Practical divide-and-conquer algorithms for polynomial arithmetic

William Hart[1]* and Andrew Novocin[2]**

[1] University of Warwick, Mathematics Institute, Coventry CV4 7AL, UK
W.B.Hart@warwick.ac.uk – http://maths.warwick.ac.uk/~masfaw
[2] LIP/INRIA/ENS, 46 allée d'Italie, F-69364 Lyon Cedex 07, France
Andrew.Novocin@ens-lyon.fr – http://andy.novocin.com/pro

Abstract. We investigate two practical divide-and-conquer style algorithms for univariate polynomial arithmetic. First we revisit an 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 $\mathbb{Z}[x]$. Secondly we provide an improvement to Mulders' polynomial division algorithm. We show that it is particularly efficient compared with the multimodular algorithm. The algorithms are straightforward to implement and available in the open source FLINT C library. We offer a practical comparison of our implementations with various computer algebra systems.

Introduction

Univariate integer polynomials are important basic objects for computer algebra systems. In this paper we investigate two algorithms for univariate polynomial arithmetic over $\mathbb{Z}$. In particular, we study divide-and-conquer style algorithms for composition and division of polynomials.

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 [10], 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 [13]).

Given $f, g \in \mathbb{Z}[x]$ the *division problem* is to find polynomials $q, r \in \mathbb{Z}[x]$ such that $f = gq + r$ where the $\deg(r) <= \deg(f)$, but the coefficients of terms of r whose degree is at least $\deg(g)$ are reduced modulo the leading coefficient of g.

Standard approaches to the division problem are the naive $\mathcal{O}(n^2)$ "school-book" method, a divide-and-conquer approach based on the middle product and a multimodular approach (see for example Victor Shoup's NTL [14] which employs the latter approach). The approach we present has the advantage of being simpler to implement than the middle product approach with comparable performance but much better performance than the "school-book" or multimodular approaches in real-world cases.

Some important applications include: i) exact division, i.e. where $r = 0$, ii) division by g with leading coefficient ±1, iii) divisibility testing, i.e. to test if $f = gq$ for some $q \in \mathbb{Z}[x]$, with the algorithm returning false if (and as soon as) a non-zero coefficient is detected in the remainder r and iv) a basecase for power series division (with normalised divisor).

1 Polynomial Composition

We begin with a divide-and-conquer approach to polynomial composition.

* 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 instead 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 [9]. 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 [9], and we are grateful to Joris van der Hoeven for pointing out its first occurrence in the literature.

In [11] 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 (degree $n-1$) 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 [13, Prob 3.4.2]. In this method, $K = 2^k$ is computed such that $mn+1 \le K < 2mn+2$. 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 [13]) 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 1.1 we present Horner's method and Ranged Horner's method along with a complexity analysis. In section 1.2 we present the algorithm itself. In subsection 1.2 we provide a worst-case asymptotic complexity analysis. Finally, in section 1.3 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 $\mathbb{Z}$.

1.1 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.

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 $ans$
```

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

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

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 $\mathbb{Z}$.

1.2 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 $j = 0, \dots, k_i - 1$
   (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 $j = 0, \dots, k_{i+1} - 1$ 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 $h := h_{i,0}$
```

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 will cost $\mathcal{O}(k_{i+1}2^i m\log(2^i m))$ 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.

1.3 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

$n\backslash m$	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						

Table 1. Divide-and-conquer polynomial composition in FLINT

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 $h = f(g)$.

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.

$n\backslash m$	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						

Table 2. Polynomial composition in Magma

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 $\ell = n$.

2 Divide and conquer division

In his paper [12], Mulders describes recursive divide-and-conquer type algorithms for the *short product* of polynomials (returning only the low degree terms of the product) and the *opposite short product* (returning only the higher degree terms).

Suppose two polynomials of length at least N are multiplied, but one only wishes to compute the terms of the product of degree less than N. We will denote such a short product by $\mathrm{SM}(N)$.

A basic algorithm for computing $\mathrm{SM}(N)$ is to compute a full $N \times N$ product using a standard polynomial multiplication algorithm and discard the unwanted terms. But this is often wasteful. For example, the

Karatsuba algorithm breaks the full product up into three half sized products, but in two of the half sized products, one again doesn't require all the terms.

This leads naturally to a recursive Karatsuba-type algorithm where a short product $SM(N)$ is replaced by one full product with polynomials of half the size, $FM(N/2)$, and two short products $SM(N/2)$. At the bottom of the recursion, below some cutoff, the short products are computed using classical multiplication, computing only the required terms.

Mulders' algorithm for the short product, denoted $\mathrm{SM}_\beta(N)$ for some parameter $\frac{1}{2} \leq \beta \leq 1$, is a generalisation of this technique, also breaking the short product up into three multiplications. But this time there is a full $\beta N \times \beta N$ product $FM(\beta N)$ and two short products $\mathrm{SM}_\beta((1-\beta)N)$. The Karatsuba-type algorithm above, is the special case $\beta = \frac{1}{2}$.

In general, the recursion for Mulders' algorithm can be expressed:

$$\mathrm{SM}_\beta(N) = \mathrm{FM}(\beta N) + 2\mathrm{SM}_\beta((1-\beta)N). \tag{1}$$

This is completely general in that the full multiplications $\mathrm{FM}(\beta N)$ can be performed using any algorithm for ordinary polynomial multiplication.

When the full products are computed using Karatsuba multiplication, Mulders derives the optimal value $\beta = 0.694$ for his algorithm.

In their paper [8], Hanrot and Zimmermann give a slight variant of Mulders algorithm in which the original product is split into a full $k \times k$ product and two short $(N-k) \times (N-k)$ products, where the cutoff k now depends on N. Their method gives a significant improvement over Mulders' original fixed cutoff $k = \beta N$.

Mulders, In section 7 of his paper, gives a brief description of a recursive divide-and-conquer technique for performing what he calls *short division*, namely division of a polynomial of length at most $2n - 1$ by a polynomial of length n without computing a remainder. This algorithm is faster than a *long division*, in which one computes a quotient and remainder, and is based on the same principle as his short multiplication algorithm.

Mulders' short division algorithm reduces the problem recursively to one long division, one short multiplication and one short division. As with his short multiplication algorithm, Mulders uses a fixed cutoff, not depending on the length of the polynomials.

Mulders reported that the optimal cutoff for his algorithm was very nearly $\beta = 1/2$ and that there was little practical benefit in introducing a different cutoff.

In this paper we describe a variant of Mulders' algorithm which uses a variable cutoff in the manner of Hanrot and Zimmerman. In addition, instead of computing a remainder directly, we compute only the product of the quotient and divisor (from which the remainder is easily obtained by subtraction from the dividend). We show that this simple variant of Mulders' algorithm has very good performance in practice whilst remaining simple to implement. We call this algorithm Mulders' algorithm for simplicity.

We also give a slightly faster variant of this algorithm in which we replace the full division in Mulders recursion with a third recursive algorithm which returns only a short product of divisor and quotient. We optimistically call this *half full division*.

We provide details of timing experiments below, performed with our implementation of these algorithms. Our implementation is included in the FLINT (Fast Library for Number Theory) package.

We compare an implementation Mulders' algorithm, with parameter $\beta = \frac{1}{2}$, with our improved algorithm (with the same parameter). We then show that if the parameter is allowed to vary with the size of the polynomials, a further improvement is possible on some architectures. The optimal parameter can then often be quite far from $\beta = 1/2$.

We compare our short division implementation with the implementations of polynomial division in the packages NTL [14] and Magma [6].

2.1 Description of the short division algorithm

We assume throughout the following that we have available an algorithm MUL(f, g) for computing a full product of polynomials. We also require the classical algorithms for long and short division, DIV(f, g) and DIV_SHORT(f, g) respectively, returning a quotient q and remainder r (or in the case of short division, just the quotient) such that $f = gq + r$.

We also require that we have available algorithms for computing the following short products:

Algorithm 2.11 MUL_SHORT(f, g, n)

Input: *Polynomials* $f = \sum_{i=0}^{n_1} a_i x^i$ *and* $g = \sum_{j=0}^{n_2} b_j x^j$ *and a non-negative integer* n.

Output: *The low* n *terms of the product of* f *and* g, *i.e.* $\sum_{k=0}^{n-1} c_k x^k$ *where* $c_k = \sum_{i+j=k} a_i b_j$.

Algorithm 2.12 MUL_SHORT_OPP(f, g, n)

Input: *As for* MUL_SHORT.

Output: *All terms of the product of* f *and* g except *the first* n, *i.e.* $\sum_{k=n}^{n_1+n_2-1} c_k x^k$ *where* $c_k = \sum_{i+j=k} a_i b_j$, *if* $n_1 + n_2 - 1 \geq n$ *and* 0 *otherwise.*

Firstly we describe the basic divide-and-conquer type algorithm for doing a long division. However we do not return the remainder r, but the product of the divisor and quotient, qg, from which the remainder can be computed as $r = f - gq$.

In all of the algorithms below, we always reduce to the case where deg(f) is $m - 1$ and deg(g) is $n - 1$ with $m = 2n - 1$, so that the quotient also has degree $n - 1$. In the case where $m > 2n - 1$ we can truncate f to length $2n - 1$, do a division with remainder reducing the problem to one with a shorter dividend. When $m < 2n - 1$ the quotient will have length $l = m - n + 1$ and only depends on the leading $2l - 1$ terms of f and the leading l terms of g. We can compute this using a short division and multiply out and subtract to obtain the remainder $r = f - gq$.

Algorithm 2.13 DIVIDE_CONQUER_DIV(f, g)

Input: *Polynomials* $f = \sum_{i=0}^{m-1} a_i x^i$ *and* $g = \sum_{j=0}^{n-1} b_j x^j$

Output: *The quotient* q *and the full product* gq.

```
1. If $n <$ CUTOFF return DIV$(f, g)$
2. If $m \neq 2n - 1$ reduce to the case $m = 2n - 1$
3. $n_1 := \lceil n/2 \rceil$, $n_2 := n - n_1$
4. Let $b_1, b_2$ be such that $g = b_1 x^{n_2} + b_2$ with deg $b_2 < n_2$
5. Let $b_3, b_4$ be such that $g = b_3 x^{n_1} + b_4$ with deg $b_4 < n_1$
6. Let $a_1, a_2, a_3$ be such that $f = a_1 x^{n_2+n-1} + a_2 x^{n-1} + a_3$ with deg $a_2 < n_2$, deg $a_3 < n - 1$
7. $(q_1, b_1 q_1) :=$DIVIDE_CONQUER_DIV$(a_1 x^{n_1-1}, b_1)$
8. $bq_1 := b_1 q_1 x^{n_2} +$ MUL$(b_2, q_1)$ (††)
9. $t := a_1 x^{2n_2-1} + a_2 x^{n_2-1} - bq_1$ div $x^{n_1-n_2}$
10. $t' := t$ mod $x^{2n_2-1}$
11. $(q_2, b_3 q_2) :=$ DIVIDE_CONQUER_DIV$(t', b_3)$ (*)
12. $bq_2 := b_3 q_2 x^{n_1} +$ MUL$(b_4, q_2)$ (**)
13. Return $q := q_1 x^{n_2} + q_2$, $qg := bq_3 x^{n_1} + bq_4$ (†)
```

It is easy to turn this algorithm into an algorithm for short division, returning the quotient q only.

Algorithm 2.14 Divide_conquer_div_short(f, g)

Input: *As for* Divide_conquer_div.

Output: *The quotient q of f and g.*

In Divide_conquer_div(f, g) replace the call to Divide_conquer_div at (*) by a call to the function Divide_conquer_div_short, remove the line (**), replace Div with Div_short, replace Mul(b_2, q_1) at (††) with Mul_short_opp$(b_2, q_1, n_1 - 1)$ and return only the quotient q at (†). □

This is our first variant of Mulders' short division algorithm (with $\beta = \frac{1}{2}$). We now describe an improved version of this algorithm.

From now on, we assume that we have available an algorithm for classical division which only returns the lowest $n - 1$ terms of the product of the divisor and quotient. We call it Div_classical_half_full(f, g) since it returns about half of the product that our long division returns. The ordinary classical algorithm for long division can be modified in an obvious way to return this half product.

With this algorithm available we are now able to introduce an algorithm which we call *half full division.* As for the half full classical algorithm, this recursive algorithm performs the same operation as a full division, but only returns the lowest $n - 1$ terms of the product of the divisor and quotient.

At the bottom of the recursion, this algorithm does classical half full division which has fewer operations than a full long division.

We will use this algorithm instead of a full division in our improved version of Mulders' division algorithm, thus achieving a faster short division.

Algorithm 2.15 Half_full_div(f, g)

Input: *As for* Divide_conquer_div_short.

Output: *The quotient q and the low $n - 1$ terms of the product gq.*

```
If $m \neq 2n - 1$ reduce to the case $m = 2n - 1$
If $n <$ Cutoff return Div_classical_half_full$(f, g)$
Let $n_1 = \lceil n/2 \rceil$, $n_2 = n - n_1$
Let $b_1, b_2$ be such that $g = b_1 x^{n_2} + b_2$ with deg $b_2 < n_2$
Let $b_3, b_4$ be such that $g = b_3 x^{n_1} + b_4$ with deg $b_4 < n_1$
Let $a_1, a_2, a_3$ be such that $f = a_1 x^{n_2+n-1} + a_2 x^{n-1} + a_3$ with deg $a_2 < n_2$, deg $a_3 < n - 1$
$(q_1, b_1 q_1) :=$ Half_full_div$(a_1 x^{n_1 - 1}, b_1)$
$bq_1 := b_1 q_1 x^{n_2} +$ Mul$(b_2, q_1)$
$t := a_1 x^{2n_2 - 1} + a_2 x^{n_2 - 1} - bq_1$ div $x^{n_1 - n_2}$
$t' := t \bmod x^{2n_2 - 1}$
$(q_2, b_3 q_2) :=$ Half_full_div$(t', b_3)$
$bq_2 := b_3 q_2 x^{n_1} +$ Mul$(b_4, q_2)$
Return $q := q_1 x^{n_2} + q_2$, $qg := bq_3 x^{n_1} + bq_4$
```

Finally we are able to describe our improved version of Mulders' short division algorithm. As in Mulders' paper we allow the algorithm to split the inputs into unequal parts, however as per Hanrot and Zimmerman, we will allow the cutoff to vary with the length of the input polynomial g. For this purpose we define a parameter $k = k(n)$ whose optimal value will be determined experimentally.

Algorithm 2.16 Divide_conquer_div_short_improved(f, g)

Input: *As for* Divide_conquer_div_short.

Output: *The quotient q of f and g.*

```
1. If $n <$ Cutoff return Div_short$(f, g)$
2. If $m \neq 2n - 1$ reduce to the case $m = 2n - 1$
3. Let $n_1 = \lceil n/2 \rceil +$ k(n), $n_2 = n - n_1$
4. Let $b_1, b_2$ be such that $g = b_1 x^{n_2} + b_2$ with deg $b_2 < n_2$
5. Let $b_3, b_4$ be such that $g = b_3 x^{n_1} + b_4$ with deg $b_4 < n_1$
6. Let $a_1, a_2, a_3$ be such that $f = a_1 x^{n_2+n-1} + a_2 x^{n-1} + a_3$ with deg $a_2 < n_2$, deg $a_3 < n - 1$
7. $(q_1, b_1 q_1) :=$ Half_full_div$(a_1 x^{n_1 - 1}, b_1)$
8. $bq_1 := b_1 q_1 x^{n_2} +$ Mul_short_opp$(b_2, q_1, n_1 - 1)$
9. $t := a_1 x^{2n_2 - 1} + a_2 x^{n_2 - 1} - bq_1$ div $x^{n_1 - n_2}$
10. $t' := t$ mod $x^{2n_2 - 1}$
11. $q_2 :=$ Divide_conquer_div_short_improved$(t', b_3)$
12. Return $q := q_1 x^{n_2} + q_2$
```

2.2 Mulders' vs divide-conquer-div-short-improved

We began our timing experiments by comparing times for our first variant of Mulders' division algorithm and the improved short division algorithm with $k(n) = 0$.

Unless otherwise noted, all our division timings and comparisons in this paper were performed on a 2.4GHz AMD Opteron Server. Each computation was (automatically) repeated many times and the lowest timing was recorded in each case.

Given a length n (degree $n - 1$) and a number of bits b we let f be a polynomial of length $2n - 1$ with uniformly random coefficients of $2b$ bits and g be a polynomial of length n with uniformly random coefficients of b bits. We computed the short division of f and g using both algorithms.

Our timings showed that the improved algorithm was marginally faster (up to 23% but on average much less). The biggest improvements are where n is large and b is small. In the interests of space we omit all but the timings in this range. Timings for the improved algorithm are shown on the right and those for the original on the left.

$n\backslash b$	8	16	32	64
64	90.6μs	92.6μs	106μs	123μs
128	266μs	276μs	300μs	346μs
256	758μs	763μs	854μs	964μs
512	2.43ms	2.32ms	2.82ms	3.01ms

$n\backslash b$	8	16	32	64
64	87.4μs	90.0μs	100μs	117μs
128	243μs	256μs	278μs	328μs
256	714μs	752μs	788μs	975μs
512	1.98ms	1.96ms	2.50ms	2.81ms

Table 3. Comparison of simple and improved Mulders' variants

A second timing experiment we performed was to adjust the value of $k(n)$ in the improved short division algorithm. Whether or not this had an effect proved to be highly architecture sensitive. On a 1.8GHz AMD K8 machine we found that if n is the length of g then for $20 < n \leq 100$ the value of $k(n)$ should be about $n/5$ and for $n \leq 20$ the value of $k(n)$ should be $n/4$.

However, on a 2.4GHz AMD K10 machine, a value of $k(n) = 0$ was roughly optimal for all sizes.

2.3 Comparison with other implementations

The most important comparison for the purposes of this paper is the comparison between our variant of Mulders' algorithm and the polynomial division available in other packages. For this purpose we compare with the best open source and the best proprietary packages we are aware of. In the former case we compare with NTL v5.5.2 and in the latter case with Magma v2.16-7.

Whilst we do not know the algorithm used by Magma, we know that NTL uses a multimodular approach, performing the division using multiple primes then recombining with the Chinese Remainder Algorithm. It leverages highly optimised functions for division over $\mathbb{Z}/p\mathbb{Z}$.

Our first comparison is made using random polynomials of lengths $2n-1$ and n respectively, as described in the previous section. This tests the most general case for our algorithm. NTL does not offer inexact division over $\mathbb{Z}$, thus the first comparison is with Magma only (the top row in each case).

$n\backslash b$	8	16	32	64	128	256	512	1024	2048	4096	8192
32	32.6μs	70.8μs	99.4μs	101μs	124μs	155μs	259μs	574μs	1.62ms	4.58ms	14.0ms
	29.0μs	30.0μs	33.4μs	39.4μs	49.1μs	70.8μs	113μs	237μs	597μs	1.60ms	4.63ms
64	162μs	260μs	353μs	394μs	460μs	573μs	960μs	2.20ms	6.59ms	20.0ms	60.0ms
	87.4μs	90.0μs	100μs	117μs	150μs	239μs	375μs	741μs	1.80ms	4.80ms	13.7ms
128	767μs	1.03ms	1.36ms	1.45ms	1.78ms	2.28ms	3.90ms	9.33ms	25.5ms	77.5ms	
	243μs	256μs	278μs	328μs	433μs	711μs	1.12ms	2.21ms	5.27ms	13.4ms	
256	3.67ms	4.33ms	5.58ms	6.04ms	7.50ms	10.3ms	16.7ms	37.5ms	107ms		
	714μs	752μs	788μs	975μs	1.31ms	1.98ms	3.07ms	6.18ms	13.7ms		
512	16.0ms	17.5ms	23.3ms	25.5ms	30.0ms	40.0ms	64.0ms	153ms			
	1.98ms	1.96ms	2.50ms	2.81ms	3.67ms	5.60ms	8.80ms	16.6ms			

Table 4. Short division in Magma and FLINT

Finally we compare NTL, Magma and our improved divide-and-conquer algorithm (timing rows in that order) on exact divisions. Here we construct polynomials f, g with the given lengths n and uniformly random coefficients with the given number of bits b and perform the division h/g where $h = f * g$.

$n\backslash b$	8	16	32	64	128	256	512	1024	2048	4096	8192
32	43.0μs	43.8μs	49.8μs	76.6μs	145μs	294μs	684μs	1.93ms	5.87ms	20.1ms	73.3ms
	23.2μs	49.1μs	82.5μs	92.8μs	106μs	139μs	237μs	533μs	1.57ms	4.59ms	14.0ms
	29.1μs	29.0μs	32.0μs	34.8μs	46.3μs	68.0μs	106μs	227μs	558μs	1.57ms	4.57ms
64	104μs	105μs	116μs	176μs	323μs	646μs	1.48ms	4.00ms	12.1ms	40.9ms	148ms
	81.1μs	180μs	294μs	340μs	370μs	510μs	905μs	2.10ms	6.12ms	18.3ms	57.5ms
	83.3μs	84.4μs	88.1μs	98.5μs	136μs	231μs	360μs	721μs	1.73ms	4.56ms	13.3ms
128	284μs	285μs	306μs	461μs	785μs	1.53ms	3.37ms	8.76ms	28.1ms	84.7ms	
	310μs	668μs	1.17ms	1.30ms	1.46ms	2.00ms	3.54ms	8.24ms	25.0ms	70.0ms	
	211μs	212μs	228μs	262μs	376μs	688μs	1.06ms	2.12ms	4.98ms	12.5ms	
256	838μs	839μs	880μs	1.33ms	2.11ms	4.00ms	8.35ms	20.6ms	57.8ms		
	1.19ms	2.66ms	4.50ms	5.17ms	5.92ms	8.00ms	14.7ms	32.9ms	97.5ms		
	502μs	512μs	553μs	661μs	982μs	1.87ms	2.93ms	5.71ms	13.5ms		
512	1.71ms	1.71ms	1.80ms	2.68ms	4.26ms	8.33ms	17.0ms	43.4ms			
	4.67ms	10.7ms	18.3ms	20.0ms	23.9ms	32.2ms	57.5ms	133ms			
	1.18ms	1.19ms	1.31ms	1.63ms	2.50ms	5.00ms	7.77ms	15.9ms			

Table 5. Exact division in NTL, Magma and FLINT

3 Conclusions

We have provided efficient divide-and-conquer style algorithms for the composition and division of univariate polynomials over $\mathbb{Z}$.

In the former case, we show that the algorithm is asymptotically fast with respect to bit complexity, effectively handling coefficient explosion.

In the latter case we have provided two easy to implement variants of Mulders' algorithm and shown that, at least on modern computers, the divide-and-conquer approach deserves a closer look, often outperforming other commonly used methods.

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, $\mathcal{O}((n \log n)^3/2)$ *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 2.17 (2010) `http://magma.maths.usyd.edu.au/magma`
7. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.
8. G. Hanrot and P. Zimmermann. *A long note on Mulder's short product*, Journal of Symbolic Computation, v. 37 3 2004, pp. 391–401.
9. W. Hart *Fast Library for Number Theory: an introduction*, Mathematical Software - ICMS 2010 Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010, Proceedings Series: Lecture Notes in Computer Science, Vol. 6327, pp 88–91. `http://www.flintlib.org`
10. D. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997, pp. 486–488.
11. W. Liu and S. Mann *An analysis of polynomial composition algorithms*, University of Waterloo Research Report CS-95-24, (1995).
12. T. Mulders, *On Short Multiplications and Divisions*, AAECC, vol. 11, 2000, pp. 69–88.
13. V. Pan *Structured matrices and polynomials: unified superfast algorithms*, Springer-Verlag, 2001, pg. 81.
14. V. Shoup *NTL: A Library for doing Number Theory*, open-source library. http://shoup.net/ntl/