CS 6505, Computability and Algorithms

Fast Fourier Transform

Spring 2010; Lecturer: Santosh Vempala

1 Evaluating polynomials at many points

Suppose that we want to evaluate a polynomial $A(x) = a_0 + a_1x + \cdots + a_nx^n$ at many points – say n points in all. If we were just evaluating A at one point, say x_0, then we can naively perform the multiplication in $O(n^2)$ multiplications, but divide-and-conquer algorithms should make us think that we can do better. First, we can rewrite our computation of A as

$$A(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + xa_n)))$$

This is known as ***Horner's Rule***: each coefficient a_i is associated with a single multiplication with x. So calculating A can be done in n multiplications for a single value of x. If we want to use this method to evaluate, say, n points, then we need $O\left(n^2\right)$ multiplications.

Can we do better than this? With divide-and-conquer methods, we would want to find some way to eliminate redundant multiplications. If the x_j are chosen randomly, it seems like we will have no hope for overlapping or weeding out our multiplications. What if we chose our x_j cleverly so that a multiplication for one gives us the same answer for some other x_i? This is at the heart of the ***Fast Fourier Transform***: we will choose our x_j so that we can cut out multiplications.

Suppose we choose $x_j = -x_i$. Then all of the even monomials are the same, and the odd monomials have opposite sign. In other words, $a_{2k}x^{2k} = a_{2k}(-x)^{2k}$ and $a_{2k+1}x^{2k+1} = -a_{2k+1}(-x)^{2k+1}$ for $0 \leq k \leq \frac{n}{2}$. There is a set of numbers that have this property. In complex numbers, we have n solutions to the equation $x^n = 1$, and these solutions are known as the n^{th} ***roots of unity.*** These are denoted as $1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1}$, where $\omega = e^{\frac{2\Pi i}{n}}$.

There are some special properties of the nth roots of unity that we can see right away. For example, if n is even and $x^n = 1$, then $(-x)^n = 1$. How do we find $-x$? If n is even, then we have $\omega^{\frac{n}{2}} = -1$. So if $x = \omega^k$, then we know that $\omega^{k+\frac{n}{2}} = x * -1 = -x$.

Now let's take a look at our polynomial $A(x)$ again. Let's examine $A(\omega)$ and A(-ω):

$$\begin{aligned} A(\omega) &= a_0 + a_1\omega + a_2\omega^2 + a_3\omega^3 + \ldots \\ A\left(\omega^{\frac{n}{2}+1}\right) = A(-\omega) &= a_0 - a_1\omega + a_2\omega^2 - a_3\omega^3 + \ldots \end{aligned}$$

This looks like something that can be tackled with divide-and-conquer techniques. All of the even-exponent monomials, such as $a_0, a_2\omega^2, a_4\ \omega^4$, and so on, are the same in $A(\omega)$ and $A(-\omega)$, while all of the odd-exponent monomials, such as $a_1, a_3\omega^3, a_5\omega^5$, and so on, are of the opposite sign in $A(\omega)$ and $A(-\omega)$.So now we can split up A into two polynomials of even and odd degree, say A_0 and A_1, as follows:

$$A(\omega) = A_0(\omega^2) + \omega A_1(\omega^2)$$

$$A_0(\omega) = a_0 + a_2\omega + a_4\omega^2 + \cdots + a_d\omega^{\frac{d}{2}}$$

$$A_1(\omega) = a_1 + a_3\omega + a_5\omega^2 + \cdots + a_{d-1}\omega^{\frac{d-1}{2}}$$

So what has happened? We have broken the original polynomial A into two smaller polynomials A_0 and A_1 of degree $\frac{d}{2}$. How many distinct n^{th} roots of unity (ω^k) do we need? We don't need to calculate all n - since we're squaring them, we only need $\frac{n}{2}$ of them. So our original problem has now been halved: we originally were evaluating A, which has degree d, at n points, and now we're evaluating two polynomials A_0 and A_1 that are of degree $\frac{d}{2}$ at $\frac{n}{2}$ points. So our recurrence for evaluating A will involve two variables: n and d. If $n = d$, then we have:

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

Then we can evaluate A over n points in time $O(n \log n)$. Now what if $n \neq d$? If $d = 1$, then all polynomials are of degree 1 and it just takes $O(n)$ time to evaluate them. Otherwise, we have

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

From Horner's Rule, we also have $T(1, d) = O(d)$ (i.e., we're evaluating one point of degree k). Then we can derive $T(n, d) = O(d \log n)$. Likewise, if the degree d is larger, then we could derive a running time of $O(n \log d)$.

2 Equivalence between function values and coefficients

So we can go from values of the polynomial, say $A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1})$, to the coefficients $a_0, a_1, \ldots, a_{n-1}$ of A (so we are assuming that $d = n$)? If we want to calculate all of A's values, we could represent them in matrix form as follows:

$$\begin{pmatrix} A(1) \\ A(\omega) \\ \vdots \\ A(\omega^{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ \vdots & & & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

We can write the nxn matrix of ω values as $M(\omega)$. It is also known as the ***Vandermonde matrix***.

How do we solve for the coefficients of A? If we have to calculate an inverse naively, then this calculation could take a long time to compute – $O(n^3)$. However, our nxn matrix with the values of ω^k has some special properties. First, note that

$$\omega^n - 1 = (\omega - 1)\left(\omega^{n-1} + \omega^{n-2} + \cdots + \omega + 1\right)$$

If we take $\omega \neq 1$, then we have $\omega^{n-1} + \ \omega^{n-2} + \cdots + \omega + 1 = \frac{\omega^n - 1}{\omega - 1} = 0$ because we know that ω is an nth root of unity (and so $\omega^n = 1$). This helps when we generate $M(\omega)^{-1}$, since the identity matrix $I = M(\omega) * M(\omega)^{-1}$ will have 1s on the diagonal and 0s everywhere else.

We now claim that $M(\omega)^{-1} = \frac{1}{n} * M(-\omega)$. We can validate this for both the diagonal and off-diagonal entries. For the kth diagonal entries, we will take the dot product of the kth row of $M(\omega)$ with the kth column of $M(\omega)^{-1}$. This gives us

$$\begin{aligned}
&\frac{1}{n}\left(1 \ \omega^k \ \omega^{2k} \dots \omega^{k(n-1)}\right)\left(1 \ (-\omega)^k \ (-\omega)^{2k} \dots (-\omega)^{(n-1)k}\right) \\
&= \frac{1}{n}\left(1 \ \omega^k \omega^{2k} \dots \omega^{k(n-1)}\right)\left(1 \ \omega^{-k} \ \omega^{-2k} \dots \omega^{-(n-1)k}\right) \\
&= \frac{1}{n}\left(1 + 1 + \cdots + 1\right) = \ 1
\end{aligned}$$

For the off-diagonal entry at position (j,k), we multiply the j row of $M(\omega)$ with the k column of $M(-\omega)$. That gives

$$\begin{aligned}
&\frac{1}{n}\left(1 \ \omega^j \omega^{2j} \dots \omega^{(n-1)j}\right)\left(1 \ \omega^{-k} \ \omega^{-2k} \dots \omega^{-(n-1)k}\right) \\
&= \frac{1}{n}\left(1 + \ \omega^{j-k} + \omega^{2j-2k} + \dots \omega^{(n-1)(j-k)}\right) \\
&= \frac{1}{n}\frac{\omega^{n(j-k)} - 1}{\omega^{j-k} - 1} = 0
\end{aligned}$$

So the inverse matrix $M(\omega)^{-1}$ is easy to calculate. That means that, using the method we previously described, we can recover the coefficients of the polynomial A in time $O(n \log n)$. This has important applications to signal analysis, where we can uncover the frequencies of a signal.

3 Multiplying Polynomials

Suppose that we now have two polynomials, $A(x)$ and $B(x)$, of degree d, and we want to calculate $C(x) = \ A(x)B(x)$. Then $C(x)$ has degree $2d$, so it is determined by any $2d+1$ points. So we can determine enough information to reconstruct $C(x)$ from any $2d+1$ values of $A(x)$ and $B(x)$. So our procedure for determining C is as follows:

1. Calculate $A(x)$ and $B(x)$ at $n = 2d + 1$ points, which will require time $O(n \log n)$;
2. Calculate $C(x) = \ A(x)B(x)$ at the n selected points, which will require time $O(n)$;
3. Determine C's coefficients, which will take time $O(n \log n)$.

All three of these steps have already been described in detail. We just need to select the n^{th} roots of unity for the n selected points, and we're done.

4 String Matching

Suppose that we have a pattern $p = p_{m-1}p_{m-2}\dots p_0$ that we want to match against an n-character string $s = s_{n-1}s_{n-2}\dots s_0$. We could compare the m-character pattern against all possible $n-(m-1)$ starting positions of the pattern, but that gives a running time of $O(m(n-m+1))$. For example, take $p =$ abba and $s =$ aababbabba. Then we might have to compare abba against aaba, abab, babb, and so on. Can we do better using the FFT?

Let's assume that the pattern and string have the same binary characters, say a and b. Now let's map a to -1 and b to 1. Then the product of the mapped pattern, *abba* $\rightarrow$ (-1)11(-1), with the first four characters of the string, *aaba* $\rightarrow$(-1)(-1)1(-1) gives us a dot product of (-1)(-1)+1*(-1) +1*1 +(-1)(-1) = 2. When we do have a match, the dot product is (-1)(-1)+1*1+1*1+(-1)(-1) = 4. So when we have a match, each term in the dot product is 1 and the dot product is m. If we do not have a match, then at least one term in the dot product is -1 and so the dot product is less than m.

This looks just like the polynomial multiplication that we just saw – the pattern can be translated to one polynomial, and the text string can be translated to another polynomial. We want to know if the text starting at position k agrees with the input pattern. Let the pattern's polynomial $A(x) = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_0$ and take the m text polynomial to be $B(x) = s_{n-1}x^{n-1} + s_{n-2}x^{n-2} + \cdots + s_0$. Then $C(x) = A(x)\,B(x)$ will not give us information about whether the pattern matches up at position k in the text – we cannot just look at a coefficient of C and determine if we have m matches at the corresponding text position. Instead, we want coefficient c_k to tell us if position k in the text matches with position m in the pattern, if position $k+1$ in the text matches with position $m-1$ in the pattern, and so on. So we want to know if $s_k = p_{m-1}$, $s_{k+1} = p_{m-2}$, and so on until $s_{k+m-1} = p_0$. Notice that the sum of the two indices is always $k+m-1$.

Our insight is to reverse the order of one of these strings with respect to how its polynomial is created. So now let's try $A(x) = p_m + p_{m-1}x + \cdots + p_0x^m$ (i.e., $A(x)$ with its coefficients reversed) and use the same $B(x)$. We can see that the j^{th} coefficient of C, which we will denote c_j, is as follows:

$$c_j = \Sigma_i p_i s_{j-1}$$

So we get exactly what we want: if we want to know if the text and pattern match up in the k^{th} position, then we just need to examine c_k.

How does FFT help with pattern matching? We can see that we have just reduced the problem of pattern matching to polynomial multiplication: we multiplied the polynomial $A(x)$ with $B(x)$. The polynomial multiplication algorithm above tells us that we can do this in $n \log m$, where n is the length of the text and m is the length of the pattern. This is not the last word on string matching, but it is an interesting application of FFT and divide-and-conquer methods in general.