EVOLUTION OF MULTIPLICATION ALGORITHMS FOR INTEGERS, MATRICES, AND MORE

SHAMIK KHOWALA

1. Abstract

We explain increasingly faster algorithms for computing the multiplication of two integers and of two matrices. We explain the Karatsuba, Toom-Cook, and Schönhage-Strassen algorithms, which successively improve on the naive integer multiplication approach taught in school. Then we take a slight detour into complex number and polynomial multiplication algorithms, which employ similar techniques for basic optimization as the algorithms for integers. Finally we discuss improvements in matrix multiplication, everything from the naive $\mathcal{O}(n^3)$ approach to the first algorithm introduced by Strassen with $\mathcal{O}(n^{2.81})$ time to the Laser Method and Refined Laser Method, which achieved $\mathcal{O}(n^{2.3729})$ time. Both integer and matrix multiplication have had even more recent developments reducing these computational complexities further; we do not discuss these, but provide references.

2. Introduction

Before the 1960s, mathematicians and computer programs alike employed naïve approaches for integer and matrix multiplication. They would multiply the digits of the integers digit by digit and add up the results multiplied by powers of ten; matrices would be multiplied via a laborious row by row, column by column, entry by entry approach. Anatoly Karatsuba's 1960 discovery of his namesake Karatsuba algorithm (published in 1962, [KO62]) sparked the beginning of a search for more optimal multiplication algorithms. Karatsuba showed that the number of single-digit multiplications required in multiplying two two-digit numbers can be reduced from four (in the naive approach) to *three*, at the cost of a few additions and subtractions. This, combined with a divide-and-conquer approach, led to a significant improvement to integer multiplication. Then Toom described a 'generalized' approach of Karatsuba which further improved on the time complexity of integer multiplication by splitting the input integers into more parts and utilizing a polynomial interpretation; his approach was refined by Cook in 1966 and came to be known as the Toom-Cook algorithm. The next major improvement was by Schönhage and Strassen in 1971 ([SS71]), utilizing the Fast Fourier Transform over the integers $\pmod{2^n + 1}$. After this, improvements continued by Fürer, Harvey, and van der Hoeven which led to a galactic algorithm with complexity $\mathcal{O}(n \log n)$ in 2021 (we do not discuss these in this paper, but see [Für09],[HvdHL16],[HvdH21]).

In the realm of matrix multiplication, Volker Strassen published his Strassen algorithm in 1969 which proved that the normal, widely used matrix multiplication algorithm was in fact not optimal ([Str69]). Similar to Karatsuba, he showed that the number of multiplications required in multiplying two 2×2 matrices could be reduced from eight (in the naive approach) to only seven; this combined with a recursive divide-and-conquer method

Date: July 13, 2025.

led to an improvement in the time complexity of matrix multiplication algorithms. After this the next major improvement came with the Laser Method, introduced by Strassen in 1987 and optimized by Coppersmith and Winograd the same year, which brought down the time complexity to $\mathcal{O}(n^{2.38})$ ([CW90][Str88]). Further refinements continued, using the Laser Method, until very recent breakthroughs which introduced the Refined Laser Method and techniques such as asymmetric hashing ([LG14],[AW24],[ADW$^+$24],[WXXZ24]); however, recent improvements have only reduced the computational complexity in increments of one-hundredths or one-thousandths (to the exponent of n). DeepMind's groundbreaking AlphaTensor has also contributed to the optimization of matrix multiplication algorithms, by discovering a vast number of more algorithms as well as improvements to existing algorithms; for example, it found an algorithm for multiplying two 4×4 matrices using only 47 multiplications, an improvement from the 49 used via Strassen's algorithm ([FBH$^+$22]).

Besides integer and matrix multiplication, we will also go through some methods for complex number and polynomial multiplication, which are very similar to integer multiplication techniques. The paper finally ends with a discussion of applications of multiplication algorithms, further questions, and suggestions for further reading.

3. Multiplication in $\mathbb{Z}$, $\mathbb{C}$, and $\mathbb{Z}[x]$

3.1. **Karatsuba Algorithm.**

We first show that we can reduce the number of multiplications required to multiply two two-digit numbers. First, note that multiplying by a constant such as 10 or 2^{2n} will not be considered a true multiplication. These are usually cyclic shifts in the base we are using (and cyclic shifts are only linear time), or they are of much smaller length compared to the inputs in the algorithms; thus, they are considered negligible with respect to computational complexity. Secondly, note that the "number" of multiplications we refer to will be the number of *unique* multiplications needed. Also note that $\overline{a_n a_{n-1} \ldots a_0}$ denotes the quantity $10^n a_n + 10^{n-1} a_{n-1} + \ldots + 10 a_1 + a_0$.

To multiply the two integers $\overline{ab}$ and $\overline{cd}$, regular techniques would use four multiplications: $x_1 = a \cdot c$, $x_2 = b \cdot c$, $x_3 = a \cdot d$, and $x_4 = b \cdot d$. Then,

$$\overline{ab} \cdot \overline{cd} = 100x_1 + 10(x_2 + x_3) + x_4. \tag{3.1}$$

Karatsuba showed that we can define $y_1 = a \cdot c$, $y_2 = b \cdot d$, and $y_3 = (a + b)(c + d)$. Now $ad + bc = y_3 - y_1 - y_2$ so we can write

$$\overline{ab} \cdot \overline{cd} = 100y_1 + 10(y_3 - y_1 - y_2) + y_2. \tag{3.2}$$

Thus, we have reduced the number of multiplications from four to three at the cost of a few extra additions. With this base case, Karatsuba applies a divide-and-conquer algorithm. We define two $2n$-digit integers, $u = \overline{u_{2n-1} u_{2n-2} \ldots u_0}$ and $v = \overline{v_{2n-1} v_{2n-2} \ldots v_0}$. Then we split u into $p = \overline{u_{2n-1} \ldots u_n}$ and $q = \overline{u_{n-1} \ldots u_0}$ and v into $r = \overline{v_{2n-1} \ldots v_n}$ and $s = \overline{v_{n-1} \ldots v_0}$. Since $u = p10^n + q$ and $v = r10^n + s$, we see that

$$u \cdot v = pr\ 10^{2n} + ((p + q)(r + s) - pr - qs)\ 10^n + qs, \tag{3.3}$$

which only requires three multiplications. To summarize, Karatsuba's algorithm simplifies the multiplication of two n-digit numbers into three multiplications involving four $n/2$-digit numbers, which are split parts of the n-digit numbers. Then the multiplications involving the $n/2$-digit numbers are further simplified into multiplications involving $n/4$-digit numbers, and so on. This divide-and-conquer approach combined with the mathematical trick to reduce the number of multiplications results in a more time efficient approach. We also note that, if n is not a power of two or the two numbers being multiplied are of different lengths, we can simply "pad" our numbers by prepending zeroes to them.

Moreover, as multiplication algorithms are very useful in the context of computers, we can find the complexity of the Karatsuba algorithm. Let $T(n)$ denote the time required to multiply two $2n$-digit numbers. Then, with the algorithm described above, it can be seen that

$$T(n) = 3 \cdot T(\lceil n/2 \rceil) + \mathcal{O}(n). \tag{3.4}$$

Using the well-known master theorem for divide-and-conquer recurrences, we obtain

$$T(n) = \mathcal{O}(n^{\log_2 3}). \tag{3.5}$$

This is a clear improvement over the normal method of multiplying numbers, which is of complexity n^2, as $n^{\log_2 3} \approx n^{1.58}$.

Finally, we end our discussion of the Karatsuba algorithm with an example: performing the multiplication $1234 \cdot 567$. First, we must "pad" 567 with zeroes, making 0567, in order to match the length of 1234. Since both numbers are of length four, a power of two, we can now proceed with the algorithm. We split 1234 into 12 and 34 as well as 0567 into 05 and 67. So now we have

$$1234 \cdot 567 = (12 \cdot 05) \cdot 100^2 + ((12+34)(05+67) - 12 \cdot 05 - 34 \cdot 67) \cdot 100 + 34 \cdot 67.$$

Now we perform $12 \cdot 05$ with only three multiplications:

$$12 \cdot 05 = (1 \cdot 0) \cdot 10^2 + ((1+2)(0+5) - 1 \cdot 0 - 2 \cdot 5) \cdot 10 + 2 \cdot 5 = 60.$$

Similarly, we perform $34 \cdot 67$,

$$34 \cdot 67 = (3 \cdot 6) \cdot 10^2 + ((3+4)(6+7) - 3 \cdot 6 - 4 \cdot 7) \cdot 10 + 4 \cdot 7 = 2278,$$

and $(12+34) \cdot (05+67) = 46 \cdot 72$,

$$46 \cdot 72 = (4 \cdot 7) \cdot 10^2 + ((4+6)(7+2) - 4 \cdot 7 - 6 \cdot 2) \cdot 10 + 6 \cdot 2 = 3312.$$

We then plug these results back into our original multiplication to obtain

$$1234 \cdot 567 = 60 \cdot 100^2 + (3312 - 60 - 2278) \cdot 100 + 2278 = \boxed{699678}.$$

Below is a Python implementation of Karatsuba's algorithm, following the method outlined above:

```python
def karatsuba(x, y):
    if x<10 or y<10:
        return x * y
    maxl = max(len(str(x)), len(str(y)))
    maxl2 = maxl // 2
    p = x // (10 ** (maxl2))
    q = x % (10 ** (maxl2))
    r = y // (10 ** (maxl2))
    s = y % (10 ** (maxl2))

    a = karatsuba(q, s)
    b = karatsuba((p + q), (r + s))
    c = karatsuba(p, r)
    return c*(10**(2*maxl2)) + (b-a-c)*(10**(maxl2)) + a
```

3.2. **Toom-Cook Algorithm.**

The Toom-Cook algorithm is a 'generalized' version of Karatsuba, as it splits the input numbers into more parts. We begin with a discussion of Toom-3 and then discuss Toom-n. Toom-3 works much like Karatsuba, except that it splits the operands into *three* parts instead of two. Note that the lengths of these parts do not have to be all exactly equal; they can differ by at most 1.

Let x and y be the numbers we wish to multiply. Using the idea of Toom-3, we split x into x_2, x_1, and x_0; likewise, we split y into y_2, y_1, and y_0. Now an important idea comes into play: defining a polynomial in which the coefficients are the split parts. We let $X(t) = x_2t^2+x_1t+x_0$ and $Y(t) = y_2t^2 + y_1t + y_0$. Then our desired output is easily determined by $W(t) = X(t) \cdot Y(t) = w_4t^4 + w_3t^3 + w_2t^2 + w_1t + w_0$. Using regular polynomial multiplication and matching up the coefficients, we can find w_4, w_3, w_2, w_1, and w_0 in terms of x_2, x_1, x_0, y_2, y_1, and y_0 as

$$\begin{aligned}
w_4 &= x_2 \cdot y_2, \\
w_3 &= x_2 \cdot y_1 + x_1 \cdot y_2, \\
w_2 &= x_2 \cdot y_0 + x_1 \cdot y_1 + x_0 \cdot y_2, \\
w_1 &= x_1 \cdot y_0 + x_0 \cdot y_1, \text{ and} \\
w_0 &= x_0 \cdot y_0.
\end{aligned} \tag{3.6}$$

However, this would require 9 multiplications. Instead, we introduce a method that only requires 5 multiplications, at the cost of more additions and subtractions. We evaluate $W(t) = X(t) \cdot Y(t)$ at $t = 0, 1, -1, 2$, and ∞:

$$\begin{aligned}
w_0 &= x_0 \cdot y_0, \\
w_4 + w_3 + w_2 + w_1 + w_0 &= (x_2 + x_1 + x_0)(y_2 + y_1 + y_0), \\
w_4 - w_3 + w_2 - w_1 + w_0 &= (x_2 - x_1 + x_0)(y_2 - y_1 + y_0), \\
16w_4 + 8w_3 + 4w_2 + 2w_1 + w_0 &= (4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0), \text{ and} \\
w_4 &= x_2 \cdot y_2.
\end{aligned} \tag{3.7}$$

Then, this system of equations can be solved (since there are five unknown variables and five equations) to determine w_4, w_3, w_2, w_1, and w_0. First, note that, for the evaluation of $t = \infty$, we are simply using the leading coefficients; more rigorously, we are using

$$\begin{aligned}
w_4 &= \lim_{z\to\infty} \tfrac{W(z)}{z^4}, \\
x_2 &= \lim_{z\to\infty} \tfrac{X(z)}{z^2}, \\
y_2 &= \lim_{z\to\infty} \tfrac{Y(z)}{z^2}.
\end{aligned}$$

Also note that our new method only requires 5 multiplications, albeit more additions and subtractions. With this approach reducing the amount of multiplications, the Toom-Cook algorithm further utilizes a divide-and-conquer method to perform the 5 multiplications above. Since Toom-3 divides the input numbers into three parts through five multiplications, we see that its computational complexity is $\mathcal{O}(n^{\log_3 5}) \approx \mathcal{O}(n^{1.46})$. This is better than the Karatsuba algorithm, which we recall is around $\mathcal{O}(n^{1.58})$. However, as Toom-3 needs *many* more additions, subtractions, and multiplication by constants (more 'elementary' operations than multiplication) than Karatsuba, it only upholds an advantage for inputs of large lengths. Finally, to end our discussion of Toom-3, we provide an example.

Let's multiply $x = 123$ and $y = 456$ via the Toom-3 algorithm. Then we have $X(t) = t^2 + 2t + 3$ and $Y(t) = 4t^2 + 5t + 6$. Now we evaluate $W(t) = X(t) \cdot Y(t)$ at $t = 0, 1, -1, 2,$ and ∞:

$$\begin{aligned} w_0 &= 3 \cdot 6 = 18, \\ w_4 + w_3 + w_2 + w_1 + w_0 &= (1+2+3) \cdot (4+5+6) = 90, \\ w_4 - w_3 + w_2 - w_1 + w_0 &= (1-2+3)(4-5+6) = 10, \\ 16w_4 + 8w_3 + 4w_2 + 2w_1 + w_0 &= (4 \cdot 1 + 2 \cdot 2 + 3)(4 \cdot 4 + 2 \cdot 5 + 6) = 352, \text{ and} \\ w_4 &= 1 \cdot 4 = 4. \end{aligned} \tag{3.8}$$

Solving this system of equations, we obtain $w_0 = 18, w_1 = 27, w_2 = 28, w_3 = 13$, and $w_4 = 4$, so $W(t) = 4t^4 + 13t^3 + 28t^2 + 27t + 18$. Now we evaluate at $t = 10$ to get

$$x \cdot y = X(10) \cdot Y(10) = W(10) = 4 \cdot 10^4 + 13 \cdot 10^3 + 28 \cdot 10^2 + 27 \cdot 10 + 18 = \boxed{56088},$$

which is indeed correct.

Now we describe the Toom-n algorithm on the multiplication of two numbers, x and y. We split both x and y into n parts to obtain $X(t) = x_{n-1}t^{n-1} + \ldots + x_0$ and $Y(t) = y_{n-1}t^{n-1} + \ldots + y_0$. Then $W(t) = X(t) \cdot Y(t)$ is a degree $2n-2$ polynomial. The interpolation theorem shows that $k+1$ points are needed to determine a polynomial of degree k. Thus, we must evaluate our degree $2n-2$ polynomial W at $(2n-2)+1 = 2n-1$ points to obtain an expression for it. While any set of $2n-1$ t's would work, it is popular among implementations of the Toom-cook algorithm to use $t = 0, \infty, 1, -1$, and $\pm 2^i$, as some of the additions and subtractions needed are repeated and so running time is shortened. With this splitting and evaluation, Toom-n then runs a divide-and-conquer method to determine the $2n-1$ necessary multiplications. Overall, Toom-n splits multiplication of two numbers into n parts via $2n-1$ multiplications. Note that Toom-2 is simply the Karatsuba algorithm in section 4.1 and Toom-3 is the algorithm described above.

We can also find the computational complexity of Toom-n. If we let $T(N)$ be the time required to multiply two $N - digit$ numbers, then it can be seen that

$$T(N) = (2n-1) \cdot T(\frac{N}{n}) + \mathcal{O}(n). \tag{3.9}$$

Thus, we obtain

$$T(N) = \mathcal{O}(n^{\log_n 2n-1}). \tag{3.10}$$

For $n = 2, 3, 4$ we obtain $\mathcal{O}(n^{1.58})$, $\mathcal{O}(n^{1.46})$, and $\mathcal{O}(n^{1.40})$. We see that as n gets larger and larger, our computational complexity approaches $\mathcal{O}(n)$, since $\lim_{n\to\infty} \log_n (2n-1) = 1$. However, this becomes impractical for large numbers, as the number of additions, subtractions, and multiplications by constants (i.e., the constant term we have omitted in the computational complexity) grows unreasonably. At that point, we turn to FFT-based multiplication, as described in the next section. Thus, the Toom-Cook algorithm is practical only for intermediate-sized multiplication. Below is a Python implementation of the Toom-3 algorithm (running the code in an IDE for $x = 123$ and $y = 456$ returns 56088, which is indeed what we obtained in our above example):

```
def toom3(x, y):
    if x < 10 or y < 10:
        return x * y

    n_x = len(str(abs(x)))
    n_y = len(str(abs(y)))
    maxl = max(n_x, n_y)
    k = (maxl + 2) // 3  #digits per part (rounded up)

    base = 10**k
    base2 = base * base

    x2 = x // base2
    x1 = (x // base) % base
    x0 = x % base
    y2 = y // base2
    y1 = (y // base) % base
    y0 = y % base

    a = toom3(x2, y2) # t -> infinity
    b = toom3(4*x2 + 2*x1 + x0, 4*y2 + 2*y1 + y0) # t = 2
    c = toom3(x2 - x1 + x0, y2 - y1 + y0) # t = -1
    d = toom3(x2 + x1 + x0, y2 + y1 + y0) # t = 1
    e = toom3(x0, y0) # t = 0

    w4 = a
    w2 = (c + d - 2*a - 2*e) // 2
    w3 = (b - 16*a - 4*w2 - d + c - e) // 6
    w1 = (d - c) // 2 - w3
    w0 = e

    return (w4 * base**4) +
           (w3 * base**3) +
           (w2 * base**2) +
           (w1 * base) +
           w0
```

3.3. **Schönhage-Strassen Algorithm.**

Before introducing the actual Schönhage-Strassen algorithm, we discuss FFT, a.k.a. the Fast Fourier Transform. In addition to being used in multiplication algorithms, FFT can be applied in various other fields such as error-correcting codes and signal processing. Its most popular implementation, introduced by Cooley and Tukey in 1965, requires $\mathcal{O}(n \log n)$ operations. Many languages already have FFT implemented, and it can be called with a simple method imported from a library.

So what is FFT? FFT is an algorithm to efficiently evaluate a polynomial. Let us evaluate $A(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ at n points, $x_0, \ldots x_{n-1}$. We can write these n evaluations as a matrix multiplication:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \ldots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} x_0^0 & x_0^1 & \ldots & x_0^{n-1} \\ x_1^0 & x_1^1 & \ldots & x_1^{n-1} \\ \ldots \\ x_{n-1}^0 & x_{n-1}^1 & \ldots & x_{n-1}^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{bmatrix}. \tag{3.11}$$

Now we also define ω_n to be a primitive nth root of unity in a finite field if $\omega_n^i \neq 1$ for $0 < i < n$ and $\omega_n^n = 1$. For example, in $\mathbb{Z}_{37}$, the set of all integers modulo 37, 2 is a primitive 36th root of unity and in $\mathbb{Z}_5$, the set of all integers modulo 5, 4 is a 2nd root of unity since $4^2 \equiv 1 \pmod 5$. Now we plug into equation 3.11 values for $x_0, \ldots, x_{n-1}$ as $x_0 = \omega^0, \ldots, x_i = \omega^i, \ldots, x_n = \omega^n$, where ω is a primitive nth root of unity. Thus, we get

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \ldots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 1 & \omega & \ldots & \omega^{n-1} \\ \ldots \\ 1 & \omega^{n-1} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{bmatrix}. \tag{3.12}$$

When the Fourier Transform is used outside of mathematics, typically in engineering fields, ω is a primitive nth root of unity in the complex plane, i.e. $\omega = e^{2\pi i/n}$. Now to make this a *Fast* Fourier Transform, we apply an algorithm introduced by Cooley and Tukey [CT65].

If n is an even number, $n = 2m$, then we can split $A(x)$ into the even-indexed and odd-indexed coefficients. Let $A_0(x) = \sum_{i=0}^{m-1} a_{2i}x^i$ and $A_1(x) = \sum_{i=0}^{m-1} a_{2i+1}x^i$. Then we see that $A(x) = A_0(x^2) + xA_1(x^2)$. Now we can rewrite the left-hand side of equation 3.12 as

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \ldots \\ A(\omega^m) \\ \ldots \\ A(\omega^{2m-1}) \end{bmatrix} = \begin{bmatrix} A_0(1) + A_1(1) \\ A_0(\omega^2) + \omega A_1(\omega^2) \\ \ldots \\ A_0(1) + \omega^m A_1(1) \\ \ldots \\ A_0(\omega^{2m-2}) + \omega^{2m-1}A_1(\omega^{2m-2}) \end{bmatrix}, \tag{3.13}$$

using the fact that $\omega^{2m} = \omega^n = 1$ by our definitions of n and ω. Equation 3.13 can be further simplified to

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \ldots \\ A(\omega^m) \\ \ldots \\ A(\omega^{2m-1}) \end{bmatrix} = \begin{bmatrix} A_0(1) \\ A_0(\omega^2) \\ \ldots \\ A_0(1) \\ \ldots \\ A_0(\omega^{2m-2}) \end{bmatrix} + \begin{bmatrix} A_1(1) \\ \omega A_1(\omega^2) \\ \ldots \\ \omega^m A_1(1) \\ \ldots \\ \omega^{2m-1}A_1(\omega^{2m-2}) \end{bmatrix}. \tag{3.14}$$

The red term is simply the $2m$ Fourier Transform of A. The blue term is the m Fourier Transform of A_0 duplicated twice vertically. The orange term, besides the constant w^i's, is the k Fourier Transform of A_1 duplicated twice vertically. Subsequently, the smaller transforms can be computed recursively. Thus, FFT splits the Fourier Transform into two smaller transforms and more additions.

We can also find the complexity of FFT. Specifically, if $T(n)$ is the running time for a degree n polynomial, we obtain

$$T(n) = T(n/2) + \mathcal{O}(n), \tag{3.15}$$

because of the splitting and additional $\mathcal{O}(n)$ additions. Then we obtain the running time of

$$T(n) = \mathcal{O}(n \log_2 n). \tag{3.16}$$

Below is a Python implementation of Fast Fourier Transform, following the method outlined above:

```
def fft(a): #where a is the list of coefficients of A
    n = len(a) #degree of A
    if n <= 1:
        return a
    a_0 = fft(a [::2]) #even indices
    a_1 = fft(a [1::2]) #odd indices
    b = [0] * n
    for i in range (0, n//2):
        b[i] = a_0[i] + root_of_unity(n, i) * a_1[i]
        b[i+n//2] = a_0[i] + root_of_unity(n, i+n//2) * a_1[i]
    return b
```

A second component necessary for completing the Schönhage-Strassen algorithm is the concept of the inverse FFT. In other words, if we are given the n evaluations $A(1), A(\omega), \ldots,$ and $A(\omega^{n-1})$, then can we compute A, and how fast can we do this? The answer is *yes*, it can in fact be done in the same $\mathcal{O}(n \log_2 n)$ time. Let $B(x) = \sum_{i=0}^{n-1} A_i x^i$ where $A_i = A(\omega^i)$. For the inverse FFT, we evaluate $B(x)$ at $x = \omega^j$ for some j in the range $[0, n-1]$. Then we get

$$B(\omega^j) = \sum_{i=0}^{n-1} A_i \omega^{ij} = \sum_{i=0}^{n-1} (\sum_{m=0}^{n-1} a_m \omega^{km}) \omega^{ij} = \sum_{m=0}^{n-1} a_m \sum_{i=0}^{n-1} \omega^{i(j+k)}. \tag{3.17}$$

We can use the geometric series sum formula to get that

$$\textstyle\sum_{i=0}^{n-1} \omega^{i(j+k)} = \begin{cases} n & j+k \equiv 0 \pmod{n} \\ \frac{\omega^{n(j+k)}-1}{\omega^{j+k}-1} = 0 & j+k \not\equiv 0 \pmod{n} \end{cases}$$

Therefore, using this fact, equation 3.17 becomes

$$B(\omega^j) = n \cdot a_{n-j}. \tag{3.18}$$

We can then scale this result by multiplying by $\frac{1}{n}$ and then reversing it (so that the order is reverted back) for all such j to finally get the coefficients of our original polynomial $A(x)$. Thus, the inverse FFT of the FFT of $A(x)$ returns $A(x)$ – an important property of FFT. Moreover, since inverse FFT is simply another FFT (of $\mathcal{O}(n \log_2 n)$) and more multiplications (of $\mathcal{O}(n)$, it bears the same complexity of $\mathcal{O}(n \log_2 n)$. From now on in our discussion, we will use the name NTT, standing for number-theoretic transform, to refer to FFT over integers (as the more widely used FFT refers to the transform over complex numbers).

The final idea that we need for the Schönhage-Strassen algorithm is the idea of convolutions, both *positive-wrapped* and *negative-wrapped*. But first, what even is a convolution?

Suppose that $A(x)$ and $B(x)$ are degree $n-1$ polynomials in the ring $\mathbb{Z}_q[x]$ (as a reminder, $\mathbb{Z}_q$ is the set of integers modulo q, or just $0, 1, 2, \ldots, q-1$; also, $\mathbb{Z}_{||}[x]$ is the set of all polynomials where the coefficients are members of the ring $\mathbb{Z}_q$). Then polynomial multiplication gives

$$C(x) = A(x) \cdot B(x) = \sum_{i=0}^{2n-2} c_i x^i, \tag{3.19}$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}. \tag{3.20}$$

We define the **convolution** of the coefficient vectors of the polynomials $A(x)$ and $B(x)$ as

$$a * b = \sum_{i=0}^{n} a_i b_{n-i}. \tag{3.21}$$

Then, from equations 3.19 and 3.20, we simply obtain

$$a * b = c, \tag{3.22}$$

which is the coefficient vector of the output polynomial $C(x)$. Now, for a **positive wrapped convolution**, we also have the restriction that $A(x)$ and $B(x)$ must be in the ring $\mathbb{Z}_q[x]/(x^n - 1)$ (as a reminder, this denotes the set of polynomials in $\mathbb{Z}_q[x]$ taken modulo $x^n - 1$). Then the positive wrapped convolution, denoted as the polynomial C^+, is defined as

$$C^+(x) = \sum_{i=0}^{n-1} c_i x^i \tag{3.23}$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \pmod{q}. \tag{3.24}$$

Specifically,

$$C^+(x) \equiv C(x) \pmod{x^n - 1}. \tag{3.25}$$

We similarly define the **negative wrapped convolution**, sometimes called the *negacylic* convolution, in the ring $\mathbb{Z}_q[x]/(x^n + 1)$, as

$$C^-(x) = \sum_{i=0}^{n-1} c_i x^i, \tag{3.26}$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} \ - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \pmod{q}. \tag{3.27}$$

Specifically,

$$C^-(x) \equiv C(x) \pmod{x^n + 1}. \tag{3.28}$$

We can use NTT and the inverse NTT, denoted as NTT^{-1} to calculate positive wrapped convolutions as

$$c = \mathrm{NTT}^{-1}(\mathrm{NTT}(a) \otimes \mathrm{NTT}(b)), \tag{3.29}$$

where the $\otimes$ is an element-wise vector multiplication; for example, $[a_1, \ldots, a_n] \otimes [b_1, \ldots, b_n] = [a_1 b_1, \ldots, a_n b_n]$. However, Schoönhage and Strassen aim to find the negacyclic convolution. They do this with a special trick: 'weighting' the coefficients of $A(x)$ and $B(x)$ prior to applying a positive wrapped convolution. Define θ as a primitive $2n$th root of unity, where $\theta^n = -1$ and $\theta^2 = \omega$ (using our previous definition of ω as the primitive nth root of unity). Then we weight the coefficients as

$$a'_i = \theta^i a_i \text{ and } b'_i = \theta^i b_i, \tag{3.30}$$

for all i in $[0, n-1]$. Then we apply the positive wrapped convolution with a 'counterweight', multiplying each c_k with θ^k, as follows:

$$\begin{aligned} c_k &= \theta^{-k}\Big(\sum_{i+j=k} a'_i b'_j + \sum_{i+j=n+k} a'_i b'_j\Big) \\ &= \theta^{-k} \sum_{i+j=k} \theta^{i+j} a_i b_j + \theta^{-k} \sum_{i+j=n+k} \theta^{i+j} a_i b_j \\ &= \theta^{-k} \sum_{i+j=k} \theta^{k} a_i b_j + \theta^{-k} \sum_{i+j=n+k} \theta^{n+k} a_i b_j \\ &= \sum_{i+j=k} a_i b_j + \theta^n \sum_{i+j=n+k} a_i b_j \\ &= \sum_{i+j=k} a_i b_j - \sum_{i+j=n+k} a_i b_j, \end{aligned} \tag{3.31}$$

which is, according to our definition, the negacyclic convolution! We also note that, if c_k is negative, we can add multiples of $2^n + 1$ to get c_k into the range $[0, 2^n]$.

We quickly sidetrack to a discussion on finding primitive nth roots of unity. We can then choose a prime $p = kn + 1$ where $k = 2^i$ for some i. We subsequently work in the prime field $\mathbb{Z}_p$. Then, to find a nth root of unity, we randomly pick any q then set $r = q^k$. r is an nth root of unity because $r^n = q^{kn} = q^{p-1} \equiv 1 \pmod{p}$, where the last step follows by Fermat's little theorem. However, r is not always primitive. If $q^{k/2} \neq 1$ then $q^k = r$ is a primitive nth root of unity; however, if $q^{k/2} = 1$, then we can apply this method again (i.e. check if $q^{k/4}$ is 1, and so on). This is a fast method of finding primitive nth roots of unity; usually, however, computers will already have hard-coded lists of primitive roots up to large n, so this is unnecessary. Moreover, if we desire a $2n$th root of unity in $\mathbb{Z}_{2^n+1}$ for θ, we see that 2 is a possible value, since $2^n \equiv -1 \pmod{2^n+1}$ and $2^{2n} \equiv (2^n-1)(2^n+1)+1 \equiv 1 \pmod{2^n+1}$. This is very helpful, since multiplications by power of two in the equations above can simply be done by bit shifts (i.e. multiplying by 4 is equivalent to appending 00 to the end of a binary number) and some modular reductions (i.e. subtracting/adding by a multiple of $2^n + 1$), only taking linear $\mathcal{O}(n)$ time.

Now we wish to apply FFT in order to achieve a faster integer multiplication algorithm. Suppose we are multiplying x and y. The Schönhage-Strassen algorithm splits x and y into parts, transforms them into the coefficients of a polynomial, evaluates the polynomial, multiplies these values, and interpolates back to find the product polynomial, yielding the final product $x \cdot y$. Furthermore, this algorithm applies FFT in the field $\mathbb{Z}_{2^n+1}$. Note that n must be chosen appropriately such that it can store the convolution sums and ultimately the final product. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. r will the size of our FFT we use in the procedure. Now, the algorithm is as follows:

(1) Split x and y into r parts, each of length m, and store these into two arrays, which can be thought of as coefficients of two r degree polynomials.
(2) Weight both coefficient vectors with the powers of θ as described above – cyclic shifts makes this process faster.
(3) Perform NTT on the coefficient vectors – powers of ω are also cyclic shifts here.
(4) Multiply the coefficient vectors as defined previously, with $\otimes$. These multiplications are performed recursively, via this algorithm again, alike to a divide-and-conquer approach.
(5) Perform NTT^{-1} on the c coefficient vector.
(6) Apply the 'counterweight' discussed previously, θ^{-k}.
(7) Multiply the c coefficient vector by $\frac{1}{r}$, as discussed with the inverse NTT previously. Since $r = 2^m$, multiplying by $\frac{1}{r} = 2^{-m}$ is simply a cyclic shift.
(8) Reconstruct the integer product from the coefficient vector by adding and carrying, or in other words evaluate $C(x)$ at $x = 10$.

Note that, throughout this process, the sequence of the coefficients in the coefficient vector should be handled carefully.

We will now find the runtime of the Schönhage-Strassen algorithm – the main reason why it is very useful for larger numbers. Let $T(n)$ denote the time taken to multiply two n-digit numbers. Then we have that

$$T(n) = \mathcal{O}(n \log_2 n) + n \cdot \mathcal{O}(\log_2 n \log_2 \log_2 n), \tag{3.32}$$

ignoring the lower order $\mathcal{O}(n)$ time operations. $\mathcal{O}(n \log_2 n)$ arises from the FFT multiplication in the multiplication of the two n-digit numbers. Then the $n \cdot \mathcal{O}(\log_2 n \log_2 \log_2 n)$ arises from the n pointwise multiplications of pairwise integers, which are approximately of length $\log_2 n$. Equation 3.32 then turns into

$$T(n) = \mathcal{O}(n \cdot \log_2 n \cdot \log_2(\log_2 n)). \tag{3.33}$$

A more detailed analysis can also be found at [Lüd15]. Schönhage and Strassen in fact predicted that this time complexity could be reduced to simply $\mathcal{O}(n \log_2 n)$. This was almost achieved by Harvey and van der Hoeven with a galactic algorithm [HvdH21].

3.4. **Complex Number Multiplication.**

In this section we show a faster method of multiplying two complex numbers. Suppose we wish to perform the multiplication $(a + bi)(c + di)$. Usual multiplication techniques, i.e. the FOIL method, would use four multiplications and two additions: $(a + bi)(c + di) = ac - bd + (bc + ad)i$. However, as you may have guessed, we can simplify the number of multiplications to three. Define $r = c(a + b), s = a(d - c)$, and $t = b(c + d)$. Then, we have that

$$\begin{aligned} (r - t) + (r + s)i &= (c(a + b) - b(c + d)) + (c(a + b) + a(d - c))i \\ &= (ac - bd) + (bc + ad)i \\ &= (a + bi)(c + di). \end{aligned} \tag{3.34}$$

The computation of $(r - t) + (r + s)i$ requires three multiplications and five additions. There are other possibilities also to simplify the number of multiplications to three. Take for example the following. Define $x = ac, y = bd$, and $z = (a+b)(c+d)$. Then we have that

$$\begin{aligned}(x-y)+(z-x-y)i &= (ac-bd)+((a+b)(c+d)-ac-bd)i \\ &= (ac-bd)+(bc+ad)i \\ &= (a+bi)(c+di).\end{aligned} \tag{3.35}$$

This also uses five additions/subtractions. A third interesting method is also possible as follows. We assume that $a^2+b^2=1$. We can then define $t = \frac{1-a}{b}$ and the three variables $x = c - td, y = d + bx$, and $z = x - ty$. Then we can see that

$$\begin{aligned}z + yi &= (x-ty)+(d+bx)i \\ &= ((c-td)-t(d+b(c-td)))+(d+b(c-td))i \\ &= (c-2td-tbc+t^2bd)+(d+bc-tbd)i \\ &= (ac-bd)+(bc+ad)i \\ &= (a+bi)(c+di),\end{aligned} \tag{3.36}$$

where the second to last step follows from numerous algebraic manipulations and use of our assumption that $a^2+b^2=1$. This algorithm requires three multiplications. The calculation of t is not actually a multiplication, since if we interpret a and b as $\cos\theta$ and $\sin\theta$, respectively, then $t = \frac{1-a}{b} = \tan\frac{\theta}{2}$ via trigonometric identities. While this algorithm has only three multiplications, its assumption that $a^2+b^2=1$ limits its use. Generalizing this to an $a+bi$ of any norm (i.e. no restrictions on a^2+b^2), we would need to multiply by this norm to our final result. For example, if $a = 3\cos\theta$ and $b = 4\sin\theta$, then we would need to compute $\sqrt{a^2+b^2} = 5$ to perform the multiplication via this algorithm. The calculation of the norm would make this algorithm longer than those previously suggested in equations 4.19 and 4.20.

3.5. **Polynomial Multiplication.**

Techniques for polynomial multiplication follow from the previous Karatsuba, Toom-Cook, and other algorithms since we interpret our input numbers as polynomials. Thus the same methods of splitting recursively and solving a system of equations or interpolating to obtain the final result work. We will discuss another method for polynomial multiplication, Kronecker substitution. This technique reduces polynomial multiplication to a potentially simpler problem of multiplying two integers. If we are given $p(x)$ and $q(x)$ and wish to find $r(x) = p(x)\cdot q(x)$, we can input a large enough power of two for x such that we can simply read off the coefficients of $r(x)$ from the binary product. In base ten, we could likwise substitute a sufficiently large power of ten. For example, if we had $p(x) = 12x^2+34x+56$ and $q(x) = 98x^2+76x+54$, we can compute the product $p(10^4)\cdot q(10^4)$, which is just $r(10^4)$, as follows:

$$12|0034|0056 \;\cdot\; 98|0076|0054 = 1176|4244|8720|6092|3024.$$

Note that using 10^3 or 10^2 results in overlap in the product, so then the result is indecipherable. The numbers are also marked with bars every four digits from the right for clarity. Finally, we simply read off the answer: $r(x) = 1176x^4+4244x^3+8720x^2+6092x+3024$.

A "multipoint" Kronecker substitution has been introduced by Harvey [Har09]. Instead of evaluating at only point, like 10^4 in the above example or just one power of two, we evaluate at multiple points. If the normal Kronecker substitution evaluates the polynomials at 10^z, then a two-point Kronecker substitution would evaluate them at $\pm 10^{z'}$, where $z' = \lceil \frac{z}{2} \rceil$. Continuing the example from above, we can evaluate at $\pm 10^2$ to get

$$r(10^2) = p(10^2) \cdot q(10^2) = 123456 \cdot 987654 = 121931812224 \text{ and}$$
$$r(-10^2) = p(-10^2) \cdot q(-10^2) = 116656 \cdot 972454 = 113442593824.$$

Then we simply do

$$\tfrac{r(10^2)+r(-10^2)}{2} = 1176|8720|3024 \text{ and}$$
$$\tfrac{r(10^2)-r(-10^2)}{2} = 4244|6092|00.$$

We have marked the splitting into the numbers. We can then combine these two to obtain 1176|4244|8720|6092|3024 as our product, from which we read off the answer: $r(x) = 1176x^4 + 4244x^3 + 8720x^2 + 6092x + 3024$. This method works because inputting the positive and negative values produces one evaluation in which the coefficients are all added together and one evaluation in which the coefficients are added and subtracted in an alternating fashion. Adding these two evaluations and dividing by two then results in the coefficients of x to the power of an even number while subtracting the two evaluations and dividing by two results in the coefficients of x to the power of an odd number. Superimposing these two thus gives us our entire product polynomial. Harvey also proposes evaluating at $10^{\pm z'}$ and even a "four-point" Kronecker substitution by evaluating at $\pm 10^{\pm z'}$.

4. Matrix Multiplication

4.1. **Strassen Algorithm.**

The Strassen algorithm is a method to optimize the multiplication of two matrices. From here on, we will define our matrix multiplication problem to be $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$, where $\mathbf{A}$ and $\mathbf{B}$ are square matrices of size 2^n. Note that, if given matrices where the dimensions are not powers of two, we can simply 'pad' the remaining rows and columns with zeroes to obtain the desired dimensions. We will first show how the number of multiplications required for the multiplication of two 2×2 matrices can be reduced. Define

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \text{ and } \mathbf{C} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

Then the "naive" algorithm would compute C as

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$$
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}. \tag{4.1}$$

We can see that this requires 8 multiplications. The complexity of such an algorithm would be $\mathcal{O}(n^3)$.

The Strassen algorithm, much like Karatsuba, introduces seven variables which can be calculating in 7 multiplications and then manipulated with some extra additions and subtractions to obtain the desired result. The variables defined are

$$\begin{aligned} X_1 &= (A_{11}+A_{22})(B_{11}+B_{22}),\\ X_2 &= B_{11}(A_{21}+A_{22}),\\ X_3 &= A_{11}(B_{12}-B_{22}),\\ X_4 &= A_{22}(B_{21}-B_{11}),\\ X_5 &= B_{22}(A_{11}+A_{12}),\\ X_6 &= (A_{21}-A_{11})(B_{11}+B_{12}),\\ X_7 &= (A_{12}-A_{22})(B_{21}+B_{22}). \end{aligned} \tag{4.2}$$

Then we see that

$$\begin{aligned} \begin{bmatrix} C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix} &= \begin{bmatrix} A_{11}\cdot B_{11}+A_{12}\cdot B_{21} & A_{11}\cdot B_{12}+A_{12}\cdot B_{22}\\ A_{21}\cdot B_{11}+A_{22}\cdot B_{21} & A_{21}\cdot B_{12}+A_{22}\cdot B_{22}\end{bmatrix}\\ &= \begin{bmatrix} X_1+X_4-X_5+X_7 & X_3+X_5\\ X_2+X_4 & X_1-X_2+X_3+X_6\end{bmatrix} \end{aligned} \tag{4.3}$$

using the variables defined in equation 4.2. It can be counted that this method uses 7 multiplications and 18 addition/subtractions. Using this trick, the Strassen algorithm employs a divide-and-conquer strategy which subdivides the input matrices until reaching this base case of 2×2, performs the multiplications, and then reconstructs the final $\mathbf{C}$. Also note that if we had padded $\mathbf{A}$ and/or $\mathbf{B}$ with rows/columns of zeroes, we would delete those in the $\mathbf{C}$ obtained. We can analyze the computational complexity of the Strassen algorithm. If $T(n)$ is the time required to multiply two $2^n \times 2^n$ square matrices, then since the Strassen algorithm is performing seven multiplications for every division by two, we obtain

$$T(n) = \mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.807}). \tag{4.4}$$

However, note that for small enough matrices the n^3 approach is more efficient; the cutoff point between these two algorithms depends on numerous factors though, including hardware efficiency and code optimization.

Winograd (see [Knu98],[Win71]) reduces the number of additions/subtractions needed from 18 to 15 by defining a different set of variables:

$$\begin{aligned} Y_1 &= A_{11}B_{11},\\ Y_2 &= (A_{21}-A_{11})(B_{12}-B_{22}),\\ Y_3 &= (A_{21}+A_{22})(B_{12}-B_{11}),\\ Y_4 &= Y_1 + (A_{21}+A_{22}-A_{11})(B_{11}+B_{22}-B_{21}). \end{aligned} \tag{4.5}$$

Then we see that

$$\begin{aligned} &\mathbf{C}\\ &= \begin{bmatrix} A_{11}\cdot B_{11}+A_{12}\cdot B_{21} & A_{11}\cdot B_{12}+A_{12}\cdot B_{22}\\ A_{21}\cdot B_{11}+A_{22}\cdot B_{21} & A_{21}\cdot B_{12}+A_{22}\cdot B_{22}\end{bmatrix}\\ &= \begin{bmatrix} Y_1+A_{12}\cdot B_{21} & Y_3+Y_4+B_{22}(A_{11}+A_{12}-A_{21}-A_{22})\\ Y_2+Y_4+A_{22}(B_{12}+B_{21}-B_{11}-B_{22}) & Y_2+Y_3+Y_4\end{bmatrix} \end{aligned} \tag{4.6}$$

using the variables defined in equation 4.5. Note that this only requires 15 additions *because* we can store intermediate calculations. For example, the calculation of $A_{11}+A_{12}-A_{21}-A_{22}$ only actually requires one addition since it is equal to $A_{12}-(A_{21}+A_{22}-A_{11})$. $A_{21}+A_{22}-A_{11}$ was already calculated in the calculation of Y_4, so we need only perform its subtraction from A_{12}. Furthermore, the minimum number of multiplications in the multiplication of two 2×2 matrices was proven by Winograd (again, see [Win71]) to be 7 – this fact will be useful later on, in section 4.2.

Below is a Python implementation of the Strassen algorithm, following the 18 addition method outlined above, which only works for $\mathbf{A}$ and $\mathbf{B}$ which have dimensions of powers of two (implementations for different dimensions need to include code which pads with zeroes, as discussed previously):

```python
def strassen(A, B):
    n=len(A)
    if n<3:
        return numpy.dot(A, B)

    #split A and B into four submatrices each
    A_11 = A[:n//2, :n//2]
    A_12 = A[:n//2, n//2:]
    A_21 = A[n//2:, :n//2]
    A_22 = A[n//2:, n//2:]
    B_11 = B[:n//2, :n//2]
    B_12 = B[:n//2, n//2:]
    B_21 = B[n//2:, :n//2]
    B_22 = B[n//2:, n//2:]

    #define our seven variables
    M_1 = strassen(A_11 + A_22, B_11 + B_22)
    M_2 = strassen(B_11, A_21 + A_22)
    M_3 = strassen(A_11, B_12 - B_22)
    M_4 = strassen(A_22, B_21 - B_11)
    M_5 = strassen(A_11 + A_12, B_22)
    M_6 = strassen(A_21 - A_11, B_11 + B_12)
    M_7 = strassen(A_12 - A_22, B_21 + B_22)

    #find submatrices of C
    C_11 = M_1+M_4-M_5+M_7
    C_12 = M_3+M_5
    C_21 = M_2+M_4
    C_22 = M_1-M_2+M_3+M_6
    C = numpy.vstack((numpy.hstack((C_11, C_12)),
                      numpy.hstack((C_21, C_22))))
    return C
```

4.2. **Laser Method.**

We start with two $n \times n$ matrices, A and B. We define $C = A \cdot B$ with $C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$. Moreover, we will refer to the exponent of n in the time complexity of the algorithms. If the time complexity is $\mathcal{O}(n^{\omega})$, we will just refer to ω now. Note that there is a trivial lower bound on ω, namely $\omega \geq 2$. Any algorithm for multiplying the two $n \times n$ matrices must at least go through all $2n^2$ entries in both matrices, thus the lower bound of $\mathcal{O}(n^2)$. For reference, we provide a timeline of improvement in bounds for ω (along with the years in which these discoveries were made, by whom, and with what methods):

Year	Author(s)	ω	Note
1969	Strassen	2.8074	First Sub-Cubic Algorithm
1978	Pan	2.795	
1981	Schönhage	2.522	
1987	Strassen; Coppersmith, Winograd	2.3755	Laser Method and CW tensor
2010	Stothers	2.3737	$CW^{\otimes 4}$
2011	Williams	2.372873	$CW^{\otimes 8}$
2014	Le Gall	2.372864	$CW^{\otimes 32}$
2020	Alman, Williams	2.3728596	$CW^{\otimes 32}$ with Refined Laser Method
2022	Duan, Wu, Zhou	2.371866	Asymmetric Hashing
2024	Williams et al.	2.371552	
2024	Alman et al.	2.371339	More Asymmetry

An important component of the improvements after the Strassen algorithm utilize **tensors**. Tensors can be thought of as hypermatrices, bilinear maps, trilinear maps, and multilinear polynomials. Here we use the last form, specifically representing our tensor T as a trilinear polynomial. If we have the sets $X = \{x_1, x_2, \dots, x_n\}$, $Y = \{y_1, y_2, \dots, y_n\}$, and $Z = \{z_1, z_2, \dots, z_n\}$, then we define T as

$$T = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} q_{ijk} \cdot x_i \cdot y_j \cdot z_k. \tag{4.7}$$

Matrix Multiplication as a Tensor. We can represent the multiplication of two $n \times n$ matrices as a tensor. We use the notation $\langle n, n, n \rangle$ to define this tensor. Then we have that

$$T = \langle n, n, n \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} x_{ij} y_{jk} z_{ki}, \tag{4.8}$$

where x_{ij} is in matrix **A** and y_{jk} is in matrix **B**. Then the coefficient of z_{ki} is just the entry at position (i, k) in matrix **C**. This also generalizes to the multiplication of an $n \times m$ and $m \times p$ matrix, where we represent T by $\langle n, m, p \rangle$ and sum from 1 to n, 1 to m, and 1 to p.

Tensor Rank and the Bound on ω. Now we define tensor rank, which is like the 'complexity' of a tensor. A rank **one** tensor is a tensor which can be expressed as a product of linear combinations of the x, y, and z variables. In other words, the trilinear form of a tensor S from equation 4.7 can be split into the product of three linear polynomials, as

$$S = (\sum_{i=1}^{n} a_i x_i)(\sum_{j=1}^{n} b_j y_j)(\sum_{k=1}^{n} c_k z_k) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} a_i b_j c_k x_i y_j z_k. \tag{4.9}$$

Then the *rank* of a tensor, $R(T)$, is the minimum number of rank one tensors that sum to T (this is also analogous to the rank of matrices). If $R(T) = r$, we also have the following lemma:

Lemma 4.1. *If $R(\langle n, n, n\rangle) = r$, then $\omega \leq \log_n r$.*

Proof. We first express $\langle n, n, n\rangle$ as a sum of r rank one tensors, by our definition of tensor rank:

$$\begin{aligned}\langle n, n, n\rangle &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} x_{ij}y_{jk}z_{ki} \\ &= \sum_{p=1}^{r}((\sum_{i',j} a_{pi'j}x_{i'j})(\sum_{j',k'} b_{pj'k'}x_{j'k'})(\sum_{k,i} c_{pki}z_{ki})).\end{aligned} \tag{4.10}$$

Now we also use the following recursive algorithm: split each $n\times n$ matrix into four $n/2\times n/2$ matrices. Then we *recursively* compute $S_p = (\sum_{i',j} a_{pi'j}x_{i'j})(\sum_{j',k'} b_{pj'k'}x_{j'k'})$ for all $1 \leq p \leq r$. Then the coefficient of z_{ki} is just $\sum_{p=1}^{r} c_{pki}S_p$. Computing the linear combinations $\sum_{i',j} a_{pi'j}x_{i'j}$ and $\sum_{j',k'} b_{pj'k'}x_{j'k'}$ can be done in linear time. $\sum_{p=1}^{r} c_{pki}S_p$ is also just a linear combination, done easily in linear time. The recursive steps give us the $\mathcal{O}(\log_n r)$ time. ■

Note that for $n = 2$, Strassen showed that $R(\langle 2, 2, 2\rangle) \leq 7$, giving $\omega \leq \log_2 7 \approx 2.8074$, exactly what we got in section 4.1! Pan, in his 1978 paper, showed that $R(\langle 2, 2, 2\rangle) \leq 143640$, giving $\omega \leq \log_{70} 143640 \approx 2.795$.

Zeroing Out. One main idea of the laser method is to start with a tensor for which we know a decent rank bound, then reduce our matrix multiplication tensor to that tensor. This can be done via *zeroing out*. For example, if we had the tensor

$$P = (x_{11}y_{11} + x_{12}y_{21})z_{11} + (x_{11}y_{12} + x_{12}y_{22} + x_{33}y_{33})z_{12} + (x_{21}y_{11} + x_{22}y_{21})z_{21} + \\ (x_{21}y_{12} + x_{22}y_{22})z_{22} + (x_{11}y_{11} + x_{22}y_{22})z_3,$$

then we can 'zero out' some variables, e.g. set x_{33} and z_3 to 0. This gives us the tensor

$$P' = (x_{11}y_{11} + x_{12}y_{21})z_{11} + (x_{11}y_{12} + x_{12}y_{22})z_{12} + (x_{21}y_{11} + x_{22}y_{21})z_{21} + (x_{21}y_{12} + x_{22}y_{22})z_{22},$$

which is in fact just our matrix multiplication tensor for $n = 2$.

Direct Sums of Tensors. Now we introduce another critical component, *direct sums.* Let $T = x_1y_2z_1 + x_2y_1z_2$. Then the direct sum of two copies of T is the sum of T for independent variable sets (i.e. the copies of T cannot have overlap of variables). The variable sets of T are x_1, x_2 and y_1, y_2. Now the second copy of T must shifted over to the variable sets x_3, x_4 and y_3, y_4 as $x_3y_4z_3 + x_4y_3z_4$. Then the direct sum of two copies of T, which we will denote now by A, is $A = x_1y_2z_1 + x_2y_1z_2 + x_3y_4z_3 + x_4y_3z_4$.

Let B denote the direct sum of k copies of T. Then $R(B) \leq k \cdot R(T)$. This is because T is expressed as the sum of $R(T)$ rank one tensors and so B is expressed as the sum of $k \cdot R(T)$ rank one tensors. If it is possible to combine some of these rank one tensors to somehow still yield a rank one tensor, that would only decrease the rank of B, so $R(B)$ has an upper bound of $k \cdot R(T)$. If $R(B) = r$, then $R(T) \geq \frac{r}{k}$. Applying this to Lemma 4.1, we obtain $\omega \leq \log_n \frac{r}{k}$. This is an important result and is a simplified version of Schönhage's τ-theorem (see [Sch81]).

Products of Tensors and Asymptotic Rank. We also define the Kronecker product of two tensors, $S = \sum p_{ijk} x_i y_j z_k$ (over the variable sets X, Y, Z) and $T = \sum q_{i'j'k'} x_{i'} y_{j'} z_{k'}$ (over the variable sets X', Y', Z'), as

$$S \otimes T = \sum p_{ijk} q_{i'j'k'} x_i x_{i'} y_j y_{j'} z_k z_{k'},$$

where we are summing over all i, j, k, i', j', k' (i.e. all the variables in the variable sets). One can think of the Kronecker product as simply multiplying the trilinear polynomials which represent S and T. Also note that we had defined the Kronecker product for two vectors as an element-wise multiplication, previously in section 3.3, which would be consistent with the above definition. Another notation we will use from now on is $T^{\otimes m} = T \otimes T \otimes \ldots \otimes T$, where the Kronecker product is done m times.

Another useful property of the Kronecker product is explained in the following lemma:

Lemma 4.2. $R(S \otimes T) \leq R(S) \cdot R(T)$.

Proof. We first prove that the Kronecker product of two rank one tensors is also a rank one tensor. Write the rank one tensors

$$W = (\textstyle\sum_{i=1}^n a_i x_i)(\sum_{j=1}^n b_j y_j)(\sum_{k=1}^n c_k z_k) \text{ and}$$
$$W' = (\textstyle\sum_{i=1}^n a'_i x'_i)(\sum_{j=1}^n b'_j y'_j)(\sum_{k=1}^n c'_k z'_k).$$

Then

$$W \otimes W' = (\textstyle\sum_{i=1}^n a_i x_i)(\sum_{j=1}^n b_j y_j)(\sum_{k=1}^n c_k z_k)(\sum_{i=1}^n a'_i x'_i)(\sum_{j=1}^n b'_j y'_j)(\sum_{k=1}^n c'_k z'_k),$$

which is also has a rank of one since it can be written as linear combinations of x, y, z, x', y', z'. Now we write $S \otimes T$ while expressing S and T as in equation 4.10, with the sums of the rank one tensors:

$$S \otimes T = \tag{4.11}$$
$$(\sum_{p=1}^{R(S)} ((\sum_{i',j} a_{pi'j} x_{i'j})(\sum_{j',k'} b_{pj'k'} x_{j'k'})(\sum_{k,i} c_{pki} z_{ki}))) \cdot (\sum_{p=1}^{R(T)} ((\sum_{i',j} a_{pi'j} x_{i'j})(\sum_{j',k'} b_{pj'k'} x_{j'k'})(\sum_{k,i} c_{pki} z_{ki}))),$$

which (applying the distributive property, expanding) turns into the sum of $R(S) \cdot R(T)$ rank one tensors. Now, it is possible for some of these rank one tensors to be combined and simplified into another rank one tensors – this would only reduce the number of rank one tensors in the final expression. Thus, $R(S \otimes T)$ is bounded above by $R(S) \cdot R(T)$. ■

Lemma 4.2 implies that

$$R(T^{\otimes m}) \leq R(T)^m = r^m. \tag{4.12}$$

We finally define the *asymptotic rank* of a tensor T as

$$\tilde{R}(T) = \lim_{n \to \infty} R(T^{\otimes m})^{\frac{1}{m}}. \tag{4.13}$$

$\tilde{R}(T)$ is well-defined and inspired by Lemma 4.2. There are many tensors for which $\tilde{R}(T) < R(T)$; these will be important for the Laser Method later.

Tensor Partitioning. Additionally, we discuss how to partition our tensor into *subtensors*. Recall that the variable sets from which we defined T were $X = \{x_1, x_2, \ldots, x_n\}$, $Y = \{y_1, y_2, \ldots, y_n\}$, and $Z = \{z_1, z_2, \ldots, z_n\}$. We can partition our variable sets *disjointly*: X is partitioned into $X_1, X_2, \ldots X_d$, Y into $Y_1, \ldots, Y_d$, and Z into $Z_1, \ldots Z_d$. Then we define

$$T_{ijk} = \sum_{x_f \in X_i} \sum_{y_g \in Y_j} \sum_{z_h \in Z_k} p_{fgh} x_f y_g z_h. \tag{4.14}$$

With this definition, $T = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k=1}^{d} T_{ijk}$. An example of this partitioning is in the Coppersmith-Winograd tensor. This tensor is defined as

$$CW_q = \sum_{i=1}^{q} (x_0 y_i z_i + x_i y_0 z_i + x_i y_i z_0) + x_0 y_0 z_{q+1} + x_0 y_{q+1} z_0 + x_{q+1} y_0 z_0. \tag{4.15}$$

Here, the variable sets $X = \{x_0, \ldots, x_{q+1}\}$, $Y = \{y_0, \ldots, y_{q+1}\}$, and $Z = \{z_0, \ldots, z_{q+1}\}$ have been split; specifically, X into $X_0 = \{x_0\}$, $X_1 = \{x_1, \ldots, x_q\}$, and $X_2 = \{x_{q+1}\}$ and similarly for Y and Z. The definition of T_{ijk} in equation 4.14 leads us to rewrite equation 4.15 as

$$CW_q = T_{002} + T_{020} + T_{200} + \sum_{i=1}^{q} (T_{011} + T_{101} + T_{110}). \tag{4.16}$$

Furthermore, we can actually rewrite equation 4.16 in terms of *matrix multiplication tensors*. Note that $x_0 y_0 z_{q+1}$, $x_0 y_{q+1} z_0$, and $x_{q+1} y_0 z_0$ can each be represented by $\langle 1, 1, 1 \rangle$, since they are each just one term; they are akin to just multiplying two numbers. Moreover, $\sum_{i=1}^{q} x_0 y_i z_i$, $\sum_{i=1}^{q} x_i y_0 z_i$, and $\sum_{i=1}^{q} x_i y_i z_0$ are represented by $\langle 1, 1, q \rangle$, $\langle 1, q, 1 \rangle$, and $\langle q, 1, 1 \rangle$, respectively (this can be seen from our definition of $\langle n, m, p \rangle$ previously). If we were to zero out some variables in CW_q, many of the matrix multiplication tensors would be canceled; for example, zeroing out x_0 would remove three of the terms in equation 4.15. What the Laser Method does is that it takes a large power of the tensor, e.g. CW_q, and then zeroes out some variables; taking the large power gives more 'freedom' in a sense. Also, Coppersmith and Winograd showed that the asymptotic rank of CW_q is $\tilde{R}(CW_q) = q + 2$ ([CW90]). This is significant, because the rank is exactly the number of variables of each variable set of CW_q, or its dimensions. Thus, the asymptotic rank of CW_q is as low as it can be.

The Method. We can combine the tensor partitioning and idea of taking the tensor to a large power. We partition T into T_{ijk}, which are all matrix multiplication tensors. Then, taking the mth power of T, we have

$$T^{\otimes m} = (\sum T_{ijk})^{\otimes m} = T_1 \otimes \ldots \otimes T_m, \tag{4.17}$$

where $T_1, \ldots, T_m$ are each chosen independently from the set $\{T_{ijk} \mid i, j, k \in \{1, 2, \ldots, d\}\}$. Note that, since all of $T_1, \ldots, T_m$ are matrix multiplication tensors, equation 4.17 actually writes $T^{\otimes m}$ as a sum of (much bigger) matrix multiplication tensors. Now we zero out our tensors. Specifically, we set some variables to 0 such that all the products $T_1 \otimes \ldots \otimes T_m$ that we are left with will consist of the same distribution of T_{ijk}; in other words, we want to be left with a direct sum. If we were left with some k products, then the result from Lemma 4.1 that we found previously, $\omega \leq \log_n \frac{r}{k}$, tells us that

$$\omega \leq \log_n \frac{R(T^{\otimes m})}{k}. \tag{4.18}$$

To minimize this upper bound on ω, we want to make k as large as possible; basically, we desire to zero out the minimum number of variables such that the remaining subtensors can be expressed disjointly, as a direct sum. To maximize this k, we first define $a_1, \dots, a_n$. Each a_l (for l from 1 to n) is the number of times that one of the tensors $T_1, \dots, T_m$ (from before) is of the form T_{ljk}. Then, by our definition of direct sums, we must have the variable sets of our subtensors be unique; therefore, the maximum number of disjoint subtensors in our post-zeroing out expression will be

$$k \le \binom{m}{a_1, a_2, \dots, a_m}, \tag{4.19}$$

which denotes the multinomial coefficient, or $\frac{m!}{a_1! \dots a_m!}$. This is because, having chosen $a_1, \dots, a_m$, the maximum number of possible different variable sets (as this is will be the maximum possible number of disjoint subtensors) is given, by basic combinatorics, according to the multinomial coefficient. To summarize, we have zeroed out a power of a tensor into a direct sum of other subtensors.

Now, when zeroing out, if we end up with k subtensors, where we have made k as close to the multinomial coefficient as possible, then we will inevitably end up with some "bad" subtensors: junk terms that share variables with the "good" k subtensors and which we want to remove (this only really becomes a major issue for larger powers of CW_q). However, removing these extra subtensors by zeroing out more variables will also reduce the number of good subtensors. In particular, if we had k good subtensors and $s \cdot k$ bad subtensors, our approach is to repeatedly pick a good subtensor S which shares variables with $\mathcal{O}(s)$ bad subtensors. Then we zero out the $\mathcal{O}(s)$ variables that are not in S to remove the $\mathcal{O}(s)$ bad subtensors. Each variable that we zero out will zero out at most one good subtensor. Thus, for each good subtensor, we are actually zeroing out roughly $\mathcal{O}(s)$ other good subtensors. Repeating this, we end up with roughly $\frac{k}{s}$ good subtensors. This greedy approach can be improved by the Refined Laser Method by Alman and Williams ([AW24]). In this method, we *randomly* pick a subset of roughly $k/\sqrt{s}$ good subtensors and zero out variables which are not used in any of them. This will actually remove *all* the bad subtensors with some probability. Thus, this probabilistic approach allows more good subtensors to remain, which is especially helpful for large s, as improving from k/s to $k/\sqrt{s}$ is a significant improvement. Large s generally occur for large tensors such as $CW^{\otimes 32}$.

To summarize this entire section, we took our matrix multiplication tensor $\langle n, n, n \rangle$, wrote it as a direct sum of other matrix multiplication tensors (going between these two results in our upper bound of ω from the simplified version of Schönhage's τ-theorem previously), then expressed this direct sum as a power of a tensor via the laser method (and refined laser method, which both use tensor partitioning and the idea of zeroing out variables) described in the previous paragraphs; the rank of the power of this tensor is bounded by the power of the rank of the tensor. Therefore, having a known rank-bounded tensor ultimately leads to a bound on ω via these three steps taken backwards.

5. Further Questions, Reading, and Applications

Improvements in integer multiplication algorithms have had great impacts on numerous fields, everything from financial calculations to physical modeling. Optimizing the intense calculations involved in these leads to faster results. Matrix multiplication is utilized possibly even more. Multiplying two matrices is the heart of numerous operations in linear algebra, such as finding matrices' inverses and ranks. Linear algebra in turn is applied to a variety of areas, such as cryptography, data analysis, and graphics. Graph theory also heavily relies on matrices: shortest path problem, maximum matching, spanning trees, etc. Matrices used in graph theory include everything from adjacency matrices to degree matrices to Laplacian matrices, all of which represent various properties of graphs.

We encourage interested readers to learn the integer multiplication algorithms published by Fürer ([Für09]); Harvey, van der Hoeven, and Lecerf ([HvdHL16]); and Harvey and van der Hoeven ([HvdH21]). There are also many tricks for optimization that can be employed in integer multiplication problems, such as the $\sqrt{2}$ trick, Harley's trick, Granlund's trick, and Nussbaumer's trick (see [GKZ07],[Nus80]). Implementations of Karatsuba, Toom-Cook, and Schönhage-Strassen can be found in the GNU Multiple Precision Arithmetic Library, along with useful documentation. On the matrix multiplication side, we encourage readers to learn the recent papers published by Alman, William, and others in 2022 and 2024 ([WXXZ24], [ADW+24]) which utilize asymmetry to improve bounds on ω. Textbooks such as [Knu98] and [CLRS09] were also very useful in researching for this paper.

Some further areas of research include improving on Harvey and van der Hoeven's galactic algorithm for integer multiplication to achieve practical usage; improving on matrix multiplication algorithms by finding a different starting tensor than the Coppersmith-Winograd tensor for the Laser Method or discovering an entirely new method.

Acknowledgements

I would like to thank Simon Rubinstein-Salzado and Dean Menezes for their mentorship and help while researching and writing this paper.

References

[ADW+24] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication, 2024.

[AW24] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. *TheoretiCS*, 3:32, 2024. Id/No 21.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* Cambridge, MA: MIT Press, 3rd ed. edition, 2009.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[FBH+22] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature, London*, 610(7930):47–53, 2022.

[Für09] Martin Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009.

[GKZ07] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *Proceedings of the 2007 international symposium on symbolic and algebraic computation, ISSAC 2007, Waterloo, ON, Canada, July 29–August 1, 2007.*, pages 167–174. New York, NY: Association for Computing Machinery (ACM), 2007.

[Har09] David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. *J. Symb. Comput.*, 44(10):1502–1510, 2009.

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n\log n)$. *Ann. Math. (2)*, 193(2):563–617, 2021.

[HvdHL16] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *J. Complexity*, 36:1–30, 2016.

[Knu98] Donald E. Knuth. *The art of computer programming. Vol. 2: Seminumerical algorithms.* Bonn: Addison-Wesley, 3rd ed. edition, 1998.

[KO62] Anatoly Karatsuba and Yuri Ofman. Multiplication of many-digital numbers by automatic computers. *Dokl. Akad. Nauk SSSR*, 145:293–294, 1962.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation, ISSAC 2014, Kobe, Japan, July 23–25, 2014*, pages 296–303. New York, NY: Association for Computing Machinery (ACM), 2014.

[Lüd15] Christoph Lüders. Fast multiplication of large integers: Implementation and analysis of the DKSS algorithm. *CoRR*, abs/1503.04955, 2015.

[Nus80] Henri J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Trans. Acoust. Speech Signal Process.*, 28:205–215, 1980.

[Sch81] A. Schönhage. Partial and total matrix multiplication. *SIAM J. Comput.*, 10:434–455, 1981.

[SS71] A. Schönhage and V. Strassen. Fast multiplication of large numbers. *Computing*, 7:281–292, 1971.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[Str88] V. Strassen. The asymptotic spectrum of tensors. *Journal für die reine und angewandte Mathematik*, 384:102–152, 1988.

[Win71] S. Winograd. On multiplication of 2 × 2 matrices. *Linear Algebra Appl.*, 4:381–388, 1971.

[WXXZ24] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 35th annual ACM-SIAM symposium on discrete algorithms, SODA 2024, Alexandria, Virginia, January 7–10, 2024*, pages 3792–3835. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM); New York, NY: Association for Computing Machinery (ACM), 2024.