MATHEMATICS OF COMPUTATION, VOLUME 31, NUMBER 137
JANUARY 1977, PAGES 235–250

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 which 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 characteristic of the field is of the form $p = L \cdot 2^l + 1$, where $l \simeq L$, then the roots of a polynomial of degree n may be found in $O(n^2 \log p + n \log^2 p)$ steps.

As a result, a modification of Berlekamp's method can be performed in $O(n^3 + n^2 \log p + n \log^2 p)$ steps. If n is very large then an alternative method finds the factors of the polynomial in $O(n^2 \log^2 n + n^2 \log n \log p)$. Some consequences and empirical evidence are discussed.

I. Introduction and Overview. Polynomial factorization is an important operatior in algebraic manipulation. It is important not only in itself but also as a subalgorithm in other processes such as symbolic integration [1] or simplification [2] or solving poly nomial equations [3]. 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 factorization problems can generally be reduced to this case. A method due to Kronecker [4] 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 a basis of an algorithm for factoring polynomials [5]. 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.

The inefficiency of the above method has 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 $Z_p = \mathrm{GF}(p)$. Any factors over Z_p are used to determine factors over Z. Currently, the best method for doing this is based on Hensel's lemma. Musser [6] or Wang and Rothschild [7] give a detailed exposition of the considerations involved in applying Hensel's lemma to the factoring process.

Here we are mainly concerned with the problem of finding a factoring over Z_p.

Received August 22, 1974; revised March 18, 1976.

AMS (MOS) subject classifications (1970). Primary 68A15, 68A20, 10M05, 12C05; Secondary 12–04, 13F20.

Key words and phrases. Algebraic manipulation, polynomial factoring, roots in finite fields, analysis of algorithms.

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

Much of the work in this area has been done by Berlekamp [8]. He produced the first complete factoring algorithm which works in $O(n^3p)$ 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 [9] refined his method so that the factoring problem reduced to computing the roots of a polynomial in a finite field. He showed how the latter problem could be solved in time proportional to $p^{1/4}\log p^{3/2}$.

In this paper we give (Sections II–IV) a more direct reduction to the root finding problem and a method for finding the roots of a polynomial of degree k in $O(k^2\log p + k\log^2 p)$** steps for special choices of p. These imply that Berlekamp's method can be performed in $O(n^3 + n^2\log p + n\log^2 p)$ steps for most cases. It is further shown (Sections V–VI) that a polynomial can be factored in $O(n^2(\log^2 n + \log n \log p))$ steps if N is large. Finally (Section VIII) we indicate methods for computing primitive roots of unity and irreducible polynomials as are used in the new algorithms. In Section VIII we present a few empirical results and draw some conclusions regarding these new methods.

II. An Overview of Berlekamp's Algorithm. First let us note that the general factoring problem reduces to that of factoring a monic polynomial with no repeated factors (i.e., square-free). This is because we can divide by the leading coefficient to make the polynomial monic and we can use the following well-known method for finding repeated factors. Consider the case of a polynomial $U(x)$ with repeated factors $f_2(x)$, i.e.,

$$U(x) = f_1(x)\cdot f_2^n(x)$$

differentiating:

$$\begin{aligned} U'(x) &= f_1'(x)f_2^n(x) + nf_1(x)f_2^{n-1}(x)f_2'(x) \\ &= f_2^{n-1}(x)(f_1'(x)f_2(x) + nf_1(x)f_2'(x)), \end{aligned}$$

$$f_2^{n-1}(x) = \mathrm{GCD}(U(x), U'(x)). \tag{2.1}$$

Here and throughout the paper GCD denotes the monic greatest common divisor of the two polynomials. The repeated factors may now be divided out of $U(x)$ to make it square free.

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

(i) The k factors $f_i(x)$, $1 \leqslant i \leqslant k$, of a square-free monic polynomial $U(x)$ are relatively prime in the Euclidean domain $Z_p[x]$. This means they may be applied in

---

** *Notation.* All logarithms in this paper are base 2. The notation $O(k^2\log p + k\log^2 p)$ has been adopted here to indicate the dominant terms in a two parameter cost function. This would be more precisely $O(c_1k^2\log p + c_2k\log^2 p)$ for some constants c_1 and c_2. The constants have been dropped here and throughout the paper to simplify the expressions.

the following specialization of the well-known Chinese Remainder Theorem [10].

THEOREM 1. *Given a set* $\{f_i(x)\}$ *of pairwise relatively prime polynomials and a set of residues* $\{S_i(x)\}$, $\deg(S_i) < \deg(f_i)$ *in* $Z_p[x]$, *there is a unique polynomial* $V(x) \in Z_p[x]$ *such that*:

$$\text{(2.2)} \qquad \begin{aligned} &\text{(a)} \quad \deg(V) < \deg(\Pi f_i) = \deg(U), \\ &\text{(b)} \quad V(x) \equiv S_i(x) \bmod f_i(x), \qquad 1 \leqslant i \leqslant k. \quad \square \end{aligned}$$

This implies that given $V(x)$ and the residues $S_i(x)$ we can compute the factors by taking GCDs

$$\text{(2.3)} \qquad f_i(x) = \mathrm{GCD}(V(x) - S_i(x), U(x)).$$

(ii) 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. *For all* $a \in Z_p$, $a^p = a$. $\square$

When applied to the relationship (2.2) of the residues we get

$$\begin{aligned} V(x)^p &\equiv S_i^p \mod f_i(x), \\ &= S_i \equiv V(x) \mod f_i(x). \end{aligned}$$

Now, $V(x)^p \equiv V(x^p)$ by the Multinomial Theorem in $Z_p[x]$. Therefore, we should look 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

$$\text{(2.4)} \qquad V'(x^p) - V'(x) \equiv 0 \mod U(x),$$

since $f_i(x) \mid U(x)$ implies that

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

(iii) The third observation is that if we construct a matrix Q such that its rows q_j are of the form

$$x^{pj} \equiv \sum_{i=0}^{n-1} q_{ij} x^i \bmod U(x) \quad \text{for } 0 \leqslant j \leqslant n-1,$$

then finding a polynomial $V(x)$ can be viewed in terms of matrix operations as

$$\text{(2.5)} \qquad \overline{V}[Q - I] = \overline{0}$$

where $\overline{V}$ is the vector of coefficients of $V(x)$. In other words, the problem reduces to finding the null space of the system (2.5).

In general one gets a set of null space vectors $\{\overline{V}_i\}$ including the trivial one $\overline{V}_0(1, 0, \ldots, 0)$. Berlekamp [8] shows that the number of such vectors is equal to k 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 are found which produce nontrivial factors.

Note that if two (or more) factors give the same residue of $V(x)$:

$$V(x) - S = f_1 \cdot a = f_2 \cdot b = f_1 \cdot f_2 \cdot c, \tag{2.6}$$

then the GCD operation of (2.3) will yield their product and not the individual factors. However, each $V(x)$ will produce a different factoring and by trying all the $\{V(x)\}$ all the factors can eventually be produced.

Timing of the Algorithm. We can analyze the number of steps in the algorithm as a function of the cardinality of p and of n the degree of U. First, we assume that the prime p can fit into a computer word and thus all operations on field elements, except finding inverses, use $O(1)$ steps. If p is large, it is necessary to compute field inverses when they are required. The best methods [11] to do this computation use $O(\log p)$ field operations.

We note that multiplying or dividing a polynomial of degree n by one of degree m can be done in $O(mn)$ 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 $O(n^2)$ field operations. Since $u(x)$ is monic, $x^p \bmod u(x)$ can be computed by repeatedly squaring in $O(n^2 \log p)$ steps. The remaining $n-2$ rows of the matrix Q, $x^{pj} \bmod u(x)$ can be produced in $O(n^3)$ steps. Computing the null space of the matrix $Q - 1$ can be done in $O(n^3 + n \log p)$ steps using a standard triangularization algorithm. Collins [12] has shown that the GCD operation of (2.3) can be performed in $O(n^2 + n \log p)$ 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 bounded by the last step which requires $O(kp(n^2 + n \log p))$ field operations. If $k = O(n)$, the algorithm may require $O(n^3 p)$ steps. It is the factor p in this expression which restricts the application of the algorithm to small primes.

III. Improvements to Berlekamp's Algorithm. A method for improving Berlekamp's algorithm is to aid the computation of the residues $\{S_i\}$ given a null space vector of coefficients $\overline{V}$. This may be done by following a recommendation of Knuth [13, p. 396] and Berlekamp to retain s as a parameter in the computation of

$$\mathrm{GCD}(U(x), V(x) - s). \tag{3.1}$$

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

THEOREM 3 (RESULTANT THEOREM [14]). *Given two polynomials $A(x, s)$, $B(x, s)$, their GCD will be nontrivial if and only if their resultant*

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

Thus the required residues $\{S_i\}$ will be the points at which $r(s)$ is zero, i.e., the roots of $r(s)$. 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 [15] has given an efficient modular algorithm for computing resultants. From his work we see that if $\deg(U) = n$, then $\deg(\mathrm{Res}(U, V - s)) \leqslant n$.

The basic steps of Collins' algorithm, applied to this case are:

(1) for $i := 0$ until n do
    begin
        substitute a new value S_i for s in $V(x) - s$;
        compute a univariate resultant: $r_i := \text{Res}(U, V - S_i)$;
    end

(2) Interpolate the values $\{r_i\}$ at the points $\{S_i\}$ to get a polynomial $r(s)$. □

This procedure can be carried out in $O(n^3 + n^2 \log p)$ steps. Collins' Reduced PRS Algorithm provides an alternative method for computing resultants. However, it would have the same $O(n^3 + n^2 \log p)$ bound as the method above. An advantage to the modular method is that if any of the r_i are zero they disclose a residue S_i which will give a nontrivial factor without having to do any more work.

Since it costs $O(n^3)$ to compute the resultant for each polynomial $V(x)$, we want to avoid having to do it once for each $V_i(x)$. We will only have to do this if the first nontrivial basis polynomial $V_1(x)$ does not yield all the k irreducible factors. Relation (2.6) shows that this will occur if two or more of the residues S_i of $V_1(x)$ are the same. If the S_i are independent random variables in the range $0 \leqslant S_i \leqslant p - 1$, then the probability that they will be distinct is:

$$\text{prob}(p, k) = \prod_{i=0}^{k-1} (p - i)/p^k.$$

If $k << p$, then this probability is close to 1.

Therefore, the probability that more than one $V(x)$ must be used to find all the factors of $U(x)$ is very low if the field characteristic p is large. In practice, it is very unusual for the first nontrivial basis polynomial $V_1(x)$ not to provide all the factors if p is large with respect to k and n.

We have reduced the factoring problem to that of finding the roots of a polynomial $r(s)$ of degree n in the finite field Z_p. To solve this problem, we can use the well-known result:

LEMMA 4 [16, p. 128]. *In a finite field* GF(q); $b(s, m) = s^{q^m} - s$, *is the product of all monic irreducible polynomials of degree which divides* m. □

A special case,

$$b(s, 1) = s^p - s,$$

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

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

If there are k factors of $U(x)$, $g(s)$ will be of degree k and will have multiple roots if the basis polynomial $V(x)$ does not yield all the irreducible factors. We can test for these multiple roots and remove them using the construction (2.1). The multiple roots will also tell us which of the factors given by $V(x) - S_i$ are degenerate. This information will aid in application of other basis polynomials $V(x)$, should it prove necessary.

One method of finding the roots of $g(s)$ is to test each element of the field. Previously, we had to compute a GCD each time we tested a field element at a cost of $O(n^2 + n \log p)$ steps. Now we need only evaluate the polynomial $g(s)$ at a cost of $2k$ operations. So the time for this root finding is bounded by $O(np)$ steps.

In order for this evaluation not to dominate the previously largest step of the algorithm,

$$n^3 + n^2 \log p \geqslant cnp,$$

for some constant c, or $p = O(n^2)$. Therefore, this method is appropriate for medium sized primes. We will call the procedure described in this section the Modified Berlekamp Algorithm. In summary we can state:

THEOREM 5. *Most monic square-free polynomials of degree n can be factored over a finite field Z_p in $O(n^3 + n^2 \log p + R(n))$ steps, if $p >> n$ and $R(n)$ is the number of steps needed 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 [9] discusses a probabilistic method for finding the roots of a polynomial in a finite field Z_p. He gives the timing of the method as $O(p^{1/4} \log p^{2/3})$. In this section we will discuss ways in which this method may be improved upon.

We can observe that if we are free to choose the prime characteristic (as we are in this case), then we can choose fields which expedite the discovery of the roots of the polynomial.

In particular it is useful to choose primes p such that $p - 1$ is highly composite (e.g. $p = L \cdot 2^l + 1$, where L is small and $l \simeq L$). In this case we can employ a "divide and conquer" technique to refine the multiplicative subgroup of Z_p containing a root. At most $l \leqslant \log p$ such refinements are necessary to find a root. This means that the time to find the roots of the polynomial is a function of $\log p$ rather than of p.

We assume that the roots of $g(s)$ are nonzero and distinct. This can easily be checked by the construction (2.1).

A special case of Lemma 4 is that $c(s) = s^{p-1} - 1$ is the product of all nonzero linear terms over Z_p. Its roots are members of the multiplicative subgroup Z_p^*. Members of this group are called $p - 1$st roots of unity and a generator of the group is a primitive $p - 1$st root of unity.

Note that $c(s)$ factors as

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

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 $l + 1$ factors of the form:

$$c_i(s) = s^{L \cdot 2^{l-i}} - 1, \qquad 0 \leqslant i \leqslant l,$$

$c_i(s)$ is the product of all $(p - 1)/2^i$th roots of unity of Z_p.

Using this fact, we can separate the roots of a polynomial $g(s)$ into those roots

which are $L \cdot 2^{l-i-1}$st roots of unity and those which are not, by taking

$$r_{i-1}(s) = \mathrm{GCD}(g(s), s^{L \cdot 2^{l-i-1}} - 1).$$

Now we can describe the refinement process. Let $r_{i-1}(s)$ be a product of $L \cdot 2^{l-i+1} = (p-1)/2^{i-1}$st roots of unity. Then

$$r_i(s) = \mathrm{GCD}(r_{i-1}(s), s^{L \cdot 2^{l-i}} - 1) \tag{4.1}$$

is a product of all roots of $r_i(s)$ which are $(p-1)/2^i$th roots of unity. If

$$r_{i-1}(s) = r_i(s) \cdot t_i(s) \tag{4.2}$$

then $t_i(s)$ is the product of all roots of $r_{i-1}(s)$ which are $(p-1)/2^{i-1}$st roots of unity but not $(p-1)/2^i$th roots of unity. This means that t_i has the form

$$t_i(s) = \prod (s - w^{j \cdot 2^{i-1}}),$$

where w is a primitive $p-1$st root of unity and the j's are odd. If ψ is another (not necessarily distinct) primitive $p-1$st root of unity, then $\psi = w^m$ where $(p-1, m) = 1$. In particular, m is odd. Consider forming

$$t_i'(s) = \prod (s - w^{j \cdot 2^{i-1}} \cdot \psi^{2^{i-1}}) = \prod (s - w^{(j+m)2^{i-1}}) = \prod (s - w^{2^i \cdot (j+m)/2}), \tag{4.3}$$

since j and m are odd.

Thus the roots of $t_i'(s)$ are $(p-1)/2^i$th roots of unity and the refinement process can be applied recursively to $r_i(s)$ and $t_i'(s)$. Once the roots of $t_i'(s)$ are found the roots of $t_i(s)$ can be computed by dividing by $\psi^{2^{i-1}}$.

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

If

$$t_i(s) = \sum_{j=0}^{k} a_j s^j, \qquad a_k = 1,$$

then expanding (4.3) we see

$$t_i'(s) = \sum_{j=0}^{k} a_j x^j \psi^{(k-j)2^{i-1}}. \tag{4.4}$$

This means the transformation of $t_i(s)$ to $t_i'(s)$ can be performed in k operations given the coefficients of $t_i(s)$ and $\psi^{2^{i-1}}$. We will call the procedure to perform this transformation, Convert.

Now we can state an algorithm to find the roots of a polynomial based on relations (4.1), (4.2) and (4.4).

Algorithm: Roots(r, ψ, i, H).

Input: (1) the polynomial $r(s)$;

(2) ψ a primitive $(p-1)/2^i$th root of unity;

(3) an integer i;

(4) H a list of polynomials of the form:

$$h_i = s^{(p-1)/2^i} - 1 \bmod r(s), \qquad 0 < i \leqslant l.$$

Output: The roots of $r(s)$ in Z_p.
Steps.
(1) Basis: if $\deg(r) = 1$ then return $\{-r_0\}$;
(2) Direct-Search: else if $2 \nmid (p-1)/2^i$
then return Direct-search $(r, \psi, (p-1)/2^i)$;
(3) Separation Root: else begin
$g(s) := \text{GCD}(r(s), h_i(s))$;
$f(s) := \text{Convert}(r(s)/g(s), \psi)$;
$R := \varnothing$;
(4) Recursion: if $\deg(g) > 0$
then $R := R \cup \text{Roots}(g, \psi^2, i+1, H)$;
if $\deg(f) > 0$
then $R := R \cup \text{Roots}(f, \psi^2, i+1, H)/\psi$;
Return R
end. □

The algorithm would be invoked as:

$$R := \text{Roots}(r(s), \psi, 1, H);$$

where ψ is a primitive $p-1$st root of unity. The operation $\cup$ is union.

The algorithm Direct-search $(r, \psi, (p-1)/2^i)$ is invoked to find the roots of $r(s)$ in the multiplicative group of ψ by direct evaluation. Since there are only L members of this group and L is chosen to be small, the operation does not take long to do. Also if L is composite, 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 polynomials.

Timing of the Algorithm. The polynomials $H_i = s^{(p-1)/2^i} - 1 \bmod r(s)$, $1 \leqslant i \leqslant l$, can be computed in $O(k^2 \log p)$ steps where $k = \deg(r)$. Similarly, each GCD operation can be performed in $O(k^2 + k \log p)$ steps. The worse case situation for the algorithm is when the refinement of the subgroups does not separate the roots at all. In this case the Direct-search algorithm must be used to separate them. There can be at most $\log p$ refinements and so the total cost is $O(k^2 \log p + k \log^2 p + kL)$ steps.

In summary we have:

THEOREM 6. *In a finite field Z_p where $p = L \cdot 2^l + 1$ and $L \simeq l$, the roots of a polynomial of degree k can be computed in $O(k^2 \log p + k \log^2 p)$ steps.* □

COROLLARY 7. *Over such a field a polynomial of degree n can be factored in $O(n^3 + n^2 \log p + n \log^2 p)$ steps, if $p >> n$.* □

V. Faster Methods. The removal of the limit on the size of the field for the algorithm raises the possibility that the number of steps can be pared down further. This can be done if we are considering the problem of factoring polynomials of very high degree.

The timing estimates for the algorithms which we have used so far are reasonably consistent with the observed behavior of the algorithms in real algebraic manipulation systems for the size of problems usually encountered. However, if we are dealing with very large problems, we can use a set of algorithms with slower growing timing functions.

It can be shown (see for example [17]) that two polynomials of degree n can be multiplied together in $O(n \log n)$ steps. Similarly, a polynomial of degree $2n$ can be divided by one of degree n in $O(n \log n)$ steps and the GCD of two polynomials of degree n can be computed in $O(n \log^2 n + n \log p)$ field operations.

If we allow such algorithms to be considered, then we might ask if they can be of help? The answer is yes, but we must extensively reformulate the algorithm. The bounding step now becomes the matrix operations. Although Strassen [18] has shown that matrices can be multiplied or triangularized in $O(n^{2.81})$ steps, we can even improve on this limit.

Our first observation is that we can find a partial factoring of a polynomial using a method due to Golomb et al. [19] based on Lemma 4. This partitions the factors of a polynomial into products of all factors of the same degree. The following algorithm achieves this:

Algorithm: Distinct-Degree Factors (U).

Input: the polynomial $U(x) \in Z_p[x]$.

Output: the distinct-degree factors $d_i(x)$

$$U(x) = \prod d_i(x), \quad d_i(x) = \prod f_{i,j}(x), \quad \text{where } \deg(f_{i,j}) = c, \forall j.$$

Steps.

(1) Initialization: $h(x) := x^p \bmod U(x)$;
$g(x) := i := j := 1$;

(2) Iteration: while $j \leqslant \deg(U)/2$ do
begin
$h(x) = h(x) \uparrow p \bmod U(x)$;
$j := j + 1$;
$d_i(x) := \mathrm{GCD}(h(x) - x, U(x))$;

Nontrivial Factors: if $d_i(x) \neq$ then
begin
$U(x) := U(x)/d_i(x)$;
$h(x) := h(x) \bmod U(x)$;
$i := i + 1$
end
end

(3) Completion: if $U(x) \neq 1$ then $d_i(x) := U(x)$; □

By Lemma 4 at the jth iteration $g(x)$ is the product of all polynomials in $Z_p[x]$ of degree dividing j. Since all lower degree factors have been removed from $U(x)$, the GCD operation finds all factors of U of degree j. Using the classical methods of Sections II–IV the algorithm can be shown to take $O(n^3 + n^2 \log p)$ steps. However, we have:

THEOREM 7. *The Distinct-Degree Factors algorithm can find the factors of a polynomial of degree n in $O(n^2 \log^2 n + n^2 \log n \log p)$ steps.*

Proof. $x^p \bmod U(x)$ can be built up using the repeated squaring technique in $O(n \log n \log p)$ steps. The GCD operation can be performed in $O(n \log^2 n + n \log p)$

steps [17]. Since the loop may be executed at most $n/2$ times, the bound is $O(n^2 \log^2 n + n^2 \log n \log p)$ steps. □

VI. Splitting Distinct-Degree Factors. In the previous section we reduced the factoring problem to that of separating products of the same degree. We are given

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

where $\deg(d) = n = k \cdot m$, $\deg(f_i) = m$, $1 \leqslant i \leqslant k$.

If we can find a monic irreducible polynomial $r(x) \in Z_p[x]$, $\deg(r) = m$, then we can reduce $d(x) \bmod (r(x))^i$ to compute the polynomial

$$F(y) = \sum_{k=0}^{k} S_i(x) y^i,$$

where

$$d_i'(x) = S_i(x) \cdot (r(x))^i + d_{i-1}'(x) \quad \text{for } i = k, k-1, \ldots, 1,$$

$$d_k'(x) = d(x).$$

Now $F(y)$ is a monic polynomial of degree k with coefficients $S_i(x)$ in the finite field $Z_p[x]/(r(x)) \simeq \mathrm{GF}(p^m)$. If we can find the roots $\{t_i\}$ of $F(y)$ in the field, then we can find the factors f_i of $d(x)$.

Since

$$F(y) = \prod_{i=0}^{k} (y - t_i),$$

this implies

$$d(x) = \prod_{i=0}^{k} (r(x) - t_i(x)).$$

Unfortunately, we can no longer apply the Roots Algorithm of Section IV, since $p^m - 1$ is not highly composite. Instead we can use a construction of Berlekamp's [9] based on:

LEMMA 8. *In* $\mathrm{GF}(p^m)$ *the product of all linear factors is*

$$y^{p^m} - y = \prod_{s=0}^{p-1} (\mathrm{Tr}(y) - s), \tag{6.1}$$

where $\mathrm{Tr}(y)$ *is the trace*:

$$\mathrm{Tr}(y) = \sum_{i=0}^{m-1} y^{p^i}. \tag{6.2}$$

Then

$$F(y) = \prod_{s=0}^{m-1} GCD(F(y), \mathrm{Tr}(y) - s). \quad \square$$

Therefore, we can find a factorization of $F(y)$ by checking:

$$G(y) = \mathrm{GCD}(f(y), \mathrm{Tr}(y) - s), \qquad 0 \leqslant s < p. \tag{6.4}$$

We can find the $\{S_i\}$ which will yield nontrivial factors by employing the resultant construction of Section III. This will give

$$\mathrm{Res}(f(y), \mathrm{Tr}(y) - s) = r(s, x) = \sum_{i=0}^{m-1} C_i(s) x^i$$

where

$$C_i(s) = \sum_{i=0}^{k} C_{ij} s^j, \qquad C_{ij} \in Z_p.$$

The set of common zeros of the $\{C_i(s)\}$ will be the zeros of the resultant $r(s, x)$. This can be computed by forming:

$$C(s) = C_0(s), \qquad C(s) = \mathrm{GCD}(C(s), C_i(s)) \quad \text{for } i = 1, \ldots, m-1.$$

The zeros of $C(s)$ a polynomial over Z_p can now be computed using the Roots Algorithm of Section IV.

Each trace $\mathrm{Tr}(y) - s$ for fixed s, $0 \leqslant s < p$, contains p^{m-1} roots of the field $\mathrm{GF}(p^m)$.

This means that in general (6.4) will not yield just one root but a factor of $F(y)$. The worst case situation occurs when all the roots of $F(y)$ fall into the set of one trace, i.e., $F(y) | \mathrm{Tr}(y) - s$. The ideal situation occurs when each set contains only one root of $F(y)$. When this happens, all the roots can be found by applying (6.4) $k-1$ times. If the roots are independent random variables, the optimum situation will occur with probability:

$$\mathrm{prob}(p, k) = \prod_{i=0}^{k-1} (p - i)/p^k.$$

If $k << p$ and for large p, this will be very close to 1. This implies that with a very high probability only one trace must be computed to find all the roots of $F(y)$.

If only one trace does not yield all the roots then, as Berlekamp shows (6.1) may be rewritten:

$$y^{p^m} - y = \prod_{s=0}^{p-1} (\mathrm{Tr}(\alpha^j y) - s), \qquad 0 < j < m,$$

where α is a root of an irreducible polynomial of degree m over Z_p. In our case, $\mathrm{GF}(p^m) = Z_p[x]/(r(x))$; and so, we can choose the irreducible polynomial to be $r(y)$ and a suitable root will be $\alpha = x$. Thus we can find factors of $F(y)$ by checking:

$$\mathrm{GCD}(F(y), \mathrm{Tr}(x^j y) - s), \qquad 0 < j < m, \; 0 \leqslant s < p. \tag{6.5}$$

Berlekamp shows that not all j in (6.5) can give a trivial factorization of $F(y)$; and therefore, all roots of $F(y)$ must be eventually found by (6.5).

Timing of the Algorithm. For arithmetic operations on $a, b \in Z_p[x]/(r(x))$:

$a + b$ uses $O(m)$	basic steps,
$a * b$ uses $O(m \log m)$	basic steps,
a^{-1} uses $O(m \log^2 m + m \log p)$	basic steps.

For $A(y), B(y) \in \mathrm{GF}(p^m)[y]$, $\deg(a) = \deg(b) = k$:

$A + B$ uses $O(km)$	basic steps,
$A * B$ uses $O(k \log k(m \log m))$	basic steps,
$\mathrm{GCD}(A, B)$ uses $O(k \log^2 k(m \log m) + k(m \log^2 m + m \log p))$	basic steps.

So to form $y^p \bmod F(y)$ uses $O(m \cdot k \cdot \log k \cdot \log m \cdot \log p)$ basic steps. From this

$$\mathrm{Tr}(y) = \sum_{i=0}^{m-1} y^{p^i} \mod F(y)$$

uses $O((\log p + m)m \cdot k \cdot \log k \cdot \log m)$ basic steps.

Computing the resultant $\mathrm{Res}(F(y), \mathrm{Tr}(y) - s)$ is no harder than forming k GCDs and so takes

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

The m GCDs of the coefficient polynomials takes at most $O(mk(\log^2 k + \log p))$ basic steps and as shown in Section IV the Roots Algorithm can be performed in $O(k^2 \log p + k \log^2 p)$ steps.

Assuming that only one trace need be computed, the step which bounds the algorithm is computing the resultant. In summary, we have:

THEOREM 9. *Over the finite field* $\mathrm{GF}(p^m)$ *where* $p = L \cdot 2^l + 1$ *and* $l \simeq L$ *the roots of a polynomial of degree k can be found in* $O(k^2 m(\log^2 k \log m + \log^2 m + \log p))$ *basic steps.* □

COROLLARY 10. *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 found in* $O(n^2(\log^2 n + \log p))$ *basic steps, where* $\deg(d) = n$.

Proof. The time for finding the roots in $\mathrm{GF}(p^m)$ is maximized when $k = O(n)$ and $m \simeq O(1)$ since $n = km$. Combining this result with that of Theorem 7, we get:

COROLLARY 11. *The factors of a polynomial of degree n over a finite field* Z_p *can be found in* $O(n^2(\log^2 n + \log n \log p))$ *steps, where* $p - 1$ *is highly composite.* □

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 $O(n^3 + n^2 \log p)$ steps. This is the same bound as achieved by the methods of Sections II to IV.

As a nonlinear lower bound on the factoring problem we have:

THEOREM 12. *At least* $n \log(n/e)$ *rational multiplications are required 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 the extreme case where all factors are linear, Strassen [20] has shown that such a product requires $n \log(n/e)$ 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 Strassen's work it could be conjectured that a bound of

$$n(1 - 1/p) \log\left(\frac{n(1 - 1/p)}{e}\right)$$

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 13 [16]. *In the finite field* GF(q), *e is a primitive* $q-1$*st root of unity if and only if*

$$e^{(q-1)/a_i} \neq 1 \mod q$$

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 $e^{(q-1)/a_i}$, there can be taken 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 14 [21, p. 221]. *Let K be a field and n an integer* $\geqslant 2$. *Let* $a \in K$ *and* $a \neq 0$. *If for all prime divisors c of n,* $a \notin K^c$ (*a does not have a cth 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 cth 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

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

We see that

$$\begin{aligned} x^p &\equiv a x^{p-c} \mod x^c - a, \\ &\equiv a^l x^{p-lc} \mod x^c - a, \end{aligned}$$

where $l = \lfloor p/c \rfloor$.

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

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 $O(n^2 \log np)$ 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 were programmed using an implementation of the SAC-1 [22] algebraic manipulation system on a Honeywell 6050 at the University of Waterloo. SAC-1 contains the original Berlekamp algorithm and the distinct degree method as part of the system. The four algorithms tested were:

(i) the original Berlekamp algorithm;

(ii) the modified method, computing roots of the resultant by evaluating at sufficiently many points in the field;

(iii) the modified method using the Roots Algorithm;

(iv) the distinct degree factoring method (DDF).

Some sample times for these four methods applied to a polynomial of degree 14

for various primes are given in Table I. The top of the table shows that for small and moderate sized primes, the modifications are slightly slower than Berlekamp's algorithm. This is to be expected since they have the overhead of computing a resultant. However, for primes of the order 100, the modifications can already be faster than the original algorithm. Above this point the Berlekamp algorithm may occasionally be faster than the modifications. This occurs when the algorithm is "lucky" enough to find the factors after only trying a few GCDs.

The lower part of the table shows the results for large primes. Blank entries in the table indicate the program ran out of time (2 mins.) before it found all the factors. It can be seen that the Roots Algorithm is no slower than the modified method for small primes and is significantly faster for large primes. Also the Roots modification is always within a *factor of two* of the speed of the distinct degree method even though the latter gives less information. The time for the Roots modification using a prime $\sim 10^{10}$ is only four times that using the prime 17.

A problem posed by Johnson and Graham [23] has been used as a bench mark for algebraic manipulation systems. This problem involves computing the resultant of two bivariate polynomials, factoring the result and finding the real roots of the factors. The most time-consuming part of the problem is the factoring operation since it involves a polynomial of degree 40 with integer coefficients in the range $\pm 10^{23}$. The result is four polynomials of degree 10 with integer coefficients in the range $\pm 10^{6}$.

The fastest computers, with the most powerful routines, based on Berlekamp-Hensel algorithms used more than five minutes of computer time to factor this polynomial [24], [25]. The SAC-1 implementation of the Berlekamp-Hensel algorithms [6] used 15 minutes to factor the polynomial on a Honeywell 6050. However, a special purpose SAC-1 program, on a Honeywell 6050, using the Roots modification, working modulo $1790967809 = 427 \cdot 2^{22} + 1$, was able to discover the factors in under three minutes. The distinct degree method operating with the same prime required 1.5 minutes to discover the degrees of the factors.

These tests raise the possibility of dispensing with Hensel's lemma when using the modular factoring to find factors over the integers. The prime p would be chosen to majorize any coefficient of a factor and combinations of the resulting Z_p-factors could be tested as trial integer factors. This approach would be most reasonable if the chosen prime is less than the word size of the computer.

However, if the bound on the coefficients of factors is larger than the word size, we expect that a hybrid method would be more reasonable. In this case the overhead of doing multiprecision modular arithmetic throughout the course of the computation would slow the algorithm considerably. The hybrid would involve factoring modulo a word sized prime and then applying Hensel's lemma to the results. Since the Hensel construction would not have to lift the results as far as usual, this approach would be faster.

An additional benefit of factoring with large primes, is that in general, fewer irreducible factors are produced. This means that to find the factors over the integers fewer combinations of small factors need be tried.

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, 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 $= 5\cdot 2^{13} + 1$		42	7.96	3.85	1, 1, 1, 2, 2, 3, 4
1790967809 $= 427\cdot 2^{22} + 1$			10.8	6.5	1, 2, 4, 7
1811939329 $= 27\cdot 2^{26} + 1$			13.1	6.28	1, 1, 2, 3, 3, 4
1835008001 $= 875\cdot 2^{21} + 1$			14.8	6.28	1, 1, 2, 2, 3, 5
1863319553 $= 1777\cdot 2^{20} + 1$			14.5	6.62	1, 1, 5, 7

TABLE 1

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

1. R. H. RISCH, *Symbolic Integration of Elementary Functions,* Proc. 1968 IBM Summer Inst. on Symbolic and Algebraic Manipulation, IBM, R. Tobey (Editor), pp. 133–148.

2. B. F. CAVINESS, *On Canonical Forms and Simplification,* Ph.D. Thesis, Carnegie-Mellon Univ., 1967.

3. D. Y. Y. YUN, "On algorithms for solving systems of polynomial equations," *SIGSAM Bull.,* No. 27, ACM, New York, September 1973.

4. B. L. VAN DER WAERDEN, *Modern Algebra,* Vol. 1, 2nd rev. ed., Springer, Berlin, 1937; English transl., Ungar, New York, 1949. MR **10**, 587.

5. D. JORDAN, R. KAIN & L. CLAPP, "Symbolic factoring of polynomials in several variables," *Comm. ACM,* v. 9, 1966.

6. D. R. MUSSER, *Algorithms for Polynomial Factorization,* Ph.D. Thesis, Univ. of Wisconsin, 1971.

7. P. S. WANG & L. P. ROTHSCHILD, "Factoring polynomials over the integers," *SIGSAM Bull.,* No. 28, ACM, New York, December 1973.

8. E. R. BERLEKAMP, *Algebraic Coding Theory,* Chap. 6, McGraw-Hill, New York, 1968. MR **38** #6873.

9. E. R. BERLEKAMP, "Factoring polynomials over large finite fields," *Math. Comp.,* v. 24, 1970, pp. 713–735. MR **43** #1948.

10. J. D. LIPSON, *Chinese Remainder and Interpolation Algorithms,* Proc. 2nd SIGSAM Sympos., ACM, New York, 1971.

11. G. E. COLLINS, "Computing multiplicative inverses in GF(p)," *Math. Comp.,* v. 23, 1969, pp. 197–200. MR **39** #3676.

12. G. E. COLLINS, *Computing Time Analyses of Some Arithmetic and Algebraic Algorithms,* Proc. 1968 IBM Summer Inst. on Symbolic and Algebraic Computation.

13. D. E. KNUTH, *The Art of Computer Programming.* Vol. 2: *Seminumerical Algorithms,* Addison-Wesley, Reading, Mass., 1969. MR **44** #3531.

14. J. V. USPENSKY, *The Theory of Equations,* Chap. 11, McGraw-Hill, New York, 1948.

15. G. E. COLLINS, "The calculation of multivariate polynomial resultants," *J. Assoc. Comput. Mach.,* v. 18, 1971, pp. 515–532. MR **45** #7970.

16. A. A. ALBERT, *Fundamental Concepts of Higher Algebra,* Univ. of Chicago Press, Chicago, 1958. MR **20** #5190.

17. R. T. MOENCK, *Studies in Fast Algebraic Algorithms,* Ph.D. Thesis, Univ. of Toronto, 1973.

18. V. STRASSEN, "Gaussian elimination is not optimal," *Numer. Math.,* v. 13, 1969, pp. 354–356. MR **40** #2223.

19. S. GOLOMB, L. WELCH & A. HALES, "On the factorization of trinomials over GF(2)," JPL Memo 20-189 (July 1959)(as referred to in [13]).

20. V. STRASSEN, "Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten," *Numer. Math.,* v. 17, 1972/73, pp. 238–251. MR **48** #3296.

21. S. LANG, *Algebra,* Addison-Wesley, Reading, Mass., 1968. MR **33** #5416.

22. G. E. COLLINS, *The SAC-1 System: An Introduction and Survey,* Proc. 2nd SIGSAM Sympos., ACM, New York, 1971.

23. S. C. JOHNSON & R. L. GRAHAM, "Problem #7," *SIGSAM Bull.,* v. 8, No. 1, ACM, New York, February 1974, p. 4.

24. R. FATEMAN, J. MOSES & P. WANG, "Solution to problem #7 using MACSYMA," *SIGSAM Bull.,* v. 8, No. 2, ACM, New York, May 1974, pp. 14–16.

25. G. E. COLLINS, D. R. MUSSER & M. ROTHSTEIN, "SAC-1 solution of problem #7," *SIGSAM Bull.,* v. 8, No. 2, ACM, New York, May 1974, pp. 17–19.