A study on Number Theoretic Transform(NTT) and it's improvements

Submitted by:

HARSH YADAV(180123015)

BHARGAB GAUTOM(180123008)

Submitted to:

Prof. K.V. KRISHNA

Department of Mathematics, IITG

FAST FOURIER TRANSFORM

- Let there be a polynomial of degree n−1:
- $A(x)=a_0x^0+a_1x^1+\cdots+a_{n-1}x^{n-1}$

- Without loss of generality we assume that n - the number of coefficients is a power of 2. If n is not a power of 2, then we simply add the missing terms and set the coefficients to 0.
- Let w be the nth root of unity equal to $e^{2\pi i/n}$
- Clearly, $w^n=1$ and n is the least positive power of w to result in 1.
- The **discrete Fourier transform (DFT)** of the polynomial A(x) (or equivalently the vector of coefficients (a0,a1,…,an−1) is defined as the values of the polynomial at the points $x=w^k$, i.e. it is the vector:
- $DFT(a_0,a_1,\ldots,a_{n-1})=(A(w^0),A(w^1),\ldots A(w^{n-1}))=(y^0,y^1,\ldots, y^n)$
- Similarly, the IDFT or inverse dft of a given sequence $(y^0,y^1,\ldots, y^n)$ is defined to be $(a_0,a_1,\ldots,a_{n-1})$ wherein dft of the sequence is defined in the preceeding point.

- The dft equation can be written in a matrix format. Under the assumptions in the previous slide on w, the matrix equation has a unique solution, in other words there is an one to one correspondence between the input vector and the dft. The matrix equation is also known as Vandermonde matrix

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

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

- The inverse matrix has the following format,

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

- This makes it clear that the inverse transform is much like the forward transform with w^{-1} in place of w and the entire thing further divided by n.
- The **fast Fourier transform** is a method that allows computing the DFT in O(nlogn) time. The basic idea of the FFT is to apply divide and conquer. We divide the coefficient vector of the polynomial into two vectors, recursively compute the DFT for each of them, and combine the results to compute the DFT of the complete polynomial.
- So let there be a polynomial A(x) with degree n−1, where n is a power of 2, and n>1:
- $A(x)=a_0x^0+a_1x^1+\cdots+a_{n-1}x^{n-1}$
- We divide it into two smaller polynomials, the one containing only the coefficients of the even positions, and the one containing the coefficients of the odd positions:
- $A0(x)=a_0x^0+a_2x^1+\cdots+a_{n-2}x^{n/2-1}$
- $A1(x)=a_1x^0+a_3x^1+\cdots+a_{n-1}x^{n/2-1}$

- WE OBSERVE THAT:
- $A(x)=A0(x^2)+xA1(x^2)$.
- The polynomials A0 and A1 are only half as much coefficients as the polynomial A. If we can compute the DFT(A) in linear time using DFT(A0) and DFT(A1), then we get the recurrence $T_{DFT}(n)=2T_{DFT}(n/2)+O(n)$ for the time complexity, which results in $T_{DFT}(n)=O(n\log n)$ by the **master theorem**.

- Suppose we have computed the vectors $(y^0{}_k)(k=0 \text{ to } k=n/2-1)=DFT(A0)$ and $(y^1{}_k)(k=0 \text{ to } k=n/2-1)=DFT(A1)$. Let us find a expression for $(y^k)(k=0 \text{ to } k=n-1)=DFT(A)$ from it.
- For the first n/2 values we can just use the previously noted equation $A(x)=A0(x^2)+xA1(x^2)$.
- So, $y^k=y^0{}_k+w^k y^1{}_k$, $k=0\ldots n/2-1$.
- However for the second n/2 values we need to find a slightly, different expression:

  Calculation shows that, $y^{k+n/2}=y^0{}_k+w^k y^1{}_k$

Implementation:

- Being a divide and conquer algorithm, it is naturally implemented as a recursive module, wherein the polynomials are repeatedly subdivided into the halves until it reaches a base level of zero degree polynomial, from that point the successive results are combined in a butterfly recombination as explained in the previous slides!
- This natural form of implementation leads to a lot of defunct recursive calls which complicate the execution as the value of n increases! This can be slightly improved by a non conventional technique called bit reversal wherein the implementation is changed into an iterative scheme from a recursive one. In addition, it also allows inplace computation which improves space complexity to a great extent as the recursive solution assigns new arrays at each step.

BIT REVERSAL SCHEME

- Clearly, at the first recursion of the usual implementation, the even and the odd terms are separated into the two sub-polynomials, i.e. the elements whose position number has last bit 0 get asigned to the first sub-polynomial and the ones with 1 to the second sub-polynomial. The recursion at the second stage does the same to the individual sub-polynomials but now sorts them according to the second last bit(because they already have been sorted as per the last bit). The above mechanism continues and it is clear that the recursion ultimately sorts the positions in terms of their reversed bit order before the recombination phase. The bit reversal scheme takes advantage of this fact and sorts the positions as per their reversed bit pattern before applying the recombination process. For example,
- For example the desired order for n=8 has the format {[(a0,a4),(a2,a6)],[(a1,a5),(a3,a7)]} wherein the bit reversed sorting has been applied.

RECOMBINATION

- The idea of recombination converts the recursive fft into an iterative one, basically it combines the blocks of successive sizes in successive iterations. It can be best understood with the help of an example, for example the 4 sized blocks are ready in an 8 sized fft, then they can be combined as::
- $\{[y^0_0,y^0_1,y^0_2,y^0_3],[y^1_0,y^1_1,y^1_2,y^1_3]\}$ can be combined using the butterfly to yield the overall result of:
- $\{a=\{[y^0_0+w^0y^1_0,y^0_1+wy^1_1,y^0_2+w^2y^1_2,y^0_3+w^3y^1_3],[y^0_0-w^0y^1_0,\ldots]\}\ \}$
- At the lower level, the same happens to two successive blocks of size 4 and so on. The least it can go is upto blocks of size 2. Using the above facts, we propose the following algorithm(implementation) of the bit reversal scheme in the next slide.

PSEUDO CODE FOR BIT REVERSED RECOMBINATION

```
PI=3.14152
fft(vector  a, bool invert) {
    n = a.size();
    lg_n = number of digits in n;
    for ( i = 0; i < n; i++) {
        if (i < reverse(i, lg_n))
            swap(a[i], a[reverse(i)]);}
          for (len = 2; len <= n; len = len*2) {
                ang = 2 * PI / len * (invert ? -1 : 1);
                wlen=(cos(ang)+ isin(ang));
                for (i = 0; i < n; i += len) {
                    w=1;
                    for ( j = 0; j < len / 2; j++) {
                        cd u = a[i+j], v = a[i+j+len/2] * w;
                        a[i+j] = u + v;
                        a[i+j+len/2] = u - v;
                        w *= wlen;
            }
        }
    }
}
```

NUMBER THEORETIC TRANSFORM

- Till here, we have been interested in computing dft using the complex roots of unity,but in numerous applications the arithmetic is needed to be computed modulo certain prime numbers. To tackle the situation, a variant of dft using primitive roots of unity in prime sized fields/rings was introduced.
- A n-th root of unity under a primitive field is such a number w that satisfies:$(w^n=1(modp),w^k\neq 1(modp),1\leq k<n)$ where p is a prime,
- Number theoretic transforms are nothing but dfts utilizing modular arithmetic instead of the field operations of complex numbers and all the properties of dft apply here when the operations are done modulo p!
- FERMAT NUMBERS ARE SPECIFICALLY USED AS THE PRIMES AS THEY YIELD w=2, this greatly helps in operations such as multiplication as they can be achieved by simple bit shifting operations.

Radix-r Cooley-Tukey Butterfly Algorithm

- The Cooley –Tuckey FFT is the most universal of all FFT algorithms ,because any factorization of N is possible

- Radix-r algorithms ,$N=r^s$,S is the number of stages

- The most popular algorithms are of basis r=2 or r=4,because the necessary basic DFTs or NTTs can be implemented without any multiplications

- The algorithm receives the inputs in standard ordering and produces a result in bit-reversed ordering

IMPLEMENTATION USING DIVIDE AND CONQUER

- HOW DOES A SIMPLE DFT LOOKS LIKE:
- Eq^n 1:

$$X_r = \sum_{\ell=0}^{N-1} x_\ell \omega^{r\ell}, r = 0, 1, \ldots, N-1.$$

$$(\omega_N)^{\frac{N}{2}} = -1, \quad \omega_{\frac{N}{2}} = \omega_N^2, \quad \omega_N^N = 1.$$

- THE RADIX-2 DFT DERIVED BY REWRITING THE ABOVE EQUATION :

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k}\omega_N^{r(2k)} + \omega_N^r \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1}\omega_N^{r(2k)}, \quad r = 0, 1, \ldots, N-1.$$

- Using the identity $\omega_{N/2} = \omega^2{}_N$ IN ABOVE EQUATION:

$$X_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k}\omega_N^{r(2k)} + \omega_N^r \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1}\omega_N^{r(2k)}, \quad r = 0, 1, \ldots, N-1.$$

- DEFINE $y_k=x_{2k}$ AND $z_k=x_{2k+1}$, THE ABOVE EQUATION CAN BE SPLITTED INTO TWO PARTS:

$$Y_r = \sum_{k=0}^{\frac{N}{2}-1} y_k\omega_{\frac{N}{2}}^{rk}, \quad r = 0, 1, \ldots, N/2-1,$$

$$Z_r = \sum_{k=0}^{\frac{N}{2}-1} z_k\omega_{\frac{N}{2}}^{rk}, \quad r = 0, 1, \ldots, N/2-1.$$

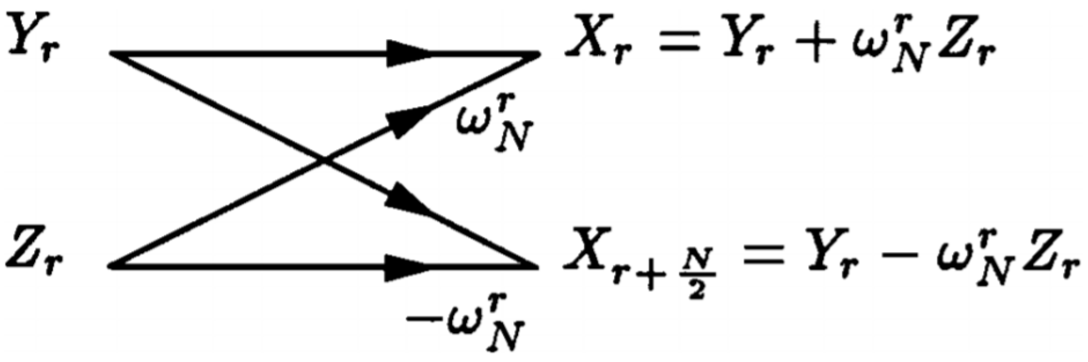
- After these two subproblems are each (recursively) solved, the solution to the original problem of size N can obtained. The first N/2 terms are given by:

$$X_r = Y_r + \omega_N^r Z_r, \quad r = 0, 1, \ldots, N/2 - 1,$$

- Using the fact that $\omega^{N/2+r}{}_N = -\omega^r{}_N$ and $\omega^{N/2}{}_{N/2} = 1$, the remaining terms are given by:

$$\begin{aligned} X_{r+\frac{N}{2}} &= \sum_{k=0}^{\frac{N}{2}-1} y_k \omega_{\frac{N}{2}}^{\left(r+\frac{N}{2}\right)k} + \omega_N^{r+\frac{N}{2}} \sum_{k=0}^{\frac{N}{2}-1} z_k \omega_{\frac{N}{2}}^{\left(r+\frac{N}{2}\right)k} \\ &= \sum_{k=0}^{\frac{N}{2}-1} y_k \omega_{\frac{N}{2}}^{rk} - \omega_N^r \sum_{k=0}^{\frac{N}{2}-1} z_k \omega_{\frac{N}{2}}^{rk} \\ &= Y_r - \omega_N^r Z_r, \quad r = 0, 1, \ldots, N/2 - 1. \end{aligned}$$

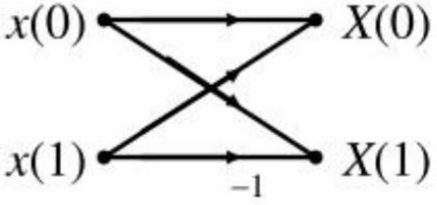
The Cooley-Tukey butterfly.


Y_r
Z_r
$X_r = Y_r + \omega_N^r Z_r$
ω_N^r
$X_{r+\frac{N}{2}} = Y_r - \omega_N^r Z_r$
$-\omega_N^r$

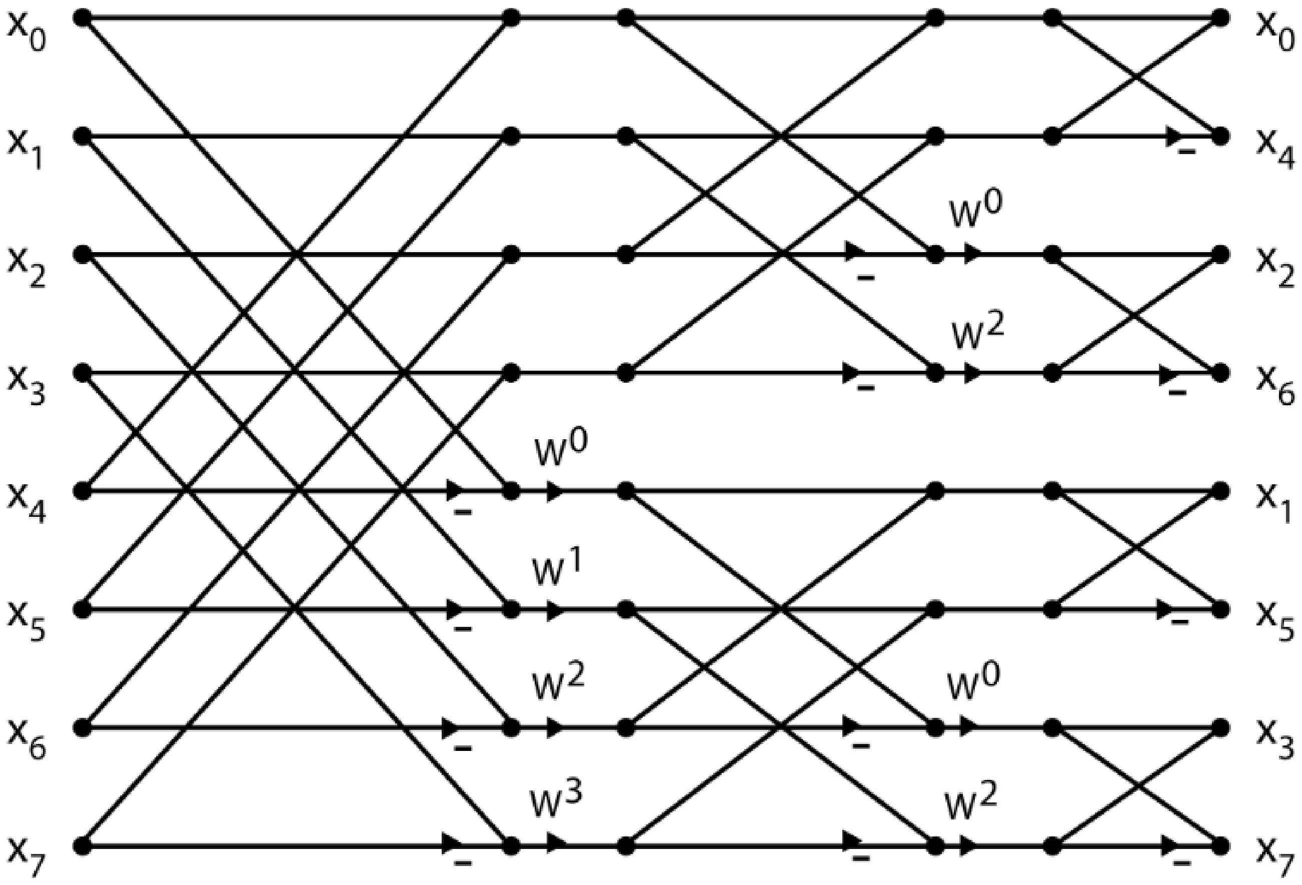
- 2-point CT butterfly:

$$X(0) = x(0)e^0 + x(1)e^0$$
$$= x(0) + x(1)$$

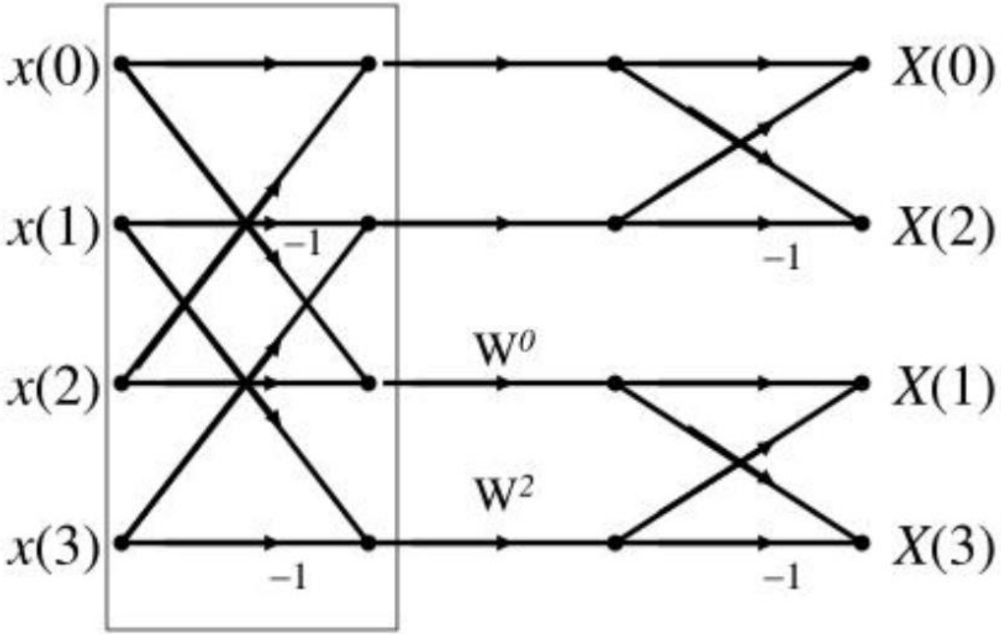
$$X(1) = x(0)e^0 + x(1)e^{-j2\pi/2}$$
$$= x(0) - x(1)$$



- 8-point CT butterfly:



- 4-point CT butterfly:



NOTE: The above three examples of 2-point,8-point and 4-point CT butterfly shows that CT algorithm based NTT/DFT takes receives the inputs in standard ordering and produces a result in bit-reversed ordering

NOTE: The Cooley-Tukey butterfly algorithm is used for calculating the forward DFT/NTT in our report.

Radix-2 Gentleman-Sande Butterfly Algorithm

- The basic form of the DIF (decimation in frequency) FFT i.e. algorithm for calculation of inverse FFT presented in the following section was found by Gentleman and Sande
- This algorithm is also based on Divide And Conquer technique. This algorithm also takes the inputs in standard ordering and produces a result in bit-reversed ordering.
- With several modifications and some pre-computation of $\Psi^{-1}(\Psi^{2}=\omega)$ in an optimal order ,the algorithm can be made to take input in bit-reversed ordering and produces an output in standard ordering.

IMPLEMENTATION USING DIVIDE AND CONQUER

- To define the two half-size subproblems, equation for a DFT (given in previous slides :Eq^{n} 1) is rewritten as:

$$X_r = \sum_{\ell=0}^{\frac{N}{2}-1} x_\ell \omega_N^{r\ell} + \sum_{\ell=\frac{N}{2}}^{N-1} x_\ell \omega_N^{r\ell} = \sum_{\ell=0}^{\frac{N}{2}-1} x_\ell \omega_N^{r\ell} + \sum_{\ell=0}^{\frac{N}{2}-1} x_{\ell+\frac{N}{2}} \omega_N^{r\left(\ell+\frac{N}{2}\right)}$$

$$= \sum_{\ell=0}^{\frac{N}{2}-1} \left(x_\ell + x_{\ell+\frac{N}{2}} \omega_N^{r\frac{N}{2}} \right) \omega_N^{r\ell}, \quad r = 0, 1, \ldots, N-1.$$

- For r even ,using information about ω in EQ^{n}1:

$$X_{2k} = \sum_{\ell=0}^{\frac{N}{2}-1} \left(x_\ell + x_{\ell+\frac{N}{2}} \omega_N^{kN} \right) \omega_N^{2k\ell}$$

$$= \sum_{\ell=0}^{\frac{N}{2}-1} \left(x_\ell + x_{\ell+\frac{N}{2}} \right) \omega_{\frac{N}{2}}^{k\ell}, \quad k = 0, 1, \ldots, N/2-1.$$

- Similarly, for r odd, using information about ω in EQ^{n}1:

$$X_{2k+1} = \sum_{\ell=0}^{\frac{N}{2}-1} \left(x_\ell + x_{\ell+\frac{N}{2}} \omega_N^{(2k+1)\frac{N}{2}} \right) \omega_N^{(2k+1)\ell}$$

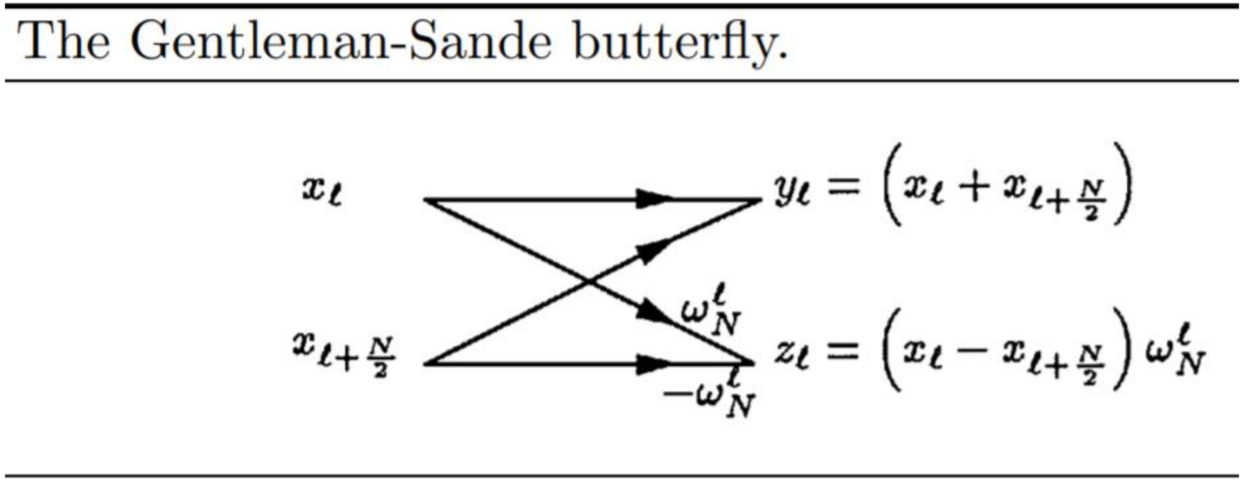
$$= \sum_{\ell=0}^{\frac{N}{2}-1} \left(\left(x_\ell - x_{\ell+\frac{N}{2}} \right) \omega_N^{\ell} \right) \omega_{\frac{N}{2}}^{k\ell}, \quad k = 0, 1, \ldots, N/2 - 1.$$

- Defining $Y_k = X_{2k}$ and $y_l = x_l + x_{l+N/2}$, $Z_k = X_{2k+1}$ and $z_l = (x_l - x_{l+N/2})$ ω^{l}_{N} yields the two sub problems:

$$Y_k = \sum_{\ell=0}^{\frac{N}{2}-1} y_\ell \omega_{\frac{N}{2}}^{k\ell}, \quad k = 0, 1, \ldots, N/2 - 1.$$

$$Z_k = \sum_{\ell=0}^{\frac{N}{2}-1} z_\ell \omega_{\frac{N}{2}}^{k\ell}, \quad k = 0, 1, \ldots, N/2 - 1.$$

- Note that because $X_{2k}= Y_k$ and $X_{2k+1} = Z_k$, no more computation is needed to obtain the solution for the original problems after the two subproblems are solved. Therefore, in the implementation of the inverse FFT, the major work is done during the subdivision step, i.e., the set-up of appropriate subproblems, and there is no combination step.
- The computation of y_l and z_l in the subdivision step as defined above is referred to as the Gentleman-Sande butterfly.



The Gentleman-Sande butterfly.

x_ℓ
$x_{\ell+\frac{N}{2}}$
$y_\ell = \left(x_\ell + x_{\ell+\frac{N}{2}}\right)$
ω_N^ℓ
$z_\ell = \left(x_\ell - x_{\ell+\frac{N}{2}}\right)\omega_N^\ell$
$-\omega_N^\ell$

- NOTE: The Gentleman-Sande butterfly algorithm is used for calculating the forward inverse DFT/NTT in our report.

- Due to these modifications we can combine Cooley-Tukey(CT) algorithm and modified Gentleman-Sande(GS) algorithm for multiplication of two large degree polynomials or two large numbers.
- The output of CT algorithm based forward NTT in bit-reversed order is passed as an input in modified GS algorithm based inverse NTT to further produce final output in standard order.
- This method reduces the steps of bit-reversal algorithm from the in place-implementation of standard NTTs giving additional speedups.
- This whole process requires precomputation of $\Psi^{-1}(\Psi^{2}=\omega)$ and other powers of Ψ^{-1} in an optimal order.
- Algorithm-1 and Algorithm-2 in report gives the implementation for the above method.

Modular Reduction and Speeding Up the NTT

- The new modular reduction algorithm allows coefficients to grow up to 32 bits in size, which eliminates the need for modular reductions after any addition during the NTT.
- As a consequence, reductions are only carried out after multiplications.
- The new modular reduction is very flexible and enables efficient implementations using either integer arithmetic or floating point arithmetic.
- The new algorithm for NTT using this modular reductions show how to merge the scaling by n^{-1} with our conversion from redundant to standard integer representation at the end of the inverse NTT.
- In addition, by pulling this conversion into the last stage of the inverse NTT, we eliminate n=2 multiplications and reductions, all at the cost of precomputing only two integers.

- The algorithm deals with special type of modulus $q=k.2^m+l$ where $k\geq 3$ and $l \geq1$
- The algorithm uses two functions namely K-RED(C) and K-RED-2x(C) which can take any integer C as input. It then returns an integer D such that $D \equiv kC \pmod q$ and $|D| < q + |C|/2^m$.
- In the context of a specific, longer computation, K-RED-2x(C) is used for reductions
- As said earlier in this presentation, In the context of the NTT ,the algorithm, use a redundant representation of integers modulo q by allowing them to grow up to 32 bits
- When necessary, the reduction function K-RED is applied to reduce the sizes of coefficients.
- The main idea is to apply the function K-RED only after multiplications, i.e., one reduction per iteration in the inner loop, letting intermediate coefficient values grow such that the final coefficient values become congruent to K .a[.] mod q for a fixed factor K.
- After computing $INTT^k(NTT^k(a) \circ NTT^k(b))$, The required array c[.] mod q is obtained with a multiplication factor of K(where $K=k^s$,s=generally no. of stages the inner loop runs). This factor can then be used at the end of the NTT-based polynomial multiplication to correct the result to the desired value.

- At the end of the computation, the final results can be converted back to the standard representation by multiplying with the inverse of the factor K.
- This conversion can be obtained for free if the computation is merged with the scaling by n^{-1} during the inverse transformation, that is, if scaling is performed by multiplying the resulting vector with the value $n^{-1}K^{-1}$
- The pre-computed values of of $\Psi^{-1}(\Psi^{2}=\omega)$ and other powers of Ψ^{-1} are scaled by a factor of k^{-1} ,this limits the growth of the power of k introduced by the reduction functions.
- On comparing this algorithm with the previous optimization which used CT butterfly for forward NTT and GS butterfly for inverse NTT ,it can be easily noted that the mod operation which was applied after every addition step (i.e. two times in inner loop) is compensated with a single mod operation in the new algorithm which the reduction functions are responsible for.
- This gives additional speedup , moreover the outer loop as in the previous optimization doesn't go upto 1 instead it goes to 2 which creates a speedup by reducing n/2 multiplications and modular operations.
- And the above leftover multiplications and modular operations are applied for free in last loop which was earlier used only for scaling purposes.
- This whole process is described in Algorithm-3 and Algorithm-4 in the report

CONCLUSION:

Through this project, we have demonstrated and learnt the basics of fast Fourier transforms in the implementation of dft and nft along with some of the subtler and minute changes in the implementation of the basic algorithms which bring in a lot of computational relief in terms of reducing the number of arithmetic modulo operations in a clever way while retaining the correct answer which have considerable impact in the cases of asymptotic inputs. In addition to these non conventional improvements we also saw certain basic yet significant improvements in implementation like bit reversal and recombination which reduce the space complexity and recursive calls of the naïve algorithm. Nevertheless fft has been an area of considerable interest for both electrical and computer scientists and most of its developments were brought about by keeping an eye on implementation perspective.