Lazy multiplication of formal power series

Joris van der Hoeven

Abstract

For most fast algorithms to manipulate formal power series, a fast multiplication algorithm is essential. If one desires to compute all coefficients of a product of two power series up to a given order, then several efficient algorithms are available, such as fast Fourier multiplication. However, one often needs a lazy multiplication algorithm, for instance when the product computation is part of the computation of the coefficients of an implicitly defined power series. In this paper, we describe two lazy multiplication algorithms, which are faster than the naive method. In particular, we give an algorithm of time complexity $O(n \log^2 n)$.

Key words: power series, multiplication, algorithm.

1 Introduction

In this paper, we are concerned with fast effective computations with power series in z over an effective field of constants $\mathfrak{C}$. The time (and space) complexities we consider will be measured in terms of operations on the constants (and the number of constants we need to store).

Most known algorithms for fast manipulations of formal power series are based on a fast multiplication algorithm. Since power series are infinite objects, we will be interested in computing the first (but an arbitrary large number of) coefficients of them. Now we distinguish two types of multiplication algorithms.

First, we have *static* multiplication algorithms: let f and g be the power series we want to multiply. Then a static multiplication algorithm takes a natural number n and the first n coefficients $f_0, \cdots, f_{n-1}, g_0, \cdots, g_{n-1}$ of f and g on input, and returns the first n coefficients $(fg)_0, \cdots, (fg)_{n-1}$ of the product fg. Classical static multiplication algorithms are naive multiplication, Karatsuba's algorithm, and FFT multiplication, of respective complexities $O(n^2), O(n^{\log 3/\log 2})$ and $O(n \log n)$. We refer to [3] for more details.

On the other hand, one often needs *lazy* multiplication algorithms. In this case, the successive coefficients of f and g are only given one by one, and the lazy multiplication algorithm should output the coefficient of z^i in fg, as soon as $f_0, \cdots, f_i, g_0, \cdots, g_i$ are known.

The advantage of lazy multiplication is that the coefficients of z^i in f and g may be defined in terms of the first i coefficients of fg. For this reason, lazy multiplication can be used as a subalgorithm, when we want to compute the coefficients of an implicitly defined power series. An easy example is the exponentiation of a power series h with $h_0 = 0$, which can be computed lazily using the formula

$$e^h = \int h' e^h .$$

Although the complexity of static multiplication has been studied extensively, the best known bound for lazy multiplication is the naive bound $O(n^2)$. In section 2, we show that Karatsuba's algorithm can actually be made lazy, which leads to the following:

Theorem 1. *There exists a lazy multiplication algorithm for formal power series, of time complexity $O(n^{\log 3/\log 2})$ and space complexity $O(n \log n)$.*

Let $M(n) = n^\alpha \log^\beta n \log^\gamma \log n$ be a complexity bound for the multiplication of two polynomials of degree $n-1$ and coefficients in $\mathfrak{C}$. In section 3, we prove the main result of this paper:

Theorem 2. *There exists a lazy multiplication algorithm for formal power series, of time complexity $O(M(n) \log n)$ and space complexity $O(n)$.*

In particular, if $\mathfrak{C}$ supports FFT multiplication (i.e. the fast Fourier transform), then we may take $M(n) =$

$n \log n$ and we obtain a lazy multiplication algorithm of complexity $O(n \log^2 n)$. In general, one may take $M(N) = n \log n \log\log n$ (see [5]).

In section 4 we apply theorem 2 to the resolution of implicit equations, and, in this context, we compare the performances of our and more classical algorithms.

2 Karatsuba's multiplication algorithm is lazy

Karatsuba's algorithm for polynomials. Let us first recall Karatsuba's method for multiplying two polynomials P and Q in z of degrees $n-1 \geqslant 1$. We write

$$\begin{aligned} P &= P_{lo} + P_{hi} z^{\lfloor n/2 \rfloor} \\ Q &= Q_{lo} + Q_{hi} z^{\lfloor n/2 \rfloor}, \end{aligned}$$

where the *lower* parts P_{lo} and Q_{lo} of P resp. Q have degrees $\leqslant \lfloor n/2 \rfloor - 1$. Then we have

$$\begin{aligned} PQ &= P_{lo}Q_{lo} \\ &+ ((P_{lo}+P_{hi})(Q_{lo}+Q_{hi}) - P_{lo}Q_{lo} - P_{hi}Q_{hi}) z^{\lfloor n/2 \rfloor} \\ &+ P_{hi}Q_{hi} z^{2\lfloor n/2 \rfloor}. \end{aligned}$$

Karatsuba's algorithm consists of recursively applying the above formula in order to compute the product PQ. Since the multiplication of two polynomials of degree $n-1$ involves only three multiplications of polynomials of degrees $\leqslant \lfloor n/2 \rfloor$, the asymptotic time complexity of the method is $O(n^{\log 3/\log 2})$.

Important observation. If we apply the above algorithm symbolically to compute a formula for $(PQ)_i$, then we notice that this only depends on the coefficients $P_0, \cdots, P_i$ and $Q_0, \cdots, Q_i$. Actually, this observation implies at once the existence of a lazy version of Karatsuba's algorithm, when applied to power series. We will now show how to implement such an algorithm.

Truncated lazy multiplication. We first consider the case when we want to multiply two power series f and g lazily up to the order $O(z^n)$ only, where $n = 2^p$ is a power of two (with $n \geqslant 2$). We define

$$\begin{aligned} P &= f_0 + f_1 z + \cdots + f_{n-1} z^{n-1}; \\ Q &= g_0 + g_1 z + \cdots + g_{n-1} z^{n-1}, \end{aligned}$$

and we want to use the above formulas to compute PQ. Hence, we are interested in multiplying P and Q, where the coefficients of P and Q are given one by one, and we require the first i coefficients of PQ to be output as soon as the first i coefficients of P and Q are known.

The truncated lazy multiplication of P and Q corresponds to the instance of a new type of hybrid lazy/static data structure $\mathfrak{H}$: as long as P and Q are only partially known, the status of the data structure is "lazy", and the data structure contains extra information (see below) to continue the lazy multiplication. If P and Q are completely known, then this extra information is handled back to the memory manager and the status of the data structure becomes "complete"; only the result of the multiplication PQ is kept into memory.

Now our algorithm consists simply of using the hybrid data structure $\mathfrak{H}$ recursively for the computation of the products $P_{lo}Q_{lo}$, $(P_{lo}+P_{hi})(Q_{lo}+Q_{hi})$ and $P_{hi}Q_{hi}$. More precisely, if the status of the computation is "lazy", then the extra information mentioned above consists of three pointers to objects of type $\mathfrak{H}$; these three objects correspond precisely to the lazy computations of the products $P_{lo}Q_{lo}$, $(P_{lo}+P_{hi})(Q_{lo}+Q_{hi})$ and $P_{hi}Q_{hi}$. Of course, the case when $n = 2$ is treated apart, since in this case P_{lo}, P_{hi}, Q_{lo} and Q_{hi} are just constants, and the multiplication algorithm in $\mathfrak{C}$ is used.

Complexity analysis. It is clear that the time complexity of our algorithm is the same (up to a constant factor) as Karatsuba's static algorithm. As to the memory storage $S(n)$ needed by the algorithm, we observe that

$$S(n) \leqslant 2S(n/2) + O(n). \tag{1}$$

Indeed, as long as less than $n/2$ coefficients of P and Q are known, P_{hi} and Q_{hi} are not needed at all. As soon as $n/2$ coefficients are known, P_{lo} and Q_{lo} are entirely determined, whence the computation of $P_{lo}Q_{lo}$ is completed, and the result takes $O(n)$ memory storage. Furthermore, $P_{lo} + P_{hi}$ and $Q_{lo} + Q_{hi}$ require another $O(n)$ memory storage, while the computations of $(P_{lo} + P_{hi})(Q_{lo} + Q_{hi})$ and $P_{hi}Q_{hi}$ require $2S(n/2)$ memory storage, by induction. From (1), we deduce that

$$S(n) = O(n \log n).$$

The general case. Let us finally treat the original case, when we want to compute fg up to any order, and not merely up to order $O(z^n)$. In this case, we use the algorithm from above between successive powers of two. Each time we have computed the first $n = 2^p$ coefficients of fg, we multiply n by two, and let the old P and Q play the rôles of P_{lo} and Q_{lo} for the new P and Q. Clearly, the time and space complexities of this algorithm are $O(n^{\log 3/\log 2})$ and $O(n \log n)$ as above. This proves theorem 1.

3 A fast lazy multiplication algorithm

Premature computations. The algorithm of the previous section has the important property that we compute *more* than we actually need at each stage. For instance, $f_1 g_1$ is computed as soon as f_1 and g_1 are

$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
g_7	7	8	9	10	11	12	13	14	$\cdots$
g_6	6	7	8	9	10	11	12	13	$\cdots$
g_5	5	6	7	8	9	10	11	12	$\cdots$
g_4	4	5	6	7	8	9	10	11	$\cdots$
g_3	3	4	5	6	7	8	9	10	$\cdots$
g_2	2	3	4	5	6	7	8	9	$\cdots$
g_1	1	2	3	4	5	6	7	8	$\cdots$
g_0	0	1	2	3	4	5	6	7	$\cdots$
$\times$	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	$\cdots$

Figure 1: Lazy multiplication by the naive algorithm

$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
g_7	7	7	7	7	7	7	7	7	$\cdots$
g_6	6	7	6	7	6	7	6	7	$\cdots$
g_5	5	5	7	7	5	5	7	7	$\cdots$
g_4	4	5	6	7	4	5	6	7	$\cdots$
g_3	3	3	3	3	7	7	7	7	$\cdots$
g_2	2	3	2	3	6	7	6	7	$\cdots$
g_1	1	1	3	3	5	5	7	7	$\cdots$
g_0	0	1	2	3	4	5	6	7	$\cdots$
$\times$	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	$\cdots$

Figure 2: Lazy multiplication by Karatsuba's algorithm

known. Although $f_1 g_1$ does not contribute to the coefficient $(fg)_1 = f_0 g_1 + f_1 g_0$, its premature computation accelerates the computation of $(fg)_2$ at the next stage.

The lazy computation process of fg by Karatsuba's algorithm is represented schematically in figure 2. Each box corresponds to the contribution of $f_i g_j$ to the sum $(fg)_{i+j} = \sum_{k=0}^{i+j} f_k g_{i+j-k}$. The number $s_{i,j}$ in the box corresponds to the round when the contribution of $f_i g_j$ is computed, i.e. when $(fg)_{s_{i,j}}$ is output. For comparison, we have represented the computation process of fg by the naive algorithm (i.e. $(fg)_n$ is computed by $(fg)_n = \sum_{k=0}^{n} f_k g_{n-k}$) in figure 1.

Let us now show how the idea of premature computations can be exploited even more. Observe first, that if the first 2^{p+1} coefficients of f and g are known, then the multiplication

$$\begin{aligned} \Pi_{2^p,2^p} &= (f_{2^p} z^{2^p} + \cdots + f_{2^{p+1}-1} z^{2^{p+1}-1}) \\ &\quad (g_{2^p} z^{2^p} + \cdots + g_{2^{p+1}-1} z^{2^{p+1}-1}) \end{aligned}$$

can be performed prematurely by *any* fast static multiplication algorithm. More generally, if the first $n = (k+1)2^p$ coefficients of f and g are known, with $k \in \{2, 3, \cdots\}$ and $p \geqslant 1$, then the multiplications

$$\begin{aligned} \Pi_{2^p,k2^p} &= (f_{2^p} z^{2^p} + \cdots + f_{2^{p+1}-1} z^{2^{p+1}-1}) \\ &\quad (g_{k2^p} z^{k2^p} + \cdots + g_{(k+1)2^p-1} z^{(k+1)2^p-1}) \end{aligned}$$

and

$$\begin{aligned} \Pi_{k2^p,2^p} &= (f_{k2^p} z^{k2^p} + \cdots + f_{(k+1)2^p-1} z^{(k+1)2^p-1}) \\ &\quad (g_{2^p} z^{2^p} + \cdots + g_{2^{p+1}-1} z^{2^{p+1}-1}) \end{aligned}$$

can be performed prematurely.

Fast lazy multiplication. Let us now detail how the above observations can be transformed into a lazy multiplication algorithm. The coefficients $(fg)_0, (fg)_1, \cdots$ of the result are stored in an array A, which is resized automatically, each time when necessary; the default value of new array elements is zero. Now we undertake the following action to compute $A[n]$ (where we assume that $f_0, \cdots, f_n, g_0, \cdots, g_n$ are known and $A[0], \cdots, A[n-1]$ have already been computed):

Step 1. [Border] If $n = 0$, then set $A[0] := f_0 g_0$. Otherwise, set $A[n] := A[n] + f_0 g_n + f_n g_0$.

Step 2. [Diagonal] If $n = 2^{p+1}$ for some $p \geqslant 0$, then compute $\Pi_{2^p,2^p}$ and set $A[i] := A[i] + \Pi_{2^p,2^p,i}$ for all $2^{p+1} \leqslant i \leqslant 2^{p+2} - 2$.

Step 3. [Main] For each $k \geqslant 2$ and $p \geqslant 0$ such that $n = (k+1)2^p$, do the following:

- Compute $\Pi_{2^p,k2^p}$ and set $A[i] := A[i] + \Pi_{2^p,k2^p,i}$ for all $(k+1)2^p \leqslant i \leqslant (k+3)2^p - 2$.
- Compute $\Pi_{k2^p,2^p}$ and set $A[i] := A[i] + \Pi_{k2^p,2^p,i}$ for all $(k+1)2^p \leqslant i \leqslant (k+3)2^p - 2$.

Correctness proof. The computation process is schematically represented in figure 3. From this figure, it is easily seen that the contribution of each $f_i g_j$ to $(fg)_{i+j}$ is computed exactly once and before the coefficient $(fg)_{i+j}$ is output. This proves the correctness of our algorithm.

Complexity analysis. Let us now estimate the asymptotic complexity $T(n)$ of the computation of $(fg)_n$ by our algorithm. If $n = 2^p$, then in order to compute the contribution of

$$(f_0 + \cdots + f_{n-1} z^{n-1})(g_0 + \cdots + g_{n-1} z^{n-1}), \qquad (2)$$

to fg, we need $(2n-1) + (2n-3)$ constant multiplications, $n-3$ multiplications of polynomials of degree one, $n/2 - 3$ multiplications of polynomials of degree three, etc. (by looking at figure 3).

Let $M(d) = d^\alpha \log^\beta d \log^\gamma \log d$ be a complexity bound for the multiplication of two polynomials of degree $d-1$, as in the introduction. In view of what precedes, the time $\tilde{T}(n)$ needed to compute the contribution

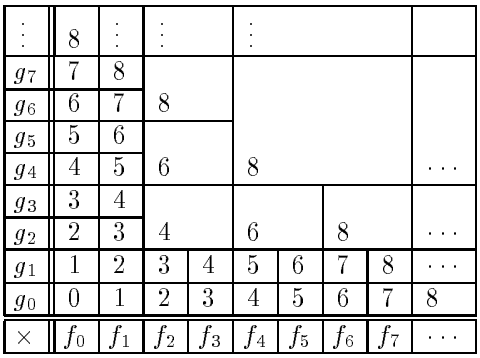

Figure 3: Fast lazy multiplication

of (2) to fg is bounded by

$$2\sum_{k=0}^{p-1}\frac{n}{2^k}M(2^k)+O(n)=O(M(n)\log n). \tag{3}$$

For general $n \geqslant 2$, this yields $T(n) = O(M(n)\log n)$, since

$$\tilde{T}(2^{\lfloor \log_2 n\rfloor}) \leqslant T(n) \leqslant \tilde{T}(2^{\lfloor \log_2 n\rfloor+1}).$$

The space needed for the computation of $(fg)_n$ is clearly bounded by $O(n)$. This completes the proof of theorem 2.

Remark. Sometimes, we need a lazy multiplication algorithm for power series, which computes only the first n coefficients of the product (i.e. for the expansion of implicit functions). In this case, it is not hard to modify the above algorithm, so that no "unnecessary premature computations" are done, which anticipate the computations of $(fg)_n, (fg)_{n+1}, \cdots$.

4 Conclusion

Let us discuss some of the consequences of theorem 2. An important application of lazy multiplication is the resolution of algebraic differential equations

$$\sum_{i_0,\cdots,i_r} P_{i_0,\cdots,i_r}\, f^{i_0}\cdots(f^{(r)})^{i_r}=0 \tag{4}$$

with power series coefficients and suitable initial conditions. Although Brent and Kung have given an asymptotically better algorithm to solve such equations in [1], the constant involved in their asymptotic estimate $O(n\log n)$ of the time complexity depends badly on r, whereas the constant involved in our algorithm only depends on the size of (4) as an expression. Therefore, we expect our algorithm to be often faster for not all to large n.

Similarly, both our algorithm and Brent and Kung's algorithm require $O(n)$ memory space, but the constant factor involved in this bound is usually much smaller for our algorithm. Here we notice that for many applications the available memory space is the main bottleneck. Another advantage of our algorithm is that it is easy to implement and that it can easily be applied to solve systems of ordinary differential equations as well.

Our lazy multiplication algorithm can also be used to solve more general functional equations, such as

$$s(z)=1+z\frac{s(z)^3+2s(z^3)}{3}, \tag{5}$$

which occurs in the study of the number of stereoisomeres of alcohols of the form $C_nH_{2n+1}OH$ (see [4]). Then theorem 2 implies that the asymptotic complexity to compute the first n coefficients of $s(z)$ is $O(n\log^2 n)$, which is much better than the previously best known bound $O(n^2)$. Many other differential difference equations arising in combinatorics and the analysis of algorithms are similar to (5) (see also [2]). This is in particular so for binary splitting algorithms, such as the algorithms presented in this paper themselves!

Let us finally remark that for the fast expansion of power series which satisfy more complicated differential difference equations, it would be nice to extend the ideas of this paper in order to obtain lazy versions of Brent and Kung's fast algorithms for functional composition and inversion (see [1]).

Acknowledgment. The author thanks the referees for their detailed comments, and for having pointed out some mistakes in the original version of this paper.

References

[1] Brent, R., and Kung, H. Fast algorithms for manipulating formal power series. *Journal of the Association for Computing Machinery 25*, 4 (1978), 581–595.

[2] Flajolet, P., and Sedgewick, R. *An introduction to the analysis of algorithms.* Addison Wesley, Reading, Massachusetts, 1996.

[3] Knuth, D. *The art of computer programming*, vol. 2: seminumerical algorithms. Addison Wesley, Reading, Massachusetts, 1981.

[4] Pólya, G. Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica 68* (1937), 145–254.

[5] Schönhage, A. Schnelle Multiplikation über Körpern der Charakteristik 2. *Acta Inform. 7* (1977), 395–398.