**6.S897 Algebra and Computation** Feb 23, 2015

# Lecture 6

*Lecturer: Madhu Sudan* *Scribe: Prashant Vasudevan*

**Outline:**

- Nearly linear-time algorithms
  - Division
  - GCD
- Factorisation
  - Over finite fields.

**From last week:** For the running time of the algorithm for multiplying polynomials from last class, we had the following recurrence:

$$T(n) = c\sqrt{n} + T(\sqrt{n}) + O(n \log n)$$

It turns out that the precise value of $c$ does matter in determining the asymptotic behaviour of $T$. If $c = 1$, we get $T(n) = O(n \log n)$, and if $c$ is something larger, in general we get $T(n) = O(n \log^{O(1)} n)$. In our case it turns out that $c = 2$, and $T(n) = O(n \log n \log \log n)$.

The fact that the above problem of multiplying polynomials has a nearly linear-time algorihm is not as surprising as it might seem at first glance, as one way one might do this is by abandoning finite fields and multiplying the polynomials as though they were over $\mathbb{C}$, where the requisite roots of unity are present to multiply using Fourier transforms. The catch is that these roots may not have short descriptions, but this may be overcome by taking only $O(\log n)$ bits of precision, which should still give us the answer in $O(n \log^{O(1)} n)$ time.

That the problems we shall now study also have nearly linear-time algorithms, though, should actually be surprising. But also perhaps not, as we do not really know a natural problem that has inherent super-linear ($\omega(n \log^{O(1)} n)$) complexity.

## 1 Division

The problem of univariate polynomial division over a field $\mathbb{F}$ is as follows:

- Input: $f, g \in \mathbb{F}[x]$
- Output: $q, r \in \mathbb{F}[x]$ such that $f = q.g + r$, and $deg(r) < deg(g)$

As it turns out, the complexity of division is $\Theta$(complexity of multiplication). This is as a result of the reduction of division to the problem of inverting polynomials, which is as follows:

- Input: $g \in \mathbb{F}[x]$ such that $g(0) \neq 0$ and $t \in \mathbb{Z}^+$.
- OutputL $a \in \mathbb{F}[x]$ such that $a.g = 1 (mod\ x^t)$.

## 1.1 Complexity of Inversion

Note that without loss of generality, we may assume $deg(g) \leq t$.

Modulo $x^t$, no multiple of $x$ is invertible, but any other polynomial is (this is an implication of Bezout's theorem, which we shall see in a later class), which is why $g(0) \neq 0$ is part of the input promise.

We shall show now how to solve the above problem in nearly-linear time, making use of the algorithm for multiplication that we know. We shall use the idea of Hensel lifting, which is a general technique useful in several cases to solve a problem modulo $x^{2t}$ given the ability to solve it modulo $x^t$.

Suppose we have obtained an $a$ such that $a.g = 1(mod\ x^t) \implies a.g = 1 + x^t b$. We want to lift this to get an $\tilde{a}$ such that $\tilde{a}.g = 1(mod\ x^{2t})$.

First, as we only care about $a.g$ modulo $x^t$ to start with, we don't care about terms of $a$ that have degree greater than $x^t$. We shall set these higher degree terms in an appropriate manner to get $\tilde{a}$.

Let $\tilde{a} = a + x^t a_1$. Then, $\tilde{a}g = ag + x^t a_1 g = 1 + x^t(b + a_1 g)$. What we want now is for $(b + a_1 g)$ to be divisible by $x^t$, that is, $b + a_1 g = 0(mod\ x^t)$, which is the same as $a_1 = -bg^{-1}(mod\ x^t)$. But $g^{-1}$ modulo $x^t$ is $a$, and we get that $a_1 = -ba$ will do.

Substituting, we can check that $\tilde{a}g = a(1 - bx^t)g = (1 + bx^t)(1 - bx^t) = 1 - b^2x^{2t}$.

This way, from a solution to the inversion problem modulo $x^t$, we have obtained the inverse of a polynomial module $x^{2t}$ with just a constant number (2, in fact) of extra polynomial multiplications. This gives us a nearly-linear time algorithm for inversion (which may be seen by writing down the recursion for the running time as in the case of multiplication).

Note that Hensel lifting may be done not just from $x^t$ to $x^{2t}$, but from any $p$ to $p^2$, and in general, even modulo ideals.

## 1.2 Reduction to Inversion

Next we describe how to reduce division to inversion, which is a bit of a hack, rather.

Given $f = \sum_i a_i x^i$, define $Rev_n(f) = \sum_i a_i x^{n-i}$, when $n \geq deg(f)$.

If $f = qg + r$, it may be verified that $Rev_n(f) = Rev_{deg(g)}(g)Rev_{n-deg(g)}(q) + Rev_n(r)$.

As $r$ has degree at most $deg(g)$, $Rev_n(r)$ is divisible by $x^{n-deg(g)}$. So if we work modulo $x^{n-deg(g)}$, we can find $Rev_{n-deg(g)}(q)$ modulo $x^{n-deg(g)}$ by inverting $R_{deg(g)}(g)$ modulo $x^{n-deg(g)}$ and using $Rev_{n-deg(g)}(q)Rev_{deg(g)}(g) = Rev_n(f)(mod\ x^{n-deg(g)})$. This is enough to retrieve $q$ because $deg(q) = n - deg(g)$, and everything just works out.

## 2 GCD

It is somewhat remarkable that Euclid's algorithm from all that long ago continues to work to compute GCD of polynomials in polynomial time. This algorithm, $GCD(f, g)$ (where $deg(f) > deg(g)$) is as follows:

- If $g = 0$, output $f$.
- Else, run $GCD(g, f\ mod\ g)$.

It is clear that this algorithm makes at most as many iterations as the degree of either of its arguments. But it is still far from linear-time, as it is unclear how one would compute $f\ mod\ g$ fast enough. To remedy this, note that each iteration of the algorithm is something like this:

$$\begin{pmatrix} g \\ f\ mod\ g \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix}$$

And the whole algorithm looks something like this:

$$\begin{pmatrix} GCD(f,g) \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_m \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix}$$

Note that there is a point in the algorithm such that till that point the degree of at least one arguments to $GCD$ was more than $n/2$, and after this point both degrees are less than $n/2$. The idea, roughly, is that till that point, we find the $q_i$'s by ignoring terms of $f$ and $g$ of degree less than $n/2$, and at that point put the terms back appropriately and resume. This breaks the computation of $GCD$ of degree $n$ polynomials down into two computations of $GCD$ of degree $n/2$ polynomials, and some more work.

Can we have pathological cases where the original algorithm is much worse than the improved algorithm? Yes. In fact, it would seem that most cases would be so.

## 3 Factorisation

"Something of a miracle."

Without loss of generality, we work always with monic polynomials, as we are working over finite fields. We start with factorisation of quadratic polynomials.

Given $x^2 + ax + b \in \mathbb{F}[x]$, how do we even know whether it can be factorised? One might do something like check if the $(a^2 - 4b)$ is a quadratic residue, but we want something more general.

We take $(x^q - x)$, which has exactly all linear terms as factors, and compute $GCD(x^2 + ax + b, x^q - x)$. Before seeing what this gives us, note that doing so normally would require time linear in $q$, and this is too much for us. But we can actually compute $f(mod\ g)$ in time $poly(deg(g), \log deg(f), sparsity(f))$, by looking at each monomial $c_i x^i$, and using $x\ mod\ g, x^2\ mod\ g, \ldots$ to find $c_i x^i (mod\ g)$. This is very good for us as $x^q - x$ is very sparse, and using this to compute the $GCD$ algorithm lets us do so fast enough.

So if $deg(f) = 2$, and $GCD(f, x^q - x)$ is of degree 0, then $f$ is irreducible, and if it is of degree 2, then $f$ is reducible. If the degree is 1, then $f$ has repeated roots.

In general, to deal with repeated roots in polynomials, we can take the $GCD$ of the polynomial with its derivative, which separates the repeated part which we can deal with separately. While it is easy to see that this works over $\mathbb{R}$, it also works over finite fields, for somewhat non-trivial reasons, as sometime the derivative of a polynomial of degree $n$ has degree less than $n - 1$, for example, $f(x) = x^p$. What we can say is the following:

**Lemma 1 ()** *If $f' \neq 0$, $GCD(f, f') = 1$ iff $f$ has no repeated factors.*

If $f' = 0$, then $f$ (in a field of characteristic $p$) is of the form $f = \sum_i c_i x^{ip}$, which can be written as $(\sum_i c_i^{1/p} x^i)^p$. In $\mathbb{F}_q$ of characteristic $p$, $c^{1/p} = c^{q/p}$, and $q/p$ is an integer. Now we just factorise $\sum_i c_i^{1/p} x^i$.

Now that we have dealt with repeated factors, we have a reducible quadratic polynomial whose factors are distinct, and $GCD(x^q - x, f) = f$. How do we factorise this?

Suppose $x^2+ax+b = (x-\alpha)(x-\beta)$, and it happened to be the case that $\alpha$ is a quadratic residue while $\beta$ is not. Then, as we know that all quadratic residues are roots of $(x^{(q-1)/2}-1)$, we have $GCD((x^{(q-1)/2}-1), f) = (x - \alpha)$, and we can factorise $f$. (Note that $x^q - x = x(x^{(q-1)/2} - 1)(x^{(q-1)/2} + 1)$.

If we are not so fortunate as to have this happen to be the case with our polynomial, we are going to come up with a randomised polynomial such that knowledge of its factorisation enables us to factorise our polynomial, and it is likely to be of this form.

Given $x^2 + ax + b$ with roots $\alpha, \beta$, the quadratic polynomial with roots $(c\alpha + d), (c\beta + d)$ is $\left(\frac{x-d}{c}\right)^2 + a\left(\frac{x-d}{c}\right) + b$. If we choose $c$ and $d$ at random, with good probability we end up with a polynomial that fits the case above.

This extends to factorising polynomials of higher degrees with all linear factors, by repeating above process till we find all of them. This is a special case of the algorithm due to Berlekamp from 1972, before notions like NP were even defined, and the theory of randomised algorithms wasn't as developed as it is today.

## 3.1 Even Characteristic

Note that all we did above was only for fields of odd characteristic, as quadratic residuosity only makes sense then. What we were essentially doing earlier was looking at the multiplicative group of the field and using the fact that it has a non-trivial subgroup and looking at it and its coset (namely, roots of $(x^{(q-1)/2} - 1)$ and $(x^{(q-1)/2} + 1)$). Now we are going to try to do that with the additive group.

Given field $\mathbb{F}_q$ of characteristic 2, we define $Tr : \mathbb{F}_q \to \mathbb{F}_2$ as $Tr(x) = x + x^2 + \cdots + x^{2^{s-1}}$, where $q = 2^s$, and use the fact that $x^q - x = Tr(x)(Tr(x) - 1)$. Now we use this identity to do with the additive group what we did with the multiplicative group earlier. Note that we can do this for any constant characteristic field.

## 3.2 Higher Degree Factors

But what about polynomials that have non-linear factors? Well, to begin with, we can still get all the linear factors by the above method.

For the rest, we use the following identity: $x^{q^d} - x = \prod g(x)$ over all $g$ that are monic, irreducible and $deg(g)$ divides $d$. So to find product of factors of degree $d$ of $f$ if $f$ has no factors of degree smaller than $d$, we just take $GCD(x^{q^d}, f)$. So, for a square-free $f$, we have the sequence $(f_1 = GCD(x^q - x, f), f_2 = GCD(x^{q^2} - x, f/f_1), \ldots)$, which are factors of $f$ such that all of their factors have the same degree. So if we can factorise all of the $f_i$'s, we will be done.

This we leave for the next lecture, where we shall also talk about making some of these algorithms deterministic.