A *correct*[*] exposition of the Schönhage-Strassen algorithm

Ramprasad Saptharishi

May 17, 2022

Abstract

This is a short exposition of the polynomial multiplication algorithm of Schönhage and Strassen [SS71] with almost all the details. I've have made the mistake (twice!) of assuming that a weaker recurrence also yields the main result of Schönhage and Strassen and this article is to write down the algorithm in as much details as possible, and also to show why weaker versions fall short.

1 Preliminaries

1.1 Principal roots of unity

Definition 1.1 (Principal roots of unity)**.** *An element ω of a commutative ring R is said to be a n-th* principal root of unity (n-PROU) *if*

- $\omega^n = 1$,
- *For all $0 < i < n-1$, we have $\sum_{j=0}^{n-1} \omega^{ij} = 0$.* ◊

Lemma 1.2 (Sufficient condition for n-PROU)**.** *Suppose n is a power of 2 and $\omega \in R$ such that $\omega^{n/2} + 1 = 0$. Then ω is an n-PROU.* □

1.2 Discrete Fourier Transform

Definition 1.3 (Discrete Fourier Transform)**.** *Suppose $\omega \in \mathbb{R}$ is an n-PROU. The* Discrete Fourier Transform (DFT) with respect to ω *of a polynomial $f(x) \in R[x]$ with $\deg(f) < n$ is the tuple of evaluations:*

$$\mathrm{DFT}_\omega(f) = \left(f(1), f(\omega), \ldots, f(\omega^{n-1})\right). \qquad \text{◊}$$

[*]hopefully

Observation 1.4 (Inverse DFT is a DFT too)**.** *If $\omega \in R$ is an n-PROU and $f \in R[x]$ with $\deg(f) < n$, then*

$$\mathrm{DFT}_{\omega^{-1}} \circ \mathrm{DFT}_{\omega}(f) = n \cdot f,$$

where we are interpreting the tuple $\mathrm{DFT}_{\omega}(f)$ as the coefficients of a polynomial. □

In other words, $\mathrm{DFT}_{\omega^{-1}}$ is the inverse of DFT_{ω}, up to a multiplication by n. When n is a power of two, there is a very efficient algorithm for computing the Discrete Fourier Transform.

Algorithm 1: FASTFOURIERTRANSFORM(f, ω)

Data: $f_0, f_1, \ldots, f_{n-1} \in R$, and $\omega \in R$ that is an n-PROU
Result: $(f(1), f(\omega), \ldots, f(\omega^{n-1}))$

if $n = 1$ **then**
  return f_0
$f_{\text{even}} \leftarrow (f_0, f_2, f_4, \ldots, f_{n-2})$
$f_{\text{odd}} \leftarrow (f_1, f_3, f_5, \ldots, f_{n-1})$
Pre-compute $1, \omega, \omega^2, \ldots, \omega^{n-1}$
$a_0, \ldots, a_{\frac{n}{2}-1} = \text{FASTFOURIERTRANSFORM}(f_{\text{even}}, \omega^2)$
$b_0, \ldots, b_{\frac{n}{2}-1} = \text{FASTFOURIERTRANSFORM}(f_{\text{even}}, \omega^2)$
for $i = 0, \ldots, \frac{n}{2} - 1$ **do**
  $\gamma_i \leftarrow a_i + \omega^i b_i$
  $\gamma_{i+\frac{n}{2}} \leftarrow a_i + \omega^{i+\frac{n}{2}} b_i$
return $\gamma_0, \ldots, \gamma_{n-1}$.

Lemma 1.5 (Running time of the Fast Fourier Transform)**.** *When n is a power of 2, Algorithm 1 on a polynomial $f \in R[x]$ of degree less than n and an n-PROU $\omega \in R$ performs*

- *$O(n \log n)$ additions of two arbitrary elements in R,*
- *$O(n \log n)$ multiplications of an arbitrary element of R with a power of ω.* □

1.3 Polynomial multiplication in rings that support FFT

Suppose $f(x), g(x) \in R[x]$ with $\deg(fg) < n$ where n is a power of 2. Suppose $\omega \in R$ is a n-PROU, and we can divide by 2 in R. Then, we can compute the product $f(x) \cdot g(x)$ very efficiently.

Lemma 1.6 (Polynomial multplication over rings supporting FFT)**.** *Suppose R is a ring that contains an n-PROU ω. Then two polynomials $f(x), g(x) \in R[x]$ with $\deg(fg) < n$ can be multiplied using Algorithm 2 by performing*

- *$O(n \log n)$ additions of two arbitrary elements in R,*
- *$O(n \log n)$ multiplications of an arbitrary element of R with a power of ω,*

Algorithm 2: PolyMultWithPROU(f, g, ω)

Data: Polynomials $f(x), g(x)$ given by coefficients from R with $\deg(f) + \deg(g) < n$, and an $\omega \in R$ that is a n-PROU
Result: Coefficients of $f(x) \cdot g(x)$

Interpret both f and g as polynomials of degree less than n (by padding with zeroes if necessary).
$a_0, \ldots, a_{n-1} = \text{FastFourierTransform}(f, \omega)$
$b_0, \ldots, b_{n-1} = \text{FastFourierTransform}(g, \omega)$
for $i = 0, \ldots, n-1$ **do**
    $c_i = a_i \cdot b_i$
$h_0, \ldots, h_{n-1} \leftarrow \frac{1}{n} \cdot \text{FastFourierTransform}([c_0, \ldots, c_{n-1}], \omega^{-1})$
return $h_0, \ldots, h_{n-1}$

- *n multiplications of two arbitrary elements in R,*
- *n divisons by n.*

2 The Schönhage-Strassen approach

Suppose we are working over a ring R that possibly does not contain a n-PROU. In that case, we are not in a position to directly use Algorithm 2. Schönhage and Strassen used a clever idea of artificially adding a suitable root of unity to the ring and working with that. The main theorem we are heading towards is the following.

Theorem 2.1 (Schönhage-Strassen [SS71])**.** *For an arbitrary ring R, multiplication of two polynomials $f, g \in R[x]$ with $\deg(fg) < n$ can be computed using $O(n \log n \log\log n)$ operations in R.*

Attempt 1

The first attempt is to work expand our ring of coefficients to $R' = \frac{R[t]}{t^{n/2}+1}$ as $t \in R'$ satisfies $t^{n/2} + 1 = 0$ and is hence an n-PROU by Lemma 1.2.

Thus, we can interpret $f(x), g(x) \in R[x]$ as in fact elements of $R'[x]$ and multiply them using Algorithm 2 with $\omega = t$. However, this would use roughly $n \log n$ additions of arbitrary elements of R', each of which costs $O(n)$ additions in R. Thus, the running time of the algorithm is at least as large as $n^2 \log n$, which is worse than the standard naïve multiplication.

Attempt 2

The next idea is to somehow reduce the degree of the polynomial so that we only need to add a smaller root of unity. This is done by rewriting f and g as *bivariate* polynomials instead. Suppose k and m are powers of two with $k \cdot m = n$ (we'll eventually choose those to about $\sqrt{n}$ each). Once

again, let us pad f and g and think of both $f(x), g(x)$ as polynomials of degree less than n.

$$\begin{aligned} f(x) &= f_0 + f_1 x + \cdots + f_{n_1-1} x^{n_1-1} \\ &= F_0 + F_1 x^m + F_2 x^{2m} + \cdots + F_{k_1-1} x^{(k_1-1)m} \\ \text{where } F_j(x) &= f_{jm} + f_{jm+1} x + \cdots + f_{jm+(m-1)} x^{m-1} \\ \therefore \text{If } \tilde{f}(x,y) &:= F_0 + F_1 y + \cdots + F_{k_1-1} y^{k_1-1} \\ \text{then } f(x) &= \tilde{f}(x, x^m) \end{aligned}$$

Thus, we can work with the bivariate polynomial $\tilde{f}(x,y)$ and $\tilde{g}(x,y)$. If we can multiply the polynomials $\tilde{f}$ and $\tilde{g}$ quickly, then we can substitute $y = x^m$ in $\tilde{f} \cdot \tilde{g}$ to get $f(x) \cdot g(x)$.

We can interpret $\tilde{f}$ and $\tilde{g}$ as elements in $R'[y]$ where $R' = R[x]$ and, since $mk = n = \deg(fg)$, we have set it up so that $\deg_y(\tilde{f}\tilde{g}) < k$. Thus, in order to multiply these two polynomials, we can use Algorithm 2 if somehow R' contained a k-PROU. Unfortunately, that may not be the case.

Here comes another cool idea of Schönhage and Strassen. Let R'' be the ring

$$R'' = \frac{R[x]}{x^{2m}+1}.$$

On the face of it, it seems like we are making a typo of putting $2m$ instead of k but bear with it for a moment. The key observation is the following.

Observation 2.2. *The product of $\tilde{f}$ and $\tilde{g}$ when intepreted as elements of $R'[y]$ is the same as the product when $\tilde{f}$ and $\tilde{g}$ are intepreted as elements of $R''[y]$.*

Proof. Note that $\deg_x(\tilde{f}\tilde{g}) < 2m$ as $\deg(F_i), \deg(G_j) < m$ by construction. Since the difference between R' and R'' is the relation $x^{2m}+1$, the product $\tilde{f} \cdot \tilde{g}$ remains unchanged when considered modulo $x^{2m}+1$. □

Therefore, we may as well think that the $\tilde{f}$ and $\tilde{g}$ are polynomials in $R''[y]$. Now, fortunately, $x \in R''$ is a $4m$-PROU due to Lemma 1.2. If we can arrange for the parameters such that $4m \geq k$, then this is sufficient to multiply $\tilde{f}$ and $\tilde{g}$ using Algorithm 2. Let us make it so. If $n = 2^\ell$, let $m = 2^{\lfloor \ell/2 \rfloor}$ and $k = 2^{\lceil \ell/2 \rceil}$. This ensures that $2m \geq k$ and both are powers of 2 with their product being n.

Hence, we can compute the product $\tilde{h}(x,y)$ using Algorithm 2 with the k-PROU ω suitably chosen ($\omega = x^2$ if ℓ is odd (and hence $2m = k$), and $\omega = x^4$ if ℓ is even (and hence $m = k$)).

Thus, we can compute the product $\tilde{h} = \tilde{f} \cdot \tilde{g}$ using

(a) $O(k \log k)$ additions of elements in R'',

(b) $O(k \log k)$ multiplications of elements in R'' with powers of x,

(c) k multiplications of arbitrary elements in R''.

Note that each addition of elements in R'' corresponds to m additions over R. Also, multiplications of elements in R'' by powers of x just amounts to "shifts" and sign-changes and hence once again correspond to $O(m)$ operations.

As for the third item, these are elements of R'', which correspond to polynomials of degree less than $2m$ that need to be multiplied (and hence its product has degree less than $4m$) and then the relation "$x^{2m}+1$" performed. These multiplications can be called recursively! (And performing the "modulo $x^{2m}+1$" is inexpensive as it is once again involves some shifts and additions in R). Thus, overall, we get the following recurrence. For simplicity, let us assume $k = m = \sqrt{n}$.

$$\begin{aligned} T(n) &= O(mk \log k) + k \cdot T(4m) \\ &= O(n \log n) + \sqrt{n} \cdot T(4\sqrt{n}). \end{aligned}$$

Solving this recurrence should give us $T(n) = O(n \log n \log\log n)$, right? No! It doesn't![1] Solving this recurrence actually yields $T(n) = O(n(\log n)^{\alpha})$, and not $T(n) = O(n \log n \log\log n)$ as was claimed in Theorem 2.1. (Roughly speaking, the next expansion yields and additional yields the factor of 4 but the difference between $\log n$ and $\log \sqrt{n}$ will only absorb a factor of 2.) Hence this doesn't yield what we want.

Where was the loss, and how do we improve?

Had the recurrence instead been $T(n) = O(mk \log k) + k \cdot T(2m)$, then this recurrence would have resulted in $T(n) = O(n \log n \log\log n)$. This extra factor of 2 appears to just come from the fact that we want to compute a product of two coefficients in $R'' = \frac{R[x]}{x^{2m}+1}$, and we intepreted had to "pad" them to degree $4m$ each, compute the product (which is a polynomial of degree $4m$), and then go modulo $x^{2m}+1$. Can we avoid this artefact of first increasing the degree and then decreasing it again?

Let's turn the whole thing around. Why don't we start with the task of computing $f(x) \cdot g(x) \bmod x^n + 1$?

3 Convolutions

Definition 3.1 (Convolution)**.** *Suppose $f = [f_0, \ldots, f_{n-1}]$ and $g = [g_0, \ldots, g_{n-1}]$ are two "vectors" with each $f_i, g_i \in R$. The convolution of f and g, denoted by $f * g$, is a "vector" $[h_0, \ldots, h_{2n-1}]$ such that*

$$h_\ell = \sum_{i=0}^{n-1} f_i \cdot g_{\ell-i} \quad \text{for all } \ell = 0, \ldots, 2n-1. \qquad \Diamond$$

The above is just a fancy way of saying that the "vector" h, when interpreted as the coefficients of a polynomial is just the product of the polynomials f and g.

[1] I have made this mistake at least twice, and there appears to be multiple notes on the web that also makes the same mistake.

A modification of the above convolution definition, that is *incredibly* useful (as we shall soon see) is the notion of *wrapped convolutions*.

Definition 3.2 (Wrapped convolutions)**.** *Given two "vectors" $f = [f_0, \ldots, f_{n-1}]$ and $g = [g_0, \ldots, g_{n-1}]$ with each $f_i, g_j \in R$, the* positively wrapped convolution *of f and g is given by the "vector" $h^+ = [h_0^+, \ldots, h_{n-1}^+]$ defined as*

$$h_\ell^+ = \sum_{i=0}^{n} (f_i \cdot g_{\ell-i} + f_i \cdot g_{n+\ell-i}) \quad \text{for all } \ell = 0, \ldots, 2n-1. \qquad \Diamond$$

Similarly, the negatively wrapped convolution (PWC) *of f and g is given by the "vector" $h^- = [h_0^-, \ldots, h_{n-1}^-]$ defined as*

$$h_\ell^- = \sum_{i=0}^{n} (f_i \cdot g_{\ell-i} - f_i \cdot g_{n+\ell-i}) \quad \text{for all } \ell = 0, \ldots, n-1.$$

In other words, both the above wrapped convolutions is a way of "folding" the standard convolution $h = f * g$ to half its length, with the positively wrapped convolution setting $h_\ell^+ = h_\ell + h_{n+\ell}$ and the negatively wrapped convolution setting $h_\ell^- = h_\ell - h_{n+\ell}$.

Observation 3.3. *Suppose f and g are intepreted as polynomials of degree less than n, then the polynomials h^+ and h^- corresponding the the positively and negatively wrapped convolutions respectively are given by*

$$h^+(x) = f(x) \cdot g(x) \bmod x^n - 1,$$
$$h^-(x) = f(x) \cdot g(x) \bmod x^n + 1.$$

We'll now see why both the positively and negatively wrapped convolutions can be computed quickly using the Fast Fourier Transform Algorithm 1.

3.1 Computing positively wrapped convolutions using FFT

The Fourier Transform can actually be used to compute the positively wrapped convolutions (PWC) quite quickly.

This almost seems identical to the Algorithm 2 and it is, except for the fact that in that case we would have used a $2n$-th PROU and not an n-th PROU as used above. Roughly speaking, the above algorithm ensures that the output polynomial $h^+(x)$, of degree less than n, satisfies

$$\begin{aligned}
& h(\omega^i) = f(\omega^i) \cdot g(\omega^i) && \text{for } i = 1, \ldots, n-1 \\
\implies\ & h(x) - f(x)g(x) = 0 \bmod (x - \omega^i) && \text{for } i = 1, \ldots, n-1 \\
\implies\ & h(x) - f(x)g(x) = 0 \bmod \prod_{i=0}^{n} (x - \omega^i) && \text{(some version of CRT)} \\
& = 0 \bmod x^n - 1.
\end{aligned}$$

Algorithm 3: PWC-WITH-PROU(f, g, ω)

Data: "Vectors" $f = [f_0, \ldots, f_{n-1}], g = [g_0, \ldots, g_{n-1}]$ with entries in R and an $\omega \in R$ that is a n-PROU

Result: The positively wrapped convolution $h^+ = [h_0, \ldots, h_{n-1}]$ of f and g.

$a_0, \ldots, a_{n-1} = \text{FASTFOURIERTRANSFORM}(f, \omega)$

$b_0, \ldots, b_{n-1} = \text{FASTFOURIERTRANSFORM}(g, \omega)$

for $i = 0, \ldots, n-1$ **do**

  $c_i = a_i \cdot b_i$

$h_0, \ldots, h_{n-1} \leftarrow \frac{1}{n} \cdot \text{FASTFOURIERTRANSFORM}([c_0, \ldots, c_{n-1}], \omega^{-1})$

return $h_0, \ldots, h_{n-1}$

Lemma 3.4 (PWC over rings supporting FFT)**.** *Suppose R is a ring that contains an n-PROU ω. Then the PWC of two polynomials $f(x), g(x) \in R[x]$ with $\deg(f), \deg(g) < n$ can be computed using Algorithm 3 by performing*

- *$O(n \log n)$ additions of two arbitrary elements in R,*
- *$O(n \log n)$ multiplications of an arbitrary element of R with a power of ω,*
- *n multiplications of two arbitrary elements in R,*
- *n divisons by n.*

The key difference between the above lemma and Lemma 1.6 is that when $\deg(f), \deg(g) < n$, we only have that $\deg(fg) < 2n - 1$ and hence would need about $2n$ multiplications of two arbitrary elements in R whereas the above algorithm shows that n multiplications are enough to compute the PWC.

3.2 Computing negatively wrapped convolution

Negatively wrapped convolutions (NWC) of two vectors can also be computed efficiently using FFT, except we now need a $2n$-PROU in the ring. The main idea is as follows. Suppose n is a power of 2 and ω satisfies $\omega^n + 1 = 0$ and hence is a $2n$-PROU (Lemma 1.2). Then, observe that

$$-(x^n + 1) = -x^n - 1 = (\omega \cdot x)^n - 1 = (\omega^{-1} \cdot x)^n - 1$$

Therefore, if $\tilde{f}(x) = f(\omega \cdot x)$ and $\tilde{g}(x) = g(\omega \cdot x)$, and $\tilde{h}$ is the PWC of $\tilde{f}$ and $\tilde{g}$, we have

$$\begin{aligned}
\tilde{h}(x) &= \tilde{f}(x)\tilde{g}(x) \bmod x^n - 1 \\
\implies \tilde{h}(x) &= \tilde{f}(x)\tilde{g}(x) + q(x) \cdot (x^n - 1) \\
&= f(\omega \cdot x) \cdot g(\omega \cdot x) + q(x) \cdot (x^n - 1). \\
\therefore h(x) := \tilde{h}(\omega^{-1} \cdot x) &= f(x) \cdot g(x) + q(\omega^{-1} \cdot x) \cdot ((\omega^{-1} \cdot x)^n - 1) \\
&= f(x) \cdot g(x) + q'(x) \cdot (x^n + 1) \\
\implies h(x) &= f(x)g(x) \bmod x^n + 1.
\end{aligned}$$

This immediately yields an algorithm.

Algorithm 4: NWC-WITH-PROU(f, g, ω)

Data: "Vectors" $f = [f_0, \ldots, f_{n-1}], g = [g_0, \ldots, g_{n-1}]$ with entries in R and an $\omega \in R$ that is a $2n$-PROU

Result: The negatively wrapped convolution $h^- = [h_0, \ldots, h_{n-1}]$ of f and g.

1 Compute the coefficients of $\tilde{f}(x) := f(\omega \cdot x)$ and $\tilde{g}(x) := g(\omega \cdot x)$.
2 $\tilde{h}(x) \leftarrow$ PWC-WITH-PROU(f, g, ω^2)
3 Compute the coefficients $[h_0, \ldots, h_{n-1}]$ of $h^-(x) := \tilde{h}(\omega^{-1} \cdot x)$.
4 **return** $h_0, \ldots, h_{n-1}$

Lemma 3.5 (NWC over rings supporting FFT)**.** *Suppose R is a ring that contains a $2n$-PROU ω. Then the NWC of two polynomials $f(x), g(x) \in R[x]$ with $\deg(f), \deg(g) < n$ can be computed using Algorithm 4 by performing*

- $O(n \log n)$ *additions of two arbitrary elements in R,*
- $O(n \log n)$ *multiplications of an arbitrary element of R with a power of ω,*
- n *multiplications of two arbitrary elements in R,*
- n *divisons by n.*

Once again, it is crucial that we only need n multiplications of arbitrary elements in R, and not $2n$.

3.3 NWC in rings without a PROU

We are now going to follow the same strategy that we tried earlier for polynomial multiplication, but this time instead for computing NWC in rings without a $2n$-PROU.

Suppose we are given $f, g \in R[x]$ with $\deg(f), \deg(g) < n$ and we wish to compute the NWC $h(x)$ of f and g, which we know satisfies

$$h(x) = f(x)g(x) \bmod x^n + 1.$$

Once again, if $n = 2^\ell$, let $m = 2^{\lfloor \ell/2 \rfloor}$ and $k = 2^{\lceil \ell/2 \rceil}$ and consider the following bivariate polynomials

$$\begin{aligned}
\tilde{f}(x,y) &:= F_0 + F_1 y + \cdots + F_{k-1} y^{k-1} \\
\text{where } F_j(x) &= f_{jm} + f_{jm+1} x + \cdots + f_{jm+(m-1)} x^{m-1} \\
\text{so that } f(x) &= \tilde{f}(x, x^m). \\
\text{Similarly, } \quad \tilde{g}(x,y) &:= G_0 + G_1 y + \cdots + G_{k-1} y^{k-1} \\
\text{where } G_j(x) &= g_{jm} + g_{jm+1} x + \cdots + g_{jm+(m-1)} x^{m-1}.
\end{aligned}$$

Note that $\deg_y \tilde{f}, \deg_y \tilde{g} < k$. Suppose we could compute the negative convolution $\tilde{h}^-(x,y)$ of $\tilde{f}(x,y)$ and $\tilde{g}(x,y)$. Then, note that

$$\begin{aligned}
& \tilde{h}^-(x,y) = \tilde{f}(x,y) \cdot \tilde{g}(x,y) \bmod y^k + 1 \\
\implies & \tilde{h}^-(x,y) - \tilde{f}(x,y) \cdot \tilde{g}(x,y) = q(x,y) \cdot (y^k + 1) \\
\implies & \tilde{h}^-(x,x^m) - \tilde{f}(x,x^m) \cdot \tilde{g}(x,x^m) = q(x,x^m) \cdot (x^{km} + 1) \\
\implies & h(x) := \tilde{h}^-(x,x^m) = f(x) \cdot g(x) \bmod x^n + 1.
\end{aligned}$$

Therefore, in order to compute the NWC of f and g, it suffices to compute the NWC of $\tilde{f}$ and $\tilde{g}$. Since $\deg_x \tilde{f}, \deg_x \tilde{g} < m$, we have that $\deg_x(\tilde{f} \cdot \tilde{g}) < 2m$ and hence we can safely interpret the polynomials $\tilde{f}$ and $\tilde{g}$ as elements of $R'[y]$ where

$$R'' = \frac{R[x]}{x^{2m} + 1}.$$

Now, since the ring R'' now contains a $4m$-PROU, namely x, and since $2m > k$ by our choice of parameters, we certainly have a $2k$-PROU $\omega \in R''$ (if ℓ is even, then $m = k$ and $\omega = x^2$; if ℓ is odd, then $2m = k$ and hence $\omega = x$). Thus, we can certainly use Algorithm 4 to compute the k-length NWC $\tilde{h}^-$ of $\tilde{f}$ and $\tilde{g}$. Once we have $\tilde{h}^-(x,y)$, we can compute $h(x) = \tilde{h}^-(x,x^m)$ which is the desired output.

By Lemma 3.5, we can compute the m-length NWC of $\tilde{f}$ and $\tilde{g}$ using $O(m \log m)$ additions of arbitrary elements in R'', and $O(m \log m)$ multiplications of elements of R' with powers of x, and m multiplications of arbitrary elements of R''. But each multiplication in R'' is also yet another NWC computation of length $2m$ and hence these will be recursive calls. Thus, the running time can be expressed as

$$\begin{aligned}
T(n) &= O(k \log k) \times \text{ additions in } R'' \\
&\quad + O(k \log k) \times \text{ multiplications of the form } x^i \times R'' \\
&\quad + k \times \text{NWC computations of length-}2m \\
\implies T(n) &= O(n \log n) + k \cdot T(2m)
\end{aligned}$$

and that's the better recurrence we were looking for! Solving this recurrence (Lemma A.1) yields $T(n) = O(n \log n \log\log n)$.

Theorem 3.6 (Fast negatively wrapped convolutions [SS71])**.** *Given two polynomials $f(x), g(x) \in R[x]$ with $\deg f, \deg g < n = 2^\ell$, and if we can divide by 2 in R, then we can compute the length-n negatively wrapped convolution of f and g in deterministic $O(n \log n \log\log n)$.*

Algorithm 5: FAST-NWC(f, g)

Data: "Vectors" $f = [f_0, \ldots, f_{n-1}], g = [g_0, \ldots, g_{n-1}]$ with entries in R, with $n = 2^\ell$.
Result: The negatively wrapped convolution $h^- = [h_0, \ldots, h_{n-1}]$ of f and g.

1 $k \leftarrow 2^{\lceil \ell/2 \rceil}$ and $m \leftarrow 2^{\lfloor \ell/2 \rfloor}$.
2 Express $f(x)$ and $g(x)$ as bivariates $\tilde{f}(x, y)$ and $\tilde{g}(x, y)$ satisfying $\deg_x \tilde{f}, \deg_x \tilde{g} < m$ and $\deg_y \tilde{f}, \deg_y \tilde{g} < k$, with $f(x) = \tilde{f}(x, x^m)$ and $g(x) = \tilde{g}(x, x^m)$.
3 Interpret $\tilde{f}$ and $\tilde{g}$ as elements of $R'[y]$ where $R' = \frac{R[x]}{x^{2m}+1}$.
4 **if** ℓ *is even* **then**
5 $\quad \omega \leftarrow x^2$.
6 **else**
7 $\quad \omega \leftarrow x$.
8 $\tilde{h}^-(x, y) \leftarrow$ NWC-WITH-PROU$(\tilde{f}, \tilde{g}, \omega)$
9 Compute the coefficients $[h_0, \ldots, h_{n-1}]$ of $\tilde{h}(x, x^m)$
10 **return** $h_0, \ldots, h_{n-1}$.

Corollary 3.7 (Theorem 2.1)**.** *Suppose R is a ring in which we can divide by 2. Then, we can compute the product of two polynomials $f(x), g(x) \in R[x]$ in deterministic time $O(n \log n \log\log n)$, where n is an upper bound on their degrees.*

Proof. Just think of f and g as polynomials of degree less than $2n$ by padding, and their negatively wrapped convolution $h(x)$, which can be computed using Algorithm 5 is the product fg. □

References

[SS71] A Schönhage and V Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

A Solving the recurrence

Considering I've made a mistake earlier (twice!) in assuming that the wrong recurrence yields $O(n \log n \log\log n)$, it is perhaps prudent to prove it completely.

Lemma A.1. *The recurrence (defined for powers of 2) given by*

$$T(n) = O(n \log n) + k \cdot T(2m) \quad , \quad T(1) = 1$$

where $n = 2^\ell, m = 2^{\lfloor \ell/2 \rfloor}$ *and* $k = 2^{\lceil \ell/2 \rceil}$, *solves to* $T(n) = O(n \log n \log\log n)$.

Proof. Let $S(n) = T(n)/n$. Thus, the above recurrence can be written as

$$\begin{aligned} S(n) &= O(\log n) + 2 \cdot S(2m) \\ &\leq c \log n + 2 \cdot S(2\sqrt{n}) \end{aligned}$$

for some absolute constant c. Therefore,

$$\begin{aligned} S(n) &\leq c \log n + 2(c \log(2\sqrt{n}) + 2 \cdot S(2^{1+\frac{1}{2}} n^{1/4})) \\ &\leq c \log n + c \log(2^2 n) + 2^2 \cdot S(2^{1+\frac{1}{2}} n^{1/2^2}) \\ &\leq c \log(2^{2^1} n) + c \log(2^{2^2} n) + 2^2 \cdot S(2^{1+\frac{1}{2}} n^{1/2^2}) \\ &\leq c \log(2^{2^1} n) + c \log(2^{2^2} n) + c \log(2^{2^2(1+\frac{1}{2})} n) + 2^3 \cdot S(2^{1+\frac{1}{2}+\frac{1}{4}} n^{1/2^3}) \\ &\leq c \log(2^{2^1} n) + c \log(2^{2^2} n) + c \log(2^{2^3} n) + 2^3 \cdot S(2^{1+\frac{1}{2}+\frac{1}{4}} n^{1/2^3}) \\ &\vdots \\ &\leq c \sum_{i=1}^{\log\log n} \log(2^{2^i} n) = c \sum_{i=1}^{\log\log n} \left(\log n + 2^i \right) \\ &= c \log n \log\log n + c(1 + 2 + 2^2 + \cdots + \log n) \\ &= c \log n \log\log n + 2c \log n = O(\log n \log\log n) \end{aligned}$$

$$\therefore T(n) = O(n \log n \log\log n).$$ □