SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

THE COMPUTATION OF POWERS OF SYMBOLIC POLYNOMIALS*

ELLIS HOROWITZ† AND SARTAJ SAHNI‡

Abstract. Recent results on the computation of powers of symbolic polynomials are reviewed in perspective. Then a new algorithm is given which computes the nth power of a completely sparse polynomial using a linear number of multiplications. This is followed by experimental results comparing the new algorithm to iteration using both completely sparse and completely dense polynomials as data.

Key words. polynomial powers, symbolic powers, sparse polynomial powers

1. Introduction. Let $P(x_1, \cdots, x_v)$ be a polynomial in v variables with integral coefficients. Suppose that $d = \text{degree }(P)$ in x_i, $1 \leqq i \leqq v$, and that all possible terms of P are present. Then P has $(d + 1)^v$ terms and is said to be *completely dense*. If P^i has $(id + 1)^v$ terms $1 \leqq i \leqq n$, then P^i remains *completely dense to power n*. Using this worst case assumption of polynomial growth, and the classical polynomial multiplication algorithm [4, p. 362], Heindel in [2] showed that computing P^n by iteration was faster than using the binary method (binary expansion of the exponent, see [3, p. 399]). Briefly reviewing that result, we see that it follows from the completely dense assumption that the cost for iteration is asymptotically

$$\sum_{1 \leqq i \leqq n-1} (d + 1)^v(id + 1)^v < ((n - 1)d + 1)^v(d + 1)^v(n - 1) < n^{v+1}(d + 1)^{2v},$$

while the cost for the binary method is bounded by

$$\sum_{1 \leqq i \leqq \log_2 n} (2^i d + 1)^{2v} < (n(d + 1))^{2v}.$$

Thus the ratio of these methods, iteration/binary $= n^{v+1}/n^{2v} = 1/n^{v-1}$, and so for $v > 1$ variables, iteration becomes asymptotically superior. This was a somewhat nonintuitive result in the sense that the binary method requires only $O(\log_2 n)$ polynomial multiplications, whereas iteration requires $O(n - 1)$, and therefore one might naturally conclude (e.g., see Knuth [4]) that binary would be better.

The binary method and iteration have one thing in common; namely, they are whole polynomial methods. This is an intuitive idea by which we mean that at every step where a multiplication is done, it is done with polynomials. There are however, other methods for computing powers which do not rely on this whole polynomial property. One such approach, based upon evaluation and subsequent interpolation, was presented by Horowitz [3]. Using the previous assumptions, that method will compute P^n in time proportional to $(n(d + 1))^{v+1}$. At the heart of this algorithm is a routine which computes the nth power of an integer using the binary method. Hence this algorithm, in addition to having a

* Received by the editors October 1, 1973, and in revised form June 4, 1974. This work was supported by the National Science Foundation under Grant GJ-33169.

† Computer Science Program, University of Southern California, at Los Angeles, Los Angeles, California 90007.

‡ Department of Computer, Information and Control Sciences, University of Minnesota, Minneapolis, Minnesota 55455.

better asymptotic time, showed that one could operate on multivariate polynomials via some transformational technique and return the problem of computing polynomial powers to computing powers of single precision numbers.

At present, the method which has the best asymptotic computing time is obtained by using the fast Fourier transform and its convolution property; see Pollard [5].

Table 1 gives the asymptotic computing times for four "polynomial power" algorithms applied to dense polynomials. The work factor gives the amount of work per term in the answer that each method requires. A work factor of 1 would be optimal; however, the best known is $\log(n(d+1))$. The asymptotic computing times for the first 3 methods were obtained assuming the classical multiplication method is used. These could be reduced by using faster polynomial multiplication methods, though the direct use of the fast Fourier transform (FFT) would still yield the lowest upper bound.

TABLE 1
Asymptotic times, P^n completely dense

Method	Time	= Terms	*	Work factor
Binary	$n^{2v}(d+1)^{2v}$	$= n^v(d+1)^v$	*	$n^v(d+1)^v$
Iteration	$n^{v+1}(d+1)^{2v}$	$= n^v(d+1)^v$	*	$n(d+1)^v$
Eval-Interp	$n^{v+1}(d+1)^{v+1}$	$= n^v(d+1)^v$	*	$n(d+1)$
FFT	$n^v(d+1)^v \log(n(d+1))$	$= n^v(d+1)^v$	*	$\log(n(d+1))$

The reliance on the completely dense model alone is somewhat limited because of the exponential growth of the number of terms in the answer. Practical computation dictates that dense polynomials in 3 or more variables can only be raised to quite small powers, e.g. see [3], before either core or time become excessive.

Existing algebra systems need to handle multivariate problems, but often these problems are of a sparse nature. In [1], Gentleman suggested the definition for a totally sparse polynomial, the intuitive opposite of the completely dense case. If P initially has t terms and P^i has $\binom{t+i-1}{t-1}$ terms, $1 \leqq i \leqq n$, then P is said to be *completely sparse to power n*. The motivation for this definition is simply that for P^i to grow exactly as $\binom{t+i-1}{t-1}$, it must be the case that the fewest possible number of terms combine as we compute each new iterate. An example of such a polynomial is

$$P(x_1, \cdots, x_t) = x_1 + \cdots + x_t,$$

which is completely sparse for all i. Now in [1], Gentleman gives a result similar in spirit to Heindel's: for completely sparse polynomials, computing P^n by iteration is faster than using the binary method. The computing time for iteration is

$$\sum_{1 \leqq i \leqq n-1} t\binom{t+i-1}{t-1} = t\binom{t+n-1}{t} - t.$$

No closed formula for the time using the binary method has been obtained, but in [1] it is shown that the ratio of the costs of binary to iteration needed to compute P^{2^n}, where P is completely sparse and initially has t terms, is given by

$$\frac{1}{2n}\binom{2n}{n}\left(1 - \frac{n^2}{t}\right) + O(t^{-2}),$$

This implies that the binary method is more costly by at least a binomial factor.

In this paper, we will present a new algorithm for computing a power of an arbitrary polynomial. Its motivation comes from the definition of a completely sparse polynomial, and its computing time has a logarithmic work factor. Thus, this new method corresponds in complexity to the use of the FFT for computing powers of completely dense polynomials. We then present empirical results comparing this new algorithm to iteration.

2. The algorithm. First let us consider a specific instance of a completely sparse polynomial, namely,

$$P(x_1, \cdots, x_t) = x_1 + \cdots + x_t.$$

By the multinomial expansion theorem (e.g., see [6, p. 64]), it follows that

$$(x_1 + \cdots + x_t)^n = \sum_{n_1 + \cdots + n_t = n} \binom{n}{n_1, \cdots, n_t} x_1^{n_1} \cdots x_t^{n_t}, \tag{2.1}$$

where the n_i are integers in the range $0 \leqq n_i \leqq n$. The number of distinct t-tuples which sum to n is precisely $\binom{t+n-1}{t-1}$, corresponding to the definition of a completely sparse polynomial. The definition of the multinomial coefficient is

$$\binom{n}{n_1, \cdots, n_t} = \frac{n!}{n_1! \cdots n_t!}.$$

Moreover, we emphasize that each time the n_i change, the next multinomial coefficient may be obtained from the previous one using one multiplication and one division, i.e.,

$$\binom{n}{n_1, \cdots, n_i - 1, n_{i+1} + 1, \cdots, n_t} = \frac{n_i}{n_{i+1} + 1}\binom{n}{n_1, \cdots, n_i, n_{i+1}, \cdots, n_t}. \tag{2.2}$$

Thus, if we generate the t-tuples in lexicographic order, it requires $2\binom{t+n-1}{t-1}$ coefficient multiplications to compute the nth power of $P(x_1, \cdots, x_t)$. Unfortunately, for general sparse polynomials, it becomes necessary to sort the terms thus adding a log factor to the computing time.

The general algorithm begins with an arbitrary polynomial, say

$$P(y_1, \cdots, y_v) = \sum_{1 \leqq i \leqq t} a_i y_1^{e_{i1}} \cdots y_v^{e_{iv}}$$

in v variables with t nonzero terms. Conceptually, the method then proceeds by setting

$$x_i = a_i y_1^{e_{i1}} \cdots y_v^{e_{iv}}, \qquad 1 \leqq i \leqq t,$$

producing the new polynomial

$$\bar{P}(x_1, \cdots, x_t) = x_1 + \cdots + x_t.$$

The nth power of $\bar{P}$ is computed in linear time and the substitutions back to the y_i followed by a sort increase the bound by a log factor. This algorithm is now given in complete detail.

The input polynomial with t terms is assumed to be stored term by term in the array TERM($1:t$). The array N($1:t$) contains the exponent vector and is initialized to:

$$\mathrm{N}(1) \leftarrow n, \quad \mathrm{N}(2) \leftarrow \cdots \mathrm{N}(t) \leftarrow 0;$$

and the global variable POW is set to $(\mathrm{TERM}(1))^n$. t is a global variable whose value is the number of terms in the input polynomial. Then the following routine is called using

$$\mathrm{MULT}(\mathrm{TERM}(1)^n, 1, 1). \tag{2.3}$$

```
ALGORITHM MULT(POL, COEF, $i$).
  Input:  POL, a multivariate polynomial
          COEF, an integer
          POW, a global variable initialized to $(\mathrm{TERM}(1))^n$
          $i$, a nonnegative integer
  Output: the global variable POW is set to: $(\mathrm{TERM}(1) + \cdots + \mathrm{TERM}(t))^n$
 1. if $i \neg= t$ then /* move forward */
 2.    do;  do while $(\mathrm{N}(i) \neg= 0)$;
 3.           $\mathrm{N}(i) \leftarrow \mathrm{N}(i) - 1$;
 4.           $\mathrm{N}(i+1) \leftarrow \mathrm{N}(i+1) + 1$;
 5.           $\mathrm{COEF} \leftarrow \mathrm{COEF} * (\mathrm{N}(i) + 1)/\mathrm{N}(i+1)$;
 6.           $\mathrm{POL} \leftarrow (\mathrm{POL}/\mathrm{TERM}(i)) * \mathrm{TERM}(i+1)$;
 7.           $\mathrm{POW} \leftarrow \mathrm{POW} + \mathrm{POL} * \mathrm{COEF}$;
 8.           CALL MULT (POL, COEF, $i + 1$):
 9.         end;
10.         if $i = 1$  then return
11.                     else do/* backtrack */
12.                        $\mathrm{N}(i) \leftarrow \mathrm{N}(i+1)$;
13.                        $\mathrm{N}(i+1) \leftarrow 0$;
14.                        return;
15.                        end;
16.    end;
17. end POWER;
```

We now show that algorithm MULT when called as in (2.3) with POW $= (\mathrm{TERM}(1))^n$ and $\mathrm{N}(1) = n$, $\mathrm{N}(2) = 0$ results in the desired solution POW $= (\sum_{1 \leqq i \leqq t} \mathrm{TERM}(i))^n$. It is clear that if all the terms of the sum in (2.1) are generated and then TERM(i) is substituted for x_i, we obtain the desired result. Associated with each term in the sum for (2.1) is a power sequence $(n_1, n_2, \cdots, n_t)$ and a coefficient $\binom{n}{n_1, \cdots, n_t}$. For any power sequence $(n_1, n_2, \cdots, n_t)$ and $2 \leqq i \leqq t$, define the i-prefix to be $(n_1, \cdots, n_{i-1})$ and, for $i = 1$, the 1-prefix is ( ). To see that only correct power sequences are generated and that each such sequence is generated exactly once, we note that:

(i) Steps 3 and 4, 12 and 13 are the only ones that alter the power sequence. Both pairs of steps preserve the value of $\sum \mathrm{N}(i)$ and maintain $\mathrm{N}(i) \geqq 0$ (note that the conditional of step 2 ensures that $\mathrm{N}(i) = 0$ when steps 12 and 13 are executed). Hence only valid power sequences are generated.

(ii) Each time a call to MULT is made, either initially or from step 8, the i- or $(i + 1)$-prefix, respectively, is different from all other calls with the same i-value. Hence each power sequence is generated only once.

(iii) For any i-prefix, a call to MULT results in the generation of all power sequences with the same i-prefix.

From (2.2) and steps 3, 4 and 5 of MULT, it follows that at all times the value of COEF is $\binom{n}{\mathrm{N}(1) \cdots \mathrm{N}(t)}$. Steps 3, 4 and 6 imply that POL at any time has the value $\prod (\mathrm{TERM}(i))^{\mathrm{N}(i)}$. Hence it follows that the routine MULT, when called as described above, results in the computation of $(\sum_{1 \leqq i \leqq t} \mathrm{TERM}(i))^n$.

To get an estimate of the computing time, we note that each call to MULT from step 8 results in 2 multiplication/divisions (abbreviated 2 M/D) in step 5, 2 M/D in step 6 and another call to MULT. However, each such call results in the generation of a new term. There are exactly $\binom{t + n - 1}{t - 1}$ such terms. To compute $(\mathrm{TERM}(1))^n$, $\log n$ multiplications are needed. Hence MULT requires

$$\log n + 4\binom{t + n - 1}{t - 1} \ \mathrm{M/D} \sim O\left(\binom{t + n - 1}{t - 1}\right) \ \mathrm{M/D}.$$

The only other cost to be considered is that of the addition in step 7. The best way to do this appears to be to just generate all $\binom{t + n - 1}{t - 1}$ terms, then sort them adding together terms with identical power sequences (this will be required only if the original polynomial $P = \sum \mathrm{TERM}(i)$ is not sparse to power n). This sort-add step can be done in

$$O\left(\binom{t + n - 1}{t - 1} \log \binom{t + n - 1}{t - 1}\right),$$

resulting in an overall computing time of $O(T \log T)$, where T is the number of terms in the answer. This is the same as for FFT over dense polynomials.

For comparison, let us consider computing P^n by computing the sequence $P, P^2, \cdots, P^n$ (i.e., iteration). Then the number of multiplications is

$$\sum_{i=1}^{n-1} t\binom{t+n-1}{t-1} = O\left(n\binom{t+n-1}{t-1}\right).$$

Here, too, a sort-add step is needed, thus adding a log factor to the computing time. The total computing time is then bounded by the sort-add time, which is

$$O\left(n\binom{t+n-1}{t-1}\log t\right).$$

Hence we see that as far as an M/D count is concerned, MULT is optimal to within a constant factor. It requires about $O(n)$ times fewer multiplications than iteration.

3. Empirical results. In this section we present the results of several tests that were made to determine the global efficiency of these 2 algorithms. Though asymptotic analyses are important, the value of practical testing should not be underestimated. This is especially true when dealing with symbolic problems, since the domain of actual computation is often moderately small, thus placing added importance on constants and less on asymptotic results. All tests were carried out on an IBM 360/65 using the SAC-1 System which provides, in part, for arithmetic operations on multivariate polynomials.

Both completely dense and completely sparse polynomials were used as test data for these algorithms. For completely sparse polynomials in v variables the polynomials used were

$$P(x_1, \cdots, x_v) = x_1 + \cdots + x_v,$$

except when $v = 1$, in which case $P(x_1) = x_1 + 1$. The completely dense polynomial in 1 variable had degree = 7, while the corresponding polynomial in 2 variables had maximum degree = 2 in each variable. Completely dense polynomials in 3 and 4 variables each have maximum degree = 1 in each variable. All coefficients of P were one. Table 2 gives the results in milliseconds for completely sparse powers, while Table 3 contains the completely dense results. The addition of step 7 was done using a standard polynomial add routine rather than a sort-add at the end as described in the analysis of MULT. Finally, a nonrecursive version of MULT was programmed so as to reduce the overhead of repeated procedure calls.

Similarly the additions required by iteration were not carried out by a sort-add. Considering that only relatively small problems were tested, it is unlikely that the advantage of using the asymptotically superior sort-add would have been reflected in the computing times of Tables 2 and 3.

4. Conclusion. We have seen that there are two basic complementary models for which one does an analysis of powering algorithms: completely dense and completely sparse polynomials. The main result here has been to exhibit an

TABLE 2
Completely sparse P^n

No. of variables	1		2		3		4	
n	Iter	Mult	Iter	Mult	Iter	Mult	Iter	Mult
2	16.6	16.6	16.6	16.6	33.2	133.1	133.1	266.2
4	16.6	49.9	83.2	66.5	316.1	366.0	898.5	1064.9
6	83.2	83.2	183.0	99.8	931.8	665.6	3095.0	2579.2
8	116.4	83.2	332.8	149.7	1896.9	1098.2	7937.2	5308.1
10	216.3	133.1	499.2	199.6	3577.6	1580.8	16872.9	9434.8
12	282.8	299.8	748.8	266.2	5990.4	2312.9	**	
14	432.6	166.4	1064.9	299.5	9085.4	2995.2		
16	449.2	232.9	1248.0	316.1	13295.3	3966.3		

** Power could not be computed with 23k words of work space.

TABLE 3
Completely dense P^n

No. of variables	1		2		3		4	
max degree	7		2		1		1	
n	Iter	Mult	Iter	Mult	Iter	Mult	Iter	Mult
2	116.4	332.8	116.4	399.3	116.4	915.2	515.8	2362.8
4	715.5	3927.0	848.6	3810.5	1913.6	8636.1	9368.3	63763.2
6	1597.4	22763.5	3461.1	17555.2	8253.4	46475.5	**	
8	2812.1	99274.2	7987.2	63015.6	**			
10	4509.4	>400 sec	14809.6	172689.9				

** Power could not be computed with 23k words of work space.

algorithm which requires $O(T)$ multiplications and $O(T \log T)$ exponent comparisons (T is the number of terms in the nth power of a completely sparse polynomial). From a complexity point of view, this means the best methods we know of for computing powers take $O(T \log T)$ operations (T is the number of terms in the result) for both the completely dense and completely sparse models.

In practice, a specific problem may have characteristics that give an advantage to any one of the other known methods; e.g., see [7], [8]. In addition to the number of arithmetic operations, one may have to consider other relevant factors such as the efficiency/inefficiency of recursion, procedure calls, etc., in the source language. We have shown that for completely sparse polynomials using a FORTRAN-based system, our new algorithm is better than iteration. But for any symbol manipulation system which wants to provide only a single powering routine, iteration seems the best choice (i) because of its simplicity, (ii) because it yields all intermediate powers which may be useful, e.g., in substituting a polynomial for x in $P(x)$, and (iii) because it is uniformly good for both polynomial models. The best known methods for either model are, on the average, better than iteration by a factor of 2. Unfortunately, these specialized algorithms, FFT for dense and MULT for sparse polynomials, perform very poorly on sparse and dense polynomials, respectively.

REFERENCES

[1] W. M. GENTLEMAN, *Optimal multiplication chains for computing a power of a symbolic polynomial*, Math. Comp., 26 (1972), pp. 935–939.

[2] L. HEINDEL, *Computation of powers of multivariate polynomials over the integers*, J. Comput. System Sci., 6 (1972), pp. 1–8.

[3] E. HOROWITZ, *The efficient calculation of powers of polynomials*, Ibid., 7 (1973), pp. 469–480.

[4] D. KNUTH, *The Art of Computer Programming. Vol. II: Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[5] J. M. POLLARD, *The fast Fourier transform in a finite field*, Math. Comp., 25 (1971), pp. 365–374.

[6] D. KNUTH, *The Art of Computer Programming. Vol. I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1969.

[7] R. FATEMAN, *Polynomial multiplication, powers and asymptotic analysis: Some comments*, this Journal, 3 (1974), pp. 196–213.

[8] ———, *On the computation of powers of sparse polynomials*, Studies in Appl. Math., 52 (1974), pp. 145–155.