6.S897 Algebra and Computation February 21, 2012

Lecture 4

Lecturer: Madhu Sudan *Scribe: Zachary Abel*

1 Administrivia

- Email list: Please make sure you are on the course email list. Email Prof. Sudan to be added.
- Blog: The course now has a blog: `http://algebraandcomputation.wordpress.com/`. Responses to "your favorite or most surprising algorithms" have been posted there, and you are encouraged to use this space to ask questions and contribute to course-related discussions. Instructions for posting can be found on the blog.
- Scribing: Please sign up to scribe at least one lecture, whether you are taking the course for credit or not. Instructions for choosing dates are on the course website.

2 Polynomial Arithmetic: Setup

We now dicuss algorithms for performing basic operations with polynomials, such as polynomial multiplication, division with remainder, and evaluation at multiple points. Most of today focuses on efficiently multiplying two degree n polynomials. We will see a simple $O(n \log n)$ Fourier transformation (FT) based algorithm under sufficiently nice conditions and then show how this gives rise to a general $O(n \log n \log \log n)$ algorithm.

2.1 Computational Model

While we are mostly concerned with polynomials over a field $\mathbb{F}$, it will be helpful to work more generally with polynomials over a commutative ring R. Where appropriate, we will assume that our polynomials are *monic*, i.e., have leading coefficient 1. This can always be arranged over a field (by scalar multiplication), but this extra assumption will be necessary over general rings, for example, to even define the division-with-remainder problem. (What are the quotient and remainder when dividing x^2 by $2x+1$ in $\mathbb{Z}[x]$?)

In order to assume as little about the representation of the base ring R as possible, we will measure the runtime of our algorithms in terms of basic operations in R. Specifically, we will separately keep track of three types of operations: additions in R (which are usually cheap), general multiplications in R (which are usually expensive), and "special" multiplications in R, such as by a power of 2 or other known constants. The identity of the "special" multiplications depends on the ring R and the nature of the algorithm, but we count these separately because they are usually cheaper than general multiplications.

Note also that polynomials are represented in dense form: a polynomial $f \in R[x]$ of degree n is represented by a list of $n+1$ coefficients in R, even if many of these coefficients are 0.

Here are some of the fundamental problems we are interested in solving efficiently. Let the polynomials have degree at most n unless otherwise specified:

1. Addition. Given $f, g \in R[x]$, compute $f+g$. This takes $O(n)$ additions.
2. Multiplication. Given $f, g \in R[x]$, compute $f \cdot g$. An algorithm based on the Fast Fourier Transformation will give an $O(n \log n)$ algorithm in special cases, which then leads to a general $O(n \log n \log \log n)$ algorithm.
3. Multipoint evaluation. Given f and $\alpha_1, \ldots, \alpha_n \in R$, compute $f(\alpha_1), \ldots, f(\alpha_n)$. This can be done in $O(n \cdot \text{poly} \log n)$ time, which seems surprising since each of the n independent evaluations would take $O(n)$ time if done separately. Somehow we save time by doing many at once.

4. Interpolation. Given $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n \in R$, find $f \in R[x]$ of degree at most $n-1$ such that $f(\alpha_i) = \beta_i$ for $1 \leq i \leq n$. This is the inverse of multipoint evaluation, and also has $O(n \cdot \text{poly}\log n)$ algorithms.

   This problem is not as well-behaved over general rings, because such an f may not exist, and/or there may be many solutions. For example, the interpolation problem $f(0) = f(1) = 0$, $f(2) = 1$ has no solutions over $\mathbb{Z}$ (over $\mathbb{Q}$ it is $f(x) = x(x+1)/2$). Over $\mathbb{Z}/6\mathbb{Z}$, $(x-2)(x-3)$ and the 0 polynomial both solve the interpolation problem $f(0) = f(2) = f(3) = f(5) = 0$.

   If R is a field, then the interpolating polynomial f will always exist uniquely. More generally, if f is an *integral domain* (doesn't contain zero-divisors), then any f will be unique if it exists.

5. Division with remainder. Given $f, g \in R[x]$ with g monic, compute the unique polynomials $q, r \in R[x]$ with $\deg r < \deg g$ such that $f = g \cdot q + r$. This too can be done in nearly linear time.

6. GCD. Given $f, g \in \mathbb{F}[x]$ (over a field!), compute their Greatest Common Divisor.

All of these algorithms are "folklore" from the 1960s, and appear in the introductory textbook *Algorithms and Data Structures* by Aho, Hopcroft, and Ullman (1983). These algorithms seem to have been dropped in more modern references like *Introduction to Algorithms* by Cormen et al. and thus seem less well-known.

Another important problem is that of polynomial factorization. The general problem of factoring a polynomial $f \in R[x]$ into irreducibles seems hard: for example, it includes factoring integers as a special case by taking $R = \mathbb{Z}$. But if we care only about breaking f into lower-degree pieces, and not about factoring scalars in R, then there are polynomial-time algorithms (Berlekamp and Zassenhaus, 1970s), though no linear or nearly-linear algorithms are yet known. The most efficient factoring algorithm currently known is due to Kedlaya and Umans in 2009, which factors polynomials in $\mathbb{F}_p[x]$ in $\tilde{O}(n^{1.5})$ time. Their methods involve yet another related problem called Modular composition: given $f, g, h \in R[x]$ of degree n, compute $f(g(x)) \bmod h(x)$.

3 Review of FT-based Multiplication

Here we review the efficient multiplication algorithm based on the Fourier Transform (FT), and we explore what assumptions need to be placed on the ring R to allow us to run this algorithm.

The basic framework for the interpolation-based polynomial multiplication algorithm is given as follows. Suppose that $f, g \in R[x]$ with $\deg f + \deg g < n$.

- Step 1 (Multipoint Evaluation). Pick $\alpha_0, \ldots, \alpha_{n-1} \in R$, and compute $f(\alpha_0), \ldots, f(\alpha_{n-1})$ and $g(\alpha_0), \ldots, g(\alpha_{n-1})$.
- Step 2 (Multiplication in R). Compute the n products $f(\alpha_i) \cdot g(\alpha_i)$ in R. This takes n general multiplications.
- Step 3 (Interpolation). Compute $h = f \cdot g$ as the[1] polynomial of degree $\leq n-1$ such that $h(\alpha_i) = f(\alpha_i) \cdot g(\alpha_i)$ for $0 \leq i \leq n-1$.

The FT-based method makes this algorithm efficient by choosing special elements $\alpha_0, \ldots, \alpha_{n-1}$ that enable Steps 1 and 3 to be accomplished efficiently (and uniquely, for Step 3). Specifically, it chooses $\alpha_i = \omega^i$ where $\omega \in R$ is a *primitive nth root of unity*:

Definition 1 *An element $\omega \in R$ is an nth* root of unity *if $\omega^n = 1$. It is* primitive *if $\omega^i \neq 1$ for all $1 \leq i \leq n-1$, or equivalently, if $\omega^d \neq 1$ for all proper divisors d of n.*

For the FT multiplication algorithm, we will assume $n = 2^m$ is a power of 2, $\deg f, \deg g < n/2$, and that a primitive nth root of unity $\omega \in R$ is known.

[1]Any such algorithm needs to come with a proof that this interpolation problem is uniquely solvable. This comes automatically if R is an integral domain, but we do not make this assumption.

3.1 Implementing Step 1: the FFT

The key idea to speeding up Step 1 is that $a \mapsto a^2$ is a 2-to-1 map from $\{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$ to $\{1, \omega^2, \omega^4, \ldots, \omega^{n-2}\}$, and that ω^2 is a primitive $n/2$th root of unity. So if we decompose $f(x) = f_0(x^2) + x \cdot f_1(x^2)$ (which amounts to reorganizing the monomials in f), the multipoint evaluation of f at the n powers of ω reduces to the multipoint evaluations of f_0 and f_1 at the $n/2$ powers of ω^2.

In detail, for any polynomial $f(x) = \sum_{i=0}^{n-1} c_i x^i \in R[x]$ with degree $\leq n-1$ and any primitive nth root of unity ω, define the *Fast Fourier Transform* as $\mathrm{FFT}(f(x), \omega) = \mathrm{FFT}((c_0, \ldots, c_{n-1}), \omega) = (f(1), f(\omega), \ldots, f(\omega^{n-1}))$. (For convenience, we overload the notation to accept either the polynomial or its list of coefficients). The following recursive algorithm computes $\mathrm{FFT}(f, \omega)$:

- Write $f(x) = f_0(x^2) + x \cdot f_1(x^2)$. Note that $\deg f_0, \deg f_1 < \frac{n}{4}$. (This takes $O(n)$ time and no ring operations.)
- Recursively compute $\mathrm{FFT}(f_0, \omega^2) = (f_0(\omega^{2i}))_{i=0}^{n/2-1}$, and likewise for $\mathrm{FFT}(f_1, \omega^2)$.
- Compute $f(\omega^i) = f_0(\omega^{2i}) + \omega^i \cdot f_1(\omega^{2i})$ for each $0 \leq i \leq n-1$, using a total of n special multiplications (by some power of ω) and n additions.

The number of additions and special multiplications on an instance of size n satisfies the recursion $T(n) = 2T(n/2) + O(n)$, so the algorithm uses $T(n) = O(n \log n)$ of each (and no general multiplications).

3.2 Implementing Step 3

Step 3 asks for an "inverse Fast Fourier Transform", where we interpolate a polynomial from its evaluations at the roots of unity. Note that $\mathrm{FFT}(\cdot, \omega)$ is a linear operator on R^n. Under nice conditions, we can express the inverse FFT simply as follows:

Theorem 2 *Suppose $\frac{1}{n} \in R$ (equivalently, $\frac{1}{n} = \frac{1}{2^m} \in R$) and $\omega^{n/2} = -1$. Then $\mathrm{FFT}(\cdot, \omega)$ and $\frac{1}{n}\mathrm{FFT}(\cdot, \omega^{-1})$, as linear operators on R^n, are inverses.*

Observation 3 *This means that, for any n-tuple $v = (v_0, \ldots, v_{n-1}) \in R^n$, the polynomial h whose coefficients are given by $\frac{1}{n}\mathrm{FFT}(v, \omega^{-1})$ is the unique polynomial with $h(\omega^i) = v_i$ for $0 \leq i \leq n-1$. This both provides the algorithm for Step 3 as well as its proof of correctness.*

Proof Write $\mathrm{FFT}(\cdot, \omega)$ as a matrix $M(\omega)$ where $M(\omega)_{ij} = \omega^{ij}$ for $0 \leq i, j \leq n-1$. In other words, $\mathrm{FFT}((c_0, \ldots, c_{n-1}), \omega) = M(\omega) \cdot (c_0, \ldots, c_{n-1})^T$. The theorem reduces to the fact that $A = M(\omega)$ and $B = \frac{1}{n} M(\omega^{-1})$ are inverse matrices.

We have

$$(AB)_{ii} = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{ik} \omega^{-ik} = 1,$$

as necessary, and when $i \neq j$ we find

$$(AB)_{ij} = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{ik} \omega^{-jk} = \frac{1}{n} \sum_{k=0}^{n-1} (\omega^{\ell})^k$$

where $\ell = i - j \neq 0 \bmod n$. Multiplying this quantity by $1 - \omega^{\ell}$ results in $\frac{1}{n}(1 - \omega^{\ell n}) = \frac{1}{n}(1 - 1) = 0$, so it suffices to show that $1 - \omega^{\ell}$ is not a zero divisor. We claim more strongly that $1 - \omega^{\ell}$ is not a zero divisor for *all* integers ℓ not divisible by $n = 2^m$.

The following general fact is useful: if $a, b \in R \setminus \{0\}$ and ab is not a zero divisor, then a and b are not zero divisors. Indeed, if b were a zero divisor, then there would be some $c \in R \setminus \{0\}$ with $bc = 0$, which would imply $abc = 0$, contradicting our assumption that ab is not a zero divisor.

Now, back to the claim. Write $\ell = u \cdot 2^v$ where u is odd and $v < m$; we proceed by downward induction on v. If $v = m-1$ then $1-\omega^\ell = 1-(-1)^u = 2$, which is not a zero divisor because it is invertible. When $v \leq m-2$, $1-\omega^{2\ell}$ is not a zero-divisor by induction and $(1-\omega^\ell)(1+\omega^\ell) = 1-\omega^{2\ell}$, so $1-\omega^\ell$ is not a zero-divisor by the fact above. ■

This concludes the analysis of the FT-based multiplication algorithm. To tie it all together, we have shown the following:

Theorem 4 *Let $n = 2^m$, let R be a commutative ring with $\frac{1}{2} \in R$, and suppose we are given a primitive nth root of unity $\omega \in R$ with $\omega^{n/2} = -1$. Then any polynomials $f, g \in R[x]$, each of degree at most $\frac{n}{2}-1$, can be multiplied using the the FT multiplication algorithm with $O(n \log n)$ additions, $O(n \log n)$ special multiplications by powers of ω, and n general multiplications.*

3.3 Drawbacks

While the FT algorithm is simple and convenient, it made a number of generally unrealistic assumptions about the ring R. First, it assumed that $\frac{1}{2} \in R$. Sometimes we can simply adjoin an inverse for 2 and proceed, but in many cases this is impossible. For example, if R is a field of characteristic 2 then $2 = 0$ can never be made into a unit. The following exercise describes an alternate method to deal with such fields.

Exercise 5 *Let $R = \mathbb{F}_{2^t}$ be a finite field of characteristic 2.*

1. *Given $f \in \mathbb{F}_{2^t}[x]$ with $\deg f < n$ and an element $\alpha \in \mathbb{F}_{2^t}$, compute $f_0, f_1 \in \mathbb{F}_{2^t}[x]$, each with degree less than $n/2$, such that $f(x) = f_0(x^2-\alpha x) + x f_1(x^2 - \alpha x)$. This should take $O(n \log n)$ time.*
2. *Let $S \subset \mathbb{F}_{2^t}$ be a subset of size $|S| = n$ that is an $\mathbb{F}_2$-subspace of $\mathbb{F}_{2^t}$, i.e., S is closed under addition. By picking $\alpha \in S \setminus \{0\}$, use the previous part to give algorithms for multipoint evaluation and interpolation over S, and use these to provide an $O(n \log^2 n)$ algorithm for polynomial multiplication in $\mathbb{F}_{2^t}[x]$.*

The other assumption made by the FT algorithm was access to a special root of unity $\omega \in R$ with $\omega^{n/2} = -1$. While many rings do not have such an element, the algorithm in the next section circumvents this by cleverly adjoining the desired root of unity.

4 Schönhage-Strassen Multiplication

Here we discuss the Schönhage-Strassen Multiplication algorithm, which works over any ring where 2 is not a zero-divisor. (By adjoining 1/2, we may as well assume that 2 is a unit.)

4.1 Motivation

The idea of the algorithm is that, while R may not have the desired nth root of unity ω with $\omega^{n/2} = -1$, we can replace R with $R' = R[y]/(y^{n/2}+1)$, where y now fills that role. The downside is that elements of R' are naturally represented by degree $\leq \frac{n}{2}-1$ degree polynomials over R, so general multiplications in R' are just as hard as the original problem.

To fix this, Schönhage-Strassen compromises between two opposing needs. Looking instead at $R'_\ell = R[y]/(y^\ell+1)$ for some $\ell < n/2$, choosing a larger ℓ provides a root of unity that allows the FT-algorithm to multiply larger-degree polynomials in $R'_\ell[x]$, but choosing smaller ℓ makes operations in R'_ℓ easier. The algorithm chooses $\ell = O(\sqrt{n})$, which balances both needs. Note that general multiplications in R'_ℓ are like degree-ℓ multiplications in $R[x]$, but as ℓ is much smaller than $n/2$ we can solve these by recursion.

4.2 Details

Our only assumption on R is that $2 \in R$ is a unit. Let $f, g \in R[x]$ have $\deg f, \deg g < n/2$, where $n = 2^m$ is a power of 2. Choose integers k and ℓ, both powers of two and both $O(\sqrt{n})$, satisfying $n = k\ell$ and $k < \ell/2$.

Define $D = R[y]/(y^\ell + 1)$. Note that the FT multiplication algorithm works over D, with $y \in D$ as the primitive 2ℓth root of unity, so polynomials in $D[x]$ of degree less than ℓ can be multiplied in $O(\ell \log \ell)$ additions, the same number of special multiplications by powers of y, and and $O(\ell)$ general multiplications in D. How are each of the operations in D implemented in terms of R? Additions in D are done coefficient-wise and take $O(\ell)$ R-additions each. Special multiplications by y^t take $O(\ell)$ time and require only permuting and negating the R-coefficients. General multiplication of elements of D can be solved recursively by multiplying two degree $< \ell$ polynomials in $R[y]$ and then reducing modulo $y^\ell + 1$.

It remains to see how the FT algorithm over D lets us multiply $f, g \in R[x]$. We do this by translating our degree $n/2$ multiplication problem in $R[x]$ into a degree $\ell/2$ problem in $D[x]$, via $R[x, y]$. Note that for any polynomial $s'' \in D[x]$, there is a unique *reduced representative* $s' \in R[x, y]$ that maps to s'' in the quotient and is reduced modulo $y^\ell + 1$. Now we can finally give the Schönhage-Strassen algorithm:

- First, let $f'(x, y) \in R[x, y]$ be the polynomial satisfying $f(x) = f'(x, x^{\ell/2})$. This is simply a rewriting of the monomials of $f(x)$ under the correspondence $x^{q \cdot \ell/2 + r} \leftrightarrow y^q x^r$, and it requires no ring operations. Define $g'(x, y) \in R[x, y]$ similarly by $g(x) = g'(x, x^{\ell/2})$. Note that $\deg_x f', \deg_x g' < \frac{\ell}{2}$ and $\deg_y f', \deg_y g' \le k < \ell/2$.

- Let $f''(x) \in D[x] = R[x, y]/(y^\ell + 1)$ be the image of $f'(x, y)$ in the quotient ring, and likewise for $g''(x)$. Because $\deg_y f', \deg_y g' \le k < \ell/2$, f' and g' are reduced representatives.

- Because $\deg_x f''(x), \deg_x g''(x) < \frac{\ell}{2} < \ell$ and $y \in D$ is a special 2ℓth root of unity, we may apply the FFT multiplication algorithm to compute $h''(x) = f''(x) \cdot g''(x)$ in $D[x]$. Operations in D are computed as described above, including recursive calls to the Schönhage-Strassen algorithm for the ℓ general multiplications in D.

- Let $h'(x, y) \in R[x, y]$ be the reduced representative of $h''(x)$.

- The output of the algorithm is $h'(x, x^{\ell/2}) \in R[x]$.

To see that this algorithm is correct, notice that $\deg_y(f'(x, y) \cdot g'(x, y)) \le 2k < \ell$, so $f'(x, y) \cdot g'(x, y) \in R[x, y]$ is a reduced representative of $h''(x) \in D[x]$, which means $f'(x, y) \cdot g'(x, y) = h'(x, y)$ by uniqueness. It follows that the output of the algorithm is

$$h'(x, x^{\ell/2}) = f'(x, x^{\ell/2}) \cdot g'(x, x^{\ell/2}) = f(x) \cdot g(x),$$

as necessary.

In the recursive calls to the Schönhage-Strassen algorithm, the new polynomial degrees are at most $\ell/2 = O(\sqrt{n})$, so there are $O(\log \log n)$ levels in the recursion. With careful analysis, this algorithm can be shown to use $O(n \log n \log \log n)$ operations in R. (Details omitted.)

5 A Web of Interconnected Problems

The general polynomial multiplication algorithm presented above relies on algorithms to solve special cases of the multipoint evaluation, interpolation, multiplication, and division problems. Moving forward, a general division-with-remainder algorithm can be constructed from the general multiplication algorithm (and a few other tricks, like Hensel Lifting); we may cover this in the next lecture. Similarly, general multipoint-evaluation follows from general multiplication and division algorithms, interpolation follows from these three algorithms, and so on.

As another concrete example, let's briefly see how multipoint evaluation follows from multiplication and division. For $f(x) \in R[x]$, the remainder when dividing $f(x)$ by $(x-\alpha)$ is exactly $f(\alpha)$. To evaluate f at all of $\alpha_1, \dots, \alpha_n$, we therefore need to evaluate the n remainders when dividing by $(x-\alpha_1), \dots, (x-\alpha_n)$. This can be done with a divide and conquer method:

- Compute $h_0(x) = \prod_{i=1}^{n/2}(x-\alpha_i)$ and $h_1(x) = \prod_{i=n/2+1}^{n}(x-\alpha_i)$ with an iterative divide-and-conquer application of the general multiplication algorithm.
- Divide $f(x)$ by h_0 and h_1, producing remainders $f_0(x)$ and $f_1(x)$.
- Recursively solve the multipoint evaluation problem for f_0 at $\alpha_1, \dots, \alpha_{n/2}$ and for f_1 at $\alpha_{n/2+1}, \dots, \alpha_n$.

This can be checked to run in $O(n \cdot \operatorname{poly}\log n)$ time.