3 Evaluation, Interpolation and Multiplication of Polynomials

3.1 Evaluation of polynomials with respect to different cost measures

Problem 3.1 (Polynomial evaluation)**.** *Let* R *be some ring in which we can perform the basic operations* $(+, -, *)$*. Given* $n \in \mathbb{N}$*, find an algorithm that, on input* $\alpha, a_0, \cdots, a_n \in \mathsf{R}$*, computes* $f(\alpha) \in \mathsf{R}$*, where*

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \in \mathsf{R}[x].$$

This is the problem of polynomial evaluation.

Obvious (school) algorithm. The obvious algorithm to evaluate the polynomial is to compute $\alpha^2, \alpha^3, \ldots, \alpha^n$, requiring $n-1$ multiplications, and then computing the terms $a_i x^i$, requiring a further n multiplications. Finally, adding the terms together takes another n additions.

Horner's evaluation. We can improve the number of multiplications by a factor of two if we change the order in which we do our adding and multiplying. Consider rewriting the polynomial as

$$f(\alpha) = (\cdots((a_n\alpha + a_{n-1})\alpha + a_{n-2})\alpha + \cdots)\alpha + a_0.$$

Computing the innermost bracket, $a_n\alpha + a_{n-1}$ requires one multiplication and one addition. This process is repeated n times to get $f(\alpha)$. The cost of the algorithm using this algorithm known as *Horner's rule* is n multiplications and n additions as opposed to $2n-1$ multiplications and n additions for the above naïve method.

Many consider the birthday of *algebraic complexity theory* was in 1954 when Alexander Ostrowski asked whether or not Horner's rule was optimal.

Ostrowski also asked about a different notion of cost which can also be very useful. Consider the case where we have an algorithm which takes extended-precision integers as inputs and produces extended precision integers as output. Multiplying these integers can be expensive (quadratic time with what you've seen so far). Multiplication by numbers known in the algorithm is generally quicker (we can generally optimize). Additions are still quite quick (and certainly dominated by the cost of the multiplications unless there were a *huge* number of them). So let's only count extended precision integer multiplications of "input dependent" quantities. (The same reasoning motivates algorithms which take matrices as inputs; matrix multiplication is even more expensive).

More formally, Ostrowski introduced the notion of *non-scalar complexity.* He posed the following problem: Suppose that F is a field, and $\mathsf{R} = \mathsf{F}[\alpha, a_0, \ldots, a_n]$ the ring of polynomials in indeterminates $\alpha, a_0, \ldots, a_n$. *Scalar* operations are either addition of any two elements in R, or multiplication of an element of R by a fixed constant from F (i.e., a constant hard-coded in the algorithm). Other operations (basically multiplication of two input or non-scalar quantities) are called *non-scalar* operations.

The question is: can you then improve the number non-scalar operations in the evaluation of polynomials, over Horner's method. Victor Pan in 1959 showed the answer to be no: Horner's rule

is optimal!

This is a powerful idea because:

- Multiplication by constants in the algorithm can be optimized by the compiler/hardware.
- Constants in the algorithm are thought of as "small". Inputs may be big (as in the above long integer discussion).
- Addition is generally in theory and practice.
- The computational model separates out what is (often) important.

But what if we want to evaluate a known polynomial. I.e., if the coefficients $a_0, \ldots, a_n$ of the polynomial are not indeterminates, but fixed field elements (scalars), can we do better? The following important result shows the answer is yes!

Theorem 3.2 (Paterson and Stockmeyer 1973)**.** *Let $f \in \mathsf{F}[x]$, F any field, have degree n. Then $f(\alpha)$ can be evaluated at any $\alpha \in \mathsf{F}$ with $2\lceil\sqrt{n}\rceil - 1$ non-scalar multiplications.*

Proof. Begin by partitioning f into roughly $\sqrt{n}$ blocks of length $\sqrt{n}$. Let $m = \lceil\sqrt{n}\rceil$ and $k = \lceil n/m\rceil + 1$. Then

$$f(x) = \sum_{0\le i\le n} a_i x^i = (a_{km-1}x^{m-1} + \cdots + a_{(k-1)m})x^{(k-1)m} + \\ \cdots + (a_{2m-1}x^{m-1} + \cdots + a_m)x^m + (a_{m-1}x^{m-1} + \cdots + a_0).$$

(The a_i with $i > n$ are set to zero.)

Evaluation then can be done by first computing

$$\alpha, \alpha^2, \cdots, \alpha^m$$

using $m-1$ non-scalar operations. We then compute $\beta_i = \sum_{0\le j<m} a_{im+j}\alpha^j$, for $0 \le i < k$, at no cost because we only multiply by scalars and add. Horner's rule, applied to evaluating $\sum_{0\le i<k} \beta_i \alpha^{im}$ can be done with an additional $k-1$ non-scalar multiplications and some additions which we don't count. The total cost is $(m-1)+(k-1) \le 2\lceil\sqrt{n}\rceil - 1$. □

Sometimes this is referred to as "Baby-steps, Giant-steps" evaluation. In practice, it can be very useful, but only in some special cases. For example, when trying to evaluate polynomials at matrices in which the coefficients may be integers and the indeterminate x is an $m \times m$ matrix. It makes sense when evaluating such a polynomial to reduce, as much as possible, the number of matrix multiplications used (which are expensive). If we regard the integers as scalars then the above result implies that the polynomial can be evaluated using about $2\sqrt{n}$ matrix multiplications. The naïve algorithm above and Horner's rule give algorithms that use $2n-1$ and n matrix multiplications, respectively.

3.2 Back to Polynomial Multiplication

We have seen that to multiply two polynomials of degree n together takes $O(n^2)$ operations in the coefficient field: $(n+1)^2$ multiplications and n^2 additions. For instance, multiplying

$$(ax+b)(cx+d) = acx^2 + (ad+bc)x + bd$$

uses four multiplications ac, ad, bc, bd, and one addition $ad+bc$. Note that we are not calculating the value of ab at x, but are only computing the coefficient sequence.

Surprisingly, there is an easy method of doing better. Assume $n = 2^k$ for some $k \in \mathbb{N}$, and that a and b are polynomials of degree less than n. Let $m = n/2$. Then rewrite

$$a = A_1 x^m + A_0$$
$$b = B_1 x^m + B_0$$

where $\deg A_i, \deg B_i < m$ for $i = \{0,1\}$. If $\deg a < n-1$, then some of the top coefficients are zero. Now

$$ab = A_1B_1x^n + (A_0B_1 + A_1B_0)x^m + A_0B_0.$$

In this form, multiplication of a and b has been reduced to 4 multiplications of polynomials of degree less than m and four additions of polynomials with degree less than $2n$. Note that multiplication by a power of x does not count as a multiplication, since it corresponds merely to a shift of the coefficients. Does this help? No.

So far we have not really achieved anything. But this new expression for ab can be rearranged to reduce the number of multiplications of the smaller polynomials at the expense of increasing the number of additions. Since multiplication is slower than addition, a saving is obtained when n is sufficiently large.

The idea is due to Karatsuba & Ofman (1965). Rewrite the product as

$$ab = A_1B_1(x^n - x^m) + (A_1 + A_0)(B_1 + B_0)x^m + A_0B_0(1 - x^m).$$

With this expression, we see that multiplication of a and b requires only three multiplications of polynomials of degree less than m and six additions of polynomials of degree at most $2n$.

If $T(n)$ denotes the time necessary to multiply two polynomials of degree less than n, then $T(2n) \le 3T(n) + cn$, for some constant c. The linear term comes from the observation that addition of two polynomials of degree less than ℓ can be done with ℓ operations.

Theorem 3.3. *If $T(2^k) \le 3T(2^{k-1}) + c2^k$, then $T(2^k) \le 3^k - 2c \cdot 2^k$ for $k \ge 1$.*

Proof. By induction on k.

$$\begin{aligned} T(2^k) &\le 3T(2^{k-1}) + cn \\ &\le 3 \cdot (3^{k-1} - 2c \cdot 2^{k-1}) + c2^k \\ &= 3^k - 2c \cdot 2^k. \end{aligned}$$

□

Corollary 3.4. *The Karatsuba-Ofman algorithm for multiplying polynomials over a ring can be done with $O(n^{\log_2 3})$ or $O(n^{1.59})$ ring operations.*

Proof. The above discussion and the theorem imply that the algorithm requires $O(3^{\log_2 n})$ steps. The result follows from noting that $3^{\log_2 n} = 2^{\log_2 3 \cdot \log_2 n} = n^{\log_2 3}$, and $\log_2 3 < 1.59$.

□

This is a substantial improvement over the classical method, since $\log_2 3 < 2$. It works in practice too.

Question: Work out the smallest value of c. Check that the classical method requires $2(n+1)^2 - (2n+1)$ operations. For which values of $n = 2^k$ is $3^k - 2c2^k$ smaller than the classical $2(n+1)^2 - (2n+1)$?

Polynomial multiplication in the non-scalar setting This problem can also be considered from the non-scalar perspective. Here we consider the non-scalars to be the coefficients on the polynomials we want to multiply.

The algorithms we develop here may not be any faster in the standard model (they may in fact be slower). However, the designs will lead to much better algorithms.

Theorem 3.5. *Polynomial multiplication over a field can be performed using only $2n-1$ non-scalar multiplications, if both polynomials have degree at most n and the field has at least $2n-1$ elements.*

Proof. Pick $2n+1$ distinct scalars $u_i \in \mathsf{F}$, $i = 0, \cdots, 2n$. Evaluate a and b at the u_i's: $\alpha_i = a(u_i)$ and $\beta_i = b(u_i)$ for $i = 0, \cdots, 2n$. This costs nothing since we only perform additions and scalar multiplications (think about it).

Next we compute $\gamma_i = \alpha_i \beta_i$. This requires $2n+1$ non-scalar multiplications.

Finally, we interpolate to get $c = ab$. The Lagrange formula,

$$L_i = \prod_{j \neq i} \frac{(x - u_j)}{(u_i - u_j)} \in \mathsf{F}[x],$$

has the property that $L_i(u_k)$ is 0 if $i \neq k$ and 1 when $i = k$. Now

$$c = \sum_{0 \leq i \leq 2n} \gamma_i L_i = \sum_{0 \leq i \leq 2n} \gamma_i \prod_{i \neq j} \frac{(x - u_j)}{(u_i - u_j)}$$

is the product polynomial. All operations occuring in the last formula are additions and scalar multiplications, which are for free in the non-scalar model. □

In the scalar model, both the evaluation step and the interpolation step look expensive. Note that we are free to choose the points at which to evaluate the polynomial. Sometimes it is possible to pick these points so that evaluation and interpolation can be solved more quickly than $O(n^2)$.

Evaluation and interpolation are closely related to matrix-vector multiplication. Let

$$VDM(u_1,\ldots,u_n) = \begin{pmatrix} u_1^0 & u_1^1 & \ldots & u_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ u_n^0 & u_n^1 & \ldots & u_n^{n-1} \end{pmatrix} \in \mathsf{F}^{n\times n}$$

be a *Vandermonde* matrix. It is easy to see from the expression

$$VDM(u_1,\ldots,u_n)\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} a(u_1) \\ \vdots \\ a(u_n) \end{pmatrix}$$

that matrix multiplication can be used to evaluate a polynomial. Using this notation we can see that interpolation can then be effected by computing a matrix inverse and then multiplying, i.e.,

$$\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = (VDM(u_1,\ldots,u_n))^{-1}\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}.$$

We will use these ideas later to get an asymptotically fast algorithm for polynomial multiplication.

Karatsuba's Algorithm is an example of a divide and conquer strategy for algorithms. The non-scalar algorithm, which changed the representation of the polynomials, is a simple example of a technique called computing by homomorphic images which we will be examining later in this course.

3.3 Multiplication of Integers

As we have seen, the classical algorithm for integer multiplication learned in school requires $O(n^2)$ bit operations when a and b have at most n digits. Karatsuba's algorithm works in much the same way for integers as for polynomials. Write $a = A_1 2^n + A_0$ and $b = B_1 2^n + B_0$ where the binary lengths of a and b are less than n. The product ab can be rewritten as $(2^{2n}+2^n)A_1B_1 + 2^n(A_1 - A_0)(B_0 - B_1) + (2^n+1)A_0B_0$. As in the polynomial case, multiplication of two integers has been reduced to multiplication of three integers of at most half the size plus a few ($O(n)$) operations. We conclude, as for polynomials, that multiplication of two n bit integers requires $O(n^{\log_2 3})$ digit operations.

In existing computer algebra systems (and especially in underlying arithmetic packages like GMP and NTL), asymptotically fast algorithms are often used for larger integers and higher degree polynomials. The issue of representation plays a crucial role in determining at what point the "fast" methods beat the "classical" algorithms. As well, hybrid algorithms are often the ultimate choice, much as they are for sorting and searching methods.