CPSC 518 — Introduction to Computer Algebra
Asymptotically Fast Integer Multiplication

1 Introduction

We have now seen that the Fast Fourier Transform can be applied to perform polynomial multiplication over arbitrary commutative rings. This handout presents an older and similar result, namely, the application of these techniques to obtain an asymptotically fast algorithm for integer multiplication.

The algorithm discussed in this handout was discovered and presented by Schönhage and Strassen [4]. The text of Aho, Hopcroft and Ullman [1] is another good reference for this material.

2 Overview and Presentation of the Algorithm

Inputs and Outputs

Suppose once again that $n = 2^k$ is a power of two and that we are given the binary representation of two integers u and v, whose product is less than n, as input.

In this case, $u \cdot v$ is trivially recovered from $u \cdot v \bmod 2^n + 1$.

Consequently (since this will simplify the development of a recursive algorithm) we will consider a slightly more general problem, where u and v given as n-bit integers

$$u = u_0 + u_1 \cdot 2 + u_2 \cdot 2^2 + \cdots + u_{n-1}2^{n-1}$$

and

$$v = v_0 + v_1 \cdot 2 + v_2 \cdot 2^2 + \cdots + v_{n-1}2^{n-1}$$

where $u_i, v_i \in \{0, 1\}$ for $0 \le i \le n-1$, and where we wish to compute the bits $w_0, w_1, \dots, w_n$ of an integer

$$w = w_0 + w_1 \cdot 2 + w_2 \cdot 2^2 + \cdots + w_{n-1}2^{n-1} + w_n 2^n$$

such that

$$0 \le w \le 2^n \qquad \text{and} \qquad w \equiv u \cdot v \bmod 2^n + 1. \tag{1}$$

Computation When n is Small

If n is small (that is, less than a threshold that is set during the implementation and analysis of this algorithm) then standard integer multiplication — or, perhaps, Karatsuba's algorithm — will be used to compute $u \cdot v$. Suppose that

$$u \cdot v = \widehat{w}_0 + \widehat{w}_1 \cdot 2^n$$

for integers $\widehat{w}_0$ and $\widehat{w}_1$ such that $0 \le \widehat{w}_0, \widehat{w}_1 < 2^n$; then it is easily checked (since $2^n \equiv -1 \bmod 2^n + 1$) that

$$u \cdot v \equiv \widehat{w}_0 - \widehat{w}_1 \bmod 2^n + 1$$

and, indeed, that

$$w = \begin{cases} \widehat{w}_0 - \widehat{w}_1 & \text{if } \widehat{w}_0 \ge \widehat{w}_1, \\ 2^n + 1 + \widehat{w}_0 - \widehat{w}_1 & \text{if } \widehat{w}_0 < \widehat{w}_1, \end{cases}$$

so that w can then be recovered from $u \cdot v$ using a constant number of additions and subtractions.

Since this technique is only used when n is small its asymptotic complexity does not influence the asymptotic complexity of the algorithm that is being developed. However — since Karatsuba's algorithm will be used in another part of the algorithm, described below — it is worth noting that this complexity is subquadratic if Karatsuba's algorithm is used: $O(n^{\log_2 3})$ operations on bits (hence, $O(n^{1.6})$ operations, since $\log_2 3 < 1.6$) can be used to carry out the computations that are described above.

Reduction to Computation and Evaluation of a Convolution of Polynomials

For larger n we will attempt to solve the given problem by recursively forming and solving instances of the problem involving integers that have length approximately $\sqrt{n}$. Recall that, by assumption, $n = 2^k$ for a nonnegative integer k. Let

$$m = 2^{\lfloor k/2 \rfloor} \qquad \text{and} \qquad t = n/m = 2^{\lceil k/2 \rceil} \tag{2}$$

so that $n = m \cdot t$ and either $t = m$ (if k is even) or $t = 2m$ (if k is odd). Consider polynomials

$$\widehat{u} = \widehat{u}_{m-1}x^{m-1} + \widehat{u}_{m-2}x^{m-2} + \cdots + \widehat{u}_1 x + \widehat{u}_0 \in \mathbb{Z}[x] \tag{3}$$

and

$$\widehat{v} = \widehat{v}_{m-1}x^{m-1} + \widehat{v}_{m-2}x^{m-2} + \cdots + \widehat{v}_1 x + \widehat{v}_0 \in \mathbb{Z}[x] \tag{4}$$

where $0 \le \widehat{u}_i, \widehat{v}_i < 2^t$ for $0 \le i \le m-1$ and where

$$u = \widehat{u}(2^t) \qquad \text{and} \qquad v = \widehat{v}(2^t).$$

Binary representations of $\widehat{u}_0, \widehat{u}_1, \ldots, \widehat{u}_{m-1}$ (respectively, $\widehat{v}_0, \widehat{v}_1, \ldots, \widehat{v}_{m-1}$) can be obtained by splitting the binary representation of u (respectively, v) into m equally sized pieces using a linear number of operations on bits.

Suppose that $\widehat{w} \in \mathbb{Z}[x]$ is a polynomial such that

$$\widehat{w} = \widehat{w}_{m-1}x^{m-1} + \widehat{w}_{m-2}x^{m-2} + \cdots + \widehat{w}_1 x + \widehat{w}_0 \equiv \widehat{u} \cdot \widehat{v} \bmod x^m + 1; \tag{5}$$

that is, suppose $\widehat{w}$ is a negative wrapped convolution of the polynomials $\widehat{u}$ and $\widehat{v}$. Then it must be the case that

$$\widehat{u}(x) \cdot \widehat{v}(x) = \widehat{q}(x) \cdot (x^m + 1) + \widehat{w}(x)$$

for some polynomial $\widehat{q} \in \mathbb{Q}[x]$. Furthermore, since $\widehat{u}$, $\widehat{v}$ and $x^m + 1$ all have integer coefficients, and $x^m + 1$ is monic, it is easily checked that

$$\widehat{q} \in \mathbb{Z}[x].$$

Consequently

$$\widehat{q}(2^t) \in \mathbb{Z},$$

and

$$\begin{aligned} u \cdot v &= \widehat{u}(2^t) \cdot \widehat{v}(2^t) \\ &= \widehat{q}(2^t) \cdot ((2^t)^m + 1) + \widehat{w}(2^t) \\ &= \widehat{q}(2^t) \cdot (2^n + 1) + \widehat{w}(2^t), \end{aligned}$$

so that

$$w = u \cdot v \bmod 2n + 1 = \widehat{w}(2^t) \bmod 2^n + 1. \tag{6}$$

We may therefore compute the desired value w by computing the coefficients of the polynomial $\widehat{w}$, evaluating this polynomial at 2^t, and then dividing this value by $2^n + 1$ (returning the remainder that is obtained).

Bounding the Coefficients of $\widehat{w}$

By construction, the polynomials $\widehat{u}$ and $\widehat{v}$ each have degree less than m, and the coefficients of each are between 0 and $2^t - 1$. Note that the coefficient of x^i in the product of $\widehat{u}$ and $\widehat{v}$ is a sum of products

$$\widehat{u}_0 \cdot \widehat{v}_i + \widehat{u}_1 \cdot \widehat{v}_{i-1} + \cdots + \widehat{u}_{i-1} \cdot \widehat{v}_1 + \widehat{u}_i \cdot \widehat{v}_0,$$

where $\widehat{u}_j = \widehat{v}_j = 0$ if $j \geq m$. At most m of the terms in the above sum are nonzero, so it follows that the above coefficient is greater than or equal to zero and strictly less than $m \cdot 2^{2t}$.

The product of $\widehat{u}$ and $\widehat{v}$ has degree less than $2m$. Suppose that

$$\widehat{u} \cdot \widehat{v} = \widehat{z}_0 + \widehat{z}_1 x^m$$

for polynomials $\widehat{z}_0, \widehat{z}_1 \in \mathbb{Z}[x]$ with degree at most $m-1$; then

$$\widehat{w} = \widehat{u} \cdot \widehat{v} \bmod (x^m + 1) = \widehat{z}_0 - \widehat{z}_1,$$

and each coefficient of $\widehat{w}$ is the *difference* between two of the coefficients of $\widehat{u} \cdot \widehat{v}$. It follows from the above that if $\widehat{w}$ is as shown in equation (5), above, then

$$-m \cdot (2^{2t}+1) < -m \cdot 2^{2t} < \widehat{w}_i < m \cdot 2^{2t} < m \cdot (2^{2t}+1) \qquad \text{for } 0 \le i < m, \tag{7}$$

so that each coefficient of $\widehat{w}$ can be recovered from its residue modulo $2m \cdot (2^{2t}+1)$. Most of the rest of the algorithm that is presented below will therefore be devoted to the computation of the coefficients of

$$\widehat{w} \bmod 2m \cdot (2^{2t}+1).$$

Recall that m is a power of two, so that $2m$ and $2^{2t}+1$ are relatively prime. We can therefore break the above computation down into three stages:

1. Compute $\widehat{w} \bmod 2^{2t}+1$.
2. Compute $\widehat{w} \bmod 2m$.
3. Use the above residues and an efficient Chinese remaindering process to recover the desired residue $\widehat{w} \bmod 2m \cdot (2^{2t}+1)$.

Each of these stages (and the final computation of $w = \widehat{w}(2^t) \bmod 2^{2t}+1$ from the above values) are discussed separately, below.

Computation of $\widehat{w}$ mod $2^{2t}+1$

It is sufficient for this stage of the computation to consider

$$\overline{u} = \widehat{u} \bmod 2^{2t}+1 \qquad \text{and} \qquad \overline{v} = \widehat{v} \bmod 2^{2t}+1$$

as polynomials whose coefficients belong to the domain

$$D = \mathbb{Z}_{2^{2t}+1}$$

and to notice that the desired output is a negative wrapped convolution (specifically, the convolution $\overline{u} \times \overline{v} \bmod x^m + 1$) of the above polynomials in this domain.

We can apply an FFT-based algorithm to compute this computation if we can confirm that there is a $2m^{\text{th}}$ primitive root of unity in the domain D. Notice, therefore, that $2t$ is a power of two, since t is, and recall that either $t = m$ or $t = 2m$. It can be shown that 2 is a $4t^{\text{th}}$ primitive root of unity in D, and it follows that $\eta = 4$ is a $2m^{\text{th}}$ primitive root of unity in D if $t = m$ and that $\eta = 8$ is a $2m^{\text{th}}$ primitive root of unity in D if $t = 2m$.

A consideration of the FFT-based algorithms that have already presented will confirm that it is necessary and sufficient to perform $O(m \log m)$ arithmetic operations in D — all of which are additions, subtractions, and multiplications by powers of η — along with m multiplications of pairs of elements of D, in order to apply such an algorithm.

Note that it is possible to perform an addition, subtraction, or multiplication by a power of η in D using $O(t)$ operations on bits. It follows that the total number of operations on bits, required for everything except the above-mentioned multiplications in D, is in $O(tm \log m) = O(n \log n)$.

Suppose, now, that $T(n)$ is an upper bound on the cost to compute the product of two n-bit integers mod $2^n + 1$ using the algorithm that is now being described. Consider any one of the multiplications of pairs of elements of D that remain to be performed; suppose, in particular, that we wish to compute the product of two elements α and β of D. Since α and β are each represented as residues of integers mod $2^{2t} + 1$ we may treat these as integers between 0 and 2^{2t}. The following cases should be checked for and handled.

(a) $0 \le \alpha, \beta < 2^{2t}$: In this case α and β are each integers with length at most $2t$ and their product can be computed modulo $2^{2t} + 1$ by a recursive application of the algorithm that is being described, with cost at most $T(2t)$.

(b) $0 \le \alpha < 2^{2t}$ and $\beta = 2^{2t}$. Since $\beta \equiv -1 \bmod 2^{2t} + 1$, the desired product is 0 if $\alpha = 0$ and it is $2^{2t} + 1 - \alpha$ otherwise. In either case the desired product can be computed using $O(t)$ operations on bits.

(c) $\alpha = 2^{2t}$ and $0 \le \beta < 2^{2t}$. Exchanging α and β we obtain the previous case. The desired product can be computed using $O(t)$ operations on bits as well.

(d) $\alpha = \beta = 2^{2t}$. Since $\alpha \equiv \beta \equiv -1 \bmod 2^{2t} + 1$ in this case, the desired product is 1. This can be computed using $O(t)$ operations on bits in this case once again.

Since $n = m \cdot t$ the overhead associated with the detection and separate handling of these cases, and with all computations that are not part of recursive applications of this algorithm, can be carried out using $O(n \log n)$ operations on bits. Thus the total number of operations on bits required for this stage of the computation is at most

$$mT(2t) + \widetilde{c} n \log n$$

for some constant $\widetilde{c}$ and for sufficiently large n.

Computation of $\widehat{w}$ mod $2m$

This stage of the computation is carried out using essentially a reversal of the trick that is used at the top level of the algorithm: Instead of reducing integer multiplication to the computation of the convolution of polynomials, we will reduce the computation of a convolution to a (much smaller) integer multiplication.

Consider polynomials $\widetilde{u}, \widetilde{v} \in \mathbb{Z}[x]$ having coefficients that are between 0 and $2m-1$ such that

$$\widetilde{u} \equiv u \bmod 2m \qquad \text{and} \qquad \widetilde{v} \equiv v \bmod 2m,$$

and let

$$\widetilde{z} = \widetilde{u} \cdot \widetilde{v},$$

so that each of the coefficients of $\widetilde{u}$ or $\widetilde{v}$ is an integer between 0 and $2m-1$ and we wish to compute the coefficients of $\widetilde{z} \bmod x^m + 1$.

Since $\widetilde{u}$ and $\widetilde{v}$ each have degree less than m it is easily checked that each coefficient of $\widetilde{z}$ is an integer that is greater than or equal to 0 and strictly less than $m \cdot (2m)^2 = 4m^3$. Recall that $m = 2^{\lfloor k/2 \rfloor} \leq \sqrt{n}$, so that $4m^3 = 2^{3\lfloor k/2 \rfloor + 2}$ is a power of two and, furthermore, that the binary representation of $4m^3$ has length that is in $O(\log n)$, since this length is linear in k. Now notice that the binary representations of the coefficients of the above polynomial $\widetilde{z}$ are easily obtained from the binary representation of the integer $\widetilde{z}(4m^3)$: In particular, the binary representations of the coefficients are obtained by splitting the binary representation of $\widetilde{z}(4m^3)$ into pieces that each have length $3\lfloor k/2 \rfloor + 2$.

We may therefore compute the coefficients of the above polynomial $\widetilde{z}$ as follows.

1. Compute $u^* = \widetilde{u}(4m^3)$ and $v^* = \widetilde{v}(4m^3)$. (Note that the binary representations of these integers have length in $O(\sqrt{n} \log n)$.
2. Compute $z^* = u^* \cdot v^*$ from u^* and v^*.
3. Compute the coefficients of $\widetilde{z}$ from $z^* = \widetilde{z}(4m^3)$.

The binary representation of u^* (respectively, v^*) can be obtained by padding the binary representations of the coefficients of $\widetilde{u}$ and $\widetilde{v}$ with sufficiently many 0's to ensure that each has length $3\lfloor k/2 \rfloor + 2$, and then concatenating these representations together. Thus the number of operations on bits needed to carry out the first step is linear in the lengths of the output integers, that is, in $O(\sqrt{n} \log n)$.

As mentioned above, the final step can be carried out essentially by reversing the process that was used to carry out the first step. This can be implemented to use $O(\sqrt{n} \log n)$ operations on bits as well.

If Karatsuba's algorithm is used to carry out the integer multiplication needed in the middle step then this can be performed without adding recursive applications of the algorithm that is being described, and at reasonably low cost. In particular, since $\frac{1}{2}\log_3 2 < 1$, this step can be carried out using $O(n)$ operations on bits.

Finally, notice that $\widehat{w} \bmod 2m$ is equal to $\widetilde{z} \bmod (x^m + 1)$, where both are considered as polynomials with coefficients in $\mathbb{Z}_{2m}$, and where $\widetilde{z}$ is the polynomial whose coefficients have been computed above. Each coefficient of $\widehat{w} \bmod 2m$ can be computed as the difference between a pair of coefficients of $\widetilde{z}$, modulo $2m$, so that the desired output can be computed from the coefficients of $\widetilde{z}$ using $O(n)$ operations on bits as well.

Thus the total number of operations on bits needed for this stage of the computation is in $O(n)$.

Computation of $\widehat{w} \bmod 2m \cdot (2^{2t} + 1)$

Let
$$\overline{w} = \overline{w}_0 + \overline{w}_1 x + \cdots + \overline{w}_{m-1}x^{m-1} = \widehat{w} \bmod 2^{2t} + 1$$
and let
$$\widetilde{w} = \widetilde{w}_0 + \widetilde{w}_1 x + \cdots + \widetilde{w}_{m-1}x^{m-1} = \widehat{w} \bmod 2m,$$
so that $0 \le \overline{w}_i \le 2^{2t}$ and $0 \le \widetilde{w}_i \le 2m - 1$ for $0 \le i \le m - 1$; these are the results of the last two stages of the algorithm that have been described. We now wish to describe an efficient process that uses the above values to compute the polynomial
$$\breve{w}_0 + \breve{w}_1 x + \cdots + \breve{w}_{m-1}x^{m-1} = \widehat{w} \bmod 2m(2^{2t} + 1),$$
so that
$$0 \le \breve{w}_i \le 2m(2^{2t} + 1) - 1 \quad \text{and} \quad \breve{w}_i \equiv \begin{cases} \overline{w}_i & \bmod (2^{2t} + 1) \\ \widetilde{w}_i & \bmod 2m \end{cases} \quad \text{for } 0 \le i \le m - 1. \tag{8}$$

If the Chinese remainder algorithm was used for this computation, and the worst case running time of this general algorithm was used in an algorithm analysis, then the resulting time bound would be too high to be useful here. Fortunately, it is possible to solve the above instances of Chinese remaindering much more efficiently than is generally the case, so that this technique can still be used.

Recall that m and t are each powers of two and that $m \le t$, so that
$$2m \le 2t \le 2^{2t}.$$

Thus $\left(\frac{2^{2t}}{2m}\right)$ is an integer, and a power of two, as well. Furthermore, if
$$\sigma = -\left(\frac{2^{2t}}{2m}\right) \qquad \text{and} \qquad \tau = 1 \tag{9}$$

then

$$\sigma \cdot (2m) + \tau \cdot (2^{2t} + 1) = 1. \tag{10}$$

It is easily checked that

$$\begin{aligned} 2^{2t} \cdot (\widetilde{w}_i - \overline{w}_i) + \widetilde{w}_i &= -2^{2t} \cdot \overline{w}_i + (2^{2t} + 1) \cdot \widetilde{w}_i \\ &= \sigma \cdot 2m \cdot \overline{w}_i + \tau \cdot (2^{2t} + 1) \cdot \widetilde{w}_i \end{aligned}$$

for σ and τ as given in equation (9). Now equation (10) and the above derivation can be used to establish that

$$2^{2t} \cdot (\widetilde{w}_i - \overline{w}_i) + \widetilde{w}_i \equiv \begin{cases} \overline{w}_i & \bmod (2^{2t} + 1) \\ \widetilde{w}_i & \bmod 2m \end{cases}$$

so that the conditions on $\breve{w}_i$ given at line (8) are achieved if we set

$$\breve{w}_i = \left(2^{2t} \cdot (\widetilde{w}_i - \overline{w}_i) + \widetilde{w}_i\right) \bmod 2m \cdot (2^{2t} + 1). \tag{11}$$

Now, since the binary representations of $\widetilde{w}_i$ and $\overline{w}_i$ each have length in $O(t)$, it should be clear that one can compute the binary representation of

$$2^{2t} \cdot (\widetilde{w}_i - \overline{w}_i) + \widetilde{w}_i \tag{12}$$

from the representations of $\widetilde{w}_i$ and $\overline{w}_i$ using $O(t)$ operations on bits. Since the binary representation of $2m \cdot (2^{2t} + 1)$ is quite sparse (its binary representation has only two 1's in it), division with remainder of the value at line (12) by $2m \cdot (2^{2t} + 1)$ can be carried out using a constant number of additions and subtractions of integers that have length in $O(t)$ as well, so that this can also be carried out using $O(t)$ operations on bits.

Consequently, a binary representation of $\breve{w}_i$ can be computed from representations of $\widetilde{w}_i$ and $\overline{w}_i$ using $O(t)$ operations on bits. Since $n = m \cdot t$ and the polynomial $\widehat{w}$ has degree less than m, it follows that $\widehat{w} \bmod 2m \cdot (2^{2t} + 1)$ can be computed from $\widehat{w} \bmod (2^{2t} + 1)$ and $\widehat{w} \bmod 2m$ using $O(n)$ operations on bits in total.

Computation of w from $\widehat{w} \bmod 2m \cdot (2^{2t} + 1)$

Notice that if the coefficient of x^i in $\widehat{w}$ is $\widehat{w}_i$, as shown in equation (5) on page 3, and if $\breve{w}_i$ is as described in the above section, then

$$\breve{w}_i = \widehat{w}_i \bmod 2m \cdot (2^{2t} + 1),$$

and the bounds given at line (7) on page 4 can be used to establish that

$$\widehat{w}_i = \begin{cases} \breve{w}_i & \text{if } \breve{w}_i < m \cdot 2^{2t}, \\ \breve{w}_i - 2m \cdot (2^{2t} + 1) & \text{if } \breve{w}_i \geq m \cdot 2^{2t}, \end{cases}$$

so that all of the coefficients of $\widehat{w}$ can be recovered from their residues modulo $2m \cdot (2^{2t}+1)$ using a total of $O(m \cdot t) = O(n)$ operations on bits.

Since $m \le t \le 2^t$, $m \cdot 2^{2t} \le 2^{3t}$ and the binary representation of each of the coefficients of $\widehat{w}$ has length at most $3t$. With a bit of work (associated mainly with a consideration of how to handle negative coefficients) one can see that it is possible to evaluate $\widehat{w}$ at 2^t, computing the value $\widehat{w}(2^t)$ from the coefficients of $\widehat{w}$, using $O(mt) = O(n)$ operations on bits as well. The binary representation of this integer has length $O(n)$, and it can be argued that the desired value

$$w = \widehat{w}(2^t) \bmod 2^n + 1$$

can be computed from $\widehat{w}(2^t)$ using $O(n)$ additional operations on bits after that.

3 Analysis of the Algorithm

Recall that $n = 2^k$, $t = 2^{\lceil k/2 \rceil}$, and $m = 2^{\lfloor k/2 \rfloor}$, so that $2t \le 4\sqrt{n}$. If the threshold mentioned in the section concerning the "computation when n is small" is chosen to be greater than or equal to 17 (and, in practice, a significantly higher threshold than that should be used) then this will ensure that the input size decreases whenever the above algorithm is recursively applied. Termination of the algorithm is then easily established. Partial correctness (that is, correctness of the outputs on termination, assuming that the algorithm does eventually halt) can be established by a consideration of the information given above. Thus the above algorithm can be shown to be correct.

Suppose once again that $T(n)$ is the number of operations used by this algorithm on bits, for inputs having length at most n, in the worst case. Suppose again that n is a power of two; then the information given above can be used to show that — again, for sufficiently large n —

$$T(n) \le mT(2t) + cn \log n \tag{13}$$

for some positive constant c.

Let us increase our bound on the threshold again — suppose that $n \ge 256 = 2^8$, so that $n^{1/4} \ge 2^2 = 4$ and $4\sqrt{n} \le n^{3/4}$.

Suppose, as well, that

$$T(\ell) \le \widehat{c}\ell \log \ell \log\log \ell$$

for "sufficiently large" powers of two that are less than n and, in particular, that this inequality is satisfied when $\ell = 2t$. Then — assuming here, and hereafter, that logarithms in formulas have base two —

$$\begin{aligned}
T(n) &\le mT(2t) + cn\log n && \text{(as noted above)}\\
&\le m\cdot(\widehat{c}\cdot 2t\log(2t)\log\log(2t)) + cn\log n && \text{(by the above assumption)}\\
&= \widehat{c}\cdot 2n\log(2t)\log\log(2t) + cn\log n && \text{(since } n = m\cdot t\text{)}\\
&\le \widehat{c}\cdot 2n\log(4\sqrt{n})\log\log(4\sqrt{n}) + cn\log n && \text{(since } t \le 2\sqrt{n}\text{)}\\
&\le \widehat{c}\cdot 2n\log(4\sqrt{n})\log\log(n^{3/4}) + cn\log n\\
&\qquad \text{(since } 4\sqrt{n} \le n^{3/4} \text{ when } n \ge 256\text{, as noted above)}\\
&= \widehat{c}\cdot 2n\cdot(\tfrac{1}{2}\log n + 2)\cdot(\log\log n - \log(4/3)) + cn\log n\\
&= \widehat{c}n\log n\log\log n - ((\log(4/3))\cdot\widehat{c} - c)n\log n + 4\widehat{c}n\log\log n - 4(\log(4/3))\widehat{c}n\\
&\le \widehat{c}n\log n\log\log n - ((\log(4/3)\cdot\widehat{c} - c)n\log n + 4\widehat{c}n\log\log n\\
&\le \widehat{c}n\log n\log\log n,
\end{aligned}$$

for sufficiently large n (since $n\log\log n \in o(n\log n)$), provided that $\widehat{c}$ is chosen such that the inequality

$$(\log(4/3))\cdot\widehat{c} > c$$

is satisfied as well.

To make this more concrete — and confirm that all the constraints, mentioned above, can be satisfied — one can check that $\log(4/3) > 0.4$ so that, for any given value of c, it is possible to choose $\widehat{c}$ such that

$$(\log(4/3))\widehat{c} - c > 0.4\widehat{c} - c \ge 0.2\widehat{c}.$$

In particular, this is the case whenever $\widehat{c}$ is chosen such that

$$\widehat{c} \ge 5c. \tag{14}$$

Suppose that $\widehat{c}$ is chosen to satisfy this inequality. Then the above derivation is correct (because the value at the penultimate line is less than the value at the bottom line) if n is large enough to ensure that

$$0.2\log n > 4\log\log n,$$

that is,

$$\log n > 20\log\log n.$$

This is true when $\log n \ge 144$, that is, when $n > 2^{144}$.

Notice that if $2^{144} < n \le 2^{286}$ then

$$2^{73} < 2t \le 2^{144}.$$

We should therefore choose c such that the number of operations on bits needed to apply this algorithm, excluding recursive calls, is at most $cn\log n$ for sufficiently large n, in order to ensure that the recurrence given at line (13) is valid. If $\widehat{c}$ is chosen such that $\widehat{c} \geq 5c$, as indicated at line (14), above, and also so that

$$T(\ell) \leq \widehat{c}\ell \log \ell \log \log \ell$$

whenever $\ell = 2^{\widehat{k}}$ such that $73 \leq \widehat{k} \leq 144$ (as needed to complete the proof of a "basis"), then an inductive argument can be used to prove that

$$T(n) \leq \widehat{c}n \log n \log \log n$$

whenever $n = 2^k$ for $k \geq 144$ by following the argument outlined above.

If n is not a power of two then one should observe that

$$n \leq \widehat{n}$$

for some power $\widehat{n}$ of two such that $n \leq \widehat{n} < 2n$ and observe that

$$T(n) \leq T(\widehat{n}) \leq \widehat{c}\,\widehat{n} \log \widehat{n} \log \log \widehat{n} \leq 8\widehat{c}n \log n \log \log n$$

in order to establish that $T(n) \in O(n \log n \log \log n)$ — that is,

$$T(n) \leq c^* n \log n \log \log n$$

for *all* sufficiently large n and for some positive constant c^*.

It should be clear (noting, in particular, the above assumption that $n > 2^{144}$) that the above argument is only sufficient to obtain an asymptotic result — it does not establish that FFT-based algorithms are of any practical interest. For evidence that they *are* of some practical use and, indeed, are already included in at least one widely used library, see the information that is available online about the GNU MP Bignum Library [5].

4 A More Recent Result

Suppose the function $\log^{(i)} n$ is defined recursively, for nonnegative integers i, as follows.

$$\log^{(i)} n = \begin{cases} n & \text{if } i = 0, \\ \log(\log^{(i-1)} n) & \text{if } i > 0 \text{ and } \log^{(i-1)} n > 0, \\ \text{undefined} & \text{if } i > 0 \text{ and } \log^{(i-1)} n \leq 0 \text{ or } \log^{(i-1)} n \text{ is undefined.} \end{cases}$$

The *iterated logarithm function* $\log^* n$ is defined as

$$\log^* n = \min\{i \geq 0 \,:\, \log^{(i)} n \leq 1\}.$$

This is a *very* slowly growing function: $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$, $\log^* 65536 = 4$, and $\log^*(2^{65536}) = 5$. One rarely encounters a value n, in any realistic application, such that $\log^* n > 5$.

Quite recently, an integer multiplication algorithm that is asymptotically faster than the algorithm presented here has been given by Martin Fürer [3]: Fürer's multiplication algorithm can be used to compute the product of two n-bit integers using $O(n \cdot (\log n) \cdot 2^{c \log^* n})$ bit operations for some positive constant c.

Fürer's algorithm uses (approximate) arithmetic over the complex numbers; a modular algorithm (like the one that has been presented here) with a similar asymptotic complexity has been given, quite recently, by De, Kurer, Saha and Saptharish [2].

It is not known, yet, whether these new algorithms will be of any practical interest. Again, though, they certainly do improve our understanding of the asymptotic complexity of integer multiplication.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. Available at http://arxiv.org/pdf/0801.1416v1, 2008.

[3] M. Fürer. Faster integer multiplication. In *Proceedings of the 48th ACM Symposium on Theory of Computing*, pages 57–66, 2007. Extended Abstract. A more complete version is available online at http://www.cse.psu.edu/~furer/Papers/mult.pdf.

[4] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[5] *The GNU MP Bignum Library*. Available online at http://gmplib.org.