ON THE EFFICIENCY OF ALGORITHMS FOR POLYNOMIAL FACTORING

by

Robert T. Moenck

Research Report CS-74-12

Department of Applied Analysis and Computer Science

University of Waterloo

Waterloo, Ontario, Canada

July, 1974

Keywords: Algebraic Manipulation, Polynomial Factoring, Roots in Finite Fields, Analysis of Algorithms

This research was supported by NRC Grants A5549 & A8237.

ON THE EFFICIENCY OF ALGORITHMS FOR POLYNOMIAL FACTORING

by Robert T. Moenck*

*Department of Applied Analysis
and Computer Science
University of Waterloo
Waterloo, Ontario, Canada.

Keywords: Algebraic Manipulation, Polynomial Factoring, Roots in Finite Fields, Analysis of Algorithms

*This research was supported by NRC Grants No. A-5549, A-8237.

ON THE EFFICIENCY OF ALGORITHMS FOR POLYNOMIAL FACTORING

by Robert T. Moenck*

ABSTRACT

Algorithms for factoring polynomials over finite fields are discussed. A construction is shown that reduces the final step of Berlekamp's algorithm to the problem of finding the roots of a polynomial in a finite field Z_p. It is shown that if the field is of the form $p = L \cdot 2^{\ell+1}$ where $\ell \simeq L$ then the roots of a polynomial of degree N can be found in $O(N^2 \log^2 p)$ steps. As a consequence Berlekamp's method can be performed in $O(N^3 \log p + k^2 \log^2 p)$ steps. If N is very large then it is shown that the factors of a polynomial can be found in $O(N^2 \log^4 N \log p)$ steps. Some consequences and empirical evidence is discussed.

*Department of Applied Analysis
and Computer Science,
University of Waterloo,
Waterloo, Ontario, Canada.

Keywords: Algebraic Manipulation, Polynomial Factoring, Roots in Finite Fields, Analysis of Algorithms

*This research was supported by NRC Grant No. A-5549.

I.) Introduction and Overview:

Polynomial factoring is an important operation in algebraic manipulation. It is important not only in itself but also as a sub-algorithm in other processes such as symbolic integration (cf Ris 68) or simplification (cf Cav 69) or solving polynomial equations (cf Yun 73). Naturally we wish to have a method which is quick and so we are led to consider the efficiency of factoring algorithms.

Generally in computer algebra one is concerned with factoring monic polynomials in one or more variables over the integers. Other factoring problems can generally be reduced to this case. A method due to Kronecker (cf Wae 49) is generally used to prove that such polynomials can be factored uniquely up to the order of the factors. Kronecker's method can be used as the basis of an algorithm for factoring polynomials (Jor 69). However the algorithm is very inefficient and the time it requires can be shown to grow exponentially in the degree of the polynomial to be factored.

This had led to the development of homomorphism methods. These methods reduce the problem to the univariate case with the polynomial reduced modulo a prime p. The resulting polynomial is factored over the finite field $Z_p = GF(p)$. Any factors over Z_p are used to determine factors over Z. Currently the best method to do this is based on Hensel's lemma. Musser (Mus 71) or Wang and Rothschild (Wan73) give a detailed exposition of the considerations involved in performing this step in the factoring process.

Here we are mainly concerned with the problem of finding a factoring over Z_p. Much of the work in this area has been done by Berlekamp. He produced (Ber 68) the first complete factoring algorithm

which works in $O(N^3 p)$ steps, to factor a polynomial of degree N over Z_p. One of the handicaps of this method was the p term in the timing analysis. This restricts the method to relatively small fields. Later Berlekamp (Ber 70) refined his method so that the factoring problem reduces to that of computing the roots of a polynomial in a finite field and showed how the latter problem could be solved in time proportional to $p^{1/4} \log p^{3/2}$.

In this paper we show (Sec. II-IV) a more straight-forward reduction to the root finding problem and a method for finding the roots of a polynomial of degree k in $O(k^2 \log^2 p)^*$ steps for special choices of p. These imply that Berlekamp's method can be performed in $O(N^3 \log P + k^2 \log^2 p)$ steps. It is further shown (in sec. V-VI) that a polynomial can be factored in $O(N^2 \log^4 N \log p)$ steps if N is very large. Finally (see VII) we indicate methods for computing primitive roots of unity and irreducible polynomials as are used in the new algorithms. In sec VIII we present a few empirical results and draw some conclusions.

First let us note that we need only consider the problem of factoring a monic polynomial with no repeated factors (square free). This is because we can divide by the leading coefficient and we can find repeated factors easily. Consider the case of a polynomial u(x) with repeated factors $f_2(x)$. ie.

$$u(x) = f_1(x)\, f_2^n(x)$$

differentiating: $u'(x) = f_1'(x)f_2^n(x) + nf_1(x)f_2^{n-1}(x)f_2'(x)$

$$= f_2^{n-1}(x)(f_1'(x)f_2(x)+nf_1(x)f_2'(x)).$$

This means that the GCD of a polynomial and its derivative will be the product of the repeated factors. These can then be divided out.

*All logarithms in this paper are base 2.

II) An Overview of Berlekamp's Algorithm

Berlekamp's Algorithm for factoring polynomials over a finite field is a major milestone in the study of the factoring problem. Since we are going to look at methods of improving it, it is pertinent to briefly review the basic method. The algorithm rests on three major observations:

a) For a square free monic polynomial u(x):

$$u(x) = \Pi f_i(x)$$

its factors $\{f_i\}$ are pairwise relatively prime in the Euclidean Domain $Z_p[x]$. Thus they can be used in the:

Chinese Remainder Theorem 1: (Lip 71). Any polynomial: $v(x) \in Z_p[x]$ s.t. deg (v) < deg (Πf_i) = deg (u)

can be uniquely represented by its residues $s_i(x)$ with respect to the $f_i(x)$. i.e.

2.1) $$v(x) \equiv s_i(x) \bmod f_i(x) \quad \forall i.$$

This implies that for a given v(x), if we could find the residues $\{s_i(x)\}$ then we can compute the factors. This is done by taking GCD's.

$$f_i(x) = GCD(v(x) - s_i(x), u(x))$$

b) The second observation provides a method finding the $\{s_i\}$ and the corresponding v(x). We see that it is worthwhile to choose a v(x) such that $s_i \in Z_p$ i.e. the residues are field elements, not polynomials. Then we can apply:

Fermat's Theorem 2 (Alb 56): For all $a \in GF(q)$, $a^q = a$.

When applied to the relationship of the residues (2.1) we get:

(2.2) $\forall i \qquad v(x)^p \equiv s_i^p \mod f_i(x)$

$= s_i$ by Fermat

$= v(x) \mod f_i(x)$

Now: $v(x)^p = v(x^p)$ in $Z_p[x]$ by the multinomial Theorem.

So we are looking for a polynomial $v(x)$ such that:

$$v(x^p) - v(x) \equiv 0 \mod f_i(x) \ .$$

It is sufficient to find a $v'(x)$ such that

(2.3) $$v'(x^p) - v'(x) \equiv 0 \mod u(x)$$

since $f_i(x) | u(x)$ implies that:

$$v'(x^p) - v'(x) \equiv 0 \mod f_i(x) \ .$$

c) The third observation is that if we build up a matrix Q such that its row vectors q_j are of the form

$$x^{pj} \equiv \sum_{i=0}^{N-1} q_{ij} \, x^i \mod u(x)$$

than finding a polynomial $v(x)$ can be viewed in terms of matrix operations as:

$$\bar{v}Q - I = \bar{0}$$

where $\bar{v}$ is the vector of coefficients of $v(x)$. In other words the problem reduces to finding the null space of the system:

$$\bar{v}[Q-I] = \bar{0}$$

with scalars in Z_p. In general one gets a set of null space vectors $\{\bar{v}_i\}$ including the trivial one $\bar{v}_0 = (1,0,\ldots,0)$. Berlekamp shows (Ber 68) that the number of such vectors is equal to the number of factors of

$u(x)$. To find the residues $\{s_i\}$ which correspond to the factors, Berlekamp suggested trying successive elements of the field until some were found which produced non-trivial factors.

Note that the factors produced by a given $v(x)$ may not be prime (irreducible) even though they are relatively prime. However each $v(x)$ will produce a different factoring and by trying all the $\{v(x)\}$ all the factors may eventually be produced.

Timing of the Algorithm

We can analyse the number of steps required by the algorithm as a function of the cardinality of p and N the degree of u. First we note that if p is large (say of the order of a computer word) it is necessary to compute field inverses when they are required. The best methods to do this computation use $O(\log p)$ field operations (Col 69).

Also we note that multiplying or dividing a polynomial of degree N by one of degree M can be done in $O(NM)$ field operations using the standard methods. As a corollary we see that squaring a polynomial of degree N-1 and computing its residue with respect to another polynomial of degree N can be done in $O(N^2 + N \log p)$ field operations. Thus building up the Q matrix involves computing $x^p \bmod u(x)$ in $O(N^2 \log p)$ steps and producing $x^{pj} \bmod u(x)$ for $2 \le j \le N-1$ in $O(N^3 + N^2 \log p)$. Computing the null space of the matrix Q-I can be done in $O(N^3 + N^2 \log p)$ steps using a standard triangularization algorithm. Collins (Col 68) has shown that the GCD operation to compute the factors can be computed in $O(N^2 \log p)$ steps.

If there are k factors, in the worst case each $v(x)$ will yield only one prime factor. To find this factor we might have to try every element in the field. This means that the algorithm is bound by the last step which

requires $O(N^2 k p \log p)$ operations. If p is large and $k=O(N)$ then the algorithm may require $O(N^3 p \log p)$ steps. It is the factor p in this expression which limits the algorithm. It is the factor p in this expression which limits the algorithm. It is because of it that the algorithm is efficient only for small primes.

III. Improvements to Berlekamp's Algorithm

A method of improving Berlekamp's Algorithm is to aid the computation of the residues $\{s_i\}$ given a null-space vector of coefficients $\bar{v}$. A way to do this is to follow a recommendation of Knuth (Knu 69 pp. 396) and Berlekamp and retain s as a parameter in the computation of:

3.1 $$\gcd(u(x),\ v(x) - s)$$

This will compute a polynomial in s and is an application of:

Resultant Theorem 3: (Usp 48) Given two polynomials A(x,s),B(x,s) their GCD will be non-trivial iff their resultant

3.2 $$\text{Res}(A,B) = r(s) = 0 \quad .$$

The resultant of two polynomials is closely related to their GCD in that it can be considered as the determinant of their Sylvester Matrix. Collins (Col 71) has given an efficient homomorphism algorithm for computing resultants. From his work we can see that:

if deg (u) = N then deg (Res (u(x),v(x)-s)) $\leq$ N

and that Res (u,v-s) can be computed by doing N univariate polynomial resultants and interpolating the results at a total cost of $O(N^3 \log p)$ steps. Thus we can find a set of residues $\{s_i\}$ by:

a) Computing the resultant

$$r(s) = \text{Res}(u(x),v(x)-s)$$

b) The GCD (u(x),v(x)-s) will be non-trivial when r(s) is zero i.e. at the roots of r(s).

Since it does cost $O(N^3 \log p)$ steps to find the resultant we want to avoid doing it once for each basis polynomial v(x). We do this by forming

the product V(x) of the {v(x)} mod u(x). Now $(V(x)-s_i)$ will split in the same way the {v(x)} do. So we can compute:

$$g'(s) = \text{Res}(u(x), V(x)-s)$$

and find the roots of g'(s). However, since all the factors of the {v(x)-s} may be duplicated in (V(x)-s) there may be as many as k^2 roots of g'(s).

This means that we must be more careful about how we find the factors of u(x). One method we can use is to store the factors in a binary tree which forms a seive for factors. For each node the right and left subtree will hold a factor of the polynomial stored at the node and its cofactor. The polynomial u(x) will be stored at the root. The following recursive algorithm Find_Factor might be used.

Algorithm: Find_Factor (f,k)

Input: the polynomial f(x) to be decomposed and stored in the tree rooted at r.

Steps:

```
1) Basis:  If r = null
           then begin r:= node (f,null,null); j:=j + 1
                      end
2) Test equality: else if f(x) > = r(x)
                       then begin
3) Split:                 g(x):=gcd(f(x),r(x));
                          f(x):=f(x)/g(x);
4) Recursion:             if g(x) >=1
                          then Find_Factor (g, left (r));
                          if f(x) > = 1
                          then Find_Factor (f,right (r));
                          end                              □
```

The algorithm is called by:

k=0; R:=node (u,null, null);

for $s_i \in$ roots of g'(s) while $j < k$

do Find_Factor (gcd(u(x), V(x)-s_i),R);

Using this algorithm we see that in the worst case the tree can become unbalanced and have a depth as large as k . In which case $O(N^2 k \log p)$ steps would have been required to produce it. If k=O(N) this is $O(N^3 \log p)$ which is the current limiting step on the algorithm. If care was taken to balance the tree at each stage this could be reduced to $O(N^2 \log N \log p)$.

On the other hand we could take the view that it is very unusual for the first basis polynomial $v_1(x)$ not to provide all the factors, if the prime p is large with respect to k or N. In this case the bound on the original Berlekamp algorithm is $O(N^2 \log p \max (k,p))$ and the resultant need only be computed with respect to the first basis polynomial $v_1(x)$.

In any case we have reduced the problem to that of finding the roots of a polynomial g(s) of degree N, in a finite field GF(p). To do this we can use:

Lemma 4 (Alb 56, p. 128) In a finite field of characteristic q

3.2 $$t(s,m) = s^{q^m} - s$$

is the product of all irreducible polynomials of degree which divides m.

In particular $t(s,1) = s^p - s$

is the product of all linear polynomials in $Z_p[S]$. So to reduce our search for the roots of r(s) we can compute:

$$g(s) = \text{GCD } (r(s), s^p - s) .$$

Then g(s) will be the product of the k linear factors of r(s), if there are k factors of the (V(x)-s).

One method of finding the roots of g(s) is to test each element of the field. Before we needed to compute gcd each time we tested a field element at a total cost of $O(N^2\ p \log p)$. Now we need only evaluate the polynomial g(s) at a total cost of $O(k\ p) \leq O(N\ p)$.

If the prime p is sufficiently small that

$$O(N^3 \log p) \geq O(N^2\ p)$$

$$\text{i.e.}\quad O(p) \leq O(N^2 \log p)$$

then we can use this method to find the roots of g(s) and thus the factors of u(x) in $O(N^3 \log p)$ steps (i.e. the time to manipulate the Q matrix dominates the algorithm).

In summary we can state:

Theorem 5. A monic square free polynomial of degree N can be factored over a finite field Z_p in $O(N^3 \log p + R(N))$ steps where R(N) is the time to find the roots in the field of a polynomial of degree N.

IV. Finding Roots of Polynomials in Large Finite Fields

In the previous section we reduced the factoring problem to that of finding the roots of a polynomial g(s). Berlekamp (Ber 70) discusses probablistic methods for finding the roots of a polynomial in a finite field Z_p. He gives the timing of the method as $O(p^{1/4} \log p^{3/2})$. In this section we will discuss ways this may be improved upon.

What we can observe is that if we are free to choose the prime characteristic, (as we are when we are using homomorphism algorithms to do factoring) then we can choose fields which expedite the discovery of the roots of a polynomial.

In particular it is useful to choose primes p such that p-1 is highly composite (e.g. $p = L \cdot 2^{\ell} + 1$ where L is small, $\ell \simeq L$). In this case we can find the roots of a polynomial by a process of refining the multiplicative subgroup of the field in which the roots lie. At most $\ell \leq \log p$ such refinements need be made in order to find a root. This means that the time to find the roots of a polynomial in a field Z_p and thus the factors of a polynomial over Z_p is algebraic in log p and not proportional to p.

First we assume that the roots of the polynomial are nonzero and square free. This is something that can easily be checked by the construction of section I. Then we note the special case of lemma 4 that:

4.1) $$c(s) = s^{p-1}-1$$

is the product of all nonzero linear factors in Z_p.

Its roots are the members of the multiplicative group Z_p^*. Members of this group are called p-1 st roots of unity and generators are called

primitive p-1 st roots of unity. Note that c(s) factors as:

4.2) $$c(s) = (s^{(p-1)/2} + 1)(s^{(p-1)/2} - 1)$$

so that half of the p-1 st roots of unity are also (p-1)/2 -th roots of unity. In general c(s) has $\ell+1$ factors of the form:

4.3) $$c_i(s) = s^{L\cdot 2^{\ell-i}} - 1 \quad , \quad 0 \le i \le \ell$$

where $c_i(s)$ is the product of all $L\cdot 2^{\ell-i}$ -th roots of unity of Z_p (i.e. elements of a subgroup of Z_p^*). Using this fact we can separate the roots of a polynomial g(s) into those roots which are $L\cdot 2^{\ell-i}$ -th roots and those which are not, by taking

4.4) $$g_i(s) = \text{GCD}\ (g(s),\ s^{L\cdot 2^{\ell-i}} - 1) \quad .$$

We can use this fact to refine the subgroup in which a set of roots are contained. We could compute a sequence $\{f_i(s)\}$ of polynomials where $f_i(s)$ is the product of all $L\cdot 2^{\ell-i}$ -th roots of unity of g(s) which are not $L\cdot 2^{\ell-i-1}$ st roots of unity. This means $f_i(s)$ has the form:

4.5) $$f_i(s) = \Pi\ (s - \omega^{j\cdot 2^i})$$

where j is odd. If ψ is another (not necessarily distinct) primitive p-1 st root of unity then $\psi = \omega^m$ where (p-1,m) = 1, in particular m is odd. Then we form $f_i'(s)$ as follows:

4.6) $$\begin{aligned} f_i'(s) &= \pi\ (s - \omega^{j\ 2^i} \cdot \psi^{\ 2^i}) \\ &= \pi\ (s - \omega^{\ .2^i\ (j+m)}) \\ &= \pi\ (s - \omega^{2^{i+1}(j+m)/2}) \text{ since j and m are odd.} \end{aligned}$$

Thus the roots of $f_i'(s)$ are $L\cdot 2^{\ell-i-1}$ st roots of unity and so the refinement can be applied to them. Once we have found the roots of $f_i'(s)$ we can compute the roots of $f_i(s)$ by dividing by ψ^{2^i} .

We can express the transformation of $f_i(s)$ to $f_i'(s)$ in terms of coefficients of $f_i(s)$. If

$$f_i(s) = \sum_{j=0}^{N} a_j x^j \quad , a_N=1$$

then expanding (4.6) we see:

4.7) $$f_i'(s) = \sum_{j=0}^{N} a_j x^j \psi^{2^i(N-j)}$$

So the transformation of $f_i(s)$ to $f_i'(s)$ can be performed in 2N operations given the coefficients of $f_i(x)$ and ψ^{2^i}. Let the conversion be performed by an algorithm called CONVERT. Then the following algorithm can be used to compute the roots of a polynomial in a finite field.

Algorithm: Roots (u,ω, i, H, p)

Input: the polynomial u(s), ω the primitive $(p-1)/2^i$ th root of unity, in field Z_p where $p = L\cdot 2^{\ell}+1$, H a (possibly null) list of polynomials of the form:

$$h_j(x) = s^{L\cdot 2^j} - 1 \bmod u(s),\ 0 \le j \le \ell - i$$

Output: R a list of roots of u(s) in Z_p.

Steps:

1) Basis: If deg(u)=1 then return $-u_0$

2) Direct search: else if $2 \mid (p-1)/2^i$

then return Direct-search $(u,\omega,(p-1)/2^i,p)$

3) Root separation: else begin.

if H is null then compute H;

h(s):=head(H); H:=tail (H);

g(s):=GCD(u(s), h(s))

f(s):=u(s)/g(s);

R:=null;

4) Recursion: if deg(g) > 0

then R := (R, Roots (g, ω^2, i+1, H, p));

if deg (f) > 0

then begin

f:=Convert (f,ω);

R:=(R,Roots(f,ω^2,i+1,null,p)/ω);

end;

return R;

end.

The algorithm would be invoked as:

R := Roots(u(s),ω,0,null,p);

where ω is a primitive p-1 st root of unity. The algorithm Direct-Search (U,ω,L,p) is invoked in the Roots algorithm to find the roots of u(s) in the multiplicative group of ω by direct evaluation. Since there are only L members of this group and L is chosen to be small, this operation does not take long. Also if L is composite and the product of small primes then a method related to the Roots algorithm can be used to further refine the group structure of the field and the roots of the polynomial.

Timing of the Root Algorithm:

The polynomials $h_j = s^{L \cdot 2^j - 1} \bmod u(s)$, $0 \le j \le \ell - i$

can be computed in $O(N^2(\ell-i)) \le O(N^2 \log p)$ steps where deg(u) = N.

Similarly the GCD operations can be computed in $O(N^2 \log p)$ steps. The worst case situation for the algorithm is that the refinement of the subgroups doesn't separate the roots at all. In which case the Direct-Search algorithm must be used to separate them. There can be at most log p refinements. Thus the total cost is $O(N^2 \log^2 p + NL)$.

In summary we have

Theorem 6: In a finite field Z_p where $p = L \cdot 2^k + 1$ and $L \simeq k$. The roots of a polynomial of degree N can be computed in $O(N^2 \log^2 p)$ steps.

Corollary 1: Over such a finite field a polynomial of degree N can be factored into k factors in $O(N^3 \log p + k^2 \log^2 p)$ steps.

V) Asymptotic Methods

The removal of the limit on the size of the field for the algorithm raises the question: can the number of steps taken be pared down further? The answer is that,they can if we are prepared to accept asymptotic analysis.

The timing estimates for the algorithms which we have used so far are reasonably consistent with the observed behaviour of algorithms in real algebraic manipulation systems for the size of problems usually encountered. However if we are prepared to consider very large problems then we can use a set of algorithms with slower growing timing functions. It can be shown (see for example Moe 73) that two polynomials of degree N can be multiplied together in O(N log N) field operations. Similarly a polynomial of degree 2N can be divided by one of degree N in O(N log N) field operations and the GCD of two polynomials of degree N can be computed in $O(N \log^2 N \log p)$ field operations.

If we admit such algorithms into consideration do they help us at all? The answer is yes but we must extensively reformulate the algorithm. The bounding step now becomes the matrix operations. Although Strassen (Str 69) has shown that matrices can be multiplied or triangularised in $O(N^{2.81})$ we can even improve on this limit of the algorithm.

Our first observation is that we can find a partial factoring of a polynomial using a method based on lemma 4: due to Golomb et al(Gol 59). This partitions the factors of a polynomial into products of all the factors of the same degree. The following algorithm achieves this:

Algorithm: Distinct-Degree Factors (u)

Input: the polynomial $u(x)$ over Z_p, deg (u) = N;

Output: The distinct degree factors $d_i(x)$

$$u(x) = \prod d_i(x) , \qquad d_j(x) = \prod fj(i)(x)$$

Step:

1) Initialisation: $h(x) = x^p \bmod u(x)$

$g(x) := i := j := 1;\ v(x) := u(x);$

2) Iteration: while $j <$ deg $(v)/2$

do begin

$g(x) := h(x) * g(x) \bmod u(x);$

$j := j + 1;$

$d_i(x) :=$ GCD $(g(x) - x,\ (x))$

3) Non trivial factors:

if $d_i(x) \neq 1$ then

begin

$v(x) := v(x)/d_i(x);$

$i := i+1$

end

end;

4) Completion: if $v(x) > 1$ then $d_i(x) := v(x)$

end.

By lemma 4 at the j-th iteration of the loop g(x) is the product of all irreducible polynomials of degree dividing j. Hence the GCD operation finds all factors of degree j. Using the classical methods of sections 2-4 the algorithm can be shown to take $O(N^3 \log p)$ steps. However we have.

Theorem 8: The Distinct Degree factors algorithm can find the factors of a polynomial of degree N in $O(N^2 \log^2 N \log p)$ steps.

Proof:

$x^p \bmod u(x)$ can be built up by $O(\log p)$ squarings for a total of $O(N \log N \log p)$ steps. The GCD operation can be performed in $O(N \log^2 N \log p)$ steps. Since the loop may be executed N/2 times the bound is $O(N^2 \log^2 N \log p)$ steps.

VI. Splitting Distinct Degree Factors

We have reduced the factoring problem to that of separating products of factors of the same degree.

i.e.

6.1) $$d(x) = \prod_{i=1}^{k} f_i(x)$$

where deg (d) = N = mk, deg (f_i) = m for all i. To do this we find a monic irreducible polynomial r(x) of degree m. Then we can reduce d(x) mod r(x) to compute the polynomial:

6.2) $$F(y) = \sum_{i=0}^{k} y^i s_i \text{ where } s_i \in Z_p[x]/(r(x))$$

This produces a polynomial of degree k over the finite field $GF(p^m) = Z_p[x]/(r(x))$. If we can compute the roots $\{t_i\}$ of the polynomial F(y) then we can find the factors $f_i(x)$ of d(x). Since:

6.3) $$f_i(x) = r(x) - t_i(x)$$

$$= y - t_i$$

If p^m-1 is highly composite we could apply the algorithm of section IV to find the roots of F(y). However in general it is not true that p^m-1 is highly composite if p-1 is and it is difficult to choose a suitable p a priori when we don't know m.

Instead we can use a construction of Berlekamp's (Ber 70) to convert the polynomial back to one over Z_p. This is based on the

Lemma 9: In $GF(p^m) = Z_p[x]/(r(x))$

6.3) $$y^{p^m} - y = \prod_{s=0}^{p-1} (T_r(y)-s)$$

where

6.4) $$T_r(y) = \sum_{i=0}^{m-1} y^{p^i}$$

This fact can be used to find the roots of F(y) in $GF(p^m)$ by testing:

6.5) $$GCD(F(y), T_r(y) - s)$$

for each of the $s \in Z_p$. In general it is necessary to test

6.6) $$GCD(F(y), T_r(x^i y) - s) \quad , \quad \text{for } 0 \leq i \leq m-1$$

In order to find all the roots of F(y). We can find the values of s in (6.5) which yield roots of F(y) using the resultant construction and the Roots algorithm as outlined in sections III - IV. Let the algorithm that transforms $T_r(x^i y)$ to $T_r(x^{i+1} y)$ be called Trans then the root finding can be performed by the following algorithm.

Algorithm: Roots-in-$GF(p^m)$ $(F,p,\omega,r(x),m)$;

Input: The polynomial F(y), the prime p with primitive p-1 root of unity ω in Z_p, the field modulus r(x) where deg (r) = m.

Output: A list of roots L.

Step:

1) Initialisation: $T_r(y) := \sum_{i=0}^{m-1} y^{p^i} \bmod F(y)$;

j:=1; l:=()

2) Iteration: while deg (F(y)) > 1 and j < m

do begin

if j > 1 then $T_r(y)$:= Trans $(T_r(y))$

j := j+1

H := Roots (Res(F(y),$T_r(y)$-s),ω,1,0,p);

3) Find the roots in $GF(p^m)$: for each $s_i \in H$

begin

$y-t_i := GCD(F(y), T_r(y)-s_i);$

$F(y) := F(y)/y-t_i$

$L:= (L,y-t_i);$

end

end;

4) Test F(y): if deg (F(y)) = 1 then L:=(L,F(y))

End.

Timing of the Roots in $GF(p^m)$ algorithm:

First we note that

$$T_r(y) \equiv \sum_{i=0}^{k-1} t_{r_i}(x)\, y^i \bmod F(y)$$

where $$t_{r_i}(x) = \sum_{\ell=0}^{m-1} t_{i\ell}\, x^{\ell} \quad ,$$

and

$$T_r(xy) \equiv \sum_{i=0}^{k-1} t_{r_i}(x)\, x^i y^i \qquad \bmod F(y).$$

So the process of forming $T_r(x^j y)$ from $T_r(x^{j-1}y)$ corresponds to reducing $t_i(x * x^i) \bmod r(x)$ for the k coefficients $t_i(x) \in GF(p^m)$. Each such reduction can be made in $2i$ m operations in Z_p. So the transformation of $T_r(x^{j-1}y)$ to $T_r(x^j y)$ can be made in

$$2 \sum_{i=0}^{k-1} im \simeq 2km = 2N \text{ operations.}$$

In general operations in $GF(p^m)$ can be performed in m operations in Z_p for addition, $O(m \log m)$ operations in Z_p for multiplication and $O(m \log^2 m \log p)$ for computing inverses using the extended euclidean algorithm (Moe 73).

This means that $y^p \bmod F(y)$ can be formed in $O(\log p)$ squarings mod $F(y)$ doing arithmetic in $GF(p^m)$. This involves $O(\log p \cdot k \log k \cdot m \log m)$ operations in Z_p or $O(N \log k \log m \log p)$ steps. Similarly the resultant of step 2 can be computed in $O(k^2 \log^2 k \log p)$ operations in $GF(p^m)$ using a method similar to that used for polynomial GCDs in (Moe 73). This involves a total of $(k^2 \log^2 (k) \, m \log^2 (m) \log p)$ steps. Since the iteration is performed a maximum of m times the time spent computing resultants during the course of the algorithm is at most $O(N^2 \log^2 k \log^2 m \log p)$ steps. In fact it is this that bounds the computation in the algorithm since the Roots algorithm must deal with a polynomial of degree at most k. Theorem 6 shows that the roots of such a polynomial can be found in $O(k^2 \log^2 p)$ steps. In summary we have:

Theorem 10: Over the finite field $GF(p^m)$ where $p = L \cdot 2^{\ell} + 1$ and $L \simeq \ell$, the roots of a polynomial of degree k can be computed in $O(k^2 \log^2 k \, m^2 \log^2 m \log p)$ steps.

Corollary 11: If $d(x)$ is a distinct degree partition of the factors $\{f_i(x)\}$ of a polynomial over Z_p then the k factors can be discovered in

$$O(N^2 \log^2 k \log^2 m \log p) \text{ steps.}$$

where $\deg(d)=N$, $\deg(f_i) = m$.

Corollary 12: The factors of a polynomial of degree N over a finite field Z_p can be found in $O(N^2 \log^4 N \log p)$ steps.

Proof:

The worst case for the factoring algorithm is when all k of the factors are of the same degree m and $m=k=\sqrt{N}$. This situation can be solved in

$O(N^2 \log^2 N \log p)$ by theorem 8

$+ O(N^2 \log^2 k \log^2 m \log p)$ steps by theorem 10.

This is majorised by $O(N^2 \log^4 N \log p)$ steps.

Finally we should note that if we dismiss the use of asymptotic algorithms then the methods described in the last two sections can be performed in $O(N^3 \log p)$ steps. This is the same bound as achieved by the methods in sections II - IV.

As a non-trivial lower bound on the factoring problem we have:

Theorem 13: At least N log N/e rational multiplications are needed to factor a polynomial of degree N over the integers.

Proof:

Any factoring algorithm must divide out the factors it generates, from the factored polynomial. Clearly this operation must be at least as difficult as multiplying the factors together to verify that they form the polynomial to be factored.

In one extreme case where all the factors are linear Strassen (Str 73) has shown that such a multiplication needs $N \log \frac{N}{e}$ field multiplications.

While the lower bound is for an infinite field and we have been considering finite fields it is probably reasonable to assume that a similar

result holds for finite fields. From similar results by Strassen it could be conjectured that a bound of

$$N(1 - \frac{1}{p}) \log (\frac{N(1 - 1/p)}{e})$$

would hold in a field of characteristic p.

VII. Finding Primitive Roots and Polynomials

In the root finding algorithms we have used certain field elements as part of the algorithm. We should indicate that these are fairly easy to find.

Primitive roots of unity can be found quite readily using a method which depends on the:

Theorem 14 (Alb 56) In the finite field GF(q),e is a primitive q-1 st root of unity iff $e^{(q-1)/a_i} \neq 1 \bmod q$

for all prime divisors $a_1,\ldots,a_r$ of q-1

There can be at most log q prime divisors of q-1 and to form $e^{(q-1)a_i}$ can take at most O(log q) field operations. So, to test a trial primitive root requires $O(\log^2 p)$ operations in Z_p. Not only that, such elements are fairly common in the field since there are $\phi(q-1) = O(q-1)$ of them in the field. This implies that primitive p-1 st roots can be easily found.

Finding irreducible polynomials for the Roots in $GF(p^m)$ algorithm is more difficult but we can use:

Theorem 15 (Lang 67 p. 221): Let K be a field and n an integer ≥ 2. Let $a \in K$ and $a \neq 0$. If for all prime divisors c of n, $a \notin K^c$ (a doesn't have a c-th root in K) and if $4 \mid n$ and $a \notin -4K^4$ then x^n-a is irreducible in K[x].

We can test if a has a c-th root in Z_p by testing for the existence of any linear factors of x^c-a. Applying the method of section V this involves computing:

$$e_c(x) = GCD(x^p - x,\ x^c - a).$$

We see that:

$$x^p \equiv a\, x^{p-c} \mod x^c - a$$

$$\equiv a^{\ell}\, x^{p-\ell c} \mod x^c - a$$

where $\ell = \lfloor p/c \rfloor$

Therefore $e_c(x) = \mathrm{GCD}(a^{\ell}\, y^{p-\ell c} - y,\ y^c - a)$.

Therefore if $e_c(x) = 1$ for all prime divisors c of n then $x_n - a$ is irreducible in K[x]. Again there can be at most log n prime divisors of n. To compute e_c for all of them would take $O(n^2 \log n \log p)$ steps or $O(n \log^3 n \log p)$ steps using an asymptotic method.

VIII. CONCLUSIONS:

As a test of practicality the modifications of Berlekamp's algorithm have been programmed using the SAC-1 (Col 71a) algebraic manipulation system. SAC-1 contains both the original Berlekamp algorithm and the distinct degree method as part of the system. Some sample times for these four algorithms applied to a polynomial of degree 14 for various primes are given in table I. These times are taken from an implementation of SAC-1 on the Honeywell 6050 at the University of Waterloo.

The first modified method computes the roots of the resultant by evaluating it at sufficiently many points in the field. The second modification employs the Roots algorithm to find them. The upper half of the table shows that for small and moderate sized primes the modifications are slightly slower than Berlekamp's methods. This is to be expected since they have the overhead of computing a resultant. However at primes of the order of 100 the modifications can already be significantly faster than the original algorithm. Above this point the Berlekamp algorithm may on occasion be faster than the modifications. This occurs when it is lucky enough to find the factors after only trying a few GCD's. However in general it is slower than the modifications.

This raises the possibility of dispensing with Hensel's lemma when using the modular factoring to find factors over the integers. If a prime be chosen so as to be twice as large as any coefficient of a factor, then we could factor modulus this prime and test combinations of the resulting factors as trial integer factors. This would by-pass the application of Hensel's lemma for the univariate case. This approach is most reasonable if the chosen prime is less than the word size of the computer.

Time (in seconds) for factoring and polynomial of degree 14

Prime P	Berlekamp	Modified	With Roots Algorithm	DDF algorithm	Degree of Factors
17	2.41	3.62	3.62	1.95	1,2,3,3,5
47	3.81	4.66	4.69	2.5	1,1,3,3,6
83	2.5	2.5	2.5	3.32	14
107	7.05	4.72	4.66	2.91	4,10
137	6.3	4.78	4.86	2.5	1,1,1,1,2,2,3,
199	3.56	4.4	4.45	3.3	7,7
251	8.88	5.63	5.66	3.04	2,2,2,4,4,
331	4.4	5.17	5.22	3.15	4,10
449	11.7	5.17	5.05	3.32	2,12
$40961_{13} = 5.2^{13}+1$		42	7.96	3.85	1,1,1,2,2,3,4
$1790967809 = 427.2^{22}+1$			10.8	6.5	1,2,4,7
$1811939329 = 27.2^{26}+1$			13.1	6.28	1,1,2,3,3,4
$1835008001 = 875.2^{21}+1$			14.8	6.28	1,1,2,2,3,5
$1863319553 = 1777.2^{20}+1$			14.25	6.62	1,1,5,7

TABLE 1

However if the bound on the coefficients of the factors is larger than the word size, I would expect that the overhead of doing multi-precision modular arithmetic throughout the course of the computation would slow down the algorithm considerably. In such a case a reasonable strategy to use would be a hybrid method using factors modulo a word size prime and Hensel's lemma. In general the smaller the prime the more factors are produced. This means that in order to find the factors over the integers more combinations of small factors must be tried. This is an additional benefit of factoring with large primes.

The lower part of the table shows the results for large primes; in particular for primes close to the word size (2^{36}) of the computer. It can be seen that for primes of the order of fifty thousand the first modification is significantly slower than that using the Roots algorithm. Also that the modification using the Roots algorithm is always within a factor of 2 of the speed of the distinct degree method which is in general the fastest method. The time for the Roots modification using a word sized prime is only 5 times that using the prime 17.

While I feel that these times could be improved by using a different algebraic manipulation system and turning the code and using a faster computer; I do not think that their relative magnitudes will change much.

BIBLIOGRAPHY

Alb 56 — Albert, A.A. : Fundamental Concepts of Higher Algebra, University of Chicago Press, 1956.

Ber 68 — Berlekamp, E.R. : Algebraic Coding Theory: Chap 6; McGraw Hill, New York 1968.

Ber 70 — Berlekamp, E.R. : Factoring Polynomials over Large Finite Fields; Math. Comp. Vol. 24, No. 111, pp. 713-735, July 1970.

Cav 69 — Caviness, B.F. : On Canonical Forms and Simplification: Ph.D. Carnegie Mellon University 1967.

Col 68 — Collins, G.F. : Computing Time Analyses of Some Arithmetic and Algebraic Algorithms; Proc. 1968 IBM Summer Inst. in Symbolic and Algebraic Computation.

Col 69 — Collins, G.E. : Computing Multiplicative Inverses in GF(p); Math. Comp. vol. 23 no. 5 (Jan 69) pp. 197-200.

Col 71 — Collins G.E. : The Calculation of Multivariate Polynomial Resultants; JACM vol 18, no 4 (Oct 1971) pp 515-532.

Col 71a — Collins, G.E. : The SAC-1 System: An Introduction and Survey; Proc 2nd SIGSAM Symp., ACM New York 1971.

Gol 59 — Golamb, S, Welch, L, Hales, A., : On the Factorization of Trinomials over GF(2); JPL memo 20 - 189 (July 14, 1959) (as referred to in Knu 69).

Jor 69 — Jordan, D. and Kain, R. Clapp, L., : Symbolic Factoring of Polynomials in Several Variables CACM (Aug 1966) vol 9 no 8.

Knu 69 — Knuth, D., : The Art of Computer Programming vol II: Seminumerical Algorithms; Addison-Wesley Reading Mass. 1969.

Lang 71 — Lang S., : Algebra Addison-Wesley Reading Mass. 1971.

Lip 71 — Lipson, J.D.: Chinese Remainder and Interpolation Algorithms; Proc 2nd SIGSAM Symp. ACM New York 1971.

Moe 73 — Moenck, R.T.: Studies in Fast Algebraic Algorithms: Ph.D. Thesis, U. of T. 1973.

Mus 71 — Musser, D.R.: Algorithms for Polynomial Factorization; Ph.D. Thesis, U. of Wisconsin, 1971

Ris 68 — Risch R.H. : Symbolic Integration of Elementary Functions; Proc 1968 IBM Summer Inst. on Symbolic and Algebraic Manipulation.

Str 69 Strassen, V. : Gaussian Elimination is Not Optimal; Numerische Mathematik Vol 13 (1969).

Str 73 Strassen, V. : Die Berechungs komplexität elementarsymetrischen Funktionen und von Interpolations koeffizienten; Numerische Mathematik Vol 17 (1973).

Usp 48 Uspensky, J.V.: The Theory of Equations Chap 11; McGraw Hill, New York 1948.

Wae 49 Van den Waerden, B.L. : Modern Algebra vol 1: Fredrick Ungar, New York, 1949.

Wan 73 Wang P.S. and Rothschild L.P.: Factoring Polynomials over the Integers; SIGSAM Bull. No. 28 Dec. 1973.

Yun 73 Yun, D.Y.Y. : On Algorithms for Solving Systems of Polynomials Equations; SIGSAM Bull. No. 27 Sept. 1973.