Evaluation/Interpolation (II)

Lecture 8

Backtrack from the GCD ideas a bit

- We can do some operations faster in a simpler domain, even if we need to do them repeatedly

$$\begin{tikzcd}
GCD(F(x,y,z),G(x,y,z)) \arrow[rr] \arrow[d, shift left=1] \arrow[d, shift right=1] & & H(x,y,z) \\
GCD(F_p(x,y,z),G_p(x,y,z)) \arrow[d, shift left=1] \arrow[d, shift right=1] \arrow[d, shift left=3] \arrow[d, shift right=3] & & H_p(x,y,z) \arrow[u, shift left=1] \arrow[u, shift right=1] \\
GCD(F_p(x,0,0),G_p(x,0,0)) \arrow[rr, shift left=1] \arrow[rr, shift right=1] & & H_p(x,0,0) \arrow[u, shift left=1] \arrow[u, shift right=1] \arrow[u, shift left=3] \arrow[u, shift right=3]
\end{tikzcd}$$

Backtrack from the GCD ideas a bit

- Some of the details

$$
\begin{array}{ccc}
GCD(F(x,y,z),G(x,y,z)) & \longrightarrow & H(x,y,z) \\
\Downarrow \text{ reduce mod } p_1, p_2, \ldots & & \Uparrow \text{ CRA } p_1, p_2, \ldots \\
GCD(F_p(x,y,z),G_p(x,y,z)) & & H_p(x,y,z) \\
\Downarrow \text{ evaluate at } z=0,1,2,\ldots \text{ evaluate at } y=0,1,2,\ldots & & \Uparrow \text{ interpolate OR sparse interpolate} \\
GCD(F_p(x,0,0),G_p(x,0,0)) & \Longrightarrow & H_p(x,0,0)
\end{array}
$$

compute (many) GCDs

How does this work in some more detail

- How many primes p_1, p_2, ...?
  - Bound the coeffs by p_1*p_2 *...p_n ? (or +/- half that)
    - bad idea, the bounds are poor. *given* ||f|| what is ||h|| where h is factor of f ? What is a bad example? A pyramid X a difference operation? Are there worse?
    - $(x-1)*(1+2x+3x^2+....+nx^{n-1}+ (n-1)x^n+..+2x^{2n-1}+x^{2n})$ is
    - $-1-x-x^2-x^3...-x^n+x^{n+1} + ... +x^{2n+1} ...$
    - Some bound.... $||h|| \cdot (d+1)^{1/2}2^d \max(||f||)$
    - Don't bound but try an answer and test to see if it divides?
    - See if WHAT divides?
      - compute cofactors A, B, A*H=F, B*H=G, and when A*H= F or B*H=G, you are done.

How does this work in some more detail

- The process doesn't recover the leading coefficient since F modulo p etc might as well be monic.
- The inputs F and G are assumed primitive; restore the contents.
- There may be unlucky primes or evaluation points.

Chinese Remainder Theorem

- (Integer) Chinese Remainder Theorem:
- We can represent a number x by its remainders modulo some collection
- of relatively prime integers n_1 n_2 n_3...

- Let N = n_0*n_1* ...*n_k. Then the Chinese Remainder Thm. tells us that we can represent any number x in the range -(N-1)/2 to +(N-1)/2 by its residues modulo n_0, n_1, n_2, ..., n_k. {or some other similar sized range, 0 to N-1 would do}

Chinese Remainder Example

Example $n_0=3$, $n_2=5$, $N=3*5=15$

x	x mod 3	x mod 5
-7	-1	-2 note: if you hate balanced notation -7+15=8. mod 3 is 2->-1
-6	0	-1
-5	1	0
-4	-1	1
-3	0	2
-2	1	-2 note: x mod 5 agrees with x, for small x 2 [-2,2], +-(n-1)/2
-1	-1	-1 note:
0	0	0 note:
1	1	1 note:
2	-1	2
3	0	-2
4	1	-1
5	-1	0 note: symmetry with -5
6	0	1
7	1	2 note: also 22, 37, ... and -8, ...

Conversion to modular CRA form

Converting from normal notation to modular representation is easy in principle;

you do remainder computations (one instruction if x is small, otherwise a software routine simulating long division)

Conversion to Normal Form (Garner's Alg.)

Converting to normal rep. takes k^2 steps.

Beforehand, compute

inverse of n_1 mod n_0, inverse of n_2 mod n_0*n_1, and also the products n_0*n_1, etc.

Aside: how to compute these inverses:

These can be done by using the Extended Euclidean Algorithm.

Given r= n_0, s= n_1, or any 2 relatively prime numbers, EEA computes a, b such that a*r+b*s=1 = gcd(r,s)

Look at this equation mod s: b*s is 0 (s is 0 mod s) and so we have a solution a*r=1 and hence a = inverse of r mod s. That's what we want. It's not too expensive since in practice we can precompute all we need, and computing these is modest anyway. (Proof of this has occupied quite a few people..)

Conversion to Normal Form (Garner's Alg.)

Here is Garner's algorithm:

Input: x as a list of residues $\{u_i\}$: $u_0 = x \bmod n_0$, $u_1 = x \bmod n_1$, ...

Output: x as an integer in [-(N-1)/2,(N-1)/2]. (Other possibilities include x in another range, also x as a rational fraction!)

Consider the mixed radix representation

$$x = v_0 + v_1 * n_0 + v_2 * (n_0 * n_1) + \ldots + v_k * (n_0 * \ldots n_{k-1}) \qquad [G]$$

if we find the v_i, we are done, after k more mults. These products are small X bignum, so the cost is more like $k^2/2$.

Conversion to Normal Form (Garner's Alg.)

$x = v_0 + v_1 * n_0 + v_2 * (n_0 * n_1) + \ldots + v_k * (n_0 * \ldots n_{k-1})$ [G]

It should be clear that

$v_0 = u_0$, each being x mod n_0

Next, computing remainder mod n_1 of each side of [G]

$u_1 = v_0 + v_1 * n_0$ mod n_1, with everything else dropping out

so $v_1 = (u_1 - v_0) * n_0^{-1}$ all mod n_1

in general,

$v_k = (u_k - [v_0 + v_1 * n_0 + \ldots + v_{k-1} * (n_0 \ldots n_{k-2})]) * (n_0 * \ldots n_{k-1})^{-1}$ mod n_k. mod n_k

Conversion to Normal Form (Garner's Alg.)

Note that all the v_k are "small" numbers and the items in green are precomputed.

Cost: if we find all these values in sequence, we have k^2 multiplication operations, where k is the number of primes needed. In practice we pick the k largest single-precision primes, and so k is about 2* log (SomeBound/2^{31})

Interpolation

- Abstractly THE SAME AS CRA
  - change primes p_1, p_2, ... to $(x-x_1)$ ...
  - change residues $u_1 = x \bmod p_1$ for some integer x to $y_k=F(x_k)$ for a polynomial F in one variable

  - Two ways it is usually presented, sometimes more (solving linear vandermonde system...)

polynomial interpolation (Lagrange)

- The problem: find the (unique) polynomial $f(x)$ of degree $k-1$ given a set of evaluation points $\{x_i\}_{i=1,k}$ and a set of values $\{y_i=f(x_i)\}$

Solution: for each $i=1,\ldots,k$

find a polynomial $p_i(x)$ that takes on the value y_i at x_i, and is zero for all other abscissae..

$x_1, \ldots, x_{i-1}, .. x_{i+1}, .. x_k$

That's easy here's one solution:

$p_i(x)=y_i + (x-x_1)(x-x_2)¢ \ldots ¢(x-x_{i-1})(x-x_{i+1})\ldots$

polynomial interpolation

- $p_i(x)=y_i +(x-x_1)(x-x_2)¢ \ldots ¢(x-x_{i-1})(x-x_{i+1})\ldots$
- Here's another one, with the additional desirable property that $p_i(x)$ is itself zero at the OTHER points.
- $p_i(x)= y_i* (x-x_1)(x-x_2)¢ \ldots ¢(x-x_{i-1})(x-x_{i+1})\ldots /(x_i-x_1)(x_i-x_2)¢ \ldots ¢(x_i-x_{i-1})(x_i-x_{i+1})\ldots$
- i.e. $p_i(x)= y_i \prod_{i\neq j} (x-x_j) /\prod_{i\neq j} (x_i-x_j)$
- Now to get a polynomial (of appropriate degree), just add them. $\sum_i p_i(x)$ agrees with all the specified points.

Another, incremental, approach (Newton Interpolation, CRA like)

- let $f(x)$ be determined by (x_i,y_i) for $i=1,\ldots,k$. We claim that there are constants such that $f(x)=\lambda_0+\lambda_1(x-x_1)+\lambda_2(x-x_1)(x-x_2)+\ldots$
- separating out the n-level approximation
- … $f^{(n)}+\lambda_n(x-x_1)¢…¢(x-x_n)+\ldots$
- Find the λ s. By inspection, $\lambda_0 = f(x_1)=y_1$. In $f(x_2)$, all terms but the first two are zero, so $y_2=f(x_2)=\lambda_0+\lambda_1(x_2-x_1)$ so $\lambda_1=(y_2-\lambda_0)/(x_2-x_1)$ and
- $f^{(2)} = y_1+ (y_2-\lambda_0)/(x_2-x_1)*x$

An incremental approach (Newton Interpolation)

- … $f^{(n+1)} = f^{(n)} + \lambda_n (x-x_1)$¢…¢$(x-x_n)$+ …
- In $f(x_{n+1})$, all terms but the first two are zero, so $y_{n+1} = f^{(n)} + \lambda_n (x_{n+1}-x_1)$ ¢ …¢ $(x_{n+1}-x_n)$ so $\lambda_n = (y_{n+1} - f^{(n)}) / (x_{n+1}-x_1)$¢ … ¢ $(x_{n+1}-x_n)$

- That is, given an n-point fit $f^{(n)}$, and one more (different) point, we can find λ_n and get an n+1 point fit $f^{(n+1)}$ at a cost of n+1 adds, n multiplies, and a divide.

polynomial interpolation

- Additional notes. The Lagrange and Newton forms of interpolation are effectively the same, with an identical computation with different order of operations:
- Note that y_i can, without loss of generality, be a polynomial in other variables.

Sparse Multivariate Interpolation

- There is a better way to do interpolation if you think that the number of terms in the answer is much lower than the degree, and you have several variables. (R. Zippel)

Sparse Multivariate Interpolation: Basic Idea / example

- Assume you have a way of evaluating (say as a black box) a function F(x,y,z) that is supposed to be a polynomial in the 3 variables x,y,z. For simplicity assume that D bounds the degree of F in x, y, z separately. Thus if D=5, there could be (5+1)^3 or 216 different coefficients. The black box is, in our current context, a machine to compute a GCD.

First stage: Find a "skeleton" in variable x

- evaluate F(0,0,0), F(1,0,0), F(2,0,0)... or any other set of values for x, keeping y and z constant. 6 values.
- one useful choice turns out to be $2^0, 2, 2^2, 2^3$,... or other powers.
- This gives us a representation for F(x,0,0)= $c_5x^5+c_4x^4+c_3x^3+c_2x^2+c_1x+ c_0$.
- If we were doing regular interpolation we would find each of the c_i in $(5+1)^2$ more evaluations.

(in 6 more evals at say F(0,1,0), F(1,1,0)... we get (poly in x)*y+(poly in x);
in another 6 we get (poly in x)*y^2+(poly in x)*y + (poly in x) etc. We still have to do the z coeffs.)

- If F is sparse, most of the c_i we find will be zero.
- For arguments' sake, say we have $c_5x^5+c_1x+c_0$, with other coefficients zero. And the heuristic hypothesis is that these are the ONLY powers of x that ever appear.

What if the first stage is wrong?

- Say c_4 is not zero, but is zero only for the chosen values of y=0,z=0. We find this out eventually, and choose other values, say y=1, z=-1.
- Or we could try to find 2 skeletons and see if they agree. (How likely is this to save work?)If F is sparse, most of the c_i we find will be zero.

Second stage

Recall we claim the answer is $c_5x^5+c_1x+c_0$, with other coefficients zero.

Let's construct the coefficients as polynomials in y. There are only 3 of them. Consider evaluating, varying y:

F(0,0,0), F(0,1,0), F(0,2,0), ..., F(0,5,0)

F(1,0,0), F(1,1,0), F(1,2,0), ..., F(1,5,0)

F(2,0,0), F(2,1,0), F(2,2,0), ..., F(2,5,0)

total of 18 evaluations (only 15 new ones) to create enough information to construct, say

$c_5= d_5y^5+d_4y^4+d_3y^3+d_2y^2+d_1y+d_0$.

Second stage assumption of sparseness

Let us assume that c_5, c_1 and c_0 are also sparse, and for example

$c_5 = d_1$

$c_1 = e_4 y^4 + e_1 y$

$c_0 = c_0$

We also assume that these skeletons are accurate, and that the shape of the correct polynomial is

$F(x,y,z) = f_{51}(z)\,x^5 y + f_{14}(z)\,x y^4 + f_{11}(z)\,x y + f_{05}(z)\,y$

Third stage

$F=f_{51}x^5y+f_{14}xy^4+f_{11}xy+f_{05}y$

There are 4 unknowns. For 4 given value pairs of x,y we can generate 4 equations, and solve this set of linear equations in time n^3. (n=4, here) Actually we can, by choosing specific "given values" solve the system in time n^2. (a Vandermonde system). Sample: pick a number r; choose (x=1,y=1), (x=r,y=r),($x=r^2$,$y=r^2$) ...

We do this 6 times (5 new ones) to get 6 values for f_{51} etc

We put these values together with a univariate interpolation algorithm. Number of evals is 6+5*3+5*4=41, much less than 216 required by dense interpolation. Remember the eval was a modular GCD image calculation.

How good is this?

- Dependent on sparseness assumptions that skeletons are accurate
- The VDM matrices must be non-singular (could be a problem for Z_p --not fields of char. zero.)
- Probabilistic analysis suggests it is very fast, $O(ndt(d+t))$ for n variables, t terms, degree d. Compare to $O((d+1)^n)$ conventional interpolation.
- Probability of error is bounded by $n(n-1)dt/B$ where B is cardinality of field. (Must avoid polynomials zeros).
- Finding a proof that this technique could be made deterministic in polynomial time was a puzzle solved, eventually, with $n^4d^2t^2$ bound (combined efforts of Zippel, Grigor'ev, Karpinski, Singer, Ben Or, Tiwari)

Polynomial evaluation

- What's to say?

Single polynomial evaluation

- $p(x_i)$, evaluation in the common way is n multiplies and n adds for degree n. There are faster ways by precomputation if either the polynomial coefficients or the points are available earlier. (Minor niche of complexity analysis for pre-computed polynomials, major win if you can pick special points.)
- "Horner's rule".. $a_0+a_1*(x+a_2*(x+ \ldots))$

Two-point polynomial evaluation, at r, -r

- Takes about $C(p(r))+3$ mults:
- $P(r)=P_e(r^2)+rP_o(r^2)$
- $P(-r)=P_e(r^2)-rP_o(r^2)$

Generalize to FFT, so this is (probably mistakenly) not pursued.

Short break... Factoring a polynomial h(x) using interpolation and factoring integers

- Evaluate h(0); find all factors. $h_{0,0}$, $h_{0,1}$ …
  - E.g. if h(0)=8, factors are –8,-4,-2,-1,1,2,4,8
- Repeat … until
- Factor h(n)
- Find a factor: What polynomial f(x) assumes the value $h_{0,k}$ at 0, $h_{1,j}$ at 1, …. ?
- Questions:
  - Does this always work?
  - What details need to be resolved?
  - How much does this cost?

Two possible directions to go from here

- Yet another way of avoiding costly interpolation in GCDs (Hensel, Zassenhaus Lemmas) see readings/hensel.pdf
- What has the FFT done for us lately, and why are we always obliquely referring to it? fft.pdf