CS 860 Spring 2007 Research Project: Adaptive Polynomial Multiplication

Daniel S. Roche

August 20, 2007

1 Introduction

Polynomial multiplication is as close to any problem comes to being "classical" in the field of computer algebra. Elementary arithmetic on polynomials with coefficients in an arbitrary domain is one of the basic primitives supplied by any computer algebra or symbolic computation software.

Currently, there are essentially two different representations for polynomials: dense and sparse. These will be discussed in more detail later, but for now it suffices to say that in the dense representation, we write down every coefficient, while the sparse representation omits all zero coefficients.

von zur Gathen and Gerhard point out some similarities between dense polynomial multiplication and sorting, as for both problems the "school method" has quadratic complexity, while "fast methods" reduce this to superlinear complexity [13]. Based on the tremendous success of adaptive algorithms and analysis for the sorting problem, an obvious question is whether adaptive algorithms can also be used for polynomilal multiplication.

We present and analyze a few ideas for adaptive polynomial multiplication as a first step towards improving polynomial multiplication without actually reducing the worst-case complexity. The new representations also present a finer gradient between the sparse and dense polynomial representations currently known and in use.

After giving a brief overview of past results and some algebraic and notational preliminaries, we discuss possible approaches for an adaptive analysis and their relative strenghts and weaknesses. Next, the three main ideas for adaptive polynomial multiplication are presented and analyzed. Finally, we discuss future directions for research and some conclusions.

2 Preliminaries

2.1 Notation

Let R be an arbitrary ring, commutative with identity. If $f(x) \in \mathsf{R}[x]$ has degree less than n, then we can write

$$\begin{aligned} f(x) &= c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1} && (2.1) \\ &= a_1 x^{e_1} + a_2 x^{e_2} + \cdots a_s x^{e_s}, && (2.2) \end{aligned}$$

with $e_1 < e_2 < \cdots < e_s < n$ and each $a_i = c_{e_i} \neq 0$.

(2.1) corresponds to the dense representation of $f(x)$ as a vector of coefficients of length n, and (2.2) corresponds to the sparse representation of $f(x)$ as a list of pairs of nonzero coefficient and exponent. The integer s is exactly the number of nonzero terms in $f(x)$; we call this the *sparsity* of $f(x)$.

Whenever we write $\lg n$, for $n \in \mathbb{Z}^+$, we mean the number of bits required to represent n. So $\lg n = \lceil \log_2(n+1) \rceil$. Of course, $\lg n \in O(\log n)$, so asymptotically there is no distinction.

For the remainder, assume $f(x), g(x) \in \mathsf{R}[x]$ with degrees less than n and m and sparsities s and t, respectively. These will be the two polynomials to be multiplied.

2.2 Cost Model

In order to be as general as possible, the complexity of dense polynomial operations is usually measured in the number of ring operations $(+, -, \times)$ in R. This is more realistic for finite rings, such as the integers mod a word-size prime, where any single arithmetic operation can be performed in constant time. For larger rings where the size of the coefficients can be significant, more careful analysis may be necessary, but we will ignore this issue for now.

Complexity measures should also take into account the "overhead" cost — the number of word operations needed *other than* for performing basic arithmetic operations in R. Algorithms for densely-represented polynomials usually have running time dominated by the cost of the ring operations, so this is not an issue (we can safely assume that each ring operation has at least some constant cost). When dealing with sparse polynomials, however, the size of the coefficients can be very large, requiring multiple-precision integer arithmetic. So here the cost will often be given in terms of the number of ring operations *and* the number of word operations required.

2.3 Dense Polynomials

The dense representation is perhaps the more obvious one and seems to be the first one considered and examined by researchers. The "school method" for addition runs in linear time in the size of the input and is therefore optimal. However, the obvious method for multiplication requires a quadratic number of multiplications in the coefficient domain, and this is where significant effort and progress have been made.

For convenience, a common notation is a "multiplication time" function $\mathsf{M}(n)$, which is the complexity of multiplying two polynomials of degrees less than n. So the "school method" gives $\mathsf{M}(n) \in O(n^2)$.

In 1963, Karatsuba introduced a divide-and-conquer method which reduces the number of multiplications required by increasing the number of additions, resulting in a complexity of $O(n^{log_2 3})$, or $O(n^{1.59})$ [6].

An arbitrary-precision integer multiplication method introduced by Schönhage and Strassen in [9] in 1971 using a change of representation approach with the discrete Fourier transform (DFT) was later applied to polynomials [8, 2]. Using the fast Fourier transform algorithm (FFT) from [3] gives $\mathsf{M}(n) \in O(n \lg n \lg \lg n)$.

A brand-new result for integer multiplication is likely applicable to polynomials and may reduce the complexity to $O(n \lg n 2^{O(\lg^* n)})$ [4]. A lower bound of $\Omega(n \lg n)$ has been proven in the bounded-coefficient model over $\mathbb{C}$ [1], and it is widely believed that this lower bound holds in general. In fact, the DFT algorithm has complexity $O(n \lg n)$ in many circumstances under certain rings [13] — for instance when $\mathsf{R} = \mathbb{C}$ this is always true.

Assuming for the moment that $m \leq n$, all the results above can actually be improved slightly by splitting $f(x)$ into n/m groups and then multiplying each of these by $g(x)$ and summing the result. So the cost of dense multiplication of $f(x)$ and $g(x)$ is $\frac{n}{m}\mathsf{M}(m)$. Using the DFT algorithm we know this is at worst $O(n \lg m \lg \lg m)$. Occasionally for simplicity we will the cost as just $\Theta(n \lg m)$.

2.4 Sparse Polynomials

Addition of two sparse univariate polynomials is essentially equivalent to merging two sorted lists; a worst-case optimal algorithm gives complexity which is linear in the size of the input.

For multiplication, the "school method" requires st multiplications in R. And the number of nonzero terms in the product fg can be as large as st in the worst case, so this is in fact worst-case optimal.

Most improvements in sparse polynomial multiplication have come in the number of word operations (i.e. operations with the possibly-large exponents), as well as the space complexity. The school method uses $O(s^2 t \lg n)$ word operations and $O(st)$ space. In [14], the number of word operations is reduced to $O(st \lg s \lg n)$. Finally, using some old results from [5], the space complexity can be reduced to $O(t + r)$, where r is the sparsity of the product [7].

2.5 Adaptive Approach

The adaptive algorithms studied here are all based on a change of representation, like the FFT-based dense polynomial multiplication algorithms, and thus will proceed in three stages, as follows:

Step 1. Convert input polynomials to adaptive representation.

Step 2. Perform multiplication in the adaptive representation.

Step 3. Convert back to input representation.

Step two (the actual multiplication) usually just involves slight modifications and usages of existing algorithms. And the third step is usually trivial and may actually be combined with the second for efficiency. The most difficult step is the first one, which involves recognizing the difficulty of the input and performing the change of representation as fast as possible. Unlike the FFT-based multiplication algorithms, we do not want this step to be the dominating one in terms of computational cost. In fact, since in the best (i.e. least difficult) case it should be possible to multiply two polynomials in linear time, the first step of the algorithm should run in as close to linear time as possible (and may in itself be adaptive).

Our first instinct for how to construct the change of representation algorithm (step 1) may be to define some canonical form for the adaptive representation which will minimize the cost of the actual multiplication in step 2. Then converting the input polynomials to the adaptive representation can be performed independently (even in parallel). This is highly advantageous for a few reasons, chief among them the fact that in many (or even most) cases we may not be performing just a single multiplication operation but a series of arithmetic operations including many multiplications and additions. So, if there is some well-defined adaptive representation which depends only on the single polynomial itself, then we can avoid the cost of converting to and from the standard dense or sparse representations whenever intermediate results are not needed.

Unfortunately, a canonical form which always minimizes the cost of multiplication does not exist in general! This is perhaps most easily seen with an example. Recall that $f(x), g(x) \in \mathsf{R}[x]$ with degrees less than n, m and sparsities s, t (respectively). And suppose we have the most primitive adaptive algorithm which simply chooses either the standard sparse or dense representation.

So, if a canonical form exists, then given n and s, we should be able to choose the best representation for $f(x)$. For this example, say $n = s^2$. To show this is impossible, consider first $m = s^3$ and $t = s$. Then dense multiplication uses $\Omega(s^3)$ ring operations, while sparse multiplication only uses $O(s^2)$, so the sparse representation is better. But if instead $m = t = s$, then dense multiplication only uses $O(s \lg s \lg \lg s)$ ring operations, while sparse uses $\Omega(s^2)$, so in this case the dense representation is better.

This shows that the choice of representation will not be optimal (in general) unless both arguments are considered. So one way to choose the parameters for the adaptive representation of input polynomials is to examine both arguments simultaneously and choose the representations that minimize the cost of that particular multiplication operation. We call this approach the *dependent method*.

But, as we have mentioned, there are inherent problems with considering both operands, most notably when multiple operations are to be performed in sequence. Furthermore, especially when there are many parameters to the choice of adaptive representation, the time spent to find the optimal adaptive representation might be more than the time saved, of course negating any benifit.

For the *independent method*, we choose the adaptive representation separately for each input polynomial, therefore defining some sort of "canonical form". The analysis for choice of adaptive representation here is usually conservative, in the sense that we may sacrifice some best-case performance in order to guarantee, say, that the algorithm always performs at least as good as the standard dense/sparse algorithms. Such sacrifices are in some sense unavoidable without examining both operands, and of course the alternative, optimizing for the best case, can result in very bad worst-case complexities.

Some sacrifices in the performance of the actual multiplication algorithm (step 2) may also be made so that the conversion step (step 1) always runs fast (i.e. linear time). As mentioned before, when performing just a single multiplication, it makes no sense to spend more time parsing the input than the cost would be just to multiply using another method. Perhaps it would be beneficial to spend more time on step 1 when many operations are to be performed in sequence; this case has not yet been examined.

3 Dense "Chunks"

The first adaptive representation we consider is perhaps the most elegant. This is what we call the "chunky" representation, and it corresponds to writing a polynomial $f(x) \in \mathsf{R}[x]$ as:

$$f_1(x)x^{e_1} + f_2(x)x^{e_2} + \cdots + f_k(x)x^{e_k}, \tag{3.1}$$

with each $f_i(x) \in \mathsf{R}[x]$ a dense polynomial with degree less than some integer d_i. Specifically, we represent $f(x)$ by a vector of pairs of dense polynomial and exponent:

$$\langle (f_1(x), e_1), (f_2(x), e_2), \ldots, (f_k(x), e_k) \rangle.$$

Any reasonable implementation will have each $f_i(0) \neq 0$ and $s \leq \sum d_i \leq n$.

The elegance of this representation is in the way it provides a very fine gradient between the standard dense and sparse representations. At one extreme, if $k = 1$ and $e_1 = 0$, then there is only one dense chunk, $f_1(x)$, which is just the dense representation of $f(x)$. At the other extreme, if $k = s$ and each $d_i = 1$, then each $f_i(x) = a_i$ (the coefficients in (2.2)), and this corresponds to the sparse representation of $f(x)$. This duality will be exploited to achieve a good adaptive algorithm.

3.1 Multiplication (Step 2)

Suppose that, similarly, $g(x)$ is written as a sum of l dense "chunks" $g_1(x), g_2(x), \ldots, g_l(x)$ with degrees less than $e_1, e_2, \ldots, e_l$ (we are recycling some notation from earlier). Then multiplication of $f(x)$ and $g(x)$ will look like sparse multiplication on the outer loop, and dense multiplication in the inner loop.

The basic algorithm is to compute the product of each pair of dense coefficient polynomials $f_i(x)g_j(x)$ and merge them all together, according to the exponents, for the final result.

Since the cost of dense multiplication depends on which polynomial has greater degree, we need to define the relationships between the d_i's and e_i's in order to have a precise complexity analysis of chunky multiplication. So we sort the dense polynomial coefficients by degree; that is, assume $d_1 \leq d_2 \leq \cdots \leq d_k$ and $e_1 \leq e_2 \leq \cdots \leq e_l$. Next, define $c_1, c_2, \ldots, c_l \in \{0, 1, \ldots, k\}$ to be the "crossover points" such that $d_{c_i} \leq e_i \leq d_{c_i+1}$ for all i (with the convention that $d_0 = 0$. Then the cost is given as follows:

Theorem 1. *Multiplication of $f(x)$ and $g(x)$, when written in the chunky representation as described above, can be performed with*

$$O\left(\sum_{j=1}^{l}\left[\sum_{i=1}^{c_j}\frac{e_j}{d_i}\mathsf{M}(d_i) + \sum_{i=c_j+1}^{k}\frac{d_i}{e_j}\mathsf{M}(e_j)\right]\right)$$

ring operations in R.

Proof. We compute each product $f_i(x)g_j(x)$, for $1 \leq i \leq k$ and $1 \leq j \leq l$. If $1 \leq i \leq c_j$, then $d_i \leq d_{c_j} \leq e_j$, so the cost of computing this dense product is $\frac{e_j}{d_i}\mathsf{M}(d_i)$. And when $c_j + 1 \leq i \leq k$, $d_i \geq d_{c_j+1} \geq e_j$, so the cost of the dense product computation is $\frac{d_i}{e_j}\mathsf{M}(e_j)$. Merging each product into the result will just involve (at most) a linear number of additions in R and thus does not affect the total cost. □

It remains to discuss how the products are actually merged into the result, but we will postpone this discussion until after we see how to parse the input polynomials into the chunky representation.

3.2 Parsing (Step 1)

First, notice that our algorithm for chunky multiplication can be thought of as multiplying each chunky term $f_i(x)$ of $f(x)$ by the entire polynomial $g(x)$ and then merging the results. Hence minimizing the cost of multiplying $g(x)$ by an arbitrary dense polynomial (not chunky) will also minimize the cost of the full chunky multiplication.

So consider $g(x)$ to have just be one dense chunk with degree less than m. This means $l = 1$, so define $c = c_1$, and we have, from Theorem 1, that the cost of chunky multiplication will be

$$\sum_{i=1}^{c}\frac{m}{d_i}\mathsf{M}(d_i) + \sum_{i=c+1}^{k}\frac{d_i}{m}\mathsf{M}(m). \tag{3.2}$$

With the approximation $\frac{a}{b}\mathsf{M}(b) \approx a \lg b$, (3.2) becomes

$$\begin{aligned}
& \sum_{i=1}^{c} m \lg d_i + \sum_{i=c+1}^{k} d_i \lg m \\
< \;& m \sum_{i=1}^{c} (\log_2(d_i+1)+1) + \lg m \sum_{i=c+1}^{k} d_i \\
= \;& mc + m \log_2 \prod_{i=1}^{c} (d_i+1) + \lg m \sum_{i=c+1}^{k} d_i \qquad (3.3)
\end{aligned}$$

Since m is arbitrary and c is essentially a function of m, if we want to minimize this cost without a-priori knowledge of m, we must minimize the two terms $\prod(d_i+1)$ and $\sum d_i$.

Theorem 2. *Let $f(x), g(x), n, m, s, k, d_1, \ldots, d_k$ be as above. Then the chunky multiplication is always asymptotically at least as good as both the standard dense and sparse multiplication methods iff*

$$\prod_{i=1}^{k}(d_i+1) \in O(n) \qquad \text{and} \qquad \sum_{i=1}^{k} d_i \in O(s).$$

Proof. From the preceding discussion, we can see that the cost of chunky multiplication will be less than (3.3) above. First, note that each d_i must be at least 1; otherwise the corresponding dense "chunk" $f_i(x)$ is the zero polynomial and can just be eliminated. This means that

$$m \log_2 \prod_{i=1}^{c}(d_i+1) \geq m \log_2 \prod_{i=1}^{c} 2 = mc,$$

and so the second term in (3.3) always dominates the first. Therefore the first term is asymptotically insignificant and can be safely ignored for our purposes here.

Now we know the cost of dense multiplication will be

$$\Omega\left((\max\{n \log m, m \log n\}\right),$$

depending on which of n, m is larger. Given the conditions in the statement of the proof, and using the fact that $\sum d_i \leq n$, the cost of chunky multiplication is

$$\begin{aligned}
& O\left(m \log_2 \prod_{i=1}^{c}(d_i+1) + \lg m \sum_{i=c+1}^{k} d_i\right) \\
\in \;& O\left(m \log \prod_{i=1}^{k}(d_i+1) + \log m \sum_{i=1}^{k} d_i\right) \\
\in \;& O\left(m \log n + n \log m\right) \\
\in \;& O\left(\max\{n \log m, m \log n\}\right),
\end{aligned}$$

and is therefore asymptotically at least as fast as standard dense multiplication.

Again since each $d_i \geq 1$, $\log_2(d_i + 1) \leq d_i$. Therefore

$$\log_2 \prod (d_i + 1) = \sum \log_2(d_i + 1) \leq \sum d_i.$$

Using this fact, and again deriving from (3.3), the cost of chunky multiplication is asymptotically

$$\begin{aligned} & O\left(m \sum_{i=1}^{c} d_i + \lg m \sum_{i=c+1}^{k} d_i\right) \\ \in \; & O\left(m \sum_{i=1}^{k} d_i\right) \\ \in \; & O\left(ms\right) \end{aligned}$$

And the cost (in ring operations) of sparse multiplication will be $\Theta(ms)$. So the chunky multiplication will always be asymptotically at least as good as the standard sparse multiplication.

Two counterexamples will prove the other direction of the biconditional “iff”. First, suppose $\prod d_i \notin O(n)$; that is, suppose $n \in o(\prod d_i)$. Then suppose $g(x)$ is a dense polynomial with degree $m > n$. This means each $d_i < m$, so $c = k$, and we can use (3.2) to see that the cost of the chunky multiplication is

$$\sum_{i=1}^{k} \frac{m}{d_i} \mathsf{M}(d_i),$$

while the cost of dense multiplication will be $O(\frac{m}{n}\mathsf{M}(n))$.

Then we just need to follow through the reduction:

$$\begin{aligned} n &\in o\left(\prod_{i=1}^{k} (d_i + 1)\right) \\ \log_2 n &\in o\left(\sum_{i=1}^{k} \log_2(d_i + 1)\right) \\ \frac{m}{n}\mathsf{M}(n) &\in o\left(\sum_{i=1}^{k} \frac{m}{d_i}\mathsf{M}(d_i)\right) \end{aligned}$$

So the dense multiplication will be asymptotically faster than the chunky multiplication.

Finally, suppose $\sum d_i \notin O(s)$, so that $s \in o(\sum d_i)$. And let $g(x)$ be a constant polynomial, so that $m = 1$. In this case, $c = 0$, so the cost of chunky multiplication is

$$\sum_{i=1}^{k} \frac{d_i}{m} \mathsf{M}(m) \in \Theta\left(\sum_{i=1}^{k} d_i\right).$$

The cost of sparse multiplication will be $sm = s$. So if $s \in o(\sum d_i)$, then the sparse multiplication will always be asymptotically better than the chunky multiplication. $\square$

Theorem 2 gives a condition when the chunky multiplication will always be superior to other methods, and more importantly indicates a bit more explicitly that the two values to minimize in order to minimize the cost of chunky multiplication are $\prod(d_i + 1)$ and $\sum d_i$.

Unfortunately, the conditions of Theorem 2 are not always attainable, for example when $f(x)$ has approximately $\sqrt{n}$ nonzero terms spaced evenly apart. We will see later how to handle this specific example, but for now we need a way to convert the input to the chunky representation under any circumstances — optimal or not — and to perform the conversion in linear time in the size of the input.

3.2.1 Dense Input

First consider the case when the input is given in the dense representation. For this case, we definitely want to be competitive with the standard dense multiplication, so we should work primarily to minimize $\prod(d_i + 1)$. It is not clear how to minimize this value absolutely using only $O(n)$ time, but Algorithm 1 does reasonably well, essentially finding local minima for $\prod(d_i + 1)$.

The basic idea of the algorithm is to start with the standard dense representation, then break off "chunks" only when doing so will reduce the value of $\prod(d_i + 1)$. Then, since we start with $\prod(d_i + 1) \in O(n)$, we are guaranteed to maintain this property.

More specifically, the algorithm searches from the low-order to the high-order coefficients of $f(x)$ for all the "gaps" — that is, runs of zero coefficients. If the gap is large enough in relation to the dense chunks surrounding it so that the product $\prod(d_i + 1)$ would be reduced, the current chunk is split at the gap to make two new chunks.

Intuitively, the list L contains information about all previously-seen gaps which will also be benificial to split at *if we split at the current gap.* So if the current gap is sufficiently large to merit a split, all the splits designated by elements in L are also performed.

Theorem 3. *Algorithm 1 runs in $O(n)$ time and always returns a chunky representation with $\prod(d_i + 1) \in O(n)$.*

Proof. First we show that L has at most one element after each time step 9 is executed. So, by way of contradiction, assume L has two elements after step 9 is executed, say (r_1, a_1, b_1) and (r_2, a_2, b_2). First, from the way the b's are defined, we must have $a_1 < b_1$ and $a_2 < b_2$, and we can see also that $c_{a_1} = c_{a_2} = 0$ and $c_{b_1}, c_{b_2} \neq 0$. Then, since $b_1 \leq a_2$, we must in fact have $b_1 < a_2$. And for every e_i we always have $c_{e_i} \neq 0$, so $e_i < a_1$. Finally, at step 9, $c_j = 0$, so $b_2 < j$. Combining all this gives

$$e_i < a_1 < b_1 < a_2 < b_2 < j.$$

```
Input: $f(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_{n-1} x^{n-1} \in \mathsf{R}[x]$
 1: $e_1 \leftarrow \max\{i \in \mathbb{Z}^+ \text{ s.t. } x^i \mid f(x)\}$
 2: $k \leftarrow 1, \quad f_1(x) \leftarrow f(x)/x^{e_1}$
 3: $i \leftarrow 1, \quad j \leftarrow e_i$
 4: $L \leftarrow$ empty linked list of triples in $\mathbb{N}^3$
 5: while $j < n$ do
 6:   if $c_j \neq 0$ then
 7:     $j \leftarrow j + 1$
 8:   else
 9:     $L \leftarrow L \setminus \{(r, a, b) \in L \mid r < j\}$
10:     $b \leftarrow j + 1$
11:     while $b < n$ and $c_b = 0$ do
12:       $b \leftarrow b + 1$
13:     $r \leftarrow \lfloor (b - j)/(j - e_i) \rfloor + b$
14:     Add $(r, j, b)$ to $L$
15:     if $r \geq n - 1$ then
16:       while $L$ not empty do
17:         Remove $(r, a, b)$ from $L$ with $r$ minimal
18:         $f_i(x) \leftarrow f_i(x) \mod x^{a - e_i}$
19:         $i \leftarrow i + 1$
20:         $f_i(x) \leftarrow (f(x) - (f(x) \mod x^a))/x^b$
21:         $e_i \leftarrow b$
22:     $j \leftarrow b$
23: return $\langle (f_1(x), e_1), (f_2(x), e_2), \ldots, (f_i(x), e_i) \rangle$
```

Algorithm 1: Dense Polynomial to Chunky Conversion

From the way the r's are defined, we know that $r_1 = \lfloor (b_1 - a_1)/(a_1 - e_i) \rfloor + b_1$, and since (r_1, a_1, b_1) remains after step 9, $r_1 \geq j$. Therefore $b_1 - a_1 \geq (j - b_1)(a_1 - e_i)$. From the inequalities above, $a_1 - e_i \geq 1$ and $j - b_1 > b_2 - a_2$. Therefore we have $b_1 - a_1 > b_2 - a_2$.

By identical reasoning, we know that $b_2 - a_2 \geq (j - b_2)(a_2 - e_i)$. And again from the inequalities we know that $j - b_2 \geq 1$ and $a_2 - e_i > b_1 - a_1$. This means that $b_2 - a_2 > b_1 - a_1$.

But this is a contradiction with $b_1 - a_1 > b_2 - a_2$ from above. So the original assumption was wrong; namely, L has at most one element after each execution of step 9.

This means that L will never have more than two elements at any step, and so each step of the outer while loop runs in constant time. Since j increases at each iteration, the outer while loop runs at most n times. Therefore the total complexity is $O(n)$.

To see that the value of $\prod(d_i + 1)$ always decreases when we create a new chunk, let (r, a, b) be any triple removed from L at step 17. Since the triple is removed, this means that the chunk which contains the coefficients of x^a and x^b now has or will have its top coefficient of degree j.

And the bottom coefficient of the current chunk is x^{e_i}. So now let us consider the contributions to $\prod(d_i + 1)$ when we include the gap from x^a to x^b and when we do not. If the gap is included, then there is just one term contributed, with cost $j - e_i + 1$. If we split at the gap, then there are two terms contributed, with cost $(a - e_i + 1)(j - b + 1)$. Then, since $j \leq r$, we have

$$\begin{aligned}
(a - e_i + 1)(j - b + 1) &= (a - e_i)j - (a - e_i)b - (b - a) + (j - e_i + 1) \\
&\leq (b - a) + (a - e_i)b - (a - e_i)b - (b - a) + (j - e_i + 1) \\
&= j - e_i + 1
\end{aligned}$$

Thus the value of $\prod(d_i + 1)$ with the split is bounded above by the cost without it. Therefore this value only decreases, and since it is in $O(n)$ at the start of the algorithm, $\prod(d_i + 1) \in O(n)$ when the algorithm finishes also. $\square$

With some minor modifications, Algorithm 1 could simultaneously proceed from both sides instead of just reading the coefficients from left-to-right. This would likely give some improvement, as it would relax the lower bounds on the size of the gaps to make splitting profitable.

3.2.2 Sparse Input

If the input is given in the sparse representation, then the algorithm to parse $f(x)$ into the chunky representation is more or less the opposite of Algorithm 1. Rather than starting with just one chunk, we start with the most number of chunks reasonable. That is, we start by recognizing all the "natural" chunks - runs of nonzero coefficients. Then we use a heuristic measure to decide which chunks to combine, making sure not to introduce too many zero terms into the representation.

The heuristic measure is defined as follows. Suppose we have two consecutive dense chunks with degrees less than p_1 and p_2 and with g zero coefficients between them. Then combining these into a single dense chunk will result in the value $\prod(d_i+1)$ being divided by

$$\delta = \frac{(p_1+1)(p_2+1)}{a+b+g+1}.$$

So if $\delta > 1$, then combining the chunks will give a reduction in $\prod(d_i+1)$ (which is good). However, combining chunks always increases $\sum d_i$, by exactly g, and we definitely want to account for this in the sparse case. So the heuristic measure is the g'th root of δ, or $\delta^{1/r}$.

The algorithm also uses a "slack variable" ω, which is really just the constant in the big-O notation in the invariant $\sum d_i \in O(s)$. Algorithm 2 gives the full details.

```
**Input:** $f(x) = a_1x^{e_1} + \cdots + a_sx^{e_s}$ and $\omega \in \mathbb{R}^+$
 1: $F \leftarrow$ empty linked list of dense polynomial-nonnegative integer pairs
 2: $H \leftarrow$ max-heap of triples, sorted by the first element of the triple
 3: $k \leftarrow 0, \quad i \leftarrow 1$
 4: $p_1, p_2, g \in \mathbb{N}$
 5: while $i \le s$ do
 6:    $k \leftarrow k+1$
 7:    $j \leftarrow i+1$
 8:    while $j \le s$ and $e_j = e_{j-1}+1$ do
 9:       $j \leftarrow j+1$
10:    $p_1 \leftarrow p_2, \quad p_2 \leftarrow j-i$
11:    $g \leftarrow e_i - e_{i-1}$
12:    Insert $(a_i + a_{i+1}x + \cdots + a_{j-1}x^{j-1-i}, e_i)$ into $F$
13:    if $k > 1$ and $(p_1+1)(p_2+1)/(p_1+p_2+g+1) > 1$ then
14:       Insert $\Big(\big((p_1+1)(p_2+1)/(p_1+p_2+g+1)\big)^{1/g}, g, \text{reference to } F[k-1]\Big)$
          into $H$
15:    $i \leftarrow j$
16: $t \leftarrow 0$
17: $\big(c, g, (f_i(x), e_i)\big) \leftarrow$ top removed from $H$
18: while $t + g < \omega s$ do
19:    $t \leftarrow t+g$
20:    $f_i(x) \leftarrow f_i(x) + f_{i+1}(x)x^{e_{i+1}-e_i}$
21:    Remove $\big(f_{i+1}(x), e_{i+1}\big)$ from $F$
22:    if $H$ is empty then
23:       break
24:    else
25:       $\big(c, g, (f_i(x), e_i)\big) \leftarrow$ top removed from $H$
26: return $F$
```

Algorithm 2: Sparse Polynomial to Chunky Conversion

Theorem 4. *Algorithm 2 runs in time $O(s \log n)$ (the size of the input) and produces a chunky representation such that $\sum d_i \in O(s)$.*

Proof. The complexity is easily seen to be $O(s \log s)$, which is $O(s \log n)$, as long as we always save a pointer to the last two nodes of F in the first while loop.

And every iteration through the second while loop adds the value g to $\sum d_i$, which is also added to the value t. Since $t \leq \omega s$ is an invariant, and since $\sum d_i = s$ after the first while loop, we know that $\sum d_i \leq (\omega + 1)s \in O(s)$ when the algorithm terminates. □

Algorithm 2 could be improved, perhaps significantly, if we updated values in the heap affected each time we combine two chunks. Since only the adjacent gaps will be affected, this would not cause an increase in the asymptotic complexity. However, it would require a more complicated data structure with more links, etc., and so we do not examine it carefully here.

3.3 Merging the $f_i(x)g_j(x)$'s and Step 3

In [7], an algorithm for sparse polynomial multiplication is given which uses heaps of pointers into dynamic arrays to avoid intermediate expression swell in the standard sparse multiplication algorithm. Exactly the same technique can be used here, so we will not elaborate on the details. Basically, each product $f_i(x)g_j(x)x^{e_{i,j}}$ is computed in increasing order of the $e_{i,j}$'s — that is, the sum of the powers of x multiplied by $f_i(x)$ and $g_j(x)$ in $f(x)$ and $g(x)$, respectively.

There is just one extra step, since each computed product is in itself a dense polynomial. If the $e_{i,j}$ for the current product falls in the middle of the previous dense chunk, then increase the size of that chunk and add the new product $f_i(x)g_j(x)$ into it. Otherwise, create a new chunk with the value $f_i(x)g_j(x)$.

Unfortunately, this may produce chunky representations which are not optimal, so either Algorithm 1 or 2 will have to be run on the output. However, these should both terminate more quickly than they would if the input were a dense or sparse polynomial, just identifying the (hopefully) small number of chunks which need to be combined or split.

If the intermediate result is needed in the standard dense or sparse form, then we can just initialize space for the result and add each intermediate product $f_i(x)g_j(x)x^{e_{i,j}}$ into it. This may result in some intermediate expression swell in the sparse case, but the space complexity is never more than the time complexity. Using ideas similar to those in [14] may avoid this intermediate expression swell, but this has not yet been looked at.

4 Equally-Spaced Terms

Next we consider an adaptive representation which is in some sense orthogonal to the chunky representation. This representation will be useful when the coefficients of the polynomial are not grouped together into dense chunks, but rather when they are spaced evenly apart.

Suppose the exponents of $f(x)$ are all divisible by some integer k. Then we can write $f(x) = a_0 + a_1x^k + a_2x^{2k} + \cdots$. So if we let $f_1(x) = a_0 + a_1x + a_2x^2 + \cdots$, then we have $f(x) = f_1(x) \circ (x^k)$ (where the symbol $\circ$ indicates functional composition); another way to write this is $f(x) = f_1(x^k)$.

The generalization of this idea is the equal-spaced representation, which corresponds to writing $f(x)$ as

$$f(x) = f_1(x^k)x^d + f_2(x), \tag{4.1}$$

with $k, d \in \mathbb{N}$, $f_1(x)$ dense with degree less than $n/k - d$, and $f_2(x)$ sparse with degree less than n.

4.1 Equal-Spaced Multiplication (Step 2)

To multiply $f(x)$ by another polynomial in the equal-spaced representation, $g(x) = g_1(x^l)x^e + g_2(x)$ (as in (4.1)), we simply sum up the four pairwise products of terms. All these are performed using standard methods except, of course, for the product $f_1(x^k)g_1(x^l)$. For this, first notice that, if k and l are relatively prime, then almost any term in the product can be nonzero. On the other hand, if $k = l$ or one divides the other, the product will be equal-spaced as well.

This indicates that the gcd of k and l is very significant. In fact, it is crucial, but as we shall see, a speedup can be achieved even when k and l are relatively prime. To multiply $f_1(x^k)$ and $g_1(x^l)$, we perform a transormation similar to the process of finding common denominators in the addition of fractions. First split $f_1(x^k)$ as follows:

$$\begin{aligned} f_1(x^k) \quad = \quad & f_{1,1}(x^{\mathrm{lcm}(k,l)}) + x^{\gcd(k,l)}f_{1,2}(x^{\mathrm{lcm}(k,l)}) + x^{2\gcd(k,l)}f_{1,3}(x^{\mathrm{lcm}(k,l)}) + \\ & + \cdots + x^{\mathrm{lcm}(k,l)-\gcd(k,l)}f_{1,\mathrm{lcm}(k,l)/\gcd(k,l)-1}(x^{\mathrm{lcm}(k,l)}). \end{aligned}$$

Similarly split $g(x^l)$ into $g_{1,1}, g_{1,2}, \ldots$. Then compute all the pairwise multiplications $f_{1,i}(x)g_{1,j}(x)$, and combine them appropriately to compute the total sum (which will be equal-spaced with right composition factor $x^{\gcd(k,l)}$).

Algorithm 3 gives the details of this method.

Theorem 5. *Let $f(x), g(x)$ be as given above with the sparsities of $f_2(x)$ and $g_2(x)$ equal to $\hat{s}$ and $\hat{t}$, respectively.*

Then Algorithm 3 correctly computes the product $f(x)g(x)$ and uses

$$O\left(\frac{n}{\gcd(k,l)} \lg \frac{m}{lcm(k,l)} \lg\lg \frac{m}{lcm(k,l)} + \frac{n\hat{t}}{k} + \frac{m\hat{s}}{l} + \hat{s}\hat{t}\right)$$

ring operations in R.

Proof. Correctness of the algorithm follows from the preceding discussion.

The degrees of $f_1(x)$ and $g_1(x)$ must be less than n/k and n/l, respectively. Then the cost of computing $f_1(x^k)g_2(x)x^d + g_1(x^l)f_2(x)x^e + f_2(x)g_2(x)$ in step

```
Input: $f(x) = f_1(x^k)x^d + f_2(x), \quad g(x) = g_1(x^k)x^e + g_2(x)$,
    with $f_1(x) = a_0 + a_1x + a_2x^2 + \cdots, \quad g_1(x) = b_0 + b_1x + b_2x^2 + \cdots$
 1: $r \leftarrow \gcd(k, l)$
 2: for $i = 0$ to $\frac{l}{r} - 1$ do
 3:    $f_{1,i}(x) \leftarrow a_i + a_{i+l/r}x + a_{i+2l/r}x^2 + \cdots$
 4: for $i = 0$ to $\frac{k}{r} - 1$ do
 5:    $g_{1,i}(x) \leftarrow b_i + b_{i+k/r}x + b_{i+2k/r}x^2 + \cdots$
 6: $h_1(x) \leftarrow 0$
 7: for $i = 0$ to $\frac{l}{r} - 1$ do
 8:    for $j = 0$ to $\frac{k}{r} - 1$ do
 9:       $h_1(x) \leftarrow h_1(x) + x^{(i+j)}(f_{1,i}(x)g_{1,j}(x)) \circ (x^{kl/r})$
10: $h_2(x) \leftarrow f_1(x^k)g_2(x)x^d + g_1(x^l)f_2(x)x^e + f_2(x)g_2(x)$
11: return $h_1(x^r)x^{d+e} + h_2(x)$
```

Algorithm 3: Equal-Spaced Polynomial Multiplication

10 by using standard sparse multiplication is $O(\frac{n\hat{t}}{k} + \frac{m\hat{s}}{l} + \hat{s}\hat{t})$ ring operations, giving the last three terms in the complexity measure.

The initialization of the polynomials $f_{1,i}, g_{1,i}$ in steps 2–5 just involves copying coefficient values, and so takes linear time. From the definitions, the degree of each $f_{1,i}$ is less than $n/\text{lcm}(k,l)$, and similarly the degree of each $g_{1,i}$ is less than $m/\text{lcm}(k,l)$. Then, since $\text{lcm}(k,l) = kl/\gcd(k,l)$, the total cost of initialization is $n/k + m/l$. This is just the size of the input polynomials $f_1(x)$ and $g_1(x)$, and so therefore this step does not dominate the complexity.

The computation of the product $f_{1,i}(x)g_{1,j}(x)$ in step 9 is performed with dense multiplication, and so uses $O((n/m)\mathsf{M}(m/\text{lcm}(k,l)))$ ring operations. Multiplication by and composition with a power of x take linear time, as does the addition to the result $h_1(x)$. Then, since step 9 is performed $kl/\gcd(k,l)^2$ times, the total cost (using the DFT multiplication method) is

$$\begin{aligned} & \frac{kln}{\gcd(k,l)^2 m} \frac{m}{\text{lcm}(k,l)} \lg \frac{m}{\text{lcm}(k,l)} \lg\lg \frac{m}{\text{lcm}(k,l)} \\ = \; & \frac{kln}{\gcd(k,l)^2} \frac{\gcd(k,l)}{kl} \lg \frac{m}{\text{lcm}(k,l)} \lg\lg \frac{m}{\text{lcm}(k,l)} \\ = \; & \frac{n}{\gcd(k,l)} \lg \frac{m}{\text{lcm}(k,l)} \lg\lg \frac{m}{\text{lcm}(k,l)}, \end{aligned}$$

which gives the first term in the comlexity measure. □

4.2 Converting from Equal-Spaced (Step 3)

Step 3 of the Equal-Spaced adaptive algorithm is extremely simple. Algorithm 3 gives output in the form $h_1(x^r)x^q + h_2(x)$, with $h_1(x)$ dense and $h_2(x)$ sparse.

To convert from this to either the standard dense or sparse representations of the result, we just need two loops. First, initialize the result to zero (with the proper degree specified). Next, for each coefficient a_i of x^i in $h_1(x)$, set the coefficient of x^{ri+q} in the result to a_i. Then for each coefficient b_i with exponent e_i in $h_2(x)$, add b_i to the coefficient of x^{e_i} in the result.

This conversion can again be combined with the algorithm of step 2 for greater efficiency when the result is not merely an intermediate step in a series of computations. Simply initialize $h_1(x)$ in step 6 of Algorithm 3 to use the same representation as the desired result representation. Then, on step 9, add the value

$$x^{(i+j)r+d+e}(f_{1,i}(x)g_{1,j}(x)) \circ (x^{kl/r})$$

to $h_1(x)$ instead. Finally, rather than create a new sparse polynomial $h_2(x)$ on step 10, just add this into $h_1(x)$, and return just the polynomial $h_1(x)$.

4.3 Converting to Equal-Spaced (Step 1)

The only question when converting $f(x)$ to the equal-spaced representation is how large we should allow $\hat{s}$ (the sparsity of $f_2(x)$) to be. For this, we will use the independent method of conversion, and to be conservative we will assume the worst case — that $g(x)$ is totally dense of degree m.

Theorem 6. *If $\hat{s} \in O(\lg k)$, the equal-spaced multiplication method is never asymptotically worse than the standard dense multiplication method.*

Proof. Theorem 5 tells us that the worst case for the algorithm (if $f(x)$ is fixed) will be when $g(x)$ is totally dense. Now we cannot assume that $n \geq m$, so we have two cases.

Case 1: $m \geq n$ The number of ring operations required for the equal-spaced method in this case will be $O(m \lg(n/k) \lg\lg(n/k) + m\hat{s})$. Since $\hat{s} \in O(\lg k)$, this is $O(m \lg n \lg\lg n)$, which is the cost of the dense multiplication method.

Case 2: $n \geq m$ In this case, the equal-spaced method requires $O(n \lg(m/k) \lg\lg(m/k) + m\hat{s})$ ring operations. Then, using the fact that $m \leq n$ and the same reduction as before, we have the cost bounded by $O(n \lg m \lg\lg m)$, which is the cost of dense multiplication.

□

A similar result comparing the equal-spaced method to the sparse method is more difficult unless we make some assumptions on the density of $f_1(x)$.

Now the question is how to find k. Specifically, if the exponents of the original polynomial $f(x)$ are $e_1, e_2, \ldots, e_s$, define the set $S \subseteq \mathbb{Z}$ by

$$S = \{e_2 - e_1, e_3 - e_2, \ldots, e_s - e_{s-1}\}.$$

Then we want to find

$$k = \max\{\gcd(T) \mid T \subseteq S,\ |T| \geq n - \log k\}.$$

Of course, this is a recursive definition, but we can simplify it by the fact that $k \leq n$, so $n - \log k > n - \log n$.

Even then, this is a non-trivial problem; in fact it is closely related to the max factor k-gcd problem, which is provably NP-hard [12]. To solve the problem, we must make a few assumptions. First, we assume that the size of the input is at least $\Omega(n)$. Of course this is true if $f(x)$ is given in the dense representation, but not necessarily if it is given as sparse.

The second assumption is a relatively minor one, namely that no subset of S of size greater than $n - \sqrt{n}\log n$ has a gcd which does not divide k. Intuitively, we need this assumption because otherwise there could be a candidate k which *almost* works, and we have no way of determining, in linear time, that it doesn't.

The idea for the following algorithm is based on how we might compute $\gcd(S)$, by constructing a list with all the elements in S, then repeatedly reducing the length of the list by one half by taking the gcd's of pairs of elements in the list (similar to mergesort). Then when the list has only one element remaining, that number must be equal to $\gcd(S)$.

For this algorithm, we instead stop when the size of the list is less than $2\sqrt{n}$, performing only about half the number of reduction steps. Then we factor each number in the list completely, and return the product of all factors of at least $\sqrt{n} - \log n$ numbers in the list, which will equal k if we make the assumption above.

```
Input: $S \subseteq \mathbb{N}$ and $n \in \mathbb{N}$, with $|S| < n$ and $\forall s \in S,\ s \leq n$.
 1: $L \leftarrow$ list of numbers in $S$ {elements accessible via $L_1, L_2, \ldots$}
 2: while $|L| \geq 2\sqrt{n}$ do
 3:    $L \leftarrow \langle \gcd(L_1, L_2), \gcd(L_3, L_4), \gcd(L_5, L_6), \ldots \rangle$
 4: $u \leftarrow |L|$
 5: $T \leftarrow$ balanced binary tree-backed dictionary {initially empty}
 6: for $i = 1$ to $u$ do
 7:    Compute the complete prime factorization of $L_i$: $p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots$.
 8:    for all $p_j^{e_j}$ do
 9:       Increase $T[p_j], T[p_j^2], \ldots, T[p_j^{e_j}]$ by 1
10: $k \leftarrow 1$
11: while $T$ not empty do
12:    $r \leftarrow$ largest key in $T$
13:    if $T[r] \geq u - \log n$ and $r \nmid k$ then
14:       $k \leftarrow rk$
15:    remove $r$ from $T$
16: return $k$
```

Algorithm 4: Algorithm to find k from the set S of coefficient differences

Theorem 7. *Algorithm 4 correctly computes k, given the assumption that every gcd of at least $n - \sqrt{n}\log n$ numbers from S divides k, and provided that $|S|$ and $\sqrt{n}$ are each greater than $\log n$. Furthermore, the algorithm uses $O(n)$ word operations and operations on integers less than or equal to n.*

Proof. First note that $|S| < s \leq n$.

Suppose the complete prime factorization of k is $p_1^{e_1} p_2^{e_2} p_3^{e_3}$. From the definition of k, there are at most $\log n$ elements in S that are not divisible by $p_i^{e_i}$, for each i. Then since, at any point in the algorithm, each L_i is the gcd of a disjoint subset of S, there are at most $\log n$ elements of L that are not divisible by $p_i^{e_i}$.

Now, if $|L| < 2\sqrt{n}$ initially, then the first while loop never runs, and so $u = |S| > \log n$. And since $|L| \geq 2\sqrt{n}$ before the last iteration of the first while loop, we know that $u \geq \sqrt{n} > \log n$. So there is always at least one element remaining in L after step 4 which is divisible by $p_i^{e_i}$.

Then at step 10 we know that $T[p_i^{e_i}] \geq u - \log n \geq 1$. Therefore by step 16, $p_i^{e_i} \mid k$. This means that the real value of k divides the value returned from the algorithm.

Now suppose the two values are not the same, that is, suppose the true k is a proper divisor of the number returned from the algorithm. Then there exists a prime power divisor of the returned value which does not divide the true k, say p^e.

Notice that every key added to T is a prime power. So define the set

$$P = \{p^j \mid T[p^j] \geq u - \log n \text{ at step } 10\}.$$

Since the returned value is a product of keys from T, we know that P has at least one element. Furthermore, $\max P$ will be removed from T and multiplied into the value k before any other values from P. Then since each element in P divides $\max P$, no other element in P is multiplied into the result on line 14.

Therefore, since p^e divides the returned value, $p^e | \max P$. Since $T[\max P] \geq u - \log n$, the value must have been incremented at least $u - \log n$ times on line 9. Therefore p^e was a divisor of at least $u - \log n$ of the L_i's. Since each one of those was a gcd of at least $\lfloor \frac{n}{u} \rfloor$ entries from S, this means that p^e divides the gcd of a subset of S with at least $n - \sqrt{n} \log n$ entries. So from the condition of the theorem, p^e divides the true value of k.

This is a contradiction, so the original statement must have been false; namely, the returned k is equal to the true value of k.

For the complexity, first note that each entry in L is always less than or equal to n, since the gcd of two numbers is never larger than either number. Then, if we kept taking gcds until $|L| = 1$, there would be $O(|S|) \in O(n)$ nodes total, and computing each gcd takes constant time, so the total cost would be $O(n)$; we are stopping early (as soon as $|L| < 2\sqrt{n}$), so the cost will still be $O(n)$.

To factor each L_i at step 7, we can use standard trial division at a cost of $O(\sqrt{n})$ for each factorization; using a faster factorization algorithm will probably be better in practice. Then, since $u < 2\sqrt{n}$, the total cost to factor all the L_i's will be $O(n)$.

Next, notice that since each $L_i \leq n$, the sum of the exponents in the prime factorization of L_i must be at most $\log_2 n$. So we only increment $O(\log n)$ values in T for each L_i. This means that $|T| \in \sqrt{n} \log n$, and so if we used any balanced binary tree algorithm for T, the total cost of step 9 for all iterations is

$$O(\sqrt{n} \log n \log(\sqrt{n} \log n)) \in O(\sqrt{n} \log^2 n) \in O(n).$$

Using a hash table for T is likely to give even better results in practice.

Finally, since $|T| \in O(\sqrt{n} \log n)$, the last while loop runs in $O(n)$. Therefore the complexity of the entire algorithm is $O(n)$. $\square$

Once k is determined, the coefficients of $f_1(x)$ and $f_2(x)$ can be determined by making one more pass through the input polynomial and testing whether each exponent is divisible by k. So this gives a complete algorithm for step 1, at least when the input is given in the dense representation.

If the input is given in the sparse representation, we can still find a k, just under the relatively heavy restriction that $\hat{s} = 0$, by continuing the first while loop in Algorithm 4 to find the gcd of all of S.

We can also speed up Algorithm 4 with some early termination conditions, making it adaptive. In the first while loop, stop whenever more than $\log n$ entries in L are equal to 1, or when at least $|L| - \log n$ entries in L are equal to each other. In the former case, return 1, and in the latter, return the common entry.

5 Coefficients in Sequence

Both of the preceding adaptive approaches require that at least one of the input polynomials has few nonzero terms in its dense representation. However, there are some polynomials which are completely dense but which can be multiplied quickly.

One class of such polynomials is those whose coefficients are in an arithmetic or a geometric sequence. Recall that these are sequences of the forms

$$\{a + bi\}_{i \geq 0} \qquad \text{and} \qquad \{cd^i\}_{i \geq 0},$$

respectively. Here, $a, b, c, d \in \mathsf{R}$ and $i \in \mathbb{Z}^+$, and the first elements in the sequences are a and c, respectively.

Specifically, the representation we will use for this approach is to represent the input polynomial $f(x)$ with the tuple $(a, b, c, d, f_2(x))$, where $a, b, c, d \in \mathsf{R}$ are the parameters of an arithmetic and a geometric sequence, as above, and $f_2(x)$ is a sparse polynomial in $\mathsf{R}[x]$. Under this representation, $f(x)$ is defined as the sum of $f_2(x)$ and the following two polynomials:

$$f_a(x) = a + (a+b)x + (a+2b)x^2 + \cdots + (a+bi)x^i + \cdots \tag{5.1}$$

$$f_g(x) = c + cdx + cd^2x^2 + \cdots + cd^ix^i + \cdots . \tag{5.2}$$

So the coefficients of $f(x)$ are the sum of an arithmetic and a geometric sequence, with a few coefficients in $f_2(x)$ that did not fit into the sequence sum. Note that this representation actually captures the coefficients being the sums of an arbitrary number of arithmetic sequences, since the sum of two arithmetic sequences is still an arithmetic sequence. However, this property does not hold for geometric sequences, so that is a limitation of this representation.

5.1 Sequential Coefficients Multiplication (Step 2)

Note that, in general, the product of two polynomials whose coefficients come from an arithmetic or from a geometric sequence will not have coefficients that form any arithmetic or geometric sequence (or sum of two of them). So it does not make much sense to produce output in the sequential-coefficients representation. And, except for very special circumstances, the output will always be dense, and so it will not be useful to give output in the sparse representation. Therefore, for this adaptive approach, the output will *always* be given in the standard dense representation; hence we ignore step 3.

To see how the algorithm works, let $g(x)$ be an arbitrary dense polynomial of degree less than m with coefficients $g_0, g_1, g_2, \ldots, g_{m-1}$, so we write $g(x)$ as

$$g(x) = g_0 + g_1x + g_2x^2 + \cdots + g_{m-1}x^{m-1}.$$

Now first consider $f_a(x)$ as in (5.1) above, with degree less than n. Then the first few coefficients of the product $f_a(x)g(x)$ are

$$ag_0, a(g_0 + g_1) + bg_0, a(g_0 + g_1 + g_2) + b(2g_0 + g_1), \ldots.$$

A similar pattern exists for the last few coefficients, which are

$$\ldots, a(g_{m-2} + g_{m-1}) + b((n-1)g_{m-2} + (n-2)g_{m-1}), ag_{m-1} + b(n-1)g_{m-1}.$$

This leads to Algorithm 5 for multiplication by a dense polynomial whose coefficients form an arithmetic sequence.

Theorem 8. *Algorithm 5 gives correct output and uses $O(n + m)$ ring operations.*

Proof. The correctness of the algorithm can be confirmed simply by examining the coefficients of direct multiplication for the different cases, as we did briefly above.

For the complexity, we can see first that the number of iterations through any for loop is exactly $n + m - 1$. Each line that updates $h(x)$ (lines 6, 11, 15, and 19) is really just setting one coefficient of the result, and so can be performed in constant time. And each iteration through any for loop uses a constant number of ring operations.

Therefore the total number of ring operations is $O(n + m)$, and this also dominates any other word operations performed. □

We take a similar approach to develop an algorithm for multiplication when the coefficients form a geometric sequence. So consider $f_g(x)$ as in (5.2) above, with degree less than n. The first few terms of the product $f_g(x)g(x)$ are:

$$cg_0, c(dg_0 + g_1), c(d^2g_0 + dg_1 + g_2), \ldots,$$

and the last few terms are:

$$\ldots, c(d^{n-1}g_{m-2} + d^{n-2}g_{m-1}), cd^{n-1}g_{m-1}.$$

```
Input: $(a, b, n) \in \mathsf{R} \times \mathsf{R} \times \mathbb{Z}^+$ representing $f_a(x)$ and $g(x)$ dense with degree
    less than $m$
 1: $h(x) \leftarrow$ dense polynomial with degree less than $n + m - 1$
 2: $\alpha, \beta \leftarrow 0 \in \mathsf{R}$
 3: for $i = 0$ to $\min\{n - 1, m - 1\}$ do
 4:   $\beta \leftarrow \beta + \alpha$
 5:   $\alpha \leftarrow \alpha + g_i$
 6:   $h(x) \leftarrow h(x) + (a\alpha + b\beta)x^i$
 7: if $n < m$ then
 8:   for $i = n$ to $m - 1$ do
 9:     $\beta \leftarrow \beta + \alpha - ng_{i-n}$
10:     $\alpha \leftarrow \alpha - g_{i-n} + g_i$
11:     $h(x) \leftarrow h(x) + (a\alpha + b\beta)x^i$
12: else if $m < n$ then
13:   for $i = m$ to $n - 1$ do
14:     $\beta \leftarrow \beta + \alpha$
15:     $h(x) \leftarrow h(x) + (a\alpha + b\beta)x^i$
16: for $i = \max\{n, m\}$ to $n + m - 2$ do
17:   $\beta \leftarrow \beta + \alpha - ng_{i-n}$
18:   $\alpha \leftarrow \alpha - g_{i-n}$
19:   $h(x) \leftarrow h(x) + (a\alpha + b\beta)x^i$
20: return $h(x)$
```

Algorithm 5: Arithmetic Sequence Coefficients Multiplication

```
Input: $(c, d, n) \in \mathsf{R} \times \mathsf{R} \times \mathbb{N}$ representing $f_g(x)$ and $g(x)$ dense with degree less
    than $m$
 1: $h(x) \leftarrow$ dense polynomial with degree less than $n + m - 1$
 2: $\alpha \leftarrow 0 \in \mathsf{R}$
 3: for $i = 0$ to $\min\{n - 1, m - 1\}$ do
 4:   $\alpha \leftarrow d\alpha + g_i$
 5:   $h(x) \leftarrow h(x) + c\alpha x^i$
 6: if $n < m$ then
 7:   for $i = n$ to $m - 1$ do
 8:     $\alpha \leftarrow d\alpha - d^n g_{i-n} + g_i$
 9:     $h(x) \leftarrow h(x) + c\alpha x^i$
10: else if $m < n$ then
11:   for $i = m$ to $n - 1$ do
12:     $\alpha \leftarrow d\alpha$
13:     $h(x) \leftarrow h(x) + c\alpha x^i$
14: for $i = \max\{n, m\}$ to $n + m - 2$ do
15:   $\alpha \leftarrow d\alpha - d^n g_{i-n}$
16:   $h(x) \leftarrow h(x) + c\alpha x^i$
17: return $h(x)$
```

Algorithm 6: Geometric Sequence Coefficients Multiplication

This leads to Algorithm 6 for multiplication by a polynomial whose coefficients form a geometric sequence. The proof of Theorem 9 is nearly identical to that of Theorem 8, and so we omit it for the sake of brevity.

Theorem 9. *Algorithm 6 gives correct output and uses $O(n+m)$ ring operations.*

Using the preceding two algorithms, it is not too difficult to construct Algorithm 7 for general multiplication of polynomials in the sequential coefficients representation:

```
Input: $f(x) = f_a(x) + f_g(x) + f_2(x)$ and $g(x) = g_a(x) + g_g(x) + g_2(x)$ with
    degrees less than $n$ and $m$, respectively.
1: $g(x) \leftarrow g_a(x) + g_g(x) + g_2(x)$ (dense representation)
2: $h(x) \leftarrow f_a(x)g(x)$ via Algorithm 5
3: $h(x) \leftarrow h(x) + f_g(x)g(x)$ via Algorithm 6
4: $h(x) \leftarrow h(x) + g_a(x)f_2(x)$ via Algorithm 5
5: $h(x) \leftarrow h(x) + g_g(x)f_2(x)$ via Algorithm 6
6: $h(x) \leftarrow h(x) + f_2(x)g_2(x)$ via Sparse Multiplication
7: return $h(x)$
```

Algorithm 7: Sequential Coefficients Polynomial Multiplication

Theorem 10. *Algorithm 7 produces the product $f(x)g(x)$ using $O(n+m+\hat{s}\hat{t})$ ring operations, where $\hat{s}$ and $\hat{t}$ are the sparsities of $f_2(x)$ and $g_2(x)$, respectively.*

Proof. Correctness follows from the definitions and Theorems 8 and 9.

Step 1 is just addition, so it uses $O(m)$ ring operations. Steps 2–5 each use $O(n+m)$ ring operations, by Theorems 8 and 9. And any sparse multiplication algorithm will use $O(\hat{s}\hat{t})$ ring operations for step 6.

Therefore the total complexity is $O(n+m+\hat{s}\hat{t})$ ring operations. □

5.2 Recognizing Coefficient Sequences (Step 1)

Suppose we are given a polynomial $f(x)$ to parse into the sequential-coefficients representation. Since the multiplication will always take at least linear time in the degree of $f(x)$ (from Theorem 10), we can always convert $f(x)$ into the dense representation first without taking any (asymptotically) extra time.

Now given a dense $f(x) \in \mathsf{R}[x]$ with degree less than n, the question is how large we should allow $\hat{s}$ — the number of terms that do not fall into the arithmetic-geometric sequence — to be. The following theorem answers this question.

Theorem 11. *The sequential-coefficients multiplication algorithm will always be asymptotically at least as fast as standard dense multiplication iff $\hat{s} \in O(\lg n)$.*

Proof. First, assume $\hat{s} \in O(\lg n)$, and suppose we are multiplying by a polynomial $g(x) \in \mathsf{R}[x]$ with degree m. Standard dense multiplication will use

$\Theta(\max\{n \lg m, m \lg n\})$ ring operations. But we will always have $\hat{t} \leq m$, so from Theorem 10, the sequential-coefficients multiplication algorithm will only use $O(n + m + \hat{s}m)$ ring operations, which is $O(n + m \lg n)$, which in turn is $O(\max\{n \lg m, m \lg n\})$. So the sequential-coefficients algorithm is never worse in this case.

For the other direction, suppose $\hat{s} \notin O(\lg n)$ — that is, $\lg n \in o(\hat{s})$. And let $g(x)$ be a dense polynomial with degree $m - 1 > n$ whose coefficients are not in any kind of arithmetic-geometric sequence. Then dense multiplication can use as few as $O(m \lg n)$ ring operations, which is $o(m + n + m\hat{s})$, the cost of sequential-coefficients multiplication. □

So for some implementation-dependant slack variable $\omega \in \mathbb{R}^+$, we will require the number of terms in $f(x)$ that do not lie in the arithmetic-geometric sequence to be at most $\omega \lg n$. Now define r to be the length of the longest *consecutive* run of coefficients in $f(x)$ that are not in $f_2(x)$ — that is, they are in the arithmetic-geometric sequence. Since the sparsity of $f_2(x)$ is at most $\omega \lg n$, a pigeon hole principle-like argument tells us that

$$r \geq \left\lceil \frac{n+1}{\omega \lg n + 1} - 1 \right\rceil .$$

Then, since $\log^2 n \in o(n)$, $\omega \lg n \in \Theta(\log n)$, and $r \in \Theta(n/\log n)$, there exists a constant n_0 such that whenever $n \geq n_0$, $r > \omega \lg n$.

If n is less than such a n_0, then its size can be considered contant, and we just perform standard dense multiplication. Otherwise we know that, if $f(x)$ can be parsed into the sequential-coefficients representation with the sparsity of $f_2(x)$ at most $\omega \lg n$, then there exists a consecutive run of at least r coefficients in $f(x)$ that do not appear in $f_2(x)$, and furthermore no other consecutive run of at least r coefficients forms an arithmetic-geometric sequence.

Additionally, we can see that the differences of terms in the arithmetic sequence $\{a + bi\}_{i \geq 0}$ will all be b, and the quotients of terms in the geometric sequence $\{cd^i\}_{i \geq 0}$ will all be d. Given that we can perform division in R, Algorithm 8 finds the sequential-coefficients representation of $f(x)$ or determines that none exists with the sparsity of $f_2(x)$ at most $\omega \lg n$.

Theorem 12. *Algorithm 8 correctly computes the sequential-coefficients representation of $f(x)$, if it exists and $n \geq n_0$. Otherwise, the algorithm outputs* FAIL. *At most $O(n)$ ring operations are used in the computation.*

Proof. As the algorithm runs, the α's store differences between coefficients, the β's store differences between the α's, and the γ's store quotients between the β's. If the coefficients are all in the arithmetic-geometric sequence defined by (a, b, c, d), then we know that each $c_i = a + bi + cd^i$. This means that the differences (i.e. the α's) will have the form $b + c(d - i)d^i$. The differences of these (i.e. the β's) will have the form $c(d-i)^2 d^i$. And the quotients of these (i.e. the γ's) will all be d. So we have the following easily-verifiable loop invariant each time the condition of the while loop on line 9 is evaluated, provided the

```
Input: $f(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{n-1}x^{n-1} \in \mathsf{R}[x]$, $\omega \in \mathbb{R}^+$
r ← $\lceil (n+1)/(\omega \lg n + 1) \rceil - 1$
if $r \le \omega \lg n$ then
    return FAIL
$\alpha_0, \beta_0, \gamma_0, \alpha_1, \beta_1, \gamma_1 \in \mathsf{R}$
$i \leftarrow 4, k \leftarrow 1 \in \mathbb{N}$
$\alpha_0 \leftarrow c_2 - c_1, \quad \alpha_1 \leftarrow c_3 - c_2$
$\beta_0 \leftarrow \alpha_0 - (c_1 - c_0), \quad \beta_1 \leftarrow \alpha_1 - \alpha_0$
$\gamma_1 \leftarrow \beta_1/\beta_0$
while $i < n$ and $k < r - 3$ do
    $(\alpha_0, \beta_0, \gamma_0) \leftarrow (\alpha_1, \beta_1, \gamma_1)$
    $\alpha_1 \leftarrow c_i - c_{i-1}$
    $\beta_1 \leftarrow \alpha_1 - \alpha_0$
    $\gamma_1 \leftarrow \beta_1/\beta_0$
    if $\gamma_1 = \gamma_0$ then
        $k \leftarrow k + 1$
    else
        $k \leftarrow 1$
    $i \leftarrow i + 1$
if $k < r - 3$ then
    return FAIL
$d \leftarrow \gamma_1$
$c \leftarrow \beta_1/[(d-1)^2 d^{i-2}]$
$b \leftarrow \alpha_0 - c(d-1)d^{i-2}$
$a \leftarrow c_{i-2} - b(i-2) - cd^{i-2}$
$f_2(x) \leftarrow 0$ (dense, degree $n - 1$)
$\hat{s} \leftarrow 0 \in \mathbb{Z}^+$
for $j = 0$ to $n - 1$ do
    if $c_j \ne a + bi + cd^i$ then
        $\hat{s} \leftarrow \hat{s} + 1$
        if $\hat{s} > \omega \lg n$ then
            return FAIL
        $f_2(x) \leftarrow f_2(x) + [c_j - (a + bi + cd^i)]x^i$
return $(a, b, c, d, f_2(x))$
```

Algorithm 8: Conversion to Sequential-Coefficients Representation

coefficients c_{i-3} through c_i are all in the arithmetic-geometric sequence (i.e. not in $f_2(x)$):

$$\begin{array}{ll} \alpha_0 = b + c(d-1)d^{i-2} & \alpha_1 = b + c(d-1)d^{i-1} \\ \beta_0 = c(d-1)^2 d^{i-3} & \beta_1 = c(d-1)^2 d^{i-2} \\ \gamma_1 = d. & \end{array}$$

So if c_{i-r+1} through c_i are all in the arithmetic-geometric sequence, then the γ's will be equal for $r-3$ iterations, causing the while loop to terminate with $k = r-3$. The converse is also true — namely, if the γ's are equal for at least $r-3$ iterations, then c_{i-r+1} through c_i are all terms in a single arithmetic-geometric sequence. If such a sequence exists with all the coefficients of $f(x)$ except for $\omega \lg n$, and $n \geq n_0$, then this means c_{i-r+1} through c_i lie in the arithmetic-geometric sequence we are looking for, and so γ_1 must equal the value of parameter d for the sequence.

Then, from the loop invariant, we can see that the values of c, b, a are correctly computed in steps 22 through 24. So if the sequence exists and $f_2(x)$ has sparsity at most $\omega \lg n$, then the if condition on line 28 is true no more than $\omega \lg n$ times, and therefore the value of $\hat{s}$ is never more than $\omega \lg n$, so the algorithm terminates normally on the last line and returns the correct parameters.

Furthermore, if the algorithm terminates on the last line and returns anything other than `FAIL`, then from the last for loop, we can see clearly that $f(x)$ must be represented by $(a, b, c, d, f_2(x))$ with the sparsity of $f_2(x)$ no more than $\omega \lg n$. Therefore the algorithm always gives the correct output.

For the complexity, clearly the while loop on line 9 and the for loop on line 27 each run at most n times. The bodies of both loops use a constant number of ring operations, provided we keep one extra local variable for the value of d^i in the for loop, and multiply it by d at each iteration rather than recomputing d^i. The value d^{i-2} for lines 22 to 24 can be computed by repeated squaring with $O(\lg i) \in O(\lg n)$ multiplications, and everything else also uses just a constant number of ring operations. So the total cost is $O(n)$. □

For Algorithm 8, as we mentioned earlier, we must be able to perform division in R efficiently. If R is a field, this is always possible. In other cases, we still may be able to perform Euclidian division (i.e. division with remainder), which is good enough for our purposes here. If division is not possible in R, then it is easy to modify the algorithm to only identify arithmetic sequences, as we currently have no other way of identifying geometric sequences.

6 Further Directions

6.1 Implementation

It remains to be seen whether the theoretical improvements we give here bear any practical fruit. To test this, a careful implementation of the above algorithms is needed. Special care must be given to memory access issues to reduce overhead from caching, which is not accounted for in the theoretical complexity measures, but which is likely to be very significant in the actual cost of the algorithms, especially since the polynomials for which the above algorithms give any improvement are likely to have very high degree.

For dense polynomial multiplication, the reference implementation would almost certainly be NTL [10], which appears to be the most efficient implementation of fast univariate polynomial arithmetic over a variety of rings and fields.

For sparse polynomial multiplication, it is not quite so clear what the reference implementation should be. It seems that practitioners are not very interested specifically in *univariate* sparse polynomial multiplication (more on this below). Every general-purpose computer algebra software program implements sparse polynomial arithmetic, often by default. However, these programs are usually not very highly tuned, and it generally isn't quite fair to compare highly specialized against general-purpose software. However, add-on packages, such as the one discussed in [7], could be useful as a reference implementation.

Designing test cases for the adaptive polynomial multiplication algorithms given above is also problematic. One avenue for exploration would be to specifically choose polynomials that should be handled well by the adaptive algorithms (for example, polynomials that actually have dense chunks with large gaps in between). Another method would be to simply choose random polynomials; however, it seems unlikely that any of the adaptive algorithms would be faster in the general case. Very useful would be a class of polynomials that fit one or more of the adaptive categories well and which are actually used in practice, but we are not currently aware of any such class at this time.

6.2 Combination of Ideas

Just as the various adaptive algorithms for sorting were combined to give asymptotically superior adaptive algorithms, so it seems desirable to have one adaptive algorithm which somehow captures all three of the ideas we present here. A brief sketch of such an algorithm is given below.

The representation (and therefore the corresponding algorithms) will be recursive. This may seem inefficient, but in fact the last comprehensive test of different polynomial representations resulted in the conclusion that a recursive representation was best [11], so there is some precedent (although in a rather different arena).

If the input polynomial $f(x)$ is dense, we first perform the sequential-coefficients conversion to give a sparse polynomial. This sparse polynomial is then parsed into the chunky representation. Next, each dense chunk is parsed into the equal-spaced representation. Then each dense polynomial resulting is given to the sequential-coefficients parsing algorithm, and the recursion continues until all three adaptive conversion algorithms give no improvement.

If $f(x)$ is given in the sparse representation, we do the same thing, but start by performing the chunky coversion and moving down recursively from there.

One key advantage to this setup is that the polynomials given to each conversion algorithm always look the same: the sequential-coefficients and equal-spaced conversion algorithms always get a densely-represented polynomial, and the chunky conversion algorithm always gets a sparsely-represented polynomial. This allows us to tune the algorithms more finely, and also avoids some of the problems that the sequential-coefficients and equal-spaced algorithms had with sparse input polynomials.

Though seemingly simple on the surface, this combination of ideas is quite difficult to analyze and clearly define. However, a careful analysis and even more careful implementation are likely to produce better results than any of the individual adaptive algorithms gives.

6.3 Multivariate

One of the main advantages in using the sparse representation of polynomials is that the total (i.e. dense) number of coefficients grows exponentially with the number of indeterminates of the polynomial. For this reason, it seems that the primary current use in practice of sparse polynomials is in representing multivariate polynomials. So in order to have a more useful adaptive algorithm for polynomial multiplication, we must handle the multivariate case.

For most of the ideas discussed here, it seems that simply imposing an ordering on the indeterminates makes the multivariate case relatively straightforward. However, there are numerous details to work out for this case. And a more powerful adaptive algorithm should be able to give the optimal term ordering for multiplication, given one, two, or more multiplicands. This is an entirely new challenge which we have not yet considered.

7 Conclusion

The three main ideas presented here, chunky, equal-spaced, and sequential-coefficients multiplications, all give substantial theoretical improvements over both the standard sparse and dense multiplaction algorithms for certain types of inputs. Furthermore, the worst-case behavior is usually no worse than the standard algorithms, making these adaptive algorithms asymptotically superior.

We do not yet know what kind of practial improvements can be achieved with these methods; however, it is clear that there are many classes of univariate polynomials which are in some sense "easier" to multiply. Adaptive algorithms which take advantage of the structure of the input polynomails seem very promising in improving these well-studied computational problems which have such a central importance in computer algebra.

References

[1] Peter Bürgisser and Martin Lotz. Lower bounds on the bounded coefficient complexity of bilinear maps. *J. ACM*, 51(3):464–482 (electronic), 2004.

[2] David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Inform.*, 28(7):693–701, 1991.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[4] Martin Fürer. Faster integer multiplication. pages 57–66, 2007.

[5] Stephen C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8(3):63–71, 1974.

[6] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Dokl. Akad. Nauk SSSR*, 7:595–596, 1963.

[7] Michael Monagan and Roman Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. Preprint; accepted to CASC 2007.

[8] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informat.*, 7(4):395–398, 1976/77.

[9] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971.

[10] Victor Shoup. Ntl: A library for doing number theory. Online, http://www.shoup.net/ntl/, 2007.

[11] David R. Stoutemeyer. Which polynomial representation is best? In *Proc. 1984 MACSYMA Users' Conference*, pages 221–244, Schenectady, NY, 1984.

[12] Staal A. Vinterbo. Maximum k-intersection, edge labeled multigraph max capacity k-path, and max factor k-gcd are all NP-hard. Technical Report DSG TR 2002/12, Decision Systems Group, Brigham and Women's Hospital, 2002.

[13] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra.* Cambridge University Press, Cambridge, second edition, 2003.

[14] Thomas Yan. The geobucket data structure for polynomials. *J. Symbolic Comput.*, 25(3):285–293, 1998.