A Practical Univariate Polynomial Composition Algorithm

William Hart, Andrew Novocin

A Practical Univariate Polynomial Composition Algorithm

William Hart[1†] and Andrew Novocin[2‡]

[1] *University of Warwick*
Mathematics Institute
Coventry CV4 7AL, UK
W.B.Hart@warwick.ac.uk
[2] *INRIA-ENSL*
46 Allée d'Italie
69364 Lyon Cedex 07, France
andy@novocin.com

received 19 April 2010, revised 19th April 2010, accepted soon.

We revisit a divide-and-conquer algorithm, originally described by Brent and Kung for composition of power series, showing that it can be applied practically to composition of polynomials in $\mathbb{Z}[x]$ given in the standard monomial basis. We offer a complexity analysis, showing that it is asymptotically fast, avoiding coefficient explosion in $\mathbb{Z}[x]$. The algorithm is straightforward to implement and practically fast, avoiding computationally expensive change of basis steps of other polynomial composition strategies. The algorithm is available in the open source FLINT C library and we offer a practical comparison with the polynomial composition algorithm in the MAGMA computer algebra system.

Keywords: Complexity Analysis, Symbolic Computation, Polynomial Composition, Divide and Conquer, Practical Implementation

Introduction

Univariate integer polynomials are important basic objects for computer algebra systems. Given two polynomials $f, g \in \mathbb{Z}[x]$ the polynomial composition problem is to compute $f(g(x)) \in \mathbb{Z}[x]$. Standard approaches include Horner's method (9), ranged Horner's method (which we describe in section 1.1), algorithms for composition of polynomials in a Bernstein basis (see (2)), and algorithms based on point evaluation followed by coefficient interpolation (see (11)).

†Author was supported by EPSRC Grant number EP/G004870/1
‡Author was partially supported by ANR project LaRedA

Our Contribution. We present and analyze the divide-and-conquer technique of Brent and Kung (5), originally a component of a power series composition algorithm, applied to the composition of two polynomials $f, g \in \mathbb{Z}[x]$ given in the standard monomial basis. We give a theoretical complexity bound which is softly optimal in the size of the output and show that the algorithm is highly practical.

Problem Statement:

Given: $f = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ and $g = b_m x^m + \cdots + b_0$ in $\mathbb{Z}[x]$.
Find: a full expansion of $h = f(g(x))$

Assumptions: In our analysis we assume the use of fast arithmetic (see (1)), which is available in FLINT (8). Also, only for the simplicity of bit-complexity analysis, we will assume throughout that coefficients of f and g are of $\mathcal{O}(m)$ bits, where m is the degree of g, the inner polynomial in the composition $f(g)$. We note that the algorithm still works when the coefficients are larger, but depending on the implementation of the fast polynomial arithmetic, the bit complexity will go up by some factor which is a quasilinear expression in the size of the coefficients.

The algorithm is simple to implement and works in the standard monomial basis. We will show that the algorithm performs well in practice by providing timings against the MAGMA computer algebra system (6). We also provide a theoretical complexity analysis showing that, in the worst case, the algorithm uses $\mathcal{O}(nm\log(n)\log(nm))$ operations in $\mathbb{Z}$ and has a bit-complexity of $\mathcal{O}(n^2m^2\log(nm))$.

Assuming that $h = f(g)$ does not have special structure (i.e. h is dense with few cancellations) then this output has $\mathcal{O}(nm)$ coefficients each with bit-length $\mathcal{O}(nm)$. Simply writing down the output requires $\mathcal{O}(n^2m^2)$ CPU-operations making our theoretical bound optimal, up to a factor $\mathcal{O}(\log(mn))$.

Related works. The presented algorithm is an application of the divide-and-conquer technique of Brent and Kung (5), originally developed as a component of an algorithm for composition of power series. In the original application the bit complexity was not considered, however we show that the algorithm is asymptotically fast for polynomial composition in $\mathbb{Z}[x]$. The algorithm was rediscovered while implementing the number theory library FLINT (8), and we are grateful to Joris van der Hoeven for pointing out its first occurrence in the literature.

In (10) an algorithm is presented which is asymptotically fast for composition of polynomials in a Bernstein basis. However for polynomials presented in the usual monomial basis one must first perform a conversion to Bernstein basis to make use of this algorithm.

Conversion of orthogonal polynomials can be done in time $\mathcal{O}(n\log^2 n\log\log n)$, assuming the use of Fast Fourier Transform techniques (see (3)), however Bernstein bases are not orthogonal.

A standard method for converting from a Bernstein basis to a monomial basis involves computing a difference table, which costs $\mathcal{O}(n^2)$ operations for a polynomial of length n in the Bernstein basis (see (4, Sect.2.8)). Thus to convert the eventual solution from Bernstein basis to monomial basis in our case will cost $\mathcal{O}((mn)^2)$ operations, each of which involves a subtraction of quantities of $\mathcal{O}(mn)$ bits. Thus the total bit complexity of the conversion alone is already significantly greater than that of our algorithm.

A different method is given in (11, Prob 3.4.2). In this method, $K = 2^k$ is computed such that $mn+1 \leq K < 2mn + 2$. If possible compute ω, a primitive K^{th} root of unity, and the $K = 2^k$ points, ω^i for all $i = 0, \ldots, K-1$. Evaluate $h = f(g)$ at those K points (using fast arithmetic) and interpolate the coefficients of h. If a K^{th} root of unity is unavailable then use K other values for evaluation. Pan suggests that this method uses $\mathcal{O}(nm[\log(n) + \log(m) + \log^2(n)])$ operations in $\mathbb{Z}$ when roots of unity are available and $\mathcal{O}(nm[\log^2(nm)])$ operations in $\mathbb{Z}$ otherwise.

In order to apply Pan's method to polynomials in $\mathbb{Z}[x]$ one may work in a ring $\mathbb{Z}/p\mathbb{Z}$ where $p = 2^{2K}+1$. There are then sufficiently many roots of unity, and moreover, the coefficients of $f(g(x))$ may be identified by their values $\pmod p$.

Interpolation of h is performed using the inverse FFT. To evaluate $f(g(x))$ at the roots of unity, Pan first evaluates $g(x)$ at the roots of unity using the FFT. This gives K values at which $f(x)$ must then be evaluated.

The Moenck-Borodin algorithm (see Algorithm 3.1.5 of (11)) evaluates $f(x)$ of degree n at n arbitrary points in $\mathcal{O}(n\log^2 n)$ operations. If the points are $w_1, w_2, \ldots, w_n$, one first reduces $f(x)$ mod $(x - w_1)(x - w_2)\cdots(x - w_n)$. One then splits this product into two balanced halves and reduces mod each half separately. This process is repeated recursively until one has the reduction of $f(x)$ modulo each of the factors $(x - w_i)$.

Of the $\mathcal{O}(n\log^2 n)$ operations there are $\mathcal{O}(n\log n)$ multiplications. Each can be performed in our case using fast arithmetic in $\mathcal{O}(mn\log mn)$ bit operations (up to higher order log factors).

As we have $\mathcal{O}(mn)$ roots of unity to evaluate at, not n, we must perform this whole operation $\mathcal{O}(m)$ times. Thus the bit complexity of Pan's algorithm is $\mathcal{O}((mn)^2\log n\log mn)$, which exceeds that of our algorithm by a factor of $\log n$.

Road map. In section 1 we present Horner's method and Ranged Horner's method along with a complexity analysis. In section 2 we present the algorithm itself. In subsection 2.1 we provide a worst-case asymptotic complexity analysis. Finally, in section 3 we provide practical timings of our FLINT implementation and a comparison with MAGMA's polynomial composition algorithm.

Notations and notes: Given two polynomials of length n, with coefficients of n bits, the Schönhage and Strassen Algorithm (SSA) for multiplying polynomials has a bit complexity of $\mathcal{O}(n^2\log(n)\log\log n)$ (for more see (7, Sect.8.3)). We will ignore $\log\log n$ factors throughout the paper. Various standard tricks allow us to multiply polynomials of degree n with coefficients of m bits in time $\mathcal{O}(mn\log(mn))$ using SSA (again ignoring lower order log factors). For each algorithm we given both the bit-complexity model cost and the number of operations in $\mathbb{Z}$.

1 Horner's Method

In this section we apply Horner's algorithm for evaluating a polynomial f at a point p, to the problem of polynomial composition.

Horner's Evaluation Algorithm
Given: $f = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ in $\mathbb{Z}[x]$, p in $\mathbb{Z}$.
Find: $ans := f(p)$ in $\mathbb{Z}$

1. $ans := a_n$
2. For $i = n-1$ down to $i = 0$ do:
   (a) $ans := ans \cdot p + a_i$
3. Return ans

This algorithm computes $a_n p^n + a_{n-1}p^{n-1} + \cdots + a_0$ using n multiplications and n additions. When the point p is a polynomial g, n polynomial multiplications and n polynomial additions are performed.

1.1 Ranged Horner Composition

We will need a variant of this approach which we call Ranged Horner's algorithm for polynomial composition. We restrict the algorithm to use only l coefficients of f, from a_i to $a_{i+\ell-1}$, and replace p by a polynomial g. If one chooses $i = 0$ and $\ell = n + 1$ then this algorithm returns a complete expansion of $h = f(g)$. The algorithm is always a direct application of Horner's method to the degree $\ell - 1$ polynomial $F := a_{i+\ell-1}x^{\ell-1} + \cdots + a_{i+1}x + a_i$.

Algorithm 1 ***Ranged Horner Compose***

Input: *$f, g \in \mathbb{Z}[x]$, i, a starting index, and ℓ the length of the ranged composition.*

Output: *An expansion of $F(g) := a_{i+\ell-1}g^{\ell-1} + \cdots + a_{i+1}g + a_i$, where F is f divided by x^i without remainder then reduced modulo x^ℓ, a shifted truncation of f.*

1. $ans := a_{i+\ell-1}$
2. For $j = \ell - 2$ down to $j = 0$ do:
   (a) $ans := ans \cdot g$
   (b) $ans := ans + a_{i+j}$
3. Return ans

1.2 Bit-Complexity

We will now outline the bit-complexity analysis of Ranged Horner Composition.

Theorem 1 *Algorithm 1 terminates after $\mathcal{O}(\ell^2 m \log(\ell m))$ operations in $\mathbb{Z}$ with a bit-complexity bound of $\mathcal{O}(\ell^3 m^2 \log(\ell m))$ CPU operations.*

Proof:

Let us analyze the cost of the k^{th} loop where $k = 1, \ldots, \ell - 1$. First we compute the degree and coefficient size of ans in the k^{th} loop.

Lemma 1 *At the beginning of the k^{th} loop of step 2 in Algorithm 1 we have the degree of $ans = (k-1)m$ and $\| ans \|_\infty \leq 2^{\mathcal{O}(km+(k-1)\log(m+1))}$.*

Proof: The degree of ans begins at 0 and increases by m in each loop giving degree $(k-1)m$ at the beginning of the k^{th} loop.

Now for an arbitrary loop let's suppose that $\| ans \|_\infty \leq 2^x$ and $ans = c_N x^N + \cdots + c_0$ where N is the current degree of ans. Recall that $g = b_m x^m + \cdots b_0$ and $\| g \|_\infty \leq 2^m$. The product $ans \cdot g$ can be written as

$$\sum_{s=0}^{s=N+m} x^s [\sum_{\{0 \leq i \leq N, 0 \leq j \leq m | s=i+j\}} (c_i \cdot b_j)].$$

In this form it can be seen that the largest coefficients of $ans \cdot g$ are the sum of $m+1$ numbers of norm $\leq 2^{m+x}$. Thus after this loop the coefficients are boundable by $2^{x+m+\log_2(m+1)}$. So the size of the coefficients of ans begin at m-bits and increase by $m + \log_2(m+1)$ finishing the proof of the lemma. □

Now using fast polynomial multiplication the bit complexity of loop k is $\mathcal{O}(k^2 m(m+\log(m)\log(km))$ and uses $\mathcal{O}(km\log(km))$ operations in $\mathbb{Z}$. Summing this over $k = 1, \ldots, \ell-1$ gives a bit-complexity of $\mathcal{O}(\ell^3 m^2 \log(\ell m))$ and $\mathcal{O}(\ell^2 m \log(\ell m))$ operations in $\mathbb{Z}$. □

2 Divide and Conquer Algorithm

In this section we describe the main algorithm for polynomial composition. First we divide f of degree n into $k_1 := \lceil (n+1)/\ell \rceil$ sub-polynomials of length ℓ for some experimentally derived (and small) value of ℓ such that:

$$f := f_0 + f_1 \cdot x^{\ell} + f_2 \cdot x^{2\ell} + \cdots + f_{k_1-1} \cdot x^{(k_1-1)\ell}.$$

In the first iteration of the algorithm we compute the k_1 compositions, $h_{1,i} := f_i(g)$ for $0 \leq i < k_i$ using (Ranged) Horner's method and we also compute g^{ℓ}. In the i^{th} iteration we start with $g^{2^{i-2}\ell}$ and compute the $k_i := \lceil (k_{i-1})/2 \rceil$ polynomials: $h_{i,j} := h_{i-1,2j} + g^{2^{i-2}\ell} \cdot h_{i-1,2j+1}$ then compute $g^{2^{i-1}\ell}$. Thus in iteration i our target polynomial $h = f(g)$ can be written:

$$h_{i,0} + h_{i,1} \cdot (g^{2^{i-1}\ell}) + h_{i,2} \cdot (g^{2^{i-1}\ell})^2 + \cdots + h_{i,k_i-1} \cdot (g^{2^{i-1}\ell})^{k_i-1}.$$

In each iteration the number of polynomials is halved while the length of the polynomials we work with is doubled. We experimentally determined that a value of $\ell = 4$ works well in practice.

Algorithm 2 ***Polynomial Composition Algorithm***
Input: $f, g \in \mathbb{Z}[x]$
Output: *An expansion of* $h := f(g)$

1. let $\ell := 4$, $i := 1$, and $k_i := \lceil \frac{n+1}{\ell} \rceil$
2. for $j = 0, \ldots, k_i - 1$
   (a) compute $h_{i,j} :=$ `Algorithm 1`$(f, g, j\ell, \ell)$
3. compute $G := g^{\ell}$.
4. while $(k_i > 1)$ do:
   (a) $k_{i+1} := \lceil k_i/2 \rceil$;
   (b) for $j = 0, \ldots, k_{i+1} - 1$ do:
      i. $h_{i+1,j} := h_{i,2j} + h_{i,2j+1} \cdot G$.
      ii. clear $h_{i,2j}$ and $h_{i,2j+1}$
   (c) if $k_{i+1} > 1$ then $G := G^2$

(d) $i := i + 1$

5. return $h := h_{i,0}$

2.1 Complexity Analysis

Theorem 2 *Algorithm 2 terminates after $\mathcal{O}(nm\log(n)\log(mn))$ operations in $\mathbb{Z}$ with a bit-complexity bound of $\mathcal{O}(n^2m^2\log(nm))$ CPU operations.*

Proof: Although we chose $\ell = 4$ we will make this proof using any constant value of ℓ. The cost of step 2 is that of $\lceil (n+1)/\ell \rceil$ calls to Algorithm 1 using ℓ coefficients. Thus theorem 1 tells us that step 2 costs $\mathcal{O}(nm\log(m))$ operations in $\mathbb{Z}$ with bit complexity bound $\mathcal{O}(nm^2\log(m))$.

Step 3 involves a constant number of multiplications (or repeated squarings) of g. By using the same logic as the proof of lemma 1 these multiplications are of polynomials with degree $\mathcal{O}(m)$ and coefficients of $\mathcal{O}(m + \log(m))$ bits, this gives $\mathcal{O}(m\log(m))$ operations in $\mathbb{Z}$ and bit complexity bound of $\mathcal{O}(m^2\log(m))$ for step 3.

In the i^{th} loop of step 4 creating the $h_{i+1,j}$ involves k_{i+1} polynomial multiplications each of degree $\mathcal{O}(2^{i-2}m\ell)$ polynomials with coefficients bounded of $\mathcal{O}(2^{i-1}m\ell)$ bits (and k_{i+1} polynomial additions). This costs $\mathcal{O}(k_{i+1}2^i m\log(2^i m))$ operations in $\mathbb{Z}$ with bit-complexity bound $\mathcal{O}(k_{i+1}2^{2i}m^2\log(2^i m))$. The cost of the i^{th} iteration of step 4c involves squaring a polynomial of degree $m\ell 2^{i-1}$ and whose coefficients are smaller than $m\ell 2^i$. The cost of this is $\mathcal{O}(m2^i\log(m2^i))$ operations in $\mathbb{Z}$ and $\mathcal{O}(m^2 2^{2i}\log(2^i m))$ bit operations. It can be shown without much difficulty that $k_i \leq (n+1)/(\ell\cdot 2^{i-1})+1$. To sum these costs over the $\mathcal{O}(\log(n))$ iterations of step 4 gives $\mathcal{O}(\sum_{i=1}^{\log(n)} k_{i+1}2^i m[i+\log(m)])$ which is $\mathcal{O}(nm[\log(n)^2 + \log(n)\log(m)])$ operations in $\mathbb{Z}$ and a bit-complexity bound of $\mathcal{O}(\sum_{i=1}^{\log(n)} 2^{2i}m^2[i+\log(m)])$ which is $\mathcal{O}(m^2 n\cdot\sum_{i=1}^{\log(n)}[2^i i + 2^i\log(m)])$. It is trivial to show via induction that $\sum_{i=1}^{k} 2^i i = 2 + 2^{k+1}(k-1)$. This gives the bit-complexity bound as $\mathcal{O}(m^2 n[n\log(n) + n\log(m)])$ proving the theorem. □

3 Practical Timings

In this section we present a timing comparison of the main algorithm as implemented in FLINT and MAGMA's polynomial composition algorithm. These tests are provided as evidence that our algorithm is indeed practical. These timings are measured in seconds and were made on a 2400MHz AMD Quad-core Opteron processor, using gcc version 4.4.1 with the -O2 optimization flag, although the processes only utilized one of the four cores. Each composition performed is of a polynomial, f, of length n with randomized coefficients of bit-length $\leq m$, and a polynomial, g, of degree m with randomized coefficients of bit-length $\leq m$ and returns an expansion of $h = f(g)$.

Timings in FLINT

$n\backslash m$	20	40	80	160	320	640	1280
20	.0009	.0038	.016	.077	0.41	1.96	8.9
40	.0036	.015	.071	0.40	2.0	9.4	
80	0.02	.072	.412	2.09	9.63		
160	0.072	0.415	2.1	9.7			
320	0.44	2.1	9.7				
640	2.05	9.64					
1280	9.46						

We also compared these timings with the function

$$(mn)^2 \ln(mn)/(.95 \cdot 10^9).$$

In this case the function accurately models the given timings, in all cases, up to a factor which varied between 0.71 and 1.29. This model matches our bit-complexity bound given in theorem 2.

Timings in MAGMA

$n\backslash m$	20	40	80	160	320	640	1280
20	.006	.053	.160	.630	2.55	12.47	64.0
40	.04	.32	1.09	4.67	21.7	110	
80	.47	2.0	8.52	38.0	196.4		
160	3.6	15	70	360			
320	28	133	659				
640	238	1267					
1280	2380						

We compared the MAGMA timings with the function

$$n^3 m^2 \ln(mn)/(2.94 \cdot 10^9).$$

This function accurately models the given timings, in all cases, up to a factor which varied between 0.54 and 1.46. This model matches our estimate for Horner's method given by theorem 1 in the case when $\ell = n$.

References

[1] D. Bernstein, *Multiprecision Multiplication for Mathematicians*, accepted by Advances in Applied Mathematics find at http://cr.yp.to/papers.html#m3, 2001.

[2] C. de Boor *B-Form Basics*, Geometric Modeling: Algorithms and New Trends, SIAM, Philadelphia (1987), pp. 131–148.

[3] A. Bostan and B. Salvy, *Fast conversion algorithms for orthogonal polynomials*, Preprint.

[4] H.Prautzsch, W.Boehm, and M.Paluszny *Bézier and B-Spline Techniques*, Springer, 2002.

[5] R. Brent and H.T. Kung, $\mathcal{O}((nlogn)^3/2)$ *Algorithms for composition and reversion of power series*, Analytic Computational Complexity, Academic Press, New York, 1975, pp. 217-225.

[6] J. J. Cannon, W. Bosma (Eds.) *Handbook of Magma Functions*, Edition 2.13 (2006)

[7] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[8] W. Hart *FLINT*, open-source C-library. `http://www.flintlib.org`

[9] D. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997, pp. 486–488.

[10] W. Liu and S. Mann *An analysis of polynomial composition algorithms*, University of Waterloo Research Report CS-95-24, (1995).

[11] V. Pan *Structured matrices and polynomials: unified superfast algorithms*, Springer-Verlag, 2001, pg. 81.