1 Number Theoretic Transform (NTT) for ML-DSA

The NTT converts polynomial multiplication from $O(n^2)$ to $O(n \log n)$, making lattice-based cryptography practical.

1.1 1. Setting

ML-DSA works in the polynomial ring:

$$R_q = \mathbb{Z}_q[X]/(X^{256}+1), \quad q = 8380417 \tag{1}$$

A polynomial $f \in R_q$ has 256 coefficients: $f = \sum_{i=0}^{255} f_i X^i$.

Multiplying two polynomials naively costs $O(n^2)$. The NTT reduces this to $O(n \log n)$ by evaluating at special points, multiplying pointwise, then interpolating back.

1.2 2. Core idea

The NTT is the finite-field analogue of the FFT. It exploits a primitive root of unity $\zeta = 1753$ in $\mathbb{Z}_q$ satisfying $\zeta^{256} \equiv -1 \bmod q$.

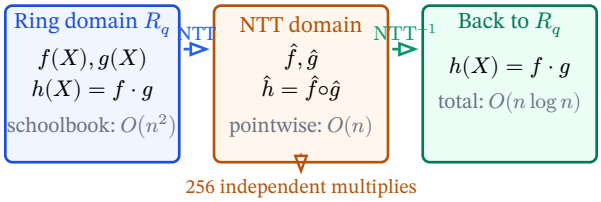

Figure 1: NTT converts ring multiplication to pointwise multiplication.

The key identity:

$$\text{NTT}^{-1}\left(\hat{f} \circ \hat{g}\right) = f \cdot g \in R_q \tag{2}$$

where $\circ$ denotes pointwise (coefficient-by-coefficient) multiplication.

1.3 3. Forward NTT (Algorithm 41)

The forward NTT uses the Cooley-Tukey butterfly. At each layer, pairs of elements are combined using a twiddle factor $\zeta^{\text{brv}(m)}$:

$$\begin{cases} w[j] & \leftarrow w[j] + \zeta^{\text{brv}(m)} \cdot w[j+\ell] \\ w[j+\ell] & \leftarrow w[j] - \zeta^{\text{brv}(m)} \cdot w[j+\ell] \end{cases} \tag{3}$$

where brv is the 8-bit bit-reversal permutation.

The transform proceeds through 8 layers ($\log_2 256 = 8$), halving the stride ℓ each time:

$\ell = (0, \text{"128"})(1, \text{"64"})(2, \text{"32"})(3, \text{"}\ell\text{"})(4, \text{"2"})(5, \text{"1"})$

Figure 2: Butterfly structure: 8 layers, stride halves each step.

Rust implementation (from `src/ntt.rs`):

```rust
fn ntt_layer<const LEN: usize, const ITERATIONS: usize>(
    w: &mut [Elem; 256], m: &mut usize,
) {
    for i in 0..ITERATIONS {
        let start = i * 2 * LEN;
        *m += 1;
        let z = ZETA_POW_BITREV[*m];  // twiddle factor
        for j in start..(start + LEN) {
            let t = z * w[j + LEN];
            w[j + LEN] = w[j] - t;    // butterfly subtract
            w[j]       = w[j] + t;    // butterfly add
        }
    }
}

// Full forward NTT: 8 layers, stride halves each time
fn ntt(f: &Polynomial) -> NttPolynomial {
    let mut w = f.coefficients();
    let mut m = 0;
    ntt_layer::<128, 1>(&mut w, &mut m);   // ℓ = 128
    ntt_layer::< 64, 2>(&mut w, &mut m);   // ℓ = 64
    ntt_layer::< 32, 4>(&mut w, &mut m);   // ℓ = 32
    ntt_layer::< 16, 8>(&mut w, &mut m);   // ℓ = 16
    ntt_layer::<  8,16>(&mut w, &mut m);   // ℓ = 8
    ntt_layer::<  4,32>(&mut w, &mut m);   // ℓ = 4
    ntt_layer::<  2,64>(&mut w, &mut m);   // ℓ = 2
    ntt_layer::<  1,128>(&mut w, &mut m);  // ℓ = 1
    NttPolynomial::new(w)
}
```

The const generics `LEN` (stride) and `ITERATIONS` (number of blocks) ensure all loop bounds are compile-time constants, avoiding runtime division.

1.4 4. Inverse NTT (Algorithm 42)

The inverse NTT uses the Gentleman-Sande butterfly, reversing the layer order and negating the twiddle factors:

$$\begin{cases} w[j] & \leftarrow w[j] + w[j+\ell] \\ w[j+\ell] & \leftarrow \left(-\zeta^{\mathrm{brv}(m)}\right) \cdot (w[j] - w[j+\ell]) \end{cases} \tag{4}$$

After all 8 layers, every coefficient is scaled by $256^{-1} \bmod q = 8347681$:

$$f_i = 256^{-1} \cdot w_i \bmod q \tag{5}$$

```rust
fn ntt_inverse(f_hat: &NttPolynomial) -> Polynomial {
    const INVERSE_256: Elem = Elem::new(8_347_681);
    let mut w = f_hat.coefficients();
    let mut m = 256;
    ntt_inverse_layer::<  1, 128>(&mut w, &mut m);  // ℓ = 1
    ntt_inverse_layer::<  2,  64>(&mut w, &mut m);  // ℓ = 2
    ntt_inverse_layer::<  4,  32>(&mut w, &mut m);  // ℓ = 4
    // ... layers 8, 16, 32, 64, 128
```

```
    ntt_inverse_layer::<128,   1>(&mut w, &mut m);  // ℓ = 128
    INVERSE_256 * Polynomial::new(w)
}
```

1.5 5. Pointwise multiplication (Algorithm 45)

In the NTT domain, polynomial multiplication reduces to pointwise multiplication of 256 coefficients:

$$\hat{h}_i = \hat{f}_i \cdot \hat{g}_i \bmod q, \quad i = 0, \ldots, 255 \tag{6}$$

This works because the NTT decomposes R_q into 256 copies of $\mathbb{Z}_q$.

```
fn multiply_ntt(f_hat: &NttPolynomial, g_hat: &NttPolynomial)
    -> NttPolynomial
{
    // Simply multiply corresponding coefficients
    NttPolynomial::new(
        f_hat.iter().zip(g_hat.iter())
            .map(|(&x, &y)| x * y)
            .collect()
    )
}
```

1.6 6. Twiddle factors

The twiddle factors are precomputed in bit-reversed order:

$$\text{ZETA\_POW\_BITREV}[i] = \zeta^{\text{brv}_8(i)} \bmod q \tag{7}$$

where brv_8 reverses the 8 bits of i. This ordering matches the access pattern of the butterfly, so the NTT reads twiddle factors sequentially ($m = 1, 2, 3, \ldots$).

Key constants:

- $\zeta = 1753$ (primitive 512th root of unity mod q)
- $\zeta^{128} \bmod q = 4808194$ (first twiddle factor, FIPS 204 Appendix B)
- $256^{-1} \bmod q = 8347681$

1.7 7. Why NTT matters for ML-DSA

ML-DSA key generation, signing, and verification all require matrix-vector products over $R_q^{k \times \ell}$. Each entry involves polynomial multiplication.

Without NTT: each multiplication is $O(n^2) = O(65536)$ operations.

With NTT: store the matrix $\boldsymbol{A}$ in NTT form permanently. Multiply via:

$$\boldsymbol{A} \cdot \boldsymbol{s} = \text{NTT}^{-1}\left(\hat{\boldsymbol{A}} \circ \text{NTT}(\boldsymbol{s})\right) \tag{8}$$

Cost per multiplication: $O(n \log n) = O(2048)$ — a 32x speedup.

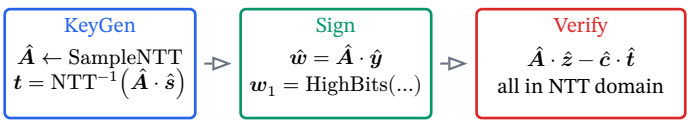

Figure 3: NTT usage in ML-DSA: matrix $\boldsymbol{A}$ stays in NTT domain.

1.8 8. Summary

Operation	Algorithm	Complexity
Forward NTT	Alg. 41 (Cooley-Tukey)	$O(n \log n)$
Inverse NTT	Alg. 42 (Gentleman-Sande)	$O(n \log n)$
Pointwise multiply	Alg. 45	$O(n)$
Naive poly multiply	schoolbook	$O(n^2)$

The NTT is the computational backbone of ML-DSA: it makes polynomial arithmetic fast enough for real-world digital signatures.