JOURNAL OF COMPUTER AND SYSTEM SCIENCES 8, 366–386 (1974)

Fast Modular Transforms*

A. BORODIN

Department of Computer Science, University of Toronto, Toronto, Ontario, M5S 1A7 Canada

AND

R. MOENCK

Department of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario, N2L-3G1 Canada

Received February 12, 1973

It is shown that if division and multiplication in a Euclidean domain can be performed in $O(N \log^a N)$ steps, then the residues of an N precision element in the domain can be computed in $O(N \log^{a+1} N)$ steps. A special case of this result is that the residues of an N precision integer can be computed in $O(N \log^2 N \log\log N)$ total operations. Using a polynomial division algorithm due to Strassen [24], it is shown that a polynomial of degree $N-1$ can be evaluated at N points in $O(N \log^2 N)$ total operations or $O(N \log N)$ multiplications.

Using the methods of Horowitz [10] and Heindel [9], it is shown that if division and multiplication in a Euclidean domain can be performed in $O(N \log^a N)$ steps, then the Chinese Remainder Algorithm (CRA) can be performed in $O(N \log^{a+1} N)$ steps. Special cases are: (a) the integer CRA can be performed in $O(N \log^2 N \log\log N)$ total operations, and (b) a polynomial of degree $N-1$ can be interpolated in $O(N \log^2 N)$ total operations or $O(N \log N)$ multiplications. Using these results, it is shown that a polynomial of degree N and all its derivatives can be evaluated at a point in $O(N \log^2 N)$ total operations.

1. INTRODUCTION

Many of the efficient algorithms which have recently been developed for polynomial and number theoretic operations fall into a class which may be described as homomorphism algorithms. These algorithms operate on sampled values (homomorphic images) of their variables. The general form of a homomorphism algorithm can be represented schematically as

$$\begin{array}{ccc} A & \xrightarrow{f_A} & A' \\ {\scriptstyle\phi}\downarrow & & \uparrow{\scriptstyle\phi'} \\ B & \xrightarrow{f_B} & B' \end{array},$$

* This research was supported by NRC Grants Nos. A-5549, A-7631, A-7641.

where A and A' are the inputs and results, respectively, of performing the operation f_A on the original problem. B, B', and f_B are the inputs, results, and operation for the sampled problem, respectively. The functions ϕ and ϕ' are used to map in and out of the sampled solution space. Analysis has shown that it is frequently better to compute A' by way of B, f_B, and B', rather than directly using f_A. Examples of such algorithms applied to linear equations, polynomial GCD's, and resultants are given by Cabay [4], Brown [3], and Collins [5], respectively.

In the polynomial and number theoretic cases (and indeed, for general Euclidean domains), the modular homomorphisms are the ones most frequently used. For a Euclidean domain D, the operation f_B corresponds to computing modulo some element in the domain. In the number theoretic case, a convenient integer (frequently a prime) is used as the modulus. The polynomial case is computed modulo a polynomial (frequently a linear polynomial).

It is apparent that the transformations ϕ and ϕ' are critical links in such algorithms. It is these transformations which we shall investigate with a view to looking for fast algorithms. (Hence the title.) The transformation ϕ corresponds to computing the residues of an element of the domain with respect to several moduli. ϕ' involves computing the Chinese Remainder Algorithm (i.e., interpolating) in the domain.

In the polynomial case, the transforms correspond to evaluating a polynomial at many points (for linear moduli) and interpolating a polynomial given the values at sufficiently many points. These transforms could be performed by the Fast Fourier Transform (FFT) (cf. Pollard [19]). However, in some algorithms, situations occur where certain sample values must be discarded (cf. Brown [3] and Collins [5]). This implies that the FFT cannot be easily used, since it depends on a strict relationship between the sample points. This leaves us with the interpolation problem and its dual the problem of evaluation at many points. In any case, the general problems of evaluation and interpolation are interesting in their own right.

The analogue of the interpolation problem in the number theoretic (integer) case is the integer Chinese Remainder Algorithm (CRA). Here we are given a set of residues corresponding to a set of moduli. The problem is to compute the unique integer with the same set of residues. The dual problem is to compute the residues of an integer with respect to a set of moduli. Lipson [12] shows that interpolation and the CRA are abstractly equivalent to the CRA for a Euclidean domain and gives a thorough exposition of the classical algorithms for these problems.

Classically, all these algorithms require $O(N^2)$ steps, i.e., to interpolate or evaluate a polynomial of degree $N - 1$, or perform the CRA for an N precision integer all require $O(N^2)$ steps. The question is whether this can be improved to something of the order of the FFT (i.e., $O(N \log N)$ steps). Intuition (perhaps) indicates that these algorithms cannot be improved upon. However, intuition is often wrong as shown by Fourier polynomial multiplication (cf. Pollard [17]) or the Schoenhage–Strassen integer multiplication [19].

Horowitz and Heindel [9] in investigating this question have produced an integer CRA which works in $O(N \log^2 N \log \log N)$ steps as a preconditioned algorithm and $O(N \log^3 N \log \log N)$ in its complete version. Borodin and Munro [7] have shown that many point polynomial evaluation can be performed in $O(N^{1.91})$ steps using Strassen's matrix multiplication algorithm assuming noniterative computation, but would require $O(N^2)$ iteratively. Horowitz [10] has given a preconditioned polynomial interpolation algorithm which operates in $O(N \log^3 N)$.

Subsequent to our original report [14], Strassen [24] has developed an improved algorithm for polynomial division and, more important, proved a number of significant lower bounds. As a result of Strassen's work, we can now say that the multiplicative complexity of the algorithms presented here is within an order of magnitude of optimality.

We must make the disclaimer that the algorithms presented here are not presently of practical usefulness. Their importance lies in being a theoretical background for the development of practical methods.

2. Some Remarks on Fast Algorithms

Since we are going to be looking for fast (i.e., $O(N \log^a N)$) algorithms, we should first look at the form of such algorithms in order to discover common features and the sort of properties such algorithms could exhibit.

One common property of many such algorithms is that they solve a problem by dividing it into two simpler problems, each of which is half as difficult as the original problem. This "Divide and Rule" formulation implies that the timing function of the algorithm is defined by a recurrence relation of the form

$$T(N) = 2 \times T(N/2) + f(N),$$

where $f(N)$ is not too large. In fact, if $f(N) = O(N \log^a N)$, then $T(N) = O(N \log^{a+1} N)$. This follows from expanding the recurrence relation:

$$\begin{aligned} T(N) &= 2 \cdot T(N/2) + O(N \log^a N) \\ &= 2(2T(N/4) + O(N/2 \log^a N/2)) + O(N \log^a N) \\ &= 4T(N/4) + O(N \log^a N/2) + O(N \log^a N) \\ &= NT(1) + O\left(N \sum_{i=0}^{\log N} i^a\right) = O(N \log^{a+1} N). \end{aligned}$$

Divide and Rule algorithms also tend to be easily expressed recursively. Good examples of such algorithms are Merge Sorting and the Fast Fourier Transform.

Another property of fast algorithms which depends on certain critical subroutines is that the efficiency of the subroutine is optimized by correct choice of the size of the

inputs. For example, if the critical subroutine is a multiplication algorithm, then the size of the inputs must be approximately the same in order to maximize the efficiency of the algorithm. This follows from the fact that both classical and any fast multiplication algorithm require N operations to multiply an N precision element by a single precision element. Examples of the balanced precision multiplication are seen in Schoenhage [21] and Horowitz and Heindel [9]. In fact, Horowitz and Heindel show that the classical algorithm for the CRA is not improved by using fast multiplication. (For another discussion on general techniques used in fast algorithms, see Moenck [15].)

3. Some Background

In order to emphasize the generality of the algorithms presented here, we shall define a common precision function for the integer and polynomial cases:

$$\text{prec}(U) = \begin{cases} \deg(U) + 1 & \text{if } U \text{ is a polynomial,} \\ \log_2(U) & \text{if } U \text{ is an integer.} \end{cases}$$

We shall analyze the polynomial form of the algorithms from two points of view. First, we shall count all arithmetic operations in an attempt to get a meaningful measure of the practical running time of the algorithms on a computer. Second, we will use a notion of complexity proposed by Ostrowski [17] and also used by Winograd [25], Hopcroft and Kerr [8], and Strassen [24]. This measure counts only the number of multiplications and divisions necessary to compute the function. Arbitrary linear combinations of partial results and multiplications by scalars are not counted.

As an example of the use of these two measures, we will consider the multiplication of two polynomials of degrees n and m, using Fourier multiplication. In the following analysis and throughout the rest of the paper, the algorithms presented will work for all N. However, in order to ease the analysis, we shall assume $N = 2^k$, for some $k \in \mathbb{N}$. This may mean that the constants of proportionality in the timing functions may be in error for general N. However, they will be "out" by a factor of at most 2. We shall use the notation ldt to mean lower degree terms. Thus, in the practical model, the polynomial multiplication involves

	Operation	*No. of arithmetic Steps*[1]
(1)	two forward Fourier transforms	$2(N + 3/2N \log N) + \text{ldt}$
(2)	multiplication of the sequences	N
(3)	and one inverse tranformation	$2N + 3/2N \log N + \text{ldt}$
	for a total of	$9/2N \log N + 5N + \text{ldt}$
	where $N = m + n$	

[1] Arithmetic steps will be the number of $\{+, -, *, \div\}$ operations from the field over which the polynomials are distributed.

In the Ostrowski model, arbitrary linear combinations of variables are allowed for free. This means that the Fourier transforms, which can be thought of as a sequence of linear combinations of the coefficients, can be performed at no cost. Thus, multiplication of two polynomials of degree n and m can be performed in the N multiplications required by step 2. Moreover, if we counted all multiplications, we could choose to perform the Fourier transform in a finite field and simulate integer multiplication by repeated additions (see, for example, Fiduccia [6]). This method also leads to an $O(N)$ multiplications algorithm.

We shall not analyze the integer form of the algorithms in the same detail, since we do not know the constants of proportionality involved for fast integer multiplication. Instead, we shall use the big-O notation and assume that integer multiplication can be performed in $O(N \log N \log\log N)$ steps using the Schoenhage–Strassen algorithm [20]. However, Lipson [13] has pointed out that for practical purposes, there is an $O(N \log N)$ algorithm performing integer multiplication for all N of conceivable interest.

4. Fast Modular Forms

As remarked above, classically the evaluation of a polynomial of degree $N-1$ at N points requires $O(N^2)$ operations. This is performed by doing N evaluations of the polynomial at one point. A similar bound holds for the computation of the N single precision residues of an N precision integer.

Evaluating a polynomial at one point can be considered as a division process (cf. Knuth [11, pp. 424]). This is a result of the remainder theorem, i.e., given a polynomial $p(x)$, if we divide by $x - a$ we get

$$p(x) = q(x)^* (x - a) + r(x), \tag{4.1}$$

where $\deg(r) = 0$ (i.e., a constant). Putting $x = a$ in (4.1), we get that $p(a) = r$. Horner's Rule and the process of synthetic division as used by the numerical analysts are directly related to this method.

The remainder theorem suggests a generalization to more points. If we wish to evaluate $p(x)$ at m points x_i, we form

$$M(x) = \prod_{i=1}^{m} (x - x_i) \tag{4.2}$$

and divide $p(x)$ by $M(x)$ to get

$$p(x) = M(x)^* q(x) + r(x),$$

where $\deg(r) < \deg(M)$. Then at the points $x = x_i$ we have

$$p(x_i) = r(x_i),$$

and assuming $\deg(M) < \deg(p)$, we have reduced the problem to a simpler one. In the more general framework of a Euclidean domain with $P = QM + R$,

$$P \bmod m_i \equiv R \bmod m_i$$

whenever $m_i \mid M$. Fiduccia [7] uses this approach in discussing one way of understanding the FFT.

Taking note of our remarks on fast algorithms, a method for evaluating a polynomial of degree $N - 1$ at N points suggests itself. First, we form a polynomial $M_1(x)$ with the first $N/2$ points $\{x_i\}$ as in (4.2) above and $M_2(x)$ from the remaining $N/2$ points. We divide $p(x)$ by $M_1(x)$ to get $R_1(x)$, and we obtain $R_2(x)$ in the same manner. This gives us two polynomials of degree $N/2 - 1$, each of which is to be evaluated at $N/2$ points. To do this, we use the method recursively, which gives us a Divide and Rule algorithm. For example, to evaluate $p(x) = x^3 - 3x + 5$ at $x = -1, 1, 2, 3$, we form

$$M_1(x) = (x + 1)(x - 1) = x^2 - 1,$$
$$M_2(x) = (x - 2)(x - 3) = x^2 - 5x + 6.$$

Dividing $p(x)$ by $M_1(x)$ and $M_2(x)$ we get

$$R_1 = -2x + 5, \qquad R_2 = 16x - 25.$$

Dividing R_1 by $(x + 1)$ and $(x - 1)$ we get from the remainders that

$$p(-1) = 7, \qquad p(1) = 3.$$

Dividing R_2 by $(x - 2)$ and $(x - 3)$ we get that

$$p(2) = 7, \qquad p(3) = 23,$$

which can be easily verified.

5. The Algorithm

In order to discuss the problem further in the general setting, we shall make the following characterization of the problem. In a Euclidean domain D, we are given a set of N moduli $\{m_i\} \in D$ and an element $U \in D$ for which we wish to compute the set of residues $u_i \in D$ such that

$$u_i \equiv U \bmod m_i, \qquad 1 \leqslant i \leqslant N.$$

For polynomials,

$$U = p(x) \in F[x], \qquad m_i = (x - x_i), \qquad \text{and} \qquad u_i = p(x_i).$$

For integers,

$$U \in Z, \qquad m_i \in Z, \qquad u_i \in Z/(m_i).^2$$

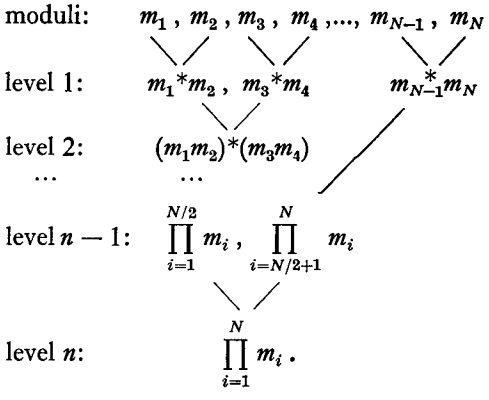
In the previous section, we reduced the evaluation problem to the problem of division in the domain. In fact, we can formalize our development in

THEOREM 1. *Given N moduli $m_i \in D$ and $U \in D$ where* $\mathrm{prec}(U) = N$, *if multiplication and division of N precision elements can be performed in $O(N \log^a N)$ operations, then the N residues $\{u_i\}$, of U, with respect to $\{m_i\}$ can be computed in $O(N \log^{a+1} N)$ steps, where $a \geqslant 0$.*

Proof. We shall give a constructive proof in the form of an algorithm to perform the computation. However, first we need an algorithm to build up the moduli M_i as required in (4.2). We shall state this in the form

LEMMA. *Under the assumptions of Theorem* 1, *the moduli M_i may be built up in a total of $O(N \log^{a+1} N)$ operations.*

Proof. Taking note of our remarks on fast algorithms, we shall try to balance the precision of the multiplication. It follows then that a binary treelike process as illustrated below will be fast. Assuming $N = 2^n$, we have



A simple iterative algorithm (which we shall call construct moduli) can be produced to implement this scheme. We shall not give any further details since the necessary subscripting would only obscure the binary treelike structure of the scheme.

We shall call the products

$$M_{jk} = \prod_{i=j}^{k} m_i \tag{5.1}$$

[2] $Z/(m_i)$ is the ring of integers modulo m_i .

formed in this manner "supermoduli". Let CM(N) be the time to compute construct moduli for N moduli; this is specified by the recurrence relation:

$$\begin{aligned}\mathrm{CM}(N) &= 2\mathrm{CM}(N/2) + M(N)\\ &= 2\mathrm{CM}(N/2) + O(N \log^a N).\end{aligned}$$

By the remarks of Section 2, $\mathrm{CM}(N) = O(N \log^{a+1} N)$.

In the polynomial case, we can be more detailed in our analysis. If we let $N = 2^n$, then at the j-th level of the scheme we form $2^n/2^j$ products of degree 2^j, for $0 \leqslant j \leqslant n$.

Thus, the total cost is

$$\begin{aligned}\mathrm{CM}(2^n) &= \sum_{j=0}^{n} M(2^j)\, 2^n/2^j\\ &= 2^n \sum_{j=0}^{n} (9/2 \cdot 2^j j + 5 \cdot 2^j + \mathrm{ldt})/2^j\\ &= 5 \cdot 2^n n + (9/2) \cdot 2^n \cdot (n(n+1)/2) + \mathrm{ldt}\\ &= 3 \cdot 2^n n^2 + (9/2) \cdot 2^n n + \mathrm{ldt}, \qquad (5.2)\end{aligned}$$

and therefore

$$\mathrm{CM}(N) = (9/4)N \log^2 N + (29/4)N \log N + \mathrm{ldt}.$$

For the Ostrowski measure of computation we have

$$\mathrm{CM}(2^n) = \sum_{j=0}^{n} M(2^j)\, 2^n/2^j = 2^n n$$

and

$$\mathrm{CM}(N) = N \log N. \qquad (5.3)$$

The coefficients of $M_{1,N}$ can be thought of as the elementary symmetric functions $\{\sigma_j(x_1, \ldots, x_N),\ 1 \leqslant j \leqslant N\}$ of the roots $\{x_i\}$ of the moduli $(x - x_i)$. Strassen [23] shows that a lower bound for the multiplicative complexity for this computation is $N \log (N/e)$. Thus, it follows that the complexity of computing the supermoduli is $N \log (N/e)$ which is met by the construct-moduli algorithm, to within an additive term. Note that for the integer case, this algorithm requires $O(N \log^2 N \log \log N)$ steps.

Now that we have the moduli, we can state the following recursive algorithm for computing the residues.

ALGORITHM. *Modular Form* (U, j, k).

Input: (1) *the requisite supermoduli* M_{jk},

(2) *the element* U *where* $\mathrm{prec}(U) \leqslant k - j + 1$.

Output: *the residues* $u_i \equiv U \bmod m_i$, $j \leqslant i \leqslant k$.

Step

(1) *Basis*: *If* $j = k$, *then begin*
output (U);
return;
end;

(2) *Division*: $e := \lfloor (j + k - 1)/2 \rfloor$; $f := e + 1$;
$R_1 := U \text{ rem } M_{je}$;
$R_2 := U \text{ rem } M_{fk}$;

(3) *Recursion*: *Call Modular Form* (R_1, j, e);
Call Modular Form (R_2, f, k);

(4) *Return*:

The algorithm is invoked as Modular Form $(U, 1, N)$.

For the analysis of the timing of the algorithm, let $E(N)$ be the time to compute the N residues of an N precision element. If we assume that division with remainder requires $R(N) = O(N \log^a N)$ steps, then $E(N)$ is defined by

$$\begin{aligned} E(N) &= 2E(N/2) + 2R(N) \\ &= 2E(N/2) + 2O(N \log^a N), \end{aligned}$$

and so $E(N) = O(N \log^{a+1} N)$ by the results of Section 2. Thus, the two steps of the evaluation algorithm require $O(N \log^{a+1} N)$, establishing the theorem.

As a special case we have

COROLLARY 1. *The* N *residues of an* N *precision integer with respect to* N *single precision moduli can be computed in* $O(N \log^2 N \log \log N)$ *steps.*

Proof. The algorithms given in Theorem 1 are easily carried over into the integer setting, since integer division can be defined as

$$P = M^* Q + R, \qquad 0 \leqslant R \leqslant M.$$

If $m_i \mid M$,

$$P \bmod m_i \equiv R \bmod m_i,$$

but R will be sufficiently smaller than P, and therefore the algorithm follows.

For the necessary division, Knuth ([11, pp. 275]) gives a method due to Cook for fast division of integers which has the same bound as the fast multiplication of integers, i.e., $O(N \log N \log \log N)$. This gives a bound for both of the algorithms used in Theorem 1 of $O(N \log^2 N \log \log N)$.

6. FAST POLYNOMIAL DIVISION

The above algorithm requires a fast algorithm for dividing polynomials in order to be effective in the polynomial setting. The division

$$U(x) = V(x)^* Q(x) + R(x), \tag{6.1}$$

where $\deg(U) = N$, $\deg(V) = K$, $\deg(R) < K$, $\deg(Q) = N - K$, classically requires $2K(N - K + 1)$ steps which in the worst case $K = N/2$ is $O(N^2)$. However, we are looking for an algorithm with a bound of the form $O(N \log^a N)$. Without loss of generality, the method can be to compute the quotient separately, then the remainder $R(x)$ can be obtained in one multiplication and one subtraction.

The observations on fast algorithms indicate that one approach could be to segment the problem into two simpler problems; namely, by computing the quotient in two parts.

This divison algorithm would give rise to a polynomial evaluation algorithm of order $O(N \log^3 N)$ (as is shown in [14]). However, Strassen [24] has shown that the remainder of a polynomial of degree N divided by a polynomial of degree $N/2$ can be performed in $(9/2)\, M(N) + N$ steps[3] (see Corollary 2). This algorithm uses another algorithm due to Seiveking [23] which computes the power series division for two power series of degree K in $7M(K)$ steps. Therefore,

$$\begin{aligned} R(N) &= (9/2)\, M(N) + N \qquad \text{total operation} \\ &= (81/4)N \log N + (47/2)N, \end{aligned}$$

or using the Ostrowski measure, $R(N) = (9/2)N$. Now we can make a more detailed analysis of the Modular-Forms algorithm in order to determine the constants involved. Let $E(N)$ be the time to evaluate a polynomial of degree $N - 1$ at N points. Then expanding as before,

$$E(N) = 20\tfrac{1}{4} N \log^2 N + 67\tfrac{1}{4} N \log N + \text{ldt}.$$

If we include the time for construct moduli, the total time for the evaluation is

$$\begin{aligned} E_{\text{Tot}}(N) &= 22\tfrac{1}{2} N \log^2 N + 74\tfrac{1}{2} N \log N + \text{ldt} \\ &= O(N \log^2 N). \end{aligned}$$

We can also analyze the time for evaluation under the Ostrowski model:

$$E(2^n) = 9 \cdot 2^n \cdot n = 9N \log N.$$

This leads us to

COROLLARY 2. *A polynomial of degree* $N - 1$ *can be evaluated at* N *distinct points in* $22(1/2)N \log^2 N + O(N \log N)$ *total arithmetic steps and* $10N \log N$ *multiplications.*

[3] Kung independently developed an $O(N \log N)$ algorithm for "preconditioned" division and the associated results of Corollary 2 and Theorem 4. Schoenhage also independently exhibited in $O(N \log N)$ polynomial division algorithm, and a more abstract development for fast division in a Euclidean domain and can be found in Moenck [15].

Strassen [24] shows that the number of multiplications required for the many point evaluation of a general Nth degree polynomial is $N \log N$. From this, it follows that the Modular-Form algorithm comes within an order of magnitude of the multiplicative bound for the process. In particular, the multipoint evaluation of the special polynomial x^N requires $N \log (N/e)$ multiplications. While the factor algorithm (cf. Brauer [2] or Knuth [11, pp. 398–418]) applied N times is optimal to within an additive term, the Modular-Forms algorithm comes within a constant multiple of the multiplicative bound.

Throughout the presentation, we have assumed that the polynomial moduli are linear, i.e., we are evaluating polynomials. Obviously, the same algorithms will work successfully and just as "efficiently" for higher-order moduli.

7. Fast Interpolation

This is the inverse operation of computing the Modular Forms as given in the previous sections. As mentioned in the Introduction, interpolation of a polynomial is a special case of the CRA over a Euclidean domain D. In our generalized framework, we can characterize the problem as follows:

Given N single precision moduli $\{m_i\}$ and N residues, we wish to compute the unique $U \in D$ such that

$$U \equiv u_i \bmod m_i, \qquad 0 \leqslant \operatorname{prec}(U) \leqslant N.$$

Thus, when we say fast interpolation, we shall in fact mean a fast CRA for a Euclidean domain which has a fast multiplication algorithm.

The classical Lagrangian and Newtonian algorithms for polynomial interpolation and the CRA require $O(N^2)$ operations (cf. Lipson [12]). Horowitz and Heindel [9] have given a method for computing the integer CRA which has a bound of $O(N \log^2 N \log \log N)$ as a preconditioned algorithm and $O(N \log^3 N \log \log N)$ as a complete algorithm.

Horowitz [10] has given a preconditioned polynomial interpolation algorithm which requires $O(N \log^3 N)$ steps. Horowitz also demonstrated that the approach to speeding the interpolation up is to appropriately factor the interpolation formula. Using a somewhat different factoring, we will now synthesize a general interpolation algorithm.

We shall start with the familiar version of the Lagrangian Interpolation Formula:

$$U(x) = \sum_{k=1}^{N} u_k L_k, \tag{8.1}$$

where L_k are the Lagrangian interpolating polynomials:

$$L_k = \left(\prod_{\substack{i=1 \\ i\neq k}}^{N} (x - x_i)\right) \Big/ \left(\prod_{\substack{i=1 \\ i\neq k}}^{N} (x_k - x_i)\right).$$

If we set

$$a_k = 1 \Big/ \prod_{\substack{i=1 \\ i\neq k}}^{N} (x_k - x_i),$$

then we can rewrite (8.1) as

$$U(x) = \sum_{k=1}^{N} u_k a_k \left(\prod_{\substack{i=1 \\ i\neq k}}^{N} (x - x_i)\right).$$

Examining the terms in the summation as the subscript runs from $k = 1$ to N, we see that most of the polynomial terms are duplicated. Taking note of our remarks on fast algorithms, we are looking for a Divide and Rule method which uses balanced precision multiplication. The necessary simplification is to separate the summation into two parts each of length $N/2$ and factor out all of the common polynomial terms from each part, i.e.,

$$\begin{aligned} U(x) = {} & \left(\prod_{i=N/2+1}^{N} (x - x_i)\right) * \left(\sum_{k=1}^{N/2} u_k a_k \left(\prod_{\substack{i=1 \\ i\neq k}}^{N/2} (x - x_i)\right)\right) \\ & + \left(\prod_{i=1}^{N/2} (x - x_i)\right) * \left(\sum_{k=N/2+1}^{N} u_k a_k \left(\prod_{\substack{i=N/2+1 \\ i\neq k}}^{N} (x - x_i)\right)\right). \end{aligned} \tag{8.2}$$

If $N = 2^n$, then the formula now involves two "interpolations" with 2^{n-1} terms and two polynomial multiplications of balanced degree and one addition. This gives us the essentials of a recursive algorithm.

In the more general context of Euclidean domains (cf. Lipson [12]), the Lagrangian formula for the moduli m_i and residues u_i is

$$U = \sum_{k=1}^{N} u_k L_k, \tag{8.3}$$

where the Lagrangian L_k is

$$L_k = \left(\prod_{\substack{i=1 \\ i\neq k}}^{N} m_i\right) \left(\prod_{\substack{i=1 \\ i\neq k}}^{N} s_{ik} |_{m_k}\right),$$

and the s_{ik} are the inverses of the moduli m_i with respect to m_k, i.e.,

$$s_{ik}m_i \equiv 1 \bmod m_k .$$

Here we make the substitution

$$a_k = \left(\prod_{\substack{i=1 \\ i \neq k}}^{N} s_{ik} \right) \bmod m_k ,$$

and we would write the analogue of (8.2) as

$$U = M_1 * \left(\sum_{k=1}^{N/2} u_k a_k \left(\prod_{\substack{i=1 \\ i \neq k}}^{N/2} m_i \right) \right) + M_2 * \left(\sum_{k=N/2+1}^{N} u_k a_k \left(\prod_{\substack{i=N/2+1 \\ i \neq k}}^{N} m_i \right) \right), \tag{8.4}$$

where

$$M_1 = \prod_{i=N/2+1}^{N} m_i , \quad M_2 = \prod_{i=1}^{N/2} m_i .$$

Note that these powers of moduli are none other than the supermoduli computed in Section 5.

8. Preconditioned Interpolation

The foregoing analysis leads us to the conclusion that preconditioned polynomial interpolation is "reducible" to polynomial multiplication (in the same sense that polynomial evaluation is reducible to polynomial division).

Theorem 3. *If multiplication in the domain can be performed in $O(N \log^a N)$ steps, and the constants a_k can be preconditioned, then the generalized* CRA *from an N precision element can be performed in $O(N \log^{a+1} N)$ steps.*

Proof. We can use the following algorithm for the process.

Algorithm: *Interp* $(u_i , e \leqslant i \leqslant f)$.

Input: (1) *the residues* $u_i , e \leqslant i \leqslant f$, *to be interpolated,*

(2) *the constants* a_k,

(3) *the supermoduli* $M_{jk} = \prod_{i=j}^{k} m_i$.

Output: *the interpolated value of* U.

Step

(1) *Basis*: *If* $e = f$, *then return* $(u_e a_e)$;

(2) *Recursion*: $k := \lfloor (e + f - 1)/2 \rfloor$;
$j := k + 1$;
$U_1 := Interp\ (u_i, e \leqslant i \leqslant k)$;
$U_2 := Interp\ (u_i, j \leqslant i \leqslant f)$;

(3) *Multiplication*: *Return* $(U_1{}^*M_{jf} + U_2{}^*M_{ek})$;

The algorithm will be invoked as $Interp\ (u_i, 1 \leqslant i \leqslant N)$.

Let $I(N)$ be the time to interpolate N values; then

$$\begin{aligned} I(N) &= 2I(N/2) + M(N) \\ &= 2I(N/2) + O(N \log^a N) \\ &= O(N \log^{a+1} N) \end{aligned}$$

by the results of Section 2, and the theorem is established.

9. Computing the a_k

In order to have a complete interpolation algorithm, we need to be able to compute the constants a_k. We first look at the polynomial domain which affords a certain simplification.

In the previous section, we made the substitution

$$a_k = 1 \Big/ \left(\prod_{\substack{i=1 \\ i \neq k}}^{N} (x_k - x_i) \right).$$

In a more familiar form,

$$a_k = 1/M'_{1N}(x)\,|_{x=x_k}, \tag{10.1}$$

where $M_{1N}(x) = \sum_{i=1}^{N} (x - x_i)$ and $M'_{1N}(x)$ is its derivative. $M_{1N}(x)$ is a by product of the construct-moduli algorithm which must be invoked to do the evaluation of (10.1). This reduces the general interpolation problem for a polynomial to the evaluation problem. This means we can state

Theorem 4. *The interpolation of a polynomial of degree* $N - 1$ *at* N *points can be performed in* $36N \log^2 N + O(N \log^2 N)$ *total steps or* $12N \log N + N$ *multiplications.*

Proof. Using the results of Theorem 3, we have that the preconditioned interpolation can be performed in

$$I(N) = 2I(N/2) + 2M(N).$$

Assuming $N = 2^n$,

$$I(N) = 4\tfrac{1}{2}N \log^2 N + 15\tfrac{1}{2}N \log N + \text{ldt}.$$

From Theorem 2, we have that a polynomial can be evaluated in $22(1/2)N \log^2 N + 74(1/2)N \log N$ steps. Since the evaluation will supply the supermoduli required by the Interp algorithm, the total is

$$I_{\text{tot}}(N) = 27N \log^2 N + 90N \log N + \text{ldt}.$$

Within the Ostrowski model of computation, we see from Theorem 3 that the time to interpolate a preconditioned polynomial is

$$\begin{aligned} I(N) &= 2I(N/2) + 2M(N) \\ &= 2N \log N + N. \end{aligned}$$

Using the results of Section 8, the total time to interpolate a polynomial of degree $N - 1$ is

$$I_{\text{tot}}(N) = 12N \log N + N.$$

Strassen [24] shows that interpolation has a lower bound of $(N + 1) \log N$ multiplications for its complexity within the Ostrowski model. This means that the multiplicative complexity of interpolation is of the order of magnitude of $N \log N$.

In the integer case or in the more general setting of a Euclidean domain D, we have that

$$a_k \equiv \left(\prod_{\substack{i=1 \\ i \neq k}}^{N} s_{ik} \right) \bmod m_k ,$$

where

$$1 \equiv (s_{ik} m_i) \bmod m_k . \tag{10.2}$$

We note that a_k is a single precision element of D and from congruence properties we have that

$$1 \equiv \left(\prod_{\substack{i=1 \\ i \neq k}}^{N} m_i \right) \left(\prod_{\substack{i=1 \\ i \neq k}}^{N} s_{ik} \right) \bmod m_k$$

by commutativity of (10.2). This implies that there exists a b_k in a coset of the ideal (m_k) such that

$$1 \equiv b_k a_k \bmod m_k , \tag{10.3}$$

where

$$b_k \equiv \left(\prod_{\substack{i=1 \\ i \neq k}}^{N} m_i \right) \bmod m_k$$

Therefore, a_k is a unit in the ring $Z_{m_k} = Z/(m_k)$. We can compute the $\{a_k\}$ from the $\{b_k\}$ using the standard extended Euclidean algorithm. Since each m_k and b_k are single precision, each computation only requires a constant number of operations or $O(N)$ operations to compute all the $\{a_k\}$ from the $\{b_k\}$. From the construct-moduli algorithm, we get

$$M_{1N} = \prod_{i=1}^{N} m_i \,.$$

Since the moduli m_i must be pairwise relatively prime M_{1N} lies in the ideal (m_k) but not in (m_k^2). Since if

$$\left(\prod_{\substack{i=1 \\ i \neq k}}^{N} m_i \right) = d \cdot m_k + b_k \,,$$

we have that

$$M_{1N} = d \cdot m_k^2 + b_k m_k \,,$$

and so $b_k m_k \equiv M_{1N} \bmod m_k^2$.
If we can compute

$$c_k \equiv M_{1N} \bmod m_k^2,$$

then

$$b_k = c_k / m_k \qquad \text{dividing exactly}$$

and

$$a_k = b_k^{-1} = (c_k/m_k)^{-1}.$$

Therefore, we can compute the constants a_k from the c_k in $O(N)$ steps using exact division and the c_k can be computed using the modular-forms algorithm in $O(N \log^2 N \log \log N)$ double precision steps. We have reduced the interpolation problem in the general setting to the evaluation problem. This gives us

THEOREM 5. *The complete integer* CRA *can be computed in* $O(N \log^2 N \log \log N)$ *total steps.*

and

THEOREM 6. *If division $R(N)$ and multiplication $M(N)$ in the Euclidean domain D can be performed in $O(N \log^a N)$ steps, then the complete* CRA *in D can be performed in $O(N \log^{a+1} N)$ total steps.*

Again in the sections on interpolation we have focused our attention on linear moduli for the polynomial case. This is because they are the most frequently encountered forms. As before, the algorithms operate with equal facility on moduli of higher degree.

10. EVALUATING A POLYNOMIAL AND ALL ITS DERIVATIVES

One further problem which can be "efficiently" solved using these algorithms is that of evaluating a polynomial and all its derivatives at a point. Shaw and Traub [22] have shown that this can be done for a polynomial of degree N in $O(N)$ multiplications. Since we know that just the evaluation of the polynomial requires N multiplications (cf. Pan [18]), this method is asymptotically optimal. However, their algorithm requires $O(N^2)$ additions. The following method can be used to evaluate a polynomial and all its normalized derivatives in $O(N \log^2 N)$ total operations. Yet it should be noted that their method is reasonably practical whereas ours is not practical.

First, we note that a polynomial of degree N can be defined by its Taylor series about a point $x = a$, i.e.,

$$p(a + h) = p(a) + \frac{h}{1!} p'(a) + \frac{h^2}{2!} p''(a) + \cdots + \frac{h^N}{N!} p^{(N)}(a). \tag{11.1}$$

So we can evaluate the polynomial $p(x)$ at $N + 1$ points $\{a + h_i\}$ for distinct $\{h_i\}$ to obtain the values $\{p(a + h_i)\}$. The series in (11.1) can now be regarded as a polynomial in h, and we can compute that polynomial which interpolates the pairs of values $\langle p(a + h_i), h_i \rangle$. The coefficients of this polynomial will then be the values of the polynomial and all its normalised derivatives at the point $x = a$, i.e.,

$$p(a), \frac{p'(a)}{1!}, \frac{p''(a)}{2!}, \ldots, \frac{p^{(N)}(a)}{N!}.$$

This gives us the following:

THEOREM 7. *The values of a polynomial of degree N and all its normalized derivatives at a point can all be computed in $O(N \log^2 N)$ total steps.*

Proof. The evaluation and interpolation each require $O(N \log^2 N)$ total steps.

11. Conclusion and Open Questions

We should say that while the algorithms which have been presented are for the moment more of theoretical interest, they may not be hopelessly beyond the bounds of practicality. In our analysis of the polynomial algorithms, we made assumptions about the time to do polynomial multiplication. The constants of proportionality involved in the multiplication and division have a considerable impact on the constants for the new algorithms. Thus, if any improvement could be made to the multiplication or division algorithms, it would also affect the others.

In our reckoning of the time for the algorithms, we counted only the arithmetic operations and excluded any implementation-dependent operations such as storage accesses. Obviously, in the real world, such things have a substantial effect on the constants of proportionality, but it may not be too unreasonable to assume that they affect all methods equally. Also, for any practical implementation, the algorithms would have to be recast from their recursive to an iterative format.

Table I shows the prohibitively high cross-over points for the timing functions of the classical versus the new algorithms. While the table does not give the exact cross-over, experimental results have been obtained which tend to confirm the table entries.

TABLE I[a]

Timing functions in terms of N	N $256 = 2^8$	$512 = 2^9$	$1024 = 2^{10}$	$2048 = 2^{11}$	$4096 = 2^{12}$
Classical polynomial division: $N/2$	128	250			
Fast polynomial division: $20\frac{1}{4} \log N + 67\frac{1}{4}$	230	250			
Classical many-point evaluation and interpolation: $2N$	512	1024	2048	4096	8192
Fast preconditioned interpolation: $4\frac{1}{2} \log^2 N + 15\frac{1}{2} \log N$	268				
Fast many-point evaluation: $22\frac{1}{2} \log^2 N + 74\frac{1}{2} \log N$			2995	3542	
Fast interpolation: $27 \log^2 N + 90 \log N$				4257	4968

[a] In order to simplify the table, only those powers of 2 above and below the cross-over have been filled in. In order to avoid numbers too large, N has been factored out of all of the timing functions. The prefix "fast" means assmyptotically fast.

We should also note the theoretically interesting open problems in the area. The fundamental theoretical problem has to do with the discrepancy of the log factor between the multiplicative and total arithmetic complexity of the algorithms. It would seem that computing the Fourier transform of length $2N$ given the transform of length N of a sequence of length N is a fairly easy operation to perform. It would not appear unreasonable to conjecture that this process which we call "transform doubling" could be performed in $O(N)$ arithmetic operations. If this were true then, the preconditioned Interp algorithm and the construct-moduli algorithm could be performed in $O(N \log N)$ total arithmetic operations.

In addition, it would seem possible that division with remainder might be performed on polynomials in transformed form in $O(N)$ arithmetic operations, much in the way multiplication is presently done. This together with the conjecture on transform doubling would imply that the complete Modular-Form and Interp algorithms could be performed in $O(N \log N)$ total arithmetic operations.

On the other hand, we have Morgenstern's [16] result that the FFT considered as a "linear algorithm with bounded scalars" requires $N \log (N/e)$ additions or subtractions. It would be remarkable if multipoint evaluation of N arbitrary points could be done in the same order of magnitude of time as evaluation at the very special set of points, the primitive roots of unity.

A practical open problem is to reduce the constants of proportionality for polynomial multiplication and division and thus for all of the algorithms, as described above.

As a final summary, we can state the results of this paper for the polynomial domain in terms of reducibilities. We shall say a process is directly reducible to another process if the former process can be computed using the latter in the same order of magnitude of time. We shall say a process is log reducible to another process if it is a log factor slower than another process it uses. Then we have:

(1) Evaluating a polynomial and all its derivatives is directly reducible to interpolation and many-point polynomial evaluation.

(2) Polynomial interpolation is directly reducible to many-point polynomial evaluation and preconditioned polynomial interpolation.

(3) Preconditioned polynomial interpolation is log reducible to polynomial multiplication.

(4) Many-point polynomial evaluation is log reducible to polynomial division.

(5) Polynomial division is directly reducible to polynomial multiplication (Strassen [24]).

Analogous results hold for general Euclidean domains.

Specifically, using Strassen's $O(N \log N)$ division, a polynomial of degree $N - 1$ can be evaluated at N points in $O(N \log^2 N)$ total steps or $O(N \log N)$ multiplications. The multiplicative bound is within a constant multiple of optimality. Using the same

method, the residues of an N precision integer can be computed in $O(N \log^2 N \log \log N)$ total steps.

We have also shown that the $N - 1$ degree polynomial interpolating N points can be computed in $O(N \log^2 N)$ total operations or $12N \log N + N$ multiplications. Again, the multiplicative bound is within a constant multiple of optimality. Using a related method, the integer CRA can be computed in $O(N \log^2 N \log \log N)$ total operations. Using these algorithms, we have shown that a polynomial and all its derivatives can be evaluated at a point in $O(N \log^2 N)$ total operations.

ACKNOWLEDGMENT

We thank Prof. J. Lipson for introducing us to modular techniques and for his many helpful suggestions.

REFERENCES

1. A. BORODIN AND I. MUNRO, Evaluating polynomials at many points, *Information Processing Letters* **1**, No. 2 (1971).
2. A. BRAUER, On addition chains, *Bull. Amer. Math. Soc.* **45** (1939), 736–739.
3. W. S. BROWN, On Euclid's algorithm and the computation of polynomial greatest common divisors, *J. Assoc. Comput. Mach.* **18**, No. 4 (1971).
4. S. CABAY, Exact solution of linear equations, Proc. of the 2nd Symp. on Symbolic and Algebraic Manipulation, March 1971.
5. G. E. COLLINS, The calculation of multivariate polynomial resultants, *J. Assoc. Comput. Mach.* **18**, No. 4 (1971).
6. C. FIDUCCIA, Fast matrix multiplication, Proc. Third Annual ACM Symposium on Theory of Computation, May 1971, pp. 45–49.
7. C. FIDUCCIA, Polynomial evaluation via the division algorithm: The fast Fourier transform revisited, Proc. 4th Symp. on Theory of Computing.
8. J. HOPCROFT AND L. KERR, On minimizing the number of multiplications necessary for matrix multiplication, *SIAM J. Appl. Math.* **20**, No. 1 (1971).
9. E. HOROWITZ AND L. HEINDEL, On decreasing the computing time for modular algorithms, Proc. of the 12th Symp. on Switching and Automata Theory, Oct. 1971.
10. E. HOROWITZ, A fast method for interpolation of polynomials using preconditioning, *Information Processing Letters* 1, No. 4 (1972).
11. D. KNUTH, "The Art of Computer Programming," Vol. 2, Addison–Wesley, 1969.
12. J. LIPSON, Chinese remainder and interpolation algorithms, Proc. of the 2nd Symp. on Symbolic and Algebraic Manipulation, March 1971.
13. J. LIPSON, private communication.
14. R. MOENCK AND A. BORODIN, Fast modular transforms via division, Proc. of the 13th Annual Symp. on Switching and Automata Theory, Oct. 1972.
15. R. MOENCK, Studies in fast algebraic algorithms, Ph.D. Thesis, University of Toronto, 1973.
16. J. MORGENSTERN, Note on a lower bound of the linear complexity of the fast Fourier transform, *J. Assoc. Comput. Mach.* **20**, No. 2 (1973).
17. A. OSTROWSKI, On Two Problems in Abstract Algebra Connected with Horner's Rule, "Studies Presented to R. von Mises," Academic Press, New York, 1954, pp. 40–48.

18. V. PAN, Methods of computing values of polynomials, Russian Math. Surveys **21**, No. 1 (1966).
19. J. POLLARD, The fast Fourier transform in a finite field, *Math. Comp.* **25**, No. 114 (1971).
20. A. SCHOENHAGE AND V. STRASSEN, Fast multiplication of large numbers, *Computing* **7** (1971), 281–292.
21. A. SCHOENHAGE, Schnelle Berechnung von Kettenbruchentwicklungen, *Acta Informatica* **1**, No. 1 (1971).
22. M. SHAW AND J. F. TRAUB, On the number of multiplications for the evaluation of a polynomial and its derivatives, Proc. of the 13th Annual Symp. on Switching and Automata Theory, Oct. 1973.
23. M. SIEVEKING, An algorithm for division of power series, *Computing* **10**, No. 1–2 (1972), 153–156.
24. V. STRASSEN, Die Berechnungskomplexität von elementarsymetrischen Funktionen und von Interpolationskoeffizienten, *Numer. Math.* **20**, No. 3 (1973), 238–251.
25. S. WINOGRAD, On the number of multiplications necessary to compute certain functions, *Comm. Pure Appl. Math.* **23** (1970).