Applications and Implementation of Number Theoretic Transforms

Discrete Fourier Transformation in modular arithmetics

Divyansh Verma
B.Tech. in Computer Engineering
National Institute of Technology, Karnataka
(Surathkal),India
dvbelieve.nitk@gmail.com

Prince Abhinav
B.Tech. in Computer Engineering
National Institute of Technology, Karnataka
(Surathkal),India
princeabhinav@gmail.com

Abstract - Today multiplying big numbers are really required for many computational need. It has become necessity for many fields like neural network domains where data can occur in the form of polynomials or big integral number multiplication may be required and for those things researchers have found many algorithms like Kasturbha algorithm, fast fourier transformation, number theoretic transforms and many others. fast fourier transformation and number theoretic transforms are the fastest way to compute such multiplication.

Objective - In this paper our aim is that we are going to show many other important applications of fast fourier transformation and number theoretic transforms (specially number theoretic transforms). First we will try to discuss implementation of fast fourier transformation and number theoretic transforms relation between and how one's algorithm can be changed to other. Then we will discuss some applications of number theoretic transforms like evaluation of a polynomial at n points, expected productivity value problems, some kind of knapsack problem and other problems. Then comparing their time complexities with other algorithms.

Result - The outcome of them paper is to present the usefulness of NTT in computational problems. Comparison with other algorithms , more faster and more precision accurate implementation and time complexities.

Keywords— Fast Fourier Transformation(FFT); Discrete Fourier Transforms (DFT); Numeric Theoretic Transforms(NTT); Inverse(INV); Primitive Roots

I. Introduction

Numeric Theoretic Transform's real objective was to multiply two polynomials of given degree and given in a form of array. After passing it through NTT it should give back coefficient of final polynomial obtained after multiplying two polynomials. DFT's are completely dependent on floating point and complex number (i.e. nth power of unity) operations which needs better computational speed or CPU performance but this is not in the case of NTT. For NTT we use a technique in which all the numbers between 1 to p-1 (where p is prime and is used for modulo operation in NTT) can be generated using exponentiation of numbers in between 1 to p-1. Thus there will be no floating pointings which will be used for implementation so much more precision and less time complexity.

II. Discrete Fourier Transformation

Discrete Fourier Transformation (DFT) helps in evaluation of polynomial at various points. It is completely based on complex numbers specially complex roots of unity.

DFT has two function :-

1. DFT which uses coefficient of polynomials and evaluates them at all complex roots of nth degree of unity. (Time domain to frequency domain).
2. INVDFT which used output of DFT to polynomial coefficient (Frequency domain to time domain).

$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 + \ldots\ldots\ldots + a_{n-1}x^{n-1}$$

$$\textit{Complex root of unity}: w_{n,k} = e^{i2\pi/n} \quad \text{Eq2.}$$

$$\text{DFT of } A(x) = DFT(a_0 + a_1 + a_2 + \ldots + a_{n-1}) =$$

$$(y_0 + y_1 + y_2 + y_2 + \ldots\ldots + y_{n-1}) =$$

$$(A(w_n^0, w_n^1, w_n^2, w_n^3, ..., w_n^{n-1}) \quad \text{Eq2.}$$

$$\text{INVDFT}(y_0, y_1, y_2, y_3, ..., y_{n-1}) = (a_0, a_1, a_2, ...a_{n-1})$$

Eq3.

It is completely based on floating points and complex roots of unity.

III. Fast Fourier Transformation

Fast Fourier Transformation is a technique to calculate DFT in O(nlogn). This method is completely based on complex roots of unity (i.e. we can get one root from the other). The algorithm is to divide the polynomial into parts until one is left and recursively compute DFT and join those parts. (Divide and Conquer)

$$A(x) = a_0x^0 + a_1x^1 + ... + a_{n-1}x^{n-1} \quad \text{Eq4.}$$

1. Divide it into two :-

$$A_0(x) = a_0x^0 + a_2x^1 + ... + a_{n-2}x^{n/2-1} \quad \text{Eq5.}$$

$$A_1(x) = a_1x^0 + a_3x^1 + ... + a_{n-1}x^{n/2-1} \quad \text{Eq6.}$$

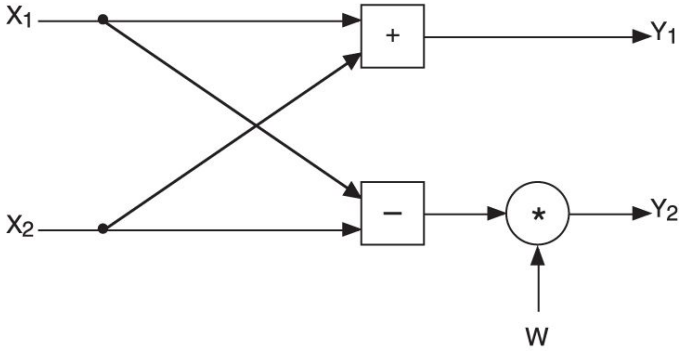
2. Relation between Eq4., Eq5. and Eq6.

$$A(x) = A_0(x^2) + A_1(x^2) \quad \text{Eq7.}$$

Outputs

- $\{y_k^0\}_{k=0}^{n/2-1} = DFT(A_0)$
- $\{y_k^1\}_{k=0}^{n/2-1} = DFT(A_1)$
- $\{y_k\}_{k=0}^{n-1} = DFT(A)$

3. Applying Butterfly Operation



$$y_k = y_k^0 + w_n^k y_k^1 \quad \text{Eq8.}$$

$$y_{k+n/2} = y_k^0 - w_n^k y_k^1 \quad \text{Eq9.}$$

4. FFT using matrix multiplication

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

5. Inverse FFT in matrix Form

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

6. Inverse of matrix can be found as

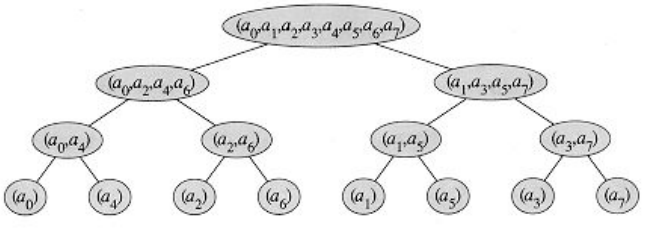
$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}.$$

7. Final Formulas

$$1. \quad y_k = \sum_{j=0}^{n-1} y_j w_n^{kj} \qquad 2. \quad a_k = (1/n) \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

The Recursive algorithm can be easily implemented and used based on above formulas. But we are most interested is Iterative algorithm as they have less time factors to reduce time complexity.

Now it can be seen that we are dividing the vector as shown in figure



So this division leads to vector of form

$$\{ a_0 , a_4 , a_2 , a_2 , a_1 , a_5 , a_3 , a_7 \}$$

whose index can be easily found by reversing bits of

$$\{ a_0 , a_1 , a_2 , a_3 , a_4 , a_5 , a_6 , a_7 \}$$

```
function FFT(a)

    n = length(a)
    j = 0
    for i = 1 to n-1
        bit = n>>1
        while j >= bit do
            j -= bit
            bit = bit>>1
        j = j + bit
        if i < j then
            swap a[i] and a[j]

    len = 2
    while len <= n do
        wn = e^(2*PI/len)
        i = 0
        while i < n do
            w = 1
            for j = 0 to j<len/2
                u = a[i+j]
                v = a[i+j+len/2]
                a[i+j] = u + v
                a[i+j+len/2] = u - v
                w = w*wlen
```

Thus Optimize algorithm of FFT

IV. Numeric Theoretic Transform

NTT has one benefit over FFT that it has no precision error as everything is calculated in terms of integers and not in floating or in complex number as in FFT. So it's computation is also fast. It's major drawback is that it is generally used for modulos which is of the form $p = 2^k c + 1$ primes and for random modes we need to use Chinese Remainder Theorem.

We need to find a generator polynomial r whose exponentiation from 1 to p-1 the value r^x goes from 1 to p-1 and $r^{p-1} \equiv 1 \ mod \ (p)$ by fermat little theorem if $gcd\ (a, b) = 1$

$$(r^c)^0 \equiv 1 \ mod \ (p) \ , (r^c)^1 \ mod \ (p) \ , (r^c)^2 mod \ (p) \ ,$$
$$(r^c)^3 mod \ (p) \ , \ \ldots\ldots\ldots \ , (r^c)^{2^k - 1} \ mod \ (p)$$

All having different values

Comparison with FFT w_n^a is same as $(r^{2^k})^c = r^{p-1} \equiv 1 \ mod(p)$

Where r is also known as primitive. Methods to find primitive roots -

Find all the factors of p-1 i.e. factors of $2kc$ where k and c are constants. Let all the factors be denoted by f_i. Now for checking for primitive roots just check one thinks that

$$if \ a^{(p-1)/f} \ ! = 1 \ mod \ (p) \ \forall \ f \ a \ factor \ of \ p - 1$$

then a is a primitive root or generator root.

For example :-

$$For \ p \ = \ 5 \ thus \ set \ \{1, 2, 3, 4\} \ the \ group \ is \ a \ cyclic \ group$$

$$Z/5^+ \ = \{1, 2, 3, 4\} \ and \ g \ = 3$$

- $3^1 \ = 3 \ mod \ (5)$
- $3^2 \ = 1 \ mod \ (5)$
- $3^3 \ = 4 \ mod \ (5)$
- $3^4 \ = 2 \ mod \ (5)$

So it covers all element from 1 to 4 for all exponentiation from 1 to 4 for 3. Thus 3 is a primitive root. We can also find it by using method given above.

To prove these stuffs there are two lemmas :-

- $(r^c)^x \ \neq 1 \ mod \ (p) \ when \ 1 \leq x \leq 2^k$ because r is defined as the (p-1)th root (of unity as comparing with fft) and any power of r less than p-1 will not be equal to $1 \ mod \ (p)$ as $p - 1 = 2^k c$.

- $(r^c)^{\alpha} = (r^c)^{\beta}$ where $\alpha \neq \beta$ and both are in range $[0, 2^k]$ is not true as this thing can be proved using lemma 1 and contraction.

$$\text{Let us assume } r^{c\alpha} = r^{c\beta} \Rightarrow r^{c\alpha} r^{c} = r^{c\beta} r^{c} \Rightarrow r^{c(\alpha+1)} = r^{c(\beta+1)}$$
$$\text{which can be generalized with } r^{c(\alpha+k)} = r^{c(\beta+k)}.$$
$$\text{Let } t = r^{k-\alpha} \Rightarrow \text{Let } r = r^{c(\alpha+t)} = \text{Let } r = r^{c2^k}$$
$$\text{when } \beta + \alpha \neq 2^k \text{ as } \alpha \neq \beta$$

One of the main difference between FFT and NTT is that the nth root of unity is changed from w to r^c.

Root of w of order n can be calculated using $w = r^k mod\ (p)$ when $p = k * n + 1$. Note that now $w^{n/2} = (-1)\ mod\ (p)$. So all these stuffs are completely similar to FFT. So, for making NTT just take the FFT and change to modulo p operation and w to r. So with NTT transformations of any arbitary length can be done.

NTT code in C++ with root as 1000 and prime p as $2^{18} + 1$

```
MOD = 786433
root = 1000
root_pw = 1<<18

function NTT(a)

    n = length(a)
    j = 0
    for i = 1 to n-1
        bit = n>>1
        while j >= bit do
            j -= bit
            bit = bit>>1
        j = j + bit
        if i < j then
            swap a[i] and a[j]

    len = 2
    while len <= n do
        wn = (wn*wn)%MOD
        i = 0
        while i < n do
            w = 1
            for j = 0 to j<len/2
                u = a[i+j]
                v = (a[i+j+len/2]*wn)%MOD
                a[i+j] = (u + v)%MOD
                a[i+j+len/2] = (u - v + MOD)%MOD
                w = (w*wlen)%MOD
```

V. More Features That Can Be Used In NTT To For Primes

Instead of dividing polynomial into two subparts we can divide the polynomial into any arbitary parts and modulo can be written as any prime number of the form $B^{kc} + 1$.

For example when B = 3

$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 + \ldots\ldots\ldots + a_{n-1}x^{n-1}$$
$$A(x) = A_{0\ mod\ 3}(x^3) + x * A_{1\ mod\ 3}(x^3) + x^2 * A_{2\ mod\ 3}(x^3)$$

This method will not work only for when square exponentiation is taken into account it will also work when other power exponentiation are taken into consideration.

General Time Complexity :-

$$T(N) = B * T(N/B) + O(2 * B * N) + 2 * B * log_b N$$
$$= O(BN log_b N)$$

Now after this section paper will tell about uses of NTT and its application in problem solving. Paper will also tell about how they are powerful than other algorithm. We will

also the comparison of the time complexities of general approach and NTT.

Three problems to discuss using NTT-

A. Polynomial evaluation at n points modulo a number

General approach -

Using horner's method which takes O(m) time to evaluate a polynomial of degree m. So for polynomial evaluation at n points modulo m it will take O(nm).

```
function horner(poly,n,x)
    result = poly[0]%MOD
    for i = 1 to n
        result = ((result*x)%MOD +
        poly[i])%MOD
    return result

function main()
    read(n)
    for i = 0 to n
        read(a[i])

    reverse(a)
    read(q)
    for i = 0 to q
        read(x)
        print(horner(a,n,x))
```

Example how horner method evaluated $2x3 - 6x2 + 2x - 1$.
The polynomial can be evaluated as $((2x - 6)x + 2)x - 1$

But NTT will help us to do this better :-

First divide the polynomial in three parts such that this formula is applicable

$$A(x) = a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 + \ldots\ldots\ldots + a_{n-1}x^{n-1}$$

$$A(x) = A_{0\ mod\ 3}(x^3) + x * A_{1\ mod\ 3}(x^3) + x^2 * A_{2\ mod\ 3}(x^3)$$

Let us take a example let p = 786433 then using primitive root algorithm we find r to be 10. So when we divide it in parts i.e.in 3 parts $x \Rightarrow x^3$. So root becomes 1000. As $786433 = 1 + 3 * 2^{18}$ thus after dividing it in 3 the maximum length of polynomial becomes 2^{18}. So root_pw = 2^{18}.

```
list A0,A1,A2,Y
input(n)
d = 512*512

for i = 0 to n
    input(v)
    if i%3 == 0 then
        A0.insert(v)
    if i%3 == 1 then
        A1.insert(v)
    else
        A2.insert(v)

ans[0] = A[0]

NTT(A0)
NTT(A1)
NTT(A2)

a2 = 10
a3 = 10*10

for i = 0 to d-1
    an = A0[i] + A1[i]*power(a2,i)
     + A2[i]*power(a3,i) % MOD
     Y.insert(an)

for i = d to 2*d
    an = A0[i-d] + A1[i-d]*power(a2,i)
     + A2[i-i]*power(a3,i) % MOD
     Y.insert(an)

for i = 2*d to 3*d
    an = A0[i-2*d] + A1[i-2*d]*power(a2,i)
     + A2[i-2*d]*power(a3,i) % MOD
     Y.insert(an)
```

After diving it in 3 parts call ntt function on all three parts with w = 1000. Use the above standard function of NTT and calculate.

And now us this formula for getting A(x):

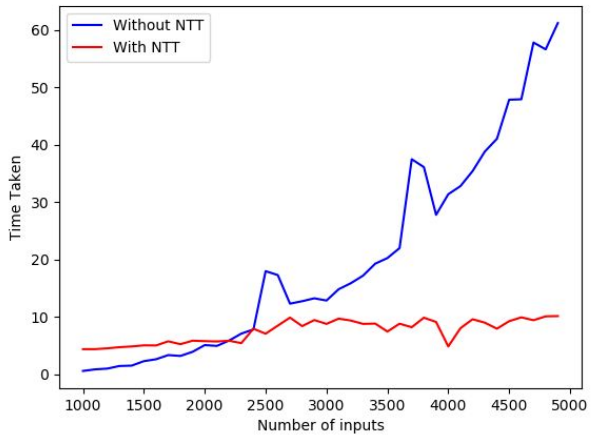
$$A(x) = A_{0\ mod\ 3}(x^3) + x * A_{1\ mod\ 3}(x^3) + x^2 * A_{2\ mod\ 3}(x^3)$$

and put it in a new vector Y. Using this thing this algorithm is able to find polynomial value at 78632 points i.e. 1 to 786432

Now to get polynomial evaluation value at a given point i use $ans[10^i] = Y[i]$.

Plot showing analysis of two algorithms :-

$$O(n^2)\ vs\ O(nlogn)$$



B. *In Mathematical operations such as finding the expected productivity of a firm if it hires K workers each having a probability of P_i to get hired and a productivity of B_i*

(Source :-https://www.codechef.com/problems/HIGHINT) [4]

For this we will get p/q as solution where p and q are co-primes then we are going to print $p * q^{-1} mod\ 786433$. For example

let the number of workers to be hired = 2

Probability of getting hired = 1/1 , 2/3, and 4/4

Productivity of workers = 1, 3, 6

Thus Expected productivity will be given by:-

Expected value = $P1 * P2 * (1 - P3) * (B1 + B2) + P2 * P3 * (1 - P1)(B2 + B3) + P3 * P1 * (1 - P2) * (B3 + B1) = 1/2 * 1/3 * (1 - 1/3) * (4 + 4) + 1/3 * 1/3 * (1 - 1/2) * (4 + 6) + 1/3 * 1/2 * (1 - 1/3) * (6 + 4) = 23/9$

Let us find the probability that B_1 is included in the expected sum, the probability is to $B_1 * P_1$ then we need probability to select $K-1$ workers from $N-1$. Which can be easily find using NTT we simply need to calculate power of $x-1$ in the polynomial

$$(1 - P_2 + P_2 x) * (1 - P_3 + P_3 x) * ..... * .(1 - P_n + P_n x)$$

This is calculated in $O(nlog^2 n)$ time . At the end we need to calculate the probability that we include B_i for all i, and sum them. This takes $O(n^2 log^2 n)$ to calculate and sum them. This thing can be further reduced to $O(nlog^2 n + MOD\ logn)$ where MOD = 786433 which is discussed in [3].

C. *Other Uses in Advance Technology*

- Matrix Multiplication - This is a most used and basic operation in linear algebra with various application in scientific application although FFT can be used but many times NTT is prefered because of no involvement of complex and floating points data types thus NTT provide a great advantage of no precision error in calculation. NTT with fermat number transform [5] can be used for multiprecision multiplication.
- Superfast Toeplitz system solvers - Toeplitz systems of equations i.e. system of form $Px = y$ where P is Toeplitz matrix which is used in many linear prediction and in adaption filtering. NTT reduces it's time complexity from $O(n^2)$ to $O(nlogn)$.
- CRC Coding and decoding - They are used for error detection and correction. In it n point code which is given or coded by Generator polynomial p. Such codes can be represented by $c(x) = s(x) * g(x)$ or as a convolution
$$c(n) = \sum_{k=0}^{N-1} g(n-k)s(k) \qquad n = 0, 1, ..., N-1$$
- Digital Signal Processing - One example where DSP (Digital Signal Processing) is frequency shifting. And by doing frequency shifting one can play music much even faster without even mixing of voices or making them to sound like squirrels.

VI. Acknowledgement

I thanks Sudashan S. (SSN college of engineering) for helping me in leaning FFT and it's faster implementation and Raghavan Sir for allowing me to choose NTT for my assignment.

VII. Conclusions

The main advantage of using Number Theoretic Transform in place of Fast Fourier Transformation is precision as Number Theoretic Transform are completely based on integers so no precision problem also as floating point calculation are slower than integer based calculation. So NTT are much faster than FFT. In comparison with general algorithms it much faster. It solves problem by changing n factor to $logn$ i.e. $O(n^2)$ *to* $O(nlogn)$. It can solve mainly all the problems related to polynomial multiplication and other related stuffs in much lesser time. It is much easier to write code for it but manipulating according to given scenario is difficult. But it application beyond its use in Polynomial

multiplication. It has various applications in Data Communication Domain like CRC, Hamming codes etc. String Manipulation, Polynomial Evaluation and many other fields like BioTechnology (DNA RNA string manipulations) , Expectation problems in many fields. The only drawback of NTT is that it is based on modulo of the form $p = 2^k c + 1$ for other prime we need to use CRT (Chinese Remainder Theorem). For non primes it is not applicable then we may need to jump to FFT or other algorithm.

References

[1] S. Gudvagen, Hogskulen i Buskerud, AUI, EMR, Kongsberd, Norway ,"Practical Application of Number Theoretic Transform", available: http://www.ux.uis.no/norsig/norsig99/Articles/gudvangen.pdf

[2] Vaibhav Gosain, "High Interview (HIGHINT)" , available: https://www.codechef.com/problems/HIGHINT

[3] Praveen Dhinwa , " POLYEVAL - EVALUATION ", available: https://discuss.codechef.com/questions/83356/polyeval-editorial

[4] Ramesh C. Aggarwal, Charles S. Burrus, "Fast Convolutional Using Fermat Number Transform with Application to Digital Filtering" , available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1162555

[5] J.J. Thomason, G.N. Larsen , J.M. Keller ,"Number Theoretic Transforms with Independent Length and Moduli" available :http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1164022&tag=1

[6] Ramsh C. Aggarwal, C.Sidney Burrus ,"Numer Theoretic Transforms to implement fast digital convolution" available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1451721

[7] Gavin Xiaoxu Yao, Ray C.C. Cheung, Cetin Kaya Koc, Kim Fung Man, "Reconfiguration Number Theoretic Transform Architechture for Cryptographic Applications" available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5681440

[8] P.C. Balla and A.Antoniou , "Number Theoretic Trasnform based on ternary arithematic" available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1172156

[9] M. Bhattacharya and J. Astola , "Number Theoretic Transform Modulo $K.2^N + 1$, A Prime , available : http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7075498

[10] E-Maxx , "Algorithms" available: https://e-maxx-eng.appspot.com/

Our project github link :- [Polyevaluation]