EVOLUTION OF FAST INTEGER MULTIPLICATION ALGORITHMS

BENJAMIN HILLARD

Abstract. This paper will discuss a few algorithms which can be used to multiply two integer factors in an efficient manor. The Karatsuba algorithm from 1960 will be described as well as how one would calculate the time complexity of using it for any two integers. The steps to the Toom-Cook algorithm are also outlined with some information on time complexity of the algorithm as well. The paper also briefly goes over various other integer multiplication algorithms developed in the more recent past that gradually reduce the time complexity to the conjectured limit of $O(n \log n)$.

1. Introduction

Throughout all of history, mathematicians have multiplied integers. This was used for tasks such as selling products, building structures, counting quickly, navigation, predicting amounts, and many more things. For nearly all that time, the best known method to multiply these two integers was to use long multiplication by writing down the two numbers, getting the product of each digit multiplied by every other digit of the other factor, and summing these products to get the final product. This was the only efficient way of multiplying integers until in 1960, when a 23-year-old Russian student Anatoli Karatsuba, in response to a claim from a professor, found a way to multiply two integers faster than with long multiplication and began a quest for the fastest way to multiply numbers.

The Karatsuba algorithm, as stated above, is an algorithm which can be used to multiply two integers with asymptotically less steps than is needed with long multiplication. The basic idea is to use more addition and subtraction in order to lower the amount of multiplication required, as multiplication is a more 'difficult' operation than addition or subtraction. The algorithm also splits down recursively such that it is used by itself to multiply integers, which intuitively makes sense but is important to keep in mind. This algorithm also led to the discovery of the Toom-Cook algorithm.

The Toom-Cook algorithm is also performed recursively, but it actually includes several different instances where different operations are done differently. One of these instances is in fact identical to the Karatsuba algorithm, and the Toom-Cook algorithm can be called a generalization of the Karatsuba algorithm. Each of these different instances has a different amount of steps required based on the size of the factors, but it can be made to perform asymptotically faster than other algorithms before it.

After Toom-Cook came the Schönhage-Strassen algorithm, which was asymptotically even faster than those before. It is also much more complex, requiring more than just basic operations such as using the Convolution Theorem, roots of unity, and using shifts of digits of numbers in binary. While the process is more complex, it still preforms faster for very large numbers than the others do.

Date: June 2022.

With all of these algorithms, there is added complexity which adds more time. As a result, all of the multiplication algorithms discussed in this paper are actually much slower for any values that can be easily written. For any human it is much faster to use long multiplication for any numbers that could be easily written down, as even the Karatsuba algorithm only becomes faster with numbers with thousands of digits, and other later ones with even larger numbers. That is why they are considered only asymptotically faster, but that doesn't mean they are not useful for many different things.

1.1. **Background.**

The time complexity of these algorithms is typically used to determine their speed/efficiency, and the notation used is big O notation. Big O notation is used to describe the behavior of a function as the input approaches a certain value, in this case the number of operations as the size of the inputs approach infinity. It is written as the greatest term in order of magnitude without any constants, as this will show the behavior of the function as it approaches infinity.

For example, the big O notation of a quadratic function as it approaches infinity would be written as

$$O(x^2).$$

Note that numbers such as the base of a logarithm function are also considered constants and removed as they will not change the end behavior of the function.

In this paper when discussing multiplication algorithms, n will be used to represent the number of digits in each of the factors in terms of any arbitrary base. So if the function for the number of operations required by an algorithm is $S(n) = 3n^4 - 6n + 2$, then the time complexity for that algorithm would be $O(n^4)$.

Long multiplication has a time complexity of $O(n^2)$, and many thought that this was the limit before Karatsuba made his algorithm (with time complexity $\approx O(n^{1.585})$). Note that this does not mean long multiplication will always take n^2 operations (in fact it takes about $2n^2$ depending on what you count as an operation), but that with very large n the number of operations used by long multiplication will behave as a quadratic.

If one wanted to calculate the time complexity of a counting algorithm, where given a sequence of all integers from 1 to an arbitrary end integer w, it counts the amount of numbers by going one by one and counting up by one (not an efficient algorithm but this is an example), they would do as follows. This algorithm will always have to check w numbers, and we will consider checking if there is a number a step and adding one to the total also as a step. Thus, this algorithm will always perform in $2w$ steps, meaning that the number of steps it takes is proportional to w, so the big O notation for the time complexity would be simply $O(w)$.

Big O notation has much more to it than this, and I would encourage a visit to some chart that shows all the different variation if one is curious. However, for this paper that is all that one will need to know.

For these algorithms, the hardest part turns out to be the multiplication of smaller factors inside the algorithm. These smaller operations are then done with some integer multiplication algorithm, which again uses smaller instances of integer multiplication in it's process. Because of this, recursive multiplication typically ends up being the most time consuming part of multiplication algorithms. In most algorithms, the final multiplication step is once it becomes two single digit factors and it cannot be broken down further, at which point a human who knows the multiplication table well will immediately know the answer, or some computer logic gives the answer.

2. The Karatsuba Algorithm

In 1962 the Karatsuba algorithm, designed by Anatoly Karatsuba, was published and in this algorithm Karatsuba showed a method in which two n-digit integers could be multiplied in less than $O(n^2)$ single-digit multiplication operations, which is the time complexity of multiplying those same numbers using the standard algorithm. Karatsuba's algorithm achieves the time complexity of $O(n^{\log_2 3})$, approximately $O(n^{1.585})$.

2.1. **The Karatsuba Algorithm Formula.**

Let a and b be integers in base B. To multiply them using the Karatsuba algorithm, we do as follows. First, without loss of generality, assume a is greater than or equal to b. Let n denote the number of digits in the base B representation of a. For any positive integer m that is less than n, the two numbers can be written as

$$a = a_1 * B^m + a_0$$

and

$$b = b_1 * B^m + b_0,$$

where a_0 and b_0 are the first m digits in base B of a and b from the right, and a_1 and b_1 are the rest of the digits in base B from the right, or the first $n - m$ digits in base B from the left, of a and b.[1] Thus, the product of a and b can be written as

$$\begin{aligned} ab &= (a_1 * B^m + a_0)(b_1 * B^m + b_0) \\ &= a_1b_1 * B^{2m} + (a_1b_0 + a_0b_1) * B^m + a_0b_0 \\ &= c_2 * B^{2m} + c_1 * B^m + c_0 \end{aligned}$$

where

$$\begin{aligned} c_2 &= a_1b_1 \\ c_1 &= a_1b_0 + a_0b_1 \\ c_0 &= a_0b_0. \end{aligned}$$

The final expression for the value of ab in terms of c_2, c_1, c_0 could be solved by computing four products, but the value of c_1 can also be written as

$$(a_1 + a_0)(b_0 + b_1) - c_2 - c_0$$

because

$$\begin{aligned} c_1 &= (a_1 + a_0)(b_0 + b_1) - c_2 - c_0 \\ &= (a_1 + a_0)(b_0 + b_1) - a_1b_1 - a_0b_0 \\ &= a_1b_0 + b_1a_0 + a_1b_1 + a_0b_0 - a_1b_1 - a_0b_0 \\ c_1 &= a_1b_0 + b_1a_0. \end{aligned}$$

Using this alternate method of getting c_1 reduces the number of products computed by one by using addition and subtraction. The final product in terms of c_0, c_1, and c_2 can then be given by

$$ab = c_2 * B^{2m} + c_1 * B^m + c_0.$$[2]

[1]The most efficient choice of m is $n/2$, rounded up, so that the number of recursive calls by the function is minimized.

[2]The multiplication by B to some power is not counted as a multiplication operation for time complexity purposes as it can be done by simply shifting the other factor to get the product.

Thus, in the example where a and b are two digit integers, the product can be calculated using just three single-digit multiplications with more addition and subtraction, instead of the four single-digit multiplications that would be needed with the standard algorithm. This holds for a and b with larger n, and as n increases in size the number of single-digit multiplications will approach $n^{\log_2 3}$ or $n^{1.585}$.

Practically, the Karatsuba algorithm is performed recursively until the point in which it is more efficient to switch to long multiplication to multiply integers. This means that in the steps where a_1b_1, a_0b_0 and $(a_1+a_0)(b_0+b_1)$ are realized, the Karatsuba algorithm would again be used for multiplication if the two integers are sufficiently large, otherwise the standard algorithm would be used.

2.2. **The Karatsuba Algorithm Time Complexity.** As stated immediately before, the Karatsuba Algorithm is used recursively; it is used to perform multiplication in itself.

Proposition 2.1 (Karatsuba Time Complexity)**.** *The Karatsuba algorithm runs with time complexity of $O(n^{log_2 3})$.*

Proof. The algorithm will perform recursively by splitting a single multiplication into three smaller multiplications, where each factor has at most half the digits. If $n = 2^k$ for some integer k, the algorithm will run at most k times, and split down into 3^k single-digit multiplication operations. And for any number, leading zeros can be placed in front of the number without changing it's value in order to give it $n = 2^k$ for some k. For any factors where $n \leq 2^k$, the number of single-digit multiplication operations with be at most 3^k. Note that $k = \log_2 n$, so if $T(n)$ is the number of single-digit multiplication operations required to multiply any two factors using the Karatsuba algorithm, then $T(n) = 3^{\log_2 n}$. $T(n)$ can also be written as $n^{\log_2 3}$ or $n^{1.585}$. ■

When considering time complexity and writing it in in big O notation, only the asymptotic behavior is shown. The number of additions and subtractions performed by the Karatsuba algorithm is proportional to n, and as such is not visible in the asymptotic behavior of the time complexity of the Karatsuba algorithm and can be ignored in big O notation. Thus, the time complexity can be written as $O(n^{\log_2 3})$ or $O(n^{1.585})$. (Karatsuba & Ofman, 1962)

2.3. **Karatsuba Algorithm Example.** Following is an example of the Karatsuba algorithm. We will be calculating $4321 * 9876$, so $a = 4321$, $b = 9876$ and base $B = 10$, so $n = 4$. The first step is to chose a good m, which (1) will be $n/2 = 2$. Thus, for $a_1 = 43$, $a_0 = 21$, $b_1 = 98$ and $b_0 = 76$,

$$a = a_1 * B^m + a_0$$
$$b = b_1 * B^m + b_0.$$

Next, we must calculate

$$c_0 = a_0b_0 = 21 * 76$$
$$c_2 = a_1b_1 = 43 * 98,$$

as well as

$$c_1 = (a_1 + a_0)(b_0 + b_1) - c_2 - c_0 = 64 * 174 - c_2 - c_0$$

These smaller products would also be calculated via some multiplication algorithm, the specific algorithm changing depending on their size. For simplification purposes, this will be skipped and the products are:

$$c_0 = 1596$$

$$c_2 = 4214$$
$$c_1 = 11136 - c_2 - c_0 = 11136 - 4214 - 1596 = 5326.$$

From here, we simply shift certain values and get the sum:

$$\begin{aligned} ab &= c_2 * B^{2m} + c_1 * B^m + c_0 \\ &= 4214 * 10^4 + 5326 * 10^2 + 1596 \\ &= 42674196. \end{aligned}$$

Which is indeed the product of 4321 and 9876.

$$\begin{tikzcd}
a_1 \arrow[r, dash, "+"] \arrow[d, dash, "*"'] & a_0 \arrow[r, dash, "="] \arrow[d, dash, "*"'] & (a_1 + a_0) \arrow[d, dash, "*"'] \\
b_1 \arrow[r, dash, "+"] \arrow[d, dash, "="'] & b_0 \arrow[r, dash, "="] \arrow[d, dash, "="'] & (b_1 + b_0) \arrow[d, dash, "-"'] \\
c_2 \arrow[r, dash, "+"] & c_0 \arrow[r, dash, "="] & (c_2 + c_0) \arrow[d, dash, "="'] \\
 & & c_1
\end{tikzcd}$$

$$\begin{tikzcd}
c_2 * B^{2m} \arrow[r, dash, "+"] & c_0 \arrow[r, dash, "+"] \arrow[d, dash, "="'] & c_1 * B^m \\
 & ab &
\end{tikzcd}$$

A diagram to visualize the Karatsuba Multiplication algorithm.

3. The Toom-Cook Algorithm

A few years later in 1963, Andrei Toom introduced another algorithm (which was improved by Stephen Cook) for multiplying integers which has an even lower asymptotic behavior of $O(n^{\log_{10}(5)/\log_{10}(3)}) \approx O(n^{1.465})$ and is a generalization of the Karatsuba Algorithm.

This algorithm has five main steps, several of which are shared by many other algorithms. These are to

- 3.1 split,
- 3.2 evaluate,
- 3.3 multiply point wise,
- 3.4 interpolate a new value and
- 3.5 recombine.

It also has different instances where a variable that determines how many smaller parts the factors are split into during the algorithm changes. This will be called k, and typically the instance is named with the form "Toom-k." The Karatsuba algorithm is an example of Toom-2, and the most well-known instance of the Toom-Cook algorithm is Toom-3.

Some important information used by this algorithm is that a polynomial of degree d can be found with $d+1$ known points on the polynomial. Additionally for polynomials p and q, $(p * q)(x) = p(x) * q(x)$.

3.1. **Split.** For integer factors l and m, first choose a base $B = b^d$, where b is the displayed base (in most cases 10) and d is $\lceil n/k \rceil + 1$. This base B will mean that l and m in base B will only have at most k digits. Next, take these digits of l and m in base B and place them in a polynomials $p(x)$ and $q(x)$ with degree $(k-1)$ such that $p(B) = l$ and $q(B) = m$.

For example, if the digits of l and m in base B in positional notation are $l_1, l_2 \ldots, l_k$ and $m_1, m_2, \ldots, m_k$, then the functions $p(x) = l_1 x^{k-1} + l_2 x^{k-2} + \ldots + l_k x^0$ and $q(x) = m_1 x^{k-1} + m_2 x^{k-2} + \ldots + m_k x^0$.

3.2. **Evaluate.**

Because there is a polynomial of degree at most d through any $(d+1)$ points, the polynomial that is the product of p and q can be calculated by using $(d+1)$ points. And because $(p * q)(x) = p(x) * q(x)$, (pq) has the property $(pq)(B) = l * m$, the product we are trying to calculate.

The degree of (pq) is the degree of p plus the degree of q, and the degree of p and of q is $k-1$, so to get this polynomial (pq) with degree $(2k-2)$ and will need $(2k-1)$ points on it.

For this step, evaluate both p and q at $2k-1$ points with the same inputs, which will be called the evaluation points. In order to make this as simple as possible, choose inputs at which the value will be simple to calculate. For Toom-3, $(k = 3)$ the common points are $0, 1, 2, -1,$ and ∞. This last point at ∞ is not actually evaluating the polynomials at ∞ but taking the limit of $p(x)/x^d$ as x goes to infinity, where d is the degree of the polynomial. This will be the highest degree coefficient, and thus doesn't require any calculations. Evaluating polynomials at the point $x = 2$ does include multiplication by small integers, but these are typically trivial in the final time calculation.

For a later step, the interpolation matrix will be defined as a matrix of size $(2k-1)\times(2k-1)$ where each row contains powers of one of these evaluation points, where the power is the column index from 0 to $(2k-2)$ (The highest degree of p and q). In the Toom-3 algorithm the interpolation matrix may look like:

$$\begin{bmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ 2^0 & 2^1 & 2^2 & 2^3 & 2^4 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.1}$$

Note that the last row is the limit, and when this matrix is multiplied by a vector of the coefficients, the final entry in the resulting vector will be the coefficient of the highest degree.

3.3. **Multiply Points.**

Next multiply each point by its partner from the other polynomial, so calculate $p(x) * q(x)$ for each of the $2k-1$ points. Remember these will all lie on the polynomial (pq) which has the property $(pq)(B) = l * m$.

This is the "hardest" part of the algorithm, because for practical implementations the algorithm is performed recursively to compute each of these multiplications, and this is the only part which is non-linear in its operations in terms of l and m.

For the next step, it will be useful to have these solutions of $(pq)(x)$ be in a vector called the solution vector.

3.4. **Interpolate.**

Now these points can be used to find the coefficients of polynomial (pq) by using matrix multiplication. To do this, note that the interpolation matrix with powers of inputs, when multiplied by a new vector of the coefficients $c_0, c_1, \ldots c_{2k-1}$ of $(pq)(x)$, will equal the vector made in the previous steps of the solutions of $(pq)(x)$. In the case of Toom-3, this may look like

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} (pq)(0) \\ (pq)(1) \\ (pq)(2) \\ (pq)(3) \\ (pq)(4) \end{bmatrix}. \tag{3.2}$$

This equation can then be solved for the vector of coefficients by multiplying both sides by the inverse of the interpolation matrix. As long as the evaluation points chosen are suitable, this is possible. This means that

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} (pq)(0) \\ (pq)(1) \\ (pq)(2) \\ (pq)(3) \\ (pq)(4) \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}. \tag{3.3}$$

Next calculate the inverse of the matrix and realize the matrix equation (which again is relatively trivial compared with the multiplication of much larger numbers). The resulting vector contains the coefficients of the polynomial (pq), which will also be the digits of the product that we want, (lm), in base B.

3.5. **Recombine.** This last step is rather simple, just solve the polynomial (pq) for B, and that will be the final product.

3.6. **Time Complexity.** The Toom-Cook algorithm runs with different a time complexity depending on what value of k is chosen. For example, Toom-2, where $k = 2$, is just the Karatsuba algorithm and runs at the same time complexity of $O(n^{\log_2 3})$. Long multiplication is just Toom-1 where $k = 1$ and has again the same time complexity of $O(n^2)$.

The Toom-Cook algorithm can be said to run in $O(T(k,n) * n^{\log_{10}(2k-1)\log_{10}(k)})$ where T is the time to perform all additions and multiplications by smaller integers and changes depending on k and n, and the power of n is the time to perform sub-multiplications. As the number k increases, the power of n will decrease but the function T will increase.

4. The Schönhage-Strassen Algorithm and $O(n \log n)$ Conjecture

This algorithm, published by Arnold Schönhage and Volker Strassen in 1971, came after the field had undergone some research. It follows a similar pattern to the Karatsuba algorithm and Toom-Cook algorithm, in which the factors are sectioned apart, evaluated in some way, interpolated to find some new value based on information known, and finally added back together. The algorithm can be considered a specialized algorithm, as instead of computing the product of two integers a and b, it computes $ab \mod 2^C + 1$, where C is also taken as an input. This is largely inconsequential, as long as the chosen C is large enough that $ab \mod 2^C + 1 = ab$.

This algorithm makes use of roots of unity, which are complex numbers that, when raised to a power, equal one. They are a bit more complicated than that, but that is all that one needs to know to use the algorithm. They are also called de Moivre numbers, and if jth root of unity is z, then $z^j = 1$. Additionally, if a jth root of unity is primitive then $z^k \neq 1$ for $k < j$ and $k \in \mathbb{N}$.

This algorithm also relies on negacyclic convolutions, which can be represented as a vector of numbers, to compute a product quickly. When multiplying two numbers using the standard algorithm, first one places one factor over the other, then multiply each digit individually, then one adds each digit in every decimal position to get a sum, and perform carrying to get the final result. The linear convolution of the two products is simply the vector of numbers that one has before performing carrying. In a more visual manner:

$$\begin{array}{cccccl}
 & & 9 & 8 & 7 & \\
 & * & 1 & 2 & 3 & \\
\hline
 & & 27 & 24 & 21 & \\
 & 18 & 16 & 14 & & \\
9 & 8 & 7 & & & \\
\hline
9 & 26 & 50 & 38 & 21 & \longleftarrow \text{The linear convolution of 987 and 123.}
\end{array}$$

This linear convolution will always have $2n-1$ elements, where n is the number of digits base B. To get what is called the acyclic convolution of the two numbers, take the first n digits of the linear convolution going from the right and add them to the first $n-1$ digits going from the left.

$$\begin{array}{ccccl}
 & 50 & 38 & 21 & \\
+ & & 9 & 26 & \\
\hline
 & 50 & 47 & 47 & \longleftarrow \text{The acyclic convolution of 987 and 123.}
\end{array}$$

Inversely, the negacyclic convolution of two numbers is calculated by taking to first n digits of the linear convolution going from the right and instead subtracting the first $n-1$ digits going from the left from them.

$$\begin{array}{ccccl}
 & 50 & 38 & 21 & \\
- & & 9 & 26 & \\
\hline
 & 50 & 29 & -5 & \longleftarrow \text{The negacyclic convolution of 987 and 123.}
\end{array}$$

One property of the negacyclic convolution of two numbers, which is used in the S and S algorithm, is that after performing carrying as one does with the standard algorithm, the result will be the product of the two original numbers reduced modulo B^n+1. For the example shown in the diagrams above, after performing carrying [3] on the negacyclic convolution $(50, 29, -5)$, the result is 5285, which is congruent to the product of $123 * 987 = 121401$ reduced modulo $10^3 + 1 = 1001$. In symbols, $5285 \equiv 121401 \mod 1001$. This is how the result of the multiplication is actually calculated.

To complete this algorithm is it also important to understand a Fourier transform. It has many applications in both math and frequency, and I would highly recommend checking it out if you would like to know more. For this paper, it will only be important to understand that, when given a continuous function in terms of space or time for a range of values, it will decompose it and give functions depending on frequency. The Fourier transform of an integrable function $f : \mathbb{R} \to \mathbb{C}$ can be defined by

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)e^{-i2\pi\xi x}dx, \quad \forall\xi \in \mathbb{R}.$$

Where the transform of function $f(x)$ at frequency ξ is given by the complex number $\hat{f}(\xi)$.

Two more pieces of background needed for this algorithm are the Discrete Fourier Transform and Fast Fourier Transform, which as they sound are an extension of the Fourier transform. A Discrete Fourier transform, given a finite sequence of equally-spaced samples of a function will give a sequence of the same size of samples of the discrete-time Fourier transform. The discrete-time Fourier transform, when given a sequence of uniformly spaced samples of a function, produces a function of frequency that is a periodic summation of the continuous Fourier Transform of the original continuous function that produced the sequence.

4.1. **The Schönhage-Strassen Algorithm Formula.** The product of x and y can be calculated as follows:[4] For this algorithm, it is necessary to form a vector $R = r^j, 0 \leq j < n$ where r is a primitive $2n$th root of unity (so $r^{2n} = 1$). We will also need a vector $R^{-1} = r^{-j}, 0 \leq j < n$ for the same r. These will be used to perform a negacyclic convolution in a later step. Choose an integer C large enough that $2^C + 1$ is larger than xy (any value of $C \geq 2n \log_2 B$ is valid, as $2^{2n \log_2 B} + 1 = B^{2n} + 1$ will always be larger than xy). Now, choose a value d such that 2^d is a factor of C, and slice x and y both into vectors X and Y with 2^d sections of equal size in terms of digits for base B. Then choose the smallest integer $f \geq 2C/2^d + d$ and divisible by 2^d. This will be used to recursively call the algorithm. Next is the main step, which will compute the negacyclic convolution of x and y and perform carrying to get the final product.

For both X and Y, multiply the vector by the weight vector R using shifts by shifting the jth entry to the left by $jc/2^d$. Next compute the Discrete Fourier Transform of both resulting vectors, which can be done using the Fast Fourier Transform by shifting the 2^kth root of unity by $2^{2c/2^d}$. Now multiply corresponding entries in X and Y. This includes some multiplication, for which in most practical cases the S and S algorithm or an instance of the Toom-Cook algorithm would be used. Now get the inverse Discrete Fourier Transform of the product vector and again only use shifts. Now once more using shifts multiply the resulting vector by R^{-1}. This vector is nearly the negacyclic convolution of the original X

[3]Using borrowing for negative numbers is also necessary as with long multiplication.

[4]This is actually a variant of the original Schönhage-Strassen algorithm which can be done in slightly less time by using the discrete Fourier transform to perform negacyclic convolutions faster.

and Y, now for all negative value add $2^d + 1$ until that entry is positive. Additionally, one can compute the largest possible positive value for the jth entry to be just $(j+1)2^{2C/2d}$, so for all values greater than this subtract the modulus $2^d + 1$.

This vector is the negacyclic convolution of X and Y, and as stated previously after performing carrying the resulting number will be the product (mod $B^2 + 1$). And because of the base chosen for our convolution (C) is large enough, this will be the product of the original integers x and y. (*Schönhage–Strassen algorithm*, 2022) (Schönhage & Strassen, 1971)

5. Integer Multiplication algorithms in the 2000s and 2010s

During the 2000s and 2010s research on fast integer multiplication algorithms was continued and would build off of previous methods found. I am going to merely mention each algorithm and some details about it in order to give an idea of how this area of study evolved over time. I will not be showing any proofs or the actual algorithms, but if anyone reading this is interested in anything mentioned here, definitely put it on your reading list. In this section, the number n will always refer to the length of the two factors.

In 2007 Martin Fürer published an algorithm with a time complexity of $O(n \log n * 2^{O(\log^* n)})$. $\log^* n$ is the iterated logarithm, which returns the number of times the logarithm must be applied onto the previous result before it reaches 1.

In 2008, Anindya De, Piyush Kurur, Chandan Saha, and Ramprasad Saptharishi published an algorithm which uses modular arithmetic instead of complex arithmetic, and achieves the same time complexity of the Fürer algorithm: $O(n \log n * 2^{O(\log^* n)})$.

In 2015, David Harvey, Joris van der Hoeven and Grégoire Lecerf published an algorithm which achieved a running time of $O(n \log n * 2^{3 \log^* n})$, creating a more 'exact' term $2^{3 \log^* n}$ instead of the earlier $2^{O(\log^* n)}$. They also proposed a version of this algorithm that runs even faster, at $O(n \log n * 2^{2 \log^* n})$, but depends on conjectures made about Mersenne primes.

In 2016, Svyatoslav Covanov and Emmanuel Thomé proposed a new algorithm that instead relies on conjectures of Fermat primes and achieves the same running time of $O(n \log n * 2^{2 \log^* n})$.

In 2018 David Harvey and Joris van der Hoeven, who had both worked on an earlier algorithm, used Minowski's theorem to unconditionally prove the upper complexity bound of $O(n \log n * 2^{2 \log^* n})$.

In 2019 they would go on to prove another algorithm which achieved the $O(n \log n)$ complexity bound conjectured by Schönhage and Strassen. (Harvey & van der Hoeven, 2021)

References

Harvey, D., & van der Hoeven, J. (2021). Integer multiplication in time $o(n \log n)$. *Annals of Mathematics*, *193*(2). doi: 10.4007/annals.2021.193.2.4

Karatsuba, A., & Ofman, Y. (1962, 12). Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, *7*, 595.

Schönhage, A., & Strassen, V. (1971). Schnelle multiplikation großer zahlen. *Computing*, *7*(3-4), 281–292. doi: 10.1007/bf02242355

Schönhage–strassen algorithm. (2022, May). Wikimedia Foundation. Retrieved from `https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm`