6.854 Advanced Algorithms

Lecture 26: 12/10/2003 Lecturer: Erik Demaine, David Karger

Scribes: Fen Zhao

Fast Fourier Transform

26.0.1 Motivation and Background

Almost anything that converts analog to digital uses the fast fourier transform. We will focus mainly o it's uses to multiply polynomials quickly.

Take any two polynomials

$$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

Their product is

$$C = AB = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j}$$

To evaluate all the coefficients would take quadratic time. However, we would like an algorithm that runs in $O(n \log n)$ time.

Note that there are many ways of representing polynomials. We can represent any polynomial of degree n as the product of n complex factors. If we have two polynomials in factored form, we can just concatenate the factors from both to make the new polynomial. However, factoring a polynomial is difficult.

26.0.2 FFT Algorithm

Instead we use the theorem that n points on a polynomial uniquely determine an $n-1$ degree polynomial. Therefore, we will take our polynomial in coefficient form, $a_1, a_2, \ldots a_{n-1}$, $b_1, b_2, \ldots b_{n-1}$, and convert it into point form, $A(x_0), A(x_1), \ldots A(x_{n-1})$, $B(x_0), B(x_1), \ldots B(x_{n-1})$. Their product is just the product at the two point values of the two polynomials: $C(x_i) = A(x_i)B(x_i)$.

But note that the polynomial C has up to twice the degree of A or B, so we have to make a small adjustment to our algorithm. We will take twice the degree of the polynomial number of points to evaluate. From here on, we will call n twice the degree of A or B, a.k.a the degree of C. We will also assume that n is a power of 2 (if not, we can just pad the polynomials with zero coefficients until they are).

How do convert $a_0 \to A(x_0)$? This process takes quadratic or cubic time. For example, we can write the polynomial $a_0 + a_1x + a_2x^2 + a_3x^3$ as $((a_3x + a_2)x + a_1)x + a_0$. This takes a linear number of multiplications to calculate, but we want to do this for a linear number of points, so it takes quadratic time.

However, we have an extra degree of freedom we have yet to use; we can choose the values of x to make our multiplication easier. Namely, we will pick the nth roots of unity.

The nth roots of unity are the n complex numbers that when taken to the nth power, equal 1. Geometrically, imagine the complex plane. The nth root of unity divide the unit circle into n equal pieces. They can be written as $e^{2\pi ik/n}$ for $k = 0...n-1$. Note that $(e^{2\pi ik/n})^n = 1$.

Now we can convert the coefficient form of a polynomial to point form using a nice, fast, divide and conquer algorithm. We can separate

$$A(x) = a_0 + a_1x + \ldots a_{n-1}x^{n-1}$$

into an even part

$$A^{even}(x) = a_0 + a2x + a_4x^2 \ldots$$

and an odd part

$$A^{odd}(x) = a_1 + a3x + a_5x^2 \ldots$$

Note that

$$A(x) = A^{even}(x^2) + xA^{odd}(x^2)$$

Now evaluate at $x = e^{2\pi ik/n}$. Note that squaring the nth roots of unity only give you two copies of the $n/2$ roots of unity. Therefore we can see that we need to evaluate A^{even} and A^{odd} at $n/2$ points (roots of unity), and each of them are polynomials of degree $n/2$. Therefore, we have reduced the problem into solving two subproblems of half the size. The recursion relation is $T(n) = 2T(n/2) + O(n)$, which means our algorithm runs in time $O(n \log n)$.

People have done this problem in many different ways. One can make the model cache oblivious, and get the sorting bound. People have studied parallel algorithms for FFT. FFTW, the Fastest Fourier Transform in the West, uses the cache oblivious model.

26.0.3 FFT Inverse Algorithm

But once we get the polynomial, how do we go back from the point form to coefficient form? The process is surprisingly similar to the algorithm from the previous part because we used the nth roots of unity.

The FFT is equivalent to multiplying a particular $n \times n$ matrix, lets call it FFT, by the vector

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ an-1 \end{pmatrix} \tag{26.1}$$

The entries of the FFT matrix are $\left(e^{2\pi ikj/n}\right)_{kj}$ for the kth row (which corresponds to the nth roots of unity) and the jth column (which correspond to the power). Look at the kth element of the resulting product; this is the product of the kth row with the input vector of coefficients.

$$(1\ e^{2\pi ik/n}\ e^{4\pi ik/n}\ \ldots)(a_0\ a_1\ \ldots\ a_{n-1})^\dagger = A(e^{2\pi ik/n})$$

The inverse transformation FFT^{-1} is just the multiplicative inverse of FFT. We will show the inverse of the matrix is

$$\frac{1}{n}\left(e^{-2\pi ikj/n}\right)_{jk}$$

We do this by multiplying it by FFT and showing that the result is the identity matrix.

$$\begin{aligned}(FFT * FFT^{-1})_{j,j'} &= \frac{1}{n}\left(e^{-2\pi ikj/n}\right)_{jk}\left(e^{2\pi ikj'/n}\right)_{kj'} \\ &= \sum_{k=0}^{n-1}\frac{e^{2\pi ik(j'-j)}}{n}\end{aligned}$$

Note that if $j' = j$, then the sum goes to 1. Otherwise use the identity

$$\sum_{k=0}^{n-1} z^k = \frac{z^n - 1}{z - 1}$$

Substitute $z = e^{2\pi i(j'-j)/n}$ and note that $z^n - 1 = 0$ because z is an nth root of unity. So the product is the identity.

So we can use the same decomposition as before:

$$\frac{1}{n}A(e^{-2\pi ik/n}) = \frac{1}{n}\left(A^{even}(e^{-4\pi k/n}) + e^{-2\pi ik/n}A^{odd}(e^{-4\pi ik/n})\right)$$

So we use the same procedure as before, and have the same running time.

26.0.4 Fourier Transform

Why is this process called the Fourier transform? Because it is the discrete version of a continuous idea called the Fourier transform.

Discrete:

$$f_k = \sum_{j=0}^{n-1} e^{2\pi ikj/n} a_j$$

Continuous:

$$f(k) = \int e^{2\pi ikt} a(t)dt$$