✓

The Efficient Calculation
of Powers of Polynomials

by

Ellis Horowitz

Computer Science Department
Cornell University
Ithaca, New York 14850

TR 72 - 129

The Efficient Computation of Powers of Polynomials

by

Ellis Horowitz

Abstract

Suppose we are given a polynomial $P(x_1, \dots, x_r)$ in $r \geq 1$ variables, let m bound the degree of P in all variables x_i , $1 \leq i \leq r$, and we wish to raise P to the n^{th} 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 $P, P^2, \dots, P^{n-1}, P^n$ 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 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 n^{th} 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 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 where the number of nonzero terms remain s severly bounded; 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 Section 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 Section 3 the new algorithm will be given plus an analysis of its computing time. In Section 4 empirical results comparing these several methods will be given. Finally in Section 5 some observations on even faster but only theoretical 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 $f(x_1, \dots, x_n)$ and $g(x_1, \dots, x_n)$ be real-valued

functions defined on some common domain D . Then $f = O(g)$ if there exists a positive real number c such that

$$|f(x_1, \dots, x_n)| \leq c|g(x_1, \dots, x_n)| \text{ for all } (x_1, \dots, x_n) \in D.$$

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 $d \geq e$. However, their method seems to be of more theoretical interest rather than practically useful and so 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 $P(x_1, \dots, x_r)$ be a polynomial in $r \geq 1$

variables with integer coefficients. Then, $\text{norm}(P)$ is the sum of the absolute values of the numerical coefficients of P .

Definition 2.3 : Let $M_r(d, m) = \{P(x_1, \dots, x_r) : \deg(P) \text{ in } x_r + 1 \leq m \text{ and } d = \text{norm}(P)\}$. $M_r(d, m)$ 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 $P(x_1, \dots, x_r), Q(x_1, \dots, x_r) \in M_r(d, m)$. 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 non-zero 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. 9.

We note that if instead P and Q had coefficients in $\text{GF}(p)$, 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 $\text{GF}(p)$ 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 $\text{GF}(p)$. 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 $A = \sum_{1 \leq i \leq m^r} a_i x_1^{e_{i1}} \dots x_r^{e_{ir}}$ and p a single precision prime. Then an upper bound on the time to compute $A^* = \sum_{1 \leq i \leq m^r} (a_i \bmod p) x_1^{e_{i1}} \dots x_r^{e_{ir}}$ is $O(m^r \ln d)$.

Proof: There are at most m^r non-zero 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 $I[x_1, \dots, x_r] \rightarrow GF(p)[x_1, \dots, x_r]$ and this mapping satisfies the properties of a homomorphism.

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

Proof: [COL69], p.17.

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

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; 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 $x-b_1, \dots, x-b_n$. We now present these two methods as they would be applied to multivariate polynomials.

Iterative Chinese Remainder Algorithm

Input: $A_1(x_1, \dots, x_r)$ with coefficients in $GF(p)$

$A_2(x_1, \dots, x_r)$ with coefficients in $GF(q)$

Output: $A_3(x_1, \dots, x_r)$: $A_3 \equiv A_1 \pmod{p}$

$A_3 \equiv A_2 \pmod{q}$

- 1) $A_2^* \leftarrow A_2 \pmod{p}$;
- 2) $q^* \leftarrow q \pmod{p}$;
- 3) $C \leftarrow (A_1 - A_2^*)/q^*$;
- 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 $O(m^r)$.

Iterative Interpolation

Input: $A_1(x_1, \dots, x_{r-1})$ with coefficients in $GF(p)$;

$b \in GF(p)$;

$A_2(x_1, \dots, x_r)$ with coefficients in $GF(p)$;

$Q(x_r)$, $\text{degree}(Q) \leq k$;

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

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

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

4) $A_3 \rightarrow CQ + A_2$;

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 $O(m)$.

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) $q \leftarrow \lfloor n/2 \rfloor; r \leftarrow n-2q; n \leftarrow q; \text{ if } r=0, \text{ go to (4);}$
- 3) $R \leftarrow R \cdot Z$
- 4) If $n=0$ then return(R);
 else $Z \leftarrow Z \cdot Z; \text{ go to (2);}$

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 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

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

The time for the iterative method is

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

Thus we see that the iterative scheme already overtakes 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 $P(x_1, \dots, x_r)$ raised to the n^{th} 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 $P(x_1, \dots, x_r)$, $\deg(P) \text{ in } x_i + 1 \leq m$
|Integer coefficients| $\leq d$ coefficients in $GF(p)$

Method

Binary	$m^{2r} n^{2r+2} (\ln d)^2$	$m^{2r} n^{2r}$
Iterative	$m^{2r} n^{r+2} (\ln d)^2$	$m^{2r} n^{r+1}$

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

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

- 1) [Initial Case] If $r=0$, compute $R^* \leftarrow (P^*)^n$ using the binary method; end.
- 2) [Initialize] $m_r \leftarrow \deg(P)$ in x_r ; $C(x_1, \dots, x_{r-1}) \leftarrow 0$; $D(x_r) \leftarrow 1$;
[Interpolation] $b \leftarrow -1$;
- 3) [Choose next] $b = b + 1$; if $b = p$ then stop.
[point]
- 4) [Evaluation] $\bar{P}(x_1, \dots, x_{r-1}) \leftarrow P^*(x_1, \dots, x_{r-1}, b)$;
[Homomorphism] If $\deg(\bar{P})$ in $x_r < m_r$ then go to (3).
- 5) [Recursion] $R^*(x_1, \dots, x_{r-1}) \leftarrow \text{CPPOWER}(\bar{P}, n)$;
- 6) [Interpolate] $C(x_1, \dots, x_r) \leftarrow \frac{R^*(x_1, \dots, x_{r-1}) - C(x_1, \dots, x_{r-1}, b)}{D(b)} \cdot D$
 $+ C(x_1, \dots, x_r)$;
- 7) [Done?] $D(x_r) \leftarrow (x_r - b)D(x_r)$;
If $\deg(D) \leq m_r n$ then go to (3);
- 8) [end !] $R^*(x_1, \dots, x_r) \leftarrow C(x_1, \dots, x_r)$; end.

Inorder to greatly simplify the analysis and following the procedure used by W.S. Brown in [BR071] let us assume that $(\deg(P^*) \text{ in } x_i + 1) \leq m$ for $1 \leq i \leq r$. Thus P^* has at most m^r terms all of whose numerical coefficients are elements of $\text{GF}(p)$.

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

Proof: If $r=0$, P^* is an element of $\text{GF}(p)$ 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

$(\deg(R^*) \text{ in } x_r + 1) \leq mn$. Each execution of step (4) for which $\deg(\bar{P}) \text{ in } x_r < m_{r-1}$ corresponds to a distinct element $b \in GF(p)$ 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

<u>step</u>	<u>time</u>
(3)	$O(mn)$
(4)	$O(m^{r+1}n)$ by Theorem 2.3
(5)	$O(mnT_{r-1}(m))$
(6)	$\sum_{1 \leq i \leq mn} im^{r-1} = O(m^{r+1}n^2)$ by Theorem 2.5

Thus the total time for CCPOWER is

$$\begin{aligned}
 T_r(m) &= m^{r+1}n^2 + mnT_{r-1}(m) \\
 &= m^{r+1}n^2 + m^{r+1}n^3 + m^2n^2T_{r-2}(m) \\
 &= \dots = m^{r+1}(n^2 + \dots + n^{r+1}) + (mn)^r T_0(m) \\
 &= O((mn)^{r+1} + (mn)^r \log n) = O((mn)^{r+1})
 \end{aligned}$$

since $n > \log_2 n$.

Now we give the algorithm for polynomials with integer coefficients.

Algorithm PPOWER

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

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

- 1) [Initialize] $d \leftarrow \text{norm}(P); m \leftarrow \deg(p) \text{ in } x_r; i \leftarrow 0;$
 $f \leftarrow (nd)^n;$
- 2) [Next Prime] $i \leftarrow i+1;$ Choose next prime, p_i ; If none left, stop.
- 3) [Modular] $P^*(x_1, \dots, x_r) \leftarrow P(x_1, \dots, x_r) \bmod p_i;$
[Homomorphism] If $\deg(P^*) \text{ in } x_r < m$, go to (2).
- 4) $[R^* = (P^*)^n]$ $R^*(x_1, \dots, x_r) \leftarrow \text{CPPOWER}(P^*, n)$
- 5) [Chinese] If $i=1$, $R \leftarrow R^*$; $q \leftarrow p_i$; go to (2);
[Remainder] Otherwise find $D(x_1, \dots, x_r)$ using the
iterative Chinese Remainder algorithm
such that
 $D(x_1, \dots, x_r) \equiv R^*(x_1, \dots, x_r) \bmod p_i;$
 $D(x_1, \dots, x_r) \equiv R(x_1, \dots, x_r) \bmod q;$
- 6) [Done ?] $R \leftarrow D; q \leftarrow p_i q$; If $q \leq f$, 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:

<u>step</u>	<u>time</u>
(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:

	<u>Binary</u>	<u>Iterative</u>	<u>Homomorphism</u>
Single Precision Coefficients	$(mn)^{2r}$	$m^{2r} n^{r+1}$	$(mn)^{r+1}$
Multiple Precision Coefficients	$m^{2r} n^{2r+2} e^2$	$m^{2r} n^{r+2} e^2$	$m^{r+1} n^{r+2} e^2$

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, II and 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 3 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 3 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 2 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 =4, 5, and 6 the modular method was initially faster and became more so for successively higher powers. Computation limits were set at 4 minutes = 240,000 milliseconds 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 3 variables. Again the modular method is better than the other 2. 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

<u>Total degree</u>	<u>Methods</u>	<u>Powers:</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>
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

<u>Total degree</u>	<u>Methods</u>	<u>Powers:</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>
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			

5. Even Lower Bounds

Recently there has been intensive work and success on developing algorithms for basic arithmetic operations whose asymptotic bounds are as low as possible. In our case we are especially interested in an algorithm which multiplies two e -precision numbers in time $M(e) = O(e \log e (\log \log e))$ which is due to S. Schonhage and V. Strassen, see [KNU69], p. 269. One can directly generalize their method so that one can multiply two m -degree polynomials with e -precision coefficients in time $M(me)$. This is done simply by considering the polynomials' coefficients to be a long integer and inserting a sufficient number of zeros to prevent carries. This bound of $O(M(me))$ is considerably better than the bound we have been using for univariate polynomial multiplication, namely $O(m^2 e^2)$. One can now ask what would be the computing time for forming $P(x)^n$ using this fast polynomial multiplication algorithm.

If we use the iterative method then at the i^{th} multiplication we would be multiplying a polynomial of degree im and precision ie by a polynomial of degree m and precision e . Thus the total time for the iterative method is

$$\begin{aligned} \sum_{1 \leq i \leq n-1} M(imie) &= n^3 me (\log n me) (\log \log n me) \\ &= n^2 M(mne). \end{aligned}$$

The corresponding time for the straightforward iterative method is $m^2 n^3 e^2$, and the ratio is $\frac{\log mne (\log \log nme)}{me}$.

Now instead if we use the binary method at the i^{th}

stage we must multiply polynomials of degree at most $2^{i-1}m$ and precision at most $2^{i-1}e$. Thus a bound on the total time would be

$$\begin{aligned} & \sum_{1 \leq i \leq \log_2 n} (n/2^i) M(2^{2(i-1)} m e) \\ & < n m e (\log m e) (\log \log n m e)^2 \sum_{1 \leq i \leq \log_2 n} 2^{i-1} (i-1) \\ & \leq n^2 m e (\log n m e)^2 (\log \log n m e)^2. \end{aligned}$$

The ratio of the binary to iterative methods then is

$$\frac{(\log n m e) (\log \log n m e)}{n}.$$

We can continue this work by analyzing a modular method which leads to a worse time than the binary method. Thus this straightforward analysis would say that the binary method has a slightly better asymptotic computing time bound. But more analysis can certainly be done on this level.

References

- [HEI72] Heindel, L.E., "Computation of Powers of Multivariate Polynomials over the Integers", J.CSS, (to appear).
- [COL69] Collins, G.E., Heindel, L.E., Horowitz, E., McClellan, M.T., Musser, D.R., The SAC-1 Modular Arithmetic System., U. of Wisconsin Technical Report No. 10, Madison, June 1969.
- [BRO71] Brown, W.S., "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors", J.ACM, Vol. 18, No. 4, Oct 1971, pp. 478-504.
- [COL71] Collins, G.E., "The Calculation of Multivariate Polynomial Resultants", J.ACM, Vol. 18, No. 4, Oct 1971, pp. 515-532.
- [KNU69] Knuth, D.E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms., 2nd edition, Addison-Wesley, Reading, Mass. 1969.
- [GEN71] Gentleman, W.M., "Optimal Multiplication Chains for Computing a Power of a Symbolic Polynomial", University of Waterloo Research Report CSRR2035, Waterloo, Ontario, Canada, March 1971.