Computer Sciences Department
The University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

ALGORITHMS FOR POLYNOMIAL FACTORIZATION*

by

David R. Musser†

Technical Report #134

September 1971

*Ph.D. thesis. Research supported by National Science Foundation grants GJ-239 and GJ-30125X, the Mathematics Research Center, and the Wisconsin Alumni Research Foundation.

† Present address: Department of Computer Sciences, The University of Texas, Austin, Texas 78732.

ALGORITHMS FOR POLYNOMIAL FACTORIZATION

BY

DAVID R. MUSSER

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN

1971

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor George E. Collins for all of the time and guidance which he contributed to the preparation of this thesis.

I would also like to acknowledge the support of the Mathematics Research Center, the National Science Foundation and the University of Wisconsin Alumni Research Foundation.

TABLE OF CONTENTS

ALGORITHMS FOR POLYNOMIAL FACTORIZATION

David R. Musser

ABSTRACT

Algorithms for factoring polynomials with arbitrarily large integer coefficients into their irreducible factors are described. These algorithms are based on the use of mod p factorizations and constructions based on Hensel's Lemma. The concept of an "abstract algorithm" is defined and used in the development of general algorithms which are applicable in numerous polynomial domains, including polynomials with either integer or finite field coefficients. Included is a new generalization of Hensel's p-adic construction which is applicable to multivariate factorization. For the case of univariate polynomials over the integers, full specifications are given for algorithms which have been implemented and tested using the SAC-1 System for Symbolic and Algebraic Calculation. Theoretical computing times for the univariate algorithms are also derived and empirical data obtained during tests of the algorithms are reported.

CHAPTER 1: PRELIMINARIES

1.1 Introduction

This thesis deals principally with computer algorithms for the factorization of polynomials in several variables with integer coefficients into polynomials which are irreducible over the integers. For the univariate case we shall present full specifications of algorithms for solving this problem which are based on the use of mod p factorizations and constructions based on Hensel's Lemma as suggested by Zassenhaus. These algorithms have been implemented and tested using the SAC-1 System for Symbolic and Algebraic Calculation. An analysis of the computing time of each of these algorithms is also included. Our discussion of the theoretical basis of these algorithms encompasses the multivariate case and also applies to other cases such as factorization of polynomials in several variables with coefficients from a finite field.

A theoretical solution to the factoring problem is provided by Kronecker's algorithm [VDW49,Section 25]. For the case of univariate polynomials with integer coefficients, the algorithm may be stated as follows:

To factor a given polynomial C(x) of degree n, let k be the greatest integer $\leq n/2$, choose k+1 integers $a_0,\ldots,a_k$, and let S_i be the set of all integral factors of $C(a_i)$, $0 \leq i \leq k$. Choose an element b_i from each S_i and compute by interpolation the unique polynomial A(x) of degree $\leq k$ such that $A(a_i) = b_i$, $0 \leq i \leq k$. Attempt to divide A(x) into C(x). If it divides exactly a factorization has been found and the method may be applied recursively to the

two factors. Otherwise, discard A(x) and try another choice of the b_i. If all of the choices are exhausted in this way, then C(x) is itself irreducible.

This method has been the basis for several computer algorithms for polynomial factorization and works well enough when the degree and coefficients of C are small. If the coefficients of C are large, however, then considerable time may be required just to factor the $C(a_i)$ into primes. And if the degree of C is large then the number of possible choices of the b_i may be enormous.

Although these obstacles are apparent to anyone who has ever tried to use the method to factor a polynomial of any size by hand, one might think that they would easily be overcome by the sheer speed of modern computers. This is not the case, however, as experience with the computer algorithms has shown. The reason is not difficult to see. Because of the number of tentative factors which must be considered, it is rather evident that the time required to perform the method is an exponential function of the degree of C. Also, the time required to factor an integer of length m (i.e. with m digits in some base) by the best methods known is an exponential function of m. Thus increases in either the degree or the length of the coefficients of C cause exponential growth of the computing time and any given speed of computation is easily overwhelmed.

There are a number of ways in which Kronecker's method can be speeded up, some of which, involving mod p factorizations for various small primes p, are discussed in [VDW49]. S. C. Johnson

suggests additional improvements in [JOH66]. However, none of these changes relieves the basic problem of exponential growth, and thus alternative methods have been sought in the past few years.

The principal breakthroughs in this area have been (1) the development of an efficient algorithm for mod p factorization by Berlekamp [BER68, Chapter 6] and (2) the suggestion by Zassenhaus [ZAS69] that one could combine Berlekamp's algorithm and Hensel's p-adic construction [VDW49, Section 76] to obtain a practical method of factoring polynomials with integer coefficients.

Berlekamp's algorithm opened the way to the design of algorithms based on mod p factorizations rather than on integer factorization. Knuth [KNU69, Section 4.6.2] sketches an algorithm based on the factorization of the given polynomial modulo several different primes and the construction of tentative factors using the Chinese Remainder Theorem. This method, viewed abstractly, has the same basic structure as Kronecker's method. Although the problem of factoring integers is replaced by that of factoring polynomials mod p, which is efficiently accomplished by Berlekamp's algorithm, there remains the problem of considering a huge number of tentative factors. The mod p - Chinese Remainder Theorem approach has been highly successful in a number of other areas, e.g. polynomial greatest common divisor calculation and solution of systems of linear equations, but it does not appear to be practical for the factorization problem.

Zassenhaus showed, however, that under certain conditions a construction based on "Hensel's Lemma" could be used to progress

from a mod p factorization to a corresponding factorization modulo any power of p. Taking p^j sufficiently large, we can determine from consideration of all mod p^j factorizations all true factorizations. The number of mod p^j factorizations is the same as the number of mod p factorizations.

In Chapter 2 we discuss in detail the theoretical background for polynomial factorization, including several "abstract" algorithms based on Hensel's Lemma. We have in many cases used abstract algorithms instead of theorems as our basic tool for the presentation of general theory, and this has permitted a smoother transition from theory to practical application.

Chapter 2 concludes with a discussion of a new "generalized Hensel algorithm" which is applicable to multivariate factorization and promises to yield algorithms which are far superior to Kronecker's algorithm.

Chapter 3 presents detailed specifications of algorithms for the case of univariate polynomials over the integers. The algorithms which are described have been implemented in the SAC-1 System for Symbolic and Algebraic Calculation, [COL71a], a system for performing operations on multivariate polynomials and rational functions with arbitrarily large coefficients. It is programmed almost entirely in ASA "Standard" Fortran and is thus extremely accessible and portable. The SAC-1 system's capabilities are discussed briefly in the remaining sections of this chapter. It should be noted here that the system makes use of mathematically sophisticated algorithms which are

generally far more efficient than the "obvious" algorithms, and the availability of these highly efficient algorithms for basic operations has simplified the task of implementing factorization algorithms tremendously.

Within Chapter 3 we have attempted to describe the factoring algorithms in an informal language which is largely independent of the idiosyncrasies of Fortran or the SAC-1 system. The descriptions are, though informal, hopefully precise and detailed enough that they can be programmed in other languages with little difficulty. For a completely precise statement of the algorithms, we refer the reader to the Fortran program listings in the appendix.

For each of the algorithms described in Chapter 3, we have included an analysis of its computing time. The methods used in these analyses, which are outlined in Section 1.4, permit computing time bounds to be expressed independently of the computer on which the algorithms are implemented. The comparison of such bounds for several alternative algorithms for the same operation often proves the superiority of one of the algorithms, making it unnecessary to implement and empirically test them all. In a number of cases in Chapter 3 we point out how such comparisons have determined our choice of a particular algorithm to implement.

We conclude in Section 3.6 with the results of some empirical tests which demonstrate the practicality of application of the univariate algorithms to polynomials with large degree and large coefficients.

1.2 Polynomials

We shall use, for the most part, standard terminology and notation for polynomials. We briefly summarize these here and discuss the SAC-1 representation of, and operations on polynomials.

A polynomial

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

with $a_n \neq 0$ is said to have degree n, leading coefficient a_n, and trailing coefficient (or constant term) a_0; we write

$$\deg A = n, \quad \text{ldcf } A = a_n, \quad \text{tlcf } A = a_0.$$

By convention, we define

$$\deg 0 = -\infty, \quad \text{ldcf } 0 = 0, \quad \text{tlcf } 0 = 0.$$

We say A(x) is a monic polynomial if ldcf A = 1.

The coefficients a_k belong in general to some ring $\mathcal{R}$; we say that A is a polynomial (in x) over $\mathcal{R}$. $\mathcal{R}$ is permitted to be a polynomial ring in other variables, say $\mathcal{R} = R[v,w]$. In this case A is actually a polynomial in v, w and x although we have explicitly shown only the variable x. We may write

$$a_k = a_k(w) = \sum_j a_{jk} w^j$$

where the a_{jk} are members of the polynomial ring R[v]:

$$a_{jk} = a_{jk}(v) = \sum_i a_{ijk} v^i$$

where the a_{ijk} belong to the ring R. Thus

$$A(x) = A(v,w,x) = \sum_{i,j,k} a_{ijk} v^i w^j x^k, \qquad (1)$$

but it is more useful, in both theory and practical computation, to consider A to be a polynomial in one variable x (called the main variable) whose coefficients are polynomials in w whose coefficients are polynomials in v, over R. The theoretical value of this has long been understood and the practical value in computation has been demonstrated by the SAC-1 Polynomial System [COL68b], which uses a recursive canonical representation of multivariate polynomials by lists and recursive algorithms for many operations.

We define the numerical coefficients of a univariate polynomial to be just its coefficients and of a multivariate polynomial

$$A(x_1,\ldots,x_n,x) = \sum_i a_i(x_1,\ldots,x_n)x^i$$

recursively to be the numerical coefficients of its coefficients $a_i(x_1,\ldots,x_n)$. For example, the numerical coefficients of A(v,w,x) in (1) are the a_{ijk}.

The SAC-1 Polynomial System provides operations on polynomials in arbitrarily many variables, whose numerical coefficients are integers. The integer coefficients may be arbitrarily large since operations on them are performed using the SAC-1 infinite precision arithmetic system [COL68a]. The polynomial operations provided include addition, subtraction, multiplication, division, substitution, differentiation and g.c.d. calculation. Similar capabilities are provided for multivariate polynomials with numerical coefficients from a finite field GF(p) by the SAC-1 Modular Arithmetic

System [COL69a].

We thus assume the ability to perform these operations without discussion of the details of the algorithms used. In Section 1.4 we shall make some assumptions about the efficiency with which some of these operations may be performed; these assumptions have been shown elsewhere to be true of the SAC-1 algorithms.

1.3 Algorithms

The concept of "algorithm" and the notational conventions for expressing algorithms which we shall use are very similar to those established by Knuth in [KNU68]. The main difference is that our definition of algorithm encompasses non-effective computational methods, allowing what we shall call abstract algorithms to play the primary role in the presentation of the theoretical background to practical factoring methods. An example of such an abstract algorithm is:

Algorithm D (Division of polynomials over a ring). Let $\mathcal{R}$ be a commutative ring with identity. Given polynomials A, B $\in \mathcal{R}[x]$ with ldcf B a unit of $\mathcal{R}$, this algorithm computes polynomials Q, R $\in \mathcal{R}[x]$ such that

$$A = BQ+R \text{ and } \deg R < \deg B.$$

(1) Set $Q \leftarrow 0$ and $R \leftarrow A$.

(2) (Now $A = BQ+R$.) If $\deg R < \deg B$, exit.

(3) Set $n \leftarrow \deg R - \deg B$, $T \leftarrow (\text{ldcf } R)(\text{ldcf } B)^{-1}x^n$, $Q \leftarrow Q+T$, $R \leftarrow R-TB$ (this reduces the degree of R), and go to (2).

This simple example illustrates most of the features of the concept and style of the algorithms appearing in Chapter 2. The algorithm begins with the definition of the domain or domains in which the inputs and outputs lie. If one or more of these domains is, as in the case of Algorithm D, an abstract set or algebraic system such as a ring or integral domain, then we shall call the

algorithm itself an abstract algorithm. Accompanying the inputs is some list of assertions involving them; we refer to these as the input assumptions. Similarly, the output generally has accompanying output assertions.

Following these definitions and assertions about the inputs and outputs is a series of numbered steps. Each step consists of one or more imperative statements, some of which are conditional. The language used is mainly that of conventional mathematics, but there are two important exceptions. The first is the use of the replacement operation, denoted by "$\leftarrow$". For example step (1) means replace the current value of Q by 0 and the current value of R by that of A. Here Q and R were previously undefined, so this step gives them initial values. In step (3), however, the values of Q and R are changed by the replacement operations "$Q \leftarrow Q+T$" and "$R \leftarrow R-TB$".

Thus each variable which appears in an algorithm generally takes on a sequence of values. Later in this section we shall give a precise formal definition of algorithm in which the replacement operation is recast in terms of the conventional mathematical concept of a sequence.

Ordinarily an algorithm is executed in sequential order of its steps; however, this may be altered by the use of a "go to" statement, as in step (3). This is the second important deviation from the language of conventional mathematics. Of course, when the execution is directed by means of a "go to" statement back to a

previous step, then a sequence of steps may be executed repeatedly until some condition is satisfied causing transfer to another step or causing termination of the algorithm (which will be indicated by the imperative "exit"). The fact that the computation does terminate in a finite number of steps for all inputs must be proved. In the case of Algorithm D a proof is indicated in the parenthetical assertion in step (3): by the choice of the term T of the quotient polynomial Q, both R and TB have the same leading coefficient, hence the new value of R, R_1 = R-TB, is of smaller degree than that of R, and thus the condition tested in step (2) must eventually be satisfied.

We shall require this finiteness property (finiteness of the number of steps executed) as part of our definition of algorithms. We shall not, however, require effectiveness; that is, we shall not require a proof that each individual step can be done in a finite amount of time by some (real or theoretical) computing machine. For abstract algorithms this would require further assumptions about the existence of effective algorithms for basic operations in the abstract domain of the inputs. The study of such questions has been undertaken with interesting results elswhere (see, for example, [RAB60]) but lies outside the scope of our objective in Chapter 2: the presentation, via abstract algorithms, of the common theory behind a number of algorithms. In Chapter 3, in contrast, we replace the abstract systems with particular systems, such as the ring of integers, for which effective algorithms for the basic

operations are known. In the "concrete" algorithms which are thereby obtained, it is obvious that the property of effectiveness holds.

If we do not require effectiveness in our abstract algorithms, the reader may well ask, by what criteria do we construct them? For we could in some steps of our algorithms merely cite the existence of some quantity without any indication of a method of constructing the quantity. While this has been done in one case in Chapter 2 (Algorithm 2.2F, a factorization algorithm for a Euclidean ring), all of the other algorithms there have been written with the purpose of generalizing methods which are known not just to be effective in particular domains, but to be "very effective", or "efficient" methods. This is meant in the sense that each step of the abstract algorithm is of sufficient simplicity that there are known to be efficient algorithms for carrying it out in at least one particular domain. In Algorithm D, for example, each step involves only simple arithmetic operations, for which efficient algorithms are known, when R is the ring of integers, or the rational number field, or a finite field. (The vague idea of "efficiency" is rendered more precise in Section 1.4.) That most of the abstract algorithms presented do satisfy this purpose is, it is hoped, adequately demonstrated in Chapter 3.

Besides the proof of termination, which is necessary to show that we do indeed have an algorithm, we are also interested in

proving the validity of the algorithm: that when applied to inputs which satisfy the input assumptions, the algorithm produces outputs which satisfy the output assertions. The method of proof used for most of the algorithms in Chapter 2 is based on the method of "inductive assertions" described in [FLO67] and [KNU68], Section 1.2.1. A formal description of the method will be given later in this section; but first we shall discuss it informally in terms of Algorithm D. The basic idea of the method is to associate with some or all of the steps or substeps of the algorithm assertions about the current state of the computation, and to prove that each assertion is true each time control reaches the corresponding step, under the assumption that the previously encountered assertions are true. If this can be done in such a way that the assertions associated with the first step are the input assumptions and those associated with the terminal step(s) are the output assertions, then the algorithm is necessarily valid, by induction on the number of steps performed.

In applying the method we have usually not attempted to list all of the assertions which actually hold at each step; in general we have tried to maintain about the same degree of explicitness as is usual in a conventional proof of a theorem. In Algorithm D, we have included only one assertion, in step (2), for the purpose of proving validity (the assertion in step (3) was included for the sake of proving termination, as discussed previously). It is trivial that this assertion, $A = BQ+R$, is true the first time step (2) is executed. Assuming it is true at a given execution of step (2), it

may be shown to be true at the next execution as follows: let $Q_1 = Q+T$ and $R_1 = R-TB$; then $BQ_1+R_1 = B(Q+T)+R-TB = BQ+R = A$; since Q is set to Q_1 and R to R_1 in step (3), the assertion $A = BQ+R$ still holds when step (2) is reached again.

The assertions associated with the steps of an algorithm, along with the input and output assertions, comprise an interpretation of the algorithm.

In order to have a rigorous basis for proofs of the validity of algorithms we shall now give formal definitions of algorithms and of interpretations of algorithms.

The following definitions are from [KNU68], pp. 7-8.

a. A computational method is a quadruple (S,I,Ω,f) in which:

S is a set (called the state set);

$I \subset S$ (the input set);

$\Omega \subset S$ (the output set);

f is a function from S into S (the computational rule) such that $f(q) = q$ for all $q \in \Omega$.

b. Each input x in I defines a computational sequence, $x_0,x_1,x_2,\ldots,$ as follows:

$x_0 = x$ and $x_{k+1} = f(x_k)$ for $k \geq 0$.

c. The computational sequence is said to terminate in k steps if k is the smallest integer for which $x_k \in \Omega$, and in this case is said to produce output x_k from x. (Note that if $x_k \in \Omega$, so is x_{k+1}, since $x_{k+1} = x_k$ in such a case.)

d. Some computational sequences may never terminate; an algorithm is a computational method which terminates in finitely many steps for all x in I.

Figure 1 shows Algorithm D rewritten to fit the above definitions. Knuth goes on to modify these definitions so as to include the property of effectiveness (essentially in terms of Markov algorithms), but the definitions as given here fit our needs precisely. Knuth does not give a formal discussion of validity proofs, but, as we shall now show, this can easily be done in the framework of the above definitions.

The following notation will be useful for this purpose:

$f^{(0)}(x) = x$ and $f^{(k+1)}(x) = f(f^{(k)}(x))$ for all $k \geq 0$;

$S^{(k)} = \{y \in S : y = f^{(k)}(x) \text{ for some } x \in I\}$;

$S^{\infty} = \{y : y \in S^{(k)} \text{ for some } k \geq 0\}$.

$S^{(k)}$ is the set of all states reachable from an input in k steps and S^{∞} is the set of all states reachable in some finite number of steps.

We formally define an interpretation of a computational method (S,I,Ω,f) to be a predicate defined on S^{∞}. Such an interpretation P is said to be valid if

1) $P(x)$ is true for all $x \in I$, and

2) $P(x) \rightarrow P(f(x))$ for all $x \in S^{\infty}$.

Theorem V. If P is a valid interpretation of a computational method (S,I,Ω,f) then $P(x)$ is true for all $x \in S^{\infty}$.

Proof: Let $x \in S^{\infty}$. Then for some $k \geq 0$ and some $x_0 \in I$, $x = f^{(k)}(x_0)$. If $k \neq 0$, then $x = x_0$, hence $x \in I$, hence $P(x)$ is true.

Let $k > 0$. Then $x = f(f^{(k-1)}(x_0)) = f(x')$ where $x' = f^{(k-1)}(x_0)$. We may assume by induction on k that $P(x')$ is true. From 2) above we have $P(x') \rightarrow P(f(x'))$, hence $P(x)$ is true.

An interpretation of the division algorithm is given in Figure 2. An informal proof of the validity of this interpretation has already been given.

A <u>termination</u> function for a computational method is a function $\emptyset$ from S^∞ into a well-ordered set (W,Θ) such that if $x \in S^\infty - \Omega$ then $\emptyset(f(x)) \Theta \emptyset(x)$.

<u>Theorem T</u>. If a computational method has a termination function, it is an algorithm.

<u>Proof</u>: Let $\emptyset$ be a termination function for (S,I,Ω,f). Given any $x \in I$ we must show that $x_k = f^{(k)}(x) \in \Omega$ for sufficiently large k.

Suppose, on the contrary, that $x_k \in S^\infty - \Omega$ for all $k \geq 0$. Then $\emptyset(x_0),\emptyset(x_1),\ldots$ is an infinite decreasing sequence in W, contradicting the well-ordering property of W.

A termination function for the division algorithm is shown in Figure 3. We shall omit proof of termination of most of the algorithms presented in Chapter 2 since it is usually obvious. In Chapter 3, where we discuss special cases of the abstract algorithms of Chapter 2, we obtain an <u>a fortiori</u> proof of termination of each algorithm by obtaining a bound on the computing time of the algorithm.

Figure 1. Formal Version of Division Algorithm

Algorithm: (S,I,Ω,f)

$I = \{(1,A,B) : A,B \in R[x],$ ldcf B a unit$\}$ where R is a commutative ring with identity

$\Omega = \{(2.1,A,B,Q,R) : A,B,Q,R \in R[x]\}$

$S = I \cup \Omega \cup \{(s,A,B,Q,R) : s = 2 \text{ or } 3,\ A,B,Q,R \in R[x]\}$

$f(1,A,B) = (2,A,B,Q,A);$

$f(2,A,B,Q,R) = (2.1,A,B,Q,R)$ if deg R < deg B,

$(3,A,B,Q,R)$ otherwise;

$f(2.1,A,B,Q,R) = (2.1,A,B,Q,R);$

$f(3,A,B,Q,R) = (2,A,B,Q+T,R-TB)$

where $T = (\text{ldcf } R/\text{ldcf } B)x^{\deg R-\deg B}$.

Figure 2. Interpretation of Division Algorithm

$P(1,A,B) \equiv (1,A,B) \in I$

$P(2,A,B,Q,R) \equiv A,B \in R[x] \wedge$ ldcf B is a unit

$\wedge\ A = BQ+R$

$P(2.1,A,B,Q,R) \equiv P(2,A,B,Q,R) \wedge \deg R < \deg B$

$P(3,A,B,Q,R) \equiv P(2,A,B,Q,R)$

Figure 3. Termination Function for Division Algorithm

Define $\delta(A) = \begin{cases} -1 \text{ if } A = 0, \\ \deg A \quad \text{otherwise;} \end{cases}$

$W = \{\text{integers} \geq -1\} \times \{1,2,3\}$;

well-ordering $\equiv$ lexicographical ordering.

Define $\emptyset: S^{\infty} \to W$ by

$\emptyset(1,A,B) = (\delta A, 3)$,

$\emptyset(2,A,B,Q,R) = (\delta R, 2)$,

$\emptyset(2.1,A,B,Q,R) = (\delta R, 1)$,

$\emptyset(3,A,B,Q,R) = (\delta R, 1)$.

1.4 Computing time analyses

The practical value of an algorithm is greatly enhanced if we have some quantitative measure of its "efficiency". Although this could be taken to include both computing time and memory requirements, we shall concentrate in this thesis entirely on analysis of the computing time of algorithms.

Methods of such analysis have been developed during the past few years, principally by Knuth [KNU68] and (particularly for algebraic algorithms) Collins ([COL69b] and [COL71]).

These methods permit computing time bounds to be expressed as functions only of the inputs to the algorithm; i.e. they are independent of the computer on which the algorithm is implemented.

The basic method for this purpose has been the use of O-notation. Recently Collins has introduced the idea of dominance which is generally more convenient than O-notation and which we shall, therefore, employ in this thesis.

The dominance relation is defined as follows ([COL71]). Let f and g be real-valued functions defined on some set S. We write $f \trianglelefteq g$ in case there is a positive real number c such that $f(x) \leq c \cdot g(x)$ for all $x \in S$ and we say that f is dominated by g (or that g is a bound for f). If $f \trianglelefteq g$ and $g \trianglelefteq f$ we write $f \sim g$ and say that f and g are codominant. Codominance is clearly an equivalence relation. Note that this definition encompasses functions of several variables since the elements of S may be n-tuples.

The following theorem lists some properties of dominance relations which are trivial consequences of the definitions.

Theorem D. Let f,g,f_1,f_2,g_1,g_2 be non-negative valued on S. Then:

a. $0 \preceq f$.

b. If c is a positive constant then $cf \sim f$.

c. If $f_1 \preceq g_1$ and $f_2 \preceq g_2$ then $f_1+f_2 \preceq g_1+g_2$ and $f_1 \cdot f_2 \preceq g_1 \cdot g_2$.

d. If $f_1 \preceq g$ and $f_2 \preceq g$ then $f_1+f_2 \preceq g$.

e. $\max(f,g) \sim f+g$.

f. If $1 \preceq f$ and $1 \preceq g$ then $f+g \preceq f \cdot g$.

g. If $1 \preceq f$ and c is a positive constant then $f \sim f+c$

h. If $S = S_1 \times \ldots \times S_n$ and $f \preceq g$ then $f(a_1,x_2,\ldots,x_n) \preceq g(a_1,x_2,\ldots,x_n)$ for any $a_1 \in S_1$.

In order to analyze the computing time of a typical algebraic algorithm it is necessary to have bounds for the computing times of the basic arithmetic and algebraic operations which the algorithm employs. For the algorithms which we shall analyze it will suffice to have bounds for operations on integers, polynomials with integer coefficients and polynomials with coefficients from a finite field GF(p). We shall now state such bounds as "axioms" on which later theory shall be based, for integer and integer polynomial operations. (Bounds for operations on polynomials over GF(p) will be cited in the references when necessary, in Chapter 3).

We shall assume that integers are represented in radix notation with some arbitrary integer base $\beta > 1$. By the length of an integer n we then mean the number of β-digits in its representation. We shall use L(n) consistently to denote the length of n, with respect to some implicit base β. We have

$$L(0) = 1$$

$$L(n) = \lfloor \log_\beta |n| \rfloor + 1, \quad n \neq 0.$$

and thus the length function has the following logarithmic properties for non-zero integers m and n:

$$L(mn) \sim L(m)+L(n)$$

$$L(m^n) \sim nL(m) \text{ (for } |m| > 1 \text{ and } n > 0). \qquad (1)$$

In fact we have

$$L(n) \sim \ln|n|$$

provided we take $S = \{n \in Z: |n| > 1\}$, and thus we could just use the natural logarithm function by making proper allowances for trivial cases. Since, however, we have the logarithmic dominance properties (1) for L and we shall be dealing only in terms of dominance relations anyway, it seems most convenient and natural to use L rather than ln.

The above assumptions are true of the representation of integers in the SAC-1 Revised Integer Arithmetic System, where an integer is represented as a list of its β-digits. (β is generally chosen to be near the largest machine integer for the particular machine on which the system is implemented.)

Axiom C_1. Let m and n be nonzero integers.

a. The time to compute m+n or m-n is $\sim\max(L(m),L(n))\sim L(m)+L(n)$.

b. The time to compute $m \cdot n$ is $\sim L(m)L(n)$.

c. Let $|m| \geq |n| > 0$. The time to compute a quotient q and remainder r such that $m = nq+r$ and $0 \leq r \leq |n|$ is $\sim L(n)L(q) \sim L(n)(L(m)-L(n)+1) \leq L(n)L(m)$.

Such statements should be taken as abbreviations for more precise statements; e.g. b abbreviates the assumption that some computer C and some algorithm A are being used, such that if t(m,n) is the amount of time required to execute A on C with inputs m and n, then the function t satisfies $t(m,n) \sim L(m)L(n)$.

Axiom C_1 can in fact easily be verified for the "classical algorithms" for integer arithmetic, where these algorithms are implemented on any computer on which basic operations such as arithmetic on machine integers, replacement, transfer of control, etc., require a bounded amount of time. In particular, the SAC-1 algorithms, executed on any known computer for which a SAC-1 implementation exists, satisfy the axiom.

For the expression of bounds on the time for arithmetic operations on polynomials it is convenient to define two norms for polynomials, as follows:

a. Let c be a complex number. Define

$$|c|_1 = |c|_\infty = |c| \quad \text{(modulus of c)}.$$

b. Let $A(x) = \sum_{i=0}^{m} a_i x^i$ be a polynomial with coefficients a_i which are complex numbers or polynomials in other variables with complex numerical coefficients. Define

$$|A|_1 = \sum_{i=0}^{m} |A_i|_1 ,$$

$$|A|_\infty = \max_{0 \le i \le m} |a_i|_\infty .$$

Thus, if A is a polynomial in several variables, $|A|_1$ is the sum of the moduli of the numerical coefficients of A and $|A|_\infty$ the maximum of these values. Both $|\ |_1$ and $|\ |_\infty$ are norms in the usual sense, i.e. they have the properties

a. $|A| > 0$ unless $A = 0$;

b. $|cA| = |c|\ |A|$;

c. $|A+B| \le |A|+|B|$.

Also, $|AB|_1 \le |A|_1 |B|_1$ but this does not hold for $|\ |_\infty$.

Axiom C_2. Let A(x) and B(x) be non-zero univariate polynomials with integer coefficients. Let m = deg A, n = deg B, $a = |A|_1$, $b = |B|_1$.

a. The time to compute A+B or A-B is

$$\le \max(m+1,n+1)\ \max(L(a),L(b)).$$

b. The time to compute A·B is

$$\le (m+1)(n+1)\cdot L(a)L(b).$$

c. Assuming B divides A and A/B has integer coefficients, the time to compute A/B is

$$\le (n+1)(m-n+1)L(b)L(|A/B|_1).$$

This axiom could be proved as a theorem, assuming the use of the classical algorithms, reasonable representations of polynomials, and Axiom C_1. Such proofs are outlined in [COL69b], and the axiom is valid for the SAC-1 Polynomial System.

Since

$$|A|_\infty \leq |A|_1 \leq (n+1)|A|_\infty,$$

where n = deg A, we have, by assuming that L(n+1) is bounded,

$$L(|A|_\infty) \sim L(|A|_1)$$

Of course, the assumption that L(n+1) is bounded implies n is bounded, but we would not want to use the latter fact; e.g. we would write

$$n^2 L(n) \sim n^2 \tag{1}$$

but not

$$n^2 L(n) \sim 1 \tag{2}$$

since the bounding constants implied in (1) would probably be small, but that in (2) might be very large. We could thus prove

Theorem C_3. Let A(x) and B(x) be univariate polynomials with integer coefficients, m = deg A, n = deg B, $a = |A|_\infty$, $b = |B|_\infty$, and assume that $L(m) \sim 1$, $L(n) \sim 1$ (i.e. that L(m) and L(n) are bounded). Then statements a,b,c, of Axiom C_2 are again true.

The reader may have noticed that in Theorem C_2 we stated the bound on the time to compute A/B as a function of one of the inputs, B, and the output C = A/B. In Section 3.4 we shall show

that $|C|_1 \le (p+1)^{2m-n+1} |A|_1$ where $p = \lfloor (2m-n)/2 \rfloor$. Hence the time to compute A/B is

$$\le (n+1)(m-n+1)L(b)L((p+1)^{2m-n+1}a)$$

$$\le (n+1)(m-n+1)L(b)[m+L(a)],$$

assuming that $L(p) \sim 1$. This gives the computing time in terms of the inputs only, but this bound is not very useful since we shall usually have $|C|_1 \le |A|_1$, whereas the bound on $|C|_1$ used here can be much larger than $|A|_1$. In cases such as this it seems most reasonable to give the computing time in terms of the output as well as the input.

We shall also have need of the computing time for trial division of polynomials A and B over the integers, by which we mean the determination of whether B divides A exactly, and, if so, determining the quotient. As an example of the analysis of a polynomial arithmetic algorithm we shall now describe and analyze a trial division algorithm.

The algorithm itself is based on Algorithm 1.3D:

Algorithm T. (Trial division) The inputs are polynomials A and B over the integers, with $B \neq 0$. If there exists a polynomial Q over the integers such that A = BQ then the output is the value d = "true" and Q (Q is uniquely determined); otherwise the value d = "false" is output.

(1) Set $R \leftarrow A$, $Q \leftarrow 0$, $i \leftarrow 0$, $d \leftarrow$ "false".

(2) (Now A = BQ+R) If R = 0, set $d \leftarrow$ "true" and exit.

(3) If deg R < deg B, exit.

(4) If ldcf B does not divide ldcf R exactly, exit.

(5) Set $n \leftarrow$ deg R-deg B, $T \leftarrow$ (ldcf R/ldcf B)x^n,

$Q \leftarrow Q+T$, $R \leftarrow R-TB$, $i \leftarrow i+1$ and go to (2).

The variable i need not actually be computed; it is included in order to simplify the assertions needed for the computing time analysis. These assertions are inductive assertions like those used for proving validity, but instead of including them in the statement of the algorithm, we shall generally state them separately afterwards.

Let $a = |A|_1$, $b = |B|_1$. The key assertion for the computing time analysis is that, whenever step (2) is executed,

$$|R|_1 \leq a(b+1)^i. \tag{3}$$

This verification of this assertion is easy and is left to the reader.

Theorem T. Let $a = |A|_1$, $b = |B|_1$, m = deg A, n = deg B. Then the computing time for Algorithm T is

$$\preceq (n+1)(m-n+1)L(a)L(b)+(n+1)(m-n+1)^2L(b)^2.$$

Proof: The time for dividing ldcf R by ldcf B is $L(\text{ldcf R})L(\text{ldcf B}) \preceq L(|R|_1)L(b)$. The addition of T to Q is trivial since no coefficient addition is required. Since T has only one term and $|T|_1 \leq |R|_1$, the time to compute T*B is $\preceq (n+1)L(|R|_1)L(b)$. Since $|TB|_1 \leq |T|_1|B|_1 \leq |R|_1|B|_1$ the time to subtract TB from R is $\preceq (n+1)[L(|R|_1)+L(b)]$. Hence $(n+1)L(|R|_1)L(b)$ dominates the time for all of these operations, and, using (3), this time is

$$\leq (n+1)[L(a)+iL(b+1)]L(b)$$

$$\leq (n+1)L(a)L(b)+(n+1)iL(b)^2,$$

since $L(b+1) \sim L(b)$. The maximum value which i can have for these operations is m-n, and the maximum number of executions of steps (4) and (5) is (m-n+1). Hence the total time has the bound stated in the theorem, and of course this bounds the time for steps (1), (2) and (3).

The SAC-1 Algorithm PQ is essentially Algorithm T (generalized for multivariate polynomials), and Theorem T is true of its computing time.

1.5 Multisets and lists

In much of the discussion in succeeding sections it will be very convenient to use the closely related concepts of "multisets" and "lists". Both concepts are very simple and fundamental, but have been introduced as mathematical concepts only recently, so we present here a brief discussion of their properties and associated notation. Knuth introduced the term multiset in [KNU69], Exercise 4.6.3-19. A multiset is like a set, but may contain identical elements repeated a finite number of times. If A and B are multisets we define new multisets $A \uplus B$, $A \cup B$, $A \cap B$, $A - B$ as follows: an element x occurring exactly a times in A and b times in B occurs exactly

$a+b$ times in $A \uplus B$,

$\max(a,b)$ times in $A \cup B$,

$\min(a,b)$ times in $A \cap B$,

$\max(a-b,0)$ times in $A-B$.

A "set" is a multiset which contains no element more than once; if A and B are sets, so are $A \cup B$, $A \cap B$ and $A-B$, and the definitions given here agree with the usual definitions of union, intersection and difference.

We write $A \subset B$ and say A is a multisubset of B iff $A \cap B = A$.

To any multiset A there corresponds a unique set, denoted by set (A), defined by: x occurs once in set (A) iff x occurs at least once in A.

The concept of a multiset could be formalized (e.g. as a mapping from a set into the set of non-negative integers), but the intuitive definitions given above will suffice for our purposes, since we shall only be dealing with finite multisets.

If A is a finite multiset with elements from a set on which a commutive addition operation is defined, then by either of

$$\Sigma A \quad \text{or} \quad \sum_{a \in A} a$$

we shall mean a sum in which an element a is included as term exactly as often as it occurs in A. For example, if A = {1,2,2,3,5,5,5} then

$$\sum_{a \in A} a = \Sigma A = 1+2+2+3+5+5+5.$$

We define $\Sigma\emptyset = 0$ where $\emptyset$ denotes the empty multiset. Similarly, assuming commutative multiplication, we may define $\prod_{a \in A} a$ and ΠA, with the convention $\Pi\emptyset = 1$.

If A is a finite multiset with elements from a set on which a function f is defined, then we define f(A) to be the multiset obtained by applying f to each occurrence of an element in A. More precisely,

$$f(A) = \biguplus_{a \in A} \{f(a)\}.$$

If A is the multiset in the above example and $f(a) = a^2$, then $f(A) = \{1,4,4,9,25,25,25\}$. As another example, let $A = \{x^2+1, x-1, x^3-3x+7, x^2+1\}$ and f = deg, the degree function; then

$$\deg(A) = \{2,1,3,2\},$$

$$\deg(\Pi A) = \Sigma \deg(A) = \Sigma\{2,1,3,2\} = 2+1+3+2 = 8.$$

We define a <u>list</u> simply to be a finite sequence. Lists are thus closely related to multisets; we may think of a list $A = (a_1,\ldots,a_n)$ as representing the multiset $\{a_1,a_2,\ldots,a_n\}$; of course, the list $(a_{\sigma(1)},\ldots,a_{\sigma(n)})$ also represents the same multiset, where σ is any permutation of $1,2,\ldots,n$. The concept of a list is very useful in the design of computer algorithms, but frequently we construct a list without really caring about the order in which elements occur in the list, and in these cases it is useful to replace the concept of list by that of a (finite) multiset.

We shall use the notations ΣA, ΠA and $f(A)$ when A is a list as well as a multiset, with the obvious meanings. The following notation applies only to lists. Let $A = (a_1,\ldots,a_n)$; then

first (A) = a_1, second (A) = a_2, etc.,

last (A) = a_n,

tail (A) = $(a_2,\ldots,a_n)$,

prefix (a,A) = $(a,a_1,\ldots,a_n)$,

inverse (A) = $(a_n,\ldots,a_1)$,

length (A) = n.

If A = ( ), the empty list, we define prefix (a,A) = (a), inverse (A) = ( ), length (A) = 0. This notation is essentially the same as that used in the SAC-1 List Processing System [COL67].

1.6 Generation of sum index sets

In some of the factoring algorithms considered in Chapters 2 and 3 a subalgorithm is required for solving the following problem: given positive integers $n_1,n_2,\ldots,n_r$ (not necessarily distinct) and an integer n, for what sets $J \subset \{1,\ldots,r\}$ is $\Sigma_{j\in J} n_j = n$? In the factoring algorithms, the n_i are the degrees of the irreducible factors of a polynomial and the problem is to produce all factors of a given degree n. In this section we present a simple, efficient solution to the general problem of determining the subsets J, which we shall refer to as sum index sets.

We first consider the solution to a simpler problem: is there any set $J \subset \{1,\ldots,r\}$ such that $\Sigma_{j\in J} n_j = n$? This problem can be attacked by constructing the sumset of $n_1,n_2,\ldots,n_r$, which we define to be the set of all sums $\Sigma_{j\in J} n_j$ such that $J \subset \{1,\ldots,r\}$; we then have merely to determine whether n is a member of the sumset of $n_1,\ldots,n_r$.

The following algorithm computes the sumset S of $n_1,\ldots,n_r$.

(1) Set $S^{(0)} \leftarrow \{0\}$. ($S^{(0)}$ is the sumset of the empty set)

(2) For $j = 1,\ldots,r$: Set $S^{(j)} \leftarrow S^{(j-1)} \cup (S^{(j-1)}+\{n_j\})$

($S^{(j)}$ is the sumset of $n_1,n_2,\ldots,n_j$).

(3) Set $S \leftarrow S^{(r)}$.

In step (2), $S^{(j-1)}+\{n_j\}$ is the set $\{a+n_j : a \in S^{(j-1)}\}$.

The assertion in step (2) is proved as follows:

$$a \in S^{(j)} \Longleftrightarrow a \in S^{(j-1)} \text{ or } a-n_j \in S^{(j-1)}$$

$$\Longleftrightarrow a \in \text{sumset of } n_1,\ldots,n_{j-1}$$

$$\text{or } a-n_j \in \text{sumset of } n_1,\ldots,n_{j-1} \text{ (by induction)}$$

$$\Longleftrightarrow a \in \text{sumset of } n_1,\ldots,n_j.$$

It will be shown in Chapter 3 that this algorithm may be implemented in a very simple and efficient manner if a binary representation of a set of nonegative integers is used.

If only the final result S is of interest then the above algorithm could be simplified by removing the superscripts from S and deleting step (3). However, for the purpose of solving the original problem of finding all sum index sets for an integer n, we shall make use of all of the sumsets $S^{(0)},\ldots,S^{(r)}$. The algorithm is most easily formulated using recursion.

Algorithm G(j,n,J) (Generation of sum index sets). Let $n_1,n_2,\ldots,n_r$ be fixed positive integers and $S^{(0)},S^{(1)},\ldots,S^{(r)}$ be sets such that $S^{(i)}$ is the sumset of $n_1,n_2,\ldots,n_i$ for $0 \leq i \leq r$. ($S^{(0)} = \{0\}$.) Given a nonnegative integer j, an integer $n \in S^{(j)}$, and a set J, this algorithm outputs all sets $J' \cup J$ such that $J' \subset \{1,\ldots,j\}$ and $\Sigma_{i\in J'} n_i = n$. (Thus, if $n \in S^{(r)}$, performing $G(r,n,\emptyset)$ causes all sets $J \subset \{1,\ldots,r\}$ such that $\Sigma_{i\in J} n_i = n$ to be output).

(1) If $n = 0$, output J and exit.

(2) If $n-n_j \in S^{(j-1)}$, perform $G(j-1,n-n_j,\{j\} \cup J)$.

(3) If $n \in S^{(j-1)}$, perform $G(j-1,n,J)$. Exit.

Note: One way to interpret the imperative "output J" is "set $\Omega \leftarrow \Omega \cup \{J\}$" where Ω is a set which is initially set to $\emptyset$ outside of the algorithm, so that the output of the algorithm is a set Ω of sum index sets.

Validity proof: If $n = 0$, the algorithm outputs J and stops; this is correct since $J = \emptyset \cup J$ and $\emptyset$ is the only set J' such that $J' \subset \{1,\ldots,j\}$ and $\Sigma_{i\in J'} n_i = 0$. If $n \neq 0$ then the assumption $n \in S^{(j)}$ implies $n > 0$ and $j > 0$.

If $n-n_j \in S^{(j-1)}$, then the algorithm is recursively performed with inputs $j-1, n-n_j, \{j\} \cup J$; by induction we may assume that this causes all sets $J' \cup \{j\} \cup J$ such that $J' \subset \{1,\ldots,j-1\}$ and $\Sigma_{i\in J'} n_i = n-n_j$ to be output. If $n-n_j \notin S^{(j-1)}$, there are no such sets; in this case step (2) correctly does nothing. Similarly, if $n \in S^{(j-1)}$, then by induction step (3) causes all sets $J' \cup J$ such that $J' \subset \{1,\ldots,j-1\}$ and $\Sigma_{i\in J'} n_i = n$ to be output; otherwise, nothing is output. The combined output of steps (2) and (3) is thus the output claimed for the algorithm.

A concrete realization of this algorithm will be considered in Section 3.3.

CHAPTER 2: ABSTRACT FACTORING ALGORITHMS

2.1 Unique factorization domains

This chapter attempts to provide a general theoretical basis for polynomial factorization in numerous domains, primarily through the presentation of abstract algorithms. In these algorithms, as explained in Section 1.3, the domain containing the inputs or outputs may be an abstract algebraic system, such as a ring, integral domain, or field. We shall assume the reader has an acquaintance with the basic definitions and theorems about such systems, as given in, for example, [VDW49], or [GOL70]. However, many of the definitions and theorems relating most closely to factorization will be reviewed in this and the two following sections.

We shall begin with some basic definitions about division. Let R be a commutative ring with identity and let $a,b \in R$. We say that b divides a and write $b|a$ if there exists c in R such that $a = bc$. If c is unique, we also denote it by a/b. If $b \neq 0$ and there exists $c \neq 0$ such that $bc = 0$, then b is called a zero divisor. If $a = bc = bc'$, then $b(c-c') = 0$; hence, if $b \neq 0$ and b is not a zero divisor then $c-c' = 0$, i.e. c is uniquely determined and we can denote it by a/b.

A unit of R is an element which has a multiplicative inverse; equivalently, u is a unit in case $u|1$.

An integral domain is a commutative ring with identity which contains no zero divisors.

Let D be an integral domain and let a be a nonzero, nonunit

element of D. A proper factorization of a is an equation a = bc where neither b nor c is a unit; b and c are called proper factors of a. We call a reducible if it has a proper factorization, irreducible otherwise. We say that a has a unique factorization into irreducible elements (also called a complete factorization) if there exist irreducible elements $p_1,p_2,\ldots,p_r$ in D such that $a = p_1p_2\ldots p_r$; and if $a = q_1q_2\ldots q_s$ is another factorization of a into irreducible elements, then we have r = s, and after a rearrangement, we have $p_1 = u_1q_1,\ldots,p_r = u_rq_r$, where $u_1,\ldots,u_r$ are units. By convention, we do not consider units to be irreducible elements.

Finally, then, we define D to be a unique factorization domain (UFD) if it is an integral domain and if every nonzero, nonunit element of D has a unique factorization into irreducible elements.

The integral domain Z of integers is a UFD (Fundamental Theorem of Arithmetic). Any field F is a UFD in which every nonzero element is a unit and there are no irreducible elements. According to a theorem of Gauss, the polynomial domain $D[x_1,\ldots,x_n]$ is a UFD whenever D is. Thus, for example, $Z[x_1,\ldots,x_n]$ and $F[x_1,\ldots,x_n]$ are UFDs. The proofs of these statements will be reviewed in the next two sections. We continue this section with the definition of more terminology that will be useful in dealing with UFDs.

In an integral domain D two elements a and b such that a = ub for some unit u are called associates. The relation "a is an associate of b" is an equivalence relation, which we will symbolize by $a \sim b$. Let us single out one element from each equivalence class

and call the resulting set of representatives an ample set D_0 for D. Thus any nonzero element of D can be uniquely expressed as $a = ua_0$ with u a unit of D and a_0 in D_0.

The units of D[x] are just those of D. It follows that an ample set for D[x] is given by the set of all polynomials with leading coefficient in D_0.

In D = Z, the units are +1 and -1, and the equivalence classes are {0} and {-n,n}, n = 1,2,....

A multiplicative set S is a subset of a ring such that 1 is in S and ab is in S wherever a and b are. We will generally want to choose an ample set for an integral domain in such a way that it is multiplicative. The nonnegative integers are a multiplicative ample set for Z; 0 and the monic polynomials form a multiplicative ample set for F[x], F a field. A unique factorization domain always has a multiplicative ample set, as will be shown below.

If S is any subset of a ring R, we shall call any polynomial in R[x] with leading coefficient in S an S polynomial. If S is multiplicative and contains no zero divisors, then the same is true of the set of all S polynomials in R[x]. In particular, if D_0 is a multiplicative ample set for an integral domain D, then the set of all D_0 polynomials is a multiplicative ample set for D[x].

In an integral domain D an element g is called a greatest common divisor (gcd) of two elements a and b if it divides both a and b and if any other common divisor of a and b also divides g. Similarly, m is called a least common multiple (lcm) of a and b if it is a

multiple of a and b and any other common multiple of a and b is also a multiple of m. Any two greatest common divisors of a and b are associates, and similarly for lcm's. If D_0 is an ample set for D, then we can define the gcd of a and b to be the one contained in D_0. We write this gcd as gcd(a,b) and we similarly define lcm(a,b).

Note that g is a gcd of a and 0 iff g is an associate of a, and 0 is the unique gcd of 0 and 0.

Two elements a and b are relatively prime if any gcd of a and b is a unit. If 1 is in D_0 then this means gcd(a,b) = 1.

Now let D be a unique factorization domain. An irreducible element p of D generates a prime ideal (p) (i.e. $ab \in (p)$ implies $a \in (p)$ or $b \in (p)$). Hence in a UFD an irreducible element will also be called a prime element, or simply a prime.

Let P be the set of all primes in D. Choose one prime out of each equivalence class of associates belonging to such a prime and call the set P_0 of such representatives an ample set of primes in D (here we are using the Axiom of Choice). If D_0 is an ample set for D, then $P_0 = P \cap D_0$ is an ample set of primes in D.

Conversely, having selected P_0 we may obtain an ample set D_0 for D. For any nonzero element a in D there is a unique expression

$$a = u\Pi A, \tag{1}$$

where u is a unit of D and A is a multiset (see Section 1.5) whose elements are chosen from P_0; in symbols, $set(A) \subset P_0$. Define

$$D_0 = \{0\} \cup \{\Pi A : A \text{ finite}, set(A) \subset P_0\}. \tag{2}$$

Then D_0 is an ample set for D; this follows immediately from the

existence and uniqueness of the expression (1). Furthermore, D_o is multiplicative. (Proof: $1 = \Pi\emptyset$ is in S since $\emptyset \subset P_o$, and ΠA, ΠB in D_o implies $\Pi A \Pi B = \Pi(A \uplus B)$ is in D_o since set $(A \uplus B) \subset P_o$.) Conversely, if D_o is a multiplicative ample set for D, and P_o is the set of primes in D_o, then D_o is given by (2); i.e. $a \in D_o$ and $a \sim \Pi A$ imply $a = \Pi A$.

The multiset notation (1) for complete factorizations is very convenient and we shall usually employ it. Whenever $u = 1$ we shall also refer to the multiset A as the complete factorization of a.

Some of the definitions previously given for integral domains can be recast in a UFD, using the representation (1). Let a and b be nonzero, with $a \sim \Pi A$ and $b \sim \Pi B$. Then

$$a|b \quad \text{iff} \quad A \subset B,$$

$$\gcd(a,b) = \Pi(A \cap B),$$

$$\operatorname{lcm}(a,b) = \Pi(A \cup B).$$

Note that $ab \sim \Pi(A \uplus B) = \Pi((A \cap B) \uplus (A \cup B)) = \Pi(A \cap B)\Pi(A \cup B) = \gcd(a,b) \operatorname{lcm}(a,b)$. Note also that a and b are relative prime iff $\gcd(a,b) = 1$ iff $A \cap B = \emptyset$.

The definitions of gcd's and lcm's extend in a natural way to an arbitrary number of elements $a_i \sim \Pi A_i$, for example

$$\gcd(a_1,a_2,\ldots,a_n) = \Pi(A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_n})$$

where $a_{i_1},\ldots,a_{i_n}$ are the nonzero elements among $a_1,\ldots,a_n$; $\gcd(0,\ldots,0) = 0$.

2.2 Euclidean UFDs

An important concept in the proof of unique factorization in various integral domains is that of a Euclidean ring. This is an integral domain R for which a mapping d from R-{0} to the nonnegative integers is defined, such that

(i) $d(ab) \geq d(a)$ if $ab \neq 0$,

(ii) (division algorithm) for any elements a and b with $b \neq 0$ there are elements q and r such that $a = bq+r$ with $r = 0$ or $d(r) < d(b)$.

The mapping d is called a degree function for R. The most important examples of Euclidean rings are the integers Z and the polynomial domains F[x] where F is a field. In Z the absolute value function serves for d; in F[x], d(a) is the degree of a. For F[x] a division algorithm is provided by Algorithm D of Section 1.3.

Theorem E: A Euclidean ring is a unique factorization domain.

We shall review the proof of this fundamental result since it contains a number of important concepts.

Lemma 1. Let R be a Euclidean ring with degree function d. If b is a proper factor of c then $d(b) < d(c)$.

Proof: Let $c = ab$ and $b = cq+r$ with $r = 0$ or $d(r) < d(c)$. If $r = 0$ then $(ab)|b$, which implies that a is a unit, a contradiction. Thus $d(c) > d(r) = d(b(1-aq)) \geq d(b)$.

From this result we obtain the following abstract algorithm (see Section 1.3 for a discussion of the concept of "algorithm" being assumed here, particularly the remarks regarding effectiveness):

Algorithm F (Factorization algorithm for a Euclidean ring R). Given a nonzero, nonunit element a of R, this algorithm finds a factorization of a into irreducible elements: i.e. it determines a multiset A whose elements are irreducible elements of R such that $a = \Pi A$.

(1) If a is irreducible, set $A \leftarrow \{a\}$ and exit.

(2) Find nonunits b and c in R such that $a = bc$.

(3) Apply this algorithm recursively to obtain factorizations $b = \Pi B$ and $c = \Pi C$.

(4) Set $A \leftarrow B \uplus C$ and exit.

Validity proof: By induction on d(a). Assume the algorithm is valid for all nonzero, nonunit elements b with $d(b) < n$, and let $d(a) = n$. If a is irreducible, the algorithm is obviously valid. So let $a = bc$ where neither b nor c is a unit. By Lemma 1, $d(b) < d(a)$ and $d(c) < d(a)$. By the induction hypothesis, step (3) correctly yields B and C. Hence $a = bc = \Pi B \Pi C = \Pi(B \uplus C)$, and the algorithm terminates correctly.

We have thus proved the existence of factorizations into irreducible elements (although not constructively).

A key result for the proof of uniqueness of factorization is provided by another algorithm:

Algorithm E (Extended Euclidean Algorithm). Given $a \neq 0$ and b in a Euclidean Ring R, this algorithm finds a gcd g of a and b and elements s and t in R such that

$$as+bt = g.$$

(1) Set $t \leftarrow 0$, $g \leftarrow a$, $T \leftarrow 1$, $G \leftarrow b$.

(2) (Now any common divisor of g and G is a common divisor of a and b; also a divides both g-bt and G-bT.) If $G = 0$, set $s \leftarrow (g-bt)/a$ and exit.

(3) Find q and r such that

$$g = qG+r,\ r = 0 \text{ or } d(r) < d(G).$$

(Then any common divisor of G and r is a common divisor of g and G, hence of a and b.)

(4) Set $t_1 \leftarrow t-qT$, $t \leftarrow T$, $T \leftarrow t_1$, $g \leftarrow G$, $G \leftarrow r$, and go to (2).

Validity proof: Let $t^* = T$, $g^* = G$, $T^* = t-qT$, $G^* = r$. Then $g^*-bt^* = G-bT$, hence $a \mid G-bT$ implies $a \mid g^*-bt^*$. Also $G^*-bT^* = g-qG-b(t-qT) = g-bt-q(G-bT)$. Assuming a divides both terms of the last expression, a divides G^*-bT^*.

To show that the output assertions are satisfied upon exit, note that $G = 0$ implies that g is a common divisor of g and G, hence of a and b. From the definition of s we have $as+bt = g$, and this equation implies that any common divisor of a and b also divides g. Hence g is a greatest common divisor of a and b.

Using this algorithm we easily obtain:

Lemma 2. Let R be a Euclidean ring. Let $p,a,b \in R$, with p irreducible. If $p \mid ab$ then $p \mid a$ or $p \mid b$.

Proof: Suppose $p \nmid a$. Then any gcd of a and p must be a unit of R. Using Algorithm E, we obtain a gcd g and s,t in R such that $as+pt = g$. Since g is a unit we may multiply by bg^{-1} to obtain $absg^{-1}+pbtg^{-1} = b$.

Since p divides both terms on the left, we conclude that $p|b$.

Theorem E1: In a Euclidean ring factorization into irreducible elements is unique.

Proof: Suppose a nonzero, nonunit element a has two factorizations

$$a = p_1 \cdots p_r = q_1 \cdots q_s,$$

into irreducible elements, with $r \leq s$. Since p_1 divides the product on the right, it also divides one of the factors, by the above lemma. Renumber, if necessary, so that this factor is q_1. Thus $q_1 = u_1 p_1$ for some unit u_1, and p_1 can be cancelled from both factorizations:

$$p_2 \cdots p_r = u_1 q_2 \cdots q_s.$$

By induction on r we may assume that $s = r$ and $p_i = u_i q_i$ for some units $u_2, \ldots, u_r$, thus completing the proof.

We have therefore established that a Euclidean ring is a UFD. Some parts of the proof could have been done in a different way. It is easily shown that a Euclidean ring is a special case of a principal ideal ring, that is, an integral domain in which all ideals are principal. A principal ideal ring may be proved to be a UFD (only uniqueness is proved in [VDW49]; see [LAN65] for an existence proof). We chose to restrict the above proof to Euclidean rings in order to be able to introduce the Extended Euclidean Algorithm. This algorithm and the simpler Euclidean algorithm, which just calculates a gcd, are very important in actual computation, as will be noted in succeeding sections.

2.3 Unique factorization in polynomial domains

Let D be a UFD and A(x) be a nonzero polynomial in D[x]. A has a factorization

$$A = cA_1 \tag{1}$$

where c is a greatest common divisor of the coefficients of A. We call c the content of A (cont(A)). The content of A_1 is a unit of D; such a polynomial is called primitive, and A_1 is called the primitive part of A (pp(A)). Note that cont(A) and pp(A) are uniquely determined up to multiplication by units.

In Section 2.2 it was shown that F[x], F a field, is a UFD. From this a more general result can be proved:

Theorem G (Gauss). If D is a UFD, so is D[x]. The prime elements are those in D and the primitive polynomials in D[x] which are irreducible in K[x], where K is the quotient field of D.

The key facts in the proof are:

Lemma 1: The product of primitive polynomials is primitive.

Lemma 2: Every nonzero polynomial A in K[x] may be expressed as

$$A = uA_1$$

where u is a unit of K (a nonzero fraction b/c with b,c in D) and A_1 is a primitive polynomial in D[x]. u and A_1 are uniquely determined up to multiplication by units of D.

The proofs of these lemmas are given, for example, in [VDW49], Section 23. Assuming the lemmas, we can complete the proof of Theorem G. We first note that the lemmas may be rephrased using the terminology of Section 2.1. Let D_0 be a multiplicative ample

set for D and F_0 be the set of all primitive D_0 polynomials. F_0 is an ample set of primitive polynomials for D[x]; i.e. every primitive polynomial in D[x] has a unique associate in F_0. By Lemma 1, F_0 is a multiplicative set. By Lemma 2, F_0 is an ample set for K[x]. Thus: F_0 is a multiplicative ample set for K[x].

Let P be the set of prime elements in K[x] and $P_0 = P \cap F_0$. Thus P_0 is an ample set of primes for K[x]. By the unique factorization property of K[x], we obtain for any nonzero A in K[x] a unique representation

$$A = u\Pi\mathcal{a} \qquad (2)$$

where u is a unit of K and $\mathcal{a}$ is a multiset such that $set(\mathcal{a}) \subset P_0$. Furthermore, from the fact that F_0 is multiplicative, $\Pi\mathcal{a}$ is in F_0, and thus: if A is in F_0, then $A = \Pi\mathcal{a}$; that is, if A is a primitive D_0 polynomial then A is the product of primitive D_0 polynomials in D[x] which are irreducible in K[x].

We therefore have a factorization of A in D[x]. The factors are irreducible in D[x], for otherwise they would be reducible in K[x]. The factors are uniquely determined, since any other factorization into irreducible elements would also be different in K[x].

Now let A be an arbitrary nonzero polynomial in D[x]. It has a unique factorization

$$A = cont(A)pp(A),\ pp(A) \in F_0.$$

Since D is a UFD, cont(A) has a unique factorization into irreducible elements in D; this remains the same in D[x] since a polynomial of degree zero can only have factors of degree zero. By the above

remarks, pp(A) has a unique factorization into irreducible elements in D[x]. The combination of these factorizations yields a unique factorization of A in D[x] and completes the proof of Theorem G.

The obvious corollary of Theorem G is:

Corollary: If D is a UFD, then so is $D[x_1,\ldots,x_n]$.

Proof: Induction.

It will be convenient to have some further terminology for dealing with complete factorization in polynomial domains. Let D be a UFD with a multiplicative ample set D_0. Let A be a nonzero D_0 polynomial over D. Thus there is a unique expression

$$A = c\Pi F$$

where c is the content of A and F is a multiset of prime, primitive D_0 polynomials. We will call this expression the complete D_0 factorization of A. If c = 1 then we will also refer to F as the complete D_0 factorization of A.

2.4 Squarefree factorization

In the previous section we have seen that the problem of factoring an arbitrary polynomial A over a unique factorization domain D can be reduced to the separate problems of factoring in D and factoring primitive polynomials over D. This reduction is achieved by means of gcd calculations in D. In this section we show that we can further reduce the problem of factoring a primitive polynomial over D to that of factoring polynomials which are "squarefree" in a sense to be defined below, by means of gcd calculations in D[x]. Given primitive polynomials $A,B \in D[x]$, we may compute a gcd C of A and B in K[x], where K is the quotient field of D, by means of the Euclidean Algorithm. Then any associate of C in K[x] which lies in D[x] and is primitive is a gcd of A and B in D[x]. Better alternative algorithms exist in special cases, e.g. when $D = Z[x_1,\ldots,x_n]$, making it much faster to compute the gcd of two polynomials than to factor a single polynomial.

We define a polynomial A to be squarefree if it has no factor of positive degree of the form B^2. Let D be a UFD and A be a primitive, squarefree polynomial over D, deg A > 0. Then A has a complete factorization $A = P_1P_2\ldots P_n$ where the P_i are distinct prime polynomials of positive degree.

Suppose $A = B^2C$. Then the derivative $A' = B^2C' + 2BB'C$ is a multiple of the squared factor B, hence $B \mid \gcd(A,A')$. Hence deg(gcd(A,A')) = 0 implies A is squarefree.

In case D is of characteristic zero, the converse also holds, as a corollary of the following theorem. (If there is a smallest

positive integer n such that nx = 0 for all x in a ring D then n is the characteristic of D; otherwise the characteristic is zero. If D is an integral domain, then the characteristic is a prime if it is not zero.)

Theorem S. Let D be a UFD of characteristic zero and A be a nonconstant primitive polynomial over D. Let $A = P_1^{e_1}\ldots P_n^{e_n}$ be a complete factorization of A. Then $\gcd(A,A') \sim P_1^{e_1-1}\ldots P_n^{e_n-1}$.

Proof: Let $B = \gcd(A,A')$, $P = P_1$, $e = e_1$ and $Q = A/P^e$. Then $A = P^eQ$ and $A' = P^eQ' + eP^{e-1}P'Q$, hence $P^{e-1}|B$. If $P^e|B$ then $P^e|A'$, hence $P^e|eP^{e-1}P'Q$, hence $P|eP'Q$. But P and Q are relatively prime, so $P|eP'$. Since the characteristic of D is zero, $eP' \neq 0$, hence $\deg(eP') \geq \deg P$, a contradiction. So the order of P_1 in B is e_1-1, and the theorem follows by symmetry.

Corollary 1. If A is squarefree then $\gcd(A,A') \sim 1$.

Corollary 2. $C = A/\gcd(A,A')$ is a squarefree factor of A; in fact $C \sim P_1\ldots P_n$.

Thus to factor A we could compute C and factor it to obtain the P_i, then divide A by P_i as often as possible, to determine the e_i. However, we can do better than this if A is not already squarefree, for we will show that we can then partially factor C and determine the e_i by means of further gcd calculations.

Let $Q_i = \Pi_{j\in E_i} P_j$, where $E_i = \{j:e_j = i\}$. ($Q_i = 1$ when $E_i = \emptyset$.) Then for $t = \max\{e_1,\ldots,e_n\}$ we have

$$A = Q_1Q_2^2\ldots Q_t^t,\ Q_i \text{ squarefree},\ \deg Q_t > 0,\ \gcd(Q_i,Q_j) \sim 1 \text{ for } i\neq j. \quad (1)$$

We call this representation of A a squarefree factorization of A,

since each Q_i is either unity or a squarefree polynomial of positive degree. The Q_i are uniquely determined by the conditions in (1), except for unit multiples.

By Theorem S, if $B = \gcd(A,A')$ and $C = A/B$, then $B \sim Q_2Q_3^2\ldots Q_t^{t-1}$ and $C \sim Q_1Q_2\ldots Q_t$. If $D = \gcd(B,C)$ then $D \sim Q_2Q_3\ldots Q_t$, hence $Q_1 \sim C/D$. The following algorithm shows how we can continue, computing $Q_2,\ldots,Q_t$:

Algorithm S (Squarefree factorization). Let D be a UFD of characteristic zero, with multiplicative ample set D_0. Given a primitive D_0 polynomial A of positive degree, let $A = Q_1Q_2^2\ldots Q_t^t$ be the squarefree factorization of A into D_0 polynomials. This algorithm computes t and $A_1 = Q_1,\ldots,A_t = Q_t$.

(1) Set $B \leftarrow \gcd(A,A')$, $C \leftarrow A/B$, $j \leftarrow 1$.

(2) (At this point $B = Q_{j+1}Q_{j+2}^2 \ldots Q_t^{t-j}$ and $C = Q_jQ_{j+1} \ldots Q_t$.) If $B = 1$ then set $t \leftarrow j$, $A_t \leftarrow C$, and exit.

(3) Set $D \leftarrow \gcd(B,C)$, $A_j \leftarrow C/D$. (Then $D = Q_{j+1}Q_{j+2} \ldots Q_t$ and $A_j = Q_j$.)

(4) Set $B \leftarrow B/D$, $C \leftarrow D$, $j \leftarrow j+1$, and go to (2). (This step preserves the assertion at step (2).)

In steps (1) and (3) we assume that the gcd's computed are D_0 polynomials. The reader may easily verify the assertions in the algorithm.

Algorithm S is based on an algorithm presented by Horowitz in [HOR69], pp. 58-60, 69-70, which in turn was based on an algorithm due to Tobey. Horowitz' version is equivalent to Algorithm S with steps (3) and (4) replaced by:

(3') Set $E \leftarrow \gcd(B,B')$, $D \leftarrow B/E$, $A_j \leftarrow C/D$. (Then $E = Q_{j+2}Q_{j+3}^2 \cdots Q_t^{t-j-1}$, $D = Q_{j+1}Q_{j+2} \cdots Q_t$, $A_j = Q_j$.)

(4') Set $B \leftarrow E$, $C \leftarrow D$, $j \leftarrow j+1$, and go to (2).

Note that D and E = B/D are computed in both versions, but in different ways. Algorithm S appears to require slightly less computation than Horowitz' version, but its main virtue seems to be that it can be easily adapted for squarefree factorization over finite fields (which are of prime rather than zero characteristic), whereas it appears to be rather difficult to adapt Horowitz' version for this problem.

We shall discuss the general case of domains of prime characteristic initially and later consider the finite field case.

The proof of Theorem S depended on the fact that the derivative of a nonconstant polynomial cannot vanish identically when the characteristic of the coefficient domain is zero. Now let D have prime characteristic p. Suppose $A(x) = \sum_{i=0}^{n} a_i x^i$; then $A'(x) = \sum_{i=0}^{n} i a_i x^{i-1} = 0$ iff $i a_i = 0$ for each i iff $p \mid i$ or $a_i = 0$ for each i. Hence, A' = 0 iff A is a polynomial in x^p.

A slight modification of the proof of Theorem S yields:

Theorem T. Let D be a UFD of arbitrary characteristic and A be a nonconstant primitive polynomial over D. Let $A = P_1^{e_1} \cdots P_n^{e_n}$ be a complete factorization and let

$$\delta_i = \begin{cases} 0, & \text{if } e_i P_i' = 0, \\ 1, & \text{otherwise.} \end{cases}$$

Then

$$\gcd(A,A') \sim P_1^{e_1-\delta_1} \ldots P_n^{e_n-\delta_n} .$$

The condition $e_iP_i' = 0$ occurs iff $P_i' = 0$ or the characteristic of D divides e_i.

If, given $A = P_1^{e_1}\ldots P_n^{e_n} \in D[x]$ where the characteristic of D is p, we let

$$R_i = \begin{cases} \Pi\{P_j : e_j = i \text{ and } P_j' \neq 0\} \text{ if } p \nmid i, \\ 1, \text{ otherwise;} \end{cases}$$

and

$$S = \Pi\{P_j^{e_j} : p \mid e_j \text{ or } P_j' = 0\},$$

then we have

$$A = R_1R_2^2\ldots R_t^t\, S,$$

where $0 \leq t \leq \max\{e_1,\ldots,e_n\}$.

According to Theorem T, $\gcd(R_i,R_i') \sim 1$. Since each factor $P_j^{e_j}$ of S satisfies $(P_j^{e_j})' = 0$, we have $S' = 0$.

It also follows from Theorem T that

$$\gcd(A,A') \sim R_2R_3^2 \ldots R_t^{t-1}S;$$

hence

$$A/\gcd(A,A') \sim R_1R_2\ldots R_t.$$

It is now easy to verify that Algorithm S applied to A will yield $A_1 = R_1,\ldots,A_t = R_t$, but will not terminate unless $S \sim 1$. We must change the test "If B = 1" in step (2) to "If B' = 0" to cause the algorithm to terminate with B = S. Also, we must append "If C = 1, set $t \leftarrow 0$ and exit." to step (1), to take care of the case A' = 0. We shall refer to the algorithm thus obtained as Algorithm T; its input is a primitive polynomial A over a domain of prime characteristic

p, and its outputs are $t, A_1, \ldots, A_t$, and B such that $A = A_1 \ldots A_t^t B$, $\gcd(A_i, A_i') \sim 1$, $B' = 0$, and $A_1, A_2, \ldots, A_t$, B are pairwise relatively prime.

If we have $B \sim 1$, then $A = A_1 \ldots A_t^t$ is a squarefree factorization of A. But if deg B > 0 then further calculation is required to obtain the complete squarefree factorization. Although it is not apparent how this may be done with polynomials over an arbitrary domain D of characteristic p > 0 (without resorting to a complete factorization algorithm), we shall derive algorithms for the important special cases when D is a finite field or a domain of polynomials in one or more variables over a finite field.

Before considering these special cases, we recall some further properties of domains of prime characteristic:

Theorem. Let D be an integral domain of prime characteristic p. Let $a,b \in D$ and $A,B \in D[x]$. Let $q = p^n$, where n is a positive integer. Then:

a. $(ab)^q = a^q b^q$ and $(a+b)^q = a^q + b^q$;

b. $(AB)^q = A^q B^q$ and $(A+B)^q = A^q + B^q$.

Proof: See [VDW49], pp. 92-93.

We now let D be a finite field of characteristic p. A finite field is also called a Galois field and is denoted by GF(q) where q is the number of elements in the field. Practical computation in GF(q) is discussed in [BER68] and [COL69a]; here we assume the ability to do arithmetic and gcd calculations in GF(q)[x].

The most basic facts about Galois fields are:

1. The number of elements q is a power of the characteristic p.
2. For a given $q = p^n$ $(n > 0)$ there exists one, and except for isomorphism only one, Galois field with precisely q elements.
3. $a^q = a$ for all $a \in GF(p)$.
4. Every $a \in GF(q)$ has a unique p^{th} root in GF(q). (In fact $a^{1/p} = a^{p^{n-1}}$, where $q = p^n$; note that if $n = 1$, $a^{1/p} = a$.)

For proofs, see [VDW49], pp. 115-117.

The theorem needed for application to squarefree factorization over GF(q) is:

Theorem P. Let $A \in GF(p^n)[x]$. Then $A' = 0$ iff A is the p^{th} power of some polynomial $B \in GF(p^n)[x]$.

Proof: If $A = B^p$, then $A' = pB^{p-1}B' = 0$. Conversely, if $A' = 0$, then A can be written in the form $A(x) = \sum_{i=0}^{k} a_{pi}x^{pi}$. Let $B(x) = \sum_{i=0}^{k} a_{pi}^{1/p}x^i$; then $B \in GF(p^n)[x]$ and $B^p = A$.

From this theorem we can obtain an algorithm which applies Algorithm T repeatedly until the complete squarefree factorization is found. In stating this algorithm we shall include a modified version of Algorithm T which uses a different test for termination.

Algorithm U (Squarefree factorization over a finite field). Let A be a nonconstant monic polynomial over $GF(p^n)$, and let $A = \prod_{i=1}^{t} Q_i^i$ be its squarefree factorization into monic polynomials. Given A, this algorithm computes t and $A_1 = Q_1, \ldots, A_t = Q_t$.

(1) Set $k \leftarrow 0$, $m \leftarrow 1$, $t \leftarrow 0$.

(2) Set $j \leftarrow 1$, $B \leftarrow \gcd(A,A')$, $C \leftarrow A/B$. If $C = 1$, go to (7).

(3) Set $r \leftarrow jm$. If $r > t$, set $A_{t+1} \leftarrow A_{t+2} \leftarrow \ldots \leftarrow A_{r-1} \leftarrow 1$, $t \leftarrow r$.

(4) Set $D \leftarrow \gcd(B,C)$, $A_r \leftarrow C/D$.

(5) If $D \neq 1$, set $B \leftarrow B/D$, $C \leftarrow D$, $j \leftarrow j+1$, and go to (3).

(6) If $B = 1$, exit.

(7) Set $A \leftarrow B^{1/p}$, $k \leftarrow k+1$, $m \leftarrow mp$, and go to (2).

Throughout the execution of this algorithm we have $m = p^k$, and whenever we arrive at step (6) the value of t is the largest index i such that $p^k \nmid i$ and A has a nonconstant factor of order i.

We assume that the gcds calculated in steps (2) and (4) are monic.

We arrive at step (7) only if $B' = 0$; hence we know that B is of the form $B(x) = \sum_{i=0}^{h} b_{ip}x^{ip}$, hence $B(x)^{1/p} = \sum_{i=0}^{h} b_{ip}^{i/p}\, x^i$, and we may calculate $b_{ip}^{1/p}$ using the identity $a^{1/p} = a^{p^{n-1}}$.

We next consider the extension of Algorithm U to polynomials in several variables. The main theorem needed is:

Theorem V. Let $A \in GF(p^n)[x_1,\ldots,x_r]$. Then $A = B^p$ for some $B \in GF(p^n)[x_1,\ldots,x_r]$ iff $\partial A/\partial x_i = 0$ for $i = 1,\ldots,r$.

The proof is similar to that of Theorem P and will be omitted.

On the basis of this theorem we obtain the following algorithm which reduces the problem of factoring in $GF(p^n)[x_1,\ldots,x_r]$ to that of factoring polynomials E which are not only squarefree but satisfy the stronger condition that $\gcd(E,\partial E/\partial x_i) = 1$ for at least one of the x_i (an algorithm which assumes this condition will be given in

Section 2.7.2)

Algorithm V (Factorization in $GF(p^n)[x_1,\ldots,x_r]$). Let $D = GF(p^n)[x_1,\ldots,x_{r-1}]$, $D_o = \{0\} \cup \{$polynomials in D with leading numerical coefficient equal to unity$\}$. Given a primitive D_o polynomial $A \in D[x_r] = GF(p^n)[x_1,\ldots,x_r]$, this algorithm finds the complete factorization F of A into D_o polynomials, using an assumed factorization algorithm for $GF(p^n)[x_1,\ldots,x_r]$ which is applied only to polynomials E which satisfy $\gcd(E,\partial E/\partial x_i) = 1$ for at least one of the x_i.

(1) Set $k \leftarrow 0$, $m \leftarrow 1$, $s \leftarrow 0$, $F \leftarrow \emptyset$.

(2) [Look for a nonvanishing partial derivative.]

(a) Set $i \leftarrow r$.

(b) If $i = s$, go to (d).

(c) Set $A_1 \leftarrow \partial A/\partial x_i$. If $A_1 \neq 0$, go to (4).

(d) Set $i \leftarrow i-1$. If $i \geq 1$, go to (b).

(3) (All partial derivatives of A vanish, hence A is a p^{th} power.) Set $A \leftarrow A^{1/p}$, $k \leftarrow k+1$, $m \leftarrow mp$, (now $m = p^k$), and go to (2).

(4) ($A_1 = \partial A/\partial x_i \neq 0$.) Set $j \leftarrow 1$, $B \leftarrow \gcd(A,A_1)$, $C \leftarrow A/B$.

(5) Set $D \leftarrow \gcd(B,C)$, $E \leftarrow C/D$.

(6) (E is the product of distinct prime polynomials P such that $\partial P/\partial x_i \neq 0$, hence $\gcd(E,\partial E/\partial x_i) = 1$.) Factor E into D_o polynomials and put each factor into F with multiplicity jm.

(7) If $D \neq 1$, set $B \leftarrow B/D$, $C \leftarrow D$, $j \leftarrow j+1$, and go to (5).

(8) If $B \neq 1$, set $A \leftarrow B$, $s \leftarrow i$, and go to (2). Otherwise, exit.

2.5. Homomorphisms and sets of representatives

In succeeding sections we shall be studying the application of homomorphic mappings to factorization. We shall assume the usual definitions and basic theorems about ring homomorphisms, isomorphisms, ideals, residue class rings, canonical homomorphisms and kernel of a homomorphism (see [VDW49] or [GOL70]). In this section we review some related concepts which are important to the use of homomorphisms in practical computation.

Throughout this section we assume that D and E are sets and $h: D \to E$ is a mapping of D onto E. The set $P = \{h^{-1}(e): e \in E\}$, where $h^{-1}(e) = \{d \in D: h(d) = e\}$, is a partition of D. Let R be a subset of D such that for each set $S \in P$, $R \cap S$ contains exactly one element. Then R is a (complete) set of representatives of P. In other words, for each $e \in E$ there is a unique $d \in R$ such that $h(d) = e$. We assume this property of R in what follows.

We denote by h_R the restriction of h to R. The map $h_R: R \to E$ is one-to-one. We denote the inverse map of E onto R by h_R^{-1}.

We now assume that $(D,+,\cdot)$ and $(E,+_1,\cdot_1)$ are commutative rings with identity, and that h is a homomorphism of D onto E. We denote the kernel of h by Ker h. We recall that the residue class ring D/Ker h= {d+Ker h:d ε D} is isomorphic to E. The partition P of D, as defined above, is in this case the set D/Ker h.

The set R of representatives of D/Ker h may be made into a ring as follows. Let $\hat{h} = h_R^{-1} \circ h$, and for $a,b \in R$ define

$$a +_2 b = \hat{h}(a+b),\ a \cdot_2 b = \hat{h}(ab). \qquad (1)$$

We know that $h_R^{-1}: E \to R$ is one-to-one and onto. Let $a_1, b_1 \in E$ and $a = h_R^{-1}(a_1)$, $b = h_R^{-1}(b_1)$. Then

$$h_R^{-1}(a_1 +_1 b_1) = h_R^{-1}(h(a) +_1 h(b)) = h_R^{-1}(h(a+b))$$
$$= \hat{h}(a+b) = a +_2 b = h_R^{-1}(a_1) +_2 h_R^{-1}(b_1),$$

and similarly $h_R^{-1}(a_1 \cdot_1 b_1) = h_R^{-1}(a_1) \cdot_2 h_R^{-1}(b_1)$. From these relations the ring axioms for $(R,+_2,\cdot_2)$ may be verified, and it follows that $(R,+_2,\cdot_2)$ is a ring isomorphic to $(E,+_1,\cdot_1)$ under h_R^{-1}.

Furthermore, $\hat{h}$ is a homomorphism of D onto R: for $a,b \in D$,

$$\hat{h}(a+b) = h_R^{-1}(h(a+b)) = h_R^{-1}(h(a) +_1 h(b))$$
$$= h_R^{-1}(h(a)) +_2 h_R^{-1}(h(b))$$
$$= \hat{h}(a) +_2 \hat{h}(b),$$

and similarly for multiplication.

The point of this is that if we can do arithmetic in the ring D (i.e. if we have algorithms for performing the operations + and · on symbols representing the elements of D), then we can also do arithmetic in E, provided that we also have an algorithm for $\hat{h}$: we represent the elements of E by the symbols representing the elements of R and perform addition and multiplication on these symbols according to (1).

As an example, take D = Z (integers); m a positive integer; E = Z/(m), the residue class ring of integers modulo m; and h the canonical map $n \to n+(m)$. $R = \{0,1,\ldots,m-1\}$ is a set of representatives of E. Define a map $\phi_m: Z \to Z$ by $\phi_m(n)$ = least non-negative remainder

on division of n by m. Then $h_R^{-1}(n+(m)) = \phi_m(n)$ and $\hat{h} = \phi_m$. Thus if we define

$$a +_2 b = \phi_m(a+b),\ a \cdot_2 b = \phi_m(ab) \qquad (2)$$

on R then $(R,+_2,\cdot_2)$ is a ring isomorphic to E, and is the homomorphic image of Z under ϕ_m.

Now suppose we take D = Z, E = {0,1,...,m-1} with $+_2$ and $\cdot_2$ defined on E by (2), and $h = \phi_m$. If we take R = E = {0,1,...,m-1}, then we have a particularly simple situation: h_R and h_R^{-1} are the identity map of R and $\hat{h} = h$. The same situation occurs in general when we have E⊂D and take R = E.

When D = Z and E is isomorphic to Z/(m), it is often most convenient to take

$$R = \{-\lfloor \tfrac{m}{2} \rfloor,\ldots,0,1,\ldots,\lfloor \tfrac{m}{2} \rfloor\}$$

(with, say, $-\lfloor \frac{m}{2} \rfloor$ omitted if m is even): this will be seen to be true in the applications discussed in the following section. With E = {0,1,...,m-1} and $h = \phi_m$, as above, we have

$$h_R^{-1}(n) = \begin{cases} n, \text{ if } n \le \lfloor m/2 \rfloor, \\ n-m, \text{ otherwise;} \end{cases}$$

$$\hat{h}(n) = \begin{cases} \phi_m(n), \text{ if } \phi_m(n) \le \lfloor m/2 \rfloor, \\ \phi_m(n)-m, \text{ otherwise.} \end{cases}$$

Another important example is D = F[x], F is a field; E = F[x]/I, where I is an ideal generated by a polynomial G(x) of degree n > 0; and h = canonical homomorphism $A(x) \mapsto A(x)+I$. R = {A(x) ε F[x]: deg A < n} is a set of representatives of E. We then have

$$h_R^{-1}: A(x)+I \to A(x) \bmod G(x),$$

$$\hat{h}: A(x) \to A(x) \bmod G(x),$$

where A(x) modG(x) is the remainder on division by G(x).

There is one more definition we need to make for use in succeeding sections, that of an induced homomorphism. In general, let $(D,+,\cdot)$ and $(E,+_1,\cdot_1)$ be rings and h be a mapping of D onto E. For $A \in D[x]$, $A(x) = a_n x^n+\ldots+a_1x+a_0$, $a_n \neq 0$, define

$$h^*(A) = h(a_n)x^n +_1 \ldots +_1 h(a_1)x +_1 h(a_0).$$

Also define $h^*(0) = h(0)$. Thus $h: D \to E$ induces a mapping $h^*: D[x] \to E[x]$. Let h be a homomorphism; then it is easily verified that h^* is also a homomorphism. We shall generally denote this induced homomorphism by h also. If R is a set of representatives of D/Ker h then $h_R^{-1}: E \to R$ similarly induces a map of E[x] onto R[x], which we also denote by h_R^{-1}. Thus for each polynomial B in E[x], $A = h_R^{-1}(B)$ is the unique polynomial in D[x] with coefficients in R such that deg A = deg B and h(A) = B.

Although we have in this section distinguished between the binary operations of D and E, we shall in the remaining sections follow the usual convention of allowing the context to determine which operation is intended; for example if $a,b \in E$ then a+b means $a +_1 b$ where $+_1$ is the addition operation on E.

2.6. Factoring via induced homomorphisms

Let D be a UFD and h be a homomorphism of D onto a ring E. Let C be a nonzero polynomial over D. In this section we begin the study of ways in which induced homomorphisms may be used to find the complete factorization of C over D. These methods are based on the factor-preserving property of homomorphisms: if C = AB then h(C) = h(A)h(B). Letting R be a set of representatives of D/Ker h, we know that each factor G of h(C) uniquely determines a polynomial $F = h_R^{-1}(G)$ with coefficients in R such that h(F) = G. Suppose that the coefficients of the factor A of C all lie in R. If we have some way of enumerating all of the factors of h(C), then when we consider the factor G = h(A) we obtain $F = h_R^{-1}(G) = A$. Thus if it is known a priori that the coefficients of every factor of C lie in R, then the factors can be determined by considering each factor G of h(C) and testing, by division, whether $h_R^{-1}(G)$ is a factor of C.

An important special case is obtained by taking D = Z, the integers; E = Z/(m), the ring of integers modulo m; and h = the canonical mapping $n \to n+(m)$. Given $C \in Z[x]$, it is possible to compute a bound b for the coefficients of any factor of C without actually factoring C. (Such bounds will be discussed at length in Section 3.4.) Let m be an odd integer > 2b and R be the set of integers in the range $-\frac{m}{2} < n < \frac{m}{2}$. Then R is a set of representatives of Z/(m) and the coefficients of every factor of C lie in R. Thus by considering all factors G of h(C) in (Z/(m))[x] and the corresponding polynomials $h_R^{-1}(G)$ we can find the factors of C.

If m = p, a prime, then Z/(p) is a field called the Galois field of order p and also denoted by GF(p). In this case, h(C) has a unique factorization into prime polynomials in GF(p)[x] from which the set of all factors of h(C) may be computed. The complete factorization of polynomials in GF(p)[x] may be accomplished by means of one of several algorithms discovered by Berlekamp. (See [BER68], [KNU69], Section 4.6.2· [COL69a], Section 3.8: and [BER70].)

If m is not prime, then (Z/(m))[x] is not a UFD. (Z/(m) is not even an integral domain.) However, if $m = p^j$, a power of a prime, then certain polynomials in (Z/(m))[x] do have a "complete factorization" in a sense to be defined in Section 2.7. A complete factorization mod p^j may be determined from a complete factorization mod p by means of algorithms to be presented in Section 2.7. Thus m may be taken as a power of a small prime p, the advantage being that it is much easier to factor over GF(p) when p is relatively small: although Berlekamp describes an algorithm in [BER70] which appears to be reasonably efficient for large primes, it is much more complicated than his original algorithm [BER68] (which is practically efficient only for small primes).

Another important case is obtained by considering homomorphisms from $D = D_1[w]$ onto E, where D_1 is a UFD, w is an indeterminate, and E is a ring. For example, let h be the canonical homomorphism from $D_1[w]$ to $E = D_1[w]/I$, where I is an ideal generated by a polynomial A(w) of degree $n > 0$, with ldcf (A) a unit of D_1.

Given a polynomial C in $D_1[w,x] = D_1[w][x]$, we may attempt to factor it by studying factors of h(C) in E[x]. A set R of representatives of E is given by the set of all polynomials in $D_1[w]$ of degree < n. If the coefficients of C (as polynomials in $D_1[w]$) are all of degree < n, then so must be the coefficients of any factor of C. Thus, again, the factors of C will be found by considering all possible factors G of h(C) and the corresponding polynomials $h_R^{-1}(G)$.

If we take D_1 to be a field and A(w) to be irreducible over D_1, then $E = D_1[w]/I$ is an extension field of D_1. Hence, if we know how to factor over this extension field, we can factor polynomials in $D_1[w,x]$ of degree < n in w.

In particular, if $D_1 = GF(p)$, then $E \cong GF(p^n)$, the Galois field of order p^n. Factorization over $GF(p^n)$ can be performed with reasonable efficiency by one of Berlekamp's more recent algorithms ([BER70]).

2.6.1. Monic algorithm

We have thus far only vaguely indicated the nature of some factoring algorithms based on the use of induced homomorphisms. Let us now consider the details of such algorithms, developing them first as abstract algorithms, then considering the special cases of factoring in Z[x] and GF(p)[w,x].

The first case we shall study will be that in which both D and E are unique factorization domains, as in the above example of Z and GF(p). We are given a polynomial C over D to be factored: to

simplify the presentation we shall initially assume that C is monic. In Section 2.6.2 we shall generalize to allow C to be primitive with arbitrary leading coefficient.

Let R be a subset of D and suppose that R contains the coefficients of every monic factor of C of degree $\leq \lfloor(\deg C)/2\rfloor$. Then we say that C is R-factorable. Obviously we have $1 \in R$. Suppose h is a homomorphism of D onto E and R is a set of representatives of D/Ker h. Then $h_R^{-1}(1) = 1$, and thus for every monic factor A_0 of h(C) of degree d we have that $h_R^{-1}(A_0)$ is monic of degree d.

Algorithm M (Factoring a monic polynomial via the complete factorization of its homomorphis image). Let D and E be UFDs, h be a homomorphism from D onto E, and R be a set of representatives of D/Kerh. Given a monic polynomial C over D which is R-factorable and the complete monic factorization G of h(C), this algorithm computes the complete monic factorization F of C. (G is the multiset of prime monic polynomials over E such that $h(C) = \Pi G$, and F is the multiset of prime monic polynomials over D such that $C = \Pi F$).

(1) Set $F \leftarrow \emptyset$, $d \leftarrow 1$.

(2) If $d > \lfloor(\deg C)/2\rfloor$, set $F \leftarrow F \uplus \{C\}$ and exit.

(3) For each $H \subset G$ such that $\Sigma \deg H = d$:

  (a) Set $A_0 \leftarrow \Pi H$, $A \leftarrow h_R^{-1}(A_0)$.

  (b) If $A \mid C$, go to (5).

(4) Set $d \leftarrow d+1$ and go to (2).

(5) Set $F \leftarrow F \uplus \{A\}$, $C \leftarrow C/A$, $G \leftarrow G-H$, and go to (2).

Note: The meaning of step (3) is that steps (3a) and (3b)

are to be performed with every distinct multiset $H \subset G$ such that the sum of the degrees of the members of H is d. If the test in step (3b) fails for each such H then control passes to step (4). If the test succeeds for some H then control passes immediately to step (5) without steps (3a) and (3b) being performed for the remaining multisubsets. A systematic way of generating all of the multisets $H \subset G$ with $\Sigma \deg H = d$ was described in Section 1.6.

Validity proof: Let C_0 be the initial value of C. We shall show that whenever we begin an execution of step (2) the following conditions hold:

a. C is R-factorable;

b. G is the complete monic factorization of h(C);

c. $C_0 = C \amalg F$;

d. all $A \in F$ are prime, monic polynomials;

e. C is monic;

f. C has no factor B such that $0 < \deg B < d$.

From the input assumptions and the initial definitions $F \leftarrow \emptyset$ and $d \leftarrow 1$, it is evident that these assertions hold after we perform step (1). Assume that we have arrived at step (2) with the assertions true and that $d \leq \lfloor (\deg C)/2 \rfloor$. Then we proceed to step (3) where we begin computing all polynomials $A = h_R^{-1}(\amalg H)$ such that $H \subset G$ and $\Sigma \deg H = d$. By the remarks preceding the algorithm, each polynomial A is monic of degree d. This ensures that if C has no factors of degree d, then the division test in step (3b) must fail for each H and control will pass to step (4). In this case d is increased, but

assertion f remains true. The other assertions are also unaffected and we return to step (2) with all of the assertions still valid.

On the other hand, suppose C does have a factor A of degree d. By e, we may assume A is monic, and by a, R contains the coefficients of A. Also h(A) is a monic factor of h(C) of degree d; hence, by b, $h(A) = \Pi H$ for some $H \subset G$ such that $\Sigma \deg H = d$. We conclude that in step (3) we must eventually find a factor of C of degree d: either A or some other monic factor of degree d.

Assume that A is the factor found; then in step (5) we put A into F. From f we deduce that A is prime, so assertion d remains valid. Assertions a,b,c,e, and f are also obviously satisfied by the new values of C and G, and thus in this case also we return to step (2) with all of the assertions true.

Termination of the algorithm is ensured by the fact that the non-negative integer deg C-d decreases between successive executions of step (2). When we find $d > \lfloor (\deg C)/2 \rfloor$, we put C into F and terminate. By f, C has no factors of positive degree $\leq \lfloor (\deg C)/2 \rfloor$, hence no proper factors at all. Thus C is prime and by c,d, and e, the final value of F contains only prime monic polynomials and $C_0 = \Pi F$. This concludes the proof.

Now suppose D is a UFD with multiplicative ample set D_0 and C is a primitive D_0 polynomial over D. Suppose also that we have an algorithm for factoring monic polynomials over D. Then C may be factored by use of a monic transformation, as in the following algorithm:

(1) Set $n \leftarrow \deg C$, $c \leftarrow \mathrm{ldcf}\ C$, $C_1(x) \leftarrow c^{n-1}C(x/c)$

(C_1 is a monic polynomial over D.)

(2) Find the complete monic factorization F_1 of C_1.

($C_1 = \Pi F_1$).

(3) Set $F \leftarrow \{B : B = \mathrm{pp}(A(cx)),\ A \in F_1,\ \mathrm{ldcf}\ B \in D_0\}$

(Then $C = \Pi F$ and F is the complete D_0 factorization of C.)

The validity of this algorithm follows from Gauss's Lemma (the product of primitive polynomials is primitive). Actually it can be shown that $c^{\deg A - 1} \mid A(cx)$ for $A \in F_1$, so in step (3) we could compute $\mathrm{pp}(A(cx)/c^{\deg A - 1})$, resulting in more efficient calculation. (See [KNU69], Exercise 4.6.2-18.)

Thus we could content ourselves with developing factoring algorithms which are restricted to monic polynomials, such as Algorithm M. However, transforming to a monic polynomial has its disadvantages. For example, if D = Z and c is a large integer, then C_1 will have very large coefficients. Thus we shall consider a generalization of Algorithm M which directly handles a primitive polynomial with arbitrary leading coefficient, thereby making the monic transformation unnecessary.

2.6.2. Primitive algorithm

Algorithm M and its validity proof demonstrate that in factoring a monic polynomial C via the factorization of its homomorphic image h(C) it suffices to consider the monic factors of h(C). The generalization of Algorithm M to be presented will analogously consider

only those factors of h((ldcf C)C) which have leading coefficient h(ldcf C). This device is based on a suggestion of Collins.

We first need to generalize the definition of R-factorable polynomials, which we previously gave only for monic polynomials. Let R be a subset of D and C be a polynomial over D with ldcf C = c. Let us say that C is R-factorable if R contains the coefficients of every factor A* of C* = cC such that deg A* $\leq \lfloor$(deg C)/2$\rfloor$ and ldcf A*|c. We have c ε R and thus if h is a homomorphism of D onto E and R is a set of representatives of D/Ker h, we have $h_R^{-1}(h(c)) = c$. Therefore, for every factor A_1 of h(C*) such that deg A_1 = d and ldcf A_1 = h(c), we have that $h_R^{-1}(A_1)$ is a polynomial of degree d with leading coefficient c.

Lemma 1. Let D be an integral domain, R be a subset of D, and C be a polynomial over D which is R-factorable. Then any factor of C is also R-factorable.

Proof: Let C = AB. We shall show that A is R-factorable. Let (ldcf A)A = A_1A_2, where deg $A_1 \leq \lfloor$(deg A)/2$\rfloor$ and ldcf A_1|ldcf A. Then A_1|(ldcf C)C, deg $A_1 \leq \lfloor$(deg C)/2$\rfloor$ and ldcf A_1|(ldcf C); hence R contains the coefficients of A_1, proving that A is R-factorable.

We shall further generalize Algorithm M by not requiring the image domain E to be a UFD. We shall only assume that E is a commutative ring with identity, since we shall want to use the algorithm in applications in which E is not even an integral domain. We thus need to extend the concept of a "complete factorization," which we have thus far defined only in the case of an integral domain. For

our purposes it seems best to give the definition directly in terms of the properties which will be needed in the algorithm. Let C be a polynomial over E and G be a multiset of polynomials over E. We shall say that G is a complete factorization of C over E iff

a. $C = e\Pi G$ for some $e \in E$;

b. for every factorization $C = AB$ such that $\deg C = \deg A + \deg B$ there is a unique $H \subset G$ such that $A = e\Pi H$ for some $e \in E$;

c. the leading coefficient of each polynomial in G is not a zero divisor.

Note that if E is a UFD, E_0 is a multiplicative ample set for E and G is the complete E_0 factorization of pp(C), as defined previously, then C and G have the above properties. We defer until Section 2.7 the presentation of a specific example of a complete factorization of a polynomial over a ring which is not an integral domain.

Lemma 2. Let E be a commutative ring with identity, C be a polynomial over E, and G be a complete factorization of C over E. Assume that $C = AB$ with $\deg C = \deg A + \deg B$ and that $A = e\Pi H$. Then G-H is a complete factorization of B over E.

Proof: We first have to show that $B = e^*\Pi(G-H)$ for some $e^* \in E$. Since $C = C \cdot 1$ where $\deg C = \deg C + \deg 1$, we know that G is the unique subset of G such that $C = e_0\Pi G$ for some $e_0 \in E$. We furthermore know that there is a unique $H^* \subset G$, such that $B = e^*\Pi H^*$ for some $e^* \in E$. Hence $C = AB = ee^*\Pi(H \uplus H^*)$, hence $H \uplus H^* = G$,

hence $H^* = G-H$, proving that $B = e^*\Pi(G-H)$.

Next let $B = B_1B_2$ with $\deg B = \deg B_1 + \deg B_2$. We must show that there is a unique $H' \subset G-H$ such that $B_1 = e'\Pi H'$ for some $e' \in E$.

We have $\deg C = \deg A + \deg B_1 + \deg B_2 \geq \deg (AB_2) + \deg B_1$, while from $C = (AB_2)B_1$ we have $\deg C \leq \deg AB_2 + \deg B_1$, hence $\deg C = \deg (AB_2) + \deg B_1$. Hence there is a unique $H' \subset G$ such that, for some $e' \in E$, $B_1 = e'\Pi H'$. We thus have

$$AB_1 = ee'\Pi(H \uplus H').$$

By the same argument as above we have $C = (AB_1)B_2$ with $\deg C = \deg (AB_1) + \deg B_2$, hence there is a unique $H'' \subset G$ such that, for some $e'' \in E$, $AB_1 = e''\Pi H''$. Hence $H'' = H \uplus H'$, showing that $H \uplus H' \subset G$, hence that $H' \subset G-H$, which completes the proof.

Algorithm P (Factoring a primitive polynomial via a factorization of its homomorphic image). Let D be a UFD with multiplicative ample set D_0, and E be a commutative ring with identity. Let h be a homomorphism from D onto E and R be a set of representatives of D/Kerh, with $0 \in R$. The inputs are a primitive D_0 polynomial C and a multiset G of polynomials over E such that:

a. C is R-factorable;

b. G is a complete factorization of h(C) over E.

The output of the algorithm is the multiset F, the complete D_0 factorization of C.

(1) Set $F \leftarrow \emptyset$, $d \leftarrow 1$.

(2) Set $c \leftarrow \text{ldcf } C$, $\overline{c} \leftarrow h(c)$, $C^* \leftarrow c \cdot C$.

(3) If $d > \lfloor (\deg C)/2 \rfloor$, set $F \leftarrow F \biguplus \{C\}$ and exit.

(4) For each $H \subset G$ such that $\Sigma \deg H = d$:

(a) Set $A_0 \leftarrow \Pi H$, $A_1 \leftarrow (\overline{c}/\text{ldcf } A_0)A_0$, $A^* \leftarrow h_R^{-1}(A_1)$.

(b) If $A^* | C^*$, set $B^* \leftarrow C^*/A^*$ and to to (6).

(5) Set $d \leftarrow d+1$ and go to (3).

(6) Set $A \leftarrow pp(A^*)$ (with ldcf $A \in D_0$), $F \leftarrow F \biguplus \{A\}$, $C \leftarrow B^*/\text{ldcf } A$, $G \leftarrow G-H$, and go to (2).

Validity proof: We let C_0 be the initial value of C and prove that at the beginning of each execution of step (3) the conditions a and b of the input assumptions and the following conditions all hold:

c. $C_0 = C \Pi F$;

d. all $A \in F$ are prime, primitive D_0 polynomials;

e. $c = \text{ldcf } C \in D_0$, $\overline{c} = h(c)$, $C^* = cC$.

f. C has no factor B such that $0 < \deg B < d$.

These conditions hold after performing steps (1) and (2). Assume then that we arive at step (3) with the conditions valid and that $d \leq \lfloor (\deg C)/2 \rfloor$.

We first show that, for any $H \subset G$ such that $\Sigma \deg H = d$, the product $A_0 = \Pi H$ is of degree d and ldcf $A_0 | \overline{c}$. Assertion b implies that the leading coefficient of every member of H is not a zero divisor; hence $\deg A_0 = \deg \Pi H = \Sigma \deg H = d$. Also $h(C) = e \Pi G$ for some $e \in E$, hence $h(C) = e \Pi H \Pi (G-H) = e A_0 B_0$, where $B_0 = \Pi(G-H)$. Neither ldcf A_0 nor ldcf B_0 can be a zero divisor, hence $\deg h(C) = \deg A_0 + \deg B_0$ and ldcf $h(C) = e(\text{ldcf } A_0)(\text{ldcf } B_0)$: thus ldcf $A_0 |$ ldcf $h(C)$, and it

remains to show that ldcf h(C) = $\overline{c}$.

Since C is R-factorable, we have c = ldcf C ε R. Since 0 ε R, we cannot have h(c) = 0; hence deg h(C) = deg C and ldcf h(C) = h(c) = $\overline{c}$; hence ldcf $A_0 | \overline{c}$.

Thus if $A_1 = (\overline{c}/\text{ldcf } A_0)A_0$ then A ε E[x] and deg A_1 = d, and if A* = $h_R^{-1}(A_1)$ then A* ε D[x] and deg A* = d.

Now consider the case in which C has no factor of degree d. For any A* which divides C*, pp(A*) divides C, hence C* can have no factor of degree d. But, as we have just shown, every A* computed in step (4a) is of degree d, so the division test in step (4b) must fail for each H. Control thus passes to step (5) where d is increased, but condition f, as well as all of the other conditions, remains valid as we return to step (3).

Now assume that C does have a factor A of degree d. We may assume that ldcf A ε D_0. Letting C = AB, we have deg h(A) $\leq$ deg A, deg h(B) $\leq$ deg B, hence deg h(A) + deg h(B) $\leq$ deg A + deg B = deg C = deg h(C) (the last equality was proved above). But from h(C) = h(A)h(B) we have deg h(A)+deg h(B) $\geq$ deg h(C), and it follows that deg h(A) + deg h(B) = deg h(C), deg h(A) = deg A, deg h(B) = deg B. Thus, by b, there is a unique H $\subset$ G such that h(A) = eΠH for some e ε E.

Let A_0 = ΠH, A_1 = (e/ldcf A_0)A_0 and A* = $h_R^{-1}(A_1)$. We shall now show that A* = bA, where b = ldcf B.

We have h(bA) = h(b)h(A) = h(b)eA_0 = h(b)(ldcf h(A)/ldcf A_0)A_0 = $(\overline{c}/\text{ldcf } A_0)A_0$, hence h(bA) = A_1. But ldcf (bA) = c, (bA)|C*, deg (bA) $\leq$ $\lfloor$(deg C)/2$\rfloor$, and C is R-factorable; hence R contains the

coefficients of bA. Thus $bA = h_R^{-1}(A_1) = A^*$, as desired.

We thus have $pp(A^*) = pp(bA) = pp(A) = A$. We see that step (4) computes from H a polynomial A^* such that $A^* | C^*$ and $pp(A^*) = a$. Thus we must eventually find a factor of C of degree d: either A or some other D_0 factor of degree d.

Assume that A is the factor found; then in step (6) we put A into F. By f, A must be prime, so condition d remains valid. Also $B^*/ldcf\ A = (C^*/A^*)/ldcf\ A = (cC/bA)/ldcf\ A = C/A$, so the new value of C satisfies condition c. By Lemma 1, condition a remains valid and by Lemma 2 so does condition b. Condition f also remains valid, and, after executing step (2), so does condition e. We thus return to step (3) with all of the conditions holding.

The rest of the argument is the same as in the proof of Algorithm M.

2.6.3. Application to factoring over Z

Assuming that we have an efficient algorithm for factoring over GF(p), the most immediate application of Algorithm P is, as we indicated earlier, to factoring over Z. We shall now consider this application in more detail.

For this purpose, let us represent the elements of GF(p) by $\{0,1,\ldots,p-1\}$ and let $h_p: Z \to GF(p)$ be the map $n \to n \bmod p$ (least nonnegative remainder on division by p).

A polynomial over Z will be called positive if its leading coefficient is positive.

Algorithm Z (Factoring over Z.) Given a primitive, positive polynomial C over Z, this algorithm finds the complete factorization F of C into prime positive polynomials over Z.

(1) Compute a bound b on the coefficients of any factor of C of degree $\leq \lfloor (\deg C)/2 \rfloor$.

(2) Choose a prime integer $p > 2b(\text{ldcf } C)$.

(3) Set $\overline{C} \leftarrow h_p(C)$ and factor $\overline{C}$ over GF(p), obtaining the multiset G of prime monic polynomials over GF(p) such that $\overline{C} = (\text{ldcf } \overline{C})\Pi G$.

(4) If order (G) = 1 (i.e. if $\overline{C}$ is prime) then set $F \leftarrow \{C\}$ and exit. (C is prime).

(5) Apply Algorithm P (with D = Z, E = GF(p), D_0 = nonnegative integers, $h = h_p$, $R = \{n: |n| < p/2\}$) to C and G, obtaining the multiset F of prime positive polynomials over Z such that $C = \Pi F$. Exit.

In step (2) we assume a method is available for finding a prime integer greater than a given positive integer.

In step (4), we know that if $\overline{C}$ is a prime then so is C. For if there were a proper factorization C = AB then deg A > 0 and deg B > 0, because C is primitive. Since p > ldcf C, p > ldcf A and p > ldcf B, hence deg $h_p(A)$ = deg A > 0, deg $h_p(B)$ = deg B > 0. Thus $\overline{C} = h_p(A)h_p(B)$ would be a proper factorization of $\overline{C}$, contrary to the assumption that $\overline{C}$ is prime.

To show that C is R-factorable with $R = \{n \in Z: |n| < p/2\}$, suppose $C^* = A^*B^*$, where deg $A^* \leq \lfloor (\deg C)/2 \rfloor$ and a^* = ldcf A^* divides c = ldcf C. Let $A = pp(A^*)$ and a = ldcf A. Thus $A|C$ and

deg $A \leq \lfloor (\deg C)/2 \rfloor$, hence $|A|_\infty \leq b$. Also $A^* = a'A$ for some a' and $a^* = a'a$ divides c, hence $|a'| \leq c$. Hence $|A^*|_\infty = |a'| \cdot |A|_\infty \leq cb < p/2$, showing that C is R-factorable.

The main virtue of this algorithm would seem to be its simplicity; however, this is somewhat deceptive since the only practical algorithm for factoring over GF(p) for large p is an algorithm recently developed by Berlekamp [BER70], and this algorithm is quite complicated. In Section 2.7.1 we shall discuss an alternative algorithm for factoring over Z which allows the use of Berlekamp's simpler algorithm [BER68] using only small primes.

2.6.4. Application to factoring in GF(p)[w,x]

As further illustration of the application of Algorithm P, we shall now give an algorithm for factoring bivariate polynomials with coefficients from GF(p) under the assumption that we have a practical algorithm for factoring univariate polynomials over $GF(p^n)$. This algorithm will be quite similar to Algorithm 2.6.3Z.

We noted earlier that $GF(p^n) \cong GF(p)[w]/I$, where I is a principal ideal generated by a prime polynomial $A \in GF(p)[w]$, deg A = n. A set of representatives of GF(p)[w]/I is given by $R = \{B \in GF(p)[w] : \deg B < n\}$. Thus we may take $GF(p^n) = R$, performing arithmetic on the elements of R modulo A. We let h_A be the map $B \to B \bmod A$.

Algorithm B (Factoring in GF(p)[w,x].) Let D = GF(p)[w], $D_0 = \{0\} \cup \{\text{monic polynomials in D}\}$. Given a primitive D_0 polynomial

C in D[x] = GF(p)[w,x], this algorithm finds the complete factorization F of C into prime D_0 polynomials over D.

(1) Set $b \leftarrow \max_{0 \le i \le k}\{\deg C_i\}$, where $C_k, C_{k-1}, \ldots, C_0$ are the coefficients of C.

(2) Choose a monic prime polynomial A in D of degree $n > b + \deg C_k$

(3) Set $\overline{C} \leftarrow h_A(C)$ and factor $\overline{C}$ over $GF(p^n)$, obtaining the multiset G of prime monic polynomials over $GF(p^n)$ such that $\overline{C} = (\text{ldcf } \overline{C})\Pi G$.

(4) If order (G) = 1 (i.e. if $\overline{C}$ is prime) then set $F \leftarrow \{C\}$ and exit. (C is prime).

(5) Apply Algorithm P (with D = GF(p)[w], E = $GF(p^n)$, D_0 as above, $h = h_A$, R = E) to C and G, obtaining the multiset F of prime D_0 polynomials in GF(p)[w,x] such that $C = \Pi F$. Exit.

2.7 Hensel algorithms

The algorithms of this section are based on the classical theory of p-adic fields, which was first investigated by Hensel about 1900. The application of Hensel's constructions to practical factorization of polynomials was suggested by Zassenhaus [ZAS69]. The next two algorithms are based, however, on Van der Waerden's presentation of Hensel's Lemma (Reducibility Criterion) ([VDW49], pp. 248-250). In Section 2.8 we shall discuss Zassenhaus' version of Hensel's algorithm and the generalized algorithms that result therefrom.

Algorithm S (Solution of a polynomial equation). Let E be a commutative ring with identity. Given $A,B,S,T,U \in E[x]$ such that ldcf A is a unit of E and $AS+BT = 1$, this algorithm computes $Y,Z \in E[x]$ such that $AY+BZ = U$ and $\deg Z < \deg A$.

(1) Set $V \leftarrow TU$.

(2) Using Algorithm 1.3D, compute $Q,Z \in E[x]$ such that

$$V = AQ+Z, \quad \deg Z < \deg A.$$

(3) Set $Y \leftarrow SU+BQ$ and exit. (Then $AY+BZ = A(SU+BQ)+B(TU-AQ) = (AS+BT)U = U$).

Theorem S. Under the assumptions of Algorithm S, the polynomials Y and Z are uniquely determined.

Proof: Let $AY_1+BZ_1 = U$ with $\deg Z_1 < \deg A$. Then $AY_1+BZ_1 = AY+BZ$, which may be written

$$A(Y_1-Y) = B(Z-Z_1). \tag{1}$$

Upon multiplying both sides by T and adding $AS(Z-Z_1)$ to both sides,

we obtain

$$A[S(Z-Z_1)+T(Y_1-Y_1)] = (AS+BT)(Z-Z_1) = Z-Z_1.$$

Unless the polynomial in brackets is zero, the degree of the product on the left side is $\geq$ deg A, since ldcf A is a unit. But $\deg(Z-Z_1) < \deg A$, so we conclude that $Z = Z_1$ and by (1) we then have $A(Y_1-Y) = 0$, which, with the fact that ldcf A is a unit, implies $Y_1 = Y$.

The following lemma will be required in the proof of the next algorithm.

Lemma 1. Let D be a commutative ring with identity and $a,b \in D$. If a is a unit modulo b then, for any positive j, a is a unit modulo b^j.

Proof: For some $s \in D$ we have $as \equiv 1 \pmod{b}$. Let $j > 1$; we may assume by induction that $as^* \equiv 1 \pmod{b^{j-1}}$ for some $s^* \in D$. Hence there exist $t, t^* \in D$ such that $as+bt = 1$, $as^*+b^{j-1}t^* = 1$. Therefore

$$asb^{j-1} + b^j t = b^{j-1},$$

$$1 = as^* + b^{j-1}t^* = as^* + (asb^{j-1} + b^j t)t^*,$$

$$= a(s^* + sb^{j-1}t^*) + b^j tt^*,$$

$$as^+ \equiv 1 \pmod{b^j},$$

where $s^+ = s^* + sb^{j-1}t^*$. Thus a is a unit modulo b^j.

Algorithm H. (Hensel method for constructing a factorization mod p^j from a given factorization mod p). Let D and E be commutative rings with identities, $p \in D$, and h be a homomorphism of D onto E with kernel (p). This algorithm takes inputs:

$m = p^j$ for some positive integer j;

$C \in D[x]$;

$\overline{A},\overline{B},\overline{S},\overline{T} \in E[x]$ such that ldcf $\overline{A}$ is a unit of E,

$h(C) = \overline{AB}$, and $\overline{AS}+\overline{BT} = 1$.

The outputs are $A,B \in D[x]$ such that $C \equiv AB \pmod m$, $h(A) = \overline{A}$, $h(B) = \overline{B}$, deg A = deg $\overline{A}$, and ldcf A is a unit modulo m.

(1) Set $q \leftarrow p$ and choose $A,B \in D[x]$ such that $h(A) = \overline{A}$, $h(B) = \overline{B}$, deg A = deg $\overline{A}$.

(2) (Now $A,B \in D[x]$, $C \equiv AB \pmod q$, $h(A) = \overline{A}$, $h(B) = \overline{B}$, deg A = deg $\overline{A}$ and ldcf A is a unit modulo q.) If $q = m$, exit.

(3) Set $U \leftarrow (C-AB)/q$, $\overline{U} \leftarrow h(U)$. (Since $C \equiv AB \pmod q$, we know $U \in D[x]$, hence $\overline{U} \in E[x]$.) Using Algorithm S with inputs $\overline{A},\overline{B},\overline{S},\overline{T},\overline{U}$, solve $\overline{AY}+\overline{BZ} = \overline{U}$ for $\overline{Y},\overline{Z} \in E[x]$ such that deg $\overline{Z}$ < deg $\overline{A}$.

(4) Choose $Y,Z \in D[x]$ such that $h(Y) = \overline{Y}$, $h(Z) = \overline{Z}$, deg Z = deg $\overline{Z}$. (Thus $AY+BZ \equiv U \pmod p$ and deg Z < deg A.)

(5) Set $A \leftarrow A+qZ$, $B \leftarrow B+qY$, $q \leftarrow qp$, and go to (2).

The assertions at step (2) obviously hold for the first execution of the step. To show that they still hold for subsequent executions, let $A^* = A+qZ$, $B^* = B+qY$, $q^* = qp$. Then

$$\begin{aligned} C-A^*B^* &= C-AB-q(AY+BZ)-q^2YZ \\ &= q(U-AY-BZ)-q^2YZ \\ &\equiv 0 \pmod{qp}, \end{aligned}$$

i.e. $C \equiv A^*B^* \pmod{q^*}$. Also we have $h(A^*) = h(A) = \overline{A}$, $h(B^*) = h(B) = \overline{B}$, deg A^* = deg A = deg $\overline{A}$ and ldcf A^* = ldcf A; since ldcf A is a unit modulo p, ldcf A^* is a unit modulo q^*, by Lemma 1. Thus from step (5)

we return to step (2) with all the assertions still valid.

We shall denote by D_Z the set of zero divisors in a ring D.

Theorem H. Let D be a commutative ring with identity, $p \in D-D_Z$, and j be a positive integer. Let $A,B,A_1,B_1 \in D[x]$ satisfy

a. $A_1B_1 \equiv AB \pmod{p^j}$

b. deg A_1 = deg A, ldcf $A_1 \equiv$ ldcf A $\pmod{p^j}$;

c. $A_1 \equiv A$ and $B_1 \equiv B \pmod{p}$;

d. ldcf A is a unit mod p.

Then $A_1 \equiv A$ and $B_1 \equiv B \pmod{p^j}$.

Proof: From c we have the conclusion when j = 1. Let j > 1. From a, we have $A_1B_1 \equiv AB \pmod{p^{j-1}}$, and from b, ldcf $A_1 \equiv$ ldcf A $\pmod{p^{j-1}}$, so we may assume by induction that $A_1 \equiv A$ and $B_1 \equiv B \pmod{p^{j-1}}$. Hence there exist $Y,Z \in D[x]$ such that $A_1 = A+p^{j-1}Z$, $B_1 = B+p^{j-1}Y$. Thus

$$A_1B_1 = AB+p^{j-1}(AY+BZ)+p^{2j-2}YZ,$$

$$0 \equiv p^{j-1}(AY+BZ) \pmod{p^j}.$$

From this congruence and the assumption that p is not a zero divisor follows

$$AY+BZ \equiv 0 \pmod{p}.$$

Also, by c we have deg Z $\leq$ deg A and in fact p|ldcf Z. Hence by Theorem S applied to the ring D/(p) we have $Y \equiv Z \equiv 0 \pmod{p}$, from which we obtain the conclusion of the theorem.

Consider now the case when the domain E in Algorithm H is a field. This will be the case, for example, if D is a Euclidean Domain and p is a prime element of D. For then if c is an element

of D not divisible by p then gcd(c,p) = 1 and we can use the Extended Euclidean Algorithm (Algorithm 2.2E) to find s and t in D such that cs+pt = 1. Hencs cs $\equiv$ 1 (mod p), hence h(c)h(s) = 1, proving that each nonzero element of the ring E has a multiplicative inverse, hence that E is a field.

In the next algorithm we use the fact that if the input $\overline{A}$ to Algorithm H is monic then in step (1) of the algorithm A may be chosen to be monic, in which case the final output A is monic.

Algorithm C (Construction of a sequence of factors mod p^j corresponding to a given sequence of factors mod p). Let D be a commutative ring with identity, p ε D and h be a homomorphism of D onto a field E with kernel (p). The inputs to the algorithm are:

$m = p^j$ for some positive integer j;

C ε D[x] such that h(C) is squarefree:

$G_1,\ldots,G_t$, a sequence of monic polynomials over E such that

$$h(C) = (\text{ldcf } h(C))\ G_1\ldots G_t.$$

The outputs are $U,F_1,F_2,\ldots,F_t \varepsilon D[x]$ such that

$C \equiv UF_1\ldots F_t \pmod{m}$;

$h(F_i) = G_i$, deg F_i = deg G_i, and F_i is monic, for i = 1,...,t.

(1) Set $\overline{C} \leftarrow h(C)$, $i \leftarrow 1$.

(2) (Now we have:

a. $C_0 \equiv CF_1\ldots F_{i-1} \pmod{m}$, where C_0 was the initial value of C;

b. $h(F_k) = G_k$, deg F_k = deg G_k, and F_k is monic for k = 1,...,i-1;

c. $\overline{C} = h(C) = (\text{ldcf } h(C))G_iG_{i+1}\ldots G_t$;

d. $\overline{C}$ is squarefree.)

Set $\overline{A} \leftarrow G_i$, $\overline{B} \leftarrow \overline{C}/\overline{A}$. (Thus $h(C) = \overline{AB}$, $\overline{A}$ is monic, and $\overline{A}$ and $\overline{B}$ are relatively prime over E, by d.)

(3) Using the Extended Euclidean Algorithm, obtain $\overline{S}$ and $\overline{T}$ over E such that $\overline{AS}+\overline{BT} = 1$.

(4) Apply Algorithm H to $m, C, \overline{A}, \overline{B}, \overline{S}, \overline{T}$ and let A and B be the output. (Thus $A,B \in D[x]$, $C \equiv AB \pmod m$, $h(A) = \overline{A}$, $h(B) = \overline{B}$, deg A = deg $\overline{A}$, and we may assume that A is monic, as noted above.)

(5) Set $F_i \leftarrow A$, $C \leftarrow B$, $\overline{C} \leftarrow \overline{B}$, $i \leftarrow i+1$. (Thus conditions a,b,c and d remain valid).

(6) If $i \leq t$, go to (2).

(7) Set $U \leftarrow C$ and exit.

The next theorem shows that, with some additional assumptions, the output F from Algorithm C is a complete factorization of C modulo m.

Theorem C. Given the assumptions of Algorithm C, let $G = \{G_1,\ldots,G_t\}$ and $F = \{F_1,\ldots,F_t\}$. Also assume $p \in D-D_Z$, $p \nmid$ ldcf C, and each G_i is prime. Then

a. $C \equiv (\text{ldcf } C)\Pi F \pmod m$

b. For every factorization $C \equiv AB \pmod m$ such that deg C = deg A+deg B, there exists a unique $F' \subset F$ such that $A \equiv (\text{ldcf } A)\Pi F' \pmod m$.

Proof: a. We have, from the algorithm, $C \equiv U\Pi F \pmod m$. Since ΠF is monic, deg C = deg U + deg ΠF. Also, since $p \nmid$ ldcf C, deg C = deg h(C) = $\deg(\Pi G) = \Sigma \deg G = \Sigma \deg F = \deg(\Pi F)$. Hence deg U = 0 and $U \equiv$ ldcf C (mod m).

b. Let $\overline{A} = h(A)$. Deg C = deg A + deg B implies

ldcf $C \equiv$ (ldcf A)(ldcf B)(mod m), hence the same congruence holds mod p. Thus $p \nmid$ ldcf C implies $p \nmid$ ldcf A, and hence deg $\overline{A}$ = deg A.

Since the G_i are prime, there exists a unique $G' \subset G$ such that $\overline{A}$ = (ldcf $\overline{A}$)$\Pi G'$. Suppose $G' = \{G_{i_1},\ldots,G_{i_r}\}$. Then define

$F' = \{F_{i_1},\ldots,F_{i_r}\}$,

A_1 = (ldcf A)$\Pi F'$,

B_1 = (ldcf B)Π(F-F').

From part a we have $A_1B_1 \equiv$ (ldcf A)(ldcf B)$\Pi F \equiv$ (ldcf C)$\Pi F \equiv C \equiv AB$ (mod m). We will show that the other assumptions of Theorem H are satisfied. First, $h(A_1)$ = h(ldcf A)h($\Pi F'$) = (ldcf $\overline{A}$)$\Pi\{h(F_i):F_i \in F'\}$ = (ldcf $\overline{A}$)$\Pi G'$ = $\overline{A}$ = h(A), hence $A_1 \equiv A$ (mod p). Similarly, $B_1 \equiv B$ (mod p). Next, deg A_1 = deg($\Pi F'$) = Σdeg F' = Σdeg G' = deg ($\Pi G'$) = deg $\overline{A}$ = deg A. Finally, $p \nmid$ ldcf A implies that ldcf A is a unit modulo p, since D/(p) is a field. Thus Theorem H implies that $A \equiv A_1$ (mod m), hence $A \equiv$ (ldcf A)$\Pi F'$ (mod m). The uniqueness of F' follows from that of G'.

We can now demonstrate a new application of Algorithm 2.6P. Let h_j be a homomorphism defined on D with kernel (p^j) and let $E_j = h_j(D)$. Under the assumptions of Algorithm C and the additional assumption $p \nmid$ ldcf C we obtain from Algorithm C a set $\overline{F}$ of polynomials such that $\overline{F}$ is a complete factorization of $h_j(C)$ over E_j. Assuming also that C is R-factorable for some suitable set of representatives of $D/(p^j)$ we can apply Algorithm 2.6P to C and G = $\overline{F}$ to obtain the complete factorization of C over D.

In the following two sub-sections we discuss in some detail two particular instances of this type of application of Algorithm 2.6P.

2.7.1. Application to factoring over Z

We have already given in Algorithm 2.6Z a method of factoring over Z which requires factoring over GF(p) for large primes p. Using the Hensel algorithms of Section 2.7, we can now present an algorithm which requires factoring over GF(p) only for relatively small primes.

Given a primitive polynomial over Z we may factor it into a product of squarefree polynomials using Algorithm 2.4S. We may therefore assume in the following algorithm that the input polynomial is squarefree.

Algorithm Z (Factoring over Z.) Given a non-constant, primitive, squarefree, positive polynomial C over Z, this algorithm finds the complete factorization F of C into prime positive polynomials over Z.

(1) Generate a new positive prime integer $p > 2$. (See discussion below).

(2) If $p \mid \text{ldcf } C$, go to (1).

(3) Set $\overline{C} \leftarrow h_p(C)$, $B \leftarrow \gcd(\overline{C},\overline{C}')$. If deg $B \neq 0$, go to (1). (We have deg $B = 0$ iff $\overline{C}$ is squarefree.)

(4) Factor $\overline{C}$, obtaining the set G of prime monic polynomials over GF(p) such that $\overline{C} = (\text{ldcf } \overline{C})\Pi G$.

(5) If order (G) = 1 (i.e. if $\overline{C}$ is prime) then set $F \leftarrow \{C\}$ and exit. (C is prime).

(6) Compute a bound μ on the coefficients of any factor of C. Also compute the smallest power $m = p^j > 2\mu(\text{ldcf } C)$.

(7) Apply Algorithm 2.7C (with D = Z, E = GF(p), $h = h_p$) to inputs

m, C, G; let F_1 be the output. (Thus F_1 is a set of monic polynomials over Z such that $C \equiv (\text{ldcf } C)\prod F_1 \pmod m$.)

(8) Apply Algorithm 2.6P (with D = Z; E = Z/(m); h the canonical homomorphism of D onto E, $R = \{n \in Z : |n| < m/2\}$ and D_0 = non-negative integers) to C and F_1, (regarding the members of F_1 as polynomials over Z/(m)), obtaining the set F of prime positive polynomials over Z such that $C = \prod F$. Exit.

In step (1) we assume that we have an algorithm for generating arbitrarily many primes in, say, the natural sequence 3,5,7,11,13,.... (We start with 3 since the use of 2 would complicate the definition of the set of representatives in step (8)).

Of course, only finitely many primes can divide ldcf C, but it is not so obvious that $h_p(C)$ can fail to be squarefree for only finitely many primes. The proof of this is given below as a corollary to a theorem about resultants.

Let

$$\begin{aligned} A(x) &= a_m x^m + \ldots + a_1 x + a_0, \; a_m \neq 0, \; m > 0, \\ B(x) &= b_n x^n + \ldots + b_1 x + b_0, \; b_n \neq 0, \; n > 0, \end{aligned} \tag{1}$$

be polynomials over D, a UFD. The <u>resultant</u> of A and B is the determinant

$$\text{res}(A,B) = \begin{vmatrix} a_m & a_{m-1} & \cdots & a_0 & 0 & \cdots & 0 \\ 0 & a_m & a_{m-1} & \cdots & a_0 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ b_n & b_{n-1} & \cdots & b_0 & 0 & \cdots & 0 \\ 0 & b_n & b_{n-1} & \cdots & b_0 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{vmatrix},$$

which has n rows containing the coefficients of A and m rows containing the coefficients of B.

The basic theorem about the resultant is:

$$\deg(\gcd(A,B)) = 0 \text{ iff } \operatorname{res}(A,B) \neq 0 \qquad (2)$$

For a proof, see [VDW49], Section 27.

Theorem G. Let D and E be UFDs and $h:D \to E$ be a homomorphism. Let $A,B \in D[x]$ be given as in (1) and suppose that $\deg(\gcd(A,B)) = 0$, $h(a_m) \neq 0$, $h(b_n) \neq 0$, and $h(\operatorname{res}(A,B)) \neq 0$. Then $\deg(\gcd(h(A),h(B))) = 0$.

Proof: We have $\deg h(A) = \deg A = m$, $\deg h(B) = \deg B = n$. Thus both $\operatorname{res}(A,B)$ and $\operatorname{res}(h(A),h(B))$ are determinants of order $m+n$, and since h is a homomorphism,

$$h(\operatorname{res}(A,B)) = \operatorname{res}(h(A),h(B)).$$

Hence $\operatorname{res}(h(A),h(B)) \neq 0$, which by (2), is equivalent to the conclusion of the theorem.

Corollary. Let C be a non-constant squarefree polynomial over Z. Then C is squarefree mod p for all but a finite number of primes p.

Proof: Since C is squarefree and the characteristic of Z is zero, $\deg(\gcd(C,C')) = 0$. Hence $\operatorname{res}(C,C') \neq 0$. At most finitely many primes divide ldcf C, ldcf C' or $\operatorname{res}(C,C')$. For all other primes p we can apply the theorem and conclude that C and C' are relatively prime modulo p, hence that C is squarefree modulo p.

2.7.2. Application to factoring in GF(p)[w,x]

In this section we consider the application of the Hensel algorithms to factorization of bivariate polynomials over GF(p). In Section 2.6.4 we gave an algorithm (2.6B) which reduces this problem to that of factoring univariate polynomials over $GF(p^n)$ for some n. The algorithm to be discussed here will make the same reduction, but by application of the Hensel algorithms it is possible to reduce the value of n. Most often we will have n = 1; i.e. we will be able to factor a given polynomial in GF(p)[w,x] by factoring a polynomial in GF(p)[x].

Let $A \in GF(p)[w,x]$. In Algorithm 2.4V we showed that we could reduce the problem of factoring A to that of factoring polynomials $C \in GF(p)[w,x]$ which satisfy one of

$$\gcd(C,\partial C/\partial x) = 1, \tag{1}$$

$$\gcd(C,\partial C/\partial w) = 1. \tag{2}$$

If (1) holds, then the following algorithm can be applied directly; if only (2) holds then we have to write C as a polynomial in w with coefficients from GF(p)[x] and interchange the variable names before applying the algorithm.

Algorithm B (Factoring in GF(p)[w,x]). Let D = GF(p)[w], D_0={0} ∪ {monic polynomials in D}. Given a primitive D_0 polynomial C in D[x] = GF(p)[w,x] such that gcd(C,C') = 1, this algorithm finds the complete factorization F of C into prime D_0 polynomials over D.

(1) Generate a new prime monic polynomial $A \in GF(p)[w]$ (see discussion below). Set m ← deg A.

(2) If $A \mid \text{ldcf}C$, go to (1).

(3) Set $\overline{C} \leftarrow h_A(C)$, (where h_A is the map $C \to C \bmod A$) $B \leftarrow \gcd(\overline{C},\overline{C}')$. If deg $B \neq 0$, go to (1). ($\overline{C} \in GF(p^m)[x]$. We have deg $B = 0$ iff $\overline{C}$ is squarefree.)

(4) Factor $\overline{C}$, obtaining the set G of prime monic polynomials over $GF(p^m)$ such that $\overline{C} = (\text{ldcf } \overline{C})\Pi G$.

(5) If order(G) = 1 (i.e. if $\overline{C}$ is prime) then set $F \leftarrow \{C\}$ and exit. (C is prime).

(6) Set $\mu \leftarrow \max\{\deg C_i\}$, where $C_k, C_{k-1}, \ldots, C_0$ are the coefficients of C. Set $j \leftarrow \lfloor(\mu+\deg C_k + 1)/m\rfloor$, $M \leftarrow A^j$.

(7) Apply Algorithm 2.7C (with $D = GF(p)[w]$, $E = GF(p^m)$, $h = h_A$) to inputs M,C,G; let F_1 be the output. (Thus F_1 is a set of monic polynomials in GF(p)[w,x] such that $C \equiv (\text{ldcf } C)\Pi F_1 \pmod M$.

(8) Apply Algorithm 2.6P (with $D = GF(p)[w]$; $E = \{B \in GF(p)[w]:$ deg B < deg M$\}$ with arithmetic defined modulo M; $h = h_M$; D_0 as above, R = E) to C and F_1, obtaining the set F of prime D_0 polynomials over D such that $C = \Pi F$. Exit.

In step (1), we would naturally want to generated the prime polynomials $A \in GF(p)[w]$ in order of increasing degree. Unless p is small, the linear polynomials A(w) = w-a for $a \in GF(p)$ will usually suffice; but in some cases it will be necessary to generate quadratic or higher degree prime polynomials. One way to obtain all prime $A \in GF(p)[w]$ of degree d is to compute

$$B(w) = (w^{p^d}-w)/\Pi_{e \mid d, e \neq d}(w^{p^e}-w)$$

and factor B over GF(p); B is precisely the product of all prime A

of degree d. (See [BER68], p. 103.) This will be practical only if p and d are small; and even if they are, the computation would be lengthy enough that it would be desirable to perform it in advance and store a list of the prime polynomials found for use in Algorithm B.

Again it is necessary it show that $h_A(C)$ is squarefree for all but finitely many primes A. The proof is similar to that of the corollary to Theorem 2.7.1G: Since gcd(C,C') = 0, we have res(C,C') $\neq$ 0, by the basic theorem cited in Section 2.7.1. Now res(C,C') $\in$ GF(p)[w], and only finitely many prime A $\in$ GF(p)[w] can divide ldcf C, ldcf C' or res(C,C'). For other primes A we can apply Theorem 2.7.1G and conclude that $\deg(\gcd(h_A(C),h_A(C'))) = 0$, hence that $h_A(C)$ is squarefree.

2.8 Other Hensel algorithms and applications to multivariate factorization

In this section we first discuss a variation on Algorithm 2.7H (Hensel's algorithm) which was first proposed by Zassenhaus [ZAS69]. Given a factorization over a ring D modulo p, this algorithm computes factorizations modulo $p^2, p^4, p^8, \ldots$ in successive iterations. In the case D = Z, the algorithm turns out to be much more efficient than Algorithm 2.7H for factoring polynomials with large coefficients. The algorithm also has another, perhaps more important, virtue. It allows the development of a "generalized Hensel algorithm" (Algorithm G, below) which yields a practical method of factorization of multivariate polynomials.

The "quadratic Hensel's algorithm" which follows is based on a version discussed by Knuth ([KNU69], pp. 398 and 546.)

Algorithm Q (Quadratic Hensel Algorithm). Let D and E be commutative rings with identities, $p \in D$, and h a homomorphism from D onto E, with Ker h = (p). The inputs to the algorithm are:

$m = p^j$ for some positive integer j;

$C \in D[x]$;

$\overline{A}, \overline{B}, \overline{S}, \overline{T} \in E[x]$ such that ldcf $\overline{A}$ is a unit of E, $h(C) = \overline{A}\,\overline{B}$ and $\overline{A}\,\overline{S} + \overline{B}\,\overline{T} = 1$.

The outputs are $A, B, S, T \in D[x]$ such that ldcf A is a unit modulo m, $C \equiv AB$ and $AS + BT \equiv 1 \pmod{m}$, $h(A) = \overline{A}$, $h(B) = \overline{B}$ and deg A = deg $\overline{A}$.

(1) Set $q \leftarrow p$ and choose $A, B, S, T \in D[x]$ such that $h(A) = \overline{A}, \ldots, h(T) = \overline{T}$ and deg A = deg $\overline{A}$.

(2) (Now $A,B,S,T \in D[x]$, ldcf A is a unit mod q, $C \equiv AB$ and $AS+BT \equiv 1 \pmod{q}$, $h(A) = \overline{A}$, $h(B) = \overline{B}$ and deg A = deg $\overline{A}$.) If $m \mid q$, exit.

(3) Set $U \leftarrow (C-AB)/q$. (Since $C \equiv AB \pmod{q}$ we know $U \in D[x]$.) Using Algorithm 2.7S with inputs A,B,S,T,U, solve the congruence $AY+BZ \equiv U \pmod{q}$ for $Y,Z \in D[x]$ such that deg Z < deg A.

(4) Set $A^* \leftarrow A+qZ$, $B^* \leftarrow B+qY$. (Thus

$$\begin{aligned} C-A^*B^* &= C-AB-q(AY+BZ)-q^2YZ \\ &= q(U-AY-BZ)-q^2YZ \\ &\equiv 0 \pmod{q^2}; \end{aligned}$$

furthermore $h(A^*) = h(A)$, $h(B^*) = h(B)$; and, since deg Z < deg A, deg A^* = deg A = deg $\overline{A}$ and ldcf A^* = ldcf A. By Lemma 2.7U, ldcf A^* is a unit modulo q^2.)

(5) Set $U_1 \leftarrow (A^*S+B^*T-1)/q$. Using Algorithm 2.7S with inputs A,B,S,T,U_1, solve the congruence $AY_1+BZ_1 \equiv U \pmod{q}$ for $Y_1,Z_1 \in D[x]$ such that deg Z_1 < deg A.

(6) Set $S^* \leftarrow S-qY_1$, $T^* \leftarrow T-qZ_1$. (Thus

$$\begin{aligned} A^*S^*+B^*T^* &= A^*(S-qY_1)+B^*(T-qZ_1) \\ &= A^*S+B^*T-q(A^*Y_1+B^*Z_1) \\ &= 1 + q(U_1-A^*Y_1-B^*Z_1) \\ &\equiv 1 + q(U_1-AY_1-BZ_1) \pmod{q^2} \\ &\equiv 1 \pmod{q^2}.) \end{aligned}$$

(7) Replace q,A,B,S,T by q^2,A^*,B^*,S^*,T^* and go to (2).

The most immediate application of this algorithm is to substitute it for Algorithm 2.7H in Algorithm 2.7C. Note that this

does not affect the validity of Theorem 2.7C. This substitution presents no advantage at the level of abstract algorithms, but in the applications of Algorithm 2.7C to factoring in Z[x] and GF(p)[w,x], an important gain in computing time efficiency is obtained. This point will be discussed for the case of factoring over Z in Section 3.5.

The remainder of this section will be devoted to developing abstract algorithms which can be applied to factorization of multivariate polynomials. This is achieved by developing generalizations of Algorithms Q and 2.7C and Theorems 2.7H and 2.7C in which the kernel of the homomorphism h of D onto E may be generated by more than one element. We shall then be able to take, for example, D = Z[v,w] E = GF(p) and h to be the homomorphism with kernel (p,v,w), so that from $C \in Z[v,w,x]$ and a factorization $h(C) = \overline{AB}$ over GF(p) we can construct a factorization $C \equiv AB \pmod{p^i,v^j,w^k}$ for any positive integers i,j,k. Such a factorization may then be used as input to Algorithm 2.6P to obtain the complete factorization of C in Z[v,w,x].

We begin with a rather general lemma about homomorphisms.

Lemma 1. Let R,S,T be commutative rings, ν be a homomorphism of R onto S and α be a homomorphism of R onto T. Let Ker $\nu \subset$ Ker α. Then:

a. there exists a (unique) homomorphism β from S onto T such that $\beta \circ \nu = \alpha$

b. Ker $\beta = \nu$(Ker α).

Proof: This lemma could be derived as a corollary to the so-called "Rectangle Theorem" for rings (see [GOL70], p. 120). We shall, however, give a direct proof.

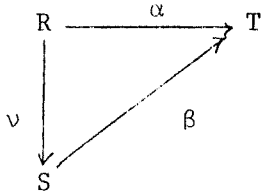
a. For $\nu(r) \in S$, define $\beta(\nu(r)) = \alpha(r)$. We first have to show that this defines a mapping of S onto T. Let $r_1, r_2 \in R$ such that $\nu(r_1) = \nu(r_2)$. Thus $\nu(r_1-r_2) = 0$, hence $r_1-r_2 \in \text{Ker } \nu$, hence $r_1-r_2 \in \text{Ker } \alpha$, hence $\alpha(r_1) = \alpha(r_2)$. β is thus a well-defined mapping. It is onto: for if $t \in T$ there is an $r \in S$ such that $\alpha(r) = t$, hence $\beta(\nu(r)) = t$. It is a homomorphism, for

$$\beta(\nu(r_1) + \nu(r_2)) = \beta(\nu(r_1+r_2)) = \alpha(r_1+r_2)$$
$$= \alpha(r_1) + \alpha(r_2) = \beta(\nu(r_1)) + \beta(\nu(r_2))$$

and similarly for multiplication. If β_1 also satisfies $\beta_1 \circ \nu = \alpha$ then $\beta_1(\nu(r)) = \alpha(r) = \beta(\nu(r))$ for all $\nu(r) \in S$; thus β is unique.

b. $\nu(r) \in \text{Ker } \beta \Leftrightarrow \beta(\nu(r)) = 0 \Leftrightarrow \alpha(r) = 0 \Leftrightarrow r \in \text{Ker } \alpha$

$\nu(r) \in \nu(\text{Ker } \alpha)$.



Algorithm G (Generalized Hensel Algorithm). Let D and E be commutative rings with identities, $p_1,\ldots,p_n$ be elements of D and h be a homomorphism from D onto E with kernel $\mathcal{P} = (p_1,\ldots,p_n)$. The inputs to this algorithm are:

$m_1 = p_1^{j_1},\ldots,m_n = p_n^{j_n}$ for some positive integers $j_1,\ldots,j_n$;

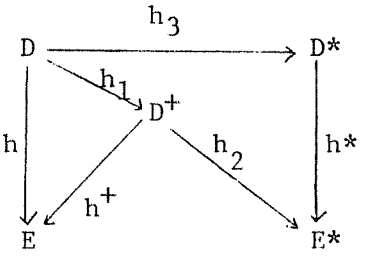
$C \in D[x]$;

$\overline{A},\overline{B},\overline{S},\overline{T} \in E[x]$ such that ldcf $\overline{A}$ is a unit of E,

$h(C) = \overline{AB}$ and $\overline{AS}+\overline{BT} = 1$

The outputs are $A,B,S,T \in D[x]$ such that ldcf A is a unit modulo

$\mathfrak{m} = (m_1,\ldots,m_n)$, $C \equiv AB$ and $AS+BT \equiv 1 \pmod{\mathfrak{m}}$, $h(A) = \overline{A}$, $h(B) = \overline{B}$, and deg A = deg $\overline{A}$.



Remark: The above diagram will aid in following the statement and proof of the algorithm. (The definitions and proofs given in the algorithm will show that the diagram commutes.)

(1) If $n = 1$, apply Algorithm Q to $m_1, C, \overline{A}, \overline{B}, \overline{S}, \overline{T}$, obtaining $A,B,S,T \in D[x]$ satisfying the required conditions. Exit.

(2) Let h_1 be a homomorphism defined on D with kernel (p_1) and let $D^+ = h_1(D)$. Let h^+ be the homomorphism of D^+ onto E such that $h^+ \circ h_1 = h$. (The existence of h^+ is guaranteed by Lemma 1; also Ker $h^+ = h_1(\text{Ker } h) = h_1((p_1,\ldots,p_n)) = (h_1(p_1),\ldots,h_1(p_n)) = (0,h_1(p_2),\ldots,h_1(p_n)) = (h_1(p_2),\ldots,h_1(p_n))$.) Set $m_2^+ \leftarrow h_1(m_2)$ $(= h_1(p_2)^{j2}),\ldots,m_n^+ \leftarrow h_1(m_n)$, $C^+ \leftarrow h_1(C)$. Working in D^+ and E, apply this algorithm recursively to $m_2^+,\ldots,m_n^+,C^+,\overline{A},\overline{B},\overline{S},\overline{T}$, obtaining outputs $A^+,B^+,S^+,T^+ \in D^+[x]$. (Thus ldcf A^+ is a unit mod $\mathfrak{m}^+ = (m_2^+,\ldots,m_n^+)$, $C^+ \equiv A^+B^+$ and $A^+S^+ + B^+T^+ \equiv 1 \pmod{\mathfrak{m}^+}$, $h^+(A^+) = \overline{A}$, $h^+(B^+) = \overline{B}$ and deg A^+ = deg $\overline{A}$.)

(3) Let h_2 be a homomorphism defined on D^+ with kernel $(m_2^+,\ldots,m_n^+)$ and let $E^* = h_2(D^+)$. Set $\overline{A}^* \leftarrow h_2(A^+)$, $\overline{B}^* \leftarrow h_2(B^+)$, $\overline{S}^* \leftarrow h_2(S^+)$, $\overline{T}^* \leftarrow h_2(T^+)$. (Thus ldcf $\overline{A}^*$ is a unit of E^*, deg $\overline{A}^*$ = deg A^+, $h_2(h_1(C)) = \overline{A}^*\overline{B}^*$ and $\overline{A}^*\overline{S}^* + \overline{B}^*\overline{T}^* = 1$).

(4) Let h_3 be a homomorphism defined on D with kernel $(m_2,\ldots,m_n)$ and let $D^* = h_3(D)$. Let h^* be the homomorphism of D^* onto E^* such that $h^* \circ h_3 = h_2 \circ h_1$. (Ker $h_2 \circ h_1 = (p_1,m_2,\ldots,m_n)$, as will be shown below, hence Lemma 1 guarantees the existence of h^* and furthermore shows that Ker $h^* = h_3(\text{Ker } h_2 \circ h_1) = h_3((p_1,m_2,\ldots,m_n)) = (h_3(p_1))$.) Set $m_1^* \leftarrow h_3(m_1)$, $C^* \leftarrow h_3(C)$. (Thus $h^*(C^*) = h^*(h_3(C)) = h_2(h_1(C)) = \overline{A}^*\overline{B}^*$.) Working in D^* and E^*, apply Algorithm Q to $m_1^*, C^*, \overline{A}^*, \overline{B}^*, \overline{S}^*, \overline{T}^*$, to obtain outputs $A^*,B^*,S^*,T^* \in D^*[x]$. (Thus ldcf A^* is a unit modulo m_1^*, $C^* \equiv A^*B^*$ and $A^*S^*+B^*T^* \equiv 1 \pmod{m_1^*}$, $h^*(A^*) = \overline{A}^*$, $h^*(B^*) = \overline{B}^*$ and deg A^* = deg $\overline{A}^*$.)

(5) Choose $A,B,S,T \in D[x]$ such that $h_3(A) = A^*$, $h_3(B) = B^*$, $h_3(S) = S^*$, $h_3(T) = T^*$ and deg A = deg A^*. (Then $h_3(C) = h_3(A)h_3(B) \pmod{h_3(m_1)}$, hence $h_3(C-AB) = h_3(P)h_3(m_1)$ for some $P \in D[x]$, hence $h_3(C-AB-Pm_1) = 0$, hence $C-AB-Pm_1 \in (m_2,\ldots,m_n)$, hence $C-AB \in (m_1,\ldots,m_n)$, hence $C \equiv AB \pmod{(m_1,\ldots,m_n)}$. Similarly, $AS+BT \equiv 1 \pmod{(m_1,\ldots,m_n)}$. Since $h_2(h_1(A)) = h^*(h_3(A)) = h^*(A^*) = \overline{A}^* = h_2(A^+)$, we have $h_1(A) \equiv A^+ \pmod{\mathfrak{m}^+}$. It follows that $h_1(A) \equiv A^+ \pmod{(h_1(p_2),\ldots,h_1(p_n))}$; i.e. $h_1(A) - A^+ \in \text{Ker } h^+$, hence $h^+(h_1(A)) = h^+(A^+) = \overline{A}$, and finally $h(A) = \overline{A}$. Similarly, $h(B) = \overline{B}$. Also deg A = deg A^* = deg $\overline{A}^*$ = deg A^+ = deg $\overline{A}$. Lastly, we have $h_3(\text{ldcf } A) = \text{ldcf } A^*$, from which it is easily shown that ldcf A is a unit modulo $(m_1,\ldots,m_n)$.)

The assertion in step (4) that Ker $h_2 \circ h_1 = (p_1,m_2,\ldots,m_n)$ may be proved as follows:

$$d \in \text{Ker } h_2 \circ h_1 \in h_1(d) \in \text{Ker } h_2 = (m_2^+,\ldots,m_n^+)$$

$$\Leftrightarrow \quad h_1(d) = h_1(d_2)h_1(m_2) + \ldots + h_1(d_n)h_1(m_n) \text{ for some } d_2,\ldots,d_n \in D$$

$\Leftrightarrow$ $h_1(d-d_2m_2-\ldots-d_nm_n) = 0$ for some $d_2,\ldots,d_n \in D$

$\Leftrightarrow$ $d-d_2m_2-\ldots-d_nm_n \in \text{Ker } h_1 = (p_1)$ for some $d_2,\ldots,d_n \in D$

$\Leftrightarrow$ $d-d_2m_2-\ldots-d_nm_n = d_1p_1$ for some $d_1,\ldots,d_n \in D$

$\Leftrightarrow$ $d \in (p_1,m_2,\ldots,m_n)$.

Before proceeding to generalize Theorem 2.7H, we shall prove a lemma which generalizes Lemma 1 of Section 2.7.

Lemma 2. Let D be a commutative ring with identity, $p_1,\ldots,p_n \in D$, $m_1 = p_1^{j_1},\ldots,m_n = p_n^{j_n}$ for some positive integers $j_1,\ldots,j_n$, $\mathfrak{p} = (p_1,\ldots,p_n)$, $\mathfrak{m} = (m_1,\ldots,m_n)$. Let $a \in D$ be a unit modulo $\mathfrak{p}$. Then a is a unit modulo $\mathfrak{m}$.

Proof: We use the notation of Algorithm G and divide the proof into steps corresponding to those of Algorithm G:

(1) If $n = 1$ then Lemma 1 of Section 2.7 applies and we are done.

(2) Assume $n > 1$ and let $a^+ = h_1(a)$. Since a is a unit modulo $\mathfrak{p}$, there is a $b \in D$ such that $ab-1 \in \mathfrak{p}$, hence $a^+b^+-1 \in \mathfrak{p}^+$, where $b^+ = h_1(b)$ and $\mathfrak{p}^+ = (h_1(p_2),\ldots,h_1(p_n))$. Hence a^+ is a unit modulo $\mathfrak{p}^+$, and we may thus assume, by induction on n, that a^+ is a unit modulo $\mathfrak{m}^+$.

(3) Let $\bar{a}^* = h_2(a^+)$. From the conclusion of step (2), $\bar{a}^*$ is a unit of E^*.

(4) Let $a^* = h_3(a)$. Then $h^*(a^*) = \bar{a}^*$, i.e. a^* is a unit modulo p_1^*. Applying Lemma 1 of Section 2.7, we find that a^* is a unit modulo m_1^*.

(5) Suppose $a^*b^* \equiv 1 \pmod{m_1^*}$. Choose $b \in D$ such that $h_3(b) = b^*$. Then $h_3(a)h_3(b)-1 = h_3(d)h_3(m_1)$ for some $d_1 \in D$, hence $h_3(ab-1-dm_1) = 0$, hence $ab-1-d_1m_1 \in (m_2,m_3,\ldots,m_n)$, hence $ab \equiv 1 \pmod{\mathfrak{m}}$.

Theorem G. Let D be a commutative ring with identity, $p_1,\ldots,p_n \in D-D_Z$, $m_1 = p_1^{j_1},\ldots,m_n = p_n^{j_n}$ for some positive integers

$j_1,\ldots,j_n$, $\mathfrak{p} = (p_1,\ldots,p_n)$, $\mathfrak{m} = (m_1,\ldots,m_n)$. Let $A,B,A_1,B_1 \in D[x]$ satisfy

a. $A_1B_1 \equiv AB \pmod{\mathfrak{m}}$;

b. $\deg A_1 = \deg A$ and $\operatorname{ldcf} A_1 \equiv \operatorname{ldcf} A \pmod{\mathfrak{m}}$;

c. $A_1 \equiv A$ and $B_1 \equiv B \pmod{\mathfrak{p}}$;

d. $\operatorname{ldcf} A$ is a unit modulo $\mathfrak{p}$.

Then $A_1 \equiv A$ and $B_1 \equiv B \pmod{\mathfrak{m}}$.

Proof: Again we use the notation of Algorithm G and divide the proof into steps corresponding to those of Algorithm G:

(1) If $n = 1$, Theorem 2.7H applies and we are done.

(2) Assume $n > 1$. Let A_1^+,B_1^+,A^+,B^+ be the images under h_1 of A_1,B_1,A,B. Let $\mathfrak{p}^+ = (h_1(p_2),\ldots,h_1(p_n))$. By c, $A_1-A = d_1p_1+\ldots+d_np_n$ for some $d_i \in D[x]$. Hence $h_1(A_1)-h_1(A) = h_1(d_2)h_1(p_2)+\ldots+h_1(d_n)h_1(p_n)$, hence $A_1^+-A^+ \in \mathfrak{p}^+$. In this way we can prove

c^+. $A_1^+ \equiv A^+$ and $B_1^+ \equiv B^+ \pmod{\mathfrak{p}^+}$,

d^+. $\operatorname{ldcf} A^+$ is a unit modulo $\mathfrak{p}^+$,

a^+. $A_1^+B_1^+ \equiv A^+B^+ \pmod{\mathfrak{m}^+}$,

and, using Lemma 2, the second part of

b^+. $\deg A_1^+ = \deg A^+$ and $\operatorname{ldcf} A_1^+ \equiv \operatorname{ldcf} A^+ \pmod{\mathfrak{m}^+}$.

The first part of b^+ follows from b and d. We may now assume, by induction on n, that $A_1^+ \equiv A^+$ and $B_1^+ \equiv B^+ \pmod{\mathfrak{m}^+}$.

(3) Let $\overline{A}_1^*,\overline{B}_1^*,\overline{A}^*,\overline{B}^*$ be the images under h_2 of A_1^+,B_1^+,A^+,B^+. Then, from the conclusion of step (2), $\overline{A}_1^* = \overline{A}^*$ and $\overline{B}_1^* = \overline{B}^*$.

(4) Let A_1^*,B_1^*,A^*,B^* be the images under h_3 of A_1,B_1,A,B. From the equation $h_2 \circ h_1 = h^* \circ h_3$, we have

$$h^*(A_1^*) = h^*(h_3(A_1)) = h_2(h_1(A_1)) = \overline{A}_1^* = \overline{A}^*$$
$$= h_2(h_1(A)) = h^*(h_3(A)) = h^*(A^*).$$

In this manner we obtain

c*. $A_1^* \equiv A^* \pmod{\quad}$ and $B_1^* \equiv B^* \pmod{p_1^*}$.

From a,b,c and the definition of h_3 and m_1^*, we have

a*. $A_1^* B_1^* \equiv A^*B^* \pmod{m_1^*}$

b*. $\deg A_1^* = \deg A$ and $\operatorname{ldcf} A_1^* \equiv \operatorname{ldcf} A^* \pmod{m_1^*}$.

We may prove

1*. $\operatorname{ldcf} A^*$ is a unit modulo p_1^*

as follows: From d^+ and Lemma 2, we know that $\operatorname{ldcf} A^+$ is a unit modulo $\mathfrak{m}^+$. Furthermore, $\operatorname{ldcf} \overline{A}^* = h_2(\operatorname{ldcf} A^+)$, hence $\operatorname{ldcf} \overline{A}^*$ is a unit of E*. Finally, $h^*(\operatorname{ldcf} A^*) = \operatorname{ldcf} \overline{A}^*$, hence d* follows. Now Theorem 2.7H implies $A_1^* \equiv A^*$ and $B_1^* \equiv B \pmod{m_1^*}$.

(5) From the conclusion of step (4), $h_3(A_1-A_1)$ is a multiple of $h_3(m_1)$, say $h_3(d_1)$. Thus $h_3(A_1-A-dm_1) = 0$, hence $A_1-A-d_1m_1 = d_2m_2+\ldots+d_nm_n$ for some $d_2,\ldots,d_n \in D[x]$. Hence $A_1 \equiv A \pmod{\mathfrak{m}}$ and similarly, $B_1 \equiv B \pmod{\mathfrak{m}}$.

In Algorithm G and Theorem G we have obtained generalizations of Algorithm 2.7H and Theorem 2.7H, replacing "p" by "$p_1,p_2,\ldots,p_n$", "(p)" by "$\mathfrak{p}$", "m" by "$m_1(=p_1^{j_1}), m_2,\ldots,m_n$" and "(m)" by "$\mathfrak{m}$". We can continue along this line obtaining generalizations of Algorithm 2.7C and Theorem 2.7C by making the same substitution in their statement. Now, however, the situation becomes much easier, for, as the reader may verify, the proofs may be obtained merely by making the same substitution in the proofs of Algorithm 2.7C and Theorem 2.7C.

CHAPTER 3: FACTORING UNIVARIATE POLYNOMIALS OVER THE INTEGERS

3.1 Introduction

In this chapter we shall give detailed specifications of algorithms for factoring univariate polynomials over the integers. These algorithms are based on the use of factorizations over GF(p) and the Quadratic Hensel Algorithm described in Section 2.8. The basic structure of each of the major algorithms described here corresponds closely to that of one of the algorithms described in Chapter 2. However, important improvements have been made in each case. For example, Algorithm PFZ1, which is based on Algorithm 2.7Z, performs factorizations over GF(p) for several primes p and chooses the prime which yields the fewest irreducible factors, instead of just taking the first prime for which the input polynomial is of the same degree and squarefree over GF(p).

These algorithms have been implemented in the SAC-1 system and our descriptions of them applies, in regard to the details of input and output, to this implementation. The notation and conventions used in stating the algorithms are essentially the same as those discussed in Section 1.3 and used in Chapter 2. We have tried to avoid inclusion of details which are dependent on the idiosyncrasies of Fortran or the SAC-1 system. We have omitted, for example, details regarding the erasure of lists which are created in the algorithms, whereas explicit erasure is required in SAC-1 since a "reference count" list processing system is used (see [COL67]). Although such details

must be attended to when actually programming in a system such as SAC-1, they would only detract from the reader's understanding of the basic structure of the algorithms. For a completely precise statement of any step of an algorithm the reader can easily refer to the Fortran program listings in the appendix, since we have maintained a close correspondence between the descriptions in this chapter and the actual programs, with regard to variable naming and step numbering.

For some of the simpler algorithms we give only a description of the input and output. Again, the reader may wish to refer to the Fortran listings in these cases.

Computing time bounds are included for each of the algorithms, accompanied by their derivations except in the simplest cases. In several instances we note how consideration of computing time bounds has determined our choice of a particular algorithm to implement.

References are made, where appropriate, to the algorithms for operations on polynomials in SAC-1. For example, the SAC-1 algorithm CPBERL, which implements Berlekamp's algorithm for factoring over GF(p), is discussed briefly in Section 3.5.4.

3.2 Modular arithmetic

The SAC-1 Modular Arithmetic System [COL69] can perform arithmetic on polynomials modulo any positive integer m which is a Fortran integer (single-precision integer). In the factoring algorithms to be described in Section 3.5 we have need of some operations modulo an integer m which can be much larger than the bound on Fortran integers. Thus in this section we describe algorithms for these operations which take as input an L-integer modulus. ("L-integer" abbreviates "list-integer", reflecting the fact that large integers are represented in SAC-1 by lists of digits.)

Besides the L-integer modulus m, which is assumed to be odd and > 1, the inputs and outputs for each algorithm are other L-integers or univariate polynomials with integer coefficients. These polynomials are assumed to have the same list representation as in the SAC-1 Polynomial Arithmetic System [COL68b]. The inputs and outputs have, however, the additional property that their coefficients are bounded by m/2 (except, of course, for the inputs to the algorithms which perform a reduction modulo m.) The choice of integers $-\lfloor m/2 \rfloor, \ldots, 0, \ldots, \lfloor m/2 \rfloor$ as a set of representatives of $Z/(m)$, rather than $0,1,\ldots,m-1$, simplifies some of the factoring algorithms (see Section 2.7.1).

Algorithm MMOD(m,a) (Modular algorithm, reduction mod m). The inputs are L-integers m and a, where m is odd and > 1. The output is the unique L-integer b such that $b \equiv a \pmod{m}$ and $|b| < m/2$. Computing time: $\preceq L(m)L(a)$.

Algorithm MPMOD(m,A) (Modular algorithm, reduction of a polynomial mod m). m is an L-integer which is odd and > 1, and A is an L-integer or a univariate polynomial with L-integer coefficients. The output is the unique polynomial B with L-integer coefficients bounded by m/2 and satisfying $B \equiv A \pmod m$. (B is an L-integer if A is.)

Computing time: $\preceq (1+\deg A)L(m)L(|A|_\infty)$.

Algorithm MPLPR(m,A) (Modular polynomial list product). The inputs are m, an odd L-integer > 1, and a list $A = (A_1,\ldots,A_r)$ of polynomials over Z with $|A_i|_\infty < m/2$ for each i. The output is a polynomial $B \equiv \Pi A \pmod m$ with $|B|_\infty < m/2$.

(1) If A is empty set $B \leftarrow 1$ and exit; otherwise, set $B \leftarrow$ first(A).

(2) Set $A \leftarrow$ tail(A). If A is empty, exit.

(3) Set $B \leftarrow$ MPMOD(m,B·first(A)) and go to (2).

Theorem MPLPR(m,A). Assume $r > 0$, $n_i = \deg A_i$, $n = n_1+\ldots+n_r$. Then the computing time for MPLPR(m,A) is $\preceq (n+1)(n+r)L(m)^2$.

Proof: When r = 1, the time is ~ 1. Assume r > 1. When first(A) = A_i the time for B · first(A) is $\preceq (n_1 +\ldots+ n_{i-1}+ 1)(n_i+1)L(m)^2$ and for MPMOD is $\preceq (n_1 +\ldots+ n_i+1)L(m)^2 \preceq (n_1 +\ldots+ n_{i-1}+1)(n_i+1)L(m)^2$. The total time is thus

$$\preceq \sum_{i=2}^{r} (n_1 +\ldots+ n_{i-1}+1)(n_i+1)L(m)^2$$

$$\preceq (n+1)L(m)^2 \sum_{i=2}^{r} (n_i+1) \preceq (n+1)L(m)^2(n+r).$$

Remarks: If $n_i > 0$ for all i then $n \geq r$, hence the time is $\preceq (n+1)^2L(m)^2$. If $n_i = 0$ for all i then the time is $\preceq rL(m)^2$.

The algorithm given here is superior to the alternative of applying MPMOD only once to the product $A_1 \cdot A_2 \cdots A_r$, which has a total computing time $\preceq r(n+1)(n+r)L(m)^2$.

Algorithm MRECIP(m,a) (Modular algorithm, reciprocal of a mod m). The inputs are an odd L-integer $m > 1$ and an L-integer a bounded by m/2 and relatively prime to m. The output is an L-integer b bounded by m/2 and satisfying $ab \equiv 1 \pmod{m}$.

The Extended Euclidean Algorithm (2.2E) is used.

Computing time: $\preceq L(m)^2$.

Algorithm MPQREM(m,A,B) (Modular polynomial quotient and remainder). The inputs are an odd L-integer $m > 1$ and univariate polynomials A and B with L-integer coefficients bounded by m/2; the leading coefficient of B must be relatively prime to m. The output is the list L = (Q,R) where Q and R are the quotient and remainder obtained upon dividing A by B using arithmetic modulo m; the coefficients of Q and R are bounded by m/2.

Algorithm 1.3D is used.

Computing time: $\preceq (k+1)(h-k+1)\ L(m)^2$ where h = deg A, k = deg B, provided $h \geq k$.

The following algorithm is based on Algorithm 2.7S.

Algorithm MPSPEQ(m,A,B,S,T,U,Y,Z) (Modular polynomial solution of a polynomial equation). The inputs are:

m, an odd L-integer > 1;

A,B,S,T, univariate polynomials with L-integer coefficients bounded by m/2, satisfying $AS+BT \equiv 1 \pmod{m}$, with ldcf A relatively

prime to m;

U, a univariate polynomial with L-integer coefficients.

The outputs are Y and Z, univariate polynomials with L-integer coefficients bounded by m/2 such that $AY+BZ \equiv U \pmod m$ and $\deg Z < \deg A$.

(1) Set $W \leftarrow$ MPMOD(m,U), $V \leftarrow$ MPMOD(m,TW).

(2) Using Algorithm MPQREM, compute $Q,Z \in Z[x]$ such that

$$V \equiv AQ+Z \pmod m, \quad \deg Z < \deg A,$$

$$|Q|_\infty < m/2, \quad |Z|_\infty < m/2.$$

(3) Set $Y \leftarrow$ MPMOD(m,SW+BQ).

Theorem MPSPEQ. Assume $h = \deg A > 0$, $k = \deg B > 0$, $n = \max(h+k, \deg U)$, $\deg T < h$, $\deg S < k$, $u = |U|_1$ and $L(n) \sim 1$. Then the computing time for Algorithm MPSPEQ is

$$\preceq n^2 L(m)^2 + n\, L(m)L(u).$$

Proof: In step (1) the time for MPMOD(m,U) is $\preceq n\, L(m)L(u)$ and for T·W is $\preceq hn\, L(m)^2$. Since $|TW|_1 \le |T|_1|W|_1 \le h(n+1)m^2$, the time for MPMOD(m,TW) is $\preceq (h+n)\, L(m)\, L(h(n+1)m^2) \preceq n\, L(m)^2$ (since $L(hn) \preceq L(n^2) \sim 1$.)

Since $\deg V \le \deg T + \deg W < h + n$, the time for MPQREM(m,V,A) is $\preceq (\deg A + 1)(\deg V - \deg A + 1)\, L(m)^2 \preceq (h+1)\, n\, L(m)^2 \preceq n^2\, L(m)^2$.

The time for computing both S·W and B·Q is $\preceq kn\, L(m)^2$. Since $|SW|_1$ and $|BQ|_1$ are both $\le (k+1)(n+1)\, m^2$, the time to compute SW+BQ is $\preceq (k+n)\, L((k+1)(n+1)m^2) \preceq n\, L(m)$. $|SW+BQ|_1 \le 2(k+1)(n+1)m^2$, hence the time for MPMOD(m,SW+BQ) is $\preceq (k+n)L(m)\, L(2(k+1)nm^2) \preceq n\, L(m)^2$.

We thus see that the dominant times are those stated in the theorem.

3.3 Set operations

This section describes SAC-1 implementations of the set operation algorithms discussed in Section 1.6. Of main interest here is the subprogram for generation of sum index sets, but for completeness we also include descriptions of the more basic operations.

Since the binary representation of an integer is unique, there is a one-to-one correspondence between non-negative integers and finite sets of non-negative integers, defined by

$$i \leftrightarrow J \text{ iff } i = 2^{j_1}+\ldots+2^{j_n} \text{ and } J = \{j_1,\ldots,j_n\}.$$

We shall write $J = \mathcal{S}(i)$ and $i = \mathcal{J}(J)$ when $i \leftrightarrow J$.

If i is a non-negative integer, let

$$i = \sum_{r\geq 0} b_r^{(i)} 2^r,\ b_r^{(i)} \in \{0,1\}$$

denote its unique binary representation. We define operations "$\wedge$" (logical product) and "$\vee$" (logical sum) on "bits" (binary digits) in the usual way, and on non-negative integers as follows:

$$i \wedge j = \sum_{r\geq 0}(b_r^{(i)} \wedge b_r^{(j)})2^r;\ i \vee j = \sum_{r\geq 0}(b_r^{(i)} \vee b_r^{(j)})2^r.$$

Thus, for example,

$$10 = (1010)_2,$$

$$12 = (1100)_2,$$

$$10 \wedge 12 = (1000)_2 = 8,$$

$$10 \vee 12 = (1110)_2 = 14.$$

With these definitions the following identities obviously hold:

$$\mathcal{S}(i \wedge j) = \mathcal{S}(i) \cap \mathcal{S}(j),$$

$$\mathcal{S}(i \vee j) = \mathcal{S}(i) \cup \mathcal{S}(j),$$

$$\mathcal{S}(2^n i) = \mathcal{S}(i) + \{n\}.$$

Thus the computation of $S \cup (S + \{n\})$ required in the sumset algorithm of Section 1.6 may be performed by computing $i \vee (i \cdot 2^n)$, where $i \leftrightarrow S$.

The following algorithm computes $k = i \wedge j$ from nonnegative integers i and j.

(1) Set $k \leftarrow 0$, $m \leftarrow i$, $n \leftarrow j$, $r \leftarrow 1$.

(2) If $m = 0$ or $n = 0$, exit.

(3) Set $p \leftarrow m \bmod 2$, $m \leftarrow \lfloor m/2 \rfloor$,

$q \leftarrow n \bmod 2$, $n \leftarrow \lfloor n/2 \rfloor$,

$$t \leftarrow \begin{cases} 1 & \text{if } p = 1 \text{ and } q = 1, \\ 0 & \text{otherwise,} \end{cases}$$

$k \leftarrow rt+k$, $r \leftarrow 2r$, and go to (2).

This algorithm simply determines the bits in the binary representations of i and j and constructs k accordingly. If the arithmetic indicated is performed using the SAC-1 Infinite Precision Integer Arithmetic System operations, then i and j may be of arbitrary size.

Most binary computers have hardware logical operations $\wedge$ and $\vee$ for single precision integers, and on such machines the above algorithm can be made more efficient: Let β be the base used in SAC-1 system (a positive integer whose base β representation is $d_n\beta^n + d_{n-1}\beta^{n-1} + \ldots + d_o$ is represented by the list $(d_o, d_1, \ldots, d_n)$; zero is represented by the empty list), and let $\ell = \lfloor \log_2 \beta \rfloor$. Replace "2" throughout the algorithm by "2^ℓ" and compute t (which will be a β-digit and therefore will be single-precision) using the hardware $\wedge$ operation.

In fact, on a binary machine β will generally be a power of 2 and we will thus have $2^{\ell} = \beta$, so that the algorithm can be rewritten using list operations in place of the arithmetic operations:

(1) Set k ← ( ), m ← i, n ← j.

(2) If m = ( ) or n = ( ), set k ← inverse(k) and exit.

(3) Set p ← first(m), m ← tail(m),

q ← first(n), n ← tail(n),

t ← p ∧ q (where "∧" here is the hardware operation),

k ← prefix(t,k),

and go to (2).

The latter algorithm (named IAND) is the one which actually appears in the program listings in the appendix. This algorithm and the other logical operation algorithms, whose descriptions follow, should be considered "primitives", i.e. machine-dependent algorithms which, for efficiency, must be written especially for a particular machine, based on the characteristics of the hardware. (If we ignored the question of efficiency then we could of course obtain machine independence by programming the first algorithm discussed for ∧ and similar algorithms for the other operations.)

Algorithm IAND(i,j) (Integer AND) The inputs are non-negative L-integers i and j; the output is the L-integer k = i ∧ j.

Computing time: $\sim\min(L(i),L(j))$

Algorithm IOR(i,j) (Integer OR). The inputs are non-negative L-integers i and j; the output is the L-integer k = i ∨ j.

Computing time: $\sim\max(L(i),L(j))$

Algorithm ILS(i,n) (Integer Left Shift) The inputs are an L-integer i and a non-negative Fortran integer n. The output is the L-integer $j = 2^n i$.

Computing time: $\sim L(i) + n$.

Using the algorithms just described, we obtain the following algorithm for computing sumsets:

Algorithm SUMSET(N) (Sumset of N). The input N is a list $(n_1,n_2,\ldots,n_r)$ of positive Fortran integers. The output is a list $S = (i_o,i_1,\ldots,i_r)$ of L-integers i_k such that $\phi(i_k)$ is the sumset of $n_1,\ldots,n_k$, for $k = 0,\ldots,r$. ($i_o = 1$, representing $\{0\}$, the sumset of the empty set.)

(1) Set $i \leftarrow 1$, $S \leftarrow (i)$, $N' \leftarrow N$.

(2) If $N' = ( )$, go to (3). Otherwise,

set $n \leftarrow \text{first}(n')$, $i \leftarrow i \vee 2^n i$, prefix i to S, set $N' \leftarrow \text{tail}(N')$, and repeat this step.

(3) Set $S \leftarrow \text{inverse}(S)$ and exit.

Theorem SUMSET. Assume $r > 0$ and $n = n_1+\ldots+n_r$. Then the computing time for SUMSET(N) is $\sim \sum_{j=1}^{r} (n_1+\ldots+n_j) \leq rn$.

Proof: The times for steps (1) and (3) are only ~ 1 and $\sim r$, respectively. In step (2) the times to compute $2^n i$ and $i \vee 2^n i$ are both $\sim n + L(i)$. Hence the total time for step (2) is

$$\sim \sum_{j=1}^{r} n_j + L(i_{j-1})$$

where i_{j-1} represents the sumset of $n_1,n_2,\ldots,n_{j-1}$. Since $L(i_{j-1}) \sim n_1+\ldots+n_{j-1}$, the total is

$$\sim \sum_{j=1}^{r} (n_1+\ldots+n_j) \leq \sum_{j=1}^{r} n = rn.$$

The following subprogram is useful for determining whether a given integer is an element of the set represented by another integer. (As with IAND, IOR, and ILS, the Fortran version given the appendix is machine-dependent, it being assumed that β is a power of 2.)

Algorithm MEMBER(n,i). The inputs are a Fortran integer n and a non-negative L-integer i. The output is a Fortran integer b such that if $n \in \mathcal{S}(i)$ then b = 1; otherwise, b = 0. ($b = \lfloor i/2^n \rfloor \bmod 2$.)

Computing time: $\sim \min(n+1, L(i))$.

We now come to the implementation of Algorithm 1.6G. We recall that this algorithm output all "sum index sets" satisfying certain criteria. We gave an interpretation of the command "output J" as "put J in the output set Ω". Here we shall be generating sum index lists J and we could produce as output a list of all sum index lists which satisfy the given criteria. Since, however, we may not use all of the sum index lists generated, this would be wasteful of time and storage. Thus we shall set up the algorithm so that each sum index lists generated is made available immediately. Also we shall show the stacking mechanism necessary to implement recursion in a language such as Fortran which does not allow explicit recursion.

Algorithm GEN (STACK,N,S,n,k,J) (Generate sum index lists). The inputs are:

STACK, a first order list (explained below);

N, a nonempty list $(n_1,\ldots,n_r)$ of positive Fortran integers;

S, a nonempty list $(i_0,\ldots,i_r)$ of L-integers such that

$\mathcal{S}(i_k)$ is the sumset of $n_1,\ldots,n_k$, for $0 \le k \le n$;

n, a Fortran integer;

k, a Fortran integer satisfying $1 \le k \le r$.

The outputs are STACK and J, a list of indices $(j_1,\ldots,j_v)$ such that

$$1 \le j_1 < \ldots < j_v \le k \text{ and } n_{j_1}+\ldots+n_{j_v} = n. \quad (*)$$

With fixed values for N, S and n, repeatedly performing GEN will yield all lists $J = (j_1,\ldots,j_v)$ which satisfy (*). The algorithm must be performed initially with STACK = ( ). When all such lists have been generated the values of STACK and J will be ( ). If there are no such lists then STACK = ( ) and J = ( ) will be output the first time the algorithm is performed.

In the algorithm, by "stack j" we mean "prefix j to STACK", and by "unstack j" we mean "set j ← first(STACK), STACK ← tail(STACK)".

(1) If STACK ≠ ( ), unstack j, set n ← 0 and go to (6).

(2) [Initialize.] Set j ← length(N), J ← ( ), N' ← inverse(N) S' ← inverse(S'), R ← 10, and, while j > k, repeat the following: set N' ← tail(N'), S' ← tail(S') and j ← j-1. (Steps (3)-(9) correspond to the recursive Algorithm 1.6G(j,n,J); this step has initialized to perform the algorithm with j = k and J = ( ).)

(3) [Output?] If n = 0, stack j, and exit. (This exit allows the current value of J to be used outside the algorithm; the algorithm may be reentered to generate another sum index list, provided that neither STACK nor J is altered outside the algorithm.)

(4) [Recursion necessary?] If n-first(N') $\notin \mathcal{S}$(second(S')), go to (7).

(5) [Perform G(j-1,n-n_j,{j} $\cup$ J).] Stack N',S',R, set n $\leftarrow$ n-first(N') N' $\leftarrow$ tail(N'), S' $\leftarrow$ tail(S'), R $\leftarrow$ 6, J $\leftarrow$ prefix(j,J), j $\leftarrow$ j-1, and go to (3).

(6) Unstack R,S',N', set j $\leftarrow$ j+1, n $\leftarrow$ n+first(N'), J $\leftarrow$ tail(J).

(7) [Recursion necessary?] If n $\notin \mathcal{S}$(second(S')), go to (R).

(8) [Perform G(j-1,n,J).] Stack N',S',R, set N' $\leftarrow$ tail(N'), S' $\leftarrow$ tail(S'), R $\leftarrow$ 9, j $\leftarrow$ j-1, and go to (3).

(9) Unstack R,S',N', set j $\leftarrow$ j+1, and go to (R).

(10) Exit (this exit is taken when all sum index lists have been generated).

Computing time: The time for each execution of GEN (producing one sum index list J) is $\mathcal{S}$rs, where r = length(N), s = ΣN.

The order in which Algorithm GEN produces the sum index lists will be important in the application made in Section 3.5.3 (Algorithm PFP1). Let J = $(j_1,\ldots,j_v)$ and K = $(k_1,\ldots,k_w)$ be sum index lists output by the algorithm. Then

$$j_v > k_w \Rightarrow \text{J precedes K (in order of output).}$$

This may be seen from the fact that the algorithm generates all index lists which end with the highest index before those which do not. (In fact the order of output could be completely characterized by saying that the inverses of the sum index lists appear in reverse lexicographical order).

We conclude this section with a description of an algorithm which will be used in conjunction with Algorithm GEN.

Given an arbitrary list $A = (a_1,\ldots,a_m)$ and a list $I = (i_1,\ldots,i_n)$ of integers satisfying $1 \le i_1 \le \ldots \le i_n \le \text{length}(A)$, we define $A_I = (a_{i_1},\ldots,a_{i_n})$. ($A_I = ( )$ if $A = ( )$).

Algorithm SELECT(A,I) (Select A_I from A). The inputs are lists A and I as described above (the integers on I being Fortran integers); the output is the list A_I. Those elements of A which are lists are borrowed for use in A_I.

Computing time: $\sim\min(i_n,m) + 1$.

3.4 Factor coefficient bounds

In this section we shall derive bounds on the integer coefficients of factors of a given polynomial over the integers. Several alternative bounds will be derived, which require varying amounts of computation.

Knuth ([KNU69]) describes two basic methods of bounding the coefficients of factors, both suggested by Collins. Suppose we are given a non-zero polynomial C over the integers and a positive integer k, and we wish to bound the coefficients of any factor of C of degree k. One method depends on the observation that C splits into linear factors over the field of complex numbers and any factor of degree k is a product of k such linear factors. Assuming for the moment that we have some way of bounding the moduli of the complex zeros of C, we obtain in the following theorem a bound on the norm of a factor of C.

Theorem A. (Collins) Let C be a polynomial over Z of degree $n > 1$, A be a factor of C over Z of degree k, and r be a bound on the moduli of the complex zeros of C. Then

$$|A|_1 \leq |\text{ldcf } C|(r+1)^k.$$

Proof: Let $C(x) = c(x-\gamma_1)\ldots(x-\gamma_n)$, where $c = \text{ldcf } C$, and the γ_i are the complex zeros of C. Then $A(x) = a(x-\alpha_1)\ldots(x-\alpha_k)$ where $a = \text{ldcf } A$ and $\alpha_1,\ldots,\alpha_k$ is a subsequence of $\gamma_1,\ldots,\gamma_n$. Hence

$$\begin{aligned} |A|_1 &\leq |a| \; |x-\alpha_1|_1 \ldots |x-\alpha_k|_1 \\ &= |a|(1 + |\alpha_i|) \ldots (1 + |\alpha_k|) \\ &\leq |c|(r+1)^k. \end{aligned}$$

It is not difficult to show that $|C|_1$ is a bound on the complex zeros of C; this bound is, however, usually rather crude. Knuth suggests the bound ([KNU69], Exercise 4.63-20):

$$r = 2 \max_{1 \le i \le n} \left| \frac{c_{n-i}}{c_n} \right|^{1/i} \qquad (1)$$

where $c_0, \ldots, c_n$ are the coefficients of C. This bound will often be much better than $|C|_1$, but it can be almost twice as large as $|C|_1$ (when $C(x) = x^n + c_{n-1}x^{n-1}$).

While the combination of Theorem A and this bound on the zeros of C may be useful in practice, it does not yield as good a theoretical bound on $|A|_1$ as do methods based on the second suggestion of Collins, to which we now turn.

This method uses the theory of Lagrange interpolation. Let $x_0, x_1, \ldots, x_k$ be distinct complex numbers and define

$$L_j(x) = \prod_{\substack{i=0 \\ i \ne j}}^{k} \frac{x - x_i}{x_j - x_i}, \quad 0 \le j \le k.$$

Thus $L_j(x_i) = 0$ if $i \ne j$ and 1 if $i = j$; and if A is a polynomial of degree $\le n$,

$$A(x) = \sum_{j=0}^{k} A(x_j)\, L_j(x), \qquad (2)$$

since both sides of this equation are polynomials of degree $\le k$ which assume the same values at $k + 1$ points. The polynomials $L_j(x)$, $0 \le j \le k$, are called the Lagrange interpolating polynomials for $x_0, x_1, \ldots, x_k$.

Using (1), we may relate $|A|_1$ to the values $A(x_j)$:

$$|A|_1 \le \sum_{j=0}^{k} |A(x_j)|\, |L_j|_1 .$$

If A is a factor of C then $A(x_j) | C(x_j)$, and if $C(x_j) \ne 0$ for each j then

$$|A|_1 \leq \sum_{j=0}^{k} |C(x_j)||L_j|_1.$$

The following lemma will be applied to determining bounds on the $|L_j|_1$.

Lemma 1. Let $x_1,\ldots,x_n$ be real numbers and let $A(x) = (x-x_1)\ldots(x-x_n)$.

a. If $x_1,\ldots,x_n \leq 0$ then $|A|_1 = \prod_{i=1}^{n} (1-x_i)$

b. If $x_1,\ldots,x_n \geq 0$ then $|A|_1 = \prod_{i=1}^{n} (1+x_i)$

c. In general, $|A|_1 \leq \prod_{i=1}^{n} (1 + |x_i|)$.

Proof: Gautchi [GAU62] gives a proof for the more general case in which the x_i are complex. Restricting the x_i to be real permits the following simpler proof.

a. Write $A(x) = \sum_{i=0}^{n} a_i x^i$. Then $a_o,a_1,\ldots,a_n \geq 0$, hence

$$|A|_1 = \sum_{o}^{n} a_i = A(1) = (1-x_1)\ldots(1-x_n).$$

b. In this case $(-1)^{n-i}a_i \geq 0$ for each i, hence

$$|A|_1 = \sum_{o}^{n} (-1)^{n-i}a_i = (-1)^n \sum_{o}^{n} a_i(-1)^i$$

$$= (-1)^n A(-1) = (-1)^n(-1-x_1) \ldots (-1-x_n)$$

$$= (1+x_1) \ldots (1+x_n).$$

c. We may assume, without loss of generality, that $x_1,\ldots,x_j \leq 0$, $x_{j+1},\ldots,x_n \geq 0$. Let $A_1(x) = (x-x_1)\ldots(x-x_j)$, $A_2(x) = (x-x_{j+1})\ldots(x-x_n)$. Then $A = A_1A_2$ and, from a and b,

$$|A|_1 \leq |A_1|_1|A_2|_1 = \prod_{i=1}^{j} (1-x_i) \prod_{i=j+1}^{n} (1+x_i) = \prod_{i=1}^{n} (1+|x_i|).$$

Theorem B. Let $x_o,\ldots,x_k$ be distinct real numbers and let $L_j(x)$, $0 \leq j \leq k$, be the Lagrange interpolating polynomials for $x_o,\ldots,x_k$.

Then

a. $|L_j|_1 \leq \prod_{\substack{i=0 \\ i\neq j}}^{k} \frac{1+|x_i|}{|x_j-x_i|}$, $0 \leq j \leq k$;

b. If the x_j are distinct integers such that $|x_j| \leq m$ for $0 \leq j \leq k$, then

$$|L_j|_1 \leq (m+1)^k \text{ , } 0 \leq j \leq k.$$

Proof: a. Follows immediately from the definition of L_j and Lemma 1, part c.

b. Since the x_j are distinct integers, $|x_j-x_i| \geq 1$ for each i and j; hence the result follows from a.

Theorem C. (Collins). Let C be a polynomial over Z of degree $n > 1$, A be a factor of C over Z of degree k, and let $m = \lceil (n+k)/2 \rceil$. Then

$$|A|_1 < (m+1)^{n+k+1} |C|_1.$$

Proof: Since $2m+1 = 2\left\lceil \frac{n+k}{2} \right\rceil + 1 \geq n+k+1$, and C is of degree n, C must be nonzero at least k+1 of the 2m+1 points $-m,\ldots,0,\ldots,m$. Choose $x_0,\ldots,x_k$ from these points so that $C(x_j) \neq 0$, $0 \leq j \leq k$, and let $L_0,\ldots,L_k$ be the corresponding Lagrange interpolating polynomials. Then

$$A(x) = \sum_{j=0}^{k} A(x_j)L_j(x)$$

and, as noted prior to Lemma 1,

$$|A|_1 \leq \sum_{j=0}^{k} |C(x_j)| \, |L_j|_1. \tag{3}$$

Hence, from Theorem B, part b,

$$|A|_1 \leq (m+1)^k \sum_{j=0}^{k} |C(x_j)| .$$

Letting $c_n,\ldots,c_o$ be the coefficients of C, we have

$$|C(x_j)| \leq \sum_{i=0}^{n} |c_i||x_j|^i \leq m^n |C|_1;$$

hence

$$|A|_1 \leq (m+1)^k(k+1)\ m^n\ |C|_1 \leq (m+1)^{n+k+1}\ |C|_1.$$

Corollary. For any factor A of C,

$$|A|_1 < (n+1)^{2n}\ |C|_1$$

Proof: If deg A = n then $|A|_1 \leq |C|_1$. Otherwise, we obtain the bound stated by substituting n-1 for k in the bound given in the theorem.

As noted earlier, the best theoretical bound obtainable from Theorem A and a bound on the complex zeros of C is probably

$$|A|_1 \leq |\text{ldcf } C|\ (|C|_1 + 1)^k.$$

Hence, for fixed n = deg C and k = deg A, the bound is proportional to $(|C|_1 + 1)^k$, whereas Theorem C gives a bound directly proportional to $|C|_1$. Of course, the bound actually computed from Theorem A and (1) may often be better than that given by Theorem C. We shall now show, however, how we may obtain from Lagrange interpolating theory bounds which generally are much better than that given by Theorem C.

The basic idea for improvement is to work directly with (3), evaluating C at 0, $\pm1,\pm2,\ldots$ until points $x_o,\ldots,x_k$ have been found such that $C(x_j) \neq 0$, $0 \leq j \leq k$. Assuming no zeros of C were actually found, we would have $|x_j| \leq m = \lceil k/2 \rceil$, and the points of interpolation

would be $-m,\ldots,0,\ldots,m$. The next theorem shows that for this special set of points we obtain bounds on the norms of the interpolating polynomials which are much better than in the general case.

Theorem D. Let $L_j^{(m)}(x)$, $-m \le j \le m$, be the Lagrange interpolating polynomials for the points, $-m,\ldots,0,\ldots,m$, where $m \ge 1$. Then

a. $|L_j^{(m)}|_1 = \dfrac{|j|+1}{j^2+1} \dfrac{\prod_{i=1}^{m}(i^2+1)}{(m+j)!(m-j)!}$;

b. $|L_o^{(m)}|_1 < 4$;

c. $\sum_{j=-m}^{m} |L_o^{(m)}|_1 < 4\sqrt{\pi m}$.

Proof: a. $L_j(x) = \prod_{\substack{i=-m \\ i\neq j}}^{m} \dfrac{x-i}{j-i}$

$$= (-1)^{m-j} \frac{(x+m)(x+m-1)\ \ldots\ (x-m)}{(m+j)!(m-j)!(x-j)}$$

$$= \frac{(-1)^{m-j}}{(m+j)!(m-j)!} \frac{x+j}{x} \prod_{\substack{i=0 \\ i\neq |j|}}^{m} (x^2-i^2).$$

Let $P_j(x) = x+j$, $Q_j(x) = \prod_{\substack{i=0 \\ i\neq |j|}}^{m} (x^2-i^2)$. By Lemma 1, part b,

$|Q_j|_1 = \prod_{\substack{i=0 \\ i\neq |j|}}^{m} (i^2+1)$, and since Q_j is a polynomial in x^2,

$$|P_jQ_j|_1 = |P_j|_1\ |Q_j|_1 = (|j|+1)\ \frac{\prod_{i=1}^{m}(i^2+1)}{j^2+1}.$$

Now a follows immediately.

b. From a.

$$|L_o^{(m)}|_1 = (m!)^{-2} \prod_{i=1}^{m} (i^2+1)$$

$$= \prod_{i=1}^{m} (1 + \frac{1}{i^2}).$$

Using complex variable theory, it may be shown that

$$\prod_{i=1}^{\infty} (1 + \frac{1}{i^2}) = \pi^{-1} \sinh \pi = 3.66....$$

c. From a,

$$|L_j^{(m)}|_1 = \frac{|j|+1}{j^2+1} \frac{(2m)!}{(m+j)!(m-j)!} \frac{(m!)^2}{(2m)!} \frac{\prod_{i=1}^{m} (i^2+1)}{(m!)^2}$$

$$= \frac{|j|+1}{j^2+1} \binom{2m}{m+j} \binom{2m}{m}^{-1} |L_o^{(m)}|.$$

Thus

$$\sum_{j=-m}^{m} |L_j^{(m)}| \leq \binom{2m}{m}^{-1} |L_o^{(m)}|_1 \sum_{j=-m}^{m} \binom{2m}{m+j}$$

$$= \binom{2m-1}{m} |L_o^{(m)}|_1 4^m.$$

Using the following refinement of Stirling's approximation,

$$\sqrt{2\pi}\ m^{m+1/2}\ e^{-m}\ e^{(12m+1)^{-1}} < m! < \sqrt{2\pi}\ m^{m+1/2}\ e^{-m}\ e^{(12m)^{-1}}$$

([FEL68], p. 54), it may be shown that

$$\binom{2m}{m}^{-1} < \sqrt{\pi m}\ 4^{-m}\ e^{(7m)^{-1}},$$

from which

$$\binom{2m}{m}^{-1} |L_o^{(m)}|_1 4^m < \left[(\pi^{-1} \sinh \pi)\ e^{(7m)^{-1}}\right] \sqrt{\pi m}$$

follows. The coefficient in brackets is < 4 for $m > 1$ and one can verify directly that $\sum_{j=-m}^{m} |L_j^{(m)}|_1 = 4 < 4\sqrt{\pi m}$ for $m = 1$.

From Theorem D we may obtain several different bounds for $|A|_1$, requiring varying amounts of computation. The next theorem gives three such bounds. In order to obtain these bounds we have to assume that C does not vanish at the points -m,...,m. This assumption is easily satisfied in the application to factoring, since if we find C(i) = 0 for some i, $|i| \le m$, then we may remove the linear factor x-i and begin again with the remaining factor.

Theorem E. Let $C \in Z[x]$ and let A be a factor of C of degree k. Assume $C(j) \neq 0$ for all integers j such that $|j| \le m = \lceil k/2 \rceil$ Then

a. $|A|_1 \le \sum_{j=-m}^{m} |C(j)| \; |L_j^{(m)}|_1$;

b. $|A|_1 \le 4 \sum_{j=-m}^{m} |C(j)|$;

c. $|A|_1 \le 4\sqrt{\pi m} \max_{|j| \le m} |C(j)|$.

Proof: a. Since $2m+1 = 2\lceil k/2 \rceil + 1 \ge k+1$, there are at least k+1 points j such that $|j| \le m$; hence A is determined by its values at these points:

$$A(x) = \sum_{j=-m}^{m} A(j) \, L_j^{(m)}(x),$$

from which a follows by an argument given previously.

b. As was shown in the proof of Theorem D, part c,

$$|L_j^{(m)}|_1 = \frac{|j|+1}{j^2+1} \binom{2m}{m+j} \binom{2m}{m}^{-1} |L_o^{(m)}|_1 ,$$

hence

$$|L_j^{(m)}|_1 < |L_o^{(m)}|_1 , \; j \neq 0.$$

Thus, from a and Theorem D, part b,

$$|A|_1 \le \max_{|i|\le m} |L_i^{(m)}|_1 \sum_{i=-m}^{m} |C(i)|$$

$$= |L_o^{(m)}|_1 \sum_{i=-m}^{m} |C(i)| < 4 \sum_{i=-m}^{m} |C(i)|.$$

c. Follows immediately from a and Theorem D, part c.

The bound given by part b is simplest to compute, but we shall now show that only a bit more computational effort is required for the bound in part a.

Let m be fixed, let $P = \prod_{i=1}^{m}(i^2+1)$, and let $f_j = P/[(m+j)!(m-j)!]$. From Theorem D, part a,

$$|L_j^{(m)}|_1 = \frac{|j|+1}{j^2+1} f_j .$$

It is easy to compute f_j from f_{j-1}:

$$f_j = \frac{P\ (m-j+1)}{(m+j)(m+j-1)!(m-j+1)!} = \frac{m-j+1}{m+j} f_{j-1} .$$

Hence the following algorithm computes $b = \sum_{j=-m}^{m} |C(j)|\ |L_j^{(m)}|$ exactly:

(1) Set $f \leftarrow \prod_{i=1}^{m}(i^2+1)/(m!)^2$, $b \leftarrow f\cdot|C(0)|$, $j \leftarrow 1$.

(2) (Now $b = \sum_{i=-j+1}^{j-1} |C(i)|\ |L_j^{(m)}|$, $f = f_{j-1}$.) If $j > m$, exit.

(3) Set $f \leftarrow (m-j+1)f/(m+j)$,

$b \leftarrow b + (j+1)f[|C(-j)| + |C(j)|]/(j^2+1)$,

$j \leftarrow j+1$, and go to (2).

This algorithm is the basis of the bounding algorithm PFB1 which we shall now describe. The main modification is to arrange the computation so that only integer arithmetic is required, rather than rational arithmetic. This gives a somewhat larger bound, which is nevertheless smaller than that given by part b of Theorem D. It is also necessary to check for cases in which C(i) = 0 for one or more of the points i used, and to remove linear factors (x-i) in such cases.

Algorithm PFB1 (C_o, m, C, L, b). (Polynomial factor bounding algorithm, 1 variable) The inputs are a polynomial C_o over Z of positive degree and a positive Fortran integer m. The outputs are a polynomial C over Z, a list L of linear polynomials such that $C_o = C\prod L$, and a positive L-integer b such that if A is any factor of C of degree $\leq m$, then $|A|_1 \leq b$.

(1) Set $C \leftarrow C_o$, $L \leftarrow ( )$.

(2) If deg C = 1, set $b \leftarrow |C|_1$ and exit.

(3) Compute C(0). If $C(0) \neq 0$, go to (5). Otherwise, set $j \leftarrow 0$.

(4) Set $C(x) \leftarrow C(x)/(x-j)$, prefix x-j to L, and go to (2).

(5) Set $f \leftarrow 4$, $b \leftarrow f \cdot |C(0)|$, $j \leftarrow 1$.

(6) (Now $b \geq \sum_{i=-j+1}^{j-1} |C(i)|\ |L_j^{(m)}|_1$, $f \geq f_{j-1}$, $C_o = C\prod L$ and $C(i) \neq 0$ for $|i| \leq j-1$.) If $j > m$, exit.

(7) Compute C(j). If C(j) = 0, go to (4).

(8) Compute C(-j). If C(-j) = 0, set $j \leftarrow -j$ and go to (4).

(9) Set $f \leftarrow \lceil (m-j+1)f/(m+j) \rceil$,

$b \leftarrow b + \lceil ((j+1)f[|C(-j)| + |C(j)|])/(j^2+1) \rceil$,

$j \leftarrow j+1$, and go to (6).

Note: The Fortran version of this algorithm is written with the further assumption that $m^2 + 1 < \beta$, where β is the base used in the SAC-1 integer arithmetic system.

Theorem PFB1. Let $n = \deg C_o$ and γ be a bound on the norm ($|\ |_1$) of any factor of C_o. Let k be the number (counting multiplicity) of linear factors x-i of C_o such that $|i| \leq m$. Assume $L(n) \sim 1$, $L(m) \sim 1$. Then the computing time for Algorithm PFB1 is

$$(k+1)\ m[n^2+nL(\gamma)].$$

Proof: The time for step (4) is $\trianglelefteq n\ L(\gamma)$ by Axiom C_3, part c, and the assumption $L(m) \sim 1$. Since evaluation of C at j involves essentially the same computation as trial division of C by x-j, (this is true of the SAC-1 Algorithm PSUBST which is used) we obtain from Theorem 1.4T a bound for steps (7) and (8): $n^2L(j)+nL(j)L(\gamma) \sim n^2+nL(\gamma)$. In step (9), the time to compute f is ~ 1 and to compute the term to be added to b is $\trianglelefteq n+L(\gamma)$. The time for the addition is $\trianglelefteq L(b_1)$ where b_1 is the sum. Since

$$b_1 \leq 4 \sum_{i=-j}^{j} |C(i)| \text{ and } |C(i)| \leq \gamma |i|^n,$$

we have

$$b_1 \leq 4\ (2j+1)\ j^n\gamma,$$

hence

$$L(b_1) \trianglelefteq n + L(\gamma).$$

Steps (6)-(9) are executed at most m times before finding one of the k linear factors x-i such that $|i| \leq m$, or terminating if the k linear factors have already been removed, hence we obtain the bound stated for all of the executions of these steps, and it is easily verified that this also bounds the time for all executions of the other steps.

3.5 Main algorithms for factoring over the integers

We are now in a position to discuss the principal algorithms for factoring univariate polynomials over the integers. These algorithms are based on the abstract algorithms presented in Sections 2.6, 2.7 and 2.8.

3.5.1 Algorithm PFH1

We shall not present here an algorithm based on Algorithm 2.7H (Hensel's algorithm), since a theoretical computing time analysis of such an algorithm shows it to be inferior to an algorithm which is based on the Quadratic Hensel Algorithm (2.8Q). We shall now discuss the latter algorithm, which we call Algorithm PFH1 (a comparison of the computing times is given following Theorem PFH1 below).

In Algorithm PFH1, unlike Algorithm 2.8Q, we shall specify a set of representatives of $D/(q) = Z/(q)$ (recall that q takes on the values $p, p^2, p^4, \ldots$) The set to be used is $\{n \in Z : |n| < q/2\}$; this choice is made in order to simplify the application of Algorithm 2.7P. The following lemma will be used to show that the coefficients of the polynomials computed for a given value of q do lie in this set.

Lemma 1. Let m and n be odd positive integers and $U, V, W \in Z[x]$ such that

$$U = V + mW, \quad |V|_\infty < m/2, \quad |W|_\infty < n/2.$$

Then $|U|_\infty < mn/2$.

Proof: $|U|_\infty \leq |V|_\infty + m|W|_\infty \leq \frac{m-1}{2} + m\frac{n-1}{2} = \frac{mn-1}{2}$.

Algorithm PFHl (p,m,C,$\overline{A}$,$\overline{B}$,$\overline{S}$,$\overline{T}$,A,B) (Polynomial factorization based on the Quadratic Hensel Algorithm (2.8Q), 1 variable). The inputs are:

p, an odd positive prime integer (Fortran integer);

$m = p^j$ for some positive integer j (m is an L-integer);

C, a primitive positive polynomial over Z;

$\overline{A},\overline{B},\overline{S},\overline{T}$, polynomials over GF(p) such that

$$h_p(C) = \overline{AB} \text{ and } \overline{AS}+\overline{BT} = 1.$$

The outputs are polynomials A,B over Z such that

$C \equiv AB \pmod m$;

$h_p(A) = \overline{A}$, $h_p(B) = \overline{B}$;

$|\text{ldcf } A| < p/2$; $|A|_\infty$, $|B|_\infty < m/2$

Note: The conditions $h_p(A) = \overline{A}$ and $|\text{ldcf } A| < p/2$ imply that ldcf A is a unit mod m and deg A = deg $\overline{A}$.

(1) [Initialize.] Set $q \leftarrow p$ and obtain $A,B,S,T \in Z[x]$ such that $h_p(A) = \overline{A},\ldots,h_p(T) = \overline{T}$, $|A|_\infty < p/2,\ldots,|T|_\infty < p/2$. (This may be done conveniently using Algorithm CPGARN described in [COL69a]).

(2) [Done?] If q = m, exit. (This exit is taken only if m = p.)

(3) [Compute Y,Z.] (Now $A,B,S,T \in Z[x]$, $C \equiv AB$ and $AS+BT \equiv 1 \pmod q$, $h_p(A) = \overline{A}$, $h_p(B) = \overline{B}$, $|\text{ldcf } A| < p/2$, $|A|_\infty$, $|B|_\infty$, $|S|_\infty$, $|T|_\infty < q/2$.) Set $U \leftarrow (C-AB)/q$. If $q^2 > m$, set $\tilde{q} \leftarrow m/q$, $\tilde{A} \leftarrow \text{MPMOD}(\tilde{q},A),\ldots,\tilde{T} \leftarrow \text{MPMOD}(\tilde{q},B)$; otherwise, set $\tilde{q} \leftarrow q$, $\tilde{A} \leftarrow A,\ldots,\tilde{T} \leftarrow T$. (Now $|\tilde{A}|_\infty,\ldots,|\tilde{T}|_\infty < \tilde{q}/2$.) Apply Algorithm MPSPEQ to $\tilde{q},\tilde{A},\tilde{B},\tilde{S},\tilde{T},U$, obtaining $Y,Z \in Z[x]$ such that $\tilde{A}Y+\tilde{B}Z \equiv U \pmod{\tilde{q}}$, $|Y|_\infty < \tilde{q}/2$, $|Z|_\infty < \tilde{q}/2$ and deg Z < deg A.

(4) [Compute A*, B* and check for end.]. Set $A^* \leftarrow A+qZ$, $B^* \leftarrow B+qY$. (Then $C \leftarrow A^*B^* \pmod{qq}$; $h_p(A^*) = \overline{A}$, $h_p(B^*) = \overline{B}$, $|\text{ldcf } A^*| < p/2$, and, by Lemma 1, $|A^*|_\infty$, $|B^*|_\infty < q\tilde{q}/2$.) If $q^2 \geq m$ (in which case $q\tilde{q} = m$), set $A \leftarrow A^*$, $B \leftarrow B^*$ and exit.

(5) [Compute Y_1, Z_1.] Set $U_1 \leftarrow (A^*S+B^*T-1)/q$. Apply Algorithm MPSPEQ to q,A,B,S,T,U_1, obtaining $Y_1, Z_1 \in Z[x]$ such that $AY_1+BZ_1 \equiv U_1 \pmod{q}$, $|Y_1|_\infty$, $|Z_1|_\infty < q/2$ and $\deg Z_1 < \deg A$.

(6) [Compute S*,T*.] Set $S^* \leftarrow S-qY_1$, $T^* \leftarrow T-qZ_1$. (Then $A^*S^*+B^*T^* \equiv 1 \pmod{q^2}$, $|S^*|_\infty$, $|T^*|_\infty < q^2/2$).

(7) [Advance.] Replace q,A,B,S,T by q^2,A^*,B^*,S^*,T^*, and go to (3).

This algorithm differs from Algorithm 2.8Q mainly in the method of termination: at the last iteration the modulus for input to Algorithm MPSPEQ has been adjusted so that the coefficients of the final output are bounded by $m/2$ (instead of $p^{2^k}/2$ where $2^{k-1} < j \leq 2^k$). Also the computation of S and T is bypassed on the final iteration, since they are not needed in the application of the algorithm. This results in a non-trivial saving in computing time. (See the remark following Theorem PFH1.)

In order to analyze the computing time of Algorithm PFH1 it is necessary to establish some additional facts about the degrees of the polynomials computed. The proofs will be based on the following theorem, which is related to Algorithm 2.7S and Theorem 2.7S.

Theorem D. Let E be a commutative ring with identity and $A,B,Y,Z,U \in E[x]$ with ldcf A a unit of E, $\deg Z < \deg A$ and $AY+BZ = U$. Then

$$\deg Y \leq \max(\deg U - \deg A, \deg B - 1).$$

Proof: Since ldcf A is a unit, $\deg(AY) = \deg A + \deg Y$. Also, from $AY+BZ = U$,

$$\begin{aligned}\deg(AY) &\leq \max(\deg U, \deg BZ)\\ &\leq \max(\deg U, \deg B + \deg Z)\\ &\leq \max(\deg U, \deg B + \deg A - 1);\end{aligned}$$

hence

$$\deg Y \leq \max(\deg U - \deg A, \deg B - 1).$$

Corollary 1. In Algorithm 2.7H let B be chosen initially in step (1) so that $\deg B = \deg \overline{B}$, and let Y be chosen in step (4) so that $\deg Y = \deg \overline{Y}$. Then A and B always satisfy

$$\deg A + \deg B \leq \deg C. \qquad (*)$$

Proof: By the assumption, $\deg A + \deg B = \deg \overline{A} + \deg \overline{B} = \deg(\overline{AB}) = \deg h(C) \leq \deg C$ after execution of step (1). Assuming (*) holds at a given execution of step (2), we shall show that it holds at the next.

In step (3) we have, by (*), that $\deg U \leq \deg C$, hence $\deg \overline{U} \leq \deg C$. By Theorem D,

$$\begin{aligned}\deg Y = \deg \overline{Y} &\leq \max(\deg \overline{U} - \deg \overline{A}, \deg \overline{B} - 1)\\ &\leq \max(\deg C - \deg A, \deg B - 1)\\ &= \deg C - \deg A.\end{aligned}$$

Let $A_1 = A+qZ$, $B_1 = B+qY$. Then

$$\deg B_1 \leq \deg C - \deg A = \deg C - \deg A_1,$$

and since step (5) sets $A \leftarrow A_1$ and $B \leftarrow B_1$, (*) is still valid when step (2) is reached again.

Corollary 2. In Algorithm 2.8Q let B be chosen initially so that deg B = deg $\overline{B}$ and let Y be chosen in step (3) with minimum degree (i.e. so that q $\nmid$ ldcf Y). Then A and B always satisfy

$$\deg A + \deg B \leq \deg C.$$

Proof: Almost identical to the proof of Corollary 1.

Corollary 3. In Algorithm PFH1, A and B always satisfy

$$\deg A + \deg B \leq \deg C.$$

Proof: The assumptions of Corollary 2 are satisfied since initially $h_p(B) = \overline{B}$ and $|B|_\infty < p/2$; and $|Y|_\infty < q/2$.

Lemma 3. In Algorithm PFH1, assume deg $\overline{S}$ < deg $\overline{B}$ and deg $\overline{T}$ < deg $\overline{A}$. Then A,B,S,T always satisfy

$$\deg T < \deg A, \quad \deg S < \deg B. \qquad (*)$$

Proof: Since A is chosen initially so that $h_p(A) = \overline{A}$ and $|A|_\infty < p/2$, we have deg A = deg $\overline{A}$ initially, and similarly for B,S,T. Thus (*) clearly holds initially. The first inequality clearly remains valid, since deg A remains constant and T is changed only by subtracting qZ_1 where deg Z_1 < deg A. A new value S* of S is computed as $S^* = S - qY_1$; we shall show that deg S* < deg B*. Since deg S < deg B $\leq$ deg B*, it suffices to show that deg Y_1 < deg B*.

Since $|Y_1|_\infty, |Z_1|_\infty < q/2$, Y_1 and Z_1 have the same degrees when regarded as polynomials over Z/(q). Therefore, from Theorem D we have

$$\deg Y_1 \leq \max(\deg U_1 - \deg A, \deg B - 1)$$
$$\leq \max(\deg U_1 - \deg A^*, \deg B^* - 1).$$

Also

$$\deg U_1 \leq \max(\deg A^*S, \deg B^*T)$$
$$\leq \max(\deg A^* + \deg S, \deg B^* + \deg T)$$
$$\leq \max(\deg A^* + \deg B - 1, \deg B^* + \deg A^* - 1)$$
$$= \deg A^* + \deg B^* - 1,$$

hence

$$\deg U_1 - \deg A^* \leq \deg B^* - 1$$

and

$$\deg Y_1 \leq \deg B^* - 1.$$

Theorem PFH1. Assume in Algorithm PFH1, that $\deg \overline{A} > 0$, $\deg \overline{B} > 0$, $\deg \overline{S} < \deg \overline{A}$, $\deg \overline{T} < \deg \overline{B}$, $n = \deg C$, $L(n) \sim 1$, and $c = |C|_1$. Then the computing time for the algorithm is

$$n^2 L(m)^2 + nL(m)L(c).$$

Proof: Let $h = \deg A$, $k = \deg B$. By the lemmas just proved, we have, at any time during execution of the algorithm

$$h + k \leq n,$$
$$\deg S < k, \deg T < h.$$

We proceed now to analyze the time for a single execution of each step.

(1) The time for all applications of CPGARN is n (see [COL69a], pp. 16-17).

(2) The time to compare q with m is $L(m)$.

(3) The time for A B is $(h+1)(k+1)L(q)^2$.

Since $|AB|_1 \leq (h+1)(k+1)q^2$ and $\deg(AB) \leq n$ the time to subtract AB from C is $(n+1)[L(c)+L(q)]$ (using $L(n) \sim 1$.) Since $|C-AB|_1 \leq C + (h+1)(k+1)q^2 \leq C(h+1)(k+1)q^2$, the time to divide C-AB by q is

$$\preceq (n+1)L(q)[L(c) + L(h+1) + L(k+1) + 2L(q)]$$

$$\preceq (n+1)L(q)[L(c) + L(q)].$$

If $q^2 \leq m$ then the time to compare q^2 with m is $\preceq L(m)$ and to obtain $\hat{q},\hat{A},\hat{B},\hat{S},\hat{T}$ is $\preceq 1$. If $q^2 > m$ then $q < m$, hence the time to compare q^2 with m is again $\preceq L(m)$; the time to apply MPMOD to A,B,S and T is $\preceq nL(\tilde{q})L(q) \preceq nL(q)^2$.

In order to apply Theorem MPSPEQ, let $\bar{n} = \max(h+k, \deg U)$; then $\bar{n} \leq n$ and thus the time for Algorithm MPSPEQ is

$$\preceq n^2L(q)^2 + nL(q)L(u),$$

where $u = |U|_1 \leq (h+1)(k+1)cq$; hence the time is

$$\preceq n^2L(q)^2 + nL(q)L(c).$$

(4) The time for $q \cdot Z$ is $\preceq h\ L(q)L(\tilde{q}) \preceq h\ L(q)^2$ and the tome to add qZ to A is $\preceq h\ L(q)$; the time for $q \cdot Y$ is $\preceq k\ L(q)^2$ for B+qY is $\preceq k\ L(q)$. Thus the time for the whole step is $\preceq n\ L(q)^2$.

(5) The time to compute both A*S and B*T is $\preceq hk\ L(q)^2$ and the time to add A*S and B*T and subtract 1 is $\preceq n\ L(q)$. To divide the result by q requires $\preceq n\ L(q)^2$. Now let $\bar{n} = \max(h+k, \deg U_1)$; again $\bar{n} \leq n$ and the time for Algorithm MPSPEQ is

$$\preceq n^2L(q)^2 + n\ L(q)L(u_1)$$

where $u_1 = |U_1|_1 \leq 2hkq$; hence the time is $\preceq n\ L(q)^2$.

(6) The time for this step has the same bound as step (4): $n\ L(q)^2$.

(7) The time for this step is $\preceq 1$.

From this analysis we conclude that the total time for a single execution of steps (3) through (7) is dominated by

$$n^2L(q)^2 + n\ L(q)L(c) + L(m).$$

To get the total for all executions, let k be chosen so that $p^{2^{k-1}} \leq m < p^{2^k}$; then

$$\sum_{i=0}^{k} L(p^{2^k}) \sim \sum_{i=0}^{k} 2^i L(p) \sim 2^{k+1} L(p) \sim L(p^{2^{k+1}}) \sim L(m),$$

$$\sum_{i=0}^{k} L(p^{2^k})^2 \sim \sum_{i=0}^{k} 4^i L(p)^2 \sim 4^{k+1} L(p)^2 \sim L(p^{2^{k+1}})^2 \sim L(m)^2,$$

$$\sum_{i=0}^{k} L(m) = (k+1)L(m) \sim L(L(m))L(m) \lesssim L(m)^2.$$

Hence the total time for all executions of steps (3)-(7) is

$$\lesssim n^2 L(m)^2 + n\, L(m)L(c);$$

this of course also bounds the time for steps (1) and (2), and is thus a bound for the entire algorithm.

This analysis shows that the time for the entire algorithms is no more than a constant multiple of the time for an iteration with $q \sim m$. Hence, the portion of time saved by not computing S and T on the last iteration does not tend to zero as m grows large.

A similar analysis for an algorithm based on Algorithm 2.7H yields a bound of $n^2 L(m)^3 + n\, L(m)^2 L(c)$, which is greater by a factor of L(m) than the time for Algorithm PFH1.

3.5.2 Algorithm PFC1.

The following algorithm corresponds to Algorithm 2.7C, or more precisely, to Algorithm 2.7C with the application of Algorithm 2.7H replaced by an application of Algorithm 2.8Q, since it uses Algorithm PFH1. A second important difference is that the assumption

that $p \nmid$ ldcf C is made, permitting the last factor F_r to be computed in a different, and probably more efficient, manner.

Algorithm PFC1 (p,m,C,G) (Polynomial factorization based on Algorithm 2.7C, 1 variable)

The inputs are:

p, an odd positive prime integer (Fortran integer);

$m = p^j$ for some positive integer j (m is an L-integer);

C, a primitive positive polynomial over Z such that

$p \nmid$ ldcf C and $h_p(C)$ is squarefree over GF(p);

$G = (G_1,\ldots,G_r)$ where $r \geq 2$, each G_i is a monic polynomial over GF(p), of positive degree, and

$h_p(C) = (\text{ldcf } h_p(C))\ G_1 \ldots G_r$.

The output is a list $F = (F_1,\ldots,F_r)$ of polynomials over Z such that

$C \equiv (\text{ldcf } C)\ F_1 \ldots F_r \pmod m$

$h_p(F_i) = G_i$, $\deg F_i = \deg G_i$, F_i is monic, $|F_i|_\infty \leq m/2$, $i = 1,\ldots,r$.

(1) Set $\overline{C} \leftarrow h_p(C)$, $G' \leftarrow G$, $F \leftarrow ( )$.

(2) Set $\overline{A} \leftarrow \text{first}(G')$, $G' \leftarrow \text{tail}(G')$, $\overline{B} \leftarrow \overline{C}/\overline{A}$.

(3) Using Algorithm CPEGCD, obtain $\overline{S}$ and $\overline{T}$ over GF(p) such that $\overline{AS}+\overline{BT} = 1$.

(4) Apply Algorithm PFH1 to $p,m,C,\overline{A},\overline{B},\overline{S},\overline{T}$ and let A and B be the output.

(5) Prefix A to F and set $C \leftarrow B$, $\overline{C} \leftarrow \overline{B}$.

(6) If $\text{tail}(G') \neq ( )$, go to (2).

(7) Set $\tilde{c} \leftarrow \text{MRECIP}(m,\text{ldcf } C)$, $A \leftarrow \text{MPMOD}(m,\tilde{c}\cdot C)$. Prefix A to F, invert F, and exit.

Theorem PFC1. Let $n = \deg C$, $c = |C|_1$, $r = \text{length}(G)$ and assume $L(n) \sim 1$. Then the computing time for Algorithm PFC1 is

$$\preceq rn^2L(m)^2 + n\,L(m)L(c).$$

Proof: The time to compute $h_p(C)$ is $\preceq (n+1)L(c)$. Letting $k = \deg \bar{A}$ in step (2), the time to compute $\bar{C}/\bar{A}$ is $\preceq (k+1)(n-k+1)$ and the time for Algorithm CPEGCD in step (3) has the same bound. By Theorem PFC1 the time for step (4) is $\preceq n^2L(m)^2 + n\,L(m)L(c)$ the first time and $\preceq n^2L(m)^2$ thereafter, since $|C|_1 \leq (n+1)m/2$ thereafter.

Steps (5) and (6) are trivial. In step (7) the time for MRECIP(m,ldcf C) is $\preceq L(m)^2$, for $\tilde{c}\cdot C$ is $(n+1)L(m)^2$, for MPMOD(m,$\tilde{c}$C) is $\preceq (n+1)L(m)^2$ and for inverting F is $\preceq r$.

The time for steps (2) and (3) is $\preceq n^2$. Steps (2)-(6) are executed r-1 times, and the conclusion of the theorem now follows easily.

3.5.3 Algorithm PFP1

The next algorithm to be described in PFP1, which is based on Algorithm 2.6P. PFP1 takes an additional input, an L-integer D which represents a set of positive integers. It is assumed in the algorithm that this set contains the degree d of every factor A of C such that $0 < d \leq d^*$, where $d^* = \lfloor (\deg C)/2 \rfloor$. The use of D allows the algorithm to take advantage of information about the possible degrees of factors of C which may have been gathered from previous computation. In Algorithm PFZ1, which is described in the next subsection, such information is obtained by comparing the degrees of

factorizations of C modulo several prime integers. (If no information about degrees of factors were known then PFP1 could be performed with $D = 2+4+8+\ldots+2^{d^*}$, representing the set $\{1,2,3,\ldots,d^*\}$.)

Another important modification of Algorithm 2.6P which is made in P1 is the inclusion of a "trailing coefficient test". Before computing a tentative factor A^* of C^* (= (ldcf C)C), its trailing coefficient (constant term) t is computed from the trailing coefficients of the selected modulo m factors. If t fails to divide t^* = tlcf C^*, then A^* cannot divide C^*, and hence the computation of A^* and the trial division of C^* by A^* may be skipped.

In the algorithm we shall take as a set of representatives of Z/(m) the set $R = \{n: |n| < m/2\}$. The input C is assumed to be R-factorable, which, we recall, means that R contains the coefficients of any factor A^* of C^* = (ldcf C)C such that deg $A^* \leq d^*$ and ldcf $A^* |$ ldcf C. We showed in Section 3.4 how a bound b on the coefficients of factors of C of degree $\leq d^*$ could be computed, and in Section 2.6 that the choice of $m > 2b$ (ldcf C) ensures that C will be R-factorable.

In the statement of the algorithm we use the notation N_J introduced at the end of Section 3.3.

Algorithm PFP1 (m,C,G,D) (Polynomial factorization based on Algorithm 2.6P, 1 variable). The inputs are:

m, an odd L-integer > 1;

C, a nonconstant, primitive, positive polynomial over Z which is R-factorable, where $R=\{n \in Z: |n| < m/2\}$;

G, a list $(G_1,\ldots,G_r)$ of monic polynomials, a complete factorization of C modulo m;

D, a positive L-integer representing a set of positive integers which contains the set $\{d:d = \deg A,\ A \mid C,\ 0 < d \leq \lfloor(\deg C)/2\rfloor\}$.

The output of the algorithm is a list F of the prime positive polynomials over Z such that $C = \Pi F$.

(1) Set $F \leftarrow ( )$; $d \leftarrow 1$; $N \leftarrow (n_1,\ldots,n_r)$, where $n_i = \deg G_i$; $T \leftarrow (t_1,\ldots,t_r)$, where $t_i = \text{tlcf } G_i$, $k \leftarrow r \leftarrow \text{length}(G)$.

(2) Set $c \leftarrow \text{ldcf } C$, $C^* \leftarrow c \cdot C$, $t^* \leftarrow \text{tlcf } C^*$, $S \leftarrow \text{SUMSET}(N)$, $\mathcal{D} \leftarrow D \wedge \text{last}(S)$.

(3) If $d > \lfloor(\deg C)/2\rfloor$, prefix C to F and exit. If $d \notin \mathcal{D}$ or $k=0$, go to (7).

(4) Using Algorithm GEN, generate a new list $J = (j_1,\ldots,j_v)$ such that $1 \leq j_1 < \ldots < j_v \leq k$ and $\Sigma N_J = d$. If all such lists have already been found, go to (7).

(5) Set $t \leftarrow \text{MPLPR}(m,\text{prefix}(c,T_J))$. If $t \nmid t^*$, go to (4).

(6) Set $A^* \leftarrow \text{MPLPR}(m,\text{prefix}(c,G_J))$. If $A^* \nmid C^*$, go to (4). Otherwise, set $B^* \leftarrow C^*/A^*$ and go to (8).

(7) Set $d \leftarrow d+1$, $k \leftarrow r$ and go to (3).

(8) Set $A \leftarrow \text{pp}(A^*)$, prefix A to F, and set $C \leftarrow B^*/\text{ldcf } A$. Construct $K = (k_1,\ldots,k_w)$ such that $1 \leq k_1 < \ldots < k_w \leq r$ and $\{j_1,\ldots,j_v\} \uplus \{k_1,\ldots,k_w\} = \{1,\ldots,r\}$. Set $G \leftarrow G_K$, $N \leftarrow N_K$, $T \leftarrow T_K$, $r \leftarrow r-v$, $k \leftarrow j_v-v$, and go to (2).

Aside from the differences between this algorithm and Algorithm 2.6P mentioned earlier, there is another important change, involving the generation of sum index lists. We make use of the fact

noted in Section 3.3, that Algorithm GEN outputs a particular sum index list $J = (j_1,\ldots,j_v)$ <u>after</u> outputting all sum index lists with higher final index. Hence, upon finding a factor A of degree d corresponding to J in step (8), we know that all sum index lists $I = (i_1,\ldots,i_u)$ with $i_u > j_v$ and $\Sigma N_I = d$ have already been tried. After removing the elements corresponding to J from lists G, N and T, and setting $r \leftarrow r-v$, the sum index lists $J' = (j_1,\ldots,j'_{v'})$ such that $1 \le j_1 < \ldots < j'_{v'} \le r$ and $\Sigma N_{J'} = d$ which have already been tried are those with $j'_{v'} > j_v - v$, hence setting $k \leftarrow j_v - v$ in this case is valid.

Thus no sum index lists are considered in the algorithm more than once, and this fact is used in the proof of the computing time analysis theorem which follows.

<u>Theorem PFP1</u>. Let $C = C_1C_2\ldots C_e$, $d_i = \deg C_i$, $1 \le d_1 \le \ldots \le d_e$, $n = \deg C$, and

$$\mu = \begin{cases} \max\{d_{e-1}, \lfloor d_e/2 \rfloor\} & \text{if } e > 1. \\ \lfloor n/2 \rfloor & \text{if } e = 1. \end{cases}$$

Let δ be the number of products $P = \Pi G_J$ such that $J = (j_1,\ldots,j_v)$, $1 \le j_1 < \ldots < j_v \le r = \text{length}(G)$, and $1 \le \deg P \le \mu$, and let T be the number of these products satisfying the additional condition that MMOD(m,(ldcf C)(tlcf P)) is a divisor of (ldcf C)(tlcf C). Finally, let γ be a bound on $|A|_1$ for any factor A of C. Then the computing time for PFP1 is

$$\begin{aligned} &\le [(T+1-e)\mu n^2 + T\mu^2 + \delta\mu]\, L(m)^2 \\ &\quad + (T\mu n + \delta)\, L(m)L(\gamma) + \delta r n \\ &\le \delta\mu n\, L(m)[n\, L(m) + L(\gamma)] \end{aligned}$$

where $\delta \leq \min(2^r, r^\mu)$.

Proof: We first consider the time for a single execution of each step.

(1) Since $r \leq n$ the time for this step is $\preceq n$.

(2) Since C is R-factorable, where $R = \{n: |n| \leq m/2\}$ we have $|c| \leq m/2$; hence the time to compute $c \cdot C$ is $n\, L(m)L(\gamma)$. The time for the other operations is $\preceq rn$.

(3) The time to test whether $d \in \mathfrak{D}$ is $\preceq n$.

(4) The time for Algorithm GEN is $\preceq rn$.

(5) The time to obtain T_J is $\preceq r$ and, since length $(T_J) \leq d \leq \mu$ the time to apply MPLPR is $\preceq \mu L(m)^2$. Testing whether t divides t* requires $\preceq L(m)L(\gamma)$, since $|t| \leq m$ and $|t^*| \leq \gamma^2$.

(6) The time to obtain G_J is $\preceq r$ and to apply MPLPR is $\preceq \mu^2 L(m)^2$. The time to test whether A* divides C* is, from Axiom $1.4C_1$, part c, and Theorem 1.4T,

$$\preceq \begin{cases} \mu n\, L(m)L(\gamma), & \text{if the quotient exists,} \\ \mu n\, L(m)L(\gamma) + \mu n^2 L(m)^2, & \text{otherwise.} \end{cases}$$

(7) $\preceq 1$.

(8) To compute pp(A*) requires $\preceq \mu L(m)^2$ and to divide B* by ldcf A requires $\preceq n\, L(m)L(\gamma)$.

We summarize these bounds in the following table and also indicate the maximum number of times each step is executed.

Step	Time for one execution	Maximum number of executions
(1)	n	1
(2)	$n\,L(m)L(\gamma) + rn$	e
(3)	n	$e + \mu$
(4)	rn	$\delta + \mu$
(5)	$r + \mu L(m)^2 + L(m)L(\gamma)$	δ
(6)	$r + \mu^2 L(m)^2$	T
	$\mu n\,L(m)L(\gamma)$	e-1
	$\mu n\,L(m)L(\gamma) + \mu n^2 L(m)^2$	$T + 1 - e$
(7)	1	μ
(8)	$\mu L(m)^2 + n\,L(m)L(\gamma)$	e-1

Assume that $D = 2+4+\ldots+2^{\lfloor (\deg C)/2 \rfloor}$; obviously the time for the general case will be dominated by the time for the case in which this assumption holds.

It is then easy to see that the values given for the maximum number of executions are exact in case $e = 1$ (i.e. when C is irreducible). The values for steps (1), (2), (7), (8) and the second part of step (6) are exact when $e > 1$ as well. That the other values given are upper bounds when $e > 1$ follows from: (1) the number of products to be considered is diminished when a factor is removed in step (8), since the length of G is reduced; and (2) no products are considered more than once (note that this would not be true if we always took $k = r$ in step (4)).

The first bound in the statement of the theorem may now be obtained from the above table using $e\text{-}1 \leq T \leq \delta$; the second bound

then follows using these relations and $\mu < n$.

In order to obtain the bound stated for δ, let $N = (n_1, n_2, \ldots, n_r)$, where $n_i = \deg G_i$, and let $s(N,d)$ denote the number of sum index lists $J = (j_1, \ldots, j_v)$ such that $1 \leq j_1 < \ldots < j_v \leq r$ and $\Sigma N_J = d$. Thus

$$\delta = \sum_{d=1}^{\mu} s(N,d).$$

If we define $s(N,d) = 0$ for $d < 0$ and observe that for $N' = (n_1, \ldots, n_r, n)$ we have the recurrence relation

$$s(N',d) = s(N,d) + s(N,d-n),$$

then we may easily prove by induction on r the following:

$$\sum_{d=0}^{n} s(N,d) = 2^r, \text{ where } r = \text{length}(N),\ n = \Sigma N,$$

$$\sum_{d=1}^{\mu} s(N,d) \leq r^{\mu}, \text{ where } r = \text{length}(N),\ k \geq 1.$$

Combining these results, we thus have $\delta \leq \min(2^r, r^{\mu})$.

Corollary. Assume the hypothesis of Theorem PFP1 and, in addition, that C is a product of polynomials of degree no greater than some fixed positive integer k. Then the computing time for Algorithm PFP1 is

$$\preceq n^{k+2} L(m)^2 + n^{k+1} L(m) L(\gamma).$$

Proof: Use the relations $e, r \leq n$, $\mu \leq k \sim 1$ and $T \leq \delta \leq r^{\mu} \leq n^k$ to simplify the second bound given in the theorem.

For example, if C is the product of linear factors, the computing time is

$$\preceq n^3 L(m)^2 + n^2 L(m) L(\gamma).$$

3.5.4 Algorithm PFZ1

This algorithm is based on Algorithm 2.7.1Z, but again there are some significant modifications. Most importantly, factorizations modulo several primes are considered, and the prime p which yields the fewest irreducible factors is chosen for input to the Hensel algorithms. This reduces the probability that the mod p factorization will have many more irreducible factors than the integer factorization. This is of critical importance since the computing time for Algorithm PFP1 can be an exponential function of the number of irreducible factors mod p.

Secondly, important information is obtained from the mod p factorizations about the possible degrees of factors of the input polynomial C. The set of degrees of factors of C must be contained in the set D_p of degrees of mod p factors for any prime p, and therefore must be contained in $D_{p_1} \cap D_{p_2} \cap \ldots \cap D_{p_\nu}$ where $p_1, p_2, \ldots, p_\nu$ are the primes for which factorizations are carried out. D_p is just the sumset of the list of degrees of the irreducible factors mod p, and is thus easily computed using Algorithm SUMSET (Section 3.3). If C is irreducible then we will often find

$$D_{p_1} \cap D_{p_2} \cap \ldots = \{0, \deg C\}$$

after a few primes $p_1, p_2, \ldots$ have been tried, thus proving irreducibility without ever having to apply the Hensel algorithms.

We should mention that irreducibility will not often be proved by finding that C is irreducible mod p for one of the primes tried, unless ν is taken close to $n = \deg C$. This may be shown by

an argument similar to Knuth's proof that almost all polynomials over the integers are irreducible [KNU69, Ex. 4.6.2-27]: The probability that a random monic polynomial of degree n over GF(p) is reducible is about 1-1/n, and by the Chinese Remainder Theorem the probability that a random polynomial of degree n over the integers is reducible modulo ν different primes is about $(1-1/n)^{\nu}$. For large n, this probability is

$$[(1 - \frac{1}{n})^n]^{\nu/n} \approx e^{-\nu/n}.$$

In order to have $e^{-\nu/n} < \frac{1}{2}$, we must have $\nu > n \ln 2 \approx .7n$; thus the probability of C being reducible modulo all of the primes $p_1, p_2, \ldots, p_\nu$ will be less than 1/2 only if we take ν greater than about .7n.

Two algorithms, which are implemented in the SAC-1 Modular Arithmetic System [COL69a] are used for the mod p factorizations. CPBERL implements Berkekamp's algorithm and computes the complete squarefree factorization of any monic squarefree polynomial A over GF(p). The computing time for CPBERL is $\preceq pn^3$ (assuming $L(p) \sim 1$).

For the purposes of finding a prime which gives few irreducible factors and of computing the degree sets D_p, it is not actually necessary to obtain the complete factorization modulo each of the primes tested. All that is necessary is a list of the degrees of the irreducible factors and this can be obtained from the output of Algorithm CPDDF (distinct degree factorization). Given a monic squarefree polynomial A over GF(p) to be factored, CPDDF produces a list $((d_1,a_1),\ldots,(d_s,A_s))$ where the d_i are positive integers, $d_1 < d_2 < \ldots < d_s$, and A_i is the product of all monic irreducible

factors of A which are of degree d_i. Thus $A = A_1 \ldots A_s$ and this is a complete factorization just in case no two irreducible factors of A have the same degree.

The computing time for CPDDF is $\frac{1}{3} n^3$ if $L(p) \sim 1$; hence this algorithm is practicable for much larger primes than CPBERL. Even for small primes it appears to be about 20 per cent faster, according to empirical results. Therefore this algorithm is used to obtain the lists of the degrees of irreducible factors over GF(p) for several primes in the first phase of Algorithm PFZ1, and CPBERL is applied only to the single prime selected and the reducible factors output by CPDDF for that prime.

Algorithm PFZ1(C) (Polynomial factorization based on Algorithm 2.7.1Z, 1 variable). The input C is a non-constant, primitive, squarefree, positive polynomial over Z. Two other inputs, ν and SPRIME, are required in COMMON block TR5 (COMMON/TR5/NU,SPRIME). ν is a positive Fortran integer which specifies the maximum number of primes for which mod p factorizations should be tried. SPRIME is a list of small odd positive prime integers (Fortran integers). This list should be of length no less than, say, min(20,2ν), so that it will not be exhausted in practical application (if the list is exhausted then the algorithm terminates with no output.)

The output of the algorithm, provided the list SPRIME is not exhausted, is a list F of the prime positive polynomials over Z such that $C = \Pi F$.

(1) [Initialize for factoring modulo ν different primes.] Set SP $\leftarrow$ SPRIME, RMIN $\leftarrow$ deg C + 1, NP $\leftarrow$ 1, D $\leftarrow 2^{d^*+1}-1$, where $d^* = \lfloor (\deg C)/2 \rfloor$. (D represents the set $\{0,1,\ldots,d^*\}$).

(2) [Get next prime.] If SP = ( ), stop. Otherwise, set $p \leftarrow$ first(SP), SP $\leftarrow$ tail(SP).

(3) [h_p(ldcf C) = 0?] If $p \mid$ ldcf C, go to (2).

(4) [h_p(C) squarefree?] Set $\overline{C} \leftarrow h_p(C)$, $B \leftarrow \gcd(\overline{C},\overline{C}')$. If deg B $\neq$ 0, go to (2). (We have deg B = 0 iff $\overline{C}$ is squarefree.)

(5) [Apply CPDDF.] Apply Algorithm CPDDF to the monic associate of $\overline{C}$, obtaining a list $G = ((d_1,A_1),\ldots,(d_s,A_s))$, where the d_i are positive integers, $d_1 < \ldots < d_s$, and A_i is the product of all prime monic factors of $\overline{C}$ which are of degree d_i. (Thus $\overline{C}$ = (ldcf $\overline{C}$) $A_1 \ldots A_s$.)

(6) [Construct $N = (n_1,\ldots,n_k)$, where the n_i are the degrees of the prime factors of $\overline{C}$.] Set N $\leftarrow$ ( ) and for each (d,A) on G, prefix d to N, (deg A)/d times.

(7) [$\overline{C}$ prime?] Set r $\leftarrow$ length(N). If r = 1, set F $\leftarrow$ (C) and exit.

(8) [Compute sumset.] Set S $\leftarrow$ SUMSET(N), D $\leftarrow$ D $\wedge$ last(S). If D = 1, set F $\leftarrow$ (C) and exit.

(9) [New minimum number of factors?] If r < RMIN, set RMIN $\leftarrow$ r, $p^* \leftarrow p$, $G^* \leftarrow$ G.

(10) [ν factorizations tried?] Set NP $\leftarrow$ NP + 1. If NP $\leq \nu$, go to (2).

(11) [Pick prime which yields minimum number of factors.] Set $p \leftarrow p^*$, $G \leftarrow G^*$, H $\leftarrow$ ( ).

(12) [Obtain complete factorization of $h_p(C)$.] For each (d,A) on G: if d = deg A, prefix A to H; otherwise apply Algorithm CPBERL to p and A, obtaining a list $\tilde{F}$ of the prime monic factors of A, and concatenate $\tilde{F}$ with H. Then set G ← inverse(H).

(13) [Obtain bound.] Set $d \leftarrow \lfloor \log_2 D \rfloor$ (d is then the maximum of the degrees in the set represented by D) and apply Algorithm PFB1 to C and d, obtaining C* over Z, a list F of monic linear polynomials such that $C = C^* \Pi F$ and an integer b such that $|A|_1 \leq b$ for any factor A of C* of degree $\leq d$. If deg C* = 1, prefix C* to F and exit. Otherwise, set C ← C* and, for each linear factor A on F, set $\overline{A} \leftarrow h_p(A)$ and search for and remove $\overline{A}$ from G. If length(G) = 1, prefix C to F and exit.

(14) [Compute modulus.] Set m ← p, q ← 2b(ldcf C). While m < q repeat: set m ← mp.

(15) [Apply PFC1.] Apply Algorithm PFC1 to p,m,C,G, obtaining a list $F^{(1)} = (F_1^{(1)},\ldots,F_t^{(1)})$ of polynomials over Z such that

$$C \equiv (\text{ldcf } C)\, F_1^{(1)} \ldots F_t^{(1)} \pmod m,$$

$$h_p(F_i^{(1)}) = G_i,\ \deg F_i^{(1)} = \deg G_i,$$

$F_i^{(1)}$ is monic, and $|F|_1^{(1)} \leq m/2$, $i = 1,\ldots,t$.

(16) [Apply PFP1.] Apply Algorithm PFP1 to m,C,$F^{(1)}$,D, obtaining a list $F^{(2)}$ of the prime positive polynomials over Z such that $C = \Pi F^{(2)}$. Concatenate $F^{(2)}$ with F and exit.

As we remarked earlier, if C is irreducible (and ν is sufficiently large), often only the first phase of the algorithm, steps (1)-(10), will be performed. Therefore we give separate consideration

in the following computing time analysis to the two phases.

Theorem PFZ1. a. Let $p_1 \leq p_2 \leq \ldots \leq p_\nu$ be the ν smallest odd positive prime integers such that $p_i \nmid$ ldcf C and C is squarefree modulo p_i for $1 \leq i \leq \nu$; and θ be the number of odd positive primes $\leq p_\nu$. Also, let $c = |C|_1$, $n = \deg C$, and assume $L(p_\nu)$, $L(n) \sim 1$. Then the computing time for steps (1)-(10) of Algorithm PFZ1 is

$$\preceq \nu n^3 + \theta n^2 + \theta n L(c).$$

b. Let $r = \min_{1 \leq i \leq \nu} N(p_i, C)$,

where $N(p,C)$ is the number of irreducible factors of C modulo p; k be the number of linear factors x-i of C such that $|i| \leq |n/2|$; γ and μ be defined as in Theorem PFP1 and m be the integer computed in step (13). Then the computing time for steps (11)-(16) of Algorithm PFZ1 is

$$\preceq p_\nu n^3 + (k+1)n^2[n+L(\gamma)] + \min(2^r, r^\mu)\, \mu n L(m)[nL(m) + L(\gamma)].$$

c. $p_\nu \preceq \nu + n\, L(c)$.

d. $L(m) \preceq n+L(\gamma)$.

e. Hence the time for steps (1)-(10) is

$$\preceq n^2 L(c)^2 + (n^3+\nu n)L(c) + \nu n^3,$$

and for steps (11)-(15) is

$$\preceq \nu n^3 + n^4 L(c) + \min(2^r, r^\mu)\mu n^2 (n+L(\gamma))^2.$$

and thus for the entire algorithm the time is

$$\preceq (n^4+\nu n)L(c) + \nu n^3 + \min(2^r, r^\mu)\mu n^2 (n+L(\gamma))^2.$$

f. If we assume $p_\nu \sim 1$ (i.e. if we restrict the set of inputs C to those for which p_ν is no greater than some fixed bound) then the time for steps (1)-(10) is $\preceq n^3 + n\, L(c)$ and for steps (11)-(15) is

$$\preceq \min(2^r, r^\mu)\mu n^2 (n + L(\gamma))^2;$$

the latter bound is a bound for the entire algorithm.

Remark: The assumption in f is actually made in the algorithm since the list SPRIME of small primes used by the algorithm is of finite length.

Proof: Let us first consider the time for a single execution of each step.

(1) The time to compute D is $\preceq n$ and the other parts are trivial.

(2) ~ 1.

(3) $\preceq L(c)$.

(4) The time for $h_p(C)$ is $\preceq n\ L(c)$ and for $\gcd(\overline{C},\overline{C}')$ is n^2 (using $p \leq p_\nu$ and the assumption $L(p_\nu) \sim 1$).

(5) The time to apply CPDDF is $\preceq n^3 + n^2 L(p) \preceq n^3$ (see [COL69a]).

(6),(7) $\preceq n$.

(8) $\preceq rn \preceq n^2$.

(9),(10),(11) ~ 1.

(12) The time to apply CPBERL to a polynomial of degree d over GF(p) is $\preceq pd^3$. Let $\tilde{G}_1,\ldots,\tilde{G}_t$ be the polynomials on G to which CPBERL is applied and let $\tilde{d}_i = \deg \tilde{G}_i$. Then the computing time for all applications is $\preceq p(\tilde{d}_1^3 + \ldots + \tilde{d}_t^3) \leq p(\tilde{d}_1 + \ldots + \tilde{d}_t)^3 \leq pn^3$.

(13) From Theorem PFB1, the time is $\preceq (k+1)n^2[n+L(\gamma)]$.

(14) The time to compute q is $\preceq L(b)L(c)$ and the time to compute m is $\preceq L(q)^2 \sim L(m)^2$.

(15) From Theorem PFC1, the time is $\preceq rn^2 L(m)^2 + n\ L(m)L(\gamma)$.

(16) From Theorem PFP1, the time is $\preceq \min(2^r, r^\mu)\mu n\ L(m)$ $[n\ L(m) + L(\gamma)]$.

a. Step (1) is executed only once, steps (2) and (3) are executed Θ times and step (4) at most Θ times. Steps (5)-(10) are each executed ν times. From these bounds and the above bounds for the time for each step we immediately obtain the bound claimed.

b. Immediate from the bounds for each step.

c. Let $q_1, q_2, \ldots, q_{\Theta-\nu}$ be the odd primes $< p_\nu$ such that $q_i \mid$ ldcf C or C is not squarefree modulo q_i. Letting R = res(C,C'), we see from the proof of the Corollary to Theorem 2.7.1G that each q_i divides either ldcf C or R. But actually ldcf C may be factored from the first column of the determinant expression for R, hence each q_i divides R. Therefore $q_1 \ldots q_{\Theta-\nu} \mid R$ and

$$3^{\Theta-\nu} \leq q_1 \ldots q_{\Theta-\nu} \leq |R|$$

$$\Theta-\nu \preceq L(R).$$

Using Hadamard's inequality [KNU69,p. 375] and the fact that

$$\left(\sum_0^n a_i^2\right)^{1/2} \leq \sum_0^n |a_i|,$$

we find that

$$|R| \leq n^n \, |C|_1^{2n-1},$$

hence

$$L(R) \preceq n\, L(n) + (2n-1)L(c) \preceq n\, L(c),$$

since $L(n) \sim 1$. Therefore

$$\Theta \preceq \nu + n\, L(c).$$

According to the Prime Number Theorem, the number of primes $\leq x$ is codominant with $x/\ln x$; hence $\Theta \sim p_\nu / L(p_\nu)$ and, since we are assuming $L(p_\nu) \sim 1$, we have $p_\nu \preceq \nu + n\, L(c)$.

d. We have $m > q = 2b$ (ldcf C) and either $m = p$ or $m < qp$. Hence $L(m) \leq L(q) + L(p) \leq L(q)$, since $L(p) \leq L(p_\nu) \sim 1$. Also

$$L(q) \leq L(b) + L(\text{ldcf } C) \leq L(B) + L(c),$$

and, as noted in the proof of Theorem PFB1, $L(b) \leq n + L(\gamma)$. Therefore

$$L(m) \leq n + L(\gamma) + L(c) \leq n + L(\gamma).$$

e,f. Immediate from a-d and the relations $\theta \leq p_{\nu}$, $k \leq n$.

3.5.5 Algorithm PFACT1

The final algorithm which we shall describe is PFACT1, which factors an arbitrary nonzero polynomial A over the integers. PFACT1 first factors out the content of A using Algorithm PCPP of the SAC-1 Polynomial System. The content itself is not factored into prime integers by the algorithm since this may not be necessary in some applications. The primitive part of A is factored into squarefree polynomials using Algorithm PSQFRE of the SAC-1 Rational Function Integration system. This algorithm is based on Horowitz's version of Algorithm 2.4S. Finally, Algorithm PFZ1 is used to factor each of the squarefree factors of degree > 1 into prime factors.

Algorithm PFACT1 (A) (Polynomial factorization, 1 variable). The input A must be a non-constant polynomial over Z. The output is a list $F = (c, A_1, A_2, \ldots, A_r)$, where c is the content of A and $A_1, A_2, \ldots, A_r$ are the unique prime positive factors of pp(A). Note that $c < 0$ if ldcf $A < 0$. (c is an L-integer).

(1) Using Algorithm PCPP obtain the positive content c and the positive primitive part P of A. If ldcf $A < 0$, set $c \leftarrow -c$. Then set

$F \leftarrow (c)$, $i \leftarrow 1$, $Q \leftarrow$ PSQFRE(P). (Then $Q = (Q_1,\ldots,Q_t)$, where $P = Q\,Q_2^2\ldots Q_t^t$, $\gcd(Q_i,Q_j) \sim 1$ for $i \neq j$, and each Q_i is a primitive squarefree positive polynomial.)

(2) Set $C \leftarrow$ first(Q), $Q \leftarrow$ tail(Q). If deg C = 0, go to (4). If deg C = 1, prefix C to F, i times.

(3) Set $\overline{F} \leftarrow$ PFZ1(C), and for each E on the list $\overline{F}$, prefix E to F, i times.

(4) Set $i \leftarrow i+1$. If $Q \neq (\ )$, go to (2). Otherwise, invert F and exit.

Theorem PFACT1. Let n = deg A, γ be a bound on $|B|_\infty$ for any factor B of A, and $A = Q_1Q_2^2Q_3^3\ldots Q_t^t$ be the complete squarefree factorization of A. Then the computing time for Algorithm PFACT1 is

$$\preceq tn^2L(\gamma)^2 + \sum_{i=1}^{t} T_i$$

where T_i is the computing time required to apply PFZ1 to Q_i.

Proof: Algorithm PCPP performs at most n integer greatest common divisor calculations each requiring a time $\preceq L(a)^2$ where $a = |A|_\infty \leq \gamma$; the total is $\preceq n\,L(\gamma)^2$.

Horowitz proves that the time to apply PSQFRE is $\preceq tn^2L(\gamma)^2$ in [HOR69]; the theorem now follows immediately.

3.6 Empirical results

We shall conclude by presenting several tables of empirical results obtained during testing of the algorithms described in this chapter.

Although numerous tests have been conducted, no systematic empirical study of computing times has been made. The data presented here, however, are indicative of the behavior of the algorithms and demonstrate the practicality of their application to polynomials with large degree and large coefficients.

The tables give the computing times observed when PFACT1 was applied to "random" polynomials. In order to concisely describe these polynomials, let us define $P(b,d_1,d_2,\ldots,d_k)$ to be the set of all polynomials over the integers having k irreducible factors $C_1,C_2,\ldots,C_k$ such that $|C_i|_\infty < b$ and deg $C_i = d_i$, $1 \leq i \leq k$. Polynomials in $P(b,d_1,d_2,\ldots,d_k)$ were obtained for the tests by generating polynomials C_i of degree d_i, $1 \leq i \leq k$, each with coefficients randomly chosen in the closed interval $[-b+1,b-1]$, and forming their product. (Actually a polynomial generated in this way is not necessarily in $P(b,d_1,d_2,\ldots,d_k)$, since C_i may be reducible. But the probability of C_i being reducible is very small and none of factors generated were found to be reducible by PFACT1). Note that the size of the coefficients of the products obtained in this way will be about b^k.

For example, Table 1 gives computing times for five polynomials in $P(2^7,2,3,5)$; these are polynomials of degree 10 whose

coefficients are about $2^{21} = 2 \cdot 10^6$ in size. The times are in seconds on a Univac 1108 computer.

Column I of each table gives the time required to compute the content and primitive part of the input polynomial and the squarefree factorization of the primitive part. (All of the polynomials tested were squarefree). Column II gives the time for steps (1)-(10) of PFZ1, in which the mod p factorizations are computed. Column III gives the time for steps (11)-(15), in which algorithms CPBERL, PFB1, and PFC1 are applied. (Most of this time was spent in PFH1). Column IV gives the time for step (16), the application of PFP1. The last column shows the total time.

The number in parentheses following the time in Column II is the number of different primes for which mod p factorizations were obtained. For reducible polynomials this was, of course, always equal to the value $\nu = 5$ used during all of the tests. Irreducible polynomials often required fewer than 5 primes, and thus steps (11)-(16) of PFZ1 were not executed. The variation in the number of primes used in the irreducible cases accounts for the large variations in the computing times in these cases.

Table 1

Computing Times for Polynomials of Degree 10 in $P(2^7,2,3,5)$

No.	I	II	III	IV	Total
1	0.21	1.69 (5)	5.21	0.46	7.57
2	0.19	1.61 (5)	6.65	0.49	8.94
3	0.18	1.44 (5)	4.79	0.42	6.83
4	0.19	1.48 (5)	8.43	0.71	10.81
5	0.18	1.95 (5)	5.55	0.48	8.26

Table 2

Computing Times for Polynomials of Degree 15 in $P(2^7,3,5,7)$

No.	I	II	III	IV	Total
1	0.29	4.65 (5)	14.10	0.87	19.91
2	0.26	4.58 (5)	19.25	0.90	24.99
3	0.27	4.18 (5)	19.40	1.17	23.85

Table 3

Computing Times for (Irreducible) Polynomials in $P(2^7,10)$

No.	I	II	III	IV	Total
1	0.16	1.25 (4)	---	---	1.41
2	0.18	1.65 (5)	4.25	0.17	6.24
3	0.16	0.22 (1)	---	---	0.38
4	0.19	0.75 (2)	---	---	0.94
5	0.18	0.56 (2)	---	---	0.74

Table 4

Computing Times for (Irreducible) Polynomials in $P(2^7,15)$

No.	I	II	III	IV	Total
1	0.22	2.80 (4)	---	---	3.02
2	0.23	4.23 (5)	9.65	0.18	14.29
3	0.22	1.06 (2)	---	---	1.28
4	0.24	4.30 (5)	9.85	0.12	14.51
5	0.24	2.48 (3)	---	---	2.72

Table 5

Computing Times for (Irreducible) Polynomials in $P(2^7,20)$

No.	I	II	III	IV	Total
1	0.30	3.09 (2)	---	---	3.39
2	0.28	3.56 (3)	---	---	3.84
3	0.31	6.34 (5)	20.71	0.23	27.36
4	0.32	8.36 (5)	16.28	0.23	25.19
5	0.32	7.87 (5)	18.90	0.32	27.41

Table 6

Computing Times for (Irreducible) Polynomials in $P(2^{20},10)$

No.	I	II	III	IV	Total
1	0.16	0.47 (2)	---	---	0.63
2	0.17	1.66 (5)	6.03	0.18	8.04
3	0.18	1.45 (5)	8.54	0.22	10.39
4	0.17	0.43 (2)	---	---	0.70
5	0.20	0.59 (2)	---	---	0.79

Table 7

Computing Times for (Irreducible) Polynomials in $P(2^{20},15)$

No.	I	II	III	IV	Total
1	0.24	2.48 (3)	---	---	2.72
2	0.26	3.12 (5)	18.20	0.22	21.80
3	0.24	1.04 (2)	---	---	1.28
4	0.26	0.77 (1)	---	---	1.03
5	0.24	1.32 (2)	---	---	1.56

Table 8

Computing Times for (Irreducible) Polynomials in $P(2^{20},20)$

No.	I	II	III	IV	Total
1	0.53	5.63 (4)	---	---	6.16
2	0.56	5.95 (4)	---	---	6.51
3	0.58	4.53 (4)	---	---	5.11
4	0.53	2.96 (2)	---	---	3.49
5	0.60	1.32 (1)	---	---	1.92

REFERENCES

[BER68] E. R. Berlekamp, Algebraic Coding Theory, McGraw-Hill, Inc., New York, 1968.

[BER70] E. R. Berlekamp, Factoring Polynomials over Large Finite Fields, Mathematics of Computation, November, 1970.

[COL67] George E. Collins, The SAC-1 List Processing System, University of Wisconsin Computing Center Technical Report, July, 1967.

[COL68a] George E. Collins and James R. Pinkert, The Revised SAC-1 Integer Arithmetic System, University of Wisconsin Computing Center Technical Report No. 9, Nov., 1968.

[COL68b] George E. Collins, The SAC-1 Polynomial System, University of Wisconsin Technical Report No. 2, March, 1968.

[COL69a] George E. Collins, L. E. Heindel, E. Horowitz, M. T. McClellan, and D. R. Musser, The SAC-1 Modular Arithmetic System, University of Wisconsin Technical Report No. 10, June, 1969.

[COL69b] George E. Collins, Computing Time Analyses for Some Arithmetic and Algebraic Algorithms, Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation (Robert G. Tobey, ed.), IBM Federal Systems Center, June, 1968, pp. 195-232.

[COL71] George E. Collins, The Calculation of Multivariate Polynomial Resultants, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March, 1971.

[COL71a] George E. Collins, The SAC-1 System: an Introduction and Survey, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March, 1971.

[FEL68] William Feller, An Introduction to Probability Theory and its Applications, Vol. 1, Wiley & Sons, Inc., New York, 1968.

[FLO67] Robert W. Floyd, Assigning Meanings to Programs, Proceedings of a Symposium in Applied Mathematics, Vol. 19 - Mathematical Aspects of Computer Science (J. T. Schwartz, ed.). American Mathematical Society, Providence, R. I., 1967, pp. 19-32.

[GAU62] Walter Gautschi, On inverses of Vandermonde and confluent Vandermonde matrices, Numerische Mathematik 4, 117-123 (1962).

[GOL70] Jacob K. Goldhaber and Gertrude Ehrlich, Algebra, Macmillan Company, 1970.

[HOR69] Ellis Horowitz, Algorithms for Symbolic Integration of Rational Functions, Ph.D. Thesis, Computer Sciences Dept., Univ. of Wisconsin, 1969.

[JOH66] S. C. Johnson, Tricks for Improving Kronecker's Method, Bell Laboratories Report, 1966.

[KNU68] Donald E. Knuth, The Art of Computer Programming, Vol I: Fundamental Algorithms, Addison-Wesley Publishing Co., Reading, Mass., 1968.

[KNU69] Donald E. Knuth, The Art of Computer Programming, Vol. II: Seminumerical Algorithms, Addison-Wesley Publishing Co., Reading, Mass., 1968.

[LAN65] Serge Lang, Algebra, Addison-Wesley Publishing Co., Reading, Mass., 1965.

[RAB60] Michael O. Rabin, Computable Algebra, General Theory and Theory of Computable Fields, Transactions of the American Mathematical Society 95 (1960), pp. 341-360.

[VDW49] B. L. Van der Waerden, Modern Algebra, Vol. 1, trans. by Fred Blum, Frederick Ungar Publishing Co., New York, 1949.

[ZAS69] Hans Zassenhaus, On Hensel Factorization, I, Journal of Number Theory 1, 291-311 (1969).

Appendix: FORTRAN PROGRAM LISTINGS

```
      SUBROUTINE GEN(STACK,NN,SS,NO,K,JJ)
      INTEGER STACK,SS,SP,R,TSP,XXX
      INTEGER FIRST,PFA,TAIL,CINV
C  GENERATION OF SUM INDEX LISTS.
 1    IF (STACK .EQ. 0) GO TO 2
      CALL DECAP(J,STACK)
      N = 0
      GO TO 6
C  INITIALIZE.
 2    J = LENGTH(NN)
      JJ = 0
      N = NO
      NP = CINV(NN)
      SP = CINV(SS)
      STACK = PFA(SP,PFA(NP,STACK))
      R = 1
 25   IF (J .LE. K) GO TO 3
      NP = TAIL(NP)
      SP = TAIL(SP)
      J = J-1
      GO TO 25
C  STEPS 3 THROUGH 9 CORRESPOND TO A RECURSIVE PROCEDURE G(J,N,JJ).
C  STEP 2 HAS INITIALIZED TO PERFORM THE PROCEDURE WITH J=K AND JJ=0.
C
C  CHECK IF A SUM INDEX LIST HAS BEEN GENERATED.
 3    IF (N .NE. 0) GO TO 4
      STACK = PFA(J,STACK)
      RETURN
C  THIS EXIT ALLOWS THE CURRENT VALUE OF JJ TO BE USED
C  OUTSIDE THE SUBROUTINE.  THE SUBROUTINE MAY BE REENTERED
C  TO GENERATE ANOTHER SUM INDEX LIST, PROVIDED THAT
C  NEITHER STACK NOR JJ IS ALTERED OUTSIDE THE SUBROUTINE.
C
C  CHECK IF RECURSION IS NECESSARY.
 4    TSP = TAIL(SP)
      IF (MEMBER(N-FIRST(NP),FIRST(TSP)).EQ. 0) GO TO 7
C  CALL RECURSIVE PROCEDURE G(J-1,N-NNG),PREFIX(J,JJ)).
 5    STACK = PFA(R,PFA(SP,PFA(NP,STACK)))
      N = N-FIRST(NP)
      NP = TAIL(NP)
      SP = TSP
      R = 2
      JJ = PFA(J,JJ)
      J = J-1
      GO TO 3
 6    CALL DECAP(R,STACK)
      CALL DECAP(SP,STACK)
      CALL DECAP(NP,STACK)
```

```
      J = J+1
      N = N+FIRST(NP)
      CALL DECAP(XXX,JJ)
      TSP = TAIL(SP)
C  CHECK IF RECURSION IS NECESSARY.
 7    IF (MEMBER(N,FIRST(TSP)) .EQ. 0) GO TO 99
C  CALL RECURSIVE PROCEDURE G(J-1,N,JJ)
 8    STACK = PFA(R,PFA(SP,PFA(NP,STACK)))
      NP = TAIL(NP)
      SP = TSP
      R = 3
      J = J-1
      GO TO 3
 9    CALL DECAP(R,STACK)
      CALL DECAP(SP,STACK)
      CALL DECAP(NP,STACK)
      J = J + 1
 99   GO TO (10,6,9), R
C  THE FOLLOWING EXIT IS TAKEN WHEN ALL SUM INDEX LISTS
C  HAVE BEEN GENERATED.
 10   CALL DECAP(SP,STACK)
      CALL DECAP(NP,STACK)
      CALL ERLA(SP)
      CALL ERLA(NP)
      RETURN
      END


      INTEGER FUNCTION IAND(A,B)
      INTEGER A,B
      INTEGER C,FIRST,INV,PFA,TAIL,T1,T2,T3
C  IT IS ASSUMED THAT BETA (THE BASE OF THE SAC-1 INTEGER ARITH.
C  SYSTEM) IS A POWER OF 2.
 1    C = 0
      T1 = A
      T2 = B
 2    IF (T1 .EQ. 0 .OR. T2 .EQ. 0) GO TO 3
      T3 = AND(FIRST(T1),FIRST(T2))
      C = PFA(T3,C)
      T1 = TAIL(T1)
      T2 = TAIL(T2)
      GO TO 2
 3    IAND = INV(C)
      RETURN
      END
```

```
      INTEGER FUNCTION ILS(L,N)
      INTEGER IPROD,PFA,Q,R,T,WL
C  IT IS ASSUMED THAT BETA (THE BASE OF THE SAC-1 INTEGER ARITH.
C  SYSTEM) IS A POWER OF 2.
      WL = 33
C  WL = BASE 2 LOGARITHM OF BETA.
1     Q = N/WL
      R = N-Q*WL
      T = PFA(2**R,0)
      ILS = IPROD(L,T)
      CALL ERLA(T)
2     IF (Q .EQ. 0) RETURN
      ILS = PFA(0,ILS)
      Q = Q-1
      GO TO 2
      END


      INTEGER FUNCTION IOR(A,B)
      INTEGER A,B
      INTEGER BORROW,C,FIRST,INV,PFA,TAIL,T1,T2,T3
C  IT IS ASSUMED THAT BETA (THE BASE OF THE SAC-1 INTEGER ARITH.
C  SYSTEM) IS A POWER OF 2.
1     C = 0
      T1 = A
      T2 = B
2     IF (T1 .EQ. 0) GO TO 4
      IF (T2 .EQ. 0) GO TO 3
      T3 = OR(FIRST(T1),FIRST(T2))
      C = PFA(T3,C)
      T1 = TAIL(T1)
      T2 = TAIL(T2)
      GO TO 2
3     T2 = T1
4     IOR = INV(C)
      CALL SSUCC(BORROW(T2),C)
      RETURN
      END


      FUNCTION LAST(L)
      INTEGER T,TAIL,FIRST
 1    M = L
      IF (L .EQ. 0) GO TO 4
 2    T = TAIL(M)
      IF (T .EQ. 0) GO TO 3
      M = T
      GO TO 2
 3    LAST = FIRST(M)
      RETURN
 4    LAST = M
      RETURN
      END
```

```
      INTEGER FUNCTION MEMBER(N,S)
      INTEGER N,S
      INTEGER FIRST,Q,R,T,TAIL,T1,WL
C  IT IS ASSUMED THAT BETA (THE BASE OF THE SAC-1 INTEGER ARITH.
C  SYSTEM) IS A POWER OF 2.
      WL = 33
C  WL = BASE 2 LOGARITHM OF BETA.
1     IF (N .LT. 0) GO TO 4
      T = S
      Q = N/WL
      R = N-Q*WL
2     IF (T .EQ. 0) GO TO 4
      IF (Q.EQ. 0) GO TO 3
      T = TAIL(T)
      Q = Q-1
      GO TO 2
3     T1 = FIRST(T)/2**R
      MEMBER = T1-(T1/2)*2
      RETURN
4     MEMBER = 0
      RETURN
      END


      INTEGER FUNCTION MMOD(M,A)
      INTEGER M,A
      INTEGER ICOMP,IDIF,IREM,ISIGNL,ISUM,R,R1,TR
1     R = IREM(A,M)
      IF (R .EQ. 0) GO TO 3
      IF (ISIGNL(R) .GT. 0) GO TO 2
      R1 = ISUM(R,M)
      CALL ERLA(R)
      R = R1
2     TR = ISUM(R,R)
      IF (ICOMP(TR,M) .LT. 0) GO TO 25
      R1 = IDIF(R,M)
      CALL ERLA(R)
      R = R1
25    CALL ERLA(TR)
3     MMOD = R
      RETURN
      END
```

```
      FUNCTION MPLPR(M,A)
      INTEGER A,AA,B,FA,T,BORROW,FIRST,PFA,PPROD,TAIL
1     IF (A .NE. 0) GO TO 15
      MPLPR = PFA(1,0)
      RETURN
15    B = BORROW(FIRST(A))
      AA = TAIL(A)
2     IF (AA .EQ. 0) GO TO 4
3     CALL ADV(FA,AA)
      T = PPROD(B,FA)
      CALL PERASE(B)
      B = MPMOD(M,T)
      GO TO 2
4     MPLPR = B
      RETURN
      END


      INTEGER FUNCTION MPMOD(M,A)
      INTEGER M,A
      INTEGER AA,B,C,D,E,INV,MMOD,PFA,PFL,PVBL,TAIL,TYPE
1     B = 0
      IF (A .EQ. 0) GO TO 4
      IF (TYPE(A) .NE. 0) GO TO 15
      B = MMOD(M,A)
      GO TO 4
15    AA = TAIL(A)
2     CALL ADV(C,AA)
      CALL ADV(E,AA)
      D = MMOD(M,C)
      IF (D .NE. 0) B = PFA(E,PFL(D,B))
      IF (AA .NE. 0 ) GO TO 2
3     IF (B .NE. 0) B = PFL(PVBL(A),INV(B))
4     MPMOD = B
      RETURN
      END
```

```
      INTEGER FUNCTION MPQREM(M,A,B)
      INTEGER M,A,B
      INTEGER BD,BL,BORROW,BR,C,D,INV,IPROD,J,MMOD,MPMOD
      INTEGER MRECIP,PDEG,PDIF,PFA,PFL,PLDCF,PRED,PSPROD,PVBL,Q
      INTEGER R,RD,RL,RR,TEMP,TEMP1
1     R = BORROW(A)
      BL = PLDCF(B)
      C = MRECIP(M,BL)
      CALL ERLA(BL)
      Q = 0
      BR = PRED(B)
      BD = PDEG(B)
2     RD = PDEG(R)
      J = RD-BD
      IF (J .LT. 0 .OR. R .EQ. 0) GO TO 3
      RL = PLDCF(R)
      TEMP = IPROD(C,RL)
      D = MMOD(M,TEMP)
      CALL ERLA(TEMP)
      TEMP = PSPROD(BR,D,J)
      RR = PRED(R)
      CALL PERASE(R)
      TEMP1 = PDIF(RR,TEMP)
      R = MPMOD(M,TEMP1)
      CALL PERASE(TEMP1)
      CALL PERASE(TEMP)
      CALL PERASE(RR)
      CALL ERLA(RL)
      Q = PFA(J,PFL(D,Q))
      GO TO 2
3     IF (Q .NE. 0) Q = PFL(PVBL(A),INV(Q))
      MPQREM = PFL(Q,PFL(R,0))
      CALL PERASE(BR)
      CALL ERLA(C)
      RETURN
      END
```

```
      SUBROUTINE MPSPEQ(M,A,B,S,T,U,Y,Z)
      INTEGER M,A,B,S,T,U,Y,Z
      INTEGER MPMOD,MPQREM,PPROD,PSUM,Q,TEMP,TEMP1,TEMP2,V,W
 1    W = MPMOD(M,U)
      TEMP = PPROD(T,W)
      V = MPMOD(M,TEMP)
      CALL PERASE(TEMP)
 2    TEMP = MPQREM(M,V,A)
      CALL DECAP(Q,TEMP)
      CALL DECAP(Z,TEMP)
      CALL PERASE(V)
 3    TEMP = PPROD(S,W)
      TEMP1 = PPROD(B,Q)
      TEMP2 = PSUM(TEMP,TEMP1)
      CALL PERASE(TEMP1)
      CALL PERASE(TEMP)
      CALL PERASE(Q)
      Y = MPMOD(M,TEMP2)
      CALL PERASE(TEMP2)
      CALL PERASE(W)
      RETURN
      END

      INTEGER FUNCTION MRECIP(M,X)
      INTEGER M,X
      INTEGER A1,A2,A3,BORROW,FIRST,IDIF,IPROD,IQR,ISIGNL,ISUM
      INTEGER PFA,Q,TAIL,TEMP,Y1,Y2,Y3
      A1 = BORROW(M)
      A2 = BORROW(X)
      IF (ISIGNL(A2) .GT. 0) GO TO 5
      TEMP = ISUM(M,A2)
      CALL ERLA(A2)
      A2 = TEMP
 5    Y1= 0
      Y2 = PFA(1,0)
      GO TO 2
 1    TEMP = IQR(A1,A2)
      CALL DECAP(Q,TEMP)
      CALL DECAP(A3,TEMP)
      TEMP = IPROD(Y2,Q)
      Y3 = IDIF(Y1,TEMP)
      CALL ERLA(TEMP)
      CALL ERLA(Q)
      CALL ERLA(A1)
      A1 = A2
      A2 = A3
      CALL ERLA(Y1)
      Y1 = Y2
      Y2 = Y3
 2    IF (FIRST(A2)  .NE. 1 .OR. TAIL(A2) .NE. 0) GO TO 1
 4    MRECIP = Y2
      CALL ERLA(A1)
      CALL ERLA(A2)
      CALL ERLA(Y1)
      RETURN
      END
```

```
      INTEGER FUNCTION PFACT1(A)
      INTEGER A
      INTEGER BORROW,C,CONT,D,E,F,FF,FIRST,I,INEG,INV,ISIGNL,J
      INTEGER PCPP,PDEG,PFL,PFZ1,PP,PSQFRE,Q,TAIL,TEMP
1     TEMP = PCPP(A)
      CALL DECAP(CONT,TEMP)
      CALL DECAP(PP,TEMP)
      IF (ISIGNL(FIRST(TAIL(A))) .GT. 0) GO TO 15
      TEMP = INEG(CONT)
      CALL ERLA(CONT)
      CONT = TEMP
15    FF = PFL(CONT,0)
      Q = PSQFRE(PP)
      CALL PERASE(PP)
      I = 1
2     CALL DECAP(C,Q)
      D = PDEG(C)
      IF (D .EQ. 0) GO TO 28
      IF (D .GT. 1) GO TO 24
      DO 22 J = 1,I
22    FF = PFL(BORROW(C),FF)
      GO TO 28
24    F = PFZ1(C)
25    CALL DECAP(E,F)
      DO 26 J = 1,I
26    FF = PFL(BORROW(E),FF)
      CALL PERASE(E)
      IF (F .NE. 0) GO TO 25
28    CALL PERASE(C)
      I = I+1
      IF (Q .NE. 0) GO TO 2
      PFACT1 = INV(FF)
      RETURN
      END
```

```
      SUBROUTINE PFB1(C0,M,C,L,B)
      INTEGER C0,C,B,V,T,F,AV,V1,V2,AV1,AV2,T1,T2
      INTEGER BORROW,PDEG,PNORMF,PTLCF,PFL,PVBL,PFA,PQ,PSUBST
1     C = BORROW(C0)
      L = 0
2     IF (PDEG(C) .NE. 1) GO TO 3
      B = PNORMF(C)
      RETURN
3     V = PTLCF(C)
      IF (V .NE. 0) GO TO 5
      J = 0
4     LF = PFL(PVBL(C),PFL(PFA(1,0),PFA(1,PFL(PFA(-J,0),PFA(0,0)))))
      T = PQ(C,LF)
      CALL PERASE(C)
      C = T
      L = PFL(LF,L)
      GO TO 2
5     F = 4
      T = PFA(F,0)
      AV = IABSL(V)
      CALL ERLA(V)
      B = IPROD(AV,T)
      CALL ERLA(AV)
      CALL ERLA(T)
      J = 1
6     IF (J .GT. M) RETURN
7     T = PFA(J,0)
      V1 = PSUBST(T,C)
      CALL ERLA(T)
      IF (V1 .EQ. 0) GO TO 4
8     T = PFA(-J,0)
      V2 = PSUBST(T,C)
      CALL ERLA(T)
      IF (V2 .NE. 0) GO TO 9
      J = -J
      GO TO 4
9     K1 = (M-J+1) * F
      K2 = M + J
      F = K1 / K2
      IF (K1 .NE. F*K2) F = F+1
      T = PFA((J+1)*F,0)
      AV1 = IABSL(V1)
      CALL ERLA(V1)
```

```
      AV2 = IABSL(V2)
      CALL ERLA(V2)
      T1 = ISUM(AV1,AV2)
      T2 = IPROD(T1,T)
      CALL ERLA(AV1)
      CALL ERLA(AV2)
      CALL ERLA(T1)
      CALL ERLA(T)
      T = PFA(J*J+1,0)
      T1 = IQRS(T2,T)
      CALL ERLA(T)
      CALL ERLA(T2)
      CALL DECAP(T,T1)
      CALL DECAP(T2,T1)
      IF (T2 .EQ. 0) GO TO 95
      CALL ERLA(T2)
      T1 = PFA(1,0)
      T2 = ISUM(T,T1)
      CALL ERLA(T)
      T = T2
      CALL ERLA(T1)
95    T1 = ISUM(B,T)
      CALL ERLA(B)
      B = T1
      CALL ERLA(T)
      J = J +1
      GO TO 6
      END
```

```
      INTEGER FUNCTION PFC1(P,M,C0,G)
      INTEGER P,M,C0,G
      INTEGER A,AB,B,BB,BORROW,C,CB,CPEGCD,CPMOD,CPQREM,CRECIP,CSPROD
      INTEGER F,FIRST,GS,INV,LC,LCI,MPMOD,MRECIP,PFL,PIP
      INTEGER PLDCF,SB,TAIL,TB,TEMP,W,WI,XXX
1     C = BORROW(C0)
      F = 0
      CB = CPMOD(P,C)
2     GS = G
21    IF (TAIL(GS) .EQ. 0) GO TO 3
      AB = FIRST(GS)
      TEMP = CPQREM(P,CB,AB)
      CALL DECAP(BB,TEMP)
      CALL DECAP(XXX,TEMP)
      IF (FIRST(AB) .GE. FIRST(BB)) GO TO 22
      TEMP = CPEGCD(P,BB,AB)
      CALL DECAP(TB,TEMP)
      CALL DECAP(SB,TEMP)
      GO TO 23
22    TEMP = CPEGCD(P,AB,BB)
      CALL DECAP(SB,TEMP)
      CALL DECAP(TB,TEMP)
23    CALL DECAP(W,TEMP)
      WI = CRECIP(P,W)
      TEMP = CSPROD(P,SB,WI,0)
      CALL ERLA(SB)
      SB = TEMP
      TEMP = CSPROD(P,TB,WI,0)
      CALL ERLA(TB)
      TB = TEMP
      CALL PFH1(P,M,C,AB,BB,SB,TB,A,B)
      F = PFL(A,F)
      CALL PERASE(C)
      C = B
      CALL ERLA(CB)
      CB = BB
      CALL ERLA(SB)
      CALL ERLA(TB)
      GS = TAIL(GS)
      GO TO 21
3     LC = PLDCF(C)
      LCI = MRECIP(M,LC)
      CALL ERLA(LC)
      TEMP = PIP(C,LCI)
      CALL ERLA(LCI)
      CALL PERASE(C)
      C = MPMOD(M,TEMP)
      CALL PERASE(TEMP)
      PFC1 = INV(PFL(C,F))
      CALL ERLA(CB)
      RETURN
      END
```

```
      SUBROUTINE PFH1(P,M,C,AB,BB,SB,TB,A,B)
      INTEGER P,M,C,AB,BB,SB,TB,A,B
      INTEGER AS,BORROW,BS,CPGARN,IC,ICOMP,IPROD,IQ,MU1,ONE,PDIF,PFA,PFI
      INTEGER PIP,PPROD,PSQ,PSUM,PVBL,Q,Q2,S,SS,T,TEMP,TEMP1,TEMP2
      INTEGER TS,U,V,Y,Y1,Z,Z1,AT,BT,ST,TT,QT
1     Q = PFA(P,0)
      Q2 = 0
      V = PFL(PVBL(C),0)
      ONE = PFA(1,0)
      A = CPGARN(ONE,0,P,AB,V)
      B = CPGARN(ONE,0,P,BB,V)
      S = CPGARN(ONE,0,P,SB,V)
      T = CPGARN(ONE,0,P,TB,V)
      CALL ERASE(V)
      CALL ERLA(ONE)
      ONE = PFL(PVBL(C),PFL(PFA(1,0),PFA(0,0)))
2     IF (ICOMP(Q,M) .LT. 0) GO TO 3
21    CALL ERLA(Q)
      CALL ERLA(Q2)
      CALL PERASE(ONE)
      CALL PERASE(S)
      CALL PERASE(T)
      RETURN
3     TEMP = PPROD(A,B)
      TEMP1 = PDIF(C,TEMP)
      CALL PERASE(TEMP)
      U = PSQ(TEMP1,Q)
      CALL PERASE(TEMP1)
      Q2 = IPROD(Q,Q)
      IC = ICOMP(Q2,M)
      IF (IC .LE. 0) GO TO 32
      QT = IQ(M,Q)
      AT = MPMOD(QT,A)
      BT = MPMOD(QT,B)
      ST = MPMOD(QT,S)
      TT = MPMOD(QT,T)
      CALL MPSPEQ(QT,AT,BT,ST,TT,U,Y,Z)
      CALL PERASE(AT)
      CALL PERASE(BT)
      CALL PERASE(ST)
      CALL PERASE(TT)
      CALL ERLA(QT)
      GO TO 34
32    CALL MPSPEQ(Q,A,B,S,T,U,Y,Z)
34    CALL PERASE(U)
4     TEMP = PIP(Z,Q)
      CALL PERASE(Z)
```

```
      AS = PSUM(A,TEMP)
      CALL PERASE(TEMP)
      TEMP = PIP(Y,Q)
      CALL PERASE(Y)
      BS = PSUM(B,TEMP)
      CALL PERASE(TEMP)
      IF (IC .LT. 0) GO TO 5
      CALL PERASE(A)
      A = AS
      CALL PERASE(B)
      B = BS
      CALL PERASE(Q2)
      GO TO 21
5     TEMP = PPROD(AS,S)
      TEMP1 = PPROD(BS,T)
      TEMP2= PSUM(TEMP,TEMP1)
      CALL PERASE(TEMP1)
      CALL PERASE(TEMP)
      TEMP = PDIF(ONE,TEMP2)
      MU1 = PSQ(TEMP,Q)
      CALL PERASE(TEMP2)
      CALL PERASE(TEMP)
      CALL MPSPEQ(Q,A,B,S,T,MU1,Y1,Z1)
      CALL PERASE(MU1)
6     TEMP = PIP(Y1,Q)
      CALL PERASE(Y1)
      SS = PSUM(S,TEMP)
      CALL PERASE(TEMP)
      TEMP = PIP(Z1,Q)
      CALL PERASE(Z1)
      TS = PSUM(T,TEMP)
      CALL PERASE(TEMP)
7     CALL ERLA(Q)
      Q = Q2
      CALL PERASE(A)
      CALL PERASE(B)
      CALL PERASE(S)
      CALL PERASE(T)
      A = AS
      B = BS
      S = SS
      T = TS
      GO TO 2
      END
```

```
      INTEGER FUNCTION PFP1(M,CC0,GG0,DS)
      INTEGER M,CC0,GG0,DS
      INTEGER  A,AA,AAS,BBS,BORROW,C,CC,CCS,D,DEGSET,FF,FIRST
      INTEGER  GG,GGJJ,GGS,I,INV,IPROD,IREM,JJ,JJP,K,KK,LENGTH,IAND
      INTEGER MMOD,MPMOD,NN,PDEG,PFA,PFL
      INTEGER PIP,PLDCF,PLPROD,PPP,PQ,PSQ,PTLCF,REM,SELECT,SS
      INTEGER STACK,SUMSET,T,TAIL,TCCS,TEMP,TEMP1,TT,TTJJ,R,V
      INTEGER OUT
1     CC = BORROW(CC0)
      GG = BORROW(GG0)
      FF = 0
      D = 1
      NN = 0
      TT = 0
      GGS = GG
1'    CALL ADV(AA,GGS)
      NN = PFA(PDEG(AA),NN)
      TT = PFL(PTLCF(AA),TT)
      IF (GGS .NE. 0) GO TO 15
      NN = INV(NN)
      TT = INV(TT)
      R = LENGTH(GG)
      K = R
2     C = PLDCF(CC)
      CCS = PIP(CC,C)
      TCCS = PTLCF(CCS)
      SS = SUMSET(NN)
      DEGSET = IAND(DS,LAST(SS))
3     IF (D .LE. PDEG(CC)/2) GO TO 35
      FF = PFL(CC,FF)
      CALL PERASE(GG)
      CALL ERLA(NN)
      CALL ERASE(TT)
      CALL ERLA(C)
      CALL ERLA(TCCS)
      CALL PERASE(CCS)
      CALL ERASE(SS)
      CALL ERLA(DEGSET)
      PFP1 = INV(FF)
      RETURN
35    IF (MEMBER(D,DEGSET) .EQ. 0 .OR. K .EQ. 0) GO TO 7
4     STACK = 0
41    CALL GEN(STACK,NN,SS,D,K,JJ)
      IF (JJ .EQ. 0) GO TO 7
5     TTJJ = SELECT(TT,JJ)
      TEMP = PFL(BORROW(C),TTJJ)
      T = MPLPR(M,TEMP)
      CALL ERASE(TEMP)
      REM = IREM(TCCS,T)
      CALL ERLA(T)
```

```
      IF (REM .EQ. 0) GO TO 6
      CALL ERLA(REM)
      GO TO 41
6     GGJJ = SELECT(GG,JJ)
      TEMP = PFL(PFL(PVBL(CC),PFL(BORROW(C),PFA(0,0))),GGJJ)
      AAS = MPLPR(M,TEMP)
      CALL ERASE(TEMP)
      BBS = PQ(CCS,AAS)
      IF (BBS .GT. 0) GO TO 8
      CALL PERASE(AAS)
      GO TO 41
      D = D+1
      K = R
      GO TO 3
8     AA = PPP(AAS)
      CALL PERASE(AAS)
      FF = PFL(AA,FF)
      CALL PERASE(CC)
      A = PLDCF(AA)
      CC = PSQ(BBS,A)
      CALL ERLA(A)
      CALL PERASE(BBS)
      JJP = JJ
      KK = 0
      DO 96 I = 1,R
      IF (JJP .EQ. 0) GO TO 95
      IF (I .NE. FIRST(JJP)) GO TO 95
      JJP = TAIL(JJP)
      GO TO 96
95    KK = PFA(I,KK)
96    CONTINUE
      KK = INV(KK)
      V = LENGTH(JJ)
      JV = LAST(JJ)
      CALL ERLA(JJ)
      CALL ERLA(STACK)
      TEMP = SELECT(GG,KK)
      CALL ERASE(GG)
      GG = TEMP
      TEMP = SELECT(NN,KK)
      CALL ERASE(NN)
      NN = TEMP
      TEMP = SELECT(TT,KK)
      CALL ERASE(TT)
      TT = TEMP
      R = R - V
      K = JV - V
      CALL ERLA(KK)
      CALL ERLA(C)
      CALL ERLA(TCCS)
      CALL PERASE(CCS)
      CALL ERASE(SS)
      CALL ERLA(DEGSET)
      GO TO 2
      END
```

```
      INTEGER FUNCTION PFZ1(C0)
      INTEGER C,C0
      INTEGER B,BORROW,CB,CBP,CB1,CMOD,CMONIC,CONC,CPBERL,CPDDF,CPDRV
      INTEGER CPGCD1,CPMOD,D,DB,DFD,DS,F,FD,FDF,FIRST,F1,G,GP,GS,GI,H
      INTEGER IDIF,INV,IPROD,ISUM,K,KD,KP,LENGTH,M,IAND,ILS,MU,NN,NP
      INTEGER ONE,P,PDEG,PFA,PFBND,PFC1,PFL,PFP1,PONE,PS,Q,R
      INTEGER RMIN,SP,SPOWER,SPRIME,SS,SUMSET,TAIL,TEMP,TEMP1
      INTEGER CS,FP,A,AP,PP
      COMMON /TR5/ NU,SPRIME
1     SP = SPRIME
      C = BORROW(C0)
      RMIN = PDEG(C)+1
      GS = 0
      NP = 1
      ONE = PFA(1,0)
      TEMP = ILS(ONE,PDEG(C)/2+1)
      DS = IDIF(TEMP,ONE)
      CALL ERLA(ONE)
      CALL ERLA(TEMP)
2     IF (SP .EQ. 0) STOP
      CALL ADV(P,SP)
3     IF (CMOD(P,FIRST(TAIL(C))) .EQ. 0) GO TO 2
4     CB = CPMOD(P,C)
      CBP = CPDRV(P,CB)
      IF (CBP .NE. 0) GO TO 44
42    CALL ERLA(CB)
      GO TO 2
44    B = CPGCD1(P,CB,CBP)
      DB = FIRST(B)
      CALL ERLA(B)
      CALL ERLA(CBP)
      IF (DB .NE. 0) GO TO 42
5     CB1 = CMONIC(P,CB)
      CALL ERLA(CB)
      G = CPDDF(P,CB1)
      CALL ERLA(CB1)
6     NN = 0
      GP = G
61    CALL ADV(DFD,GP)
      CALL ADV(D,DFD)
      CALL ADV(FD,DFD)
      KD = FIRST(FD)/D
      DO 62 K = 1,KD
62    NN = PFA(D,NN)
      IF (GP .NE. 0) GO TO 61
7     R = LENGTH(NN)
      IF (R .NE. 1) GO TO 8
      CALL ERLA(NN)
71    F = PFL(BORROW(C),0)
      CALL ERASE(G)
      CALL ERASE(GS)
```

```
      GO TO 99
8     SS = SUMSET(NN)
      TEMP = IAND(DS,LAST(SS))
      CALL ERLA(DS)
      DS = TEMP
      CALL ERASE(SS)
      CALL ERLA(NN)
      IF (PONE(DS) .EQ. 1) GO TO 71
9     IF (R .GE. RMIN) GO TO 95
      RMIN = R
      PS = P
      CALL ERASE(GS)
      GS = G
      GO TO 10
95    CALL ERASE(G)
10    NP = NP+1
      IF (NP .LE. NU) GO TO 2
11    P = PS
      G = GS
      H = 0
12    CALL DECAP(DFD,G)
      CALL DECAP(D,DFD)
      CALL DECAP(FD,DFD)
      IF (D .NE. FIRST(FD)) GO TO 122
      H = PFL(FD,H)
      GO TO 125
122   FDF = CPBERL(P,FD)
      H = CONC(FDF,H)
      CALL ERLA(FD)
125   IF (G .NE. 0) GO TO 12
      G = INV(H)
13    CALL ELPOF2(DS,D,XXX)
      CALL PFB1(C,D,CS,F,B)
      IF (PDEG(CS) .NE. 1) GO TO 132
      F = PFL(CS,F)
131   CALL ERLA(B)
      CALL ERASE(G)
      GO TO 99
132   CALL PERASE(C)
      C = CS
      FP = F
133   IF (FP .EQ. 0) GO TO 135
      CALL ADV(A,FP)
      AP = CPMOD(P,A)
      H = 0
134   CALL DECAP(GI,G)
      IF (ICOMP(GI,AP) .NE. 0) H = PFL(GI,H)
      IF (G .NE. 0) GO TO 134
      CALL ERLA(AP)
      G = INV(H)
      GO TO 133
```

```
135   IF (TAIL(G) .NE. 0) GO TO 14
      F = PFL(C,F)
      GO TO 131
14    PP = PFA(P,0)
      M = BORROW(PP)
      TEMP = IPROD(B,FIRST(TAIL(C)))
      Q = ISUM(TEMP,TEMP)
      CALL ERLA(TEMP)
142   IF(ICOMP(M,Q) .GE. 0) GO TO 145
      TEMP = IPROD(M,PP)
      CALL ERLA(M)
      M = TEMP
      GO TO 142
145   CALL ERLA(PP)
15    F1 = PFC1(P,M,C,G)
      CALL ERASE(G)
16    F = CONC(PFP1(M,C,F1,DS),F)
      CALL ERLA(M)
      CALL ERASE(F1)
99    PFZ1 = F
      CALL ERLA(DS)
      CALL PERASE(C)
      RETURN
      END


      INTEGER FUNCTION PTLCF(P)
      INTFGER P
      INTEGER BORROW,FIRST,Q,R,TAIL,TYPE
1     Q = P
      IF (Q .EQ. 0) GO TO 25
      IF (TYPE(Q) .EQ. 0) GO TO 4
      Q = TAIL(Q)
2     R = TAIL(Q)
      IF (FIRST(R) .EQ. 0) GO TO 3
      Q = TAIL(R)
      IF (Q .NE. 0) GO TO 2
25    PTLCF = 0
      RETURN
3     PTLCF = BORROW(FIRST(Q))
      RETURN
4     PTLCF = BORROW(Q)
      RETURN
      END
```

```
      INTEGER FUNCTION SELECT(A,I)
      INTEGER A,I
      INTEGER AS,B,BORROW,FIRST,INV,IS,K,N,PFA,PFL,TAIL,TYPE
1     N = 1
      AS = A
      IS = I
      B = 0
2     IF (AS .EQ. 0 .OR. IS .EQ. 0) GO TO 5
      K = FIRST(IS)
3     IF (N .GE. K) GO TO 4
      AS = TAIL(AS)
      N = N+1
      IF (AS .EQ. 0) GO TO 5
      GO TO 3
4     IF (TYPE(AS) .EQ. 1) GO TO 42
      B = PFA(FIRST(AS),B)
      GO TO 44
42    B = PFL(BORROW(FIRST(AS)),B)
44    IS = TAIL(IS)
      AS = TAIL(AS)
      N = N+1
      GO TO 2
5     SELECT = INV(B)
      RETURN
      END


      INTEGER FUNCTION SUMSET(N)
      INTEGER FIRST,INV,ILS,IOR,NP,PFA,PFL,R,S,T,TAIL
1     R = PFA(1,0)
      S = PFL(R,0)
      NP = N
2     IF (NP .EQ. 0) GO TO 3
      T = ILS(R,FIRST(NP))
      R = IOR(R,T)
      CALL ERLA(T)
      S = PFL(R,S)
      NP = TAIL(NP)
      GO TO 2
3     SUMSET = INV(S)
      RETURN
      END
```