NUMBER THEORETIC TRANSFORM

Notes

Bhumika Mittal

Contents

1 Introduction

The US National Institute of Standards and Technology (NIST) started a competition to standardize cryptographic algorithms that are resistant to attacks by quantum computers (PQC) in 2016. The three finalist lattice-based standardized cryptosystems are Dilithium, Falcon and Kyber. But the bottleneck of these lattice-based cryptography implementations is its fundamental building block: modular polynomial multiplication, which is a very time-consuming operation. The classical method takes $O(n^2)$.

There are three faster alternatives:

- *Karatsuba algorithm:* Divide and conquers algo which divides the original polynomial into two parts, resulting in $O(n^{log_2^3})$
- *Toom-Cook algorithm:* Divide and conquers algo which divides the original polynomial into k parts, resulting in $O(n^{log_k^{2k-1}})$
- *DFT:* We will discuss about this in detail here.

2 DFT

2.1 Difference between NTT and DFT

The fundamental difference between DFT and NTT is the ring they use to transform the polynomial. DFT uses a complex ring with a twiddle factor of $e^{-2\pi j/n}$, whereas NTT uses an integer polynomial ring with a twiddle factor of its n-th root of unity. In NTT, there is no need to implement fixed-point or floating-point arithmetic architecture and also eliminates the precision problem associated with such implementations.

2.2 NTT and its classical version

NTT can be utilized to multiply two polynomials via convolution theorem (more on this later!). The standard method takes $O(n^2)$ but the CT and GS butterflies architecture for FFT has a quasilinear complexity of $O(n \log n)$

3 Convolutions

Just for the sake of lingo, there are three types of convolutions: linear, cyclic, and negacyclic. All the following standard methods have $O(n^2)$ time complexity.

3.1 Linear

Definition 3.1. Polynomial multiplication is equivalent to a **discrete linear convolution** between the coefficients' vectors f and g of two polynomials of degree $n-1$ in the ring $\mathbb{Z}_q[x]$ where $q \in \mathbb{Z}$

Example 3.2. Let $f(x) = 1 + 2x + 3x^2 + 4x^3$ and $g(x) = 5 + 6x + 7x^2 + 8x^3$. The vectors will be $f = [1, 2, 3, 4]$ and $g = [5, 6, 7, 8]$. The output of the linear convolution is $f \times g = [5, 16, 34, 60, 61, 52, 32]$

3.2 Cyclic

Definition 3.3. Let $f, g \in R_q = \frac{\mathbb{Z}_q[x]}{x^n - 1}$ where $q \in \mathbb{Z}$. Then **Cyclic convolution** is defined as $f \cdot g = f \times g \mod q \mod x^n - 1$. This is also called *positive wrapped convolution.*

Example 3.4. Let $f(x) = 1 + 2x + 3x^2 + 4x^3$ and $g(x) = 5 + 6x + 7x^2 + 8x^3$. The vectors will be $f = [1, 2, 3, 4]$ and $g = [5, 6, 7, 8]$.

$$
\begin{array}{rr}
 & 32x^2 + 52x + 61 \\ \cline{2-2}
x^4 - 1 \big) & 32x^6 + 52x^5 + 61x^4 + 60x^3 + 34x^2 + 16x \quad + 5 \\
 & -\,32x^6 \qquad\qquad\qquad\qquad + 32x^2 \qquad\qquad\quad \\ \cline{2-2}
 & 52x^5 + 61x^4 + 60x^3 + 66x^2 + 16x \qquad \\
 & -\,52x^5 \qquad\qquad\qquad\qquad + 52x \qquad \\ \cline{2-2}
 & 61x^4 + 60x^3 + 66x^2 + 68x \quad + 5 \\
 & -\,61x^4 \qquad\qquad\qquad\qquad + 61 \\ \cline{2-2}
 & 60x^3 + 66x^2 + 68x + 66
\end{array}
$$

The output of the cyclic convolution is $f.g = [66, 68, 66, 60]$

Note: The main difference between cyclic and negacyclic is just of the quotient ring.

3.3 Negacyclic

Definition 3.5. Let $f, g \in R_q = \frac{\mathbb{Z}_q[x]}{x^n + 1}$ where $q \in \mathbb{Z}$. Then **Cyclic convolution** is defined as $f \cdot g = f \times g \mod q \mod x^n + 1$. This is also called *negative wrapped convolution.*

Example 3.6. Let $f(x) = 1 + 2x + 3x^2 + 4x^3$ and $g(x) = 5 + 6x + 7x^2 + 8x^3$. The vectors will be $f = [1, 2, 3, 4]$ and $g = [5, 6, 7, 8]$.

$$\begin{array}{r|rrrrrrr}
\multicolumn{1}{r}{} & & & & & 32x^2 & +52x & +61 \\ \cline{2-8}
x^4+1 & 32x^6 & +52x^5 & +61x^4 & +60x^3 & +34x^2 & +16x & +5 \\
\multicolumn{1}{r}{} & -32x^6 & & & & -32x^2 & & \\ \cline{2-6}
\multicolumn{1}{r}{} & & 52x^5 & +61x^4 & +60x^3 & +2x^2 & +16x & \\
\multicolumn{1}{r}{} & & -52x^5 & & & & -52x & \\ \cline{3-7}
\multicolumn{1}{r}{} & & & 61x^4 & +60x^3 & +2x^2 & -36x & +5 \\
\multicolumn{1}{r}{} & & & -61x^4 & & & & -61 \\ \cline{4-8}
\multicolumn{1}{r}{} & & & & 60x^3 & +2x^2 & -36x & -56
\end{array}$$

The output of the cyclic convolution is $f.g = [-56, -36, 2, 60]$

4 Direct NTT Method - Vandermonde Matrix

The classical NTT (direct computation) has quadratic complexity as well.

4.1 Primitive n-th root of unity

Definition 4.1. Let Z_q be an integer ring modulo q, and $n-1$ is the polynomial degree of $f(x)$ and $g(x)$. Such rings have a multiplicative identity (unity) of 1. ω is defined as a **primitive n-th root of unity** in Z_q if and only if:

$$\omega^n \equiv 1 \pmod q$$

and

$$\omega^j \not\equiv 1 \pmod q$$

for $j < n$.

These roots are not unique and we can use the following code to find all such possible values of ω for the given q and n

```
def primitive_roots(q,n):
    Z_q_star_list = Z_q_star(q)
    w_list = []
    for w in Z_q_star_list:
        if w**n % q == 1 and all(w**j % q != 1 for j in range(1, n)):
            w_list.append(w)
    return w_list
```

4.1.1 NTT using ω

$$\begin{aligned}
\mathrm{NTT}_w(f) = \begin{bmatrix} f(1) \\ f(\omega) \\ \vdots \\ f(\omega^{n-1}) \end{bmatrix} &= \begin{bmatrix} a_0 + a_1 + \cdots + a_{n-1} \\ \vdots \\ a_0 + a_1\omega^{n-1} + \cdots + a_{n-1}(\omega^{n-1})^{n-1} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega^{n-1} & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \\
&= V(\omega) \cdot c(f)
\end{aligned}$$

Example 4.2. For the following parameters:

$$\begin{aligned} q &= 7 \\ \mathbb{Z}_q^* &= \{1,2,3,4,5,6,7\} \\ n &= 3 \\ \omega &= 2 \\ f(x) &= 1 + 2x + x^2 \end{aligned}$$

We get the following conversion using the direct method

$$c(f) = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \rightarrow \mathrm{NTT}(f) = \begin{bmatrix} f(1) \\ f(2) \\ f(4) \end{bmatrix} = \begin{bmatrix} 1+2+1 = 4 & \bmod 7 \\ 1+4+4 = 9 & \bmod 7 \\ 1+8+16 = 25 & \bmod 7 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 4 \end{bmatrix}$$

Implementation of this method is

```python
def cycNTT(f, q, w, n):
    ntt = []
    for i in range(n):
        ntt.append(0)
        #we need to calculate f(w^i) for i = 0 to n-1 and store it in
        ↪ ntt[i]
        for j in range(n):
            ntt[i] += f[j]*(w**(i*j))
        ntt[i] = ntt[i]%q
    return ntt
```

4.1.2 Inverse NTT using ω

We take the inverse of n and ω and map the input NTT vector to another vector. Then perform the NTT operation wrt to ω^{-1} on this mapped vector to get the original vector.

$$\begin{aligned} c(f) &= V^{-1} \cdot \mathrm{NTT}_w(f) \\ &= n^{-1} V(\omega^{-1}) \cdot \mathrm{NTT}_w(f) \\ &= V(\omega^{-1}) \cdot n^{-1}\mathrm{NTT}_w(f) \\ &= V(\omega^{-1}) \cdot \begin{bmatrix} n^{-1}f(1) \\ n^{-1}f(\omega) \\ \vdots \\ n^{-1}f(\omega^{n-1}) \end{bmatrix} \end{aligned}$$

Example 4.3. For the following parameters:

$$\begin{aligned} n^{-1} &= 5 \\ \omega^{-1} &= 4 \end{aligned}$$

$$\begin{bmatrix} 4 \\ 2 \\ 4 \end{bmatrix} \to \begin{bmatrix} 5.4 = 20 & \implies & 6 \\ 5.2 = 10 & \implies & 3 \\ 5.4 = 20 & \implies & 6 \end{bmatrix}$$

This means, we have $g(x) = 6 + 3x + 6x^2$

$$\begin{aligned} \mathrm{NTT}_{\omega^{-1}}(g) &= (g(1), g(\omega), g(\omega^2)) \\ &= (g(1), g(4), g(2)) \\ &= (1, 2, 1) \end{aligned}$$

Implementation of this method is

```
def cycINTT(ntt, q, w, n):
    nInverse = inverse(n,q)
    print ("nInverse = ",nInverse)
    wInverse = inverse(w,q)
    print ("wInverse = ",wInverse)
    intt = [0]*n
    for i in range(n):
        intt[i] = nInverse*ntt[i] % q
    intt = cycNTT(intt,q,wInverse,n)
    return intt
```

4.1.3 Cyclic Convolution using NTT

Let f and g be the vectors of the polynomial coefficients. The cyclic convolution of f and g, denoted as h, can be calculated by:

$$\mathbf{h} = \text{INTT}(\text{NTT}(\mathbf{f}) \circ \text{NTT}(\mathbf{g})),$$

where $\circ$ represents the element-wise vector multiplication in Z_q.

Example 4.4. Let $f = [1, 2, 3, 4]$ and $g = [5, 6, 7, 8]$. For $\mathbb{Z}_{41}$ and $\omega = 9$, we have $NTT(f) = [10, 21, 39, 16]$ and $NTT(g) = [26, 21, 39, 16]$. Also, note that $n^{-1} = 31 \in \mathbb{Z}_{41}$ and $\omega^{-1} = 32 \in \mathbb{Z}_{41}$.
We can calculate cyclic convolution as follows:

$$INTT\left(\begin{bmatrix}10\\21\\39\\16\end{bmatrix} \circ \begin{bmatrix}26\\21\\39\\16\end{bmatrix}\right) = INTT\left(\begin{bmatrix}260\\441\\1521\\256\end{bmatrix}\right) = INTT\left(\begin{bmatrix}14\\31\\4\\10\end{bmatrix}\right) = \begin{bmatrix}66\\68\\66\\60\end{bmatrix}$$

Implementation of this method is

```
def cycCon(f, g, q, w, n):
    fntt = cycNTT(f,q,w,n)
    gntt = cycNTT(g,q,w,n)
    hntt = [0]*n
    for i in range(n):
        hntt[i] = fntt[i]*gntt[i] % q
    h = cycINTT(hntt,q,w,n)
    return h
```

4.2 Primitive $2n$-th root of unity

Definition 4.5. Let Z_q be an integer ring modulo q, and $n-1$ is the polynomial degree of $f(x)$ and $g(x)$, and ω is its primitive n-th root of unity. ψ is defined as the primitive $2n$-th root of unity if and only if:

$$\psi^2 \equiv \omega \pmod q$$

and

$$\psi^n \equiv -1 \pmod q$$

.

Implementation of this method is

```
def primitive_root_2n(q,n,w):
    Z_q_list = []
    for i in range(1,q):
        Z_q_list.append(i)
    psi_list = []
    for psi in Z_q_list:
        if psi**2 % q == w and psi**n % q == q-1:
            psi_list.append(psi)
    print ("psi_list = ",psi_list)
    return psi_list
```

4.2.1 NTT using ψ

$$\begin{aligned}
\mathrm{NTT}_\psi(f) &= \sum_{i=0}^{n-1} \psi^{2ij+i} a_i \pmod q \\
&= \begin{bmatrix} 1 & \psi & \psi^2 & \cdots & \psi^{n-1} \\ 1 & \psi^3 & \psi^6 & \cdots & \psi^{3(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \psi^{2n-1} & \psi^{2(2n-1)} & \cdots & \psi^{(2n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \\
&= V_{odd}(\psi) \cdot c(f)
\end{aligned}$$

Example 4.6. For the following parameters:

$$\begin{aligned} q &= 17 \\ n &= 4 \\ \omega &= 4 \\ \psi &= 15 \\ f(x) &= 1 + 2x + 3x^2 + 4x^3 \end{aligned}$$

We get the following conversion

$$c(f) = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \rightarrow \mathrm{NTT}^{\psi}(f) = \begin{bmatrix} 1 & 15 & 15^2 & 15^3 \\ 1 & 15^3 & 15^6 & 15^9 \\ 1 & 15^5 & 15^{10} & 15^{15} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 11 \\ 16 \\ 15 \\ 13 \end{bmatrix}$$

Implementation of this method is

```python
def negacycNTT(f, q, psi, n):
    ntt = []
    for j in range(n):
        ntt.append(0)
        for i in range(n):
            ntt[j] += f[i]*(psi**((2*i*j)+i))
        ntt[j] = ntt[j]%q
    return ntt
```

4.2.2 INTT using ψ

$$\begin{aligned} \mathrm{INTT}_{\psi}(f) &= n^{-1} \sum_{j=0}^{n-1} \psi^{-(2ij+i)} \hat{a}_j \pmod q \\ &= n^{-1} \begin{bmatrix} 1 & \psi^{-1} & \psi^{-2} & \dots & \psi^{-(n-1)} \\ 1 & \psi^{-3} & \psi^{-6} & \dots & \psi^{-3(n-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & \psi^{-(2n-1)} & \psi^{-2(2n-1)} & \dots & \psi^{-(2n-1)(n-1)} \end{bmatrix}^T \begin{bmatrix} \hat{a_0} \\ \hat{a_1} \\ \vdots \\ \hat{a_{n-1}} \end{bmatrix} \\ &= V_{odd}(\psi)^T \cdot NTT^{\psi}(f) \end{aligned}$$

Example 4.7. For the following parameters:

$$n^{-1} = 13$$
$$\psi^{-1} = 8$$

$$\text{INTT}^{\psi^{-1}}(f) = 4^{-1} \begin{bmatrix} 1 & 15^{-1} & 15^{-2} & 15^{-3} \\ 1 & 15^{-3} & 15^{-6} & 15^{-9} \\ 1 & 15^{-5} & 15^{-10} & 15^{-15} \end{bmatrix}^T \begin{bmatrix} 11 \\ 16 \\ 15 \\ 13 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Implementation of this method is

```
def negacycINTT(ntt, q, psi, n):
    nInverse = inverse(n,q)
    print ("nInverse = ",nInverse)
    psiInverse = inverse(psi,q)
    print ("psiInverse = ",psiInverse)

    intt = []
    for i in range(n):
        intt.append(0)
        for j in range(n):
            intt[i] += ntt[j]*(psiInverse**((2*i*j)+i))
        intt[i] = (intt[i]*nInverse)%q
    return intt
```

4.2.3 Negacyclic Convolution using NTT$^{\psi}$

Let f and g be the vectors of polynomial coefficients. The negacyclic convolution of f and g, denoted as h, can be calculated by:

$$\mathbf{h} = \text{INTT}_{\psi^{-1}}(\text{NTT}_{\psi}(\mathbf{f}) \circ \text{NTT}_{\psi}(\mathbf{g})),$$

where $\circ$ represents the element-wise vector multiplication in Z_q.

Note: This is what we did as isomorphism method.

Example 4.8. Let $f = [1,2,3,4]$ and $g = [5,6,7,8]$. For $\mathbb{Z}_{17}$ and $\omega = 4, \psi = 15$, we have $NTT(f) = [11,16,15,13]$ and $NTT(g) = [8,15,7,7]$. Also, note that $n^{-1} = 13 \in \mathbb{Z}_{17}$ and $\psi^{-1} = 8 \in \mathbb{Z}_{17}$.
We can calculate cyclic convolution as follows:

$$INTT\left(\begin{bmatrix}11\\16\\15\\13\end{bmatrix} \circ \begin{bmatrix}8\\15\\7\\7\end{bmatrix}\right) = INTT\left(\begin{bmatrix}3\\2\\3\\6\end{bmatrix}\right) = \begin{bmatrix}12\\15\\2\\9\end{bmatrix}$$

Implementation of this method is

```
def negacycCon(f,g, psi, q, n):
    fntt = negacycNTT(f,q,psi,n)
    gntt = negacycNTT(g,q,psi,n)
    hntt = []
    for i in range(n):
        hntt.append((fntt[i]*gntt[i])%q)
    h = negacycINTT(hntt,q,psi,n)
    return h
```

5 Bit Reversal

Example 5.1. For $n = 4$, we have the following table

Index	Binary	Bit reversed	In decimal
0	00	00	0
1	01	10	2
2	10	01	1
3	11	11	3

Implementation is as follows:

```
#Input: list f
#Output: list f_rev such that f_rev[i] = f[bit_reversal(i)]
def bit_reversal(f):
    n = len(f)
    k = int(math.log(n,2))
    f_rev = []
    for i in range(n):
        f_rev.append(0)
    for i in range(n):
        f_rev[i] = f[int(bin(i)[2:].zfill(k)[::-1],2)]
    return f_rev
```

6 Quasilinear Methods

To reduce the complexity and fasten the process of the matrix multiplication needed for the NTT transformation, we can use 'divide and conquer' techniques by utilizing the periodicity and symmetry properties of the primitive roots.

6.1 Cooley-Tukey (CT) Algorithm

6.1.1 Recursive - NTT

Algorithm 1 Recursive Cooley-Tukey (CT) Algorithm

```
function RECURSIVECT($f, w, q$)
    $n \leftarrow$ length of $f$
    if $n = 1$ then
        return $f$
    else
        $r_1 \leftarrow []$
        $r_2 \leftarrow []$
        for $i \leftarrow 0$ to $n$ do
            if $i \bmod 2 = 0$ then
                $r_1$.append($f[i]$)
            else
                $r_2$.append($f[i]$)
            end if
        end for
        $r_1 \leftarrow$ recursiveCT($r_1, (w^2) \bmod q, q$)
        $r_2 \leftarrow$ recursiveCT($r_2, (w^2) \bmod q, q$)
        $r \leftarrow [0] * n$
        for $i \leftarrow 0$ to $n//2$ do
            $r[i] \leftarrow (r_1[i] + \text{pow}(w, i, q) \times r_2[i]) \bmod q$
            $r[i + n//2] \leftarrow (r_1[i] - \text{pow}(w, i, q) \times r_2[i]) \bmod q$
        end for
        return $r$
    end if
end function
```

Add example and maybe mathematical understanding here - will need to redo this probably

6.1.2 Iterative - NTT

The main idea is to calculate similar terms once and then distribute the results instead of calculating them multiple times.

We can observe the following properties of ψ,

$$\text{Periodicity:} \psi^{k+2n} = \psi^k$$

$$\text{Symmetry:} \psi^{k+n} = -\psi^k$$

where $k \geq 0$. Now based on this, we can use the NTT notation from negacyclic version. Recall that, $NTT^{\psi}(f) = \hat{a}$

$$\begin{aligned}
\hat{a_j} &= \sum_{i=0}^{n-1} \psi^{2ij+i} a_i \pmod{q} \\
&= \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} + \sum_{i=0}^{n/2-1} \psi^{4ij+2i+2j+1} a_{2i+1} \pmod{q} \\
&= \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} + \psi^{2j+1} \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1} \pmod{q}
\end{aligned}$$

Let $A_j = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i}$ and $B_j = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1} \pmod{q}$. Then, using the symmetry we have,

$$\begin{aligned}
\hat{a_j} &= A_j + \psi^{2j+1} B_j \pmod{q} \\
\hat{a}_{j+n/2} &= A_j - \psi^{2j+1} B_j \pmod{q}
\end{aligned}$$

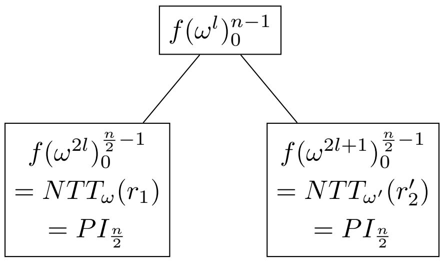
6.2 Gentleman-Sande (GS) Algorithm

CT is odd-even divide and GS is upper-lower half divide. We generally use CT for NTT and GS for INTT (why tho?)

6.2.1 Recursive - NTT

Let's define the problem as follows:
Input: $f(x) \in R_q, \omega$
Output: $f(\omega^l)_0^{n-1}$

Problem Instance:



$$\begin{aligned}
f(x) &= (x^{\frac{n}{2}} + 1) q_2(x) + r_2(x) \\
f(\omega^{2l+1}) &= ((\omega^{2l+1})^{\frac{n}{2}} + 1) q_2(\omega^{2l+1}) + r_2(\omega^{2l+1}) \\
&= r_2(\omega^{2l+1}) \\
&= r_2'(\omega'^{l})
\end{aligned}$$

Therefore,

$$f(x) = (x^{\frac{n}{2}} + 1) F_1 + (F_0 - F_1)$$

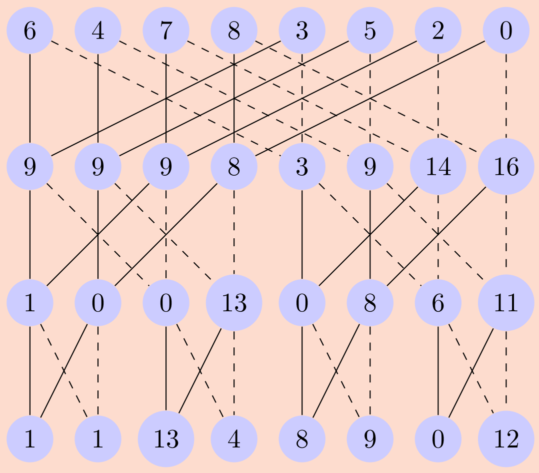
Remarks:

- If ω is the n-th root, then ω^2 is the $\frac{n}{2}$-th root
- $\omega^{\frac{n}{2}} + 1 = 0$ and $r_2'(x) = r_2(\omega x)$

Algorithm 2 NTT based on the Gentleman-Sande (GS)

```
function RECURSIVEGS($f, w, q$)
    $n \leftarrow$ length of $f$
    if $n = 1$ then
        return $f$
    else
        $r_1 \leftarrow []$
        $r_2 \leftarrow []$
        for $i \leftarrow 0$ to $n//2$ do
            $r_1$.append($(f[i] + f[i + n//2]) \bmod q$)
            $r_2$.append($(f[i] - f[i + n//2]) \times \text{pow}(w, i, q) \bmod q$)
        end for
        $r_1 \leftarrow$ recursiveGS($r_1, (w^2) \bmod q, q$)
        $r_2 \leftarrow$ recursiveGS($r_2, (w^2) \bmod q, q$)
        $r \leftarrow r_1 + r_2$
        return $r$
    end if
end function
```

Example 6.1. (Butterfly Algo) Let $n = 8 = 2^3$. $q = 17$



6.2.2 Iterative - INTT

check math again - probably some error here

Recall that $a = INTT(\hat{a})$. Ignoring n^{-1}, we have

$$\begin{aligned}
\hat{a_i} &= \sum_{j=0}^{n-1} \psi^{-(2j+1)i} \hat{a}_j \pmod q \\
&= \sum_{j=0}^{n/2-1} \psi^{-(2j+1)i} \hat{a}_j + \sum_{j=n/2}^{n-1} \psi^{-(2j+1)(i+n/2)} \hat{a}_{j+n/2} \pmod q \\
&= \psi^{-i} \left[\sum_{j=0}^{n/2-1} \psi^{-2ij} \hat{a}_j + \sum_{j=0}^{n/2-1} \psi^{-2j(i+n/2)} \hat{a}_{j+n/2} \right] \pmod q
\end{aligned}$$

For the even term,

$$a_{2i} = \psi^{-2i} \sum_{j=0}^{n/2-1} \left[\hat{a}_j + \hat{a}_{j+n/2} \right] \psi^{-4ij} \pmod q$$

For the odd term,

$$a_{2i+1} = \psi^{-2i} \sum_{j=0}^{n/2-1} \left[\hat{a}_j - \hat{a}_{j+n/2} \right] \psi^{-4ij} \pmod q$$

Let $A_i = \sum_{j=0}^{\frac{n}{2}-1} \hat{a}_j \psi^{-4ij}$ and $B_i = \sum_{j=0}^{\frac{n}{2}-1} \hat{a}_{j+n/2} \psi^{-4ij}$.

$$\begin{aligned}
a_{2i} &= (A_i + B_i)\psi^{-2i} \pmod q \\
a_{2i+1} &= (A_i - B_i)\psi^{-2i} \pmod q
\end{aligned}$$

7 Polynomial Multiplication

For polynomial multiplication , do the following:

code

1. Use CT butterflies to transform inputs to the NTT domain
2. Then element-wise multiply NTT outputs
3. The result is then transformed back using GS butterflies for INTT.

By reducing mathematical operations quasi-linearly, this approach reduces polynomial multiplication complexity from $O(n^2)$ to $O(n \log n)$.

$$\begin{aligned}
f &\to NTT(f) \\
g &\to NTT(g) \\
NTT(f).NTT(g) &= NTT(f \cdot g) \\
INTT(f \cdot g) &\to f \cdot g
\end{aligned}$$

This takes $\mathrm{O}(n \log n) + \mathrm{O}(n) + \mathrm{O}(n \log n) = \mathrm{O}(n \log n)$

Theorem 7.1. $NTT(fg) = NTT(f) \cdot NTT(g)$

Proof. Observe that

$$\begin{aligned}
NTT(fg) &= (fg(1), fg(\omega), \cdots, fg(\omega^{n-1}) \\
NTT(f) &= (f(1), f(\omega), \cdots, f(\omega^{n-1}) \\
NTT(g) &= (g(1), g(\omega), \cdots, g(\omega^{n-1}) \\
\implies fg(\omega^i) &= f(\omega^i)g(\omega^i) \forall i \in \{0, 1, \cdots, n-1\}
\end{aligned}$$

$$\begin{aligned}
f_n g_n &= (x^n - 1)q(x) + r(x) \\
\implies fg &= r(x) = f(x)g(x) - (x^n - 1)q(x)
\end{aligned}$$

$$\begin{aligned}
fg(\omega^i) &= r(\omega^i) \\
&= f(\omega^i)g(\omega^i) - ((\omega^i)^n - 1)q(\omega^i) \\
&= f(\omega^i)g(\omega^i)
\end{aligned}$$

□

8 Modular Reduction

By now, we can clearly notice that all the CT and GS butterflies operators require modular arithmetic. But this is not a standard operation for most platforms.

While modular added can be implemented using a set of adders, subtractors, and multiplexers, modular multiplication uses trial division, which is ineffi-cient, not scalable, and difficult to implement in hardware architecture.
The following algorithm is the most popular workaround to solve this.

8.1 Barrett reduction

The main idea behind Barrett reduction is to approximate the division by the modulus using pre-computed values (since q is generally standard/fixed, we can pre-computed k, r, μ), which allows for faster modular multiplication.

8.1.1 Modular Multiplication by Barrett Reduction

Given, $a, b, q \in \mathbb{Z}$, we can calculate the following:

1. $k = \lceil \log_2 g \rceil$: number of bits in q
2. $r = 2^k$
3. $\mu = \lfloor \frac{r^2}{q} \rfloor$
4. $z = a \times b$

Algorithm 3 Barrett reduction

```
1: $m_1 \leftarrow \lfloor \frac{z}{r} \rfloor$
2: $m_2 \leftarrow m1 \times \mu$
3: $m_3 \leftarrow \lfloor \frac{m_2}{r} \rfloor$
4: $t \leftarrow z - m_3 \times q$
5: if $t \geq q$ then
6:     return $t - q$
7: else
8:     return $t$
9: end if
```

8.1.2 Speeding up the NTT using Modular Reduction

Since we need primitive $2n$-th roots of unity to exist $\pmod{q}$, we have to choose the value of $q \ni q \equiv 1(2n)$.
Then $q = k \cdot 2^m + 1$.

$$\begin{aligned} 0 \leq a, b \leq q \\ a, b \in \mathbb{Z}_q \\ c = a \cdot b \\ 0 \leq c < q^2 = k^2 2^{2m} + k \cdot 2^{m+1} + 1 \end{aligned}$$

We need to find $C \pmod{q}$. We will use the fact that $k \cdot 2^m = -1 \pmod{q}$

$$\begin{aligned} C &= C_0 + 2^m C_1, 0 \leq C_0 < 2^m \\ \implies C_1 &= \frac{C - C_0}{2^m} \\ \implies 0 &\leq C_1 < C \\ \text{Note that, } C &= \frac{k^2 2^{2m} + 2k2^m + 1}{2^m} \\ &= k^2 2^m + 2k + \frac{1}{2^m} \\ &= kq + k + \frac{1}{2^m} \\ \implies 0 &\leq C_1 < kq + k + \frac{1}{2^m} \end{aligned}$$

Therefore,

$$\boxed{C = C_0 + 2^m C_1}$$
$$\boxed{0 \leq C_0 < 2^m}$$
$$\boxed{0 \leq C_1 < kq + k + \frac{1}{2^m}}$$

We know that $C = C_0 + 2^m C_1$, multiplying both sides by k,

$$\begin{aligned} kC &= kC_0 + k2^m C_1 \\ &= kC_0 + k2^m C_1 + C_1 - C_1 \\ &= qC_1 + (kC_0 - C_1) \\ \implies & \boxed{kC \equiv kC_0 - C_1 \pmod q} \end{aligned}$$

This gives an absolute bound,

$$|kC_0 - C_1| < \left(k + \frac{1}{2^m}\right) q$$

For $C_0 = 0$ and $C_1 = k(q-1)$, we get the max value of C as follows:

$$C_{max} = (q-1)^2$$

Hence, $(k-1)q$ must be added to $kC_0 - C_1$ to fully reduce the result.

Algorithm 4 K-RED function

```
function K-RED($C$)
    $C_0 \leftarrow C \pmod{2^m}$
    $C_1 \leftarrow C/2^m$
    return $kC_0 - C_1$
end function
```

insert some example here for the function maybe

Appendices

A NTT Basics - From Lecture Notes

For some $q \in \mathbb{N}$, we can define

$$R_q = \frac{\mathbb{Z}_q[x]}{x^n - 1}$$

Here R_q is a ring of polynomials with coefficients in $\mathbb{Z}_q$ modulo $x^n - 1$. The degree of these polynomials is less than n. Number of elements in R_q is q^n.

Let $f, g \in R_q$. Then $f \cdot g \in R_q$ is defined as $f \cdot g = f \times g \mod q \mod x^n - 1$ and $f + g = (f + g) \mod q$

$$h = fg = (x^n - 1)q(x) + r(x) \implies h \mod (x^n - 1) = r(x)$$

We want to optimize the multiplication of polynomials in R_q, i.e. $f \cdot g$ should take $\mathrm{O}(n \log n)$ time, not $\mathrm{O}(n^2)$.

Definition A.1. Let $f \in R_q, \omega \in \mathbb{Z}_q^*$. The order of $\omega = n \in \mathbb{Z}_q^*$. This means ω is relatively prime to q.

$$\omega^n = 1 \mod q$$

$\mathrm{DFT}_\omega(f) = (f(1), f(\omega), f(\omega^2), ..., f(\omega^{n-1})) \in \mathbb{Z}_q^n$

B Another Ring: $\mathbb{Z}_q[x]/ < x^n + 1 >$

Let $f, g \in R_q$. We have $q \equiv 1(2n), n = 2^k, k \in \mathbb{N}$.
Let $H \leqslant \mathbb{Z}_q^*$, $|H| = 2n = 2^{k+1}$ and $< H >= \{1, \omega, \omega^2, \cdots, \omega^{2n}\}$

Definition B.1. Number of generators of H = $\phi(2n)$

$$\begin{aligned}\phi(2n) &= \phi(2^{k+1}) \\ &= 2^{kn} - 2^k \\ &= 2^k(2-1) \\ &= n\end{aligned}$$

Hence, the generators of H are $\{\omega, \omega^3, \omega^5, \cdots, \omega^{2n-1}\}$

Since, ω is the primitive root of unity of order $2n$, $w^{2n} - 1 = 0 \implies w^n - 1 = 0$

$$x^n + 1 = (x - \omega)(x - \omega^3)(x - \omega^5) \cdots (x - \omega^{2n-1})$$

B.1 Isomorphism

Let us define an isomorphism I as follows:

$$I : \frac{\mathbb{Z}_q[x]}{< x^n + 1 >} \cong \frac{\mathbb{Z}_q[x]}{< x - \omega} \times \frac{\mathbb{Z}_q[x]}{< x - \omega^3 >} \cdots \frac{\mathbb{Z}_q[x]}{< x - \omega^{2n-1} >}$$

$$\begin{aligned} f &\mapsto I(f) \\ g &\mapsto I(g) \\ fg &\mapsto I(f)I(g) \\ I(fg) &\mapsto fg \end{aligned}$$

$$\boxed{fg = I^{-1}(I(f) \cdot I(g))}$$

B.2. How to compute I?
Input: f
Output: $I(f) = f(\omega^{2l+1})_0^{n-1}$

Define:

$$\begin{aligned} f' &= f(\omega x) \\ f(\omega^{2l+1}) &= f(\omega \cdot \omega^{2l}) \\ &= f'(\omega^{2l}) \\ &= f'(\bar{\omega}^l) \end{aligned}$$

$$\boxed{fg = (1, \bar{\omega}^{-1}, (\bar{\omega}^{-1})^2, \cdots, (\bar{\omega}^{-1})^{n-1}) \cdot NTT_{\bar{\omega}}(n^{-1} \cdot NTT_{\bar{\omega}}(f') \cdot NTT_{\bar{\omega}}(g'))}$$