CS 170: Efficient Algorithms and Intractable Problems

Note 3 - Fast Fourier Transform

Vishnu Iyer February 6, 2019

Polynomial arithmetic is an important problem with applications that arise in many fields. One might expect this function to be important in computer algebra systems and other mathematical tools, but similar concepts arise in areas such as signal processing as well. In this note, we will explore a divide-and-conquer technique to evaluate and interpolate polynomials.

1 Complex Numbers

You may be familiar with complex numbers from previous math classes. In this section, we review some of their main properties and set up results that will be useful to us later.

The section is included for completeness and readers with a strong background in complex numbers and polynomials need only read over the main results presented. Throughout the section, we will denote the set of real numbers as $\mathbb{R}$ and the set of complex numbers as $\mathbb{C}$.

1.1 Basics

A complex number z is a number of the form $a + bi$ where a and b are real numbers and $i^2 = -1$. The form $z = a + bi$ is known as the *rectangular form* of the complex number. This is because we can plot a complex number on what is known as the *complex plane*. The x coordinate is a and the y coordinate is b. Similar to rectangular and polar coordinates, we can also define a *polar form* $z = re^{i\theta}$ where $r, \theta \in \mathbb{R}$, $r > 0$, and $0 \leq \theta < 2\pi$. A representation of a complex number on the complex plane is shown in Figure 1 To convert between forms, we look at Euler's theorem.

Theorem 1.1 (Euler's Theorem)**.**

$$e^{i\theta} = \cos\theta + i\sin\theta$$

Proof. We consider the Taylor series for our functions.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}, \qquad \cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!!}, \qquad \cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!!}$$

The theorem follows from computing each term of the series $e^{i\theta}$ and $\cos\theta + i\sin\theta$. □

Therefore, if r is such that $r^2 = a^2 + b^2$ and θ is such that $\tan\theta = \frac{b}{a}$, we have that $a + bi = re^{i\theta}$. We call r the *magnitude* and θ the *angle*. The main reason we consider this form is to understand the process of multiplying complex numbers. The polar forms show that the product of $z_1 = r_1 e^{i\theta_1}$ and $z_2 = r_2 e^{i\theta_2}$ is $r_1 r_2 e^{i(\theta_1+\theta_2)}$. That is, *the magnitudes multiply and the angles add*. In this sense, multiplication by a complex number is equivalent to scaling by the magnitude and rotating counterclockwise by the angle.

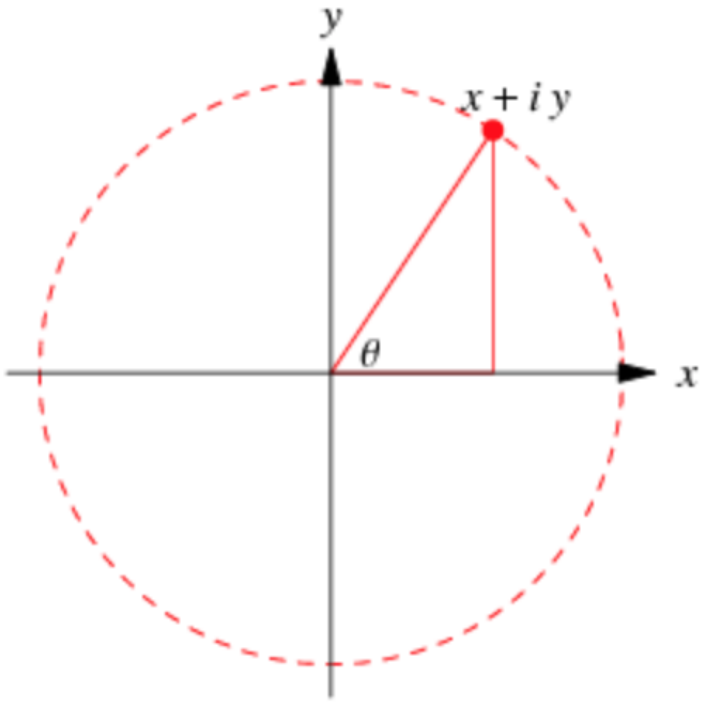

Figure 1: A complex number with rectangular form $x + iy$, magnitude $\sqrt{x^2 + y^2}$, and angle θ on the complex plane. Courtesy of Wolfram Mathworld.

1.2 Roots of Unity

Among complex numbers, the *roots of unity* are of particular interest.

Definition 1.2 (Roots of Unity)**.** The nth roots of unity are complex numbers such that $z^n = 1$.

For every number n, there are exactly n distinct roots of unity. We know this from the Fundamental Theorem of Algebra.

Theorem 1.3 (Fundamental Theorem of Algebra)**.** For any polynomial $p(x)$ of degree n with complex coefficients, the equation $p(x) = 0$ has exactly n (not necessarily distinct) complex solutions.

Now we show how the roots

Proposition 1.4. The nth roots of unity are of the form $e^{\frac{2\pi i}{n}k}$, where k is an integer.

Proof. If $z = e^{\frac{2\pi i}{n}k}$, we can see that $z^n = e^{2\pi ik} = \cos(2\pi k) + i\sin(2\pi k) = 1$. Furthermore, for a choice of $k = 0, 1, ..., n-1$, $e^{\frac{2\pi i}{n}k}$ is distinct. □

This brings us to an important point. The roots of unity are, graphically, distributed evenly around the unit circularly. Since $e^{2\pi i} = 1$, we see that if $k_1 = k_2 + n$, then $e^{\frac{2\pi i}{n}k_2} = e^{\frac{2\pi i}{n}k_1}$.

Furthermore, since $e^{\pi i} = 1$, we have that $e^{i(\theta+\pi)} = -e^{i\theta}$. In the context of roots of unity, this means that, for an even number n, if $k_1 = k_2 + \frac{n}{2}$, then $e^{\frac{2\pi i}{n}k_2} = -e^{\frac{2\pi i}{n}k_1}$ and if $z_1 = e^{\frac{2\pi i}{n}k_1}$ and $z_2 = e^{\frac{2\pi i}{n}k_2}$, then $z_1^2 = z_2^2$. This in particular will be useful to us in our divide-and-conquer approach to the Fourier Transform. We summarize our results from this section below.

Properties of Complex Numbers
Let $z_1 = e^{\frac{2\pi i}{n}k_1}$ and $z_2 = e^{\frac{2\pi i}{n}k_2}$.

1. $e^{i\theta} = \cos\theta + i\sin\theta$
2. Multiplication of a complex number z by a complex number $re^{i\theta}$ dilates the magnitude of z by a factor of r and rotates its angle counterclockwise by θ.
3. The nth roots of unity are of the form $e^{\frac{2\pi i}{n}k}$ for any integer k.
4. If $k_2 = k_1 + n$, then $z_1 = z_2$.
5. If n is even and $k_2 = k_1 + \frac{n}{2}$, then $z_1 = -z_2$ and $z_1^2 = z_2^2$.

2 The Algorithm

2.1 Setup

A polynomial of degree at most n is uniquely determined by either its $n+1$ coefficients (called the coefficient representation) or its value at *any* $n+1$ distinct points (called the value representation). This is significant because given $n+1$ ordered pairs of points, there is *exactly* one polynomial of degree n or less that passes through these points.

Example 2.1. Consider the polynomial $p(x) = x^2 - 2x + 4$. Clearly it is uniquely described by the coefficients $a_2 = 1$, $a_1 = -2$, and $a_0 = 4$. However, since the polynomial is of degree 2, we can also describe the polynomial by the points $(0,4), (1,3), (2,4)$ or the points $(0,4), (i, 3-2i), (-i, 3+2i)$. Any set of 3 distinct points that the polynomial passes through is enough to uniquely identify it amongst degree 2 and lower polynomials. That is, *it is the only one* that passes through any set of 3 points that it passes through. Note in particular that the number of coefficients is equal to the number of points required; this is not a coincidence.

Suppose we want to design an algorithm that can convert between these representations. In particular, we will first consider the task of going from the coefficient representation to the value representation, a process known as *evaluation*. Note that the naive method of evaluation takes time $\Theta(n^2)$.

Our analysis will begin with the following fact about polynomials: any polynomial $p(x)$ can be written as the sum of an even polynomial $p_1(x)$ and an odd polynomial $p_2(x)$ (recall that an even function is one such that $f(-x) = f(x)$ and and odd function is one such that $f(-x) = -f(x)$). This is because we can collect the even degree terms in $p_1(x)$ and the odd degree terms in $p_2(x)$.

Example 2.2. We can break up the polynomial $p(x) = 5x^5 - 2x^4 + 4x^3 + 7x^2 - 6x - 6$ as follows:

$$p_1(x) = -2x^4 + 7x^2 - 6$$

$$p_2(x) = 5x^5 + 4x^3 - 6x$$

Notice that $p_1(x)$ can be written as some polynomial $p_e(x^2)$ and $p_2(x)$ can be written as some polynomial $xp_o(x^2)$.

Example 2.3. From our previous example, we can write

$$p_e(x) = -2x^2 + 7x - 6$$

$$p_o(x) = 5x^2 + 4x - 6$$

Which gives us $p_1(x) = p_e(x^2)$ and $p_2(x) = p_o(x^2)$. Ultimately, $p(x) = p_e(x^2) + xp_o(x^2)$.

Now, if p is of degree d then p_e and p_o are of degree at most $d/2$. Since it is obviously faster to evaluate smaller polynomials, we can use this as the basis for our algorithm.

Another key idea is to reuse computations. Namely, if we know $p_e(x^2)$ and $p_o(x^2)$ then we can write $p(x) = p_e(x^2) + xp_o(x^2)$ and $p(-x) = p_e(x^2) - xp_o(x^2)$. Notice that both computations utilize $p_e(x^2)$ and $p_o(x^2)$. This can offer a significant speedup when combining answers to subproblems.

We now want to look at the correct basis to evaluate the polynomial in. In example 2.1, we used the points $x = 0, 1, 2$ and $x = 0, \pm i$. We want a basis that is conducive to repeatedly taking the square root of numbers to apply the odd and even trick mentioned above. This immediately suggests the use of complex numbers. In fact, we will use the **roots of unity** that we mentioned in the complex number section for a reason that will become apparent in the analysis of the algorithm.

With all this in mind, the algorithm consists of the following basic steps:

1. Base case: if the polynomial is of degree 0, simply return the constant term.
2. Given that the input polynomial is of degree m, compute the nth roots of unity, where n is a power of 2 larger than m.
3. Break up the polynomial into components p_e and p_o.
4. Recursively call the procedure to evaluate the polynomials p_e and p_o on the $n/2$ roots of unity.
5. Using the result of the recursive procedure, evaluate the polynomials at the nth roots of unity using the relation $p(x) = p_e(x^2) + xp_o(x^2)$.

2.2 Pseudocode and Analysis

It is incredibly important that the n chosen is a power of 2 for the recursive structure to be maintained. This is because at the base case, we want to evaluate a linear polynomial at 2 points. Now let's look at the pseudocode for the algorithm.

This algorithm demonstrates the power of using the roots of unity: we can evaluate the polynomial at n points by evaluating the polynomials p_e and p_o at $\frac{n}{2}$ points and reusing those computations using our trick. Now we can truly see the importance of having n be a power of 2.By splitting up the polynomial into two other polynomials of even degree, we can ensure that no computation is wasted.

Algorithm 1 Fast Fourier Transform

```
procedure FFT(p, ω, n)
▷ ω is defined to be the first nth root of unity where n is a power of 2 larger than the degree of p

    If n = 2, return p(1), p(−1)
    Decompose p(x) into the polynomials p_e(x) and p_o(x)
    Call FFT(p_e, ω^2, n/2) and FFT(p_o, ω^2, n/2) and save the results
    for j = 0 to n − 1 do
        p(ω^j) = p_e(ω^{2j}) + ω^j p_o(ω^{2j})
    return p(ω^0), ..., p(ω^{n−1})
```

Example 2.4. Compute the FFT of the polynomial $p(x) = x^3 - 2x^2 + 4x + 3$.
Solution: We have in this case that $\omega = i$. We want to evaluate p at $1, i, -1, -i$. Breaking p up, we seek to evaluate $p_e(x) = -2x + 3$ and $p_o(x) = x + 4$ at the points ± 1. This will be our base case. Indeed, we have

$$p_e(1) = 1, \qquad p_e(-1) = 5, \qquad p_o(1) = 5, \qquad p_o(-1) = 3$$

Now we piece the solutions together:

$$p(1) = p_e(1^2) + 1 \cdot p_o(1^2) = p_e(1) + p_o(1) = 6$$

$$p(i) = p_e(i^2) + i \cdot p_o(i^2) = p_e(-1) + ip_o(-1) = 5 + 3i$$

$$p(-1) = p_e((-1)^2) + (-1) \cdot p_o((-1)^2) = p_e(1) - p_o(1) = -4$$

$$p(-i) = p_e((-i)^2) + (-i) \cdot p_o((-i)^2) = p_e(-1) - ip_o(-1) = 5 - 3i$$

Thus our answer is $[6, 5 + 3i, -4, 5 - 3i]$

Example 2.5. If we wanted to construct a complete value representation of the polynomial $p(x) = 6x^4 - 4x^2 + 7x + 1$, what is the smallest value of n we should pass into the FFT algorithm?

Solution: We need n to be a power of 2. Furthermore, since the polynomial is of degree 4, we need the value representation to contain at least 5 points. Thus we can take $n = 8$.

Like with most divide-and-conquer algorithms, we can use the master theorem to analyze the runtime. We start with a problem of size n (evaluating the polynomial p at n points) and split it into two problems of size $n/2$ (evaluating p_e and p_o at $n/2$ points). To combine the solutions to the subproblems, we require linear time. Thus the recurrence relation is

$$T(n) = 2T(n/2) + O(n)$$

and the master theorem gives us that the total runtime of the algorithm is $\Theta(n \log n)$. Note furthermore that the "inverse FFT" takes a polynomial from coefficient representation to value representation. With this in mind, we have a very powerful theorem regarding the inverse FFT.

Theorem 2.6 (Inverse FFT)**.** $FFT^{-1}(p, \omega, n) = \dfrac{1}{n} FFT(p, \omega^{-1}, n)$

There is a little nuance here - we treat p's coefficient and value representations as arrays of numbers. We won't prove the theorem above in this note, but it results again from the fact that our basis is the roots of unity. We can think of the FFT as a matrix transformation and the inverse matrix turns out to be very similar to the original. Along with the $\Theta(n \log n)$ runtime, this has powerful implications regarding the efficiency of solving other problems.

Example 2.7. Find an $\Theta(n \log n)$ time algorithm to multiply two polynomials of degree at most n.

Solution: If we were to multiply the polynomials using the naive method, you can verify that this takes time $\Theta(n^2)$. This is not good enough. However, can we use the FFT in a clever way to solve this problem?

The answer is yes. If I tell you that $p(1) = 4$ and $q(1) = 5$, can you tell me what $(p \cdot q)(1)$ is? Well, we can just evaluate each polynomial individually and multiply the result, giving us $(p \cdot q)(1) = p(1)q(1) = 20$. Thus if we have a sufficiently large value representation of p and q we can multiply their values at each basis point to obtain a value representation for $p \cdot q$. Furthermore we can use the fact that $(p \cdot q)(x)$ has degree at most $2n$.

Thus our algorithm will do the following: call the FFT on p and q with input size m, where m is a power of 2 larger than $2n$, multiply the resulting value representation, and call the inverse FFT algorithm.

Example 2.8. Suppose we want to use the algorithm above to multiply the polyomials $p(x) = 4x^3 + 5x + 7$ and $q(x) = 2x^7 - x^6 - 4x^3 + 2$. What value of n would we pass into the FFT algorithm?

Solution: Since p is degree 3 and q is degree 7, $p \cdot q$ will be degree 10. We need to choose a power of 2 greater than 10 so we will choose $n = 16$.