SCHÖNHAGE–STRASSEN MULTIPLICATION

ALBERT TAM

1. Introduction

Integer multiplication seems like a mundane, simple task, but being able to multiply integers quickly is necessary for a wide range of mathematical fields today, including cryptography. The "schoolbook" algorithm, or long multiplication, runs in $O(n^2)$ time for an integer of size n. Improvements made by Karatsuba [1] brought that complexity down to $O(n^{\log_2 3})$ time, and the Toom–Cook algorithm [2] has a complexity of $O(n^{1+\epsilon})$ for some $\epsilon > 0$. In 1971, Schönhage and Strassen presented two multiplication algorithms: one with a complexity of $O(n \log n \log\log n)$ for two N-digit numbers, and a simpler one with a worse complexity [3]. In 1982, Schönhage made further improvements and greatly simplified the algorithm that was originally presented [4]. The Schönhage–Strassen algorithm usually refers to this 1982 algorithm, which uses a Fourier-like transform in the ring $\mathbb{Z}/(2^{2^k}+1)\mathbb{Z}$ and recursive calculations in smaller and smaller rings. Here, we will present the theory, implementations, and complexities of the simpler algorithm from 1971 (hereafter the "Schönhage–Strassen's first algorithm") and the revised 1982 algorithm (hereafter "the Schönhage–Strassen algorithm," or "the second algorithm").

2. The discrete Fourier transform

The discrete Fourier transform (DFT) is directly used in the first algorithm and forms the foundation of the theory for the second. The DFT transforms a sequence of complex numbers $\{\mathbf{a_n}\} := a_0, a_1, \cdots a_{N-1}$ into another sequence $\{\mathbf{A_n}\} := A_0, A_1, \cdots A_{N-1}$, defined as:

$$A_k = \sum_{n=0}^{N-1} a_n e^{-\frac{i2\pi}{N}kn}$$

The inverse of the transform, mapping $\{\mathbf{A_n}\} := A_0, A_1, \cdots A_{N-1}$ back to $\{\mathbf{a_n}\} := a_0, a_1, \cdots a_{N-1}$, is defined as:

$$a_k = \frac{1}{N} \sum_{n=0}^{N-1} A_n e^{\frac{i2\pi}{N}kn}$$

Computing the DFT is done with a number of algorithms broadly classified as fast Fourier transforms (FFTs), one of which is the Cooley–Tukey algorithm [5]. This algorithm reduces the DFT of a sequence of length 2^N into $O(N \log N)$ DFTs of length 2, a great improvement over the $O(N^2)$ result that direct evaluation from the definition requires. The FFT speeds up the calculation of convolutions, which are a crucial part of both Schönhage–Strassen algorithms.

3. Cyclic convolutions

A cyclic convolution between two sequences $\{\mathbf{a_n}\} := a_0, a_1, \cdots a_{N-1}$ and $\{\mathbf{b_n}\} := b_0, b_1, \cdots b_{N-1}$ has length N and is defined as:

$$(\mathbf{a} \times \mathbf{b})_n = \sum_{i+j \equiv n \pmod N} a_i b_j$$

A cyclic convolution can be used to calculate the product of two integers a and b. If we split each number into k parts of l bits each, the cyclic convolution calculates the product mod $(2^{kl} - 1)$. Therefore, if we pad each number's resulting sequence with kl zeros, we can calculate the product mod $(2^{2kl} - 1)$ and therefore calculate the full product.

However, calculating the cyclic convolution directly from the definition can only be done in $O(N^2)$ time. We can do better by using a theorem that relates DFTs and cyclic convolutions:

Theorem 3.1 (Cyclic convolution theorem)**.** *Let* **IDFT** *be the inverse discrete Fourier transform,* **DFT** *be the discrete Fourier transform, and* $\star$ *represent the componentwise product of two* N*-length sequences. Then* $\mathbf{a} \times \mathbf{b} = \mathbf{IDFT}(\mathbf{DFT}(\mathbf{a}) \star \mathbf{DFT}(\mathbf{b}))$.

Proof. This theorem is equivalent to the statement that $\mathbf{DFT}(\mathbf{a} \times \mathbf{b}) = \mathbf{DFT}(\mathbf{a}) \star \mathbf{DFT}(\mathbf{b})$. Therefore, let's prove this statement instead by looking at $\mathbf{DFT}(\mathbf{a} \times \mathbf{b})_n$. To make this easier, let $a_{-n} = a_{N-n}$:

$$\begin{aligned}
\mathbf{DFT}(\mathbf{a} \times \mathbf{b})_n &= \sum_{k=0}^{N-1} (\mathbf{a} \times \mathbf{b})_k e^{\frac{-2\pi i}{N} kn} \\
&= \sum_{k=0}^{N-1} \sum_{m=0}^{N-1} a_m b_{k-m} e^{\frac{-2\pi i}{N} kn} \\
&= \sum_{m=0}^{N-1} a_m \sum_{k=0}^{N-1} b_{k-m} e^{\frac{-2\pi i}{N} kn} \\
&= \sum_{m=0}^{N-1} a_m \sum_{k-m=-m}^{N-m-1} b_{k-m} e^{\frac{-2\pi i}{N}(k-m)n} e^{\frac{-2\pi i}{N} mn} \\
&= \sum_{m=0}^{N-1} a_m e^{\frac{-2\pi i}{N} mn} \mathbf{DFT}(b)_n \\
&= \mathbf{DFT}(b)_n \sum_{m=0}^{N-1} a_m e^{\frac{-2\pi i}{N} mn} \\
&= \mathbf{DFT}(a)_n \mathbf{DFT}(b)_n
\end{aligned}$$

■

Therefore, we can actually compute the cyclic convolution using 3 Fourier transforms. Using the FFT, cyclic convolutions can be computed in $O(n \log n)$ time instead of $O(n^2)$. Because cyclic convolutions can be used to multiply integers, the convolution theorem leads us to our first algorithm.

4. Schonhage–Strassen's first algorithm

This algorithm is the first one presented in Schönhage and Strassen's 1971 paper, and since it is simpler to understand, we will also present it first here. The algorithm, which calculates the product C of A and B, is as follows:

Algorithm 1: Schönhage–Strassen's first algorithm (1971)

Input : $0 \leq A, B < 2^n - 1$, integers l and K such that $l2^K \geq 2n$
Output: C

decompose $A = \sum_{j=0}^{2^K-1} a_j 2^{jl}$ with $0 \leq a_j < 2^l$ and $a_j = 0$ for $j > 2^{K-1}$
decompose B similarly
$\mathbf{a} \leftarrow \mathbf{DFT}(\mathbf{a})$, $\mathbf{b} \leftarrow \mathbf{DFT}(\mathbf{b})$
for $j \leftarrow 0$ **to** $2^K - 1$ **do**
  $c_j \leftarrow a_j b_j$; `// Call recursively`
end
$\mathbf{c} \leftarrow \mathbf{IDFT}(\mathbf{c})$
round elements in $\mathbf{c}$
for $j \leftarrow 0$ **to** $2^K - 1$ **do**
  if $c_j \geq 2^l$ **then**
    $c_j \leftarrow c_j - 2^l$; `// Ensure correct interval`
  end
end
$C = \sum_{j=0}^{2^K-1} c_j 2^{jl}$

This algorithm yields the correct result, since this is simply an application of the cyclic convolution theorem proved above. But what about its time complexity?

Theorem 4.1. *The precursor algorithm has a time complexity of* $O(K^{\log * n} n \log n \log \log n \cdots \log^{o((\log * n)-1)} n)$ *for some constant* $K > 0$.

We won't prove this theorem, since this algorithm is not the primary focus of this paper. However, a proof can be found as part of Harvey and van der Hoeven's paper about $O(n \log n)$ multiplication, which builds off of these ideas [6].

5. Number theoretic transforms

Roots of unity in $\mathbb{C}$ require high floating-point precision, so errors ecome increasingly common as larger and larger numbers are multiplied. As a result, the second algorithm uses the number theoretic transform (NTT), which is essentially an analog of the discrete Fourier transform on a finite ring. Define the NTT on $\mathbb{Z}/m\mathbb{Z}$ as follows, where ω is an element of order N in $\mathbb{Z}/m\mathbb{Z}$:

$$\mathbf{NTT}(\mathbf{a})_n = \sum_{n=0}^{N-1} a_n \omega^{-kn}$$

Similarly, the inverse transform is:

$$\mathbf{INTT}(\mathbf{a})_n = \frac{1}{N}\sum_{n=0}^{N-1} a_n \omega^{kn}$$

Because they only rely on integer arithmetic, NTTs do not run into floating-point precision problems. In addition, some rings have simple "roots of unity" (elements of order N) that we can easily input. For example, in the ring $\mathbb{Z}/(2^{2^n}+1)\mathbb{Z}$, 2 is a 2^{n+1}th root of unity. These are all perks on top of the fact that FFT algorithms can be applied to NTTs as well with only minor changes that do not affect the algorithm's $O(n \log n)$ complexity.

6. Negacyclic convolutions

Another issue with cyclic convolutions is that to calculate the actual product of numbers split into n parts, the DFT performed must be zero-padded to a size of $2n$ to preserve the final result. However, we can reduce the DFT size to n using a negacyclic convolution. The negacyclic convolution is defined as:

$$(\mathbf{a} \times_- \mathbf{b})_n = \sum_{i+j=n} a_i b_j - \sum_{i+j=n+k} a_i b_j$$

Notice that taken $\mod (x^n+1)$, term n of the negacyclic convolution is equal to $\sum_{i+j=n} a_i b_j + x^n \sum_{i+j=n+k} a_i b_j$, which is equivalent to the product. The negacyclic convolution can be calculated from a cyclic convolution of size n by multiplying each sequence by powers of a $2n$-th root of unity θ. To show this, let $\widetilde{a_i} = \theta^i a_i$ and $\widetilde{b_j} = \theta^j b_j$. Then compute the cyclic convolution $\widetilde{c_k}$:

$$\begin{aligned}
\widetilde{c_k} &= (\widetilde{\mathbf{a}} \times \widetilde{\mathbf{b}})_k \\
&= \sum_{i+j \equiv k \pmod n} \widetilde{a_i}\widetilde{b_j} \\
&= \sum_{i+j=k} \widetilde{a_i}\widetilde{b_j} + \sum_{i+j=n+k} \widetilde{a_i}\widetilde{b_j} \\
&= \sum_{i+j=k} \theta^i a_i \theta^j b_j + \sum_{i+j=n+k} \theta^i a_i \theta^j b_j \\
&= \sum_{i+j=k} \theta^{i+j} a_i b_j + \sum_{i+j=n+k} \theta^{i+j} a_i b_j \\
&= \sum_{i+j=k} \theta^{k} a_i b_j + \sum_{i+j=n+k} \theta^{n+k} a_i b_j \\
&= \sum_{i+j=k} \theta^{k} a_i b_j - \sum_{i+j=n+k} \theta^{k} a_i b_j \\
&= \theta^k \Big(\sum_{i+j=k} a_i b_j - \sum_{i+j=n+k} a_i b_j \Big)
\end{aligned}$$

Then multiplying by θ^{-k} gives the negacyclic convolution and therefore the product mod (x^n+1), using a DFT of only size n. This forms one of the critical time-saving measures of the Schönhage–Strassen algorithm.

7. Schönhage–Strassen algorithm

The Schönhage–Strassen algorithm combines an NTT and a negacyclic convolution in order to perform a multiplication mod $\mathbb{Z}/(2^{2^n}+1)\mathbb{Z}$, in which powers of 2 are fast $2n$-th roots of unity. The original algorithm published by Schönhage and Strassen is slightly clumsier, so a more refined algorithm as presented in a followup paper [4] is as follows. $\mathbf{NTT}(\mathbf{a}, \omega, K)$ refers to the number theoretic transform of $\mathbf{a}$ with length K, using ω as the root of unity:

Algorithm 2: Schönhage–Strassen's second algorithm (1982)

Input : $0 \le A, B < 2^n + 1$, an integer $K = 2^k$ such that $n = MK$
Output: $C \equiv AB \pmod{2^n+1}$

decompose $A = \sum_{j=0}^{K-1} a_j 2^{jM}$ with $0 \le a_j < 2^M$ for $0 \le j < K-1$ and $0 \le a_j \le 2^M$ for $j = K-1$
decompose B similarly
choose $n' \ge 2n/K + k$, $n'|K$; let $\theta = 2^{n'/K}$, $\omega = \theta^2$
for $j \leftarrow 0$ **to** $K-1$ **do**
  $(a_j, b_j) \leftarrow (\theta^j a_j, \theta^j b_j) \pmod{2^n+1}$; `// Weight for negacyclic convolution`
end
$\mathbf{a} \leftarrow \mathbf{NTT}(\mathbf{a}, \omega, K)$, $\mathbf{b} \leftarrow \mathbf{NTT}(\mathbf{b}, \omega, K)$
for $j \leftarrow 0$ **to** $K-1$ **do**
  $c_j \leftarrow a_j b_j \pmod{2^{n'}+1}$; `// Call algorithm to multiply` a_j, b_j `recursively for large` n'
end
$\mathbf{c} \leftarrow \mathbf{BackwardNTT}(\mathbf{c}, \omega, K)$
for $j \leftarrow 0$ **to** $K-1$ **do**
  $c_j \leftarrow c_j/\theta^j \pmod{2^{n'}+1}$; `// Remove weights`
  if $c_j \ge (j+1)2^{2M}$ **then** `// Keep` c_j `in the right interval`
    $c_j \leftarrow c_j - (2^{n'}+1)$
  end
end
$C = \sum_{j=0}^{K-1} c_j 2^{jM}$

Theorem 7.1. *Algorithm 2 has a time complexity of* $O(N \log N \log\log N)$.

Proof. Assume that $K = O(\sqrt{n})$. Then, we have $n' = O(\sqrt{n})$. Decomposition clearly costs $O(n)$. Since the weights θ are powers of 2, multiplying by them is equivalent to shifting and is therefore a linear operation. As a result, weighting and de-weighting cost $O(n)$. For the NTT and inverse NTT, the complexity is simply $O(K \log K)$ multiplied by the cost of one butterfly operation mod $(2^{n'}+1)$, which is just $O(n')$. Therefore, the cost of each NTT is $O(Kn' \log K) = O(nlogn)$. componentwise multiplication calls the algorithm recursively a

total of $O(\frac{2n}{n'})$, or $O(\sqrt{n})$ times, and each recursive call involves numbers of size n'. Therefore, letting $M(n)$ be the complexity of this algorithm we have the following recurrence relation:

$$M(n) = O(\sqrt{n})M(\sqrt{n}) + O(n \log n)$$

Unrolling the recursion, we have:

$$\begin{aligned} M(n) &= O(n^{1/2})[O(n^{1/4})M(n^{1/4}) + O(n^{1/2} \log n^{1/2})] + O(n \log n) \\ &= O(n \log n) + O(n \log n) + O(n^{3/4})M(n^{1/4}) \end{aligned}$$

If there are k terms of $O(n \log n)$, the final term becomes $O(n^{(2^k-1)/2^k})M(n^{1/2^k})$. Therefore, when there are $O(\log \log n)$ terms of $O(n \log n)$, the final term is:

$$\begin{aligned} O(n^{(2^{\log \log n}-1)/2^{\log \log n}})M(n^{1/2^{\log \log n}}) &= O(n^{(2^{\log \log n}-1)/2^{\log \log n}})M(n^{1/2^{\log \log n}}) \\ &= O(n^{(\log n-1)/\log n})M(2) \\ &\leq O(n). \end{aligned}$$

Therefore, this final term is surpassed by the complexity of the extra $O(n \log n)$ terms, of which there are $O(\log \log n)$. Therefore, the complexity is $O(n \log n \log \log n)$. ■

References

[1] Anatoly A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[3] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3):281–292, Sep 1971.

[4] Arnold Schönhage. Asymptotically fast algorithms for the numerical muitiplication and division of polynomials with complex coefficients. In Jacques Calmet, editor, *Computer Algebra*, pages 3–15, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[5] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[6] David Harvey and Joris van der Hoeven. Integer multiplication in time o(n log n). 03 2019.