CS 5150/6150: Graduate Algorithms **Aug 29, 2007**

Lecture 4: FFTs, Convolutions, and Multiplying Polynomials

Lecturer: Suresh Venkatasubramanian *Scribe:*

4.1 Convolutions

In this lecture, we will study the role of the *Fast Fourier Transform*[] in helping us multiply polynomials. In order to do this, we start with a related notion.

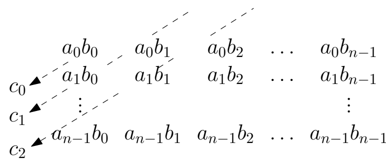
Let $a = \{a_0, a_1, \ldots a_{n-1}\}$ and $b = \{b_0, b_1, \ldots b_{n-1}\}$ be two vectors. Their *convolution* $c = a * b$ is defined to be the vector whose k^{th} component is given by

$$c_k = \sum_{\substack{i+j=k \\ 0 \le i,j \le n}} a_i b_j$$

Looking at the first few components of c,

$$\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_1 b_0 + a_0 b_1 \\
c_2 &= a_2 b_0 + a_1 b_1 + a_0 b_2 \\
\ldots
\end{aligned}$$

One way of visualizing the components of the convolution is to write the term-wise products as a matrix, and sum across the diagonals:



Notice that if a and b have n elements, then $c = a * b$ will have $2n - 1$ elements. Also, a and b don't have to have the same number of terms; to use the matrix analogy above, all this means is that we have a rectangular matrix rather than a square one. The construction of c remains the same as before: now c will have $n + m - 1$ elements if a and b have n and m elements respectively. A further point to keep in mind is that the convolution is *symmetric*: that is, $a * b = b * a$.

The convolution is a natural vector operator that appears in many guises. The most immediate example comes from the multiplication of polynomials. Suppose we wished to multiply polynomials $A(x) = \sum_0^{n-1} a_i x^i$ and $B(x) = \sum_0^{n-1} b_i x^i$. If we write down the product $C(x) = A(x) \cdot B(x) = \sum_0^{2n-2} c_j x^j$, then merely by comparing coefficients, we can see that the vector of coefficients $c = (c_0, c_1, \ldots c_{2n-2})$ can be written in terms of the coefficient vectors a and b as

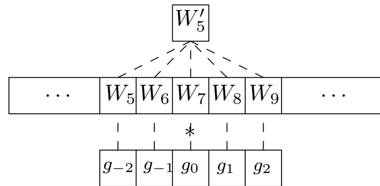
$$c = a * b$$

A second application of the convolution operator comes from the common *masking* operation in computer graphics. Suppose we have an image (which we'll pretend is one-dimensional for now), and we want to *blur* it using a Gaussian filter. We can represent the Gaussian filter discretely using a vector g of length $2b+1$, where $g_i = e^{-(b-i)^2}$, $-b \leq i \leq b$. The parameter b is the bandwidth of the filter, which controls how influential it is.

Let's say the image is represented by the vector W. To blur the image with the filter, we compute the blurred image W' by setting

$$W'_k = \sum_{i=-b}^{b} g_i W_{k+i}$$

Pictorially, imagine a filter with bandwith 2 centered at W_7:



The form of the expression $\sum_{i=-b}^{b} g_i W_{k+i}$ doesn't match what we'd expect from a convolution: we'd want the indices of the two variables to sum to a constant, rather than having their difference be a constant. It turns out that this doesn't matter ! We can convert this to a convolution by reversing g to give g', where $g'_i = g_{-i}$. Then we can rewrite the above expression as $\sum_{i=-b}^{b} g'_{-i} W_{k+i}$. Reassigning $j = -i$, the expression then becomes $\sum_{j=-b}^{b} g'_j W_{k-j}$, which is the standard form of a convolution.

4.1.1 The Complexity of Convolution

How long does it take to compute a convolution ? It's clear that we can compute all elements of the convolution in $O(n^2)$ time. This follows from the matrix interpretation of the convolution; we're summing elements of the matrix, there are n^2 of them, and each element occurs in exactly one term.

But the input vectors are of length n, and the output vector is of length $2n-1$. It seems wasteful that we'd want to take $O(n^2)$ time to compute a linear-sized output. In fact, we can compute a convolution in $O(n \log n)$ time using a complicated divide-and-conquer, and in the next sections we'll see how it's done.

4.2 Multiplying Polynomials

We already saw that a convolution can be used to multiply polynomials. We'll now flip this around, and use algebraic properties of polynomials to multiply them efficiently, and thus compute a convolution.

4.2.1 Representations And Operations

The standard way of representing a polynomial $p(x)$ is what we call the *coefficient representation*: we store the coefficient vector $p = (p_0, \ldots p_{n-1})$. Using this representation, we can evaluate polynomials using Horner's rule:

$$p(x) = p_0 + x(p_1 + x(p_2 + \ldots + (x \cdot p_{n-1})) \ldots)$$

The advantage of using Horner's rule is that it gives us a linear time procedure for evaluation. Multiplying two polynomials, as we discussed earlier, is more expensive. It can take $\Theta(n^2)$ time, or $O(n^{\log 3})$ if we're willing to be slightly clever.

There is however a different representation in which multiplication is a lot easier. We'll call it the *point-based representation*. Fix a set of values $x_0, x_1, \ldots x_{n-1}$, and write down the value of $p(x)$ at these values, giving a point vector $p = (p(x_0), p(x_1), \ldots p(x_{n-1}))$. By a fundamental theorem in polynomial interpolation called the *unisolvence theorem*, we know that as long as the x_i are distinct and n in number (note that the degree of $p(x)$ is $n-1$), there is a 1-1 map between point vectors and polynomials, and we can reconstruct $p(x)$ uniquely.

Now, multiplying two polynomials is easy: merely multiply the components of the two vectors. This can be done in linear time. However, there is one catch. *Evaluation* of polynomials is now hard. Since we only have the value of the polynomial at specific points, there doesn't appear to be any way of evaluating $p(x)$ at some *different* point x' other than reconstructing $p(x)$ and evaluating it using Horner's rule.

However, reconstructing $p(x)$ is not easy. The direct option is to solve the linear system of equations for the coefficients of $p(x)$. Given data pairs $(x_i, y_i) = p(x_i)$, this means that we need to solve the linear system

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

The matrix on the left hand side is actually a well-known matrix called the *Vandermonde matrix*. An interesting property of this matrix is that its determinant is the product

$$\Pi_{i,j}(x_i - x_j)$$

and is thus nonzero as long as the x_i are distinct, implying a unique solution to the linear system and thus a unique polynomial.

Inverting this matrix to solve the linear system requires $O(n^3)$ time. However, the special structure of the matrix allows us to do better by using a form of the interpolation polynomial called the *Lagrange* form:

$$\mathcal{L}(x) := \sum_{j=0}^{k} y_j \ell_j(x)$$

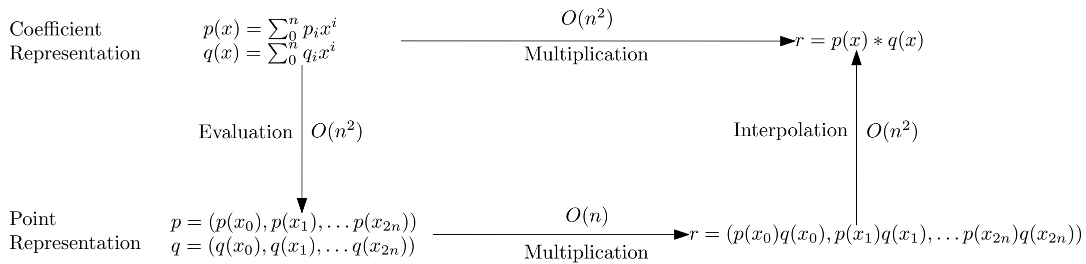
$$\ell_j(x) = \prod_{i=0,\, i\neq j}^{k} \frac{x - x_i}{x_j - x_i} = \frac{x - x_0}{x_j - x_0} \cdots \frac{x - x_{j-1}}{x_j - x_{j-1}} \frac{x - x_{j+1}}{x_j - x_{j+1}} \cdots \frac{x - x_k}{x_j - x_k}$$

The useful property of this form of the interpolation polynomial $p(x)$ is that it allows us to evaluate $p(x)$ in $O(n^2)$ time rather than in $O(n^3)$ time **(why?)**.

To summarize, here is how the picture looks:

	Evaluation	Multiplication
Coefficient Rep.	$\Theta(n)$	$\Theta(n^2)$
Point Rep.	$\Theta(n^2)$	$\Theta(n)$

It thus seems clear that if we want to multiply two polynomials, we should convert them to point representation, do the multiplication, then convert back. However, since the multiplication of two polynomials results in a polynomial of twice the degree, we need $2n$ evaluation points for both polynomials so that we have enough points to reconstruct their product. But evaluating these many values will take $2n * \Theta(n) = \Theta(n^2)$ time, leaving us with the dismal diagram:



Looking at the picture, it is clear that although we can multiply two point representations in linear time, there is no way to get to this representation, or get from it, without expending $\Theta(n^2)$ time !

The only leverage we have is in the choice of values x_i on which we evaluate the polynomials. It is possible that if we choose these correctly, we can perform the evaluation and interpolation faster than the worst-case n^2 bound. This is in fact what we will do, via the fast Fourier

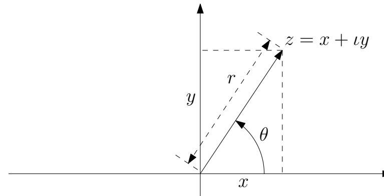
transform. Using a specially constructed set of values, we will be able to perform both the evaluation *and* interpolation in $O(n \log n)$ time, giving us an overall running time of $O(n \log n)$ for the multiplication operation.

4.3 The Fast Fourier Transform

Once again, we start with some background. The values we will be choosing are the *complex roots of unity*. Let's examine their definition and some of their properties.

4.3.1 Complex Roots Of Unity

Recall that a complex number z is of the form $z = x + \iota y$, where $\iota = \sqrt{-1}$. We can also write z in *polar form* $z = re^{\iota\theta}$, where $r = |\sqrt{x^2 + y^2}|$ and $\theta = \tan^{-1} \frac{y}{x}$. Complex numbers can be interpreted geometrically in the two-dimensional plane by using the y-axis to represent the imaginary part and the x-axis to represent the real part of a complex number. Then, we immediately see that θ is the angle that the vector (x, y) makes with the x-axis, and r is the magnitude of this vector.



A kth *root of unity* is a complex number z such that $z^k = 1$. There are exactly k such numbers z, and they can be written as $z = e^{2\pi\iota j/k}$, where $j = 0, 1, \ldots k-1$. The root $e^{2\pi\iota/k}$ is denoted ω_k, the *principal root of unity*. All other roots are powers of this root, denoted $\omega_k^2, \omega_k^3, \ldots$. Geometrically, we can think of the roots as vectors whose endpoints are evenly spaced points on the unit circle on the complex plane. Note that these roots form a group under multiplication; this group has the same structure as the additive group $(Z_n, +) \mod n$.

The complex roots have some interesting properties, which you can verify.

- $\omega_{dn}^{dk} = \omega_n^d$. (cancellation)
- $\omega_n^{n/2} = \omega_2 = -1$.
- $\omega_n^{k+n/2} = -\omega_n^k$

A crucial property that we will use is the so-called "halving lemma":

Lemma 1 (Halving) *For even $n > 0$, the squares of the n complex n^{th} roots of unity are the $n/2$ complex $n/2^{th}$ roots of unity.*

Proof: Consider one such root ω_n^k, where $k \leq n/2$. Squaring, we get $\omega^2 k_n$, which by the cancellation rule gives us $\omega_{n/2}^k$, which is an $n/2^{\text{th}}$ root of unity. If the root is instead of the form $\omega_n^{k+n/2}$, where $k \leq n/2$, then we can rewrite it as $-\omega_n^k$, which has the same square as ω_n^k. ■

The halving lemma is crucial to a divide-and-conquer attack on the polynomial evaluation and interpolation problem, as it intuitively allows us to go from evaluations on a degree-n polynomial to evaluations on a degree-$(n/2)$ polynomial.

4.3.2 Fourier Transforms And The DFT

The Fourier transform of a complex-valued function $x(t)$ is another complex-valued function $X(f)$ given by

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-\iota 2\pi f t} dt$$

The *discrete Fourier Transform* or DFT is the discrete analog of this transform, mapping a vector $a = (a_0, a_1, \ldots a_{n-1})$ to the vector $y = (y_0, y_1, \ldots y_{n-1}) = \mathbf{DFT}_n(a)$, where

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

If, as usual, we think of a as a vector of coefficients of the polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$, then y_k is merely an evaluation of $A(x)$:

$$y_k = A(\omega_n^k)$$

This relation is the crucial link we're looking for. If we can compute the DFT of a vector quickly, then we essentially get all the polynomial evaluations we want *simultaneously*. This computation is known as the *Fast Fourier Transform (FFT)*.

4.4 The Algorithm

Finally, after all this buildup, we're ready for an algorithm. In order to solve the polynomial evaluation and interpolation problem efficiently, we will use the complex roots of unity as points of evaluation, and will use the FFT to compute the DFT (and therefore all the evaluations) in $O(n \log n)$ time, using divide-and-conquer. Remember that even though the degree of the polynomial $A(x)$ is $n - 1$, and a unique reconstruction requires evaluation and n points, we need to evaluate it at $2n$ points. This is because we will need these $2n$ points when reconstructing the product that may potentially have degree $2n - 2$.

4.4.1 Evaluation

Recall that our goal is to evaluate $A(x)$ at the $2n$ $2n^{\text{th}}$ roots of unity more efficiently than the brute force $O(n^2)$ algorithm. Let's start by rewriting $A(x)$, separating out the terms with even and odd powers of x. Recall that when we studied Karatsuba's algorithm for integer multiplication, we divided using high and low order bits, which is different to what we're doing here.

For simplicity, assume that n is a power of 2. From a formal perspective, this assumption is permissible, because we can always "pad" $A(x)$ by adding terms of the form $0 \cdot x^{n+1} + 0 \cdot x^{n+2}$ to make sure that this is true. Although this increases the size of the input, it's still no more than a constant factor away from the true input size (**why?**).

Let $A_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 \ldots a_{n-2} x^{(n-2)/2}$, and let $A_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 \ldots a_{n-1} x^{(n-2)/2}$. It is easy to see that

$$A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2) \tag{4.1}$$

This is the core of our divide-and-conquer method. Both A_{even} and A_{odd} have degree $(n-2)/2$, which is less than half the degree of $A(x)$, and so if we can compute their evaluations recursively, then we might be able compute $A(x)$ using only a constant number of evaluations.

The only remaining problem is that we need to evaluate $A(x)$ on $2n^{\text{th}}$ roots of unity, whereas the polynomials A_{even} and A_{odd} will be evaluated on n^{th} roots of unity. This is where the Halving lemma comes to our rescue.

To evaluate $A(x)$ on ω_{2n}^j, we need to evaluate A_{even} and A_{odd} on $(\omega_{2n}^j)^2$. But we know from the Halving lemma that this is an nth root of unity ! This completes the recursion specification, and gives us a divide-and-conquer algorithm for evaluating $A(x)$ governed by the recurrence

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

which yields an $O(n \log n)$ running time. Algorithm 1 describes the final algorithm.

What If We Didn't Use The FFT ? The recurrence (4.1) is true regardless of which values we use for evaluation. So why do we need the complex roots of unity, and the FFT ? The key structural property we need is the Halving Lemma (Lemma 1), which *allows us to reduce the set of values to evaluate by half at each step*. Suppose this were not the case, and that at each step of the recursion, we had to evaluate all the original values. Our running time is now described by two parameters; $T(n, m)$ is the running time of the recursive evaluation procedure on a polynomial of degree $n-1$ with m variables. Our goal is to determine $T(n, n)$.

Expanding the recurrence, we get

$$T(n, m) = 2T(n/2, m) + m$$
$$T(2, m) = m$$

where the second equation comes from the fact that in order to do m evaluations on a polynomial of degree 1, we need to spend m units of time. If we unfold the recursion tree, we get a

Algorithm 1 Recursive-FFT

Input: $a = (a_0, \dots a_{n-1})$
  if n=1 **then**
    return a
  end if
  $\omega_{2n} = e^{-\iota 2\pi/2n}$
  $\omega = 1$
  $a_{\text{even}} = (a_0, a_2, \dots a_{n-2})$
  $a_{\text{odd}} = (a_1, a_3, \dots a_{n-1})$
  y_{even} = Recursive-FFT(a_{even})
  y_{odd} = Recursive-FFT(a_{odd})
  for $k = 0$ to $n - 1$ **do**
    $y_k = y_{\text{even},k} + \omega y_{\text{odd},k}$
    $y_{n+k} = y_{\text{even},k} - \omega y_{\text{odd},k}$
    $\omega * = \omega_{2n}$
  end for
  return y

tree of height $\log n$, with $n-1$ leaves of the form $T(2, m)$, where the "cost" of each leaf is m. This yields a running time of $\Omega(nm)$, and therefore $T(n, n) = \Omega(n^2)$.

Essentially, to make the scheme work, the recurrence needs to look more like

$$T(n, m) = 2T(n/2, m/2) + m$$
$$T(2, 2) = c$$

and that is what the Halving Lemma guarantees.

4.4.2 Interpolation

We've only completed one part of the process. We can now evaluate the two polynomials to be multiplied in $O(n \log n)$ time, and we can perform the point-wise multiplication in linear time. But we now need to reconstruct the coefficient representation of the product via interpolation, and using the Lagrange form takes $\Theta(n^2)$ time.

Once again, the FFT comes to our rescue. Earlier, we talked about the general Fourier transform that maps a signal $x(t)$ to a signal $X(f)$. There is an inverse transform as well, which can be written as

$$x(t) \int_{-\infty}^{\infty} X(f) e^{\iota 2\pi f t} df$$

The discrete analog is the transform

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj}$$

It's easy to see why this is true. Substituting the expression for y_j from earlier, $y_k = \sum_{i=0}^{n-1} a_i \omega_n^{ki}$, we get

$$\begin{aligned} a_j &= \frac{1}{n}\sum_{k=0}^{n-1} y_k \\ &= \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{-kj} \sum_{i=0}^{n-1} a_i \omega_n^{ki} \\ &= \frac{1}{n}\sum_{k=0}^{n-1}\sum_{i=0}^{n-1} a_i \omega_n^{k(i-j)} \end{aligned}$$

Here's a another useful property of the ω_n^k (**check it!**). For any k that's not divisible by n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

If we now look at the right hand side above, the case when $i \neq j$ can be computed using this fact, and the terms go to zero. When $i = j$, we get the expression

$$\begin{aligned} a_j &= \frac{1}{n}\sum_{k=0}^{n-1} a_j \\ &= a_j \end{aligned}$$

The inverse transform can be used to go from the point-wise representation to the coefficient representation, in the reverse manner as we did before. Given a sequence of evaluated values, we compute the inverse DFT and extract the coefficients of the desired polynomial.

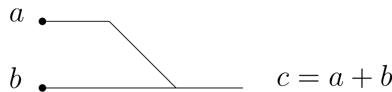
But how might we do this ? Not surprisingly, we can use the FFT, by making the following trivial observation: $\omega_n^k = \omega_n^{n-k} = \omega_n^{-k}$. This means that we can write the inverse DFT as a DFT, merely by reversing the evaluations ! Now we are home free: using the FFT we can reconstruct the product in time $O(n \log n)$, completing the description of the algorithm.

4.4.3 A Schematic Drawing

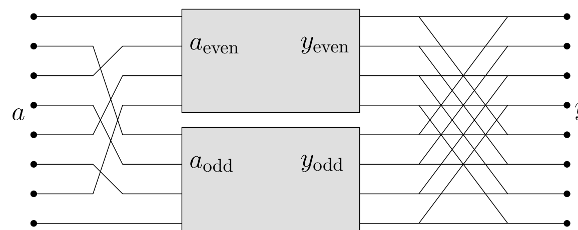
One of the reasons the FFT has been so influential is that the above recursive algorithm can be "unwound" into an iterative method that is easy to describe using simple wiring patterns,

which allows for both parallelization and hardware acceleration. We briefly explain how this works.

We can depict the top level recursive structure of the FFT by a *wiring diagram*, where each wire "carries" a particular input value. Two wires that merge represent a combination operation; for clarity I'll omit the operation performed (it's either an addition or subtraction). For example, a wiring pattern that adds two numbers might be drawn like this:



The FFT wiring diagram, with blocks for the recursive step, looks like this:



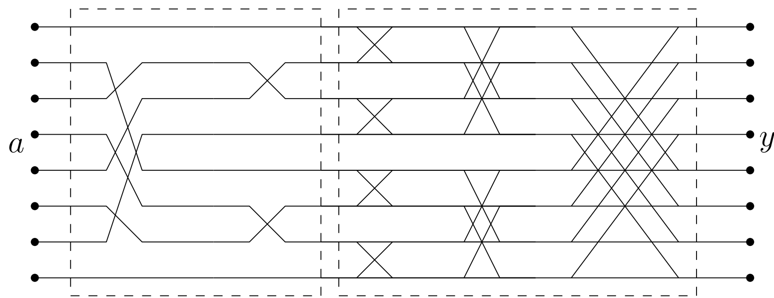
If we unwrap the recursive blocks, we get a complete wiring pattern that has two distinct parts. The left side is a bit-reversal network; it reorders the wires so that they are in ascending order of their bit-reverse. For example, wire 001 becomes wire 100, the fifth wire, and wire 110 becomes wire 011, the fourth wire. The right hand side of the circuit is a *butterfly network*, a common wiring system to connect processors in large parallel systems.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw Hill, 2nd edition, 2004.

[2] Jeff Erickson. Fast fourier transforms. http://www.cs.uiuc.edu/class/sp07/cs473g/lectures/03-fft.pdf, Spring 2007.

[3] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley, 2005.



a
y