



Tutorial on FFT/NTT — The tough made simple. (Part 1)

By [sidhant](#), [history](#), 2 years ago,  

Aim — To multiply 2 n-degree polynomials in $O(n * \log n)$ instead of the trivial $O(n^2)$

I have poked around a lot of resources to understand FFT (fast fourier transform), but the math behind it would intimidate me and I would never really try to learn it. Finally last week I learned it from some pdfs and CLRS by building up an intuition of what is actually happening in the algorithm. Using this article I intend to clarify the concept to myself and bring all that I read under one article which would be simple to understand and help others struggling with fft.

Let's get started →

$$A(x) = \sum_{i=0}^{n-1} a_i * x^i, B(x) = \sum_{i=0}^{n-1} b_i * x^i, C(x) = A(x) * B(x)$$

Here $A(x)$ and $B(x)$ are polynomials of degree $n - 1$. Now we want to retrieve $C(x)$ in $O(n * \log n)$

So our methodology would be this →

1. Convert $A(x)$ and $B(x)$ from coefficient form to point value form. (FFT)
2. Now do the $O(n)$ convolution in point value form to obtain $C(x)$ in point value form, i.e. basically $C(x) = A(x) * B(x)$ in point value form.
3. Now convert $C(x)$ from point value form to coefficient form (Inverse FFT).

Q) What is point value form ?

Ans) Well, a polynomial $A(x)$ of degree n can be represented in its point value form like this →

$A(x) = \{(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_{n-1}, y_{n-1})\}$, where $y_k = A(x_k)$ and all the x_k are distinct.

So basically the first element of the pair is the value of x for which we computed the function and second value in the pair is the value which is computed i.e $A(x_k)$.

Also the point value form and coefficient form have a mapping i.e. for each point value form there is exactly one coefficient representation, if for k degree polynomial, $k + 1$ point value forms have been used at least.

Reason is simple, the point value form has n variables i.e. a_0, a_1, \dots, a_{n-1} and n equations i.e.

$y_0 = A(x_0), y_1 = A(x_1), \dots, y_{n-1} = A(x_{n-1})$ so only one solution is there.

Now using matrix multiplication the conversion from coefficient form to point value form for the polynomial $A(x) = \sum_{i=0}^{n-1} a_i * x^i$ can be

shown like this →

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \end{pmatrix} \quad (1)$$

And the inverse, that is the conversion from point value form to coefficient form for the same polynomial can be shown as this →

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} \quad (2)$$

Now, let's assume $A(x) = x^2 + x + 1 = \{(1, 3), (2, 7), (3, 13)\}$ and $B(x) = x^2 - 3 = \{(1, -2), (2, 1), (3, 6)\}$, where degree of $A(x)$ and $B(x) = 2$

Now as $C(x) = A(x) * B(x) = x^4 + x^3 - 2x^2 - 3x - 3$

$C(1) = A(1) * B(1) = 3 * -2 = -6$, $C(2) = A(2) * B(2) = 7 * 1 = 7$, $C(3) = A(3) * B(3) = 13 * 6 = 78$

So $C(x) = \{(1, -6), (2, 7), (3, 78)\}$ where degree of $C(x) = \text{degree of } A(x) + \text{degree of } B(x) = 4$

But we know that a polynomial of degree $n - 1$ requires n point value pairs, so 3 pairs of $C(x)$ are not sufficient for determining $C(x)$ uniquely as it is a polynomial of degree 4.

Therefore we need to calculate $A(x)$ and $B(x)$, for $2n$ point value pairs instead of n point value pairs so that $C(x)$'s point value form contains $2n$ pairs which would be sufficient to uniquely determine $C(x)$ which would have a degree of $2(n - 1)$.

Now if we had performed this algorithm **naively** it would have gone on like this →

Note – This is NOT the actual FFT algorithm but I would say that understanding this would layout framework to the real thing.

Note – This is actually DFT algorithm, ie. Discrete fourier transform.

1. We construct the point value form of $A(x)$ and $B(x)$ using $x_0, x_1, \dots, x_{2n-1}$ which can be made using random distinct integers. So point value form of $A(x) = \{(x_0, \alpha_0), (x_1, \alpha_1), (x_2, \alpha_2), \dots, (x_{2n-1}, \alpha_{2n-1})\}$ and $B(x) = \{(x_0, \beta_0), (x_1, \beta_1), (x_2, \beta_2), \dots, (x_{2n-1}, \beta_{2n-1})\}$ - (1) Note – The $x_0, x_1, \dots, x_{2n-1}$ should be same for $A(x)$ and $B(x)$. This conversion takes $O(n^2)$.

2. As $C(x) = A(x) * B(x)$, then what would have been the point-value form of $C(x)$?

If we plug in x_0 to all 3 equations then we see that →

$$C(x_0) = A(x_0) * B(x_0)$$

$$C(x_0) = \alpha_0 * \beta_0$$

So $C(x)$ in point value form will be $C(x) = \{(x_0, \alpha_0 * \beta_0), (x_1, \alpha_1 * \beta_1), (x_2, \alpha_2 * \beta_2), \dots, (x_{2n-1}, \alpha_{2n-1} * \beta_{2n-1})\}$

This is the convolution, and it's time complexity is $O(n)$

3. Now converting $C(x)$ back from point value form to coefficient form can be represented by using the equation 2. Here calculating the inverse of the matrix requires **LU decomposition or Lagrange's Formula**. I won't be going into depth on how to do the inverse, as this won't be required in the REAL FFT. But we get to understand that using Lagrange's Formula we would've been able to do this step in $O(n^2)$.

Note – Here the algorithm was performed wherein we used $x_0, x_1, \dots, x_{2n-1}$ as ordinary real numbers, the FFT on the other hand uses roots of unity instead and we are able to optimize the $O(n^2)$ conversions from coefficient to point value form and vice versa to $O(n * \log n)$ because of the special mathematical properties of roots of unity which allows us to use the divide and conquer approach.

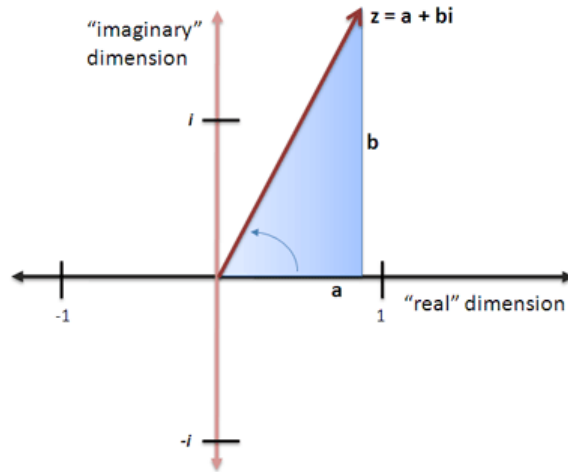
I would recommend to stop here and re-read the article till here until the algorithm is crystal clear as this is the raw concept of FFT.

A math primer on complex numbers and roots of unity would be a must now.

Q) What is a complex number ?

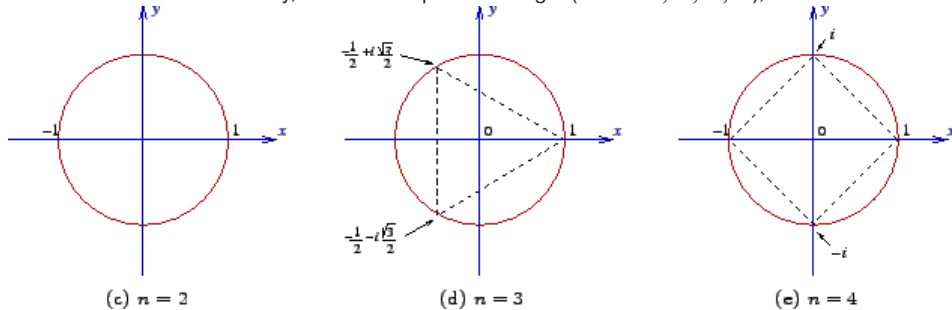
Answer — Quoting Wikipedia, “A complex number is a number that can be expressed in the form $a + bi$, where a and b are real numbers and i is the imaginary unit, that satisfies the equation $i^2 = -1$. In this expression, a is the real part and b is the imaginary part of the complex number.” The argument of a complex number is equal to the magnitude of the vector from origin $(0, 0)$ to (a, b) , therefore $\arg(z) = a^2 + b^2$ where $z = a + bi$.

$a + bi$



Q) What are the roots of unity ?

Answer — An n th root of unity, where n is a positive integer (i.e. $n = 1, 2, 3, \dots$), is a number z satisfying the equation $z^n = 1$.



In the image above, $n = 2, n = 3, n = 4$, from LEFT to RIGHT.

Intuitively, we can see that the n th root of unity lies on the circle of radius 1 unit (as its argument is equal to 1) and they are symmetrically placed i.e. they are the vertices of a n — sided regular polygon.

The n complex n th roots of unity can be represented as $e^{2\pi i k / n}$ for $k = 0, 1, \dots, n - 1$

Also $e^{i\theta} = \cos(\theta) + i * \sin(\theta)$ → Graphically see the roots of unity in a circle then this is quite intuitive.

If $n = 4$, then the 4 roots of unity would've been $e^{2\pi i * 0 / n}, e^{2\pi i * 1 / n}, e^{2\pi i * 2 / n}, e^{2\pi i * 3 / n} = (e^{2\pi i / n})^0, (e^{2\pi i / n})^1, (e^{2\pi i / n})^2, (e^{2\pi i / n})^3$ where n should be substituted by 4.

Now we notice that all the roots are actually power of $e^{2\pi i/n}$. So we can now represent the n complex n th roots of unity by $w_n^0, w_n^1, w_n^2, \dots, w_n^{n-1}$, where $w_n = e^{2\pi i/n}$

Now let us prove some lemmas before proceeding further \rightarrow

Note — Please try to prove these lemmas yourself before you look up at the solution :)

Lemma 1 — For any integer $n \geq 0, k \geq 0$ and $d \geq 0, w_{dn}^{dk} = w_n^k$

Proof — $w_{dn}^{dk} = (e^{2\pi i/dn})^{dk} = (e^{2\pi i/n})^k = w_n^k$

Lemma 2 — For any even integer $n > 0, w_n^{n/2} = w_2 = -1$

Proof — $w_n^{n/2} = w_2 * (w_n^{n/2})^{n/2} = w_d * 2^{d*1}$ where $d = n/2$

$w_d * 2^{d*1} = w_2^1$ — (Using Lemma 1)

$w_2^1 = e^{i\pi} = \cos(\pi) + i * \sin(\pi) = -1 + 0 = -1$

Lemma 3 — If $n > 0$ is even, then the squares of the n complex n th roots of unity are the $(n/2)$ complex $(n/2)$ th roots of unity, formally $(w_n^k)^2 = (w_n^{k+n/2})^2 = w_{n/2}^k$

Proof — By using lemma 1 we have $(w_n^k)^2 = w_2 * (w_n^{k/2})^{2k} = w_{n/2}^k$, for any non-negative integer k . Note that if we square all the complex n th roots of unity, then we obtain each $(n/2)$ th root of unity exactly twice since,

$(w_n^k)^2 = w_{n/2}^k \rightarrow$ (Proved above)

Also, $(w_n^{k+n/2})^2 = w_n^{2k+n} = e^{2\pi i * k'/n}$, where $k' = 2k + n$

$e^{2\pi i * k'/n} = e^{2\pi i * (2k+n)/n} = e^{2\pi i * (2k/n + 1)} = e^{(2\pi i * 2k/n) + (2\pi i)} = e^{2\pi i * 2k/n} * e^{2\pi i} = w_n^{2k} * (\cos(2\pi) + i * \sin(2\pi))$

$w_n^{2k} * (\cos(2\pi) + i * \sin(2\pi)) = w_n^{2k} * 1 = w_{n/2}^k \rightarrow$ (Proved above)

Therefore, $(w_n^k)^2 = (w_n^{k+n/2})^2 = w_{n/2}^k$

Lemma 4 — For any integer $n \geq 0, k \geq 0, w_n^{k+n/2} = -w_n^k$

Proof — $w_n^{k+n/2} = e^{2\pi i * (k+n/2)/n} = e^{2\pi i * (k/n + 1/2)} = e^{(2\pi i * k/n) + (\pi i)} = e^{2\pi i * k/n} * e^{\pi i} = w_n^k * (\cos(\pi) + i * \sin(\pi)) = w_n^k * (-1) = -w_n^k$

1. The FFT — Converting from coefficient form to point value form

Note — Let us assume that we have to multiply $2n$ — degree polynomials, when n is a power of 2. If n is not a power of 2, then make it a power of 2 by padding the polynomial's higher degree coefficients with zeroes.

Now we will see how is $A(x)$ converted from coefficient form to point value form in $O(n * \log n)$ using the special properties of n complex n th roots of unity.

$$y_k = A(x_k)$$

$$y_k = A(w_n^k) = \sum_{j=0}^{n-1} a_j * w_n^{kj}$$

Let us define \rightarrow

$$A^{\text{even}}(x) = a_0 + a_2 * x + a_4 * x^2 + \dots + a_{n-2} * x^{n/2-1}, A^{\text{odd}}(x) = a_1 + a_3 * x + a_5 * x^2 + \dots + a_{n-1} * x^{n/2-1}$$

Here, $A^{\text{even}}(x)$ contains all even-indexed coefficients of $A(x)$ and $A^{\text{odd}}(x)$ contains all odd-indexed coefficients of $A(x)$.

$$\text{It follows that } A(x) = A^{\text{even}}(x^2) + x * A^{\text{odd}}(x^2)$$

So now the problem of evaluating $A(x)$ at the n complex n th roots of unity, i.e. at $w_n^0, w_n^1, \dots, w_n^{n-1}$ reduces to \rightarrow

1. Evaluating the $n/2$ degree polynomials $A^{\text{even}}(x^2)$ and $A^{\text{odd}}(x^2)$. As $A(x)$ requires $w_n^0, w_n^1, \dots, w_n^{n-1}$ as the points on which the function is evaluated.

Therefore $A(x^2)$ would've required $(w_n^0)^2, (w_n^1)^2, \dots, (w_n^{n-1})^2$.

Extending this logic to $A^{\text{even}}(x^2)$ and $A^{\text{odd}}(x^2)$ we can say that the $A^{\text{even}}(x^2)$ and $A^{\text{odd}}(x^2)$ would require $(w_n^0)^2, (w_n^1)^2, \dots, (w_n^{n/2-1})^2 \equiv w_{n/2}^0, w_{n/2}^1, \dots, w_{n/2}^{n/2-1}$ as the points on which they should be evaluated.

Here we can clearly see that evaluating $A^{\text{even}}(x^2)$ and $A^{\text{odd}}(x^2)$ at $w_{n/2}^0, w_{n/2}^1, \dots, w_{n/2}^{n/2-1}$ is recursively solving the exact same form as that of the original problem, i.e. evaluating $A(x)$ at $w_n^0, w_n^1, \dots, w_n^{n-1}$. (The division part in the divide and conquer algorithm)

2. Combining these results using the equation $A(x) = A^{\text{even}}(x^2) + x * A^{\text{odd}}(x^2)$. (The conquer part in the divide and conquer algorithm).

Now, $A(w_n^k) = A^{\text{even}}(w_n^{2k}) + w_n^k * A^{\text{odd}}(w_n^{2k})$, if $k < n/2$, quite straightforward

And if $k \geq n/2$, then $A(w_n^k) = A^{\text{even}}(w_{n/2}^{k-n/2}) - w_n^{k-n/2} * A^{\text{odd}}(w_{n/2}^{k-n/2})$

Proof — $A(w_n^k) = A^{\text{even}}(w_n^{2k}) + w_n^k * A^{\text{odd}}(w_n^{2k}) = A^{\text{even}}(w_{n/2}^k) + w_n^k * A^{\text{odd}}(w_{n/2}^k)$ using $(w_n^k)^2 = w_{n/2}^k$

$A(w_n^k) = A^{\text{even}}(w_{n/2}^k) - w_n^{k-n/2} * A^{\text{odd}}(w_{n/2}^k)$ using $w_n^{k+n/2} = -w_n^k$ i.e. (Lemma 4), where $k' = k - n/2$.

So the pseudocode (Taken from CLRS) for FFT would be like this \rightarrow

1. RECURSIVE-FFT(a)
2. $n = a.\text{length}()$
3. If $n = 1$ then return a //Base Case
4. $w_n = e^{2\pi i/n}$
5. $w = 1$
6. $a^{\text{even}} = (a_0, a_2, \dots, a_{n-2})$
7. $a^{\text{odd}} = (a_1, a_3, \dots, a_{n-1})$
8. $y^{\text{even}} = \text{RECURSIVE-FFT}(a^{\text{even}})$

9. $y^{\text{odd}} = \text{RECURSIVE-FFT}(a^{\text{odd}})$

10. For $k = 0$ to $n/2 - 1$

11. $y_k = y_k^{\text{even}} + w * y_k^{\text{odd}}$

12. $y_{k+n/2} = y_k^{\text{even}} - w * y_k^{\text{odd}}$

13. $w * = w_n$

14. return y ;

2. The Multiplication OR Convolution

This is simply this \rightarrow

1. $a = \text{RECURSIVE-FFT}(a)$, $b = \text{RECURSIVE-FFT}(b)$ //Doing the fft.

2. For $k = 0$ to $n - 1$

3. $c(k) = a(k) * b(k)$ //Doing the convolution in $O(n)$

3. The Inverse FFT

Now we have to recover $c(x)$ from point value form to coefficient form and we are done. Well, here I am back after like 8 months, sorry for the trouble. So the whole FFT process can be show like the matrix \rightarrow

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & \dots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & \dots & w_n^{2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \dots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{pmatrix}$$

The square matrix on the left is the Vandermonde Matrix (V_n), where the (k, j) entry of V_n is w_n^{kj}

Now for finding the inverse we can write the above equation as \rightarrow

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & \dots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & \dots & w_n^{2(n-1)} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \dots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{pmatrix}$$

Now if we can find V_n^{-1} and figure out the symmetry in it like in case of FFT which enables us to solve it in $N \log N$ then we can pretty much do the inverse FFT like the FFT. Given below are Lemma 5 and Lemma 6, where in Lemma 6 shows what V_n^{-1} is by using Lemma 5 as a result.

Lemma 5 — For $n \geq 1$ and nonzero integer k not a multiple of n , $\sum_{j=0}^{n-1} (w_n^k)^j = 0$

Proof — $\sum_{j=0}^{n-1} (w_n^k)^j = ((w_n^k)^n - 1) / (w_n^k - 1) \rightarrow$ Sum of a G.P of n terms.

$$\sum_{j=0}^{n-1} (w_n^k)^j = ((w_n^k)^n - 1) / (w_n^k - 1) = ((1)^k - 1) / (w_n^k - 1) = 0$$

We required that k is not a multiple of n because $w_n^k = 1$ only when k is a multiple of n , so to ensure that the denominator is not 0 we required this constraint.

Lemma 6 — For $j, k = 0, 1, \dots, n - 1$, the (j, k) entry of V_n^{-1} is w_n^{-kj} / n

Proof — We show that $V_n^{-1} * V_n = I_n$, the $n * n$ identity matrix. Consider the (j, j') entry of $V_n^{-1} * V_n$ and let it be denoted by $[V_n^{-1} * V_n]_{jj'}$

So now \rightarrow

$$[V_n^{-1} * V_n]_{jj'} = \sum_{k=0}^{n-1} (w_n^{-kj/n}) * (w_n^{kj'}) = \sum_{k=0}^{n-1} w_n^{k(j'-j)} / n$$

Now if $j' = j$ then $w_n^{k(j'-j)} = w_n^0 = 1$ so the summation becomes 1, otherwise it is 0 in accordance with Lemma 5 given above. Note here that the constraints for Lemma 5 are satisfied here as $n \geq 1$ and $j' - j$ cannot be a multiple of n as $j' \neq j$ in this case and the maximum and minimum possible value of $j' - j$ is $(n - 1)$ and $-(n - 1)$ respectively.

So now we have it proven that the (j, k) entry of V_n^{-1} is w_n^{-kj} / n .

$$\text{Therefore, } a_k = \left(\sum_{j=0}^{n-1} y_j * w_n^{-kj} \right) / n$$

$$\text{The above equation is similar to the FFT equation } \rightarrow y_k = \sum_{j=0}^{n-1} a_j * w_n^{kj}$$

The only differences are that a and y are swapped, we have replaced w_n by w_n^{-1} and divided each element of the result by n . Therefore as rightly said by Adamant that for inverse FFT instead of the roots we use the conjugate of the roots and divide the results by n .

That is it folks. The inverse FFT might seem a bit hazy in terms of its implementation but it is just similar to the actual FFT with those slight changes and I have shown as to how we come up with those slight changes. In near future I would be writing a follow up article covering the implementation and problems related to FFT.

[Part 2 is here](#)

References used — Introduction to Algorithms (By CLRS) and Wikipedia

Feedback would be appreciated. Also please notify in the comments about any typos and formatting errors :)


 Tutorial of Crazy Skill Practice

 **fft, math, divide and conquer, polynomials, fast multiplication**

 **+394** 



 [sidhant](#)

 2 years ago

 [43](#)