Sparse Polynomial Arithmetic

Stephen C. Johnson

Bell Telephone Labs
Murray Hill, N. J.

ABSTRACT

Sparse polynomial representations are used in a number of algebraic manipulation systems, including Altran. This paper discusses the arithmetic operations with sparsely represented polynomials; we give particular attention to multiplication and division. We give new algorithms for multiplying two polynomials, with n and m terms, in time $mn\log m$; these algorithms have the property that, in the usual univariate dense case, the algorithm is bounded by mn. Division algorithms are discussed which run in comparable time.

Section 0: Introduction

A univariate polynomial of degree d in an indeterminate x is usually represented as

$$\sum_{i=0}^{d} a_i x^i$$

We shall refer to this as the *dense* representation of a polynomial.

If many of the a_i are zero, it is natural to examine the *sparse* representation

$$\sum_{i=1}^{n} a_i x^{\alpha_i}$$

where we require that each a_i be nonzero. n is called the *number of terms* in the sparse polynomial. Note that 0 is represented by a polynomial with 0 terms.

In order to permit rapid comparison of two polynomials, and simplify arithmetic computation, we usually require that the exponents α_i be strictly ordered; we shall assume that the α_i are decreasing:

$$\alpha_i > \alpha_{i+1}, \qquad i=1, \ldots, n-1$$

We shall informally call a polynomial sparse if its dense representation has many zero coefficients, and dense otherwise.

We generalize these notions to multivariate polynomials in the obvious way; if we have indeterminates $x_1, \ldots, x_v$, and a polynomial of degrees $d_1, \ldots, d_v$, the dense representation is

$$\sum_{i_1=0}^{d_1} \cdots \sum_{i_v=0}^{d_v} a_{i_1, \ldots, i_v} x_1^{i_1} \cdots x_v^{i_v}$$

and the sparse representation is

$$\sum_{i=1}^{n} a_i \, x_1^{\alpha_{i1}} \cdots x_v^{\alpha_{iv}}$$

As in the univariate case, we wish the exponent v-tuples to be distinct; it is convenient to order the v-tuples $(\alpha_{i1}, \ldots, \alpha_{iv})$ lexicographically strictly decreasing (although other orderings are used — see [1]).

In the multivariate case, the dense polynomial representation can take truly staggering amounts of space. For example, the ten term polynomial

$$y_0 + y_1 x + \cdots + y_9 x^9$$

in the indeterminates $x, y_0, y_1, \ldots, y_9$, is represented by 10240 coefficients in the dense representation, most of which are zero. For this reason, most algebraic manipulation systems represent multivariate polynomials in some manner which capitalizes on blocks of zero coefficients; the two most frequent means are recursive univariate representation and sparse representation. In this paper, we consider only the sparse representation.

In practice, we are usually prepared to set upper limits l_i on the largest exponent which we will encounter for each indeterminate x_i. In this case, we can map the multivariate computations into univariate ones as follows: each exponent v-tuple

$$(\alpha_{i1}, \ldots, \alpha_{iv})$$

gets mapped into

$$\alpha_i = (\cdots(\alpha_{i1} l_2 + \alpha_{i2}) l_3 + \cdots) l_v + \alpha_{iv}$$

If $\alpha_{ij} < l_j$, for $j=2, \ldots, v$, then the α_i are strictly decreasing if and only if the $(\alpha_{i1}, \ldots, \alpha_{ij})$ are in strictly decreasing lexicographic order. As long as the exponents remain within these bounds, the set of sparsely represented multivariate polynomials is mapped isomorphically into the set of sparsely represented univariate polynomials. Since univariate polynomials have notational and conceptual advantages, we will describe algorithms in terms of univariate sparse polynomials; the algorithms are easily translatable to the multivariate case.

Section 1: Sparse Computations

There is little theoretical knowledge about sparse polynomial algorithms. In the dense case, fast Fourier transform, interpolation, and "divide and conquer" techniques can (at least asymptotically) significantly speed up polynomial multiplication [2,3]; in contrast, the useful sparse algorithms have a decidedly "classical" flavor.

There are no known nontrivial upper or lower bounds for the time taken by multiplication, and it is not even known whether there exists an algorithm to multiply two n term sparse polynomials in time $O(n^2)$.

We shall study the four arithmetic operators on sparse univariate polynomials; our basic measure of the complexity of the problem is the number of terms in the operand polynomials. Thus, in general we wish to compute

$$\sum_{i=1}^{n} a_i x^{\alpha_i} \quad \textbf{op} \quad \sum_{j=1}^{m} b_j x^{\beta_j} = \sum_{k=1}^{K} c_k x^{\gamma_k}$$

where **op** is one of the four arithmetic operators, the exponents of each polynomial are in strictly decreasing order, and each coefficient is nonzero.

When we are given a sparse polynomial operation, the work (assuming that we stick to classical methods) can be divided into parts: computing the terms of the answer, and ordering them. The first part includes the cost of coefficient operations and (in the case of multiplication and division) exponent additions and subtractions. The second part includes the cost of exponent comparisons, and any other bookkeeping needed to order the terms in the answer.

If we compare the costs of two methods of doing a polynomial operation on given inputs, we will see that the methods differ only in the costs associated with the second parts of the algorithms. All of the operations in the first part are demanded by the mathematical definitions, and must get performed in some order. Also, in general we can assume that the bookkeeping involved in the second part is proportional to the number of exponent comparisons. Thus, *we shall compare methods on the basis of the number of exponent comparisons.*

This has the disadvantage that the true costs are not proportional to the number of exponent comparisons, since the true cost also includes coefficient operations. In Altran [5], Version 1.8, we estimate that, for large polynomial multiplications (e.g., 100 by 100 terms), the cost of exponent comparisons is about one half of the total multiplication cost.

To offset this disadvantage, there is the advantage that the number of comparisons can be measured without reference to a particular data structure, machine architecture, or implementation language. Thus, these results can hopefully be applied outside of the particular environment where they were developed.

Section 2: Addition and Subtraction

Throughout this section, we will discuss addition only; the results hold for subtraction with the obvious sign changes.

The simplest way of adding two sparse polynomials would appear to be:

1. Concatenate the two polynomials to obtain one with $n+m$ terms.
2. Sort the terms into decreasing order by exponent.
3. Make a pass through the sorted terms; two adjacent terms with the same exponent are replaced by a single term, whose coefficient is the sum of the coefficients of the components. If the resulting coefficient is zero, the entire term is deleted.

A simple analysis shows that this algorithm is dominated by the cost of the sort in Step 2, which is $(n+m)\log(n+m)$. Moreover, this algorithm uses space $n+m$ when the answer may be smaller than that due to combination or cancellation of terms. An improved algorithm is easily stated, provided that the input terms are sorted in decreasing exponent order. The central idea is the following: *generate the answer, term by term, in descending exponent order.*

At each stage of the addition, there will be an index i such that each term before x^{α_i} has been entered into the sum, but x^{α_i} has not yet entered into the sum; similarly, there will be

another index j for the other addend. The next term added to the sum will be the one of x^{α_i} and x^{β_j} of highest exponent. The complete algorithm follows:

To compute:

$$\sum_{i=1}^{n} a_i x^{\alpha_i} + \sum_{j=1}^{m} b_j x^{\beta_j} = \sum_{k=1}^{K} c_k x^{\gamma_k}$$

given that the α_i and β_j are in descending order, and the a_i and b_j are nonzero.

```
k=0
i=1
j=1
while ( i≤n and j≤m ) {
    k=k+1
    if ( α_i < β_j ) {
        c_k=b_j
        γ_k=β_j
        j=j+1
        }
    else if ( α_i = β_j ) {
        c_k=a_i+b_j
        γ_k=α_i
        if ( c_k = 0 ) k=k-1
        i=i+1
        j=j+1
        }
    else if ( α_i > β_j ) {
        c_k=a_i
        γ_k=α_i
        i=i+1
        }
    }
while ( i ≤ n ) {
    k=k+1
    c_k=a_i
    γ_k=α_i
    i=i+1
    }
while ( j ≤ m ) {
    k=k+1
    c_k=b_j
    γ_k=β_j
    j=j+1
    }
K=k
```

Again, a simple analysis serves to show that the computing time is order $n+m$. In effect, the algorithm does a merge of the exponent sequences, combining terms with equal exponents and throwing out cancelled terms as it goes. A complete analysis of the running time in terms of the inputs is not very interesting; we turn instead to multiplication.

Section 3: Multiplication — Part I

Multiplication is a much harder problem. The naive approach would again generate all mn terms, sort them, and merge terms with equal exponents; the sort once again dominates this process, and we obtain a running time bounded by $mn\log(mn)$. The space required, mn, is frequently prohibitive, as well as being considerably larger than the answer in many cases.

We may also consider the multiplication as a sum of n polynomials

$$\left(\sum_{i=1}^{n} a_i x^{\alpha_i}\right)\left(\sum_{j=1}^{m} b_j x^{\beta_j}\right) = \sum_{i=1}^{n}\left(\sum_{j=1}^{m} a_i b_j x^{\alpha_i+\beta_j}\right)$$

The n summands are easily generated; however, we must be careful how we generate the sum. If we add the n summands one by one into the final sum, in the worst case (with no combination of terms) there may be im terms after $i-1$ summations; the ith summation thus costs $im+m$. We have a worst case total cost which could grow as

$$\sum_{i=1}^{n-1} (i+1)m$$

which is order of n^2m operations.

Presumably we would choose n to be smaller than m; nevertheless, n^2m is worse asymptotic behavior than $mn\log(nm)$.

If we are clever about the way in which we add the n summands we can improve on this time, however. Using the "divide and conquer" approach, we can sum n polynomials by recursively summing the first $\frac{n}{2}$ and the last $\frac{n}{2}$ and then summing. If $C(n)$ is the cost of adding n polynomials of size m, this argument shows us that, in the worst case,

$$C(n) = 2C(\tfrac{n}{2}) + \left(\frac{mn}{2} + \frac{mn}{2}\right)$$

or

$$C(n) = mn + 2C(\tfrac{n}{2})$$

Thus $C(m)$ grows with order $mn\log n$. Note that this approach is simply a "tournament" merge of n inputs.

We now consider a class of methods based on the idea that we should attempt to generate the answer term by term in order of decreasing exponents. The desired product is the sum of the mn terms of the form

$$a_i b_j x^{\alpha_i+\beta_j}.$$

Because the α_i and the β_j are ordered, we know a great deal of *a priori* information about the ordering of these terms. In particular, we know that the term with exponent $\alpha_i+\beta_j$ appears in the product strictly before the term with exponent $\alpha_i+\beta_{j+1}$. Thus, at each step in these algorithms, and for each i with $1 \leqslant i \leqslant n$, there will be an integer f_i such that terms with exponent $\alpha_i+\beta_j$ have been included in the answer for $j<f_i$, and have not been included when $j \geqslant f_i$. The exponent of the next term to be included in the answer will be the largest of the $\alpha_i+\beta_{f_i}$, where i ranges from 1 to n. If f_i is larger than m, we need not consider this value of i any longer. The f_i are decreasing with i; otherwise, a term would have been added to the answer before another that is clearly larger. Thus, if $f_j > m$ for some index i, then $f_j > m$ for all indices j with $j \leqslant i$. We maintain an index I which is the smallest i such that $f_i \leqslant m$.

The basic algorithm follows:

To compute the product:

$$\left(\sum_{i=1}^{n} a_i x^{\alpha_i}\right)\left(\sum_{j=1}^{m} b_j x^{\beta_j}\right)=\sum_{k=1}^{K} c_k x^{\gamma_k}$$

given that the α_i and β_j are in descending order, and all the a_i and b_j are nonzero.

```
if ( m=0 or n=0 ) {
     K=0
     return
     }
k=1
c_1 =0
γ_1 =α_1 +β_1
for i=1 to n do f_i =1
I=1
while ( I ≤ n ) {
     { Find an s with I ≤ s ≤ n which
     maximizes α_s+β_{f_s} }
     if ( γ_k ≠ α_s+β_{f_s} ) {
          if ( c_k ≠ 0 ) {
               k=k+1
               c_k =0
               }
          γ_k =α_s+β_{f_s}
          }
     c_k =c_k + a_s b_{f_s}
     f_s =f_s+1
     if ( f_s > m ) I=I+1
     }
K=k
```

The step which "finds s" in the above is the central problem of this method. If we do the linear search suggested by the wording, we require $n-I$ comparisons to find s. This step is done nm times, once for each term in the product, so that the running time is order n^2m.

We may reduce the asymptotic running time by observing that we need not actually look at all the $n-I+1$ exponents at each step, provided we can always find the largest of this set. In effect, we have a set of exponents for which we wish to be able to

1. Find and remove the largest element in the set.
2. Insert a new element into this set. (Each time we increment some f_s, and $f_s \leqslant m$, it has the effect of putting the exponent $\alpha_s+\beta_{f_s}$ into the set.)

There are a number of data structures which will maintain a set of H elements and permit these two operations to be done in time bounded by order of $\log H$ (See [3,4]). Perhaps the simplest of these structures is a *heap*. In this, the H elements are kept in an array h with subscripts running from 1 to H, so that the elements satisfy

$$h_i \geqslant h_{2i}$$

and

$$h_i \geqslant h_{2i+1}$$

whenever the subscripts are in the range 1 to H. Thus, h_1 is always the largest of the set; the algorithms for adding an element and removing the largest element may be found in [3] or [4].

Applying this to multiplication, we may introduce another array s_i such that the sequence

$$\alpha_{s_i}+\beta_{f_{s_i}}, \qquad i=1,\ldots,n-I+1$$

is heapsorted. Then a maximal exponent is always given by s_1, and the per term cost of the process of running the heap is at most order of $\log n$. The total multiplication cost is thus bounded by $mn\log n$.

This method requires relatively little ($2n$ words) auxiliary storage, and computes only the terms which finally appear in the answer. Because the storage management is particularly simple and suited to a FORTRAN environment (the auxiliary arrays can be allocated before beginning the multiplication), this algorithm was chosen to implement polynomial multiplication in the initial releases of Altran [5]. It is quite possible that a system with another operating environment and data representation would find the divide and conquer algorithm superior.

The next section discusses some shortcomings of this algorithm, and some proposed improvements, Section 5 gives some empirical simu-

lations, and Section 6 discusses division.

Section 4: Multiplication — Part II

Although the above heapsort multiplication algorithm is asymptotically fast, it is far from perfect. For example, in the important special case of dense univariate multiplication this algorithm attains its worst case behavior ($mn\log n$), while even the classical dense multiplication algorithm is asymptotically mn. What is more interesting is that the divide and conquer algorithm is also asymptotically mn in this case; summing k consecutive polynomials with m terms yields at most $m+k-1$ terms when the summands arise from a univariate dense multiplication. Thus, if $C(n)$ is the cost of adding n polynomials with m terms in this case, we see that

$$C(n) \leqslant 2C(\frac{n}{2})+2(m+\frac{n}{2}-1)$$

from which we can show that $C(n)$ grows asymptotically as mn (assuming $n \leqslant m$). We ask if it is possible to make the heap algorithm work as well in this special case.

As we have noted, the divide and conquer algorithm is more efficient in the dense case because the intermediate results have substantially fewer terms than in the general case. This in turn results from the large number of terms with equal exponents generated during dense multiplications. The basic heap algorithm as given in [3] or [4] makes no use of equal elements, and thus we obtain no improvement in the dense case. Potentially, by cutting down on the size of the heap, the recognition of equal exponents could improve the heap algorithm dramatically.

When we run the heap process, we are continually making exponent comparisons, and we may detect equalities. We wish to remember and make use of these equalities without destroying the advantages of the heap structure. Although we are continuing to study this problem, we have an interim solution which is well enough understood to merit inclusion as Appendix A. This program consists of two interface routines, *insert* and *remove*, the main routine *heapify*, and a utility routine *sethole*. There are three global variables of interest: *hsize*, the size of the heap; *h*, an array of elements with indices 1 through *hsize*, which are the heaped elements; and *hole*, which is the location of a hole in the heap, if any. The routine *insert* inserts a new element in the heap; the routine *remove* returns the largest element in the heap, and *heapify* is called with the index of at most one element in *h* which may not be in heaped order; *heapify* restores the heap property to *h*. When *heapify* discovers that two elements h_i and h_j are equal, it may call *chain(i, j)*; in this case we assume that h_j is removed from the heap, and a hole is left at position *j*. Presumably, the element h_j is chained to the element h_i by a mechanism which, for simplicity, we do not describe. Thus, the elements of *h* are, in our new multiplication algorithm, pointers to chains of terms with equal exponents. The utility routine *sethole* is used to set the global variable *hole*, in order to detect the boundary condition where the hole would be the last element of the heap; in this case, *hsize* is decremented.

There is no reason to assume that this version of *heapify* is optimal; in particular, it does not always find equal exponents. As the next section shows, however, it appears to represent a substantial improvement over the current Altran multiplication method. Other mechanisms such as 2-3 trees or AVL trees [3,4] might lead to practical improvements as well, and these are being studied.

In addition to improvements that can be made by improving the heap algorithm, we can make another improvement by studying the multiplication process more carefully. We observe that a term of the form

$$a_i b_j x^{\alpha_i+\beta_j}$$

appears in the product after both of the terms

$$a_{i-1} b_j x^{\alpha_{i-1}+\beta_j} \qquad (\text{if } i > 1)$$

and

$$a_i b_{j-1} x^{\alpha_i+\beta_{j-1}} \qquad (\text{if } j > 1)$$

The algorithm given in the last section only uses the second of these constraints. The first can be quite easily used, at the cost of a slight increase in the amount of logic. Whenever we use a term with exponent $\alpha_i+\beta_{f_i}$, we must check whether the two successors with exponents $\alpha_{i+1}+\beta_{f_i}$ and $\alpha_i+\beta_{f_i+1}$ are now candidates for the next term. The first is a new candidate when $i+1 \leqslant n$ and $f_{i+1}=f_i$; the second is a new candidate when $f_i+1 \leqslant m$ and either $i=1$ or $f_{i-1} > f_i+1$. Thus, for each term placed into the answer, we may generate 0, 1, or 2 successor candidates.

These tests are easily made, and are effective in reducing the average number of elements on the heap, and thus the cost.

We have seen that, by recognizing equal exponents and examining only candidates we can cut down on the number of terms which need to be examined in order to find the next term. In

common with many asymptotically fast algorithms, however, heapsort may not be best for small problems. Thus, we shall examine another algorithm, which we shall call the *List-Insertion* or *LI* algorithm. In this algorithm, we keep the candidate exponent sets sorted on a list; equal exponent sets are chained so only one appears on the list. The largest exponent is removed from one end of the list, and a new candidate is added at the other end and "bubbled" down to its correct spot. This algorithm has an *a priori* worst case behavior of n^2m, so we can expect it to be worse than the heap algorithm for large problems. As we shall see, however, it is a surprisingly strong candidate for practical problems.

Section 5: Empirical Studies

In this section, we discuss an empirical study of the number of exponent comparisons required in the multiplication of two n term polynomials, using the current Altran algorithm, the improved heap algorithm, and the list insertion algorithm.

In obtaining empirical results, we wished to control two parameters which appeared to be crucial; problem size, n, and the frequency of equal exponents. One particularly easy way of controlling the number of equal exponents generated is by limiting the number of distinct exponent differences $\alpha_i - \alpha_{i-1}$ possible in the problem; this number will be denoted S, and called the *structure* number.

To generate a set of n exponents α_i with structure S (for S a positive integer), we set

$$\alpha_1 = 0$$
$$\alpha_i = \alpha_{i-1} + \text{rand}(S), \qquad i = 2, \ldots, n$$

Here, rand(S) is taken to be a random integer from 1 to S, chosen uniformly. Notice that $S=1$ is the dense univariate case. We empirically investigated the number of exponent comparisons required in the multiplication of two polynomials with n terms and structure S. All coefficient operation costs and other bookkeeping costs are ignored, so the actual differences reported here are larger than should be expected in practice.

Results were collected by taking the mean over 20 trials for each value of n and S; the n values went from 10 to 90 by steps of 20, and the S values were 1, 4, 16, and 64. The observed quantities were the number of exponent comparisons divided by n^2.

S	n	Altran	Heap	LI
1	10	4.69	.99	.81
	30	7.30	1.00	.93
	50	8.60	1.00	.96
	70	9.45	1.00	.97
	90	10.14	1.00	.98
4	10	4.83	2.44	1.39
	30	7.55	3.45	1.93
	50	9.02	3.68	2.19
	70	9.88	3.88	2.29
	90	10.52	4.07	2.38
16	10	4.74	3.45	1.97
	30	7.76	5.42	3.80
	50	9.05	6.33	4.83
	70	9.91	7.02	5.47
	90	10.60	7.57	5.94
64	10	4.88	3.94	2.20
	30	7.61	6.37	5.18
	50	9.20	7.53	7.46
	70	9.89	8.29	9.64
	90	10.68	8.94	11.33

Notice that the current Altran algorithm is relatively insensitive to the structure parameter S; there is roughly a 5% variation in the number of comparisons per term as S goes from 1 to 64. The heap ratios are always better than the Altran ratios, and, as expected, the heap algorithm is much better when S is small, and approaches the Altran values as S becomes large. What is perhaps most surprising is the strong showing of the LI algorithm. It is the fastest algorithm whenever either n or S is small, and does its worst when the problems are both large and unstructured. The LI and heap algorithms are within a factor of 2 of each other over the range studied, although we expect that asymptotically the heap algorithm will be better as n and S go to infinity. Because exponent comparison is only a part of the total multiplication cost, in practice the two algorithms will differ by less than this; in fact, for a practical implementation, the small differences shown here are likely to be outweighed by bookkeeping costs and/or storage requirements. However, either the heap or LI algorithm algorithm appears to do a better job than the current Altran algorithm.

Section 6: Division and Divide Test

We suppose that we are given two (sparsely represented) polynomials,

$$C = \sum_{k=1}^{K} c_k x^{\gamma_k}$$

and

$$B = \sum_{j=1}^{m} b_j x^{\beta_j}$$

We ask about the existence of a (sparsely represented) polynomial

$$A = \sum_{i=1}^{n} a_i x^{\alpha_i}$$

with the property that

$$C = A\,B.$$

If there is such an A, we say B *exactly divides* C, and call the operation of finding A from B and C, or deciding that none exists, the *divide test* operation. If B does not exactly divide C, then under certain circumstances it is meaningful to ask for the remainder of the division operation after we have removed the largest possible multiple of B. This *remainder* operation will also be briefly discussed.

In order for exact division to be possible, the leading coefficient of B must exactly divide the leading coefficient of C, and the leading exponent of B must be less than or equal to the leading exponent of C.

We may begin the computation by setting

$$a_1 x^{\alpha_1} = \frac{c_1 x^{\gamma_1}}{b_1 x^{\beta_1}}$$

We can then compute

$$C_1 = C - a_1 x^{\alpha_1} B$$

B exactly divides C if and only if B exactly divides C_1, and C_1 has lower degree than C; thus we can continue this process until either $b_1 x^{\beta_1}$ fails to exactly divide the leading term of some C_i, or some C_i becomes zero. In the first case, exact division is impossible; in the second, exact division is possible, and the quotient is given by

$$A = \sum_{i=1}^{n} a_i x^{\alpha_i}.$$

This algorithm is similar in spirit and performance to the multiplication algorithm which simply added n summands. We obtain an expected worst case time which is order of n^2m.

Notice that the timing for division is given in terms of the related multiplication. It is very difficult to get good timing bounds for divide test; in the case where exact division is impossible, it is hard to say at what stage this will be discovered, while when the division succeeds, the cost depends critically on the number of terms in the quotient, which is *a priori* unknown.

There is another subtlety that should be mentioned. When we multiply with an algorithm which costs $mn\log n$ or n^2m, we are free to choose n to be the smaller of m and n. When we divide, however, we cannot choose to divide by the quotient, because we don't know it. Thus, in the above n^2m algorithm, n is the number of terms in the quotient and m is the number of terms in the divisor; when m is small and n is large, the division still takes n^2m, when the associated multiplication could take only nm^2.

We can easily adapt the heap and list insertion algorithms to carry out a divide test operation. The central idea, as with multiplication, is to avoid large intermediate results while still generating the quotient terms one by one. The algorithm builds the quotient term by term, and carries out the multiplication of the quotient by the divisor, and the subtraction of this product from the dividend, simultaneously and term by term. When there is a term in the dividend which is not cancelled by a term in the product, we generate from this term a new term in the quotient. Thus, at each stage in the divide test algorithm, we have used a certain number of terms from the dividend, we have computed a certain number of terms in the quotient, and we are in the process of multiplying together the current quotient and the divisor. To avoid becoming lost in the details of the multiplication algorithm, we shall assume that we have two routines, *multerm* and *mulexp* which take care of running the multiplication for us. *mulexp* tells us the exponent of the next term to be generated by the multiplication of the current quotient and the divisor; if there is no next term for some reason, this exponent is returned as -1. *mulexp* does no work however, except to "peek" at the current state of the multiplication and examine the exponent. *multerm* actually computes the coefficient of this next product term, updates the f array, and does any other relevant bookkeeping; in effect, *multerm* resembles the body of the multiplication algorithm given in section 3, above. The algorithm follows:

To compute:

$$\frac{\sum_{k=1}^{K} c_k x^{\gamma_k}}{\sum_{j=1}^{m} b_j x^{\beta_j}} = \sum_{i=1}^{n} a_i x^{\alpha_i}$$

or report that exact division is impossible. It is

assumed that the γ_k and the β_j are in decreasing order, and that the c_k and the b_j are nonzero.

```
if ( m = 0 ) return "division by zero"
n = 0
k = 1
{ Initialize Multiplication }
while ( k ≤ K ) {
    δ = mulexp ( )
    if ( δ > γ_k ) {
        ε = δ
        e = − multerm ( )
        }
    else if ( δ = γ_k ) {
        ε = δ
        e = c_k − multerm ( )
        k = k+1
        }
    else if ( δ < γ_k ) {
        ε = γ_k
        e = c_k
        k = k+1
        }
    if ( e ≠ 0 ) {
        if ( ε < β_1 or b_1 doesn't divide e )
            return "no division"
        n = n+1
        a_n = e/b_1
        α_n = ε − β_1
        }
    }
if ( mulexp() = −1 ) return Σ_{i=1}^{n} a_i x^{α_i}
else return "no division"
```

If desired, this algorithm can easily be modified to deliver the remainder as well as the quotient; instead of returning when division is seen to be impossible, we simply continue to generate the terms ex^{ϵ} and add them to make up the remainder, while the quotient remains unchanged. The details are left to the reader.

Section 7: Summary

We have discussed arithmetic with sparse polynomials. Addition and subtraction are simple processes for which there are linear algorithms. Multiplication of an m term polynomial by an n term polynomial can be done asymptotically in time $mn\log n$, using a heapsort. Performance can be improved in special cases, such as the univariate dense case, by modifying the heap to recognize equal exponents. The list insertion algorithm, although asymptotically mn^2, seems very competitive in practice. Roughly speaking, we can do a divide test in about the same time as that required to do the associated multiplication.

As far as future research is concerned, probably the major theoretical question is whether there exists a sparse multiplication algorithm which runs in time mn. More generally, there are no nontrivial lower bounds on the time required for multiplication or division. In practice, we need to understand more about the interaction of data structures, algorithms, computer architectures, and bookkeeping. To make sensible implementation decisions, we have to try to understand what problems we will be called upon to do; how big are they, how sparse are they, and what is their structure? Measurement and better models appear to hold the key to improved sparse polynomial arithmetic in the 1970's.

Acknowledgments

I am grateful to many of my colleagues at Bell Labs for discussions and ideas which have shaped this work; especially to W. S. Brown, M. D. McIlroy, A. V. Aho, A. D. Hall, and C. L. Mallows. The preparation and editing of this paper was greatly aided by software written by B. W. Kernighan and L. L. Cherry; the paper was typeset by programs running under the UNIX operating system.

References

1. Barton, D., Bourne, S.R., and Fitch, J.P., An Algebra System, *Computer Journal* **13** (1970), pp. 32-39

2. Knuth, D.E., *The Art of Computer Programming, Vol 2,* Addison-Wesley, 1969.

3. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Complexity of Computer Algorithms,* Addison-Wesley, 1974 (to appear)

4. Knuth, D.E., *The Art of Computer Programming, Vol 3,* Addison-Wesley, 1973.

5. Brown, W.S., *et al,* ALTRAN, Vols I and II (1973) Bell Telephone Labs.

Appendix A: insert, remove, and heapify

```
procedure insert ( e ) {
 if ( hole ≠ 0 ) {
      h_hole = e
      heapify ( hole )
      }
 else {
      hsize = hsize+1
      h_hsize = e
      heapify ( hsize )
      }
 }

procedure remove ( ) {
    while ( hole ≠ 0 ) {
         h_hole = h_hsize
         hsize = hsize−1
         heapify ( hole )
         }
    sethole ( 1 )
    return ( h_1 )
    }
```

```
procedure heapify ( i ) {
    hole = 0
    j = i
    while ( j > 1 ) {
         if ( h_j < h_j/2 ) break
         else if ( h_j = h_j/2 ) {
              chain ( j/2, j )
              sethole ( j )
              return
              }
         else if ( h_j > h_j/2 ) {
              h_j, h_j/2 = h_j/2, h_j
              j=j/2
              }
         }
    if ( i ≠ j ) return
    while ( 2j ≤ hsize ) {
         k = 2j
         if ( k+1 ≤ hsize ) {
              if ( h_k+1 ≥ h_k ) k = k+1
              }
         if ( h_j > h_k ) return
         else if ( h_j = h_k ) {
              chain ( j, k )
              sethole (k )
              return
              }
         else if ( h_j < h_k ) {
              h_j, h_k = h_k, h_j
              j = k
              }
         }
    }

procedure sethole ( i ) {
    if ( i = hsize ) hsize = hsize−1
    else hole = i
    }
```