CS674A: Post Quantum Security

Assignment 1, READ-me

Souvik Mukherjee | 231110405

The question asks us to compute 3 things

- Firstly, we're given 2 functions, we are said to compute the Fourier transform for each one of them (using the Cooley-Tukey NTT algorithm).
- Secondly, we're asked to perform point wise multiplication to the transformed functions.
- Lastly, we're asked to do inverse NTT on the last output. (Using the Gentleman-Sande inverse INTT algorithm). *Let this answer be stored in (final)*

Verification

We need to compute convolution of the input polynomials. This answer is then asked to be bounded by a negative wrap (i.e. $R_q = Z_q[X]/X_{n+1}$).This ensures we again have a polynomial of degree at-most 'n', as convolution may increase degree. *Let this answer be stored in (ans)*

Both these (ans) and (final) needs to be same for our code to be correct.

Equations:

1. $q \equiv 1 \bmod 2n$
2. $\gamma = \sqrt{w}$
3. $w^n \equiv 1 \bmod q$

$Z_q \{0 .. q-1\}$

* $\gamma^2 \bmod q = w$

o/p   i/p

$$\underbrace{\tilde{A}_j}_{\text{Freq domain}} = \sum_{i=0}^{n-1} \gamma^i \cdot \omega_n^{ij} \cdot \underbrace{A_i}_{\text{Time domain}}$$

NTT (T → f)
o/p in bit reversed order

$$\underbrace{A_j}_{\text{Time domain}} = \frac{1}{n} \gamma^{-j} \sum_{i=0}^{n-1} \omega_n^{-ij} \cdot \underbrace{\tilde{A}_j}_{\text{Freq domain}}$$

INTT (f → T)
i/p is bit reversed

Verification:

$$\boxed{NTT^{-1}(NTT(A) . NTT(B)) = \text{Negative Wrapped Convolution } (A, B)}$$

Steps Followed in the code

1. We have the following input in the question ($n = 512$, $q = 12289$ (Finite Field, Z_{12289}), $r = 10968$ ($2n^{th}$ root of unity)

```python
n = 512
q = 12289   #(Z 12289)
r = 10968   #gamma
```

2. Using "np.random" I am creating two random arrays of 512 size. It's elements are bounded by 'q'

```python
p1 = np.random.randint(0, q, n)
p2 = np.random.randint(0, q, n)
```

3. I have then calculated our actual expected answer, using "np.polymul, np.polydiv, np.remainder", in which I have multiplied our input polynomials and divided with x_n+1 (will be used later to verify). I have stored this answer in a variable named "ans"

```python
#   *************************   Negative-wrapped Convolution (Verification) ***********
f = np.zeros(n + 1)
f[0] = 1
f[n] = 1
ans = np.remainder(np.polydiv(np.polymul(p1[::-1],p2[::-1]), f)[1], q).astype(int)[::-1]
```

4. I have created the gamma and gamma-inverse array which have values like $[1, \gamma, \gamma^2, \gamma^3, \ldots, \gamma^{n-1}]$ and $[1, \gamma^{-1}, \gamma^{-2}, \gamma^{-3}, \ldots, \gamma^{-(n-1)}]$ respectively. I have arranged them into Bit-Reversed sequence.

   A. Gamma- Array

```python
gamma=[]
for i in range(n):
    gamma.append((r**i)%q)
print("\n")
print("The Gamma array: ",gamma)

def BiReA(po):

gamma=BiReA(gamma)
print("")
print("The Gamma array after Bit Reverse is: ",gamma)
```

B. Gamma-Inverse Array

```
gammainv=[]
for i in range(n):
    gammainv.append((r_inv**i)%q)

print("")
print("The Gamma Inverse array: ",gammainv)
gammainv=BiReA(gammainv)
print("")
print("'Gamma Inverse Array' after Bit Reverse: ",gammainv)
```

5. The “BiReA” function does the work of Bit reversing.

```
def BiReA(po):
    df=[]
    dfd=[]
    for i in po:
        df.append(i)
    for i in range(n):
        s=str(bin(i)[2:])
        g=int(math.log(n,2))-len(s)
        for j in range(g):
            s="0"+s
        s=s[::-1]
        s=int(s,2)
        dfd.append(df[s])
    return(dfd)
```

6. The NTT takes input in normal ordering, and gamma in bit-reversed order and gives output in bit-reversed order. This bit-reversed ordered output after PWM (point wise multiplication) is fed to INTT (inverse NTT), which also takes gamma-inverse in bit - reversed order, and also takes in “n-inverse”. This function gives out output in normal ordering, which is present in a variable named “final”

A. <u>NTT Code Snippet</u>

```
def NTT(pq):
    t=n
    m=1
    while(m<n):
        t=t//2
        for i in range(m):
            j1=2*i*t
            j2=j1+t-1
            s=gamma[m+i]
            for j in range(j1,j2+1):
                u=pq[j]
                v=pq[j+t]*s
                pq[j]=(u+v)%q
                pq[j+t]=(u-v)%q
        m=2*m
    return(pq)

x=NTT(p1)
y=NTT(p2)

print("")
print("'P1' after NTT: ",x)
print("")
print("'P2' after NTT: ",y)
```

B. <u>INTT Code Snippet</u>

```
def INTT(po):
    t=1
    m=n
    while(m>1):
        j1=0
        h=m//2
        for i in range(h):
            j2=j1+t-1
            s=gammainv[h+i]
            for j in range(j1,j2+1):
                u=po[j]
                v=po[j+t]
                po[j]=(u+v)%q
                po[j+t]=((u-v)*s)%q
            j1=j1+(2*t)
        t=2*t
        m=m//2
    for i in range(n):
        po[i]=(po[i]*n_inv)%q
    return(po)

final=INTT(p)
```

C. PWM (Point Wise Multiplication) Code Snippet

```
#    *************************    Point wise multiplication    *************************

def PWM(p11,p12):                                                  #Point wise multiplication
    pw=[]
    for i in range(len(p11)):
        pw.append((p11[i]*p12[i])%q)
    return(pw)

p=PWM(x,y)
print("")
print("The array after 'NTT of P1 & P2', and 'Point Wise Multiplication of NTT' : ",p)

```

7. These two (final & ans) are now checked for equality. My code gives out "True" for all random arrays, hence the code can be believed to be accurate (verification: verified)

```
x=NTT(p1)
y=NTT(p2)
```

```
p=PWM(x,y)
```

```
final=INTT(p)
```

```
ans = np.remainder(np.polydiv(np.polymul(p1[::-1],p2[::-1]), f)[1], q).astype(int)[::-1]
```

```
print("")
print("'Final array, after 'NTT of P1 & P2', 'Point Wise Multiplication of NTT', and 'Inverse-NTT of the o/p from NWC':",final)
print("")

print("After negative wrapping 'i.e. after Modulo(X^n+1)' (Actual Answer Array for Verification): ",ans)
print("")
print(" **** Verification, whether \"NTT, PWM and INTT\" is same to \"Negative Wrapped Convolution **** ")
print("Both are same: ",np.array_equal(final, ans))
```

8. I have used the NTT & INTT in the butterfly fashion (Iterative Divide & Conquer fashion), to bring down the computation time complexity from θ(n^2) to θ(nlogn)

CASE A: In the trivial case, for an element, we'll need to multiply to each element, TC: θ(n), and do that for all 'n' elements. So, TC: θ(n^2)

DFT : Time → Frequency

$$\boxed{X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{kn}}$$

$$X(k) = x(0)W_N^{k\cdot 0} + x(1)W_N^{k\cdot 1} + x(2)W_N^{k\cdot 2} + \cdots$$

$$\underbrace{\cdots + x(N-1)W_N^{k\cdot(N-1)}}_{\text{In total, there's } N \text{ computations.}}$$

CASE B: In the case of butterfly, we use Divide and conquer approach, we go on dividing, until we're left with 2 elements, and while conquering, we'll solve for all the 2 elements parallelly, and then 4, then 8 and so on upto n elements in the last layer.

Here, Time complexity becomes:

Layer 0: (2 adds, 1 multiplication) * n/2 such operations TC: [(1*(n/2))]

Layer 1: 2 multiplications * n/2 such operations TC: (2*(n/4))

…. So on…

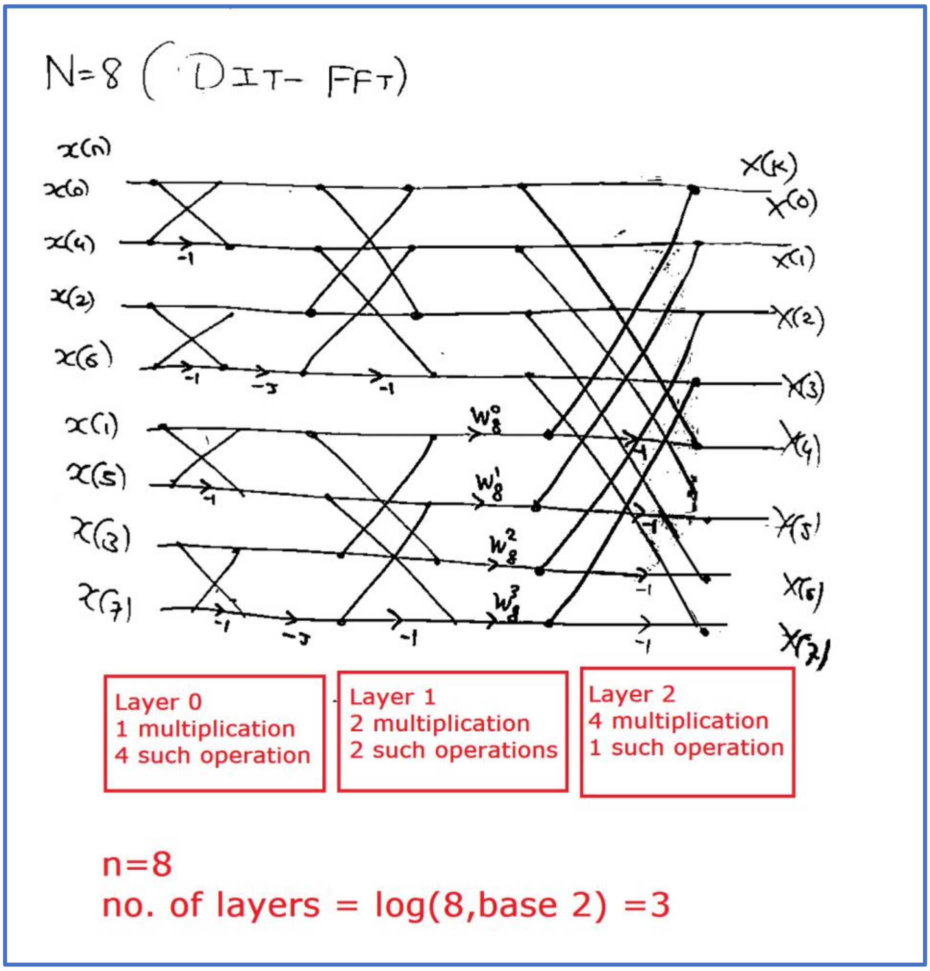
Last Layer: n/2 multiplications * 1 such operation

The mathematical equation we get is like:

(1*(n/2))+ (2*(n/4)) + (4*(n/8)) … (n/2 *(1)) = (n/2)*(depth of layers)

For n elements, we get a layer depth of logn

So, **TC: θ((n/2)*logn) = θ(nlogn)**



9. The “gamma-inverse” and “n-inverse” are the not power(-1), but multiplicative inverse in the finite field of length ‘q’. This is nothing but, for n-inverse “ $n*n^{-1}$ log(q)=1 ”

```
# gamma inverse (r*r_inv)mod q=1
r_inv=1
for i in range(2,q):
    if(i*r)%q==1:
        r_inv=i
        exit

# n inverse (n*n_inv)mod q=1
n_inv=1
for i in range(2,q):
    if(i*n)%q==1:
        n_inv=i
        exit

```

<u>My Code Output Snippet:</u>

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

  9722 11847  4886  9293  5222  6151  4091  1562  6903  7426  7026  2732
  2930 11124 11953  1998  5394  4593  7054  2604  7547  6410 11093   716
 10510  5769 11376  4827  8268  5951 10924 10798  2936  2519  9953  1859
  1828   310  9840  6920  5600  1119 10400  6437  1003  9104  8889  1821
 11993  9570   990  9759  9541 11644 10347   664  5184 11258  6612  9301
  5019  9430 11750  8276  2436  3020  3713 10907  4274  3249 11456  7049
 12221  6429  5375  9664 11934  5332 11790  6501]

 **** Verification, whether "NTT, PWM and INTT" is same to "Negative Wrapped Convolution ****
Both are same:  True
PS C:\Users\souvi\OneDrive\Desktop\Final>
```

<u>References:</u>

[1] Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography -*Patrick Longa and Michael Naehrig, Microsoft Research, USA*

[2] A note on the implementation of the Number Theoretic Transform -*Michael Scott*

************************* ***<u>Thank -You</u>*** *************************