Computability, Complexity, and Algorithms

Charles Brubaker and Lance Fortnow

Fast Fourier Transform - (Udacity)

Introduction - (Udacity, Youtube)

In this lesson, we will examine the Fast Fourier Transform and apply it in order to obtain an efficient algorithm for convolving two sequences of numbers.

If you have seen the Fourier Transform before in the context of mathematics, physics, or engineering, this lesson may have a different flavor from what you are used to. We won't be using it to solve differential equations or to characterize the behavior of an electrical circuit. Instead, we will be focused on the much more mundane tasks of multiplying polynomials and doing it quickly.

This algorithmic aspect of the Fourier transform is actually almost as old as the Fourier Transform itself, appearing in an early 19th century paper by Gauss on interpolation. The transform itself, by the way, gets it name from Jean Baptiste Fourier's 1807 paper on the propagation of heat through solids. Gauss's trick seems to have been largely forgotten until Cooley and Tukey published a paper on the Fast Fourier transform in 1965. Tukey was apparently somewhat reluctant to publish the paper, because he thought it was a simple observation and the "how-to" questions of algorithms were still considered second-class at the time. Well, much has changed since then. Their paper is now one of the most cited in scientific literature, and the idea is considered one of the most elegant in algorithm design.

Prerequisites - (Udacity, Youtube)

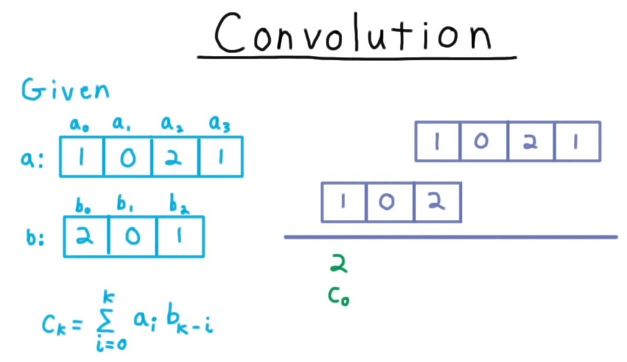
Before beginning this lesson, it might be worth brushing up on a few concepts so that you don't have to go back and review them later. We will be using complex numbers–and in particular, their polar representation. Just a familiarity with the basics is required here. We will also be using a little linear algebra, the ideas of matrix inverse and orthogonal matrices being the most important. And since we will be using a divide and conquer strategy, it might be a good idea to review the Master Theorem briefly.

Convolution - (Udacity, Youtube)

The Fast Fourier Transform is an instance of the Discrete Fourier Transform which as we've said, has its own significance in various branches of mathematics, physics, and engineering–signal processing most especially. In the study of algorithms, however, the Fast Fourier Transform is most interesting for its role in a very practical and very fast way of convolving two sequences of numbers.
We'll illustrate convolution by an example. We are given two sequences of numbers a and b as shown below, and we want to obtain a new sequence defined by the formula

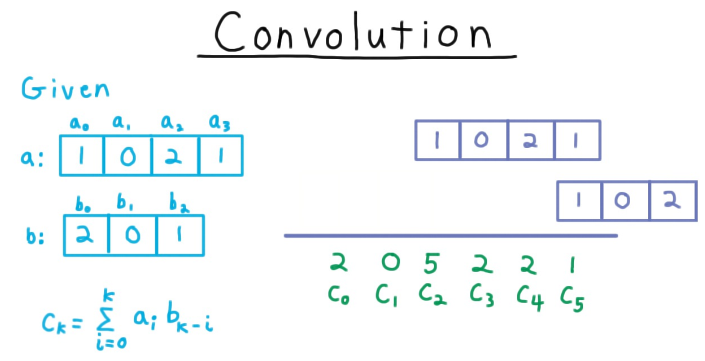
$$c_k = \sum_{i=0}^k a_i b_{k-i}$$

We can visualize convolution by reversing b and lining it up with a so that zeroth element element of b is under the kth element of a. This in the alignment for $k = 0.$

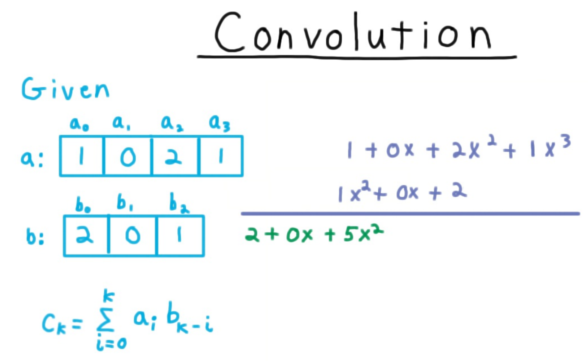


Then, we multiply all elements that overlap and add up all these products. For k=0, this is just 2\cdot 1 = 2. Therefore, c_0 = 2.

For k=1, we slide the reversed b sequence one unit to the right and perform the analogous calculation c_1 = 0 \cdot 1 + 2 \cdot 0 = 0. We continue slide b along and doing these sums until there is no more overlap left.


Convolution
Given
a:
b:
$c_k = \sum_{i=0}^{k} a_i b_{k-i}$
2 0 5 2 2 1
c_0 c_1 c_2 c_3 c_4 c_5

Convolution has many applications, but the one that will be most convenient for us to talk about is multiplying polynomials. Given the coefficients of two polynomials, we can find the coefficients of the product just by convolving the two sequences of coefficients.
In fact, we can easily repeat the example we just did but in the context of polynomial multiplication. Once the sequence b is reversed multiplying corresponding elements gives all the terms with a given power on the exponent of the variable x. For example, this alignment calculates all the x^2 terms and yields the coefficient c_2.

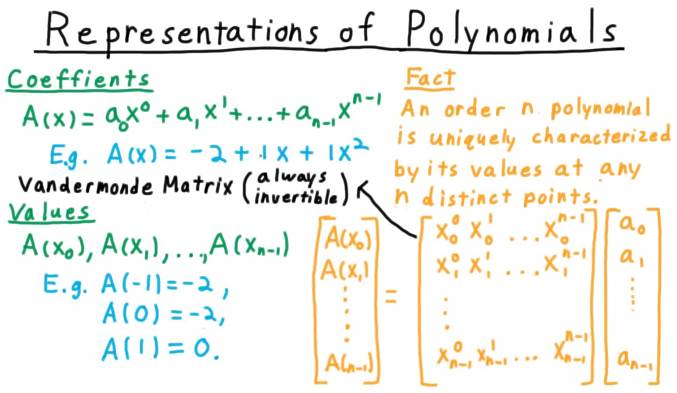


How long does this process take? Well for each element in the longer sequence, we had to do as many multiplications and additions as there are elements in the shorter sequence. Sometimes, it was a little shorter around the edges, but on average it was at least half this length. Therefore, we can say that convolving two sequences via the naive strategy outlined here takes \Theta (nm) operations, where n and m are the lengths of the two sequences. The Fast Fourier Transform will give us a way to improve upon this.

Representations of Polynomials - (Udacity, Youtube)

So far, we've assume that polynomials are represented by their coefficients. For example, A(x) = -2 + x + x^2. If you have worked with polynomial interpolation or fitting before, however, you will know that an order n polynomial is uniquely characterized by its values at any n points. (The order of a polynomial by the way, the is number of coefficients used to define it or the degree plus one.) Hence, we might just as well represent a polynomial by its values at a sequence of inputs as by its coefficients. For example, that same polynomial could be represented by saying that A(-1) = -2, and A(0) = -2 and A(1) = 0.

Going from the coefficient representation can be thought of as matrix multiplication. To calculate A at some value, I take the dot product of the corresponding row of the matrix consisting of the powers of x of the argument, with a column vector consisting of the coefficients of A.


Representations of Polynomials
Coeffients
$A(x) = a_0x^0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}$
E.g. $A(x) = -2 + 1x + 1x^2$
Vandermonde Matrix (always invertible)
Values
$A(x_0), A(x_1), \ldots, A(x_{n-1})$
E.g. $A(-1) = -2$, $A(0) = -2$, $A(1) = 0$.
Fact
An order n polynomial is uniquely characterized by its values at any n distinct points.
$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} x_0^0 & x_0^1 & \ldots & x_0^{n-1} \\ x_1^0 & x_1^1 & \ldots & x_1^{n-1} \\ \vdots \\ x_{n-1}^0 & x_{n-1}^1 & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$

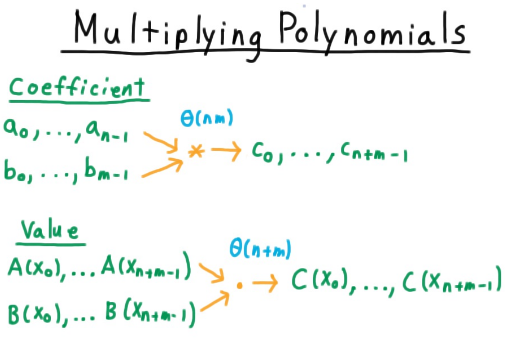
This matrix where the rows are geometric progressions of values x_i a is important enough that it gets its own name and is called a Vandermonde matrix. Its determinant is the product of the differences of all of the values for x.

\det(V) = \prod_{1 \leq i < j \leq n} (x_i - x_j)

As long as these are distinct, the matrix is invertible and we can recover the coefficients given the values!

Multiplying Polynomials - (Udacity, Youtube)

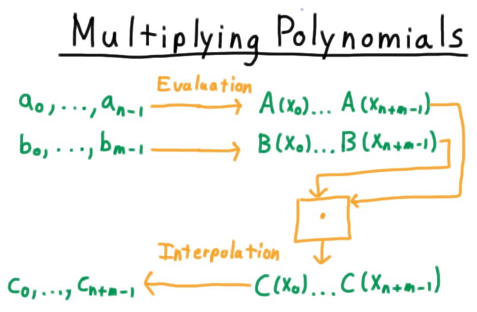
Now that we've seen an alternative way of representing polynomials, let's turn back to the problem of multiplying them. Multiplying via the convolution equation takes \Theta(nm) time as we've seen.
If the polynomials are represented as values, however, we can just multiply the corresponding values to obtain the values at the product.


Multiplying Polynomials
Coefficient
$a_0, \ldots, a_{n-1}$
$b_0, \ldots, b_{m-1}$
$\Theta(nm)$
$* \rightarrow c_0, \ldots, c_{n+m-1}$
Value
$A(x_0), \ldots A(x_{n+m-1})$
$B(x_0), \ldots B(x_{n+m-1})$
$\Theta(n+m)$
$\cdot \rightarrow C(x_0), \ldots, C(x_{n+m-1})$

Note that I did have to start with a number of points that was equal to the order of the product here.

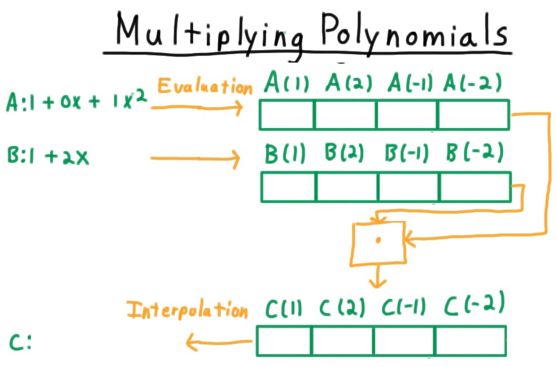
The fact that multiplying in the value representation is so much faster suggests that it might make sense to convert to the value representation, do the multiplication there, and then interpolate back to the coefficient representation.

We'll visualize the process like this. First we convert from the coefficient representation to the value representation. Then we multiply the corresponding values to get the values of C. Then we interpolate to get back the coefficients of the product via interpolation.



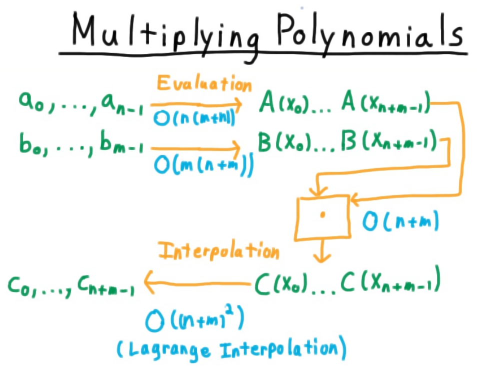
Multiplication Exercise - (Udacity)

Let’s multiply two polynomials together with this process. Start with the coefficients over here. Write their values here. Multiply them together. I’ll do the interpolation, since that computation is a bit tedious.


Multiplying Polynomials
$A: 1 + 0x + 1x^2$
Evaluation
$A(1)$ $A(2)$ $A(-1)$ $A(-2)$
$B: 1 + 2x$
$B(1)$ $B(2)$ $B(-1)$ $B(-2)$
$\cdot$
Interpolation
$C(1)$ $C(2)$ $C(-1)$ $C(-2)$
C:

Multiplying Polynomials Continued - (Udacity, Youtube)

As you might have intuited from the exercise some more cleverness will be needed to make this process efficient. Even evaluation of polynomials at arbitrary points will take quadratic time. For each point, we need to do a number of multiplications and additions proportional to the number of coefficients.



The most efficient way to do this is via Horner’s Rule. You can also think about filling out the Vandermonde matrix and then doing the matrix multiplication. Regardless, for arbitrary points we end up with a quadratic running time.

Multiplying in the value domain takes order n+m time, since we just multiply values for corresponding inputs x_j. This was fine.
Interpolation involves solving a system of equation with n+m equations and n+m unknowns. By Gaussian elimination this would take O((n+m)^3) for the worst case. There is also a method called Lagrange interpolation that allows us to do this time that is just quadratic.

Is there any hope? Well, yes there is. All of these running times pertain to an arbitrary set of points, but since we are only interested in the coefficients of C, *we get to choose the points!* As it turns out this freedom is very powerful.

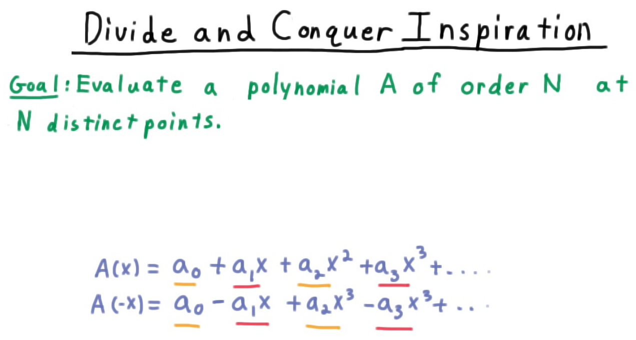
Divide and Conquer Inspiration - (Udacity, Youtube)

At this point, we've seen how polynomials can be represented by their values at a set of distinct inputs and how multiplying polynomials is easy when they are represented this way. The problem is that we are really interested in the coefficients. Recall that the coefficients of the two polynomials can represent any two sequences that we want to convolve.

To exploit the speed of multiplying in the value representation, therefore, we need an efficient way evaluate a polynomial at some distinct input points *and* an efficient way to interpolate back the result to the coefficient representation.

We'll focus on optimizing for quick evaluation first. Our goal is to evaluate a polynomial A of order N at N points. Note that I've made the order of the polynomial and the number points the same here. We can always pad the coefficients with zeros, effectively increasing the order, and we can always add more points.

As you did the calculation for the exercise, you may have taken advantage of the fact that the input values were arranged in positive-negative pairs. For higher order polynomials, this advantage becomes greater. All the even terms are the same for x and $-x$
and the odd terms are just the negatives of each other.


Divide and Conquer Inspiration
Goal: Evaluate a polynomial A of order N at N distinct points.
A(x) = a₀ + a₁x + a₂x² + a₃x³ + ...
A(-x) = a₀ - a₁x + a₂x³ - a₃x³ + . . .

Let's define A_e to be the polynomial whose coefficients are the even coefficients of A,

$$A_e(x) = a_0 + a_2 x + a_4 x^2 + \ldots$$

and define A_o to be the polynomial whose coefficients are the odd coefficients of A

$$A_o(x) = a_1 + a_3 x + a_5 x^2 + \ldots$$

Then we can write

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

and

$$A(-x) = A_e(x^2) - x A_0(x^2).$$

We get two points for the price of one!

More formally, let's say that we choose x_i such that $x_i = - x_{i+N/2}$ for $i \in \{0,\ldots, N/2-1\}$. Then we can compute the values two at a time by computing A_e and A_o at x^2 and using them in these equations.

Overall, we've changed the problem from evaluating a polynomial of order N at N points to evaluating two polynomials of half the order at half the number of points. This is good, but at best, we've only reduced the running time by a constant factor. We need to be able to apply this strategy recursively to change the asymptotic running time. A set of points that would allow us to do that would be very special indeed.

Roots of Unity - (Udacity, Youtube)

The desire to apply the trick of using positive-negative pairs recursively leads us to a very special set of points called the roots of unity.

Recall that our goal is to compute two values of a polynomial for the price of one by dividing the terms up into the even and odd powers.

From here on, we'll assume that the number of points is equal to the order of the polynomial and that this number is a power of 2. In the context of polynomial multiplication, N will the power of two that is at least as great as the number of coefficients of the product of the two polynomials, and we will pad the coefficients of the polynomials being multiplied with zeros as needed to make their number equal to N as well.

In order to be able to do this computation recursively, we need the sequence x to have the following properties:

- first, they should all be distinct ($x_j \neq x_{j'}$ unless $j=j'$). Otherwise, our efforts will be wasted and we won't have enough points to interpolate back to find the coefficients.
- second the points should be to be symmetric (or anti-symmetric, depending on how you want to look at it). That is to say, we want $x_{j+N/2} = -x_j$.
- lastly, we want all of these properties to apply to the squares of these numbers, so that we can use the trick again recursively.

If your polynomials are over some unusual field, then it may make sense to choose an unusual set of values for x. For most applications, however, the coefficients will be integers, or reals, or complex numbers and the choice of x will be the complex roots of unity.

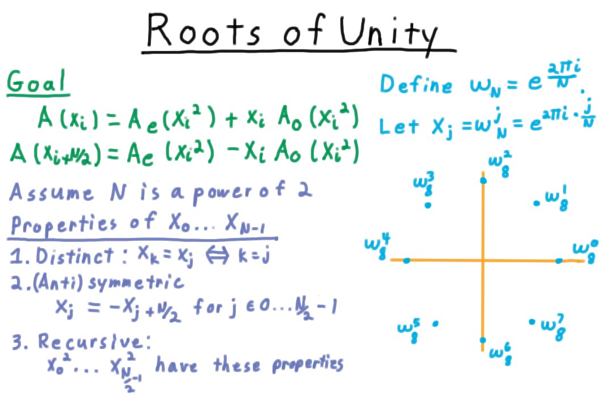
We define

$$\omega_N = e^{2\pi i/N}$$

and we let

$$x_j = \omega_N^j$$

Let's visualize these points in the complex plane for $N=8$.



Roots of Unity
Goal
$A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$
$A(x_{i+N/2}) = A_e(x_i^2) - x_i A_o(x_i^2)$
Assume N is a power of 2
Properties of $x_0 \dots x_{N-1}$
1. Distinct : $x_k = x_j \Leftrightarrow k=j$
2. (Anti) symmetric
$x_j = -x_{j+N/2}$ for $j \in 0 \dots \frac{N}{2}-1$
3. Recursive:
$x_0^2 \dots x_{\frac{N}{2}-1}^2$ have these properties
Define $\omega_N = e^{\frac{2\pi i}{N}}$.
Let $x_j = \omega_N^j = e^{2\pi i \cdot \frac{j}{N}}$
ω_8^0
ω_8^1
ω_8^2
ω_8^3
ω_8^4
ω_8^5
ω_8^6
ω_8^7

All of the points have magnitude one, so they will be arranged around the unit circle, and the angle from the positive real axis will be determined by the exponent. Thus, omega to the jth power will be j/N of the way around the unit circle.

Let's confirm that all the desired properties hold. Indeed the points are unique, as j is always less than N, so there is no wrap around.

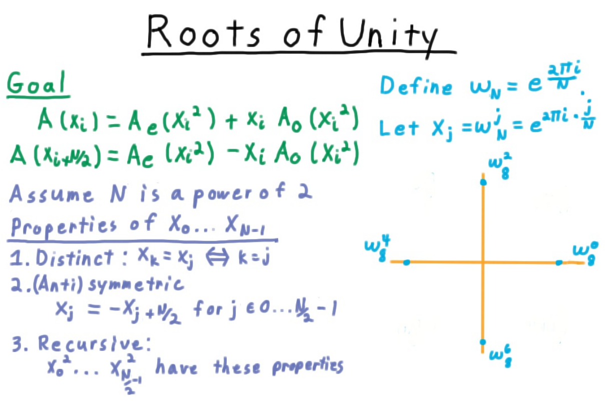
The symmetric property holds because adding $N/2$ to j corresponds to an increase in the exponent by π. This has the effect of increasing the angle by half the circle or equivalently, multiplying by negative one.

The recursive property is the hardest to confirm. Notice, however, that for all of points that have odd powers in the exponent, squaring these numbers makes the exponent even. Thus, ω^3 when squared becomes ω^6. The point ω^5

becomes \omega^{10}, which wraps around and becomes \omega^2.

Moreover, each of the even powers is the square of exactly two of the other points. Which points? Just divide the exponent by two. That gives you one. For example, for \omega^4, it's \omega^2. Where is the other point? On the opposite side of the circle, of course, \omega^6. The additional N/2 in the exponent comes an additional N when the point is squared, meaning that it maps to the same place.

The result of all of this is that when we square these numbers, the odd powers of omega go away.



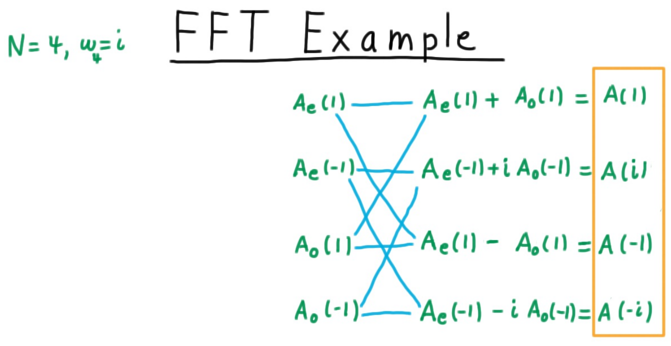
Roots of Unity
Goal
$A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$
$A(x_{i+N/2}) = A_e(x_i^2) - x_i A_o(x_i^2)$
Assume N is a power of 2
Properties of $x_0 \dots x_{N-1}$
1. Distinct : $x_k = x_j \Leftrightarrow k = j$
2. (Anti) symmetric
$x_j = -x_{j+N/2}$ for $j \in 0 \dots \frac{N}{2}-1$
3. Recursive:
$x_0^2 \dots x_{\frac{N}{2}-1}^2$ have these properties
Define $\omega_N = e^{\frac{2\pi i}{N}}$.
Let $x_j = \omega_N^j = e^{2\pi i \cdot \frac{j}{N}}$
ω_8^2
ω_8^4
ω_8^0
ω_8^6

Once we are left with only the odd powers, however, it doesn't make sense to express these points in terms of \omega_8 any more. We end up with the 4th roots of unity instead of the 8th roots. This same logic applies to any N where N is a power of two.

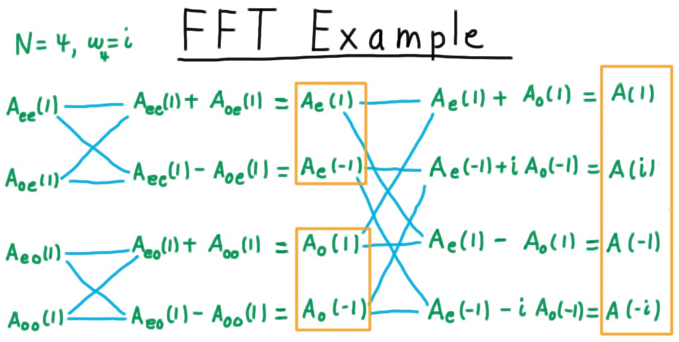
It is worth noting here how few of the properties of complex numbers were necessary for the recursion we needed. In fact, there are number theoretic algorithms that use modular arithmetic and avoid the difficulty with precision associated working with complex numbers.

FFT Example - (Udacity, Youtube)

Let's illustrate the FFT scheme by looking at the case where N = 4. That means that \omega = i, and we want to evaluate the polynomial at the points 1, i, -1, and -i.
Recall that we want to use our odd-even decomposition and recycle as much of the computation as possible. Therefore, we reduce the problem from computing A at the fourth roots of unity to computing A_e and A_o at the second roots of unity, plus some effort to combine the results.



To compute A_e and A_o at the 2nd roots of unity, we apply the same strategy again. First, rewriting them in terms of the even and odd coefficients, and recycling as much of the computation as possible.

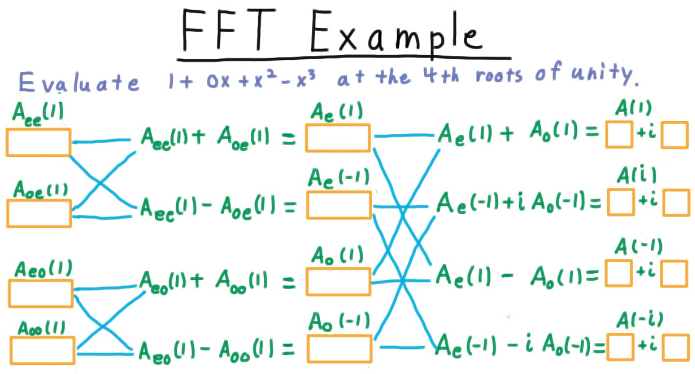


$N=4,\ \omega_4=i$
FFT Example
$A_{ee}(1)$
$A_{oe}(1)$
$A_{eo}(1)$
$A_{oo}(1)$
$A_{ee}(1)+A_{oe}(1)=A_e(1)$
$A_{ee}(1)-A_{oe}(1)=A_e(-1)$
$A_{eo}(1)+A_{oo}(1)=A_o(1)$
$A_{eo}(1)-A_{oo}(1)=A_o(-1)$
$A_e(1)+A_o(1)=A(1)$
$A_e(-1)+i\,A_o(-1)=A(i)$
$A_e(1)-A_o(1)=A(-1)$
$A_e(-1)-i\,A_o(-1)=A(-i)$

Each of the two previous problems has been reduced to evaluating an order one polynomial at one point. But this is trivial, as it only involves the constant term. The upward pass of the recursion then fills in all these intermediate values, eventually giving us the values of A at the fourth roots of unity.

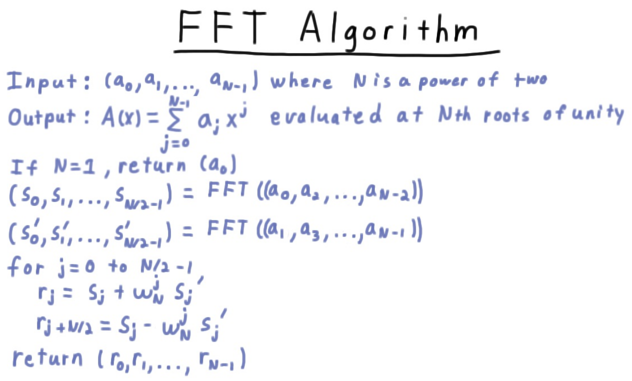
FFT Exercise - (Udacity)

To help solidify our understanding of how this process works, let's do an example. I want you to use the Fast Fourier Transform strategy to evaluate a polynomial at the 4th roots of unity.



FFT Algorithm - (Udacity, Youtube)

Having seen an example for N=4, let's now state the Fast Fourier Transform precisely for the general case. As input we have a sequence of numbers a_0,\ldots, a_{N-1} where =N is a power of two, and we want to return the values of the corresponding polynomial at the \)N\(th roots of unity.



```
FFT Algorithm
Input: $(a_0, a_1, \ldots, a_{N-1})$ where N is a power of two
Output: $A(x)=\sum_{j=0}^{N-1} a_j x^j$ evaluated at Nth roots of unity
If N=1, return $(a_0)$
$(s_0, s_1, \ldots, s_{N/2-1}) = \mathrm{FFT}((a_0, a_2, \ldots, a_{N-2}))$
$(s'_0, s'_1, \ldots, s'_{N/2-1}) = \mathrm{FFT}((a_1, a_3, \ldots, a_{N-1}))$
for $j=0$ to $N/2-1$,
    $r_j = s_j + \omega_N^j s_j'$
    $r_{j+N/2} = s_j - \omega_N^j s_j'$
return $(r_0, r_1, \ldots, r_{N-1})$
```

We'll state this as a recursive algorithm, and the base case is where \N is equal to one, in which case we just return the single element sequence. If N > 1, then we call the FFT recursively once with the even coefficients and once with the odds. Then we combine the results, taking care of paired values together. Notice the difference in sign on the contribution from the odd powers.

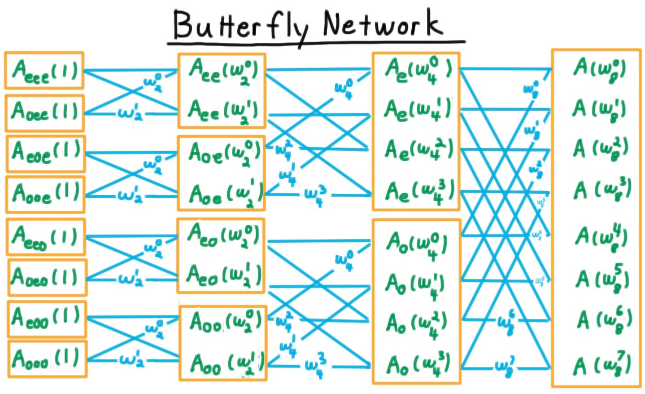
How long does this take? We traded one problem of size N for two problems of size N/2, plus theta N work for all the arithmetic in this loop. That is, T(N) = 2T(N/2) + \Theta(N).

By the master theorem, this gives us a running time of \Theta(N). This is *much* better than the \Theta(N^2) time from the naive evaluation by Horner's rule or matrix multiplication.

There is one other wrinkle that I want to add to the algorithm, and that is to say that this \omega parameter here can be any primitive Nth root of unity. The real key is only that its N powers all be roots of unity. We can add omega as a parameter to the algorithm. This will come in handy later.

Butterfly Network - (Udacity, Youtube)

Before moving on from the FFT, I want to take another look at the connections between the various subproblems.

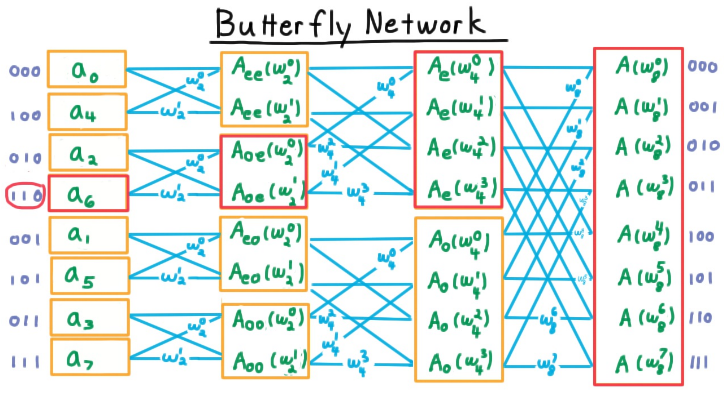


This network is called the butterfly network because these connections over here on the left look a little like a butterfly.

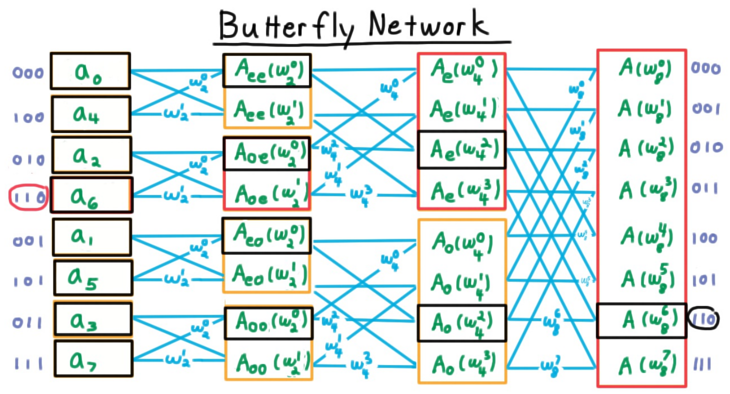
Also note that there is a unique left-to-right path between all nodes on the right and those on the left.

Another thing to note is that this sequence of even odds on these polynomials can be translated to binary. Thus, ooe becomes 110. Under this transformation, these numbers indicate which coefficient of the original polynomial gets returned. Even corresponds to grabbing the numbers with zero in the lowest order bit, odd corresponds to grabbing the numbers with 1 in the lowest order bit.

It will also be instructive to write out the power on \omega in the value here on the right-hand-side. It turns out that these numbers on the left act as instructions for any path to this node from any node on the right.



These numbers on the right act as instruction for how get here from any node on the left.


Butterfly Network
000 100 010 110 001 101 011 111
a₀ a₄ a₂ a₆ a₁ a₅ a₃ a₇
Aee(ω₂⁰) Aee(ω₂¹) Aoe(ω₂⁰) Aoe(ω₂¹) Aeo(ω₂⁰) Aeo(ω₂¹) Aoo(ω₂⁰) Aoo(ω₂¹)
Ae(ω₄⁰) Ae(ω₄¹) Ae(ω₄²) Ae(ω₄³) Ao(ω₄⁰) Ao(ω₄¹) Ao(ω₄²) Ao(ω₄³)
A(ω₈⁰) A(ω₈¹) A(ω₈²) A(ω₈³) A(ω₈⁴) A(ω₈⁵) A(ω₈⁶) A(ω₈⁷)
000 001 010 011 100 101 110 111

Recap of Progress - (Udacity, Youtube)

Recall that our original goal was not just to evaluate a polynomial at the roots of unity but rather to multiply two polynomials together, or even more generally to convolve two sequences of numbers together. The Fast Fourier transform would seem to only get us a little past half-way.

Let’s take a step back and see where we are in trying to find a faster way to multiply polynomials.

Multiplying Polynomials

$$\begin{tikzcd}
a_0,\ldots,a_{N-1} \arrow[r, "\text{Evaluation}", "O(N\log N)"'] & A(x_0)\ldots A(x_{N-1}) \arrow[dd] \\
b_0,\ldots,b_{N-1} \arrow[r, "O(N\log N)"'] & B(x_0)\ldots B(x_{N-1}) \arrow[d] \\
\text{via FFT} & \boxed{\cdot} \arrow[d, "O(N)"] \\
c_0,\ldots,c_{N-1} & C(x_0)\ldots C(x_{N-1}) \arrow[l, "\text{Interpolation}", "O(N^2)"']
\end{tikzcd}$$

We have an N \log N way to evaluate the polynomials. We can multiply them in the value representation easily in N time. But the interpolation remains a problem. Remember that this runtime involved solving a system of equations involving the Vandermonde matrix.

Vandermonde at Roots of Unity - (Udacity, Youtube)

Let’s see what the Vandermonde matrix looks like at the complex roots of unity.

Vandermonde at Roots of Unity

$$M_N(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & & \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & & \omega^{(N-1)(N-1)} \end{bmatrix}$$

Each row corresponds to the powers of a value. The powers of 1 are all 1. The powers of \omega are 1, \omega, \omega^2, \ldots . The next value is \omega^2, so its powers are 1, \omega^2, \omega^4, \ldots.

In general, element kj of the matrix is

$M_N(\omega)[k][j] = \omega^{(k-1)(j-1)}$

This matrix has some very special properties. For our purposes, however, the key one can be summarized with the following claim.

> *Let ω be a primitive Nth root of unity. Then*
> *$M_N(\omega) M_N(\omega^{-1}) = NI.$*

For the proof, consider element kj of this product. This will be the sum over of the corresponding powers of $\omega^{(k-1)}$ and $\omega^{-(j-1)}$. That is,

$$\sum_{\ell = 0}^{N-1} \omega^{(k-1)\ell} \omega^{-(j-1)\ell}$$

Gathering terms in the exponent, this becomes

$$\sum_{\ell = 0}^{N-1} \omega^{(k-j)\ell}.$$

Now, if $k = j$, then every term is 1 and the sum is N. Otherwise, we recognize this as a geometric series and rewrite it as the ratio

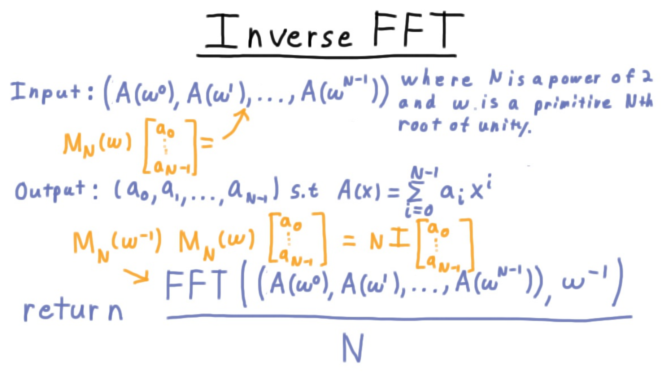
$$\frac{1 - \omega^{(k-j)N}}{1 - \omega^{(k-j)}} = 0\; \mbox{ when }\; k \neq j$$

Raising any root of unity to the Nth power is just 1, so this expression is zero when $k \neq j$ Thus, we have that entry kj is N if $k=j$, and 0 otherwise, proving the claim.

This claim is terribly important. Recall that evaluating a polynomial at the roots of unity corresponded multiplying the coefficients by the matrix $M_N(\omega)$, and we used the FFT to do that. Now, we see that we can multiply these values the inverse of this matrix also using the FFT to allow us to recover coefficients given the values! This was why it was key that the FFT work with any root of unity.

Inverse FFT - (Udacity, Youtube)

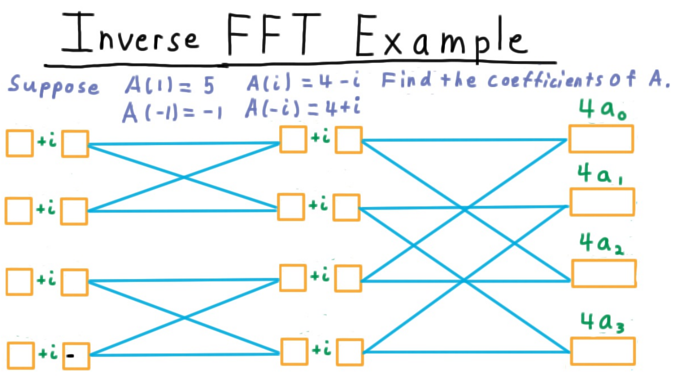
This realization about the Vandermonde matrix leads us to the inverse fast Fourier transform. We are given the values of a polynomial at the roots unity, and we want to recover the coefficients of the polynomial. The algorithm is fantastically simple given what we've established so far. Just run the regular FFT only passing in the inverse of the root of unity that we used the first time. Then divide by N.


Inverse FFT
Input: $(A(\omega^0), A(\omega^1), \dots, A(\omega^{N-1}))$
where N is a power of 2 and ω is a primitive Nth root of unity.
$M_N(\omega)\begin{bmatrix} a_0 \\ \vdots \\ a_{N-1}\end{bmatrix} =$
Output: $(a_0, a_1, \dots, a_{N-1})$ s.t $A(x) = \sum_{i=0}^{N-1} a_i x^i$
$M_N(\omega^{-1})\, M_N(\omega)\begin{bmatrix} a_0 \\ \vdots \\ a_{N-1}\end{bmatrix} = NI\begin{bmatrix} a_0 \\ \vdots \\ a_{N-1}\end{bmatrix}$
return $\frac{FFT\left((A(\omega^0), A(\omega^1), \dots, A(\omega^{N-1})), \omega^{-1}\right)}{N}$

Recall that the values that we received as input were equal to the Vandermonde matrix times the coefficients. By multiplying the vector of these values by the conjugate of Vandermonde matrix via the FFT with omega inverse, we end up with N times original coefficients. Hence, we just need to divide by N to recover the original coefficients.

Inverse FFT Example - (Udacity)

To solidify our understanding of this inverse FFT, let's do an exercise. Suppose that A is at most a cubic and the values at the fourth roots of unity are as show here. Find the coefficients of A.

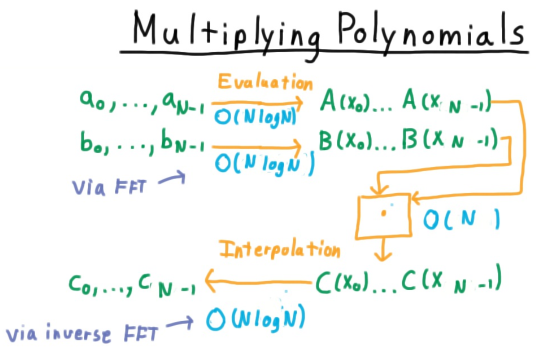


Putting It All Together - (Udacity, Youtube)

Now that we've seen how to invert the Fast Fourier Transform, we are ready to put all the pieces together. Recall that we started the lesson with this idea for multiplying polynomials: convert to the value representation, multiply the values and then convert back.
Rounding up to the nearest power of two, we might have written these running times like so in terms of our parameter N.

With the Fast Fourier Transform, we were able to do the evaluation not in quadratic time but in linearithmic time N \log N.

And even better in a sense, we were able to solve the interpolation problem using this strategy too, replacing a slower operation with once again the Fast Fourier transform and time N \log N.



The conclusion is that

> *Order N polynomials in their coefficient representation can be multiplied in O(N log n) time.*

Remember that polynomial multiplication was just a convenient way to think about the more general problem of convolution. Therefore, in general, convolving an n long sequence with an m long sequence need only take time O(n+m \log (n+m)), a remarkable and truly powerful result.

Conclusion - (Udacity, Youtube)

That concludes our discussion of the Fast Fourier Transform. From a practical perspective, probably the most important thing to remember from the lesson is that convolution can be done in O(N \log N) time, not O(N^2) as one might naively think. Don't be needlessly intimidated by the need to perform convolution in an application, and be aware that image and signal processing libraries use this technique.

From the perspective of algorithm design, the Fast Fourier Transform falls into the divide and conquer strategy family along with mergesort and Strassen's algorithm for matrix multiplication for those who are familiar with those algorithms. It has some resemblance to the dynamic programming algorithms that we studied too, however. Instead of just having one problem, we have multiple problems, as we want to evaluate a polynomial at multiple values, and their subproblems overlap in a way captured by the butterfly network. The butterfly network by the way is a fascinating structure in its own right and is sometimes used massively parallel computers.

Another thing to appreciate from the lesson is the strange twists that our development of the algorithm took. We started by thinking about the general problem of convolution, but the algorithm came much more specifically from thinking about the special case of polynomial multiplication. We started by only considering sequences of integers, yet complex numbers became an essential part of the algorithmic solution. Sometimes, the ideas you need come from unexpected places, so soak up as much mathematics as you can. You never know when it might come in handy.