Schönhage-Strassen Algorithm

Yehui Feng

1 Introduction

Efficient multiplication of large integers is a critical problem in computational mathematics with wide-ranging applications in fields such as cryptography, numerical analysis, and computer science. Over the years, numerous algorithms have been developed to tackle this challenge, each improving on the efficiency of its predecessors. A significant milestone in this evolution was the introduction of the Schönhage–Strassen algorithm by Arnold Schönhage and Volker Strassen in 1971 [1]. This algorithm revolutionized integer multiplication by employing the fast Fourier transform (FFT) in conjunction with modular arithmetic, achieving a bit complexity of $O(n \cdot \log n \cdot \log\log n)$. This marked a notable advancement over previous methods, such as Karatsuba and Toom–Cook multiplication [4], and established it as the fastest known method for large integers for decades.

The Schönhage–Strassen algorithm's impact is particularly evident when handling very large numbers, where it outperforms earlier algorithms for inputs on the order of thousands to hundreds of thousands of digits. Despite the introduction of new algorithms in recent years—such as Martin Fürer's improved method in 2007 [3] and the $O(n \log n)$ algorithm by David Harvey and Joris van der Hoeven in 2019 [2]—the Schönhage–Strassen algorithm remains a cornerstone in practical applications due to its robust performance and well-understood characteristics.

This paper provides an in-depth exploration of the Schönhage–Strassen algorithm, detailing its theoretical underpinnings and practical implementation. By focusing on its strengths and the reasons behind its historical significance, we aim to underscore the continued relevance of this algorithm in the field of integer multiplication.

2 Preliminary

This section introduces the fundamental concepts and tools necessary for understanding the Schönhage-Strassen algorithm. In particular, the concept of Principal Roots of Unity (PROU) is central to many fast algorithms for polynomial multiplication, including the Discrete Fourier Transform (DFT) and the Fast Fourier Transform (FFT). These techniques are prerequisites for the advanced ideas discussed later, such as handling polynomial multiplication in rings that do not naturally support an n-PROU.

2.1 Principal Roots of Unity (PROU)

Definition 2.1 (Principal Roots of Unity)**.** Let R be a commutative ring. An element $\omega \in R$ is said to be an n-th principal root of unity (n-PROU) if it satisfies two fundamental properties:

- $\omega^n = 1$, establishing that ω is a root of the polynomial $x^n - 1$.
- For every integer $0 < i < n$, the sum $\sum_{j=0}^{n-1} \omega^{ij} = 0$.

Lemma 2.2. *If n is a power of 2 and $\omega \in R$ such that $\omega^{n/2} = -1$, then ω is indeed an n-PROU.*

Proof. Given that $\omega^{n/2} = -1$, we have $\omega^n = (\omega^{n/2})^2 = 1$, ensuring that ω is a root of $x^n - 1$. To verify the vanishing sum condition, consider the geometric series $1 + \omega + \omega^2 + \cdots + \omega^{n-1}$. Since ω is a root of $x^n - 1$, the series represents the sum of all n-th roots of unity, which is known to be zero. Thus, ω satisfies the conditions to be an n-PROU. □

2.2 Discrete Fourier Transform (DFT)

Definition 2.3 (Discrete Fourier Transform (DFT))**.** Consider a polynomial $f(x) \in R[x]$ with degree $\deg(f) < n$, where $\omega \in R$ is an n-th principal root of unity (n-PROU). The Discrete Fourier Transform (DFT) of $f(x)$ with respect to ω is defined as the tuple:

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

This tuple represents the evaluation of the polynomial at each of the n-th roots of unity, encapsulating the frequency characteristics of $f(x)$ in the spectral domain.

The beauty of the DFT lies not only in its ability to transform a polynomial into its frequency components but also in the existence of an inverse transformation that allows us to recover the original polynomial from its spectral representation. The next lemma establishes this critical connection.

Lemma 2.4 (Inverse DFT)**.** *Let $\omega \in R$ be an n-PROU, and consider a polynomial $f(x)$ of degree less than n. The inverse Discrete Fourier Transform (DFT) is given by:*

$$f(x) = \frac{1}{n} \sum_{j=0}^{n-1} DFT_{\omega^{-1}}(f) \cdot \omega^{-jx}.$$

This formula reconstructs $f(x)$ from its frequency components, completing the circle of transformation between the time domain and the frequency domain.

Proof. The proof relies on the orthogonality of the roots of unity and the properties of geometric series. Specifically, the orthogonality ensures that the sum of cross-terms vanishes, leaving only the original polynomial when the inverse transform is applied. This establishes the DFT and its inverse as true inverses, up to a normalization factor of n. □

2.3 Fast Fourier Transform (FFT)

The power of the Discrete Fourier Transform is magnified when we consider the case where n is a power of two. In this scenario, we can leverage a highly efficient algorithm, known as the Fast Fourier Transform (FFT) as showned in Algorithm 1, to compute the DFT in $O(n \log n)$ time—a significant improvement over the naive $O(n^2)$ approach. The FFT exploits the recursive structure of the DFT, dramatically reducing the computational burden.

Lemma 2.5 (FFT Time Complexity)**.** *Let n be a power of 2, $f(x)$ a polynomial over a commutative ring R with degree less than n, and let $\omega \in R$ be an n-th principal root of unity (n-PROU). The Fast Fourier Transform (FFT) algorithm computes the Discrete Fourier Transform (DFT) of $f(x)$ with respect to ω using:*

- *$O(n \log n)$ additions of arbitrary elements in R,*
- *$O(n \log n)$ multiplications by powers of ω.*

Proof. The FFT algorithm operates by recursively dividing the problem of size n into two subproblems of size $n/2$, and combining their results. This recursive process leads to the following recurrence relation for the time complexity $T(n)$:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

where:

Algorithm 1 Fast Fourier Transform (FFT)

```
Require: A sequence of coefficients $f_0, f_1, \ldots, f_{n-1} \in R$ and an $n$-PROU $\omega \in R$
Ensure: The DFT values $(f(1), f(\omega), \ldots, f(\omega^{n-1}))$
 1: if $n = 1$ then
 2:     return $f_0$
 3: end if
 4: Separate the sequence into even and odd-indexed coefficients:
 5: $f_{\text{even}} \leftarrow (f_0, f_2, \ldots, f_{n-2})$
 6: $f_{\text{odd}} \leftarrow (f_1, f_3, \ldots, f_{n-1})$
 7: Compute the DFT recursively on the even and odd parts:
 8: $(a_0, \ldots, a_{n/2-1}) \leftarrow \text{FFT}(f_{\text{even}}, \omega^2)$
 9: $(b_0, \ldots, b_{n/2-1}) \leftarrow \text{FFT}(f_{\text{odd}}, \omega^2)$
10: for $i = 0, \ldots, n/2 - 1$ do
11:     Compute the combined DFT values:
12:     $\gamma_i \leftarrow a_i + \omega^i b_i$
13:     $\gamma_{i+n/2} \leftarrow a_i - \omega^i b_i$
14: end for
15: return $\gamma_0, \ldots, \gamma_{n-1}$
```

- $2T\left(\frac{n}{2}\right)$ accounts for the time to solve the two subproblems of size $n/2$,
- $O(n)$ represents the time to combine the results of these subproblems, involving $n/2$ additions and $n/2$ multiplications by powers of ω.

To solve this recurrence, we use the Master Theorem for divide-and-conquer recurrences, which states:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d),$$

where $a = 2$, $b = 2$, and $d = 1$. According to the Master Theorem:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d, \\ O(n^d) & \text{if } a < b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

In our case, $a = b^d$ (i.e., $2 = 2^1$), so:

$$T(n) = O(n^1 \log n) = O(n \log n).$$

For operations:

- Additions: Each recursion level requires $O(n)$ additions. With $\log_2 n$ levels, the total number of additions is $O(n \log n)$.
- Multiplications: Each recursion level requires $O(n)$ multiplications by powers of ω. Thus, the total number of multiplications is also $O(n \log n)$.

Thus, the FFT algorithm performs $O(n \log n)$ additions and multiplications, completing the proof. □

2.4 Polynomial Multiplication in Rings Supporting FFT

Consider the multiplication of two polynomials $f(x)$ and $g(x)$ with degree less than n in a ring $R[x]$. Suppose $\omega \in R$ is an n-th primitive root of unity (PROU), and R allows division by 2. The multiplication $f(x) \cdot g(x)$ can be executed with remarkable efficiency, thanks to the FFT.

Lemma 2.6 (Polynomial multiplication over rings supporting FFT)**.** *Let R be a ring that contains an n-PROU ω. Then, two polynomials $f(x)$ and $g(x)$ in $R[x]$ with $\deg(fg) < n$ can be multiplied using Algorithm 2, which involves:*

- *$O(n \log n)$ 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 divisions by n.*

Algorithm 2 Polymult with PROU(f, g, ω)

```
Require: Polynomials $f(x), g(x)$ given by coefficients from $R$ with $\deg(f) + \deg(g) < n$, and an $\omega \in R$ that
    is an $n$-PROU.
Ensure: Coefficients of $f(x) \cdot g(x)$.
1: Interpret both $f(x)$ and $g(x)$ as polynomials of degree less than $n$ (by padding with zeros if necessary).
2: $a_0, a_1, \dots, a_{n-1} \leftarrow \text{FFT}(f, \omega)$
3: $b_0, b_1, \dots, b_{n-1} \leftarrow \text{FFT}(g, \omega)$
4: for $i = 0$ to $n - 1$ do
5:     $c_i \leftarrow a_i \cdot b_i$
6: end for
7: $h_0, h_1, \dots, h_{n-1} \leftarrow \frac{1}{n} \cdot \text{FFT}([c_0, c_1, \dots, c_{n-1}], \omega^{-1})$
8: return $h_0, h_1, \dots, h_{n-1}$
```

Proof. To establish the time complexity of multiplying two polynomials $f(x)$ and $g(x)$ using the algorithm described in Algorithm 2, we analyze each step of the algorithm:

1. Computing $\text{FFT}(f, \omega)$ and $\text{FFT}(g, \omega)$ each require $O(n \log n)$ operations, as established in Algorithm 1. This involves $O(n \log n)$ additions and $O(n \log n)$ multiplications of elements by powers of ω.

2. The loop in line 5 computes n pointwise multiplications $c_i \leftarrow a_i \cdot b_i$, which requires $O(n)$ multiplications of elements in R.

3. The final step involves computing the inverse FFT of the n values $[c_0, c_1, \dots, c_{n-1}]$ using $\text{FFT}([c_0, c_1, \dots, c_{n-1}], \omega^{-1})$. This step also requires $O(n \log n)$ operations, including $O(n \log n)$ additions and $O(n \log n)$ multiplications by powers of ω^{-1}.

4. After the inverse FFT, each coefficient h_i is computed by dividing by n, which requires $O(n)$ divisions.

Combining these results, we see that the total time complexity is dominated by the FFT computations, which require $O(n \log n)$ additions and multiplications by powers of ω, and the inverse FFT computation. The additional operations (pointwise multiplication and divisions) are linear in n, $O(n)$. Thus, Algorithm 2 is efficient, with the polynomial multiplication being achieved in $O(n \log n)$ time complexity. □

3 The Schönhage-Strassen Method

Imagine you're in a ring R that, alas, lacks an n-primitive root of unity (n-PROU). Alas, this means we can't directly employ Algorithm 2. Fear not, Schönhage and Strassen came to the rescue with an ingenious workaround. They devised a clever technique of "importing" an appropriate root of unity into the ring and working with this augmented setup. The grand theorem we aim to prove is:

Theorem 3.1 (Schönhage-Strassen). *Given any ring R, the product of two polynomials f and g in $R[x]$ with $\deg(fg) < n$ can be computed using $O(n \log n \log\log n)$ operations in R.*

To achieve this impressive bound, we need to navigate the absence of an n-PROU in R with some finesse. Below, we detail the iterative improvements that lead to the elegant solution.

3.1 First Attempt: Expanding the Ring

Our initial strategy involves expanding our ring of coefficients to $R' = \frac{R[t]}{t^{n/2}+1}$, where t satisfies $t^{n/2} + 1 = 0$, and thus serves as an n-PROU by Lemma 2.2. This lets us interpret $f(x)$ and $g(x)$ as polynomials in $R'[x]$ and apply Algorithm 2 with $\omega = t$.

However, this approach introduces complications: each addition in R' translates to $O(n)$ additions in R, leading to a time complexity of at least $O(n^2 \log n)$, which is less efficient than naive polynomial multiplication.

3.2 Second Attempt: Bivariate Polynomials and Modular Arithmetic

Next, we try a different tactic—rewriting f and g as bivariate polynomials. Suppose k and m are powers of two such that $k \cdot m = n$ (eventually choosing k and m to be around $\sqrt{n}$ each). We pad f and g and view them as polynomials of degree less than n:

$$f(x) = f_0 + f_1 x + \cdots + f_{n_1-1}x^{n_1-1} = F_0 + F_1 x^m + \cdots + F_{k_1-1}x^{(k_1-1)m}$$

where $F_j(x) = f_{jm} + f_{jm+1}x + \cdots + f_{jm+(m-1)}x^{m-1}$.

Hence, $f(x)$ can be expressed as:

$$f(x) = \tilde{f}(x, x^m),$$

where

$$\tilde{f}(x, y) := F_0 + F_1 y + \cdots + F_{k_1-1}y^{k_1-1}.$$

We then work with the bivariate polynomials $\tilde{f}(x, y)$ and $\tilde{g}(x, y)$. If we can efficiently multiply these polynomials, substituting $y = x^m$ yields $f(x) \cdot g(x)$.

We can treat $\tilde{f}$ and $\tilde{g}$ as polynomials in $R''[y]$, where $R'' = R[x]$. Since $mk = n$, the degree in y of $\tilde{f}\tilde{g}$ is less than k. Thus, we can use Algorithm 2 if R'' contains a k-PROU. Unfortunately, this isn't guaranteed.

Here's where Schönhage and Strassen provide another stroke of brilliance. Define:

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

It may seem we've made a slip by using $2m$ instead of k, but bear with us. The key insight is this:

Lemma 3.2. *When interpreted as elements of $R''[y]$, the product of $\tilde{f}$ and $\tilde{g}$ is the same as when viewed in $R'''[y]$.*

Proof. Since $\deg_x(\tilde{f}\tilde{g}) < 2m$, the polynomial $\tilde{f}\tilde{g}$ remains unchanged when considered modulo $x^{2m} + 1$, as R'' and R''' differ only by this relation. $\square$

Thus, we can treat $\tilde{f}$ and $\tilde{g}$ as polynomials in $R'''[y]$. Conveniently, x in R''' is a $4m$-PROU due to Lemma 2.2. By choosing parameters such that $4m \geq k$, we can use Algorithm 2. For $n = 2^\ell$, set $m = 2^{\lfloor \ell/2 \rfloor}$ and $k = 2^{\lceil \ell/2 \rceil}$. This choice ensures $2m \geq k$ and fits our needs perfectly.

Thus, we compute $\tilde{h}(x, y) = \tilde{f} \cdot \tilde{g}$ using Algorithm 2, with the k-PROU ω appropriately chosen. The computational steps involve:

1. $O(k \log k)$ additions in R''',

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

Each addition in R''' corresponds to m additions in R. Similarly, multiplications by powers of x in R''' equate to shifts and sign changes in R, and are therefore $O(m)$. Multiplying polynomials modulo $x^{2m} + 1$ requires recursively handling polynomials of degree less than $2m$, yielding:

$$T(n) = O(mk \log k) + k \cdot T(2m).$$

With $k = m = \sqrt{n}$, solving this recurrence relation gives us:

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

Thus, we achieve the desired time complexity, validating the theorem.

4 Convolutions

With the groundwork of the Schönhage-Strassen algorithm laid, we can now explore one of its pivotal components: the efficient handling of convolutions. As we have seen, the Schönhage-Strassen algorithm leverages fast integer multiplication by reducing the problem to polynomial multiplication, which in turn hinges on convolutions. Understanding these convolutions, particularly their wrapped variants, is key to grasping the algorithm's efficiency and power.

Definition 4.1 (Convolution)**.** Consider two vectors $f = [f_0, \ldots, f_{n-1}]$ and $g = [g_0, \ldots, g_{n-1}]$, where each entry f_i and g_i is a real number. The *convolution* of f and g, denoted $f * g$, results in a vector $h = [h_0, \ldots, h_{2n-1}]$ defined by:

$$h_\ell = \sum_{i=0}^{n-1} f_i \cdot g_{\ell-i}$$

for every index $\ell = 0, \ldots, 2n-1$. This essentially says that if you view h as the coefficients of a polynomial, it represents the product of the polynomials $f(x)$ and $g(x)$.

4.1 Wrapped Convolutions

Let's make things a bit more interesting with *wrapped convolutions*, which essentially "wrap" the standard convolution to fit into a smaller space. There are two varieties: positively wrapped convolution (PWC) and negatively wrapped convolution (NWC).

Definition 4.2 (Positively Wrapped Convolution (PWC))**.** Given vectors $f = [f_0, \ldots, f_{n-1}]$ and $g = [g_0, \ldots, g_{n-1}]$ with real entries, the *positively wrapped convolution* of f and g produces a vector $h^+ = [h_0^+, \ldots, h_{n-1}^+]$ defined as:

$$h_\ell^+ = \sum_{i=0}^{n-1} (f_i \cdot g_{\ell-i} + f_i \cdot g_{n+\ell-i})$$

for $\ell = 0, \ldots, n-1$.

Definition 4.3 (Negatively Wrapped Convolution (NWC))**.** Similarly, the *negatively wrapped convolution* of f and g is a vector $h^- = [h_0^-, \ldots, h_{n-1}^-]$ defined by:

$$h_\ell^- = \sum_{i=0}^{n-1} (f_i \cdot g_{\ell-i} - f_i \cdot g_{n+\ell-i})$$

for $\ell = 0, \ldots, n-1$.

These wrapped convolutions cleverly reduce the length of the convolution output, computed modulo $x^n \pm 1$.

Observation 4.4. *If we think of $f(x)$ and $g(x)$ as polynomials with degree less than n, then the polynomials for the positively and negatively wrapped convolutions are:*

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

4.2 Computing Wrapped Convolutions Using FFT

4.2.1 Positively Wrapped Convolution (PWC)

To compute the positively wrapped convolution efficiently, we use the Fast Fourier Transform (FFT). The algorithm is similar to the standard FFT-based convolution but employs an n-th primitive root of unity (PROU) instead of a $2n$-th PROU.

Algorithm 3 PWC-WITH-PROU(f, g, ω)

```
1: Input: Vectors $f = [f_0, \dots, f_{n-1}]$, $g = [g_0, \dots, g_{n-1}]$, and an $n$-PROU $\omega \in \mathbb{R}$
2: Output: The positively wrapped convolution $h^+ = [h_0, \dots, h_{n-1}]$ of $f$ and $g$
3: $a_0, \dots, a_{n-1} \leftarrow \mathrm{FFT}(f, \omega)$
4: $b_0, \dots, b_{n-1} \leftarrow \mathrm{FFT}(g, \omega)$
5: for $i = 0$ to $n - 1$ do
6:     $c_i \leftarrow a_i \cdot b_i$
7: end for
8: $h_0, \dots, h_{n-1} \leftarrow \frac{1}{n} \cdot \mathrm{FFT}([c_0, \dots, c_{n-1}], \omega^{-1})$
9: return $h_0, \dots, h_{n-1}$
```

Lemma 4.5 (PWC over Rings Supporting FFT)**.** *If R is a ring containing an n-PROU ω, then the PWC of two polynomials $f(x), g(x) \in R[x]$ with degrees less than n can be computed using Algorithm 3 with:*

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

4.2.2 Negatively Wrapped Convolution (NWC)

To compute negatively wrapped convolutions, we again use FFT, but this time with a $2n$-PROU. The approach involves transforming the polynomials using a primitive root of unity, computing the PWC in the transformed domain, and then reversing the transformation.

Algorithm 4 NWC-WITH-PROU(f, g, ω)

```
1: Input: Vectors $f = [f_0, \dots, f_{n-1}]$, $g = [g_0, \dots, g_{n-1}]$, and a $2n$-PROU $\omega \in \mathbb{R}$
2: Output: The negatively wrapped convolution $h^- = [h_0, \dots, h_{n-1}]$ of $f$ and $g$
3: Compute $\tilde{f}(x) := f(\omega \cdot x)$ and $\tilde{g}(x) := g(\omega \cdot x)$
4: $\tilde{h}(x) \leftarrow$ PWC-WITH-PROU$(\tilde{f}, \tilde{g}, \omega^2)$
5: Compute $[h_0, \dots, h_{n-1}]$ of $h^-(x) := \tilde{h}(\omega^{-1} \cdot x)$
6: return $h_0, \dots, h_{n-1}$
```

Lemma 4.6 (NWC over Rings Supporting FFT)**.** *If R is a ring with a $2n$-PROU ω, then the NWC of two polynomials $f(x), g(x) \in R[x]$ with degrees less than n can be computed using Algorithm 4 with:*

- *$O(n \log n)$ additions of elements in R,*
- *$O(n \log n)$ multiplications of an element in R with a power of ω,*
- *$2n$ multiplications of elements in R,*
- *n divisions by n.*

Note that the number of multiplications of arbitrary elements in R is n rather than $2n$, demonstrating an efficiency gain.

5 Conclusion

The Schönhage-Strassen algorithm represents a groundbreaking advancement in computational number theory, particularly for the multiplication of large integers. By employing the Fast Fourier Transform (FFT) over rings and extending the use of roots of unity, Schönhage and Strassen achieved an impressive time complexity of $O(n \cdot \log n \cdot \log \log n)$. This efficiency established their method as the gold standard for large integer multiplication for over thirty years.

In this paper, we examined the core principles behind the Schönhage-Strassen algorithm, including Principal Roots of Unity (PROU), the Discrete Fourier Transform (DFT), and the FFT, and discussed how these concepts facilitate efficient polynomial multiplication. We also addressed the algorithm's approach to handling rings that lack suitable PROU through ring expansion and modular arithmetic.

Although newer algorithms, such as those by Fürer and Harvey–van der Hoeven, offer better asymptotic complexity, the Schönhage-Strassen algorithm continues to be highly relevant. Its practical performance remains superior for many real-world applications due to the constant factors involved in newer methods.

References

[1] A. Schönhage and V. Strassen, Schnelle Multiplikation großer Zahlen, *Computing*, 7:281–292, 1971. https://doi.org/10.1007/BF02242355.

[2] D. Harvey and J. van der Hoeven, Integer multiplication in time $O(n \log n)$, *Annals of Mathematics*, 193(2):563–617, 2021. https://doi.org/10.4007/annals.2021.193.2.4.

[3] M. Fürer, Faster integer multiplication, In *Proc. STOC '07*, pages 57–66, San Diego, Jun 2007. https://web.archive.org/web/20070305123131/http://www.cse.psu.edu/~furer/Papers/mult.pdf.

[4] A. Karatsuba and Yu. Ofman, Multiplication of Many-Digital Numbers by Automatic Computers, *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962. Translation in the academic journal *Physics-Doklady*, 7 (1963), pp. 595–596. https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=dan&paperid=26729&option_lang=eng.