

# Fast Fourier Transform

Μιχάλης Βιταντζάκης

ΑΛΜΑ - Αλγόριθμοι και Πολυπλοκότητα

October 16, 2024

Fast Fourier transform (FFT) is a divide and conquer algorithm that was invented by Gauss in the early 1800s and later reinvented by John Tukey in 1963 who published it in 1965 along with John Cooley as co-author.

The main application of FFT is in signal processing. In fact it is one of the most important algorithms in this field because it allows the fast decomposition of a signal to its frequencies.

Fast Fourier transform has also applications outside of signal processing. One example is polynomial multiplication.

# Polynomial Multiplication

Let two polynomials of degree  $d$   $A(x) = \alpha_0 + \alpha_1x + \alpha_dx^d$  and  $B(x) = b_0 + b_1x + b_dx^d$ .

Their product  $C(x) = A(x) \cdot B(x) = c_0 + c_1x + c_{2d}x^{2d}$  is a polynomial of degree  $2d$  with coefficients:

$$c_k = \alpha_0 b_k + \alpha_1 b_{k-1} + \dots + \alpha_k b_0 = \sum_{i=0}^k \alpha_i b_{k-i}$$

Using the above formula the computation of each  $c_k$  takes  $k + 1$  steps. So, the naive algorithm for polynomial multiplication takes:

$$\sum_0^{2d+1} k + 1 = 2d + 2 + \frac{(2d + 1)(2d + 2)}{2} = \Theta(d^2)$$

This means that the naive algorithm is not too slow but we can do something better.

# Polynomial Representation

The simplest way of specifying a polynomial is by its coefficients. This is called coefficient representation of a polynomial.

We will represent a polynomial in a different way, with a set of point - value pairs  $(x_i, A(x_i))$ s. This is called the value representation of a polynomial and it is more helpful for our problem.

The following theorem tells us that in order to create a value representation for a polynomial of degree  $d$  we only need a finite amount of distinct points:

## Theorem

*A degree- $d$  polynomial is uniquely characterized by its values at any  $d + 1$  distinct points.*

# Proof

Let  $A$  be a polynomial of degree at most  $n - 1$  and  $x_0, x_1, \dots, x_{n-1}$  be distinct points. We can write the  $n$  equations of the form  $A(x_i) = \alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2 + \dots + \alpha_{n-1} x_i^{n-1}$  as a matrix equation  $v = Mc$ :

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{n-1} \end{bmatrix}$$

The specialized format of  $M$  is called Vandermonde matrix and one of its properties is:

If  $x_0, x_1, \dots, x_{n-1}$  are distinct then  $M$  is invertible.

This means that given the  $n$  pairs  $(x_i, A(x_i))$  for  $i \in \{0, \dots, n - 1\}$  there is a single coefficient vector, which means a single polynomial of degree  $n - 1$  that can satisfy the equation.

# Polynomial Representation

The previous theorem tells us that, given two polynomials  $A$  and  $B$  of degree  $d$ , their product  $C$  can be identified by  $2d + 1$  points because the degree of  $C$  is  $2d$ . This means that if we evaluated  $A$  and  $B$  at  $x_0, x_1, \dots, x_{2d}$  distinct points then we can linearly evaluate  $C$  at the same  $2d + 1$  points by doing  $C(x_i) = A(x_i)B(x_i)$  for every  $i \in \{0, 1, \dots, 2d\}$ .

One problem with this idea is that we want the input polynomials  $A$ ,  $B$  and the output polynomial  $C$  to be in coefficient representation. So we need a way to convert a polynomial from the coefficient representation to the value representation and back. The conversion from the coefficient representation to the value representation is called evaluation. The inverse transformation is called interpolation.

# High Level Overview

- 1 **Selection:** Pick some points  $x_0, x_1, \dots, x_{n-1}$  where  $n \geq 2d + 1$ .
- 2 **Evaluation:** Compute  $A(x_0), A(x_1), \dots, A(x_{n-1})$  and  $B(x_0), B(x_1), \dots, B(x_{n-1})$ .
- 3 **Multiplication:** Compute  $C(x_k) = A(x_k)B(x_k)$  for  $k = 0, \dots, n - 1$
- 4 **Interpolation:** Recover  $C(x) = c_0 + c_1x + \dots + c_{2d}x^{2d}$ .

In order for the above algorithm to be useful it needs to take less time than the naive one. We already know that the multiplication step can be done in linear time. The selection step can also be done linearly because we will select points that take each  $O(1)$  time to compute.

The naive algorithm for the evaluation step works in  $\Theta(n^2)$  because it evaluates each of the  $n$  points in  $\Theta(d)$  time so we need something better for this step. We will now see how FFT evaluates a polynomial in  $O(n \log n)$  time.

# Evaluation using Divide and Conquer

In order for a divide and conquer algorithm to be effective we need to find enough overlaps in the calculations.

Lets look at a simple case. If  $A$  is an even function, because  $A(-x) = A(x)$ , we only need to evaluate  $A$  at  $\frac{n}{2}$  distinct positive points in order to know the value of  $A$  at  $n$  points. Similarly, if  $A$  is an odd function, we can use that  $A(-x) = -A(x)$  in order to halve the amount of computations.

We will generalize the idea mentioned above in more complex cases. Let  $A$  be the following polynomial:

$$15 + 2x + x^2 + 10x^3 + 14x^4 + 67x^5$$

We can split  $A$  into its even and odd powers and then factor  $x$  from the odd powers to get:

$$(15 + x^2 + 14x^4) + x(2 + 10x^2 + 67x^4)$$

Notice that the terms in the parentheses are polynomials in  $x^2$ .



# Evaluation using Divide and Conquer

More generally, we can write any polynomial  $A$  as:

$$A(x) = A_e(x^2) + xA_o(x^2)$$

Where  $A_e$  and  $A_o$  are polynomials of degree at most  $\frac{n}{2}$ .

Because  $x^2 = (-x)^2$  it is true that:

$$A(-x) = A_e(x^2) - xA_o(x^2)$$

So, if we use plus-minus paired points, we can use the values that we already have calculated for  $A(x)$  to calculate  $A(-x)$ .

We can recursively repeat this for  $A_e$  and  $A_o$  until we have a polynomial of degree zero which we can easily evaluate in  $O(1)$ .

# Time Complexity

If we have a polynomial of degree  $n$  the recursion needs  $O(n)$  time to calculate the squares of the points and splits the polynomial into two polynomials of size  $n/2$ .

Therefore, the equation that describes the time complexity of the algorithm is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Solution of  $T$ :

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n = 2\left(T\left(\frac{n}{4}\right)\right) + n + n = \\ &= 2\left(T\left(\frac{n}{8}\right)\right) + \frac{n}{2} + n + n = n + n \sum_{i=0}^{\log n} \frac{1}{2^i} < n + n \sum_{i=0}^{\infty} \frac{1}{2^i} \end{aligned}$$

Because  $\sum_{i=0}^{\infty} \frac{1}{2^i} = O(\log n)$  it is true that  $T(n) = O(n \log n)$ .

## Selecting the points

This algorithm is fast enough but there is a problem with the recursive step. In order for the recursion to work the points need to be in plus-minus pairs but the second step of the recursion will evaluate the polynomial at  $x_0^2, x_1^2, \dots, x_{\frac{n}{2}-1}^2$  which are all positive, unless we expand the domain of the polynomials to allow complex numbers as inputs. But how can we choose the  $n$  points to ensure that every recursive step will work?

We could try to find  $n$  points by hand which is easy when  $n$  is low enough but we want a generalized formula. The  $n$  points that are chosen by FFT to evaluate a polynomial are the  $n$ th roots of unity.

# Nth Roots of Unity

## Definition

The  $n$ th roots of unity are the  $n$  complex solutions to the formula  $x^n = 1$ .

The  $n$ th roots of unity are  $1, \omega, \omega^2, \dots, \omega^{n-1}$  where  $e^{\frac{2\pi i}{n}}$ .

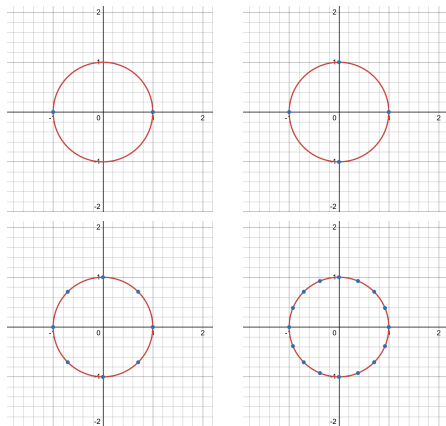
Two important properties of the  $n$ th roots of unity are:

- They are plus-minus paired. Specifically, it holds true that for every  $j \in \{0, 1, \dots, \frac{n}{2}\}$   $\omega^{n/2+j} = -\omega^j$ .
- If  $\omega^j$  is a  $n$ th root of unity then  $(\omega^j)^2$  is a  $\frac{n}{2}$ th root of unity.

Therefore, if we evaluate the polynomials using the roots of unity, then the recursive step of the algorithm described before will be well-defined.

# Nth Roots of Unity

Bellow are the 2nd, 4th, 8th and 16th roots of unity:



We can notice that the roots of unity lie on the unit circle. Also, the  $\frac{n}{2}$ th roots of unity are also  $n$ th roots of unity.

# Interpolation

Interpolation is the inverse transform of evaluation. This means that if we invert the Vandermonde matrix used by FFT we can find the coefficient vector of the polynomial.

Let  $M_n(\omega)$  be the Vandermonde matrix used by FFT which is the following:

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{j^2} & \dots & \omega^{j(n-1)} \\ & & \vdots & & \\ 1 & \omega^{n-1} & \omega^{(n-1)^2} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

We will prove that  $M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega)^*$  where  $M_n(\omega)^* = M_n(\omega^{-1})$  is the complex conjugate of  $M_n(\omega)$ .

# Proof of $M_n(\omega)^{-1} = \frac{1}{n}M_n(\omega)^*$

We will prove that  $M_n(\omega)^{-1} = \frac{1}{n}M_n(\omega)^*$  by proving that:

## Lemma

*The columns of  $M_n(\omega)$  are orthogonal to each other.*

Proof: The inner product of the  $j$ th row and the  $k$ th column of  $M_n(\omega)$  is:

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \omega^{(n-1)(j-k)}$$

If  $j = k$  then the sum is equal to  $n$ .

If  $j \neq k$  then we can rewrite the sum as a geometric sequence:

$$\sum_{m=0}^{n-1} \omega^{m(j-k)} = \frac{1 - \omega^{(n-1)(j-k)}}{1 - \omega^{j-k}} = 0$$

# Proof of $M_n(\omega)^{-1} = \frac{1}{n}M_n(\omega)^*$

This orthogonality property can be summarized in the single equation:

$$MM^* = nI$$

Where  $(M_n(\omega)M_n(\omega)^*)_{ij}$  is the inner product of the  $j$ th row and the  $k$ th column of  $M_n(\omega)$ .

From the above we get  $M_n(\omega)^{-1} = \frac{1}{n}M_n(\omega)^* = \frac{1}{n}M_n(\omega^1)$ . This means that in order to perform interpolation we just need to perform FFT using  $\omega^{-1}$  instead of  $\omega$  and then multiply the final result with  $\frac{1}{n}$ .

A fact that arises from this proof is, because  $M_n(\omega)$  is orthogonal, FFT can be viewed as a change of basis. In order to perform polynomial multiplication FFT transforms the coefficient vector to another basis, the Fourier basis, the multiplication is performed in Fourier basis in linear time and then Inverse FFT transforms the result back.





The pseudocode for FFT is:

```
function FFT( $\alpha, \omega, n$ )  
  if  $n = 1$  then return  $\alpha$   
   $s_0, s_1, \dots, s_{\frac{n}{2}-1} = \text{FFT}(\alpha_0, \alpha_2, \dots, \alpha_{n-2}, \omega^2, \frac{n}{2})$   
   $s'_0, s'_1, \dots, s'_{\frac{n}{2}-1} = \text{FFT}(\alpha_1, \alpha_3, \dots, \alpha_{n-1}, \omega^2, \frac{n}{2})$   
  for  $j = 0$  to  $\frac{n}{2} - 1$  do  
     $r_j = s_j + \omega^j s'_j$   
     $r_{j+\frac{n}{2}} = s_j - \omega^j s'_j$   
  return  $r_0, r_1, \dots, r_{n-1}$ 
```

Using the FFT function for FFT, the pseudocode for Inverse FFT is simply:

```
function IFFT( $\alpha, \omega, n$ )  
  return  $\frac{1}{n} \text{FFT}(\alpha, \omega^{-1}, n)$ 
```

-  Demaine, E.  
Fast fourier transfor.
-  S. Dasgupta, C. H. Papadimitriou, U. V. V. (2006).  
*Algorithms.*