega_2^0, \omega_2^1$:

$$\hat{a} = \begin{bmatrix} a(\omega_2^0) \\ a(\omega_2^1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad \text{and} \quad \hat{b} = \begin{bmatrix} b(\omega_2^0) \\ b(\omega_2^1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & \omega_2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}.$$

A pointwise multiplication yields

$$\hat{a} \circ \hat{b} = \begin{bmatrix} c(\omega_2^0) \\ c(\omega_2^1) \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_0b_1 + a_1b_0 + a_1b_1 \\ a_0b_0 + \omega_2 a_0b_1 + \omega_2 a_1b_0 + a_1b_1 \end{bmatrix}.$$

When we apply the $\text{NTT}^{-1}$ to that pointwise product we obtain

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = 2^{-1} \begin{bmatrix} 1 & 1 \\ 1 & \omega_2^{-1} \end{bmatrix} \begin{bmatrix} c(1) \\ c(\omega_2) \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_1b_1 \\ a_0b_1 + a_1b_0 \end{bmatrix},$$

which coincides with polynomial multiplication in $\mathbb{Z}_q[x]/(x^n - 1)$.

### Negacyclic NTT.

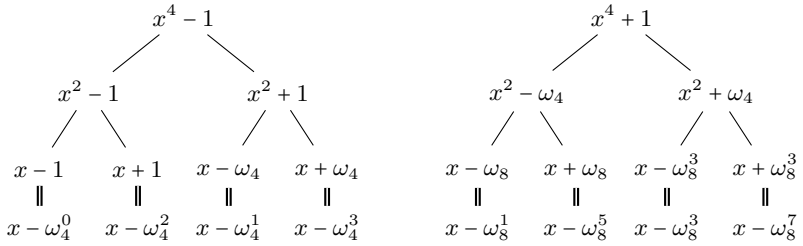
Similarly, we would like to multiply polynomials in $\mathbb{Z}_q[x]/(x^n + 1)$. This can be achieved using the negacyclic NTT [SS71]. Given a $2n$-th root of unity $\omega_{2n}$ (i.e., $\omega_{2n}^2 = \omega_n$), it is defined as

$$\hat{a} = \sum_{i=0}^{n-1} \hat{a}_i x^i \quad \text{with} \quad \hat{a}_i = \sum_{j=0}^{n-1} a_j \omega_{2n}^j \omega_n^{i \cdot j}.$$

Its inverse is defined as

$$a = \sum_{i=0}^{n-1} a_i x^i \quad \text{with} \quad a_i = n^{-1} \omega_{2n}^{-i} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-i \cdot j}.$$

Note that the negacyclic NTT is the same as multiplying each input coefficient $a_i$ by $\omega_{2n}^i$ and then applying the NTT as defined in the previous section. The multiplication by the powers of roots of unity is called twisting [Ber01]. The inverse also works similarly, but the output is twisted back by multiplying by powers of $\omega_{2n}^{-1}$.




(a) Cyclic NTT for $\mathbb{Z}_q[x]/(x^n - 1)$ (b) Negacyclic NTT for $\mathbb{Z}_q[x]/(x^n + 1)$

Figure 2.3: Splitting polynomial rings using the NTT.

**Example 9:** Let our input polynomials be $a = \sum_{i=0}^{n-1} a_i x^i$ and $b = \sum_{i=0}^{n-1} b_i x^i$. We twist both polynomials by multiplying by powers of $\omega_{2n}$ to obtain $a' = \sum_{i=0}^{n-1} a_i \omega_{2n}^i x^i$ and $b' = \sum_{i=0}^{n-1} b_i \omega_{2n}^i x^i$. Now, we perform a regular polynomial multiplication (e.g., using schoolbook multiplication) to obtain $c' = \sum_{i=0}^{2n-2} c'_i x^i \omega_{2n}^i$. Since $\omega_{2n}^n \equiv -1 \mod q$, we can rewrite this as $c' = \sum_{i=0}^{n-1} c'_i x^i \omega_{2n}^i - \sum_{i=n}^{2n-2} c'_i x^i \omega_{2n}^{i-n}$. When this is convoluted modulo $x^n - 1$ and twisted back (i.e., the powers of $\omega_{2n}$ being removed), we obtain the correct product of $a$ and $b$ modulo $x^n + 1$. Note that the regular polynomial multiplication and reduction modulo $x^n - 1$ can be replaced by an implementation using a cyclic NTT.

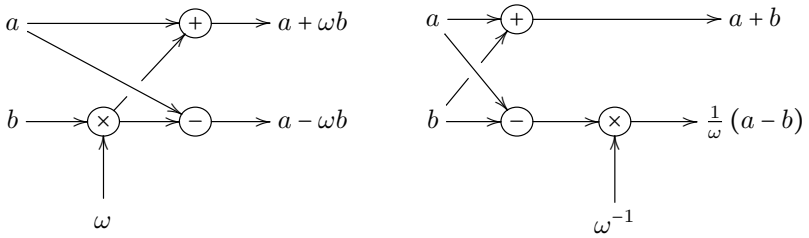
**Changing perspective: Chinese Remainder Theorem (CRT)**

There is another way to think about what an NTT is, which may appear more natural or elegant: The polynomial $x^{2n} - 1$ can be split into two factors: $x^n - 1$ and $x^n + 1$. That means that a polynomial $a$ in $\mathbb{Z}_q[x]/(x^{2n} - 1)$ can be split into two polynomials, $a'$ in $\mathbb{Z}_q[x]/(x^n - 1)$ and $a''$ in $\mathbb{Z}_q[x]/(x^n + 1)$ by simply reducing modulo $x^n - 1$ and modulo $x^n + 1$. From the two smaller polynomials, one can recover the original polynomial using the Chinese Remainder Theorem (CRT), i.e.,

$$a = \sum_{i=0}^{n-1} \frac{1}{2}(a'_i + a''_i)x^i + \sum_{i=0}^{n-1} \frac{1}{2}(a'_i - a''_i)x^{n+i}.$$

**Example 10:** Let $a = a_0 + a_1 x + a_2 x^2 + a_3 x^3$, we can then compute $a' = (a_0 + a_2) + (a_1 + a_3)x$ and $a'' = (a_0 - a_2) + (a_1 - a_3)x$. To recover the original $a$, we compute $a = \frac{1}{2}\big((a'_0 + a''_0) + (a'_1 + a''_1)x + (a'_0 - a''_0)x^2 + (a'_1 - a''_1)x^3\big)$.

For even $n$, one can apply this trick again for $x^n - 1$ since $x^n - 1 = (x^{n/2} - 1)(x^{n/2} + 1)$. However, we can also do the same for $x^n + 1$ in case there is an appropriate root of unity $\omega_4 = \sqrt{-1}$, s.t. $x^n + 1 = (x^{n/2} - \omega_4)(x^{n/2} + \omega_4)$.




(a) Cooley–Tukey Butterfly [CT65] (b) Gentleman–Sande Butterfly [GS66]

Figure 2.4: The "Butterflies" of Fast Fourier Transforms

Applying this trick recursively, it is easy to see that, we can split $x^n - 1$ into $n$ linear parts in case there exists an $n$-th root of unity $\omega_n$:

$$x^n - 1 = \prod_{i=0}^{n-1} (x - \omega_n^i).$$

This splitting is illustrated in Figure 2.3a.

Similarly, for $x^n + 1$ we can fully split into linear terms in case a $2n$-th root of unity $\omega_{2n}$ exists as illustrated in Figure 2.3b:

$$x^n + 1 = \prod_{i=0}^{n-1} (x - \omega_{2n}\omega_n^i) = \prod_{i=0}^{n-1} (x - \omega_{2n}^{2i+1}).$$

**Application:** Chapter 3, Chapter 4, and Chapter 6 cover how to efficiently implement NTTs for Kyber, Dilithium, Saber, NTRU, and LAC. On the Cortex-M4, NTTs are superior to Toom and Karatsuba multiplication for these schemes.

### 2.2.5 Algorithms for Computing NTTs

Using NTTs for fast arithmetic is only beneficial if the transformations themselves can be implemented efficiently. From the previous section, it is not obvious that these can be implemented efficiently. A straightforward implementation of the transformation requires $n^2$ multiplications and is, therefore, not at all better than just using the schoolbook method for the multiplication itself. However, there are much faster algorithms that allow computing the NTT in quasi-linear time $\mathcal{O}(n \log n)$. These algorithms are called Fast Fourier Transform (FFT) algorithms. The two most prominent ones are by Cooley–Tukey (CT) [CT65] and Gentleman–Sande (GS) [GS66]. The CT FFT algorithm is also referred to as decimation in time (DIT) FFT, while the GS FFT algorithm is sometimes called decimation in frequency (DIF) FFT. It is common practice to implement the forward NTT using the CT

**Algorithm 3** CT FFT implementing forward NTT

**Input**: Polynomial $a \in \mathbb{Z}_q/(x^{2^k} \pm 1)$; time domain, normal order
**Input**: Root of unity: $\omega_{2^k}$ for $\mathbb{Z}_q/(x^{2^k} - 1)$, $\omega_{2^{k+1}}$ for $\mathbb{Z}_q/(x^{2^k} + 1)$
**Output**: $\hat{a}$; NTT domain, bit-reversed order

1: **for** $l = k-1; l \geq 0; l \leftarrow l-1$ **do**
2: **for** $i = 0; i < 2^{k-1-l}; i \leftarrow i+1$ **do**
3: $\psi \leftarrow \begin{cases} \omega_{2^k}^{\mathtt{brv}_{k-1}(i)} \text{ for } \mathbb{Z}_q/(x^{2^k} - 1) \\ \omega_{2^{k+1}}^{\mathtt{brv}_k(2^{k-1-l}+i)} \text{ for } \mathbb{Z}_q/(x^{2^k} + 1) \end{cases}$
4: **for** $j = i \cdot 2^{l+1}; j < i \cdot 2^{l+1} + 2^l; j \leftarrow j+1$ **do**
5: $t_0 \leftarrow a_j$
6: $t_1 \leftarrow \psi \cdot a_{j+2^l}$
7: $a_j \leftarrow t_0 + t_1$
8: $a_{j+2^l} \leftarrow t_0 - t_1$
9: **end for**
10: **end for**
11: **end for**

FFT, and the $\text{NTT}^{-1}$ using the GS FFT. However, both transforms can be implemented using either of the two, and it highly depends on the target platform and parameters which one is best.

FFT algorithms are characterized by their *butterfly* operations. Figure 2.4a and Figure 2.4b show the *butterfly* of the CT and GS FFT algorithm respectively. The CT does straightforwardly implement the split of $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ into $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2^{k-1}} + c)$. The multiplicand $c$ is a constant and is called the twiddle factor. Similarly, the Gentleman–Sande butterfly computes the CRT of elements in $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2^{k-1}} + c)$ yielding an element in $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ using the twiddle factor $c^{-1}$. We can also express this as a ring isomorphism $\phi : \mathbb{Z}_q[x]/(f(x)g(x)) \cong \mathbb{Z}_q[x]/(f(x)) \times \mathbb{Z}_q[x]/(g(x))$, $\phi(h) = (h \bmod f, h \bmod g)$. When $f(x) = x^n - c$ and $g(x) = x^n + c$, $\phi$ naturally becomes

$$\phi\left(\sum_{i=0}^{2n-2} h_i x^i\right) = \left(\sum_{i=0}^{n-1} \underbrace{(h_i + c h_{n+i})}_{CT} x^i, \sum_{i=0}^{n-1} \underbrace{(h_i - c h_{n+i})}_{CT} x^i\right).$$

The inverse computation can be expressed as $\phi^{-1}$:

$$\phi^{-1}\left(\left(\sum_{i=0}^{n-1} h'_i x^i\right), \left(\sum_{i=0}^{n-1} h''_i x^i\right)\right) = \sum_{i=0}^{n-1} \frac{1}{2} \underbrace{(h'_i + h''_i)}_{GS} x^i + \sum_{i=0}^{n-1} \frac{1}{2} \underbrace{\frac{1}{c} (h'_i - h''_i)}_{GS} x^{n+i}.$$

**Algorithm 4** GS FFT implementing $\text{NTT}^{-1}$

**Input**: $\hat{a}$; NTT domain, bit-reversed order
**Input**: Root of unity: $\omega_{2^k}$ for $\mathbb{Z}_q/(x^{2^k}-1)$, $\omega_{2^{k+1}}$ for $\mathbb{Z}_q/(x^{2^k}+1)$
**Output**: Polynomial $a \in \mathbb{Z}_q/(x^{2^k} \pm 1)$; time domain, normal order

1: **for** $l = 0; l < k; l \leftarrow l+1$ **do**
2: **for** $i = 0; i < 2^{k-1-l}; i \leftarrow i+1$ **do**
3: $\psi \leftarrow \begin{cases} \omega_{2^k}^{-\texttt{brv}_{k-1}(i)} \text{ for } \mathbb{Z}_q/(x^{2^k}-1) \\ \omega_{2^k}^{-\texttt{brv}_k(2^{k-1-l}+i)} \text{ for } \mathbb{Z}_q/(x^{2^k}+1) \end{cases}$
4: **for** $j = i \cdot 2^{l+1}; j < i \cdot 2^{l+1} + 2^l; j \leftarrow j+1$ **do**
5: $t_0 \leftarrow a_j + a_{j+2^l}$
6: $t_1 \leftarrow a_j - a_{j+2^l}$
7: $a_j \leftarrow t_0$
8: $a_{j+2^l} \leftarrow \psi \cdot t_1$
9: **end for**
10: **end for**
11: **end for**
12: **for** $i = 0; i < 2^k; i \leftarrow i+1$ **do**
13: $a_i \leftarrow n^{-1} a_i$ ▷ Cancel out factor of 2 for each butterfly
14: **end for**

Algorithm 3 and Algorithm 4 show how these butterflies can be applied iteratively to larger polynomials to obtain a full NTT and $\text{NTT}^{-1}$ in-place. FFT algorithms have the peculiar property that their outputs are in a different order than one might expect: They are in so-called *bit-reversed* order. Bit-reversing an array of length $2^k$ means interpreting the index of each element as a binary string of length $k$ and reversing the binary string. For an index $i$ with $i_j$ denoting the $j$-th bit this can be expressed as

$$i = \sum_{j=0}^{k-1} i_j 2^j, \quad \texttt{brv}_k(i) = \sum_{j=0}^{k-1} i_{k-j} 2^j.$$

The reversed index becomes the new index of each element. The operation can be reversed by applying the same transformation again.

**Example 11:** Given an array of length eight in normal order: $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$.
We write the indices in binary: $[a_{000}, a_{001}, a_{010}, a_{011}, a_{100}, a_{101}, a_{110}, a_{111}]$.
Then, reverse the binary digits: $[a_{000}, a_{100}, a_{010}, a_{110}, a_{001}, a_{101}, a_{011}, a_{111}]$.
This gives us the array in bit-reversed order: $[a_0, a_4, a_2, a_5, a_1, a_5, a_3, a_7]$.

Since bit-reversing is a costly operation in software, one tries to avoid explicitly performing it. Hence, whenever possible one will keep values in

```
# computing twiddles for a cyclic NTT Z_q[x](x^n - 1)
twiddlesNtt    = brv([pow(root, i, q) for i in range(n//2)])
twiddlesInvNtt = brv([pow(root, -i, q) for i in range(n//2)])

# computing twiddles twiddles for a negacyclic NTT Z_q[x](x^n + 1)
twiddlesNtt = brv([pow(root, i, q) for i in range(n)])[1:n]
twiddlesInvNtt = brv([pow(root, -(i+1), q) for i in range(n)])[0:n-1]
```

Listing 2.1: Python code for generating twiddle factors

the NTT domain in bit-reversed order. Since the $\text{NTT}^{-1}$ restores the normal order, this has no impact if polynomial multiplication is implemented. Since multiplication and addition in NTT are element-wise operations, the order does not matter.

Looking at Algorithm 3 and Algorithm 4 one also notices that one requires the twiddle factors with the exponents being bit-reversed as well. Note that this makes it impractical to compute the twiddle factors on the fly. Virtually all fast software implementations resolve this by pre-computing the needed powers of the root of unity and storing them in memory. Those can easily be derived from the tree representations of an FFT, e.g., in Figure 2.3. Splitting $\mathbb{Z}_q[x]/(x^{2^k} - c^2)$ into $\mathbb{Z}_q[x]/(x^{2^{k-1}} - c)$ and $\mathbb{Z}_q[x]/(x^{2^{k-1}} + c)$ uses the twiddle factor $c$, while the inverse uses $c^{-1}$. By traversing the tree breadth-first, we obtain [1, 1, $w_4$] in the cyclic case, and [$\omega_8^2$, $\omega_8$, $\omega_8^3$] in the negacyclic case.

The Python code in Listing 2.1 generalizes this manual derivation given a function `brv` which reorders an array in bit-reversed order.

Note that in the cyclic case, the twiddles repeat, i.e., each layer uses the first $2^{k-1-l}$ twiddle factors from the pre-computed array, while in the negacyclic case, each twiddle is only used once due to the required twisting.

**Discrete Fourier Transform (DFT) matrices.** Another way of thinking about the FFT is by thinking about DFT matrices. A DFT matrix is defined as

$$W_n = \left(\omega_n^{jk}\right)_{j,k=0,\ldots,n-1}$$

Given this DFT matrix, a cyclic NTT transformation is multiplying the coefficient vector of the polynomial by the DFT matrix. The $\text{NTT}^{-1}$ consists of multiplying by the inverse of the DFT matrix.

**Example 12:** Let $n = 4$. The corresponding DFT matrix using a 4-th root of unity $\omega_4$ is

$$W_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix}.$$

The cyclic NTT of a polynomial $a(x) \in \mathbb{Z}_q[x]/(x^4 - 1)$ can be written as

$$\hat{a} = W \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

This can be decomposed into four Cooley–Tukey butterflies, i.e.,

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & \omega_4 \\ 0 & 0 & 1 & -\omega_4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix}.$$

Note that the second and third row are swapped, i.e., the output is in bit-reversed order.

The negacyclic NTT can be expressed similarly by considering it as twisting followed by a cyclic NTT. We write $W'_n = W_n \cdot \mathtt{diag}(w_{2n}^i)_{i=0,\ldots,n-1}$, where $\mathtt{diag}$ is a diagonal matrix containing the elements $\{w_{2n}^0, \ldots, w_{2n}^{n-1}\}$.

**Example 13:** Let $n = 4$. The negacyclic NTT of a polynomial $a(x) \in \mathbb{Z}_q[x]/(x^4 + 1)$ with a $2n$-th root of unity $\omega_{2n}$ ($\omega_{2n}^2 = \omega_4$) is

$$\hat{a} = W_4 \cdot \mathtt{diag}(w_{2n}^i)_{i=0,\ldots,n-1} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega_8 & 0 & 0 \\ 0 & 0 & \omega_8^2 & 0 \\ 0 & 0 & 0 & \omega_8^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & \omega_8 & \omega_8^2 & \omega_8^3 \\ 1 & \omega_8^3 & \omega_8^6 & \omega_8 \\ 1 & \omega_8^5 & \omega_8^2 & \omega_8^7 \\ 1 & \omega_8^7 & \omega_8^6 & \omega_8^5 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

When again accepting that outputs are in bit-reversed order, we can use 4 CT butterflies to implement the negacyclic NTT:

$$\begin{bmatrix} 1 & \omega_8 & 0 & 0 \\ 1 & -\omega_8 & 0 & 0 \\ 0 & 0 & 1 & \omega_8^3 \\ 0 & 0 & 1 & -\omega_8^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & \omega_8^2 & 0 \\ 0 & 1 & 0 & \omega_8^2 \\ 1 & 0 & -\omega_8^2 & 0 \\ 0 & 1 & 0 & -\omega_8^2 \end{bmatrix} = \begin{bmatrix} 1 & \omega_8 & \omega_8^2 & \omega_8^3 \\ 1 & \omega_8^5 & \omega_8^2 & \omega_8^7 \\ 1 & \omega_8^3 & \omega_8^6 & \omega_8 \\ 1 & \omega_8^7 & \omega_8^6 & \omega_8^5 \end{bmatrix}.$$

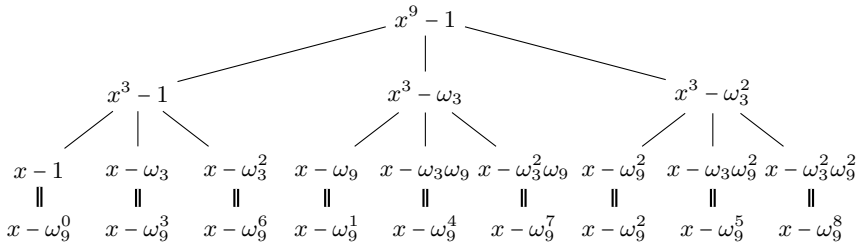



Figure 2.5: Radix-3 FFT for $\mathbb{Z}[x]/(x^9-1)$

### 2.2.6 Radix-3 FFT and Mixed-Radix FFT

The previous section exclusively covered FFTs working on polynomial rings of the form $\mathbb{Z}[x]/(x^{2^k}\pm 1)$. These are called radix-2 FFTs as each NTT layer splits into two parts. However, we can also use FFTs for a different radix. Most commonly used are radix-3 and radix-5 FFTs. They can be best studied by considering a single layer (i.e., split) of the FFT.

A radix-3 FFT splits the ring $\mathbb{Z}[x]/(x^{3^{k+1}}-c^3)$ as

$$\begin{aligned}&\mathbb{Z}[x]/(x^{3^{k+1}}-c^3)\\ \to\ &\mathbb{Z}[x]/(x^{3^k}-c)\times\mathbb{Z}[x]/(x^{3^k}-\omega_3 c)\times\mathbb{Z}[x]/(x^{3^k}-\omega_3^2 c).\end{aligned}$$

A radix-3 Cooley–Tukey-style butterfly splitting $\mathbb{Z}[x]/(x^{3^k}\pm c^3)$ for $0\le i<3^k$ can be implemented as

$$\begin{aligned}\hat{a}_i &= a_i &+&\quad c a_{i+3^k} &+&\quad c^2 a_{i+2\cdot 3^k},\\ \hat{a}_{i+3^k} &= a_i &+&\quad \omega_3 c a_{i+3^k} &+&\quad \omega_3^2 c^2 a_{i+2\cdot 3^k},\\ \hat{a}_{i+2\cdot 3^k} &= a_i &+&\quad \omega_3^2 c a_{i+3^k} &+&\quad \omega_3 c^2 a_{i+2\cdot 3^k}.\end{aligned}$$

The inverse radix-3 butterfly using a Gentleman–Sande-style is

$$\begin{aligned}a_i &= 3^{-1}\quad (a_i &+&\quad a_{i+3^k} &+&\quad a_{i+2\cdot 3^k}),\\ a_{i+3^k} &= 3^{-1}c^{-1}(a_i &+&\quad \omega_3^2 a_{i+3^k} &+&\quad \omega_3 a_{i+2\cdot 3^k}),\\ a_{i+2\cdot 3^k} &= 3^{-1}c^{-2}(a_i &+&\quad \omega_3 a_{i+3^k} &+&\quad \omega_3^2 a_{i+2\cdot 3^k}).\end{aligned}$$

**Example 14:** Of course, the radix-3 FFT trick can be applied recursively. An FFT splitting $\mathbb{Z}_q[x]/(x^9-1)$ into linear factors can be seen in Figure 2.5. The twiddle factors $(c, c^2)$ for the first layer are $(1,1)$. For the second layer they are $(1,1)$, $(\omega_9, \omega_9^2)$, and $(\omega_9^2, \omega_9)$ from left to right respectively. One may notice that the outputs are once again in a different order than they should be. However, this time the order is not bit-reversed, but the equivalent in base 3. As usual, we do not need to be concerned about the order if we are implementing polynomial multiplication as pointwise multiplication can simply operate on base-3-reversed inputs and the inverse FFT will restore the normal order. Base-3-reversing works similarly as bit-reversing: One represents each index in base 3 and reverts the order of the digits, i.e.,

$$(0,1,2,3,4,5,6,7,8) \to (00,01,02,10,11,12,20,21,22)$$
$$\to (00,10,20,01,11,21,02,12,22) \to (0,3,6,1,4,7,2,5,8).$$

**Radix-5 FFT.** The above can be easily extended to FFTs for an arbitrary radix. For post-quantum cryptography so far only radix-5 FFTs were used. One layer of a radix-5 FFT splits $\mathbb{Z}_q[x]/(x^{5^{k+1}}+c^5)$ into 5 elements in $\mathbb{Z}_q[x]/(x^{5^k}+{w_5}^i c)$ for $i = \{0,1,2,3,4\}$.

**Mixed-radix FFTs.** The FFTs for different radices can be combined. One could, for example, perform one layer of radix-2 butterflies followed by a layer of radix-3 butterflies to obtain a 6-FFT. This can be very useful for schemes that have not been specifically designed for the use of the NTT and, hence, often do not nicely split using radix-2 FFTs.

**Application:** Chapter 6 covers how to efficiently implement `ntruhps4096821` [ZCH+19] which requires polynomial multiplication in $\mathbb{Z}_{2048}/(x^{701}-1)$. This can be implemented using a cyclic NTT for $\mathbb{Z}_{3365569}/(x^{1536}-1)$, which can be efficiently implemented by using 9-layer radix-2 FFTs followed by a 1-layer radix-3 FFT.

### 2.2.7 Incomplete NTT

A standard radix-2 FTT is splitting a ring (e.g., $\mathbb{Z}_q[x]/(x^n-1)$) down into smaller rings down to having elements in $\mathbb{Z}_q[x]/(x-\omega_n^i)$ $(0 \le i < n)$, each consisting of only one coefficient. In one step of the FFT algorithm, a polynomial in $\mathbb{Z}_q[x]/(x^{2^k}-c^2)$ is split into one element in $\mathbb{Z}_q[x]/(x^{2^{k-1}}-c)$ and one element in $\mathbb{Z}_q[x]/(x^{2^{k-1}}+c)$. Those then get split into $\mathbb{Z}_q[x]/(x^{2^{k-2}}-\sqrt{c}), \mathbb{Z}_q[x]/(x^{2^{k-2}}+\sqrt{c}), \mathbb{Z}_q[x]/(x^{2^{k-1}}+\sqrt{-c}), \mathbb{Z}_q[x]/(x^{2^{k-1}}+\sqrt{-c})$ and so forth.

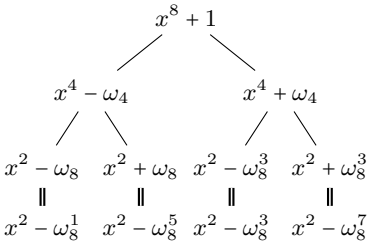



Figure 2.6: Incomplete FFT for $\mathbb{Z}[x]/(x^8+1)$

In this case, it is natural to consider to stop earlier, i.e., instead of doing $\log_2 n$ splits down to linear terms in $\mathbb{Z}_q[x]/(x-\omega_n^i)$ $(0 \le i < n)$, we could do $\log_2 n - 1$ splits and end up with elements in $\mathbb{Z}_q[x]/(x^2-\omega_{n/2}^i)$. This is referred to as an incomplete NTT in the literature.

It results in some clear advantages, the most prominent being that it results in fewer restrictions regarding the prime $q$. In the cyclic NTT example above, only a $n/2$-th root of unity is needed, while in the case of negacyclic NTTs, an $n$-th root of unity suffices. This gives much more choices for $q$. For example, Kyber [ABD+17] used $q = 7681$ in the initial submission together with a negacyclic complete NTT for $\mathbb{Z}_q[x]/(x^{256}+1)$. In the second round of the NISTPQC competition $q$ was changed to 3329. Since no 512-th root of unity modulo 3329 exists, this was only made possible by switching to an incomplete (7-layer) NTT.

Note that the multiplication in the NTT domain is affected by this change as well. The coefficient-wise multiplication becomes multiplication in $\mathbb{Z}_q[x]/(x^2-\omega_n^i)$. This multiplication is usually referred to as base multiplication. Two elements $a, b \in \mathbb{Z}_q[x]/(x^2-\omega_n^i)$ $(a = a_1x + a_0, b = b_1x + b_0)$ can be multiplied by

$$c = (a_1x + a_0)(b_1x + b_0) = (a_0b_0 + a_1b_1\omega_n^i) + (a_0b_1 + a_1b_0)x,$$

where $\omega_n^i$ is different for each of the $n/2$ multiplications.

**Why it is worth it.** Besides leaving more freedom for the choice of parameters, polynomial multiplication using incomplete NTT is often faster than complete NTT. This can be easily seen by counting operations. One butterfly costs one multiplication and two additions/subtractions, i.e., $n/2$ multiplications and $n$ additions for one layer of NTT. Since this is required for both operands and also the inverse transformation of the result, we end up needing $3 \cdot n/2$ multiplications and $3n$ additions per layer of NTTs. However, we also need to consider the cost of base multiplication: If polynomials are of degree 0, the base multiplication (pointwise multiplication) costs 1 multiplication and 0 additions. For 2-coefficient polynomials, this increases to five multiplications and two additions. Hence, by stopping one layer early,

we pay additional $4n/2$ multiplications and $2n/2$ additions, but save $3 \cdot n/2$ multiplications and $3n$ additions which results in a net saving of additions at the cost of additional multiplications. For platforms where multiplications are as cheap as additions, this results in a net speed-up.

**How incomplete should it be?** The optimal degree of incompleteness needs to be determined for each target platform separately. For example, on the Cortex-M4 it often makes sense [CHK+21] to stop two layers early as a $4 \times 4$ schoolbook implementation can be implemented rather efficiently using the available multiply-and-accumulate instructions. However, in case the NTT is being built into the specification of the cryptographic scheme (as it is the case for Kyber [ABD+17], and Dilithium [LDK+17]), the implementer is not given any choice.

**Application:** Kyber (since the second round of the NIST competition) is using incomplete NTTs to implement polynomial multiplication in $\mathbb{Z}_{3329}[x]/(x^{256}+1)$. Chapter 3 describes how to implement it efficiently on the Cortex-M4. Our implementations of Saber, NTRU, and LAC covered in Chapter 5 also make use of incomplete NTTs.

### 2.2.8 Good's Trick

Another trick that can be useful for polynomials that are not suitable for radix-2 FFTs, was proposed by Good [Goo51] which is referred to as *Good's trick* or *prime-factor FFT* in the literature. In the context of polynomials, it allows computing an FFT of a polynomial in $\mathbb{Z}_q[x]/(x^{p_0p_1}-1)$ with $p_0$ and $p_1$ being co-prime. A common choice is $p_1$ being a power of two and $p_0$ being a small prime (e.g., 3 or 5).

Good's trick maps $\mathbb{Z}_q[x]/(x^{p_0p_1}-1)$ to $\mathbb{Z}_q[y]/(y^{p_0}-1)[z]/(z^{p_1}-1)$ by setting $x = yz$, i.e., we present a polynomial in $\mathbb{Z}_q[x]/(x^{p_0p_1}-1)$ as $p_0$ polynomials in $\mathbb{Z}_q[z]/(z^{p_1}-1)$.

The mapping (usually called Good's permutation) between the two isomorphic rings corresponds to a reshuffling of the coefficients, such that $x^i = y^{i_0}z^{i_1}$. This can be seen as transforming a 1-dimensional array of $p_0p_1$ coefficients into a 2-dimensional array with dimensions $p_0 \times p_1$.

The forward Good's permutation uses

$$i_0 = i \mod p_0 \text{ and } i_1 = i \mod p_1.$$

To compute the inverse, we use the fact that $p_0$ and $p_1$ are co-prime and apply the CRT to obtain $i$ from $(i_0, i_1)$:

$$i = (p_1^{-1} \bmod p_0)p_1i_0 + (p_0^{-1} \bmod p_1)p_0i_1.$$

**Example 15:** Let $p_0 = 3$ and $p_1 = 2$. Given a polynomial $a = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5$, applying Good's permutation results in $a = (a_0 + a_3z) + (a_4 + a_1z)y + (a_2 + a_5z)y$.

When using Good's trick for polynomial multiplication of two polynomials $a$ and $b$, one proceeds as follows

1. Apply Good's permutation to both $a$ and $b$
2. Compute $p_1$-NTT for each of the $p_0$ components of both $a$ and $b$
3. Compute $p_0$-NTT for each of the $p_1$ components of both $a$ and $b$
4. Perform pointwise multiplication of the coefficients
5. Compute $p_0$-NTT$^{-1}$
6. Compute $p_1$-NTT$^{-1}$
7. Apply inverse Good's permutation

Steps (3) to (5) can alternatively be replaced by a schoolbook multiplication $\bmod (y^{p_0} - 1)$ which often turns out to be faster. To achieve competitive performance one will merge Good's permutation with the first layer of the NTT computation and, similarly, merge the inverse permutation with the last layer of the NTT$^{-1}$ computation.

**Application:** Good's trick can be beneficial for polynomial multiplication of NTRU [ZCH+19]. We make use of it in Chapter 6.