6.S897 Algebra and Computation February 27, 2012

Lecture 6

Lecturer: Madhu Sudan *Scribe: Mohmammad Bavarian*

1 Overview

Last lecture we saw how to use FFT to multiply $f, g \in R[x]$ in nearly linear time. We also briefly talked about multipoint evaluation and interpolation for general points which we will expand on later in this lecture. But the main topic today is to go over another two fundamental algebraic algorithms, namely polynomial division and greatest common divisor problem. Both of these algorithms can be solved in nearly linear time which shouldn't surprise us now; as it has been already said a few times over the course of these lectures, most routines in computational algebra, if can be done, usually can be done in nearly linear time.

Today's lecture will be the last lecture we have on the basic/ fundamental algebraic algorithms that we will need for our future applications like factoring factoring polynomials in $R[x]$ or $R[x, y]$ and etc. However, despite the fact that these algorithms such as fast GCD and polynomial division and multiplication are very classical going back all the way to 50's and 60's, there has been a some very recent interesting research on them. Specifically those who are interested may want to look up the paper by Kiran Kedlaya and Chris Umans on modular composition which could of course, serve as a very nice final project.

2 Polynomial Division Algorithm

We have the familiar polynomial division problem. Since we are working in general ring R we need to assume the polynomial we are dividing on is monic to guarantee that we do not face the awkward position of trying to divide $x^2 - 1$ over $2x$ over $\mathbb{Z}[x]$.

Input: $f, g \in R[x]$. g monic.

Output: $q, r \in R[x]$ with $f = gq + r$ and $deg\ r < deg\ q$.

The long division algorithm we learned in school solves this problem in quadratic time. To get a faster algorithm we will use a version of Hensel lifting that allows us to compute the inverse of a polynomial f^{-1} modulo x^{2l} using its inverse modulo x^l. We should note that Hensel lifting in more general form will feature prominently in our future lectures.

But before getting into the details of the algorithm we need a definitions:

Definition 1 *Let $f = \sum_{i=0}^{n} c_n x^i$ be a of degree n in $R[x]$, $c_n \neq 0$. We define $Rev(f) = \sum_{i=0}^{n} c_{n-i} x^i$.*

Now notice that when f is monic $rev(f)$ has 1 as its constant coefficient and also we have:

$$Rev(f) = x^n f(\frac{1}{x}) \qquad \Rightarrow \qquad Rev(fg) = Rev(f)Rev(g)$$

Now using above we relation $Rev(f) = x^n f(\frac{1}{x})$ we see that

$$Rev(f) = Rev(gq + r) = Rev(g)\, Rev(q) + x^{n-deg(r)} Rev(r)$$

Since $deg(g) > deg(r)$ we see that

$$Rev(f) = Rev(q)Rev(g) \qquad mod\ (x^{deg(f)-deg(g)})$$

This modular equation was the main reason that we went over the trick of reverting the coefficient of our polynomials. Now finally we have

$$Rev(q) = Rev(g)^{-1} Rev(f) \qquad mod\ (x^{deg(f)-deg(g)})$$

So if the inverse $Rev(g)^{-1}$ exist and can be easily computed, we can use our nearly linear multiplication to compute $Rev(g)$ modulo $x^{deg(f)-deg(g)}$. Since this is exactly the degree of q, we immediately recover $Rev(q)$ and hence q from this procedure.

2.1 Inverses mod x^l

Since g was monic $Rev(g)$ has 1 as its constant coefficient. Given a polynomial $h(x)$ with unit constant coefficient we observe that we can progressively choose coefficient $a_1, a_2, \ldots, a_m \in R$ such that

$$(1 + a_1 x + a_2 x^2 + \ldots + a_{l-1} x^{l-1})\ h(x) = 1 \qquad mod\ (x^l)$$

Note that this is actually a well-known fact at least in case of fields that the units in the ring $\mathbb{F}[x]/(x^l)$ are exactly those polynomials coprime with x^l which is a consequence of small bezout's theorem $x^l p(x) + h(x)q(x) = 1$. Also the fact that this inverse is unique is pretty clear.

The basic progressive algorithm to compute the coefficient a_i's is not linear time and hence we need the idea of Hensel lifting to achieve a nearly linear time algorithm.

Input: $h^1\ mod\ x^l$.

Output: $h^{-1}\ mod\ x^{2l}$.

The base case of the algorithm is trivial as for $l = 1$ we have that $h^{-1} = 1$ modulo x. Now given a with $a.h = 1$ modulo x^l. Now we want to find $\tilde{a} = a + x^l b$ such that $\tilde{a}h = 0$ modulo x^{2l}. Writing $h = h_0 + h_1 x^l$ we see that $ah_0 = 1 + cx^l$ for some polynomial c as a result of our input condition. So,

$$(a + x^l b)(h_0 + x^l h_1) = ah_0 + x^l[bh_0 + h_1 a] = 1 + x^l[c + bh_0 + h_1 a] \quad mod\ x^{2l}$$

Multiplying the term $bh_0 + c + h_1 a$ by a modulo x^l we see that it suffices to take

$$b = -h_0^{-1}(h_1 a + c) = -a(h_1 a + c) \quad mod\ x^l$$

So each step requires about two polynomial multiplication of degree l polynomials. Since we double up l everytime, our time complexity is a geometric sum which sums up to constant multiple of the last operation which is $O(m(n))$ where here $m(n)$ stands for complexity of multiplication of polynomials which is nearly linear time.

3 Multipoint Evaluation and Interpolation

3.1 Multipoint Evaluation

Multipoint evaluation and its inverse, interpolation, are fundamental algorithms used in various other algebraic algorithms as was already seen in the special case Fourier transform and inverse transform in polynomial multiplication. Here we are interested interpolation and mutipoint evaluation over generic points of R as opposed to special points which were used in FFT. Here is multipoint evaluation problem,

Input: $f = \sum_{i=0}^{n} c_i x^i \quad \alpha_1, \alpha_2, \ldots, \alpha_n \in R$.

Output: $f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_n)$.

We will solve this problem in $O(m(n)\log n)$ by a divide and conquer algorithm which in each iteration splits the problem into two multipoint evaluation problem of size half. The point is that what we want to compute is $f \ mod (x-\alpha_i)$ for every α_i. Instead of f we could instead work with $f mod \prod_{i=1}^{n}(x_{\alpha_i})$ as this preserves the f modulo $(x-\alpha_i)$. The way we split this problem in half is to notice that if we compute

$$f_1 = f \mod \prod_{i=1}^{\frac{n}{2}}(x-\alpha_i) \quad and \quad f_2 = f \mod \prod_{i=\frac{n}{2}+1}^{n}(x-\alpha_i)$$

Now if we call the mutipoint evaluation for f_1 with $(\alpha_1, \alpha_2, \ldots, \alpha_{\frac{n}{2}})$ and f_2 with $(\alpha_{\frac{n}{2}+1}, \ldots, \alpha_n)$ we will get our solution to our original problem of multipoint evaluation of f over $(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

Now notice that since f_1 and f_2 have degree less than $\frac{n}{2}$ this problem is indeed half the size of the original problem. So by repeatedly calling two instances of half the size and using polynomial division algorithm procedure to compute f_1 and f_2 by taking modular over $\prod_{i=1}^{\frac{n}{2}}(x-\alpha_i)$ and $\prod_{i=\frac{n}{2}+1}^{n}(x-\alpha_i)$ we will get our solution. However we are not quite done yet !

Notice that we have implicitly assumed all the polys of the form $\prod_{i=\frac{n}{2}+1}^{n}(x-\alpha_i)$ and their smaller counterpart which will be wanted in deeper recursions are known which in fact is not true and must be computed. The point is that we only discussed going down over the tree of our recursion, we have to actually go up as well: Starting from pairs $(x-\alpha_{2k-1})$ and $(x-\alpha_{2k})$ we can compute $(x-\alpha_{2k-1})(x-\alpha_{2k})$. Now using the product say $(x-\alpha_{2k-1})(x-\alpha_{2k})$ and $(x-\alpha_{2k+1})(x-\alpha_{2k+2})$, we can compute $(x-\alpha_{2k-1})(x-\alpha_{2k})(x-\alpha_{2k+1})(x-\alpha_{2k+2})$. As such we must go up over the $\log n$ layers of our recursion and compute the polynomials we need. Luckily we have our fast polynomial multiplication which gives near linear time computation of polynomial products in each layer and hence with $O(logn)$ overhead coming from the number of layers, we can go up the tree in $O(m(n)\log n)$ as promised. Since polynomial division also takes $O(m(n))$ in each layer of recursion, we get the total time of $O(m(n)\log n)$.

3.2 Interpolation

For interpolation we will assume $R = \mathbb{F}$ is a field because we need division over some elements which in general might not have inverse in R. This is not surprising as for general R interpolation might not have a solution. For example, you cannot find linear function in $\mathbb{Z}[x]$ that takes value zero at zero and value 1 at say $x = 2$. So here is our problem.

Input: $\alpha_1, \alpha_2, \ldots, \alpha_n, \quad \beta_1, \beta_2, \ldots, \beta_n \in R$.

Output: $c_0, c_1, \ldots, c_{n-1}$ such that $f(x) = \sum_{i=0}^{n-1} c_i x^i$ and $f(\alpha_j) = \beta_j$.

Now let's first understand why the non-algorithmic version of this problem makes sense. It is easy to see by virtue of polynomial division that there is at most one polynomial satisfying the condition of the output. Now consider the polynomials of the form $h_j = \frac{\prod_{i=1}^{n}(x-\alpha_i)}{(x-\alpha_j)}$. Note that $h_j(\alpha_i) = 0$ for all $i \neq j$ and $h_j(\alpha_j) \neq 0$. So we can easily see that we can write $f(x)$ as linear combination of h_j's. However above procedure again is quadratic and we seek a nearly linear time algorithm.

The idea is very similar to the case of fast multipoint evaluation. We again split the problem into two problems of half the size as follows: As in in the case of multipoint evaluation, by going up the tree of recursion, we can assume we know the polynomials

$$Z_1 = \prod_{i=1}^{\frac{n}{2}}(x-\alpha_i) \qquad Z_2 = \prod_{i=\frac{n}{2}+1}^{n}(x-\alpha_i)$$

We want to find f_1 and f_2 such that

$$f = f_2 Z_1 + f_1 Z_2 \qquad deg(f_1), deg(f_2) < \frac{n}{2}$$

Looking at above equation modulo $(x - \alpha_i)$ we see that $f_1(\alpha_i) = \frac{\beta_i}{Z_2(\alpha_i)}$ for all $1 \leq i \leq \frac{n}{2}$ and $f_2(\alpha_i) = \frac{\beta_i}{Z_1(\alpha_i)}$ for all $\frac{n}{2} + 1 \leq i \leq n$. So we can solve for f_1 and f_2 using interpolation of half the size. The values of $\{Z_1(\alpha_i)\}$'s and $\{Z_2(\alpha_i)\}$'s can be computed using multipoint evaluation. Looking at the recursion cost we see that the total cost is again nearly linear time with respect to n.

4 Greatest Common Divisor Problem

It is indeed a surprising fact computational aspects of algebra that while factoring seems like such a hard problem, finding common factors, i.e. the GCD problem has such simple efficient algorithm.

Given a ring R recall the following definitions :

- Units : elements such as u which have (multiplicative) inverses.
- Division: a divides b denoted by $a|b$ if $a = bk$ for some $k \in R$.
- Irreducible: a is irreducible if it is not a unit and $a = bc$ implies b or c is a unit.
- Associates: $a \sim b$ if a and b are equal up to multiplication by units, i.e. $a = bu$ and $u^{-1} \in R$.
- Unique Factorization Domain: If every $x \in R$ can be written as (finite !) product of irreducibles and if $a_1 a_2 \dots a_l = b_1 b_2 \dots b_k$ where a_i's and b_i's are irreducibles we have that $k = l$ and there exist a permutation π such that $a_i \sim b_{\pi_i}$.

Definition 2 *The greatest common divisor of two element $a, b \in R$ is $gcd(a, b) = g$ if g is a common divisor, i.e. $g|a$ and $g|b$, and for any other common divisor, $h|a$ and $h|b$ we have $h|g$.*

For the GCD problem we shall assume our ring is $R = \mathbb{F}[x]$ as these rings can be shown to be unique factorization domains. Note that the greatest common divisor is guaranteed to exist for UFD's by looking at the irreducible factorization of a and b and taking the largest intersection between the associate irreducible factors (taking account of associativity).

Definition 3 *An ideal $I \subset R$ is a set closed under addition and multiplication.*

$$a, b \in I \quad \longrightarrow \quad \lambda a + \lambda' b \in I \quad \forall \lambda, \lambda' \in R$$

Let $I(a, b) = \{\lambda a + \lambda' b \,|\, \lambda, \lambda' \in R\}$ denote the smallest ideal congaing a and b. As always in this section we assume $R = \mathbb{F}[x]$. This gives to our ring the notion of the degree. We have the following proposition relating the $gcd(a, b)$ to the notion of ideal above.

Proposition 4 *The polynomial of smallest degree in $I(a, b)$ is $gcd(a, b)$.*

Proof Let g be the smallest degree polynomial in $I(a, b)$. g must divide a and also b because any remainder of the division would be a polynomial of smaller degree in $I(a, b)$. Given any $h|a$ and $h|b$ we have $h|\lambda a + \lambda' b$ which for a particular choice of λ and λ' becomes g which gives us $h|g$ for any common divisor h. ■

The basic algorithm for computing $gcd(a, b)$ is exactly the one we learned as euclid's algorithm in school. Recall that this algorithm is as follows:

1. Given (a, b) if $b|a$ return b.

2. Return $GCD(b, a \ mod \ b)$

Since degree of polynomials can decrease only one at a time and everytime we need a at least linear time polynomial division to calculate the remainder, this will take $\tilde{O}(n^2)$.

However, there exist indeed a fast, i.e. near linear time, algorithm for $GCD(a, b)$ due to Schönhage. The algorithm is based the *Half-GCD* subroutine.

To appreciate this we have to take a closer look at the euclidean division algorithm. In each iteration euclidean division algorithm given (a, b) returns to us two elements $(b, a - qb)$. So from one generator of the ideal $I(a, b)$ we go to a different generator of the same ideal, i.e. $I(b, a - qb) = I(a, b)$. So we ask ourselves more generally for which pairs $c, d \in I(a, b)$ we in fact have $I(a, b) = I(c, d)$. Well we know that (c, d) are linear combinations of (a, b) so,

$$\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} c \\ d \end{pmatrix}$$

Given that (c, d) are linear linear combination of (a, b) for $I(a, b) = I(c, d)$ we must have a, b are also linear combination of (c, d) which exactly means that the above 2×2 matrix has an inverse. So we can more generally look at any sequence of multiplication by invertible matrices in in $M_2(R)$ as a more general way to compute $GCD(a, b)$. Schematically, the euclidean algorithm gives us a chain of maximum n matrix $m_i \in M_2(R)$ which at the end ends up with a vector that has one component zero.

$$\begin{pmatrix} a_0 \\ b_0 \end{pmatrix} \rightarrow^{m_1} \begin{pmatrix} a_1 \\ b_1 \end{pmatrix} \rightarrow^{m_2} \cdots \begin{pmatrix} Gcd(a, b) \\ 0 \end{pmatrix}$$

Now to understand *Half-GCD* subroutine look at the progression of euclidean algorithm over the generators (a_i, b_i)'s. There exist some i at which $deg(a_i) > \frac{deg(a)}{2}$ and $deg(b_i) \leq \frac{\deg(a)}{2}$. Now we had a *subroutine* that given (a, b) found (c, d) with above condition, since this subroutine each time halved the degree, it would have been very helpful because by iterating that subroutine we could solve the problem in constant times the time it takes to solves that subroutine for size n. Now that subroutine is called *Half-GCD*.

The critical idea for computing the Half-GCD efficiently comes from the following lemma:

Lemma 5 *Let polynomials $a = a_0 + x^k a_1$ and $b = b_0 + x^k b_1$ be given where $k = \frac{deg(a)}{2}$. Let $N_1, N_2, N_3, \ldots$ be the sequence of matrices we get from applying euclidean division on the pair (a, b). Analogously let $L_0, L_1, L_2, \ldots$ be the sequence of matrices we get from applying euclidean division for (a_1, b_1). Let i be an index small enough such that $(c_1, d_1)^T = L_i(a_1, b_1)^T$ and $deg(c_1) > \frac{deg(a_1)}{2}$. Then $L_i = N_i$.*

What is this lemma saying at an intuitive level? the lemma is indeed an statement about the inner-working of euclidean algorithm. It basically says that if instead of looking at the pair (a, b) you look at the pair (a_1, b_1) which have the similar structure in high order terms, the sequence of matrices generated by euclidean algorithm is exactly the same as the one from applying euclidean algorithm to (a, b) itself. Here the notion of smallness of number of iterations is quantified by the condition that i is such that $(c_1, d_1)^T = L_i(a_1, b_1)^T$ satisfy $deg(c_1) > \frac{deg(a_1)}{2}$.

Now given this lemma it is easy to imagine how the fast algorithm for computing GCD would go because for computing the half-GCD of (a, b), it is enough to compute the half-GCD matrix associated with (a_1, b_1) which halves the size of the problem and then we can iterate on this. (Note that we need the whole matrix not just (c_1, d_1))

A lot of the details for this part of lecture was omitted and we advise the interested reader to contact Prof. Sudan himself for further understanding this algorithm and its related issues.

5 References

1. K. Kedlaya and C. Umans. Fast modular composition in any characteristic. Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS). pages 146-155. 2008