Create account Log in

Article Talk Read Edit More Search

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

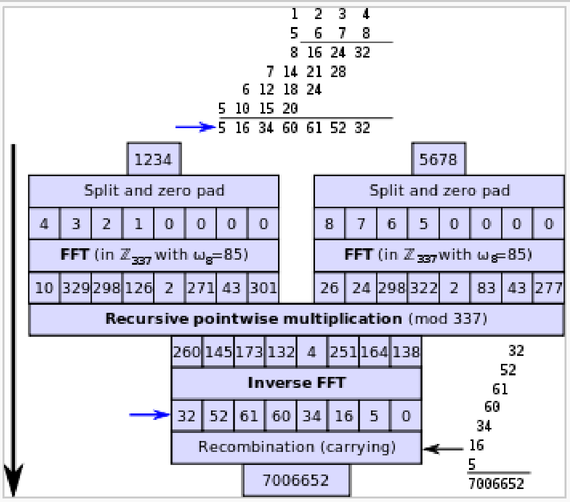
Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Français
한국어
Italiano
Português
Русский
Suomi
ไทย
Edit links

Schönhage–Strassen algorithm

From Wikipedia, the free encyclopedia

The **Schönhage–Strassen algorithm** is an asymptotically fast multiplication algorithm for large integers. It was developed by Arnold Schönhage and Volker Strassen in 1971.[1] The run-time bit complexity is, in Big O notation, $O(n \log n \log \log n)$ for two n-digit numbers. The algorithm uses recursive Fast Fourier transforms in rings with $2^{2^n} + 1$ elements, a specific type of number theoretic transform.



The Schönhage–Strassen algorithm is based on the Fast Fourier transform (FFT) method of integer multiplication. This figure demonstrates multiplying 1234 × 5678 = 7006652 using the simple FFT method. Number-theoretic transforms in the integers modulo 337 are used, selecting 85 as an 8th root of unity. Base 10 is used in place of base 2^w for illustrative purposes. Schönhage–Strassen improves on this by using negacyclic convolutions.

The Schönhage–Strassen algorithm was the asymptotically fastest multiplication method known from 1971 until 2007, when a new method, Fürer's algorithm, was announced with lower asymptotic complexity;[2] however, Fürer's algorithm currently only achieves an advantage for astronomically large values and is not used in practice.

In practice the Schönhage–Strassen algorithm starts to outperform older methods such as Karatsuba and Toom–Cook multiplication for numbers beyond $2^{2^{15}}$ to $2^{2^{17}}$ (10,000 to 40,000 decimal digits).[3][4][5] The GNU Multi-Precision Library uses it for values of at least 1728 to 7808 64-bit words (33,000 to 150,000 decimal digits), depending on architecture.[6] There is a Java implementation of Schönhage–Strassen which uses it above 74,000 decimal digits.[7]

Applications of the Schönhage–Strassen algorithm include mathematical empiricism, such as the Great Internet Mersenne Prime Search and computing approximations of π, as well as practical applications such as Kronecker substitution, in which multiplication of polynomials with integer coefficients can be efficiently reduced to large integer multiplication; this is used in practice by GMP-ECM for Lenstra elliptic curve factorization.[8]

Contents [hide]

Details [edit]

This section explains in detail how Schönhage–Strassen is implemented. It is based primarily on an overview of the method by Crandall and Pomerance in their *Prime Numbers: A Computational Perspective*.[9] This variant differs somewhat from Schönhage's original method in that it exploits the discrete weighted transform to perform negacyclic convolutions more efficiently. Another source for detailed information is Knuth's *The Art of Computer Programming*.[10]

Convolutions [edit]

Suppose we are multiplying two numbers like 123 and 456 using long multiplication with base B digits, but without performing any carrying. The result might look something like this:

		1	2	3
	×	4	5	6
		6	12	18
	5	10	15	
4	8	12		
4	13	28	27	18

This sequence (4, 13, 28, 27, 18) is called the *acyclic* or *linear convolution* of the two original sequences (1,2,3) and (4,5,6). Once you have the acyclic convolution of two sequences, computing the product of the original numbers is easy: you just perform the carrying (for example, in the rightmost column, you'd keep the 8 and add the 1 to the column containing 27). In the example this yields the correct product 56088.

There are two other types of convolutions that will be useful. Suppose the input sequences have n elements (here 3). Then the acyclic convolution has $n+n-1$ elements; if we take the rightmost n elements and add the leftmost $n-1$ elements, this produces the cyclic convolution:

	28	27	18
+		4	13
	28	31	31

If we perform carrying on the cyclic convolution, the result is equivalent to the product of the inputs mod $B^n - 1$. In the example, $10^3 - 1 = 999$, performing carrying on (28, 31, 31) yields 3141, and $3141 \equiv 56088 \pmod{999}$.

Conversely, if we take the rightmost n elements and *subtract* the leftmost $n-1$ elements, this produces the negacyclic convolution:

	28	27	18
−		4	13
	28	23	5

If we perform carrying on the negacyclic convolution, the result is equivalent to the product of the inputs mod $B^n + 1$. In the example, $10^3 + 1 = 1001$, performing carrying on (28, 23, 5) yields 3035, and $3035 \equiv 56088 \pmod{1001}$. The negacyclic convolution can contain negative numbers, which can be eliminated during carrying using borrowing, as is done in long subtraction.

Convolution theorem [edit]

Like other multiplication methods based on the Fast Fourier transform, Schönhage–Strassen depends fundamentally on the convolution theorem, which provides an efficient way to compute the cyclic convolution of two sequences. It states that:

> The cyclic convolution of two vectors can be found by taking the discrete Fourier transform (DFT) of each of them, multiplying the resulting vectors element by element, and then taking the inverse discrete Fourier transform (IDFT).

Or in symbols:

$$\text{CyclicConvolution}(X, Y) = \text{IDFT}(\text{DFT}(X) \cdot \text{DFT}(Y))$$

If we compute the DFT and IDFT using a fast Fourier transform algorithm, and invoke our multiplication algorithm recursively to multiply the entries of the transformed vectors DFT(X) and DFT(Y), this yields an efficient algorithm for computing the cyclic convolution.

In this algorithm, it will be more useful to compute the *negacyclic* convolution; as it turns out, a slightly modified version of the convolution theorem (see discrete weighted transform) can enable this as well. Suppose the vectors X and Y have length n, and a is a primitive root of unity of order $2n$ (that is, $a^{2n} = 1$ and a to all smaller powers is not 1). Then we can define a third vector A, called the *weight vector*, as:

$$A = (a^j),\ 0 \le j < n$$
$$A^{-1} = (a^{-j}),\ 0 \le j < n$$

Now, we can state:

$$\text{NegacyclicConvolution}(X, Y) = A^{-1} \cdot \text{IDFT}(\text{DFT}(A \cdot X) \cdot \text{DFT}(A \cdot Y))$$

In other words, it's the same as before except that the inputs are first multiplied by A, and the result is multiplied by A^{-1}.

Choice of ring [edit]

The discrete Fourier transform is an abstract operation that can be performed in any algebraic ring; typically it's performed in the complex numbers, but actually performing complex arithmetic to sufficient precision to ensure accurate results for multiplication is slow and error-prone. Instead, we will use the approach of the number theoretic transform, which is to perform the transform in the integers mod N for some integer N.

Just like there are primitive roots of unity of every order in the complex plane, given any order n we can choose a suitable N such that b is a primitive root of unity of order n in the integers mod N (in other words, $b^n \equiv 1$ (mod N), and no smaller power of b is equivalent to 1 mod N).

The algorithm will spend most of its time performing recursive multiplications of smaller numbers; with a naive algorithm, these occur in a number of places:

1. Inside the fast Fourier transform algorithm, where the primitive root of unity b is repeatedly powered, squared, and multiplied by other values.
2. When taking powers of the primitive root of unity a to form the weight vector A and when multiplying A or A^{-1} by other vectors.
3. When performing element-by-element multiplication of the transformed vectors.

The key insight to Schönhage–Strassen is to choose N, the modulus, to be equal to $2^n + 1$ for some integer n. This has a number of benefits in standard systems that represent large integers in binary form:

- Any value can be rapidly reduced modulo $2^n + 1$ using only shifts and adds, as explained in the next section.
- All roots of unity in this ring can be written in the form 2^k; consequently we can multiply or divide any number by a root of unity using a shift, and power or square a root of unity by operating only on its exponent.
- The element-by-element recursive multiplications of the transformed vectors can be performed using a negacyclic convolution, which is faster than an acyclic convolution and already has "for free" the effect of reducing its result mod $2^n + 1$.

To make the recursive multiplications convenient, we will frame Schönhage–Strassen as being a specialized multiplication algorithm for computing not just the product of two numbers, but the product of two numbers mod $2^n + 1$ for some given n. This is not a loss of generality, since one can always choose n large enough so that the product mod $2^n + 1$ is simply the product.

Shift optimizations [edit]

In the course of the algorithm, there are many cases in which multiplication or division by a power of two (including all roots of unity) can be profitably replaced by a small number of shifts and adds. This makes use of the observation that:

$$(2^n)^k \equiv (-1)^k \bmod (2^n + 1)$$

Note that a k-digit number in base 2^n written in positional notation can be expressed as $(d_{k-1},\dots,d_1,d_0)$. It represents the number $\sum_{i=0}^{k-1} d_i \cdot (2^n)^i$. Also note that for each d_i, $0 \le d_i < 2^n$.

This makes it simple to reduce a number represented in binary mod $2^n + 1$: take the rightmost (least significant) n bits, subtract the next n bits, add the next n bits, and so on until the bits are exhausted. If the resulting value is still not between 0 and 2^n, normalize it by adding or subtracting a multiple of the modulus $2^n + 1$. For example, if n=3 (and so the modulus is $2^3+1 = 9$) and the number being reduced is 656, we have:

$$656 = 1010010000_2 \equiv 000_2 - 010_2 + 010_2 - 1_2 = 0 - 2 + 2 - 1 = -1 \equiv 8 \pmod{2^3 + 1}.$$

Moreover, it's possible to effect very large shifts without ever constructing the shifted result. Suppose we have a number A between 0 and 2^n, and wish to multiply it by 2^k. Dividing k by n we find $k = qn + r$ with $r < n$. It follows that:

$$A(2^k) = A(2^{qn+r}) = A[(2^n)^q(2^r)] \equiv (-1)^q(\text{A shift-left } r) \pmod{2^n + 1}.$$

Since A is $\le 2^n$ and $r < n$, A shift-left r has at most $2n-1$ bits, and so only one shift and subtraction (followed by normalization) is needed.

Finally, to divide by 2^k, observe that squaring the first equivalence above yields:

$$2^{2n} \equiv 1 \pmod{2^n + 1}$$

Hence,

$$A/2^k = A(2^{-k}) \equiv A(2^{2n-k}) = \text{A shift-left } (2n - k) \pmod{2^n + 1}.$$

Overview [edit]

The algorithm follows a split, evaluate (forward FFT), pointwise multiply, interpolate (inverse FFT), and combine phases similar to Karatsuba and Toom-Cook methods.

Given input numbers *x* and *y*, and an integer *N*, the following algorithm computes the product *xy* mod $2^N + 1$. Provided N is sufficiently large this is simply the product.

1. Split each input number into vectors X and Y of 2^k parts each, where 2^k divides *N*. (e.g. 12345678 -> (12, 34, 56, 78)).
2. In order to make progress, it's necessary to use a smaller *N* for recursive multiplications. For this purpose choose *n* as the smallest integer at least $2N/2^k + k$ and divisible by 2^k.
3. Compute the product of X and Y mod $2^n + 1$ using the negacyclic convolution:
    1. Multiply X and Y each by the weight vector A using shifts (shift the *j*th entry left by $jn/2^k$).
    2. Compute the DFT of X and Y using the number-theoretic FFT (perform all multiplications using shifts; for the 2^k-th root of unity, use $2^{2n/2^k}$).
    3. Recursively apply this algorithm to multiply corresponding elements of the transformed X and Y.
    4. Compute the IDFT of the resulting vector to get the result vector C (perform all multiplications using shifts). This corresponds to interpolation phase.
    5. Multiply the result vector C by A^{-1} using shifts.
    6. Adjust signs: some elements of the result may be negative. We compute the largest possible positive value for the *j*th element of C, $(j + 1)2^{2N/2^k}$, and if it exceeds this we subtract the modulus $2^n + 1$.
4. Finally, perform carrying mod 2^N+1 to get the final result.

The optimal number of pieces to divide the input into is proportional to $\sqrt{N}$, where *N* is the number of input bits, and this setting achieves the running time of O(*N* log *N* log log *N*),[1][9] so the parameter *k* should be set accordingly. In practice, it is set empirically based on the input sizes and the architecture, typically to a value between 4 and 16.[8]

In step 2, the observation is used that:

- Each element of the input vectors has at most $n/2^k$ bits;
- The product of any two input vector elements has at most $2n/2^k$ bits;
- Each element of the convolution is the sum of at most 2^k such products, and so cannot exceed $2n/2^k + k$ bits.
- *n* must be divisible by 2^k to ensure that in the recursive calls the condition "2^k divides *N*" holds in step 1.

Optimizations [edit]

This section explains a number of important practical optimizations that have been considered when implementing Schönhage–Strassen in real systems. It is based primarily on a 2007 work by Gaudry, Kruppa, and Zimmermann describing enhancements to the GNU Multi-Precision Library.[8]

Below a certain cutoff point, it's more efficient to perform the recursive multiplications using other algorithms, such as Toom–Cook multiplication. The results must be reduced mod $2^n + 1$, which can be done efficiently as explained above in Shift optimizations with shifts and adds/subtracts.

Computing the IDFT involves dividing each entry by the primitive root of unity $2^{2n/2^k}$, an operation that is frequently combined with multiplying the vector by A^{-1} afterwards, since both involve division by a power of two.

In a system where a large number is represented as an array of 2^w-bit words, it's useful to ensure that the vector size 2^k is also a multiple of the bits per word by choosing $k \geq w$ (e.g. choose $k \geq 5$ on a 32-bit computer and $k \geq 6$ on a 64-bit computer); this allows the inputs to be broken up into pieces without bit shifts, and provides a uniform representation for values mod $2^n + 1$ where the high word can only be zero or one.

Normalization involves adding or subtracting the modulus 2^n+1; this value has only two bits set, which means this can be done in constant time on average with a specialized operation.

Iterative FFT algorithms such as the Cooley–Tukey FFT algorithm, although frequently used for FFTs on vectors of complex numbers, tend to exhibit very poor cache locality with the large vector entries used in Schönhage–Strassen. The straightforward recursive, not in-place implementation of FFT is more successful, with all operations fitting in the cache beyond a certain point in the call depth, but still makes suboptimal use of the cache in higher call depths. Gaudry, Kruppa, and Zimmerman used a technique combining Bailey's 4-step algorithm with higher radix transforms that combine multiple recursive steps. They also mix phases, going as far into the algorithm as possible on each element of the vector before moving on to the next one.

The "square root of 2 trick", first described by Schönhage, is to note that, provided $k \geq 2$, $2^{3n/4} - 2^{n/4}$ is a square root of 2 mod 2^n+1, and so a $4n$-th root of unity (since $2^{2n} \equiv 1$). This allows the transform length to be extended from 2^k to 2^{k+1}.

Finally, the authors are careful to choose the right value of k for different ranges of input numbers, noting that the optimal value of k may go back and forth between the same values several times as the input size increases.

References [edit]

1. ^ *a* *b* A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen", *Computing* **7** (1971), pp. 281–292.
2. ^ Martin Fürer, "Faster integer multiplication", STOC 2007 Proceedings, pp. 57–66.
3. ^ Rodney Van Meter and Kohei M. Itoh, "Fast quantum modular exponentiation", *Physical Review* A, Vol. 71 (2005).
4. ^ Overview of Magma V2.9 Features, arithmetic section: Discusses practical crossover points between various algorithms.
5. ^ Luis Carlos Coronado García, "Can Schönhage multiplication speed up the RSA encryption or decryption?", *University of Technology, Darmstadt* (2005)
6. ^ "MUL_FFT_THRESHOLD". *GMP developers' corner*. Retrieved 3 November 2011.
7. ^ "An improved BigInteger class which uses efficient algorithms, including Schönhage–Strassen". Oracle. Retrieved 2014-01-10.
8. ^ *a* *b* *c* Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based Implementation of Schönhage–Strassen's Large Integer Multiplication Algorithm. Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation, pp.167–174.
9. ^ *a* *b* R. Crandall & C. Pomerance. *Prime Numbers – A Computational Perspective*. Second Edition, Springer, 2005. Section 9.5.6: Schönhage method, p. 502. ISBN 0-387-94777-9
10. ^ Donald E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition), 1997. Addison-Wesley Professional, ISBN 0-201-89684-2. Section 4.3.3.C: Discrete Fourier transforms, pg.305.

v · t · e **Number-theoretic algorithms** [hide]

Primality tests	AKS test · APR test · Baillie–PSW · ECPP test · Elliptic curve · Pocklington · Fermat · Lucas · *LUCAS–LEHMER* · *LUCAS–LEHMER–RIESEL* · *PROTH'S THEOREM* · *PÉPIN'S* · Quadratic Frobenius test · Solovay–Strassen · Miller–Rabin
Prime-generating	Sieve of Atkin · Sieve of Eratosthenes · Sieve of Sundaram · Wheel factorization
Integer factorization	Continued fraction (CFRAC) · Dixon's · Lenstra elliptic curve (ECM) · Euler's · Pollard's rho · $p-1$ · $p+1$ · Quadratic sieve (QS) · General number field sieve (GNFS) · *Special number field sieve (SNFS)* · Rational sieve · Fermat's · Shanks' square forms · Trial division · Shor's
Multiplication	Ancient Egyptian · Long · Karatsuba · Toom–Cook · **Schönhage–Strassen** · Fürer's
Discrete logarithm	Baby-step giant-step · Pollard rho · Pollard kangaroo · Pohlig–Hellman · Index calculus · Function field sieve
Greatest common divisor	Binary · Euclidean · Extended Euclidean · Lehmer's
Modular square root	Cipolla · Pocklington's · Tonelli–Shanks
Other algorithms	Chakravala · Cornacchia · Integer relation · Integer square root · Modular exponentiation · Schoof's

Italics indicate that algorithm is for numbers of special forms · Smallcaps indicate a deterministic algorithm

Categories: Computer arithmetic algorithms | Multiplication