0 Introduction

So far in this class, we've discussed efficient divide and conquer algorithms for multiplying numbers and matrices. The last divide and conquer algorithm we will discuss is the Fast Fourier Transform (FFT) algorithm, which we will use to help us multiply polynomials. The FFT has been described as "the most important numerical algorithm of our lifetime" due to its important practical applications in signal processing. Due to its importance (and complexity), we'll dive much deeper into the details than usual.

In this note, we will derive the most common variant of the FFT, called the Cooley-Tukey FFT[1] in the context of speeding up polynomial multiplication. We hope to make its derivation intuitive, though feel free to skip straight to the algorithm. Finally, we will introduce FFT in relation to the Discrete Fourier Transform (DFT), and discuss some additional applications.

For this note, assume arithmetic operations take constant time.

1 Polynomial Multiplication

Forward: This section isn't necessary for understanding the FFT algorithm, but it will help build context on the derivation. Also, some of the homework/exam problems in this class involve reductions to polynomial multiplications (i.e re-expressing the problem input in terms of polynomials, then multiplying them efficiently using FFT as a black-box algorithm).

First, we'll discuss two naive ways of multiplying polynomials. Given two polynomials $A(x) = a_0 + a_1 + \cdots + a_{n-1}x^{n-1} + a_n x^n$ and $B(x) = b_0 + b_1 + \cdots + b_{n-1}x^{n-1} + b_n x^n$, we would like to calculate the product polynomial, $C(x) = A(x)B(x)$. What does the product polynomial look like in general? The easiest way to see this is to try multiplying two arbitrary low-degree polynomials by hand and rearranging the terms in order of degree. You should see that for any k, $c_k x^k$ in $C(x)$ is the result of adding all the pairs of monomials whose degree adds up to k. Explicitly:

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ &\vdots \\ c_k &= \sum_{j=0}^{k} a_j b_{k-j} \end{aligned}$$

[1]The algorithm is named after Tukey, who created the algorithm during a meeting discussing detection of nuclear-weapon tests in the Soviet Union. It was later discovered that Gauss had already invented this algorithm in 1805.

1.1 Naive algorithm 1: Multiplying in coefficient form

When multiplying two polynomials, a common manual method is to use the distributive property and write out all of the terms, then combine terms of the same degree. A naive algorithm to multiply polynomials can be easily fashioned using this approach.

To calculate the runtime of this algorithm, we must first determine how many different terms there are after distributing. Since each polynomial has $n+1$ terms, the product of the two polynomials will have $(n+1)^2 = O(n^2)$ terms in it, each of which can be calculated in constant time. Afterwards, combining terms takes $O(n^2)$ time as well. Thus, this algorithm runs in $O(n^2)$.

1.2 Naive Algorithm 2: Multiplying in value form, but with inefficient conversions

Recall that a polynomial of degree n is usually expressed in its coefficient representation: $A(x) = a_0 + a_1 + \cdots + a_{n-1}x^{n-1} + a_n x^n$. However, we can uniquely represent $A(x)$ with any $n+1$ points on the polynomial (e.g a line is uniquely determined by two points). This is the value representation: $(x_1, A(x_1)), (x_2, A(x_2)), \ldots, (x_{n+1}, A(x_{n+1}))$.

Note this holds for any $n+1$ points of our choosing. Intuitively, there are $n+1$ coefficients in the coefficient form and $n+1$ points in the value representation - we have an equal number of degrees of freedom, and they should thus encode the same amount of information, though we won't provide a proof.

This is useful, since multiplying point-wise is much more efficient than multiplication in coefficient form, which we'll discuss more in detail below.

So, given two degree n polynomials in coefficient representation, our algorithm would be:

1. Evaluate $A(x)$ and $B(x)$ each at the same set of $2n+1$ x-coordinates of your choice (i.e converting them to value representation).
2. Multiply each of these points together.
3. Interpolate the $2n+1$ points to retrieve $C(x)$ (i.e converting the points back into coefficient representation).

Note that although each $A(x)$ and $B(x)$ can be uniquely represented by $n+1$ points, the product polynomial is of degree $2n$, and therefore requires $2n+1$ points to represent it uniquely.

Runtime: To calculate the runtime, let's first analyze how to evaluate a polynomial at a given arbitrary x-coordinate, x_0. Note that to calculate, say, x_0^9, we can reuse our result of x_0^8 and multiply that by x_0 instead of recalculating it from scratch. Taking advantage of this fact, we can rewrite the polynomial:

$$A(x) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-1} + x_0 a_n)))$$

and evaluate this expression starting from the innermost parentheses.[2] This involves n additions and n multiplications, and is thus $O(n)$ time. Since we do this for $2n+1$ points, evaluation takes $O(n^2)$ in total.

Next, multiplying two points at the same x-coordinate is simply multiplying their y values together, which is $O(1)$. Since we do this for $2n+1$ points, this step takes $O(n)$ in total.

Finally, interpolation can be done in $O(n^2)$ as well using linear algebraic techniques, though they aren't the focus of this class.[3]

So, in total this algorithm is still $O(n^2)$. Our bottleneck is converting between coefficient and value representation. This is where the FFT (and the inverse FFT, aka IFFT) come into play.

2 FFT Derivation

How do we speed up evaluation? For simplicity of calculations, from here on, let's say we're given a single polynomial, $A(x)$, as a degree $n-1$ polynomial, so it needs n points to uniquely define it. Remember we have the choice of whichever n points we like - perhaps we can come up with a better set of points than just picking them arbitrarily.

2.1 Observation 1: Positive-Negative Pairs

Say you wanted to evaluate $A(x) = x^2$, for example. A simple observation is that $A(x) = A(-x)$ for any x, since our polynomial is symmetric over the y axis. Generalizing this, $A(x) = A(-x)$ is true for any polynomial consisting of only even degree terms (e.g $A(x) = 2 + x^2$), as the -1s are taken to only even powers.

To utilize this observation, let's choose our n points as $n/2$ positive-negative pairs: $\{x_1, -x_1, x_2, -x_2, \ldots, x_{n/2}, -x_{n/2}\}$; then, we essentially only need to evaluate half of our original points, since we can easily retrieve $A(-x)$ from each $A(x)$.

Right now, this only holds for even polynomials. But we can actually take advantage of this observation for all polynomials: we can decompose a polynomial into its even and odd terms, then factor an x out of the odd terms to create another even polynomial:

Example: $A(x) = 2+3x+5x^2+7x^3+11x^4+13x^5 = (2+5x^2+11x^4)+x(3+7x^2+13x^4)$.

[2] This is called Horner's method.

[3] One way is to create a system of equations produced by plugging in each point in the coefficient form and solving for the coefficients. Another is Lagrange Interpolation, which is sometimes covered in CS70 and EECS16B.

Now with this decomposition, since the polynomials within the parentheses (which we'll call 'subpolynomials' from now on) are even, $A(x) = A(-x)$ for those subpolynomials. For example, try evaluating $A(x)$ at $x = 1$ and $x = -1$. The subpolynomials yield 18 and 23 respectively for both $x = 1$ and $x = -1$. This is great, as we can evaluate the two subpolynomials at $x = 1$ once, and use the results for both $A(1)$ and $A(-1)$. Here, $A(1) = 18 + 1 \cdot 23$ and $A(-1) = 18 + -1 \cdot 23$.

This isn't actually much speed-up though, as cutting our number of evaluations in half still yields $1/2 \cdot O(n^2) = O(n^2)$. However, if we can do this *recursively*, the effect compounds.

2.2 Observation 2: Subpolynomials in x^2

Notice in the example above that all the monomial terms in the parentheses are polynomial in x^2 (simply because even numbers are divisible by two). So, we can formulate these subpolynomials as functions of x^2, which allows us to use recursion since the degree (and thereby number of necessary points) are now halved in these two polynomials! Combining our two observations, we can express our polynomial $A(x)$ and also $A(-x)$ as:

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

where

$$A_e(x^2) = a_0 + a_2x + \cdots + a_{n-2}x^{n/2-1}$$
$$A_o(x^2) = a_1 + a_3x + \cdots + a_{n-1}x^{n/2-1}$$

Example: Using the same example above, $A_e(x^2) = 2 + 5x + 11x^2$ and $xA_0(x^2) = x(3 + 7x + 13x^2)$.

So, instead of evaluating $A(x)$ at n points: $\{x_1, -x_1, x_2, -x_2, \ldots x_{n/2}, -x_{n/2}\}$, we can instead evaluate $A_e(x^2)$ and $A_0(x^2)$ at $n/2$ points: $\{x_1^2, x_2^2, \ldots x_{n/2}^2\}$ and use the above formulas to retrieve $A(x)$ and $A(-x)$ for each point.

This relation suggests a recursive algorithm, as we desire to recursively evaluate $A_e(x^2)$ and $A_o(x^2)$ in the same way, i.e further breaking them into their respective even and odd subpolynomials at $n/4$ points that are positive-negative paired, and so on.

However, there is a glaring issue. We chose our original input to consist of positive-negative pairs to reduce our work by half, but after one step of recursion, our points are no longer positive-negative paired - they are all positive, as x^2 is always positive for real numbers.

2.3 Observation 3: Complex Numbers

So, we turn to the complex numbers. Reverse-engineering what the points must be, at the bottom layer, let's say we have $x = 1$. At the previous layer of recursion, we then needed $x^2 = 1$, so $x = 1$ and $x = -1$. At the layer before that, we then needed $x^2 = 1$ and $x^2 = -1$, so $x = 1, -1, i, -i$. We can see in general, at the top layer we need the set of x such that $x^k = 1$. These are known as the roots of unity.

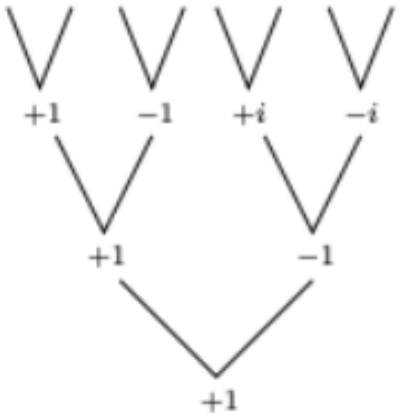

Figure 1: 4th roots of unity. Source: DPV, pg. 64

2.4 Roots of Unity

The nth roots of unity, denoted as ω_n^k for $k \in \{0, 1, \ldots, n-1\}$, are defined to be the set of x such that $x^n = 1$. Written using Euler's formula, they are $\omega_n = e^{2\pi ik/n}$ since taken to the nth power yields $e^{2\pi ik}$, which will always evaluate to 1 for any value of k, as they are multiples of 2π radians.

Notice that squaring a nth root of unity produces a $n/2$th root of unity. Algebraically, you can imagine squaring as placing a $/2$ in the denominator of the exponent, creating an $n/2$th root. Though it may be clearer to see geometrically:

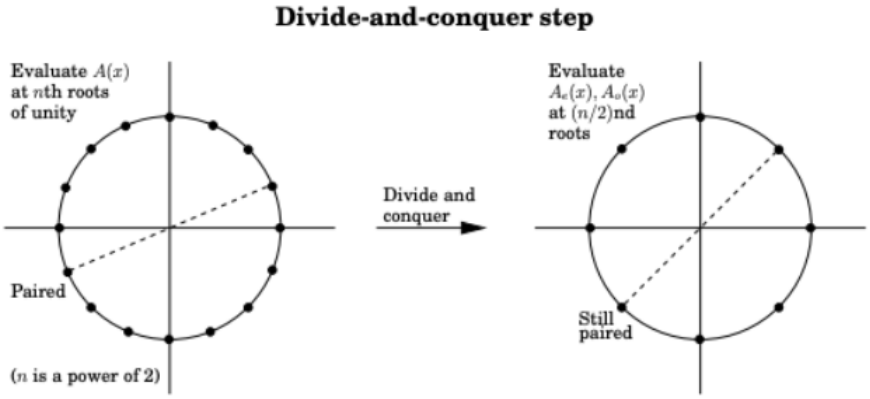

Figure 2: Source: DPV, pg. 74

So, the nth roots of unity when squared creates $n/2$th roots of unity, which are still positive-negative paired - this allows us to continue our recursion! Thus, we have our algorithm.

3 FFT Algorithm

The FFT algorithm evaluates a polynomial at the nth roots of unity. Here is the psuedocode:

Pseudocode:

Algorithm 1 FFT

```
1: procedure FFT((a_0, a_1, ..., a_{n-1}), ω)
2:     if ω = 1 then
3:         return a_0
4:     E_0, E_1, ..., E_{n/2-1} ← FFT((a_0, a_2, ..., a_{n-2}), ω^2)
5:     O_0, O_1, ..., O_{n/2-1} ← FFT((a_1, a_3, ..., a_{n-1}), ω^2)
6:     for j = 0 to n/2 − 1 do
7:         A_j ← E_j + ω^j O_j
8:         A_{j+n/2} ← E_j − ω^j O_j
9:     return (A_0, A_1, ..., A_{n-1})
```

(Note that n must be a power of 2, so we must round the number of coefficients up to the next power of two, padding with zeros if necessary.)

In this pseudocode, the input is the coefficients of a polynomial $A(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ in a list, and the A_i's that are returned represents the original polynomial $A(x)$ evaluated at ω^i i.e. the nth roots of unity. E_i and O_i represent the even and odd

polynomials evaluated at $(\omega^i)^2$, respectively. We put the positive pairs of the roots of unity in the first half of the returned list, A, and the negative pairs in the second half.

Runtime: Let $T(n)$ be the amount of time required to calculate the FFT with the nth roots of unity. At each recursive step, we create two subproblems (i.e the subpolynomials $A_e(x^2)$ and $A_o(x^2)$) each half of the original size, as they're polynomial in x^2, which means their degree (and thereby number of points) is halved. We evaluate n points in total using the two formulas (notice the for-loop over $n/2$ doing two assignments per loop) each in constant time since it's just a few arithmetic operations, yielding $O(n)$. Thus, we have the recurrence relation $T(n) = 2T(n/2) + O(n)$. By the Master Theorem, this algorithm runs in $O(n \log n)$ time, a *significant* improvement over the naive $O(n^2)$ approach!

As a sanity check, what gave us our speedup? Suppose we didn't use positive-negative pairs and just used $A(x) = A_e(x^2) + xA_o(x^2)$ recursively at n arbitrary points. In order to evaluate our original n points, we'd need to solve $A_e(x^2)$ and $A_o(x^2)$ at the full n points (in contrast with $n/2$ points) And so on - every layer requires the full n points. The runtime of that would be back to $O(n^2)$ (Exercise 2.3). Thus, our speedup came from reducing our points in half recursively.

3.1 Matrix Generalization: DFT, Inverse FFT

Why is this algorithm called the Fast Fourier Transform? Let's generalize outside the scope of polynomials. Evaluating a polynomial can be interpreted using linear algebra as a 'transformation' from the coefficient 'basis' to the value 'basis', so it can represented as a matrix-vector multiplication:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \tag{1}$$

Above, our choice of n points determines our transformation matrix, which we multiply on our vector of coefficients to give us a resulting vector of values.

The FFT chooses our n points to be the roots of unity. This transformation matrix is known as the Discrete Fourier Transform (DFT) matrix.

$$D_n(\omega) = \begin{bmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \dots & \omega^{0\cdot(n-1)} \\ \omega^{01\cdot 0} & \omega^{1\cdot 1} & \dots & \omega^{1\cdot(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(n-1)\cdot 0} & \omega^{(n-1)\cdot 1} & \dots & \omega^{(n-1)\cdot(n-1)} \end{bmatrix} \tag{2}$$

where $\omega = e^{\frac{2\pi}{n}i}$ is a nth root of unity.

This visualization is helpful as we can think of polynomial interpolation as applying the inverse DFT matrix. Using linear algebra, it turns out that the inverse of the DFT matrix is closely related the DFT matrix, namely $D_n(\omega)^{-1} = \frac{1}{n} D_n(\omega^{-1})$. Thus, the FFT algorithm can be modified easily to compute the inverse as well. All you have to do change the input of the psuedocode to take the inverse roots of unity, then divide by n at the end![4]

4 Application

4.1 Revisiting Polynomial Multiplication

We can now improve naive algorithm 2 using FFT and IFFT! Given two degree n polynomials in coefficient representation, let N equal to $2n + 1$ rounded up to the nearest power of 2. Our improved algorithm is as follows:

1. Evaluate $A(x)$ and $B(x)$ at the Nth roots of unity (i.e converting them to value representation).
2. Multiply each of these points together.
3. Interpolate these points using IFFT to retrieve $C(x)$ (i.e converting the points back into coefficient representation).

Runtime: FFT in $O(N \log N)$ + multiplication in $O(N)$ + IFFT in $O(N \log N)$ yields $O(N \log N)$. Note that $N = O(n)$, so the algorithm runs in $O(n \log n)$.

4.2 Integer Multiplication [EXTRA]

Recall that when we write an integer in base b as $(a_n \dots a_1 a_0)_b$, what we really mean is

$$a_n b^n + \cdots + a_1 b + a_0$$

This is precisely $A(x) = a_n x^n + \cdots + a_1 x + a_0$ evaluated at b! Thus, if we wanted to multiply two integers in base b, we could think of it as multiplying two polynomials, then writing down the coefficients of the resulting polynomial as the digits of our answer.

If it were this simple, why did it take researchers until 2020 to arrive at an $O(n \log n)$ algorithm for integer multiplication? Well, there's no guarantee that the coefficients are actually representable as digits. In other words, we need to carry, and that takes time.

[4]Note that in other sources the normalization factor of $\frac{1}{n}$ for the inverse DFT matrix only may be split to be $\frac{1}{\sqrt{n}}$ in front of both the DFT matrix and its inverse.

4.3 Convolution [EXTRA]

Suppose we had two discrete time signals $x[n]$ and $y[n]$, i.e. functions from integers to real values. The convolution of x and y is also a signal (i.e. function) defined by the following:

$$(x * y)[n] = \sum_{k=-\infty}^{\infty} x[k]y[n-k]$$

However, this almost the exact same formula as the product of two polynomials! In particular, if we have $C(x) = A(x)B(x)$, then the coefficients of $C(x)$ are

$$c_i = \sum_{k=0}^{\min(i,n-1)} a_k b_{i-k}$$

If x and y are only nonzero for $t = 0, 1, \ldots, n-1$, then convolution and polynomial multiplication are identical.

4.4 Cross Correlation [EXTRA]

Cross correlation can be thought of as a sliding dot product between two signals. In particular, we write

$$(x \star y)[n] = \sum_{k=-\infty}^{\infty} x[k]y[n+k]$$

where x and y are discrete time signals. The nth term computes the dot product of x and y shifted to the left by n.

Notice how this equation is similar to the equation for convolution, just with a $+k$ instead of a $-k$. In fact, the cross correlation between two signals is exactly the same as convolution between two signals, but with one of them reversed in time!

To see this, let $w[n] = x[-n]$. Then

$$\begin{aligned}
(w \star y)[n] &= \sum_{k=-\infty}^{\infty} w[k]y[n+k] \\
&= \sum_{k=-\infty}^{\infty} x[-k]y[n+k] \\
&= \sum_{i=-\infty}^{\infty} x[i]y[n-i] \\
&= (x * y)[n]
\end{aligned}$$

5 Exercises

1. Suppose you want to multiply a polynomial of degree 5 and a polynomial of degree 9 with each other. What root of unity do you need to use in the FFT algorithm?
2. Use the DFT to compute the product of $1 + 2x$ and $3 + 4x^2$.
3. Suppose we just used the $A(x) = A_e(x^2) + xA_o(x^2)$ trick recursively to evaluate a degree $n-1$ polynomial at n arbitrary points. What would the runtime of this algorithm be?
4. A *homogeneous* polynomial is defined to have terms all of the same degree, and can consist of multiple variables (e.g $x^5+2x^3y^2+9xy^4$). Find an efficient algorithm to multiply two homogeneous polynomials, each of two variables x and y.

6 Solutions

1. The end result of the product will be a degree 14 polynomial, which is determined with 15 points. Recall that the number of points we evaluate our polynomials at is always a power of two, so we round up and use the 16th roots of unity.

2. Since the end polynomial is of degree 3 and is determined by 4 points, we use the 4th roots of unity. We first evaluate the input polynomials at the 4th roots of unity:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 1+2i \\ -1 \\ 1-2i \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 3 \\ 0 \\ 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \\ 7 \\ -1 \end{bmatrix}$$

We then compute the pointwise product:

$$\begin{bmatrix} 3 \\ 1+2i \\ -1 \\ 1-2i \end{bmatrix} \circ \begin{bmatrix} 7 \\ -1 \\ 7 \\ -1 \end{bmatrix} = \begin{bmatrix} 21 \\ -1-2i \\ -7 \\ -1+2i \end{bmatrix}$$

Finally, we do polynomial interpolation with the inverse DFT:

$$\frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} 21 \\ -1-2i \\ -7 \\ -1+2i \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 4 \\ 8 \end{bmatrix}$$

Our answer is $3 + 6x + 4x^2 + 8x^3$.

3. Note that we have to be careful when using Master Theorem here, as the number of points where we are evaluating the polynomials stays static while the degree of the polynomials themselves get halved at each step. Since we're evaluating n points at each recursive step anyway, let's just compute the amount of time it takes to evaluate a single point and multiply by n.

   Let $T(n)$ be the amount of time taken to evaluate a degree n polynomial at a single arbitrary point. By splitting our polynomial into odd and even terms and evaluating those separately, we have $T(n) = 2T(n/2) + O(1)$. By Master Theorem, our runtime is $O(n)$.

Thus, the total amount of time taken to evaluate a degree n polynomial at n arbitrary points is $nT(n) = O(n^2)$, which is no better than the naive approach. We also could have reached this conclusion by observing that evaluating a polynomial at any point trivially takes $\Omega(n)$ time, so evaluating at n arbitrary points takes $\Omega(n^2)$ time.

Overall, this reaffirms what we know already: evaluating a polynomial at a single point must take linear time; the only way to save time is if we are evaluating a polynomial at many points with some sort of symmetry!

4. Let us first examine the properties of homogeneous polynomials of degree n with two variables x and y. By definition, any term in the polynomial must be some constant times $x^a y^b$, where $a + b = n$ and $a, b \geq 0$. Since there is only one degree of freedom, we can focus on the powers of x when multiplying two such polynomials, then put the powers of y back at the end.

   Thus, our algorithm consists of the following: given two polynomials of two variables x, y and of degree m and n respectively, remove all instances of y. Multiply the polynomials using FFT, then add back in powers of y to each term to ensure the degree of all terms in the result is $m + n$.

Feedback form: https://forms.gle/cM1io6eXyH65ytMP8

Contributors:

- Kevin Zhu
- Axel Li