FAST MODULAR TRANSFORMS VIA DIVISION

R. Moenck and A. Borodin

Department of Computer Science
University of Toronto
Toronto, Canada

Abstract

It is shown that the problem of evaluating an Nth degree polynomial is reducible to the problem of dividing the polynomial. A method for dividing an Nth degree polynomial by an N/2 degree polynomial in $O(N \log^2 N)$ steps is given. Using this it is shown that the evaluation of an Nth degree polynomial at N points can be done in $O(N \log^3 N)$. The related problem of computing the resides of an N precision integer is handled by the same algorithm in $O(N \log^2 N \log\log N)$ steps.

Using the methods of Horowitz[11] and Heindel[8] it is shown that interpolation of an Nth degree polynomial is reducible to the problem of evaluating an Nth degree polynomial at N points. An algorithm for preconditioned polynomial interpolation requiring $O(N \log^2 N)$ steps is presented. This is then extended to perform the complete interpolation in $O(N \log^3 N)$ steps. A modified version of this algorithm is shown to compute the Chinese Remainder Problem in $O(N \log^2 N \log\log N)$ steps.

1. INTRODUCTION

A common feature of many of the efficient algorithms recently developed for polynomial and number theoretic computation is the 'modular process'. This process involves reducing a problem to a simpler form, by computing with sample values (homomorphic images) of the original problem. We can represent the process schematically as:

$$\begin{tikzcd} A \arrow[r, "f_A"] \arrow[d, "\phi"'] & A' \\ B \arrow[r, "f_B"] & B' \arrow[u, "\phi'"'] \end{tikzcd}$$

where A and A' are the input and results respectively, of performing the operation f_A on the original problem. B,B' and f_B are the input, results and operation for the sampled problems, with mappings ϕ and ϕ' between the problem spaces. Examples of this process when used for linear equations, polynomial GCDs and polynomial resultants are given by Cabay[1], Brown[2] and Collins[3] respectively. It is the transformations ϕ and ϕ' which we shall investigate with a view to looking for fast algorithms.

In the polynomial case the transformation ϕ involves evaluating a polynomial at many points and ϕ' interpolates the results of B'. The transformations ϕ and ϕ' could be performed using the Fast Fourier Transform (FFT) (cf Polard[4]). However, in some algorithms, situations occur where certain sample values must be discarded (cf Brown[2] and Collins[3]). This implies that the FFT cannot be simply used, since it depends on a strict relationship between the sample points. This leaves us with the interpolation problem and it's dual; the problem of evaluating a polynomial at many points.

The analogue of the interpolation problem in the number theoretic (integer) case is the 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 moduli. This can be thought of as 'interpolating' an integer, given it's residues. Lipson[5] shows that interpolation and the CRA are abstractly equivalent to the Chinese Remainder Problem over a Euclidean Domain D. He also gives a thorough exposition of the classical algorithms for these problems.

In view of this equivalence, we shall refer to the process of evaluating a polynomial at many points and computing the residues of an integer as computing the Modular Form of an element in a Euclidean Domain.

2. A NOTE ON FAST ALGORITHMS

A common property of fast $(N \log^a N)$ algorithms is that they reduce a problem to a simpler one by dividing it into two problems, each of which is half as difficult as the original problem. This lets the time analysis be defined by a recurrence relation of the form:

$$T(2^n) = 2*T(2^{n-1}) + \ldots$$

and implies a recursive algorithm.

Another property of such algorithms, which depend on a fast multiplication scheme, is that the size of the elements being multiplied must be approximately the same. This follows from the fact that both fast and classical multiplications require N operations to multiply an N precision element by a single precision element. Examples of this balanced precision multiplication are seen in Schonhage[6], and in Horowitz and Heindel[7]. In fact Horowitz and Heindel show that the classical algorithm for the CRA is not improved by using fast multiplication.

3. FAST MODULAR FORMS

Classically the evaluation of an Nth degree polynomial at N points requires $O(N^2)$ operations. This is performed by doing N evaluations of the polynomial at 1 point. A similar bound holds for the computation of N single precision residues of an N precision integer. Borodin and Munro[8] have shown that polynomial evaluation can be done in $O(N^{1.91})$ using Strassen's fast matrix multiplication.

Evaluating a polynomial at one point can be thought of as a division process (cf Knuth[9] p 424). This follows from the remainder theorem.

i.e. Given a polynomial $p(x)$, if we divide by $x-a$ we get

$$p(x) = q(x)*(x-a) + r(x)$$

where $\deg(r)=0$ (i.e. a constant). Putting $x=a$ we get that $p(a)=r$.

This suggests a generalisation to more points. If we wish to evaluate $p(x)$ at m points x_i we form:

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

and divide $p(x)$ by $M(x)$:

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

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 our problem to a simpler one. Fiduccia[10] 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 $p(x)$ of degree $N-1$ at N points, suggests itself. First we form a polynomial $M_1(x)$ with the first $N/2$ points as 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 get $R_2(x)$ in a similar manner. This gives us two polynomials of degree $N/2-1$, each of which are to be evaluated at $N/2$ points. To do this we use the method recursively.

e.g. 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

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

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

$$p(-1) = 7, \quad p(1) = 3$$

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

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

4. SOME NOTATION

In order to discuss this problem further, we make the following characterisation of the problem. Given a set of N moduli $\{m_i\}\epsilon D$ and $U\epsilon D$ we wish to compute the set of residues $u_i \epsilon D$ such that:

$$u_i \equiv U \bmod m_i, \quad 1\le i\le N$$

For polynomials: $U = p(x)\epsilon F[x]$, $m_i = (x-x_i)$
$u_i = p(x_i)$

For integers: $U\epsilon Z$, $m_i \epsilon Z$, $u_i \epsilon Z/(m_i)$

In addition we shall use a precision function defined as:

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

5. THE ALGORITHM

We can formalise the results in the following:

Theorem 1: Given N moduli m_i and $U\epsilon D$ where $\text{prec}(U)=N$. If division in D requires $R(N)=O(N \log^a N)$ then the residues u_i are computable in $E(N)=O(N \log^{a+1} N)$.

Proof: We shall give a constructive proof in the form of an algorithm to perform the computation. However we first require an algorithm to build up the moduli M_i. We can best illustrate the process schematically as follows, assuming $N=2^n$.

$$\begin{tikzcd}
\text{Moduli:} & m_1, \arrow[dr, no head] & & m_2 \arrow[dl, no head] & m_3, \arrow[dr, no head] & & m_4 \arrow[dl, no head] & ,\dots & m_{N-1}, \arrow[dr, no head] & & m_N \arrow[dl, no head] \\
\text{Level 1:} & & m_1*m_2, \arrow[dr, no head] & & & m_3*m_4, \arrow[dll, no head] & & \dots, & & m_{N-1}*m_N & \\
\text{Level 2:} & & & (m_1*m_2)*(m_3*m_4),\dots . & & & & & & & \\
\text{Level } n-1: & & \prod_{i=1}^{N/2} m_i, & & & \prod_{i=N/2+1}^{N} m_i & & & & &
\end{tikzcd}$$

We shall call the products $M_{e,f} = \prod_{i=e}^{f} m_i$ formed in this manner 'super-moduli'.

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 tree like structure of the scheme. For the residues we have an:

Algorithm : Modular-Form (U,j,k);

```
   Input : 1) the requsite super-moduli $M_{e,f}$.
           2) the element U for which the
              residues $u_i \equiv U \bmod m_i$, $j\le i\le k$ are
              to be computed, where
              $\text{prec}(U)\le k-j+1$

1) Basis : If j=k then Begin Output(U rem $M_{jj}$);
                             Return ;
                       End;

2) Division : e:=(j+k-1)/2 ;  f:=e+1 ;
              $R_1$:=U rem $M_{j,e}$ ;
              $R_2$:=U rem $M_{f,k}$ ;
```

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

4) Return ;
```

The algorithm will be called initially as Modular-Form (U,1,N).

Analysis

Let $CM(2^n)$ be the time to construct the supermoduli $M_{e,f}$ for 2^n moduli m_i. Then at the jth level of the scheme we form $2^n/2^j$ products of precision 2^j, $0 \le j \le n-1$.

$$CM(2^n) = 2^n + \sum_{j=1}^{n} M(2^j)2^n/2^j$$

$$= 2^n + 2^n \sum_{j=1}^{n} \cancel{2^j} j/\cancel{2^j} = O(2n^2)$$

$$= O(N \log^2 N)$$

Here we have assumed multiplication requires $O(N \log N)$ steps. Let $E(2^n)$ be the time to form the remainders corresponding to 2^n moduli m_i. If we assume the time for division is $R(2^n)=O(2^n n^a)$, then we have the recurrance relation:

$$E(2^n) = 2E(2^{n-1}) + 2R(2^n)$$

$$= 2E(2^{n-1}) + 2*2^n n^a$$

$$= 2^n E(1) + 2*2^n \sum_{i=0}^{n} i^a$$

$$= O(2^n n^{a+1}) = O(N \log^{a+1} N)$$

Note that these algorithms work for integers since integer division can be defined as:

$$P = M*Q + R, \quad 0 \le R < M$$

If M is in the ideal (m_i) then:

$$P + (m_i) = R + (m_i)$$

But the precision of R will be half the precision of P.

Corollary: 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: Knuth[9] p 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. $O(N \log N \log\log N)$ using the Schoenhager-Strassen algorithm[12]. This gives a bound for both of the above algorithms, needed in Theorem 1 which is 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:

$$U(x) = V(x)*Q(x) + R(x) \qquad (6.1)$$

$$\deg(U)=N, \quad \deg(V)=M, \quad \deg(R)\le M, \quad \deg(Q)=N-M$$

classically requires $O(M(N-M+1))$ steps which in the worst case $(M=N/2)$ is $O(N^2)$. One approach to this problem is to look for methods to compute the quotient $Q(x)$. If we can 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 the method should be to segment the problem into two simpler problems. In this case, we compute the quotient in two parts. Since the leading coefficients of the divisor $V(x)$ and dividend $U(x)$ dictate the coefficients of the quotient, we will use the following method:

1) Take the leading half of $U(x)$ and $V(x)$ called $U_2(x)$ and $V_2(x)$.

2) Compute the quotient $Q_1(x)$ of $U_2(x)$ and $V_2(x)$.

3) Multiply $V(x)$ by $Q_1(x)$ and by an appropriate power of x and subtract this from $U(x)$, to form $U'(x)$.

4) From the polynomial $U'(x)$ extract the leading 2/3 coefficients to form a new polynomial $U_2^{1}(x)$.

5) Compute the quotient $Q_2(x)$ of $U_2^{1}(x)$ and $V_2(x)$.

6) The quotient of $U(x)$ divided by $V(x)$ is then $Q_1(x)$ multiplied by an appropriate power of x plus $Q_2(x)$.

For Example

If $U(x) = 2x^7+3x^5-x^4-2x^3-5x+4$

and $V(x) = x^4+x^3-2x^2-5x+4$

then $U_2(x) = 2x^3+3x-1$, $V_2(x) = x^2+x-2$

which when divided gives $Q_1(x) = 2x-2$

so $U'(x) = U(x) - V(x)*Q_1(x)*x^2$

$$= 9x^5+5x^4-16x^3+8x^2-5x+4$$

and $U_2^{1}(x) = 9x^3+5x^2-16x+8$

which when divided by $V_2(x)$ gives $Q_2(x)=9x-4$. Thus the quotient of $U(x)$ divided by $V(x)$ is:

$$Q(x) = Q_1(x)*x^2+Q_2(x) = 2x^3-2x^2+9x-4$$

which can be easily verified by performing the long division.

7. THE DIVISION ALGORITHM

This gives us the following theorem:

Theorem 2: If $\deg(U)=N=2^n-1$ and $\deg(V)=\lceil N/2 \rceil=M=2^{n-1}$ then polynomial division can be performed in $O(N^2 \log N)$.

Proof: Again we shall formulate the proof in terms of an algorithm. We shall assume that, $\deg(U)=N=2^n-1$, $\deg(V)=M=2^{n-1}$. We shall consider this particular case of the degrees for two reasons:

1) it is the worst case for the classical algorithm.

2) it is the case used in the Modular-Form algorithm.

```
Algorithm : Quot (U,N,V,M) ;

  Input : $U(x)$ , $\deg(U)=N$
          $V(x)$ , $\deg(V)=M$

  Output : Quotient polynomial $Q(x)$ ,
                               $\deg(Q)=N-M$

1) Basis : If M=1 then Begin use classical
                                 division;
                             Return (Quotient
                                     Q);
                       End;

2) Recursion : $Q_1$:=Quot($U_2$,$\lceil N/2\rceil -1$,$V_2$,$M/2$);
         where $U_2$ are the leading $\lceil N/2\rceil$ terms
         of U with $x^{\lceil N/2\rceil}$ factored out.
         $V_2$ are the leading $M/2$ terms of V
         with $x^{M/2}$ factored out.

3) Multiplication : $U' := U-Q_1*V*x^{M/2}$;

4) Recursion : $Q_2$:=Quot($U'_2$,$\lceil N/2\rceil -1$,$V_2$,$M/2$);
         where $U'_2$ are the leading $\lceil N/2\rceil$ terms
         of U' with $x^{\lceil N/2\rceil}$ factored out.

5) Return $(Q_1x^{M/2}+Q_2)$ ;
```

We can establish that this algorithm generates the quotient by analysing the degrees of the polynomials involved. We know that polynomial division is unique under the conditions:

$$U = QV + R, \quad \deg(R)<\deg(V) \qquad (6.2)$$

1) If $\deg(U)=2^n-1$ and $\deg(V)=2^{n-1}$ then we can segment U and V as is done in the algorithm:

i.e. $U = U_2x^{2^{n-1}}+U_1$, $\deg(U_2)=2^{n-1}-1$, $\deg(U_1)=2^{n-1}-1$

$V = V_2x^{2^{n-2}}+V_1$, $\deg(V_2)=2^{n-2}$, $\deg(V_1)=2^{n-2}-1$

2) Then we divide U_2 by V_2 to get quotient Q_1:

$$P_1 = U_2-Q_1V_2 \Rightarrow \deg(P_1)=2^{n-2}-1, \deg(Q_1)=2^{n-2}-1$$

3) From this we compute the polynomial U'

$$\begin{aligned}
U' &= U - Q_1Vx^{2^{n-2}}\\
&= (U_2x^{2^{n-1}}+U_1) - (Q_1V_2x^{2^{n-1}}+Q_1V_1x^{2^{n-2}})\\
&= (U_2-Q_1V_2)x^{2^{n-1}} + (U_1-Q_1V_1x^{2^{n-2}})\\
&= P_1x^{2^{n-1}} + (U_1-Q_1V_1x^{2^{n-2}})
\end{aligned}$$

which implies $\deg(U') \le 3*2^{n-2}-1 < 3/4*N$. We segment U' as follows:

$$U' = U'_2x^{2^{n-2}}+U'_1, \quad \deg(U'_2)\le 2^{n-1}-1, \quad \deg(U'_1)=2^{n-2}-1$$

4) Then we compute the quotient Q_2 of U'_2 and V_2

$$P_2 = U'_2-Q_2V_2 \Rightarrow \deg(Q_2)=2^{n-2}-1, \deg(P_2)\le 2^{n-2}-1$$

which gives $Q(x)=Q_1x^{2^{n-2}}+Q_1$
Now to establish the result (6.2) we consider the division

$$\begin{aligned}
R &= U - QV\\
&= (U_2x^{2^{n-1}}+U_1) - (Q_1x^{2^{n-2}}+Q_2)(V_2x^{2^{n-2}}+V_1)\\
&= ((U_2-Q_1V_2)x^{2^{n-1}}+U_1-Q_1V_1x^{2^{n-1}}) -\\
&\qquad (Q_2V_2x^{2^{n-2}}+Q_2V_1)\\
&= U' - (Q_2V_2x^{2^{n-2}}+Q_2V_1)\\
&= (U'_2x^{2^{n-2}}+U'_1) - (Q_2V_2x^{2^{n-2}}+Q_2V_1)\\
&= (U'_2-Q_2V_2)x^{2^{n-2}}+(U'_1-Q_2V_1)\\
&= P_2x^{2^{n-2}} + (U'_1-Q_2V_1)
\end{aligned}$$

$\Rightarrow \deg(R)\le 2^{n-1}-1<\deg(V)$.

<u>Analysis</u>

Let $Q(2^n)$ be the time to divide U by V. This gives us the following recurrance relation:

$$\begin{aligned}
Q(2^n) &= 2Q(2^{n-1}) + n2^n\\
&= 2(2Q(2^{n-2}) + (n-1)2^{n-1}) + n2^n\\
&= 2^nQ(1) + \sum_{i=0}^{n} i2^n\\
&= O(2^nn^2) = O(N\log^2N)
\end{aligned}$$

It can be shown that for general N and M the time to compute the quotient is:

$$Q(N,M) = \begin{cases} O((N-M)\log^2(N-M), & \text{if } 2M>N>M\\ O(N\log^2M), & \text{if } N\ge 2M\end{cases}$$

<u>Corollary</u>: A polynomial of degree $N-1$ can be evaluated at N points in $O(N\log^3N)$.

Proof: Apply Theorem 2 to the results of Theorem 1.

Note that this bound compares quite favourably to the time for the forward FFT which is $O(N\log N)$.

8. <u>FAST INTERPOLATION</u>

This is the inverse operation of those considered in the previous sections. As mentioned in the introduction, interpolation to a polynomial is a special case of the CRA over a Euclidean Domain D. In our generalised framework we can characterise the problem as follows:

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

$$U \equiv u_i \bmod m_i, \quad 0 \le \mathrm{prec}(U) \le N.$$

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 algorithms for polynomial interpolation and integer CRA require $O(N^2)$ operations (cf Lipson[6]). Horowitz and Heindel[7] 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[11] has given an interpolation algorithm which requires $O(N \log^3 N)$ both in it's preconditioned form and as a complete algorithm (if polynomial evaluation can be done in $O(N \log^3 N)$). Horowitz also demonstrated that one approach to speeding the interpolation up, is to appropriately factor the interpolation formula. Using this factoring we will now synthesise these two algorithms into one general algorithm.

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

$$U(x) = \sum_{k=1}^{N} u_k L_k \tag{7.1}$$

where the L_k are the Lagrangian interpolating polynomials:

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

if we set:

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

then we can rewrite (7.1) as:

$$U(x) = \sum_{k=1}^{N} u_k a_k \Big(\prod_{\substack{i=1\\i\ne k}}^{N} (x-x_i)\Big)$$

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. Once again taking note of our remarks on fast algorithms, a simplification suggests itself. This 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.

$$U(x) = \Big(\prod_{i=N/2+1}^{N} (x-x_i)\Big) * \Big(\sum_{k=1}^{N/2} u_k a_k \Big(\prod_{\substack{i=1\\i\ne k}}^{N/2} (x-x_i)\Big)\Big) + \Big(\prod_{i=1}^{N/2} (x-x_i)\Big) * \Big(\sum_{k=N/2+1}^{N} u_k a_k \Big(\prod_{\substack{i=N/2+1\\i\ne k}}^{N} (x-x_i)\Big)\Big) \tag{7.2}$$

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 a more general form (cf Lipson[6]) the Lagrangian formula for the moduli m_i and residues u_i is:

$$U = \sum_{k=1}^{N} u_k L_k \tag{7.3}$$

where the Lagrangian

$$L_k = \Big(\prod_{\substack{i=1\\i\ne k}}^{N} m_i\Big)\Big(\prod_{\substack{i=1\\i\ne k}}^{N} s_i^k \,\Big|_{m_k}\Big)$$

and the s_i^k are the inverses of the moduli with respect to m_k, i.e.

$$s_i^k m_i \equiv 1 \bmod m_k$$

Here we simplify $a_k = \Big(\prod_{\substack{i=1\\i\ne k}}^{N} s_i^k\Big) \bmod m_k$

and we would write (7.2) as:

$$U = M_1 * \Big(\sum_{k=1}^{N/2} u_k a_k \Big(\prod_{\substack{i=1\\i\ne k}}^{N/2} m_i\Big)\Big) + M_2 * \Big(\sum_{k=N/2+1}^{N} u_k a_k \Big(\prod_{\substack{i=N/2+1\\i\ne k}}^{N} m_i\Big)\Big)$$

where $M_1 = \prod_{i=N/2+1}^{N} m_i$, $M_2 = \prod_{i=1}^{N/2} m_i$ which gives us:

<u>Theorem 3</u>: The generalised CRA can be performed in $O(N \log^2 N)$ if multiplication in the domain can be done in $O(N \log N)$ and the constants a_k and the requisite super-moduli $M_{j,f}$ can be preconditioned.

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

```
Algorithm : Interp ($u_i, e \le i \le f$) ;
  Input : the residues to be interpolated $u_i$,
          $e \le i \le f$, the constants $a_k$, the super-
          moduli
              $M_{j,k} = \prod_{i=j}^{k} m_i$

  Output : the interpolated value of U.

1) Basis : If e=f then Return ($u_e a_e$);
2) Recursion : $l := (e+f-1)/2$ ; $j := l+1$ ;
               $U_1 :=$ Interp($u_i, e \le i \le l$) ;
               $U_2 :=$ Interp($u_i, j \le i \le f$) ;
3) Multiplication : Return ($U_1 * M_{j,f} + U_2 * M_{e,l}$) ;
```

Analysis

If $I(2^n)$ is the time required to interpolate 2^n values u_i then $I(2^n)$ is defined by the recurrance relation:

$$\begin{aligned} I(2^n) &= 2I(2^{n-1}) + 2M(2^{n-1}) \\ &= 2I(2^{n-1}) + 2^n(n-1) \\ &= 2^nI(1) + 2^n \sum_{i=1}^{n} i \\ &= O(2^n n^2) = O(N\log^2 N) \end{aligned}$$

The super-moduli $M_{j,k}$ required in Theorem 3 can be produced by the Construct-Moduli algorithm as outlined in section 2.

Corollary 1: A preconditioned version of the integer CRA can be performed in $O(N\log^2 N \log\log N)$.

Corollary 2: A preconditioned interpolation algorithm can be performed in $O(N\log^2 N)$.

9. COMPUTING THE a_k

The polynomial case and integer case now diverge for the products of the inverses a_k required by the Interp algorithm.

For polynomials (cf Horowitz[11]):

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

in a more familiar form $a_k = 1/D'(x)|_{x=x_k}$

where $D(x) = \prod_{i=1}^{N} (x-x_i)$ and $D'(x)$ is its derivative.

$D(x)$ can now be computed using the results of the algorithm Construct-Moduli and its derivative can be computed in $O(N)$ steps. The process of evaluating the a_k now reduces to evaluating the derivative at N points $x=x_i$. In a previous section this evaluation was shown to require $O(N\log^3 N)$. This gives us:

Theorem 4: Polynomial interpolation can be performed in $O(N\log^3 N)$ steps.

For the integer case we have that:

$$a_k = (\prod_{\substack{i=1\\i\neq k}}^{N} s_i^k) \bmod m_k,$$

where $1 \equiv (s_i^k m_i) \bmod m_k$.

Note that a_k is a single precision element. From the theory of ideals we have that:

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

by commutativity which implies $1 \equiv b_k a_k \bmod m_k$

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

Thus b_k is the single precision inverse of a_k 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 in $O(N)$ operations. From the Construct-Moduli algorithm we can get $D = \prod_{i=1}^{N} m_i$. Since the moduli are pairwise relatively prime, D lies in the ideal (m_k) but not (m_k^2).

Since, if: $(\prod_{\substack{i=1\\i\neq k}}^{N} m_i) = xm_k + b_k$

we have that $D = xm_k^2 + b_k m_k$

and so $b_k m_k \equiv D \bmod m_k^2$

It follows that:

$$c_k \equiv D \bmod m_k^2 \quad\text{and}\quad b_k = c_k/m_k.$$

We can compute the b_k from the c_k in $O(N)$ steps and the c_k can be computed in $O(N\log^2 N\log\log N)$ steps using the Modular-Form algorithm as described above. Which leads us to:

Theorem 5: The integer CRA can be computed in $O(N\log^2 N \log\log N)$.

10. CONCLUSION

We have shown a fast polynomial division algorithm which works in $O(N\log^2 N)$. Using this we can evaluate an Nth degree polynomial at N points in time $O(N\log^3 N)$ using the Modular-Form algorithm given. Using this same algorithm we can reduce an N precision integer to N single precision residues in $O(N\log^2 N\log\log N)$.

In the second part we have shown that preconditioned polynomial interpolation can be performed in $O(N\log^2 N)$ by applying the methods of Horowitz and Heindel[7] and Horowitz[11]. As a corollary of the polynomial evaluation algorithm, the complete polynomial interpolation algorithm can be performed in $O(N\log^3 N)$. We have also shown that this algorithm can be appropriately modified to perform the Chinese Remainder Algorithm in $O(N\log^2 N\log\log N)$.

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

11. BIBLIOGRAPHY

1) S. Cabay: 'Exact Solution of Linear Equations', Proc. of 2nd Symp. on Symbolic and Algebraic Manipulation, March 1971.

2) W.S. Brown: 'On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors', JACM vol 18 no 4, Oct. 1971.

3) G.E. Collins: 'The Calculation of Multivariate Polynomial Resultants', JACM vol 18 no 4, Oct. 1971.

4) J.M. Polard: 'The Fast Fourier Transform in a Finite Field', Math. Comp., vol 25 no 114, April 1971.

5) J.D. Lipson: 'Chinese Remainder and Interpolation Algorithms', Proc. of 2nd Symp. on Symbolic and Algebraic Manipulation, March 1971.

6) A. Schoenhager: 'Schnelle Berechnung von Kettenbruchentwicklungen', Acta Informatica, vol 1 no 1, 1971.

7) E. Horowitz and L. Heindel: 'On Decreasing the Computing Time for Modular Algorithms', Proc. of 12th Symp. on Switching and Automata Theory, Oct. 1971.

8) A. Borodin and I. Munro: 'Evaluating Polynomials at Many Points', IPL, vol 1 no 2, July 1971.

9) D. Knuth: 'The Art of Computer Programming', Vol II, Addison-Wesley, 1969.

10) C. Fiduccia: 'Polynomial Evaluation via the Division Algorithm: the Fast Fourier Transform Revisited', Proc. 4th Symp. on Theory of Computing.

11) E. Horowitz: 'A Fast Method for Interpolation Using Preconditioning', submitted to IPL.

12) A. Schoenhager and V. Stassen: 'Schnelle Multiplication Grosser Zahlen', to be published in Computing.