JOURNAL OF COMPUTER AND SYSTEM SCIENCES 7, 469–480 (1973)

The Efficient Calculation of Powers of Polynomials

ELLIS HOROWITZ*

Computer Science Department, Cornell University, Ithaca, New York 14850

Received April 3, 1972

Suppose we are given a polynomial $P(x_1, ..., x_r)$ in $r \geqslant 1$ variables, let m bound the degree of P in all variables x_i, $1 \leqslant i \leqslant r$, and we wish to raise P to the nth power, $n > 1$. In a recent paper which compared the iterative versus the binary method it was shown that their respective computing times were $O(m^{2r}n^{r+1})$ versus $O((mn)^{2r})$ when using single precision arithmetic. In this paper a new algorithm is given whose computing time is shown to be $O((mn)^{r+1})$. Also if we allow for polynomials with multiprecision integer coefficients, the new algorithm presented here will be faster by a factor of $m^{r-1}n^r$ over the binary method and faster by a factor of m^{r-1} over the iterative method. Extensive empirical studies of all three methods show that this new algorithm will be superior for polynomials of even relatively small degree, thus guaranteeing a practical as well as a useful result.

1. INTRODUCTION

Recently several analyses have appeared dealing with the problem of efficient methods for raising a polynomial to a power. The two major competing methods have been the iterative versus the binary approach. The iterative scheme would compute P^n as $P, P^2, ..., P^{n-1}, P^n$, whereas the binary method would use the binary expansion of n to decide when to square the partial result as well as when to multiply by P. In [KNU69], p. 401, Knuth implies that for the application of raising a polynomial to a power the iterative method is far *inferior* to the binary method since the number of multiplications required in the latter case can be reduced from $n - 1$ to $\log_2 n$. However, in a recent paper by Heindel [HEI72], he shows that in fact it is more efficient to use the iterative method rather than the binary method whether one is doing either single precision or multiprecision integer arithmetic on polynomials. Moreover, if r is the number of variables of P, and we wish to raise P to the nth power, then Heindel shows that the iterative scheme will be better by at least a factor of n^{r-1} in both cases. In this paper a new method for computing powers of a polynomial is given which is faster than the binary method by a factor of $m^{r-1}n^{r-1}$ for single precision arithmetic

* To be presented at the 13th Annual Symposium on Switching and Automata Theory, IEEE, Maryland, Oct. 1972.

and faster than the iterative method by a factor of m^{r-1}. Also, this new algorithm will have at its core the use of the binary method applied to elements of a finite field, thus restoring what is perhaps our intuitive notion of the most efficient way of computing powers.

In one paper related to this problem, M. Gentleman in [GEN71] shows that for his specific model of polynomials the iterative method produces an optimal multiplication chain. In particular his model is for sparse polynomials of a very particular type; nor does he concern himself with the work needed for multiprecision coefficients. This paper is concerned with a model which assumes that polynomials are dense so that we can obtain worst case computing time bounds. Besides obtaining better asymptotic bounds on the time needed to compute P^n, the algorithm presented in this paper has been tested within an existing algebraic manipulation system and has been found to give a significant improvement over the iterative method for even reasonably sized problems.

In Sec. 2 I will state some elementary results on computing times for basic operations. Then I will summarize the results of the binary and iterative methods. In Sec. 3 the new algorithm will be given plus an analysis of its computing time. In Sec. 4 empirical results comparing these several methods will be given.

2. Known Results

In this section is summarized some of the basic computing times for operations on elements of a finite field, integers and multivariate polynomials. All computing times will be given in terms of the commonly used O-notation.

Definition 2.1. Let $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ be real-valued functions defined on some common domain D. Then $f = O(g)$ if there exists a positive real number c such that

$$|f(x_1, \ldots, x_n)| \leqslant c\,|g(x_1, \ldots, x_n)| \qquad \text{for all} \quad (x_1, \ldots, x_n) \in D.$$

We shall be concerned both with fixed point, single precision as well as arbitrary multiple precision arithmetic in our later analysis. For the single precision case we will assume that all numbers are elements of the finite field with p elements, designated as GF(p). The number p will generally be chosen to be about the size of a machine word. Then the time for adding, subtracting, multiplying or dividing elements in GF(p) will be $O(1)$. The computing times for operations on multi-precision integers will be given in terms of the maximum number of digits of the integers. Thus, given two integers, say d and e, the time for addition (subtraction) and multiplication (division) is $O(\max\{\ln d, \ln e\})$ and $O((\ln d)(\ln e))$, respectively. A new method for

integer multiplication due to S. Schonhage and V. Strassen, [KNU69] p. 269 takes time $O(\ln d(\ln\ln d)(\ln\ln\ln d))$ if $d \geqslant e$. However, their method seems to be of more theoretical interest since the associated constant is quite large. Therefore we will assume the use of a "classical" multiplication algorithm.

We now need to have the times for performing operations on multivariate polynomials.

DEFINITION 2.2. Let $P(x_1, \ldots, x_r)$ be a polynomial in $r \geqslant 1$ variables with integer coefficients. Then, *norm*(P) is the sum of the absolute values of the numerical coefficients of P.

DEFINITION 2.3. Let $M_r(d, m) = \{P(x_1, \ldots, x_r)$: $(\deg(P)$ in $x_i \dotplus 1) \leqslant m$, $1 \leqslant i \leqslant r$ and $d = \text{norm}(P)\}$. $M_r(d, m)$ is the set of polynomials in r variables whose norm is d and whose total number of terms is at most m^r.

THEOREM 2.1. *Let $P(x_1, \ldots, x_r), Q(x_1, \ldots, x_r) \in M_r(d, m)$. Then an upper bound on the computing time to form $P \pm Q$ and $P \cdot Q$ using multiprecision arithmetic is $O(m^r(\ln d))$ and $O(m^{2r}(\ln d)^2)$, respectively.*

Proof. The maximum number of nonzero terms in P and Q is m^r. Thus for addition we must add m^r terms, each addition taking $O(\ln d)$. For the proof of $P \cdot Q$ see [HEI72] p. 3. We note that if instead P and Q had coefficients in GF(p), then all the additions and multiplications would take time $O(1)$ and the time for addition and multiplication for polynomials in r variables with coefficients from GF(p) would be $O(m^r)$ and $O(m^{2r})$.

There are some other operations for which we will need the computing times. In particular we need to know the times for applying modular and evaluation homomorphisms to polynomials and for applying interpolation and the Chinese remainder method to integers and elements of GF(p). These methods have already been fully documented in, for example [COL71], pp. 520–522. Let us review those results here.

THEOREM 2.2. *Let $A \in M_r(d, m)$ such that $A = \sum_{1 \leqslant i \leqslant m^r} a_i x_1^{e_{i1}} \cdots x_r^{e_{ir}}$ and p a single precision prime. Then an upper bound on the time to compute*

$$A^* = \sum_{1 \leqslant i \leqslant m^r} (a_i \bmod p)\, x_1^{e_{i1}} \cdots x_r^{e_{ir}}$$

is $O(m^r \ln d)$.

Proof. There are at most m^r nonzero terms, each term requiring a division by a single precision prime p. This implies $O(\ln d)$ for each term. This operation is referred to as a modular homomorphism since we are mapping the polynomials in $I[x_1, \ldots, x_r] \to$ GF$(p)[x_1, \ldots, x_r]$ and this mapping is a homomorphism.

THEOREM 2.3. *Let* $A^*(x_1,\ldots,x_r) \in \mathrm{GF}(p)[x_1,\ldots,x_r]$ *and suppose* $b \in \mathrm{GF}(p)$. *Then an upper bound on the time to form* $A^*(x_1,\ldots,x_{r-1},b)$ *is* $O(m^r)$.

Proof. See [COL69], p. 17.

We shall call this an *evaluation homomorphism* since we are mapping from $\mathrm{GF}(p)[x_1,\ldots,x_r] \to \mathrm{GF}(p)[x_1,\ldots,x_{r-1}]$ and this mapping also satisfies the homomorphism properties.

Now let us investigate the inverse operations of interpolation and the Chinese remainder algorithm. The analogies between these two operations have already been mentioned before, e.g. [BRO71]; namely that the Chinese remainder algorithm allows us to construct an integer from its images modulo several primes while interpolation allows us to construct a polynomial from its images modulo $x - b_1,\ldots,x - b_n$. We now present these two methods as they would be applied to multivariate polynomials.

Iterative Chinese Remainder Algorithm

Input: $A_1(x_1,\ldots,x_r)$ with coefficients in $\mathrm{GF}(p)$
$A_2(x_1,\ldots,x_r)$ with coefficients in $\mathrm{GF}(q)$

Output: $A_3(x_1,\ldots,x_r)$: $A_3 \equiv A_1 \bmod p$
$A_3 \equiv A_2 \bmod q$

(1) $A_2^* \leftarrow A_2 \bmod p$;

(2) $q^* \leftarrow q \bmod p$;

(3) $C \leftarrow (A_1 - A_2^*)/q^* \bmod p$;

(4) $A_3 \leftarrow Cq + A_2$;

THEOREM 2.4. *An upper bound on the computing time for one iteration of the Chinese remainder algorithm given above is* $O(m^r \ln q)$.

Proof. The times for steps (1) and (4) are bounded by $m^r(\ln q)$ whereas step (2) takes no more than $O(\ln q)$ and step (3) takes $O(m^r)$. In practice q will be the product of previous primes while p is the next single precision prime to be processed.

Iterative Interpolation

Input: $A_1(x_1,\ldots,x_{r-1})$ with coefficients in $\mathrm{GF}(p)$;
$b \in \mathrm{GF}(p)$;
$A_2(x_1,\ldots,x_r)$ with coefficients in $\mathrm{GF}(p)$;
$Q(x_r)$, degree$(Q) \leqslant k$; deg(A_2) in $x_r \leqslant k$

Output: $A_3(x_1,\ldots,x_r)$: $A_3(x_1,\ldots,x_{r-1},b) = A_1$,
$A_3(x_1,\ldots,x_{r-1},b_i) = A_2(x_1,\ldots,x_{r-1},b_i)$,
where b_i are the roots of Q.

(1) $A_2^* \leftarrow A_2(x_1, \ldots, x_{r-1}, b)$;

(2) $q^* \leftarrow Q(b)$;

(3) $C \leftarrow (A_1 - A_2^*)/q^*$;

(4) $A_3 \leftarrow CQ + A_2$;

THEOREM 2.5. *An upper bound on the computing time for one iteration of the interpolation algorithm above is* $O((m^{r-1})k)$.

Proof. The time for steps (1), (3), and (4) is $O((m^{r-1})k)$. The time for step (2) is $O(k)$.

We are now in a position to review the results for the iterative and binary methods that are given by Heindel in [HEI72]. The iterative algorithm is obvious so let us examine an algorithm for the binary method.

Binary Expansion

Input: P, either an element of GF(p), an integer or a multivariate polynomial and n a positive integer.

Output: $R = P^n$.

(1) $R \leftarrow 1$; $Z \leftarrow P$;

(2) $q \leftarrow \lfloor n/2 \rfloor$; $r \leftarrow n - 2q$; $n \leftarrow q$; if $r = 0$, go to (4);

(3) $R \leftarrow R \cdot Z$;

(4) *If* $n = 0$ *then* return (R);
else $Z \leftarrow Z \cdot Z$; go to (2);

We see that the total number of multiplications is $\log_2 n + v(n)$, where $v(n)$ is the number of ones in the binary representation of n. Therefore the total number of multiplications is $O(\ln n)$. If P is an element of GF(p), then the time to compute P^n is $O(\ln n)$ whereas the time for the iterative method is $O(n)$. The binary method is is clearly better. If P is an integer with e precision ($e = \lfloor \log P \rfloor + 1$), then P^n is an en-precision number and the time for the binary algorithm can be at most

$$\sum_{1 \leqslant i \leqslant \log_2 n} (2^{i-1}e)^2 = O(e^2n^2).$$

The time for the iterative method is

$$\sum_{1 \leqslant i \leqslant n-1} (ie)e = O(e^2n^2).$$

Thus we see that the iterative scheme already catches up to the binary method when applied to integers. In [HEI72], Heindel analyzes the computing time for the iterative

and binary methods when applied to a polynomial $P(x_1, \ldots, x_r)$ raised to the nth power. He considers both the case where the coefficients of P are from a finite field or from the integral domain of the integers. His results can be summarized as follows:

	Polynomial $P(x_1, \ldots, x_r)$ $\lvert$ *Integer coefficients* $\rvert \leqslant d$	$\deg(P)$ in $x_i + 1 \leqslant m$ *coefficients in* GF(p)
Method		
Binary	$O(m^{2r}n^{2r+2}(\ln d)^2)$	$O(m^{2r}n^{2r})$
Iterative	$O(m^{2r}n^{r+2}(\ln d)^2)$	$O(m^{2r}n^{r+1})$

In the next section two new algorithms based on the use of homomorphisms will be presented whose computing time will be $m^{r+1}n^{r+2}(\ln d)^2$ for polynomials with integer coefficients and $(mn)^{r+1}$ for polynomials with coefficients in GF(p).

3. Homomorphism Algorithm

In this section we develop two algorithms, CPPOWER(P, n) for computing P^n when the coefficients of P are in GF(p) and PPOWER(P, n) for computing P^n when the coefficients of P are integers. PPOWER will use CPPOWER as a subroutine. Moreover, PPOWER is based on the idea of using modular homomorphisms and the Chinese remainder method while CPPOWER uses the technique of evaluation homomorphisms and interpolation. Since PPOWER depends upon CPPOWER we will present that algorithm first.

Algorithm CPPOWER

Input: $P^*(x_1, \ldots, x_r)$ a multivariate polynomial in $r \geqslant 0$ variables with coefficients in some finite field GF(p) and an integer $n > 0$;

Output: $R^*(x_1, \ldots, x_r) = [P^*(x_1, \ldots, x_r)]^n$;

(1) [Initial case] If $r = 0$, compute $R^* \leftarrow (P^*)^n$ using the binary method; end.

(2) [Initialize] $m_{r-1} \leftarrow \max\{\deg(P) \text{ in } x_{r-1}\}$; $m_r \leftarrow \deg(P)$ in x_r; $C(x_1, \ldots, x_r) \leftarrow 0$; $D(x_r) \leftarrow 1$;

[Interpolation] $b \leftarrow -1$;

(3) [Choose next] [point] $b = b + 1$; if $b = p$ then stop.

(4) [Evaluation] $\bar{P}(x_1, \ldots, x_{r-1}) \leftarrow P^*(x_1, \ldots, x_{r-1}, b)$;
[Homomorphism] If $\deg(P)$ in $x_{r-1} < m_{r-1}$ then go to (3).

(5) [Recursion] $R^*(x_1, \ldots, x_{r-1}) \leftarrow \text{CPPOWER}(\bar{P}, n)$;

(6) [Interpolate] $$C(x_1, \ldots, x_r) \leftarrow \frac{R^*(x_1, \ldots, x_{r-1}) - C(x_1, \ldots, x_{r-1}, b)}{D(b)} \cdot D(x_r) + C(x_1, \ldots, x_r);$$

(7) [Done?] $D(x_r) \leftarrow (x_r - b)\, D(x_r)$;
If $\deg(D) \leqslant m_r n$ then go to (3);

(8) [end!] $R^*(x_1, \ldots, x_r) \leftarrow C(x_1, \ldots, x_r)$; end.

In order to greatly simplify the analysis and following the procedure used by W. S. Brown in [BRO71] let us assume that $(\deg(P^*)$ in $x_i + 1) \leqslant m$ for $1 \leqslant i \leqslant r$. Thus P^* has at most m^r terms all of whose numerical coefficients are elements of GF(p).

THEOREM 3.1. *Let $T_r(m)$ be a bound on the computing time for algorithm* CPPOWER(P^*, n) *to compute* $(P^*)^n$. *Then* $T_r(m) = O((mn)^{r+1})$.

Proof. If $r = 0$, P^* is an element of GF(p) and hence all arithmetic operations are $O(1)$. Thus the time for step (1) is $O(\log_2 n)$. Steps (3)–(7) are executed mn times since $(\deg(R^*)$ in $x_r + 1) \leqslant mn$. Each execution of step (4) for which $\deg(\bar{P})$ in $x_{r-1} < m_{r-1}$ corresponds to a distinct element $b \in \text{GF}(p)$ which is a zero of the leading coefficient of P^* considered as a polynomial in x_{r-1}. Since the degree of this leading coefficient is at most $m_{r-1} < m$, there are at most m additional executions of step (4). Therefore the total time for the remaining steps is

step	*time*
(3)	$O(mn)$
(4)	$O(m^{r+1}n)$ by Theorem 2.3
(5)	$O(mnT_{r-1}(m))$
(6)	$\sum_{1 \leqslant i \leqslant mn} (mn)^{r-1}\, i = O((mn)^{r+1})$ by Theorem 2.5.

Thus the total time for CPPOWER is

$$T_r = (mn)^{r+1} + mnT_{r-1} = 2(mn)^{n+1} + (mn)^2 T_{r-2}$$
$$= 3(mn)^{r+1} + (mn)^3 T_{r-3} = \cdots = r(mn)^{r+1} + (mn)\, T_0 < O((mn)^{r+1}) \quad \text{since}$$

$T_0 = \log_2 n < n$ and r is a small constant.

Now we give the algorithm for polynomials with integer coefficients.

Algorithm PPOWER

Input: $P(x_1, ..., x_r)$ a multivariate polynomial in $r \geqslant 1$ variables with integer coefficients and $n > 1$.

Output: $P^n(x_1, ..., x_r)$.

(1) [Initialize] $d \leftarrow \text{norm}(P)$; $m \leftarrow \deg(P)$ in x_r; $i \leftarrow 0$; $f \leftarrow d^n$;

(2) [Next Prime] $i \leftarrow i + 1$; Choose next prime, p_i; If none left, stop.

(3) [Modular] [Homomorphism] $P^*(x_1, ..., x_r) \leftarrow P(x_1, ..., x_r) \bmod p_i$; If $\deg(P^*)$ in $x_r < m$, go to (2).

(4) $[R^* = (P^*)^n]$ $R^*(x_1, ..., x_r) \leftarrow \text{CPPOWER}(P^*, n)$

(5) [Chinese] [Remainder] If $i = 1$, $R \leftarrow R^*$; $q \leftarrow p_i$; go to (2); Otherwise find $D(x_1, ..., x_r)$ using the iterative Chinese Remainder algorithm such that

$$D(x_1, ..., x_r) \equiv R^*(x_1, ..., x_r) \bmod p_i;$$
$$D(x_1, ..., x_r) \equiv R(x_1, ..., x_r) \bmod q;$$

(6) [Done?] $R \leftarrow D$; $q \leftarrow p_i q$; If $i \leqslant \log f$, go to (2)

(7) $P^n \leftarrow R$; end.

THEOREM 3.2. *Let* $P_r(d, m)$ *be the maximum computing time for Algorithm* PPOWER(P, n) *when* $P \in M_r(d, m)$. *Then* $P_r(d, m) = O(m^{r+1}n^{r+2}(\ln nd)^2)$.

Proof. Since P contains at most m^r terms the time needed to compute d in step (1) is $O(m^r \ln d)$. Steps (2)–(6) are executed at most $n(\ln nd)$ times since $f = (nd)^n$ bounds the size of the norm of P^n. Therefore we get the following table for the computing times

step	*time*
(1)	$O(m^r \ln d)$
(2)	$O(n \ln nd)$
(3)	$O(m^r \ln d \cdot n \ln nd) < O(m^r n(\ln nd)^2)$
(4)	$O((mn)^{r+1}n \ln nd) = O(m^{r+1}n^{r+2}(\ln nd))$
(5)	$O((mn)^r(n \ln nd)^2) = O(m^r n^{r+2}(\ln nd)^2)$
(6)	$O(n^2(\ln nd)^2$

Examining the table, steps (4) and (5) bound the computing time for the entire algorithm. Also

$$O(m^{r+1}n^{r+2} \ln nd + m^r n^{r+2}(\ln nd)^2) < O(m^{r+1}n^{r+2}(\ln nd)^2).$$

Looking at the proof of Theorem 3.2 we see that algorithm CPPOWER plus the time for the Chinese remainder algorithm is what bounds the total time for this method. This is what we would normally expect.

Now let us summarize the asymptotic computing times which have been arrived at for the binary, iterative and homomorphism algorithms. If $e = \ln nd$ we get the following table:

	Binary	*Iterative*	*Homomorphism*
Single Precision Coefficients	$O((mn)^{2r})$	$O(m^{2r}n^{r+1})$	$O((mn)^{r+1})$
Multiple Precision Coefficients	$O(m^{2r}n^{2r+2}e^2)$	$O(m^{2r}n^{r+2}e^2)$	$O(m^{r+1}n^{r+2}e^2)$

4. Empirical Results

In this section we examine several tests which were made on all three methods to see if we can determine just how practical the new homomorphism algorithm really is. All testing was done on an IBM 360/65 at Cornell University. The basic routines for the manipulation of multivariate polynomials were provided through the use of the SAC-1 system for symbolic and algebraic calculation, [COL69]. This is a collection of Fortran subroutines which allows in part for infinite precision integer and multivariate polynomial arithmetic.

In particular the binary, iterative and modular algorithms were tested on univariate, bivariate and trivariate polynomials whose coefficients were randomly generated in the range $[-2^{10}, 2^{10}]$. Tables I–III give the resulting computing times.

Table I gives the results obtained for polynomials in one variable. The vertical column represents the degree of the polynomials while the horizontal row represents the successive powers to which the polynomials were raised. For every polynomial there are three rows of times which stand for the binary (BIN), iterative (ITR), and modular (MOD) methods. Recalling the computing times obtained in the previous section for one variable ($r = 1$), in this case the time for all three methods is the same, $O((mn)^2)$. This is reflected in Table I as we see there is no substantial difference in any of these methods for the degree and powers which were tested.

Table II gives the results for polynomials in two variables. Now the vertical column

represents the total degree of the polynomial $P(x, y)$. This means that if m is the total degree of $P(x, y)$ the highest power of both x and y appearing in P is m. In particular $P(x, y)$ can have no more than $(m + 1)^2$ terms and will often have that many since zero is a not-so-random random number. Examining Table II we see that in all cases the modular method outstripped the other two rather quickly. In fact, for total degree $=3$, 4, and 5 the modular method was initially faster and became more so for successively higher powers. Computation limits were set at 4 min $= 240{,}000$ msec of computing so we see that often the modular method permitted the calculation of one or two more powers. This advantage will become even better as greater powers are tried.

Table III is set up exactly as Table II, but now we have the results for polynomials in three variables. Again the modular method is better than the other two. However, because of the large growth in the number of terms, practical limits in terms of core size as well as computing time are soon reached. In any case Tables II and III do show that the modular method becomes superior for relatively small powers and the theoretical analysis indicates that this advantage will grow larger as bigger problems are attempted.

TABLE I

Univariate Polynomials (times in milliseconds)

Degree	Methods	Powers: 2	3	4	5	6
2	BIN	33	66	116	150	285
	ITER	17	66	83	200	266
	MOD	66	83	166	283	383
3	BIN	50	100	216	283	382
	ITER	50	117	183	300	416
	MOD	100	114	309	432	583
4	BIN	83	167	249	433	749
	ITER	67	150	299	549	832
	MOD	116	200	365	518	1098
5	BIN	83	266	432	666	1131
	ITER	117	283	483	849	1198
	MOD	167	250	815	1365	1496
6	BIN	150	449	649	982	1548
	ITER	149	400	816	1148	1664
	MOD	183	416	882	1064	1495

TABLE II

Bivariate Polynomials (times in milliseconds)

Total degree	Methods	Powers: 2	3	4	5	6	7	8	9	10	11
2	BIN	366	965	2230	4609	9152	14476	23113	31250	46975	77759
	ITER	349	848	2064	4559	7738	11548	18703	25593	35394	49221
	MOD	400	866	3112	4509	7821	10300	15642	22831	29019	36304
3	BIN	892	2945	7880	16174	34794	59587	>180000			
	ITER	857	2716	7289	14058	25193	41184	67675	>180000		
	MOD	732	1597	6439	10616	25459	36068	70038	125454	>180000	
4	BIN	1930	6406	17772	37340	89889	>200000				
	ITER	1747	5836	19352	36073	65528	111156	>200000			
	MOD	1314	3095	13322	21448	51367	76461	>200000			
5	BIN	3411	14177	39403	81320	>200000					
	ITER	3111	13152	38489	77176	141474	>200000				
	MOD	2113	10999	21615	55910	98662	>200000				
6	BIN	5923	25892	>210000							
	ITER	5890	24058	>200000							
	MOD	3211	16707	34345	93135	>200000					

TABLE III

Trivariate Polynomials (times in milliseconds)

Total degree	Methods	Powers: 2	3	4	5	6	7
2	BIN	2562	11998	46875	>240000		
	ITER	2030	10649	35976	90006	>180000	
	MOD	2263	6537	28837	56816	112073	>200000
3	BIN	12280	68872	>240000			
	ITER	11481	65801	129832	>240000		
	MOD	6739	42365	89322	192621	>240000	
4	BIN	40568	>200000				
	ITER	39520	>200000				
	MOD	12882	163840	>200000			

REFERENCES

[HEI72] L. E. HEINDEL, "Computation of Powers of Multivariate Polynomials over the Integers," *J. Comput. System Sci.* **1** (1972), pp. 1–8.

[COL69] G. 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, Madison, June 1969.

[BRO71] W. S. BROWN, "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors," *J. Assoc. Comput. Mach.* **18** (1971), 478–504.

[COL71] G. E. COLLINS, "The Calculation of Multivariate Polynomial Resultants," *J. Assoc. Comput. Mach.* **18** (1971), 515–532.

[KNU69] D. E. KNUTH, "The Art of Computer Programming," Vol. 2: Seminumerical Algorithms., 2nd edition, Addison-Wesley, Reading, Mass. 1969.

[GEN71] W. M. GENTLEMAN, "Optimal Multiplication Chains for Computing a Power of a Symbolic Polynomial," *Math. Comp.* **26** (1972), 935–940.