7 Fast evaluation and interpolation

7.1 Fast multipoint evaluation

As usual, let R be a ring (commutative, with identity). Let $f \in \mathsf{R}[x]$ have degree less than n and $u_0, \ldots, u_{n-1} \in \mathsf{R}$. We want to evaluate

$$(f(u_0), f(u_1), \ldots, f(u_{n-1})) \in \mathsf{R}^n$$

quickly. Alternatively, let $M = (x-u_0)(x-u_1)\cdots(x-u_{n-1}) \in \mathsf{R}[x]$. The Chinese Remainder Theorem tells us that

$$\frac{\mathsf{R}[x]}{(M)} \cong \frac{\mathsf{R}[x]}{(x-u_0)} \times \frac{\mathsf{R}[x]}{(x-u_1)} \times \cdots \times \frac{\mathsf{R}[x]}{(x-u_{n-1})}$$

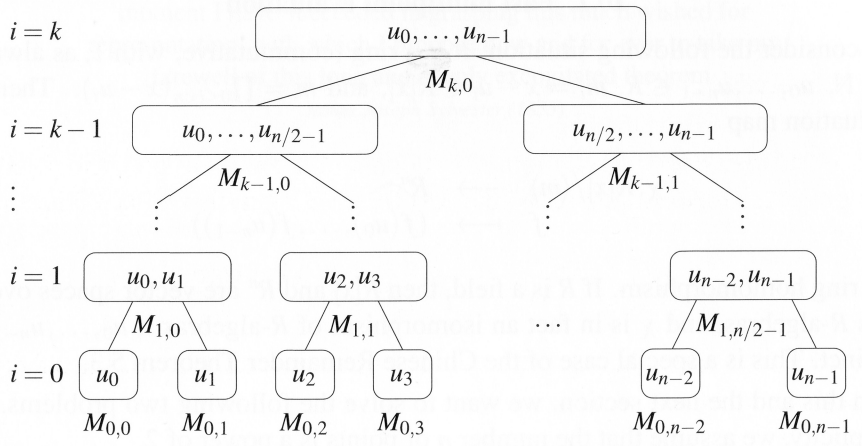
and this is realized computationally by

$$f \mapsto (f \operatorname{rem}(x-u_0), f \operatorname{rem}(x-u_1), \ldots, f \operatorname{rem}(x-u_{n-1})).$$

(remember that $f \operatorname{rem}(x-\alpha) = f(\alpha)$ for any $\alpha \in \mathsf{R}$). We have seen how to do this with $O(n^2)$ operations in R. How can we do it faster, by using fast polynomial multiplication?

Let $\mathsf{M}(n)$ be a function such that we can multiply two polynomials in $\mathsf{R}[x]$ with degrees less than n with $\mathsf{M}(n)$ operations in R. We have seen that $\mathsf{M}(n) \in O(n^2)$ using classical "school" arithmetic, $\mathsf{M}(n) \in O(n^{1.59})$ using Karatsuba's method, and $\mathsf{M}(n) \in O(n \log n)$ using the FFT (assuming we have a "nice" ring R with appropriate PRUs, etc.). To be as general as possible, we will only make three (quite reasonable) assumptions: $\mathsf{M}(n) \geq n$ (at least linear), $\mathsf{M}(n) \in O(n^2)$ (at most quadratic), and $\mathsf{M}(m+n) \geq \mathsf{M}(m) + \mathsf{M}(n)$ for all $m, n \in \mathbb{N}$.

We start by computing a so-called sub-product tree. Assume $n = 2^k$ and $u_0, \ldots, u_{n-1} \in \mathsf{R}$ and M as above. Associate with each point u_i the degree-1 polynomial $m_i = (x-u_i)$, and put these products into a "tree" as follows.



At each node compute the product of polynomials in the node. Formally

$$M_{ij} = m_{j2^i} \cdot m_{j2^i+1} \cdot m_{j2^{i+2}} \cdots m_{j2^i+(2^i-1)} = \prod_{0 \le \ell < 2^i} m_{j2^i+\ell}.$$

In other words, if R is an integral domain and $u_0, \ldots, u_{n-1}$ are distinct, then M_{ij} is the monic squarefree polynomial whose zero set is the jth node from the left at level i.

A simple observation reveals that

$$M_{i+1,j} = M_{i,2j} \cdot M_{i,2j+1},$$

that is, the polynomial at each node is the product of the polynomials at its two child nodes. This leads to the following algorithm:

Algorithm: `BuildProductTree`
Input: ‣ $m_0, m_1, \ldots, m_{r-1} \in \mathsf{R}[x]$, where $r = 2^k$ for some $k \in \mathbb{N}$
Output: ‣ $M_{i,j}$ as above, with $0 \le i \le k$ and $0 \le j < 2^{k-i}$
(1) **for** $0 \le j < r$ **do** $M_{0,j} \leftarrow m_j$
(2) **for** $1 \le i \le k$ **do**
(3) **for** $0 \le j < 2^{k-i}$ **do** $M_{i,j} \leftarrow M_{i-1,2j} \cdot M_{i-1,2j+1}$

In our discussion above, we had $m_0, \ldots, m_{n-1}$ all with degree one, but note that we do not actually require this here. The algorithm works for polynomials $m_0, \ldots, m_{r-1}$ of any degree.

Theorem 7.1. *The algorithm* `BuildProductTree` *requires* $O(\mathsf{M}(n) \log r)$ *operations in* R*, where* $n = \sum_{0 \le i < r} \deg m_i$

Proof. Step (1) uses no arithmetic operations. Let $d_{i,j} = \deg M_{i,j}$. Then the cost for the ith iteration of Step 3 is at most

$$\sum_{0 \le j < 2^{k-i}} \mathsf{M}\left(d_{i,j}\right) \le \mathsf{M}(n)$$

operations in R since $\sum_{0 \le j < 2^{k-i}} d_{i,j} = n$. There are $k = \log r$ iterations. □

Once we have built the sub-product tree we will "go down" the tree to do evaluation. The following simple algorithm accomplishes this, again in a divide-and-conquer manner.

Algorithm: `DescendProductTree`
Input: ‣ $f \in \mathsf{R}[x]$ of degree less than $n = 2^k$ for some $k \in \mathbb{N}$, $u_0, u_1, \ldots, u_{n-1} \in \mathsf{R}$, and the sub-products M_{ij} as above when $m_0 = (x - u_0), m_1 = (x - u_1), \ldots, m_{n-1} = (x - u_{n-1})$.
Output: ‣ $f(u_0), f(u_1), \ldots, f(u_{n-1}) \in \mathsf{R}$
(1) If $n = 1$ return f
(2) $r_0 \leftarrow f \text{ rem } M_{k-1,0}$, $r_1 \leftarrow f \text{ rem } M_{k-1,1}$
(3) Call `DescendProductTree` recursively to compute $r_0(u_0), r_0(u_1), \ldots, r_0(u_{n/2-1})$
(4) Call `DescendProductTree` recursively to compute $r_1(u_{n/2}), r_1(u_{n/2+1}), \ldots, r_1(u_{n-1})$
(5) Return $r_0(u_0), \ldots, r_0(u_{n/2-1}), r_1(u_{n/2}), r_1(u_{n/2+1}), \ldots, r_1(u_{n-1})$.

Theorem 7.2. *The algorithm* `DescendProductTree` *works as stated.*

Proof. We prove correctness by induction on k. If $k = 0$ then f is constant, and the correct answer is returned in Step 1.

Otherwise, if $k \geq 1$, assume that the results of Step 3 and Step 4 are correct. Let

$$q_0 = f \text{ quo } M_{k-1,0} \qquad q_1 = f \text{ quo } M_{k-1,1}.$$

Then

$$f(u_i) = \begin{cases} q_0(u_i)M_{k-1,0}(u_i) + r_0(u_i) = r_0(u_i) & \text{if } 0 \leq i < \frac{n}{2}, \\ q_1(u_i)M_{k-1,1}(u_i) + r_1(u_i) = r_1(u_i) & \text{if } \frac{n}{2} \leq i < n. \end{cases}$$

□

Theorem 7.3. *The algorithm* `DescendProductTree` *requires* $O(\mathsf{M}(n)\log n)$ *operations in* R.

Proof. Let $T(n) = T(2^k)$ be the number of operations in R required. Then $T(1) = 0$ and

$$T(2^k) = 2T(2^{k-1}) + 2M(2^{k-1}).$$

It is easily shown by induction that $T(2^k)$ is $O(\mathsf{M}(2^k)\cdot k)$ or $O(\mathsf{M}(n)\log n)$. □

Putting things together, we can evaluate any polynomial at points $u_0, \ldots, u_{n-1} \in \mathsf{R}$ by first computing the subproduct tree using algorithm `BuildProductTree` on $x-u_0, x-u_1, \ldots, x-u_{n-1}$, and then descending the subproduct tree computing remainders by f. The total cost is $O(\mathsf{M}(n)\log n)$.

7.2 Fast Interpolation

Recall the formula for Lagrange interpolation. Given distinct $u_0, \ldots, u_{n-1}$ in a field R and arbitrary values $v_0, \ldots, v_{n-1} \in \mathsf{R}$, find the unique polynomial $f \in \mathsf{R}[x]$ of degree less than n such that $f(u_i) = v_i$ for $i = 0, 1, \ldots, n-1$. We used the fact that

$$f = \sum_{0 \leq i < n} v_i s_i \frac{M}{x - u_i},$$

where $M = (x-u_0)(x-u_1)\cdots(x-u_{n-1})$ as above and

$$s_i = \prod_{j \neq i} \frac{1}{u_i - u_j}.$$

First, let's consider the simpler problem of computing

$$f = \sum_{0 \leq i < n} c_i \cdot \frac{M}{x - u_i}$$

for some given $c_0, \ldots, c_{n-1} \in \mathsf{R}$. Eventually we will choose special values for $c_0, \ldots, c_{n-1}$.

Algorithm: `LinearCombination`

Input: ‣ $u_0,\ldots,u_{n-1} \in \mathsf{R}$, $c_0,\ldots,c_{n-1} \in \mathsf{R}$, where $n = 2^k$ for some $k \in \mathbb{N}$, and the sub-products $M_{i,j}$ as above computed from the polynomials $m_i = (x-u_i)$.

Output: ‣ $\sum_{0\le i<n} c_i \frac{M}{x-u_i} \in \mathsf{R}[x]$, where $M = M_{k,0} = m_0 \cdots m_{n-1} = (x-u_0)\cdots(x-u_{n-1})$.

(1) If $n = 1$ return c_0

(2) Call `LinearCombination` recursively to compute

$$r_0 = \sum_{0\le i<n/2} c_i \frac{M_{k-1,0}}{x-u_i}$$

(3) Call `LinearCombination` recursively to compute

$$r_1 = \sum_{n/2\le i<n} c_i \frac{M_{k-1,1}}{x-u_i}.$$

(4) Return $M_{k-1,1} r_0 + M_{k-1,0} r_1$

Theorem 7.4. *The algorithm* `LinearCombination` *works as stated.*

Proof. Again, prove this by induction on k. If $k = 0$ then $M = x - u_0$ and the output is correct. if $k \ge 1$, then the results of the recursive calls in Steps 2 and 3 are correct by the induction hypothesis. The output is correct in Step 4, since $M = M_{k,0} = M_{k-1,0} \cdot M_{k-1,1}$ □

Theorem 7.5. *The algorithm* `LinearCombination` *requires* $O(\mathsf{M}(n)\log n)$ *operations in* R.

Proof. Very similar to the proof of Theorem 7.3. □

To do interpolation, what we are left with is finding $c_i = v_i s_i$ for $0 \le i < n$, and then using the algorithm `LinearCombination`. This requires a really cool observation. Again, let $M = (x-u_0)(x-u_1)\cdots(x-u_{n-1})$. Then

$$M' = \sum_{0\le k<n} \frac{m}{x-u_j}$$

so

$$M'(u_i) = \prod_{\substack{0\le j<n \\ i\ne j}} (u_i - u_j) = \frac{1}{s_i}$$

(refer back to Script 6 for the form of s_i). Thus, computing $s_0,\ldots,s_{n-1}$ can be accomplished by evaluating $M'(u_i)$ for $0 \le i < n$. Since we already have $M = M_{k,0}$ from the product tree, we can easily calculate M' by computing a derivative, and then we can compute $M'(u_0), M'(u_1), \ldots, M'(u_{n-1})$ with fast multi-point evaluation.

7.3 Fast multi-modular reduction and Chinese remaindering

The ideas of the previous section carry over to moduli of arbitrary degree in $\mathsf{R}[x]$, and to $\mathbb{Z}$. Here, evaluation and interpolation correspond to "computing the isomorphism". Over $\mathsf{R}[x]$ we have

$$\frac{\mathsf{R}[x]}{(M)} \cong \frac{\mathsf{R}[x]}{(m_0)} \times \frac{\mathsf{R}[x]}{(m_1)} \times \cdots \times \frac{\mathsf{R}[x]}{(m_{r-1})},$$

for $m_0,\ldots,m_{r-1} \in \mathsf{R}[x]$ of *any* degree, with $\gcd(m_i,m_j) = 1$ for $i \neq j$ and $M = m_0 m_1 \cdots m_{r-1}$. Over $\mathbb{Z}$ we have

$$\frac{\mathbb{Z}}{(M)} \cong \frac{\mathbb{Z}}{(m_0)} \times \frac{\mathbb{Z}}{(m_1)} \times \cdots \times \frac{\mathbb{Z}}{(m_{r-1})}$$

for $m_0,\ldots,m_{r-1} \in \mathbb{Z}$ for $\gcd(m_i,m_j) = 1$ for $i \neq j$, and $M = m_0 m_1 \cdots m_{r-1} \in \mathbb{Z}$.

First consider multi-modular reduction in the case $\mathsf{R}[x]$. The sum of the degrees of nodes at any given level in the produce tree is $\deg M$. If $r = 2^k$, and we start with a polynomial f with $\deg f < \deg m$, then using the superlinearity of M we have that the cost of all modular reductions occurring at a given level of the product tree is bounded by $c\sum_d \mathsf{M}(d) \leq c\mathsf{M}(\sum_d d) \in O(\mathsf{M}(\deg m))$. Since there are $\log r$ levels, the overall cost is $O(\mathsf{M}(n)\log r)$ operations from R, where $n = \deg m$. Over $\mathbb{Z}$ the cost of multi-modular reduction is $O(\mathsf{M}(n)\log r)$ word operations, where $n = \log m$.

Now consider Chinese remaindering. Recall that $s_i = \mathrm{rem}((M/m_i)^{-1}, m_i)$. Once the s_i's have been computed, computing $c_i = \mathrm{rem}(s_i v_i, m_i)$ for $0 \leq i \leq r-1$, and going up the product tree to recover $f = \sum_{0\leq i\leq r-1} c_i M/m_i$ also costs $O(\mathsf{M}(n)\log r)$. To compute s_i requires a really cool observations. Notice that

$$\mathrm{rem}((M/m_i)^{-1}, m_i) = \mathrm{rem}((\,\mathrm{rem}\,(M, m_i^2)/m_i)^{-1}, m_i).$$

Thus, we can compute the s_i by multi-modular reduction using a product tree with all nodes squared! In summary, both directions of the isomorphism implied by the Chinese remainder theorem can be computed in time $O(\mathsf{M}(n)\log r)$ (either operations from R or word operations).

For more details see the text (von zur Gathen & Gerhard), Section 10.3.

8 Complexity summary

Many of the basic algebraic computations we consider deal with integers or with polynomials from $\mathsf{R}[x]$, where R is a commutative ring. In modern treatments it is customary to give cost estimates in terms of a *multiplication time* $\mathsf{M}(\cdot)$, a function such that:

- $\mathsf{M}(n)$ ring operations from R are sufficient to multiply together two polynomials from $\mathsf{R}[x]$ with length bounded by n, where the length of a polynomial is one less then the degree.
- $\mathsf{M}(n)$ word operations are sufficient to multiply together two integers of length bounded by n words.

The standard algorithms (naive cost table) allow $\mathsf{M}(n) \in O(n^2)$, while Karatsuba allows $\mathsf{M}(n) \in O(n^{\log_2 3})$. The fastest algorithms allow $\mathsf{M}(n) \in O(n \log n)$ for multiplication over $\mathbb{Z}$, and $\mathsf{M}(n) \in O(n(\log n)(\log\log n))$ for multiplication over $\mathsf{R}[x]$. Note that we are overloading M, but it is usually clear from the context (or can be clarified) if we are counting ring operations from R or word operations.

Because we want our cost estimates expressed in terms of M to be valid for any of these multiplication algorithms, and to be able to simplify cost estimates, we make the mild assumption that M is superlinear and at most quadratic, that is,

- $n \leq \mathsf{M}(n)$
- $\mathsf{M}(n) + \mathsf{M}(m) \leq \mathsf{M}(n+m)$
- $m\mathsf{M}(n) \leq \mathsf{M}(mn) \leq m^2\mathsf{M}(n)$

If we restrict ourselves to these assumptions when simplifying cost estimates involving M, our analysis will remain valid assuming any of the multiplication times mentioned above.

We have seen that some operations can be reduced to multiplication, for example division with remainder can be done in time $O(\mathsf{M}(n))$. Other operations, such as constructing a product tree from n points as in §7.1, also reduce to multiplication and have cost $O(\mathsf{M}(n)\log n)$. It turns out that this extra logarithmic factor only crops up in the analysis if the multiplication time M is pseudo-linear. If $\mathsf{M}(n) \in \Omega(n^{1+\varepsilon})$ for any fixed constant $\varepsilon > 0$, then a product tree can be constructed in time $O(\mathsf{M}(n))$. For this reason it is useful to introduce a second cost function $\mathsf{B}(\cdot)$ that captures the cost of problems such as computing a product tree. The function B satisfies the same "superlinear and at most quadratic" assumptions as M, and $\mathsf{B}(t)$ can always be replaced in cost estimates with $\mathsf{M}(t)(\log t)$. All of the problems considered in this script have running time $O(\mathsf{B}(n))$ (either operations from R or word operations). We will see additional problems that have algorithms with running time $O(\mathsf{B}(n))$, including radix conversion, the extended gcd problem, and rational number/function reconstruction.