Evolution of Multiplication Algorithms for Integers and Matrices

Shamik Khowala
shamikkhowala@gmail.com

July 13, 2025

Table of Contents

Integer Multiplication

Karatsuba Algorithm

We multiply $\overline{ab}$ and $\overline{cd}$. Let $x_1 = a \cdot c$, $x_2 = b \cdot c$, $x_3 = a \cdot d$, and $x_4 = b \cdot d$. Then,

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

This is the "naive" way, using four multiplications.

Karatsuba Algorithm

We multiply $\overline{ab}$ and $\overline{cd}$. Let $x_1 = a \cdot c$, $x_2 = b \cdot c$, $x_3 = a \cdot d$, and $x_4 = b \cdot d$. Then,

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

This is the "naive" way, using four multiplications.

Instead, let $y_1 = a \cdot c$, $y_2 = b \cdot d$, and $y_3 = (a + b)(c + d)$. Then,

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

This method only uses three multiplications, at the cost of a few more additions/subtractions. Note that, when referring to the number of multiplications, we are talking about the number of *unique* multiplications involving *input variables*.

Karatsuba Algorithm Continued

The multiplication of two two-digit numbers (using only three multiplications) will be the base case for our recursion. The recursion uses a divide-and-conquer approach.

Karatsuba Algorithm Continued

The multiplication of two two-digit numbers (using only three multiplications) will be the base case for our recursion. The recursion uses a divide-and-conquer approach.

Let $u = \overline{u_{2n-1}u_{2n-2}...u_0}$ and $v = \overline{v_{2n-1}v_{2n-2}...v_0}$. Then we split u into $p = \overline{u_{2n-1}...u_n}$ and $q = \overline{u_{n-1}...u_0}$ and v into $r = \overline{v_{2n-1}...v_n}$ and $s = \overline{v_{n-1}...v_0}$. Since $u = p10^n + q$ and $v = r10^n + s$, we see that

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

which only requires three multiplications. The subsequent multiplications of pr, $(p+q)(r+s)$, and qs are done recursively via further splitting.

Karatsuba Algorithm Continued

The multiplication of two two-digit numbers (using only three multiplications) will be the base case for our recursion. The recursion uses a divide-and-conquer approach.

Let $u = \overline{u_{2n-1}u_{2n-2}...u_0}$ and $v = \overline{v_{2n-1}v_{2n-2}...v_0}$. Then we split u into $p = \overline{u_{2n-1}...u_n}$ and $q = \overline{u_{n-1}...u_0}$ and v into $r = \overline{v_{2n-1}...v_n}$ and $s = \overline{v_{n-1}...v_0}$. Since $u = p10^n + q$ and $v = r10^n + s$, we see that

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

which only requires three multiplications. The subsequent multiplications of pr, $(p+q)(r+s)$, and qs are done recursively via further splitting.

The computational complexity is $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.58})$. Check out my paper for a Python implementation.

Toom-Cook Algorithm

Toom-Cook splits the input numbers into more parts; e.g. Toom-3 splits them into three parts each (so Karatsuba can be thought of as Toom-2). We discuss Toom-3, which can be easily generalized to Toom-n.

Toom-Cook Algorithm

Toom-Cook splits the input numbers into more parts; e.g. Toom-3 splits them into three parts each (so Karatsuba can be thought of as Toom-2). We discuss Toom-3, which can be easily generalized to Toom-n.

Split x into x_2, x_1, and x_0; likewise, we split y into y_2, y_1, and y_0. Let $X(t) = x_2 t^2 + x_1 t + x_0$ and $Y(t) = y_2 t^2 + y_1 t + y_0$. Then our desired output is determined by

$$W(t) = X(t) \cdot Y(t) = w_4 t^4 + w_3 t^3 + w_2 t^2 + w_1 t + w_0.$$

Toom-3 Algorithm

Regular polynomial multiplication and matching up the coefficients gives

$$\begin{aligned} w_4 &= x_2 \cdot y_2, \\ w_3 &= x_2 \cdot y_1 + x_1 \cdot y_2, \\ w_2 &= x_2 \cdot y_0 + x_1 \cdot y_1 + x_0 \cdot y_2, \\ w_1 &= x_1 \cdot y_0 + x_0 \cdot y_1, \text{ and} \\ w_0 &= x_0 \cdot y_0. \end{aligned} \tag{1.1}$$

This can be solved as a system of equations in order to determine $w_0, \ldots w_4$. However, the total number of multiplications is 9.

Toom-3 Algorithm Continued

Instead, we introduce a method that only requires 5 multiplications. Evaluate $W(t) = X(t) \cdot Y(t)$ at $t = 0, 1, -1, 2$, and ∞:

$$\begin{aligned} w_0 &= x_0 \cdot y_0, \\ w_4 + w_3 + w_2 + w_1 + w_0 &= (x_2 + x_1 + x_0)(y_2 + y_1 + y_0), \\ w_4 - w_3 + w_2 - w_1 + w_0 &= (x_2 - x_1 + x_0)(y_2 - y_1 + y_0), \\ 16w_4 + 8w_3 + 4w_2 + 2w_1 + w_0 &= (4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0), \\ w_4 &= x_2 \cdot y_2. \end{aligned} \tag{1.2}$$

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.

Toom-3 Algorithm Continued

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 (like Karatsuba). Since Toom-3 divides the input numbers into three parts through five multiplications, we see that its computational complexity is $\mathcal{O}(n^{\log_3 5}) \approx \mathcal{O}(n^{1.46})$.

Toom-3 Algorithm Continued

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 (like Karatsuba). 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 much more additions, subtractions, and multiplication by constants (more 'elementary' operations than multiplication) than Karatsuba, it only holds an advantage for input numbers of large lengths (above a certain cutoff point).

Toom-3 Algorithm Continued

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 (like Karatsuba). 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 much more additions, subtractions, and multiplication by constants (more 'elementary' operations than multiplication) than Karatsuba, it only holds an advantage for input numbers of large lengths (above a certain cutoff point).

Check out my paper for a Python implementation of Toom-3.

Toom-n Algorithm

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. By the interpolation theorem, $k+1$ points are needed to determine a polynomial of degree k. Thus, we evaluate our degree $2n-2$ polynomial W at $2n-1$ points.

Toom-n Algorithm

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. By the interpolation theorem, $k+1$ points are needed to determine a polynomial of degree k. Thus, we evaluate our degree $2n-2$ polynomial W at $2n-1$ points.

While any set of $2n-1$ t's theoretically work, it is popular 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.

Toom-n Algorithm

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. By the interpolation theorem, $k+1$ points are needed to determine a polynomial of degree k. Thus, we evaluate our degree $2n-2$ polynomial W at $2n-1$ points.

While any set of $2n-1$ t's theoretically work, it is popular 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.

Fast Fourier Transform

FFT is an $\mathcal{O}(n \log n)$ approach to evaluate the DFT of a polynomial. The DFT of a polynomial A is

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \dots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega & \dots & \omega^{n-1} \\ \dots & & & \\ 1 & \omega^{n-1} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{bmatrix}, \tag{1.3}$$

where ω is a primitive nth root of a field (e.g. generators modulo some number). FFT splits A into A_0 and A_1 where $A(x) = A_0(x^2) + xA_1(x^2)$:

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \dots \\ A(\omega^m) \\ \dots \\ A(\omega^{2m-1}) \end{bmatrix} = \begin{bmatrix} A_0(1) \\ A_0(\omega^2) \\ \dots \\ A_0(1) \\ \dots \\ A_0(\omega^{2m-2}) \end{bmatrix} + \begin{bmatrix} A_1(1) \\ \omega A_1(\omega^2) \\ \dots \\ \omega^m A_1(1) \\ \dots \\ \omega^{2m-1} A_1(\omega^{2m-2}) \end{bmatrix}. \tag{1.4}$$

Convolutions

Suppose that $A(x)$ and $B(x)$ are degree $n-1$ polynomials in the ring $\mathbb{Z}_q[x]$. Then polynomial multiplication gives

$$C(x) = A(x) \cdot B(x) = \sum_{i=0}^{2n-2} (\sum_{j=0}^{i} a_j b_{i-j}) x^i. \tag{1.5}$$

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

$$a * b = \sum_{i=0}^{n} a_i b_{n-i}. \tag{1.6}$$

Then,

$$a * b = c, \tag{1.7}$$

which is the coefficient vector of the output polynomial $C(x)$.

Positive Wrapped Convolutions

The positive wrapped convolution, denoted as the polynomial C^+, is defined as

$$C^+(x) = \sum_{i=0}^{n-1} c_i x^i \tag{1.8}$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \pmod{q}. \tag{1.9}$$

Actually,

$$C^+(x) \equiv C(x) \pmod{x^n - 1}. \tag{1.10}$$

Negative Wrapped Convolution

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

$$C^-(x) = \sum_{i=0}^{n-1} c_i x^i, \tag{1.11}$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} \; - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \pmod{q}. \tag{1.12}$$

Actually,

$$C^-(x) \equiv C(x) \pmod{x^n + 1}. \tag{1.13}$$

Convolutions Continued

The positive wrapped convolution can be expressed as

$$c_+ = \mathrm{FFT}^{-1}(\mathrm{FFT}(a) \otimes \mathrm{FFT}(b)). \tag{1.14}$$

If we 'weight' a and b according to $a'_i = \theta^i a_i$ and $b'_i = \theta^i b_i$, where θ is a primitive $2n$th root of unity, then

$$c_- = \mathrm{FFT}^{-1}(\mathrm{FFT}(a') \otimes \mathrm{FFT}(b')). \tag{1.15}$$

Note that since ω and θ are powers of 2 in the field $\mathbb{Z}_{2^n+1}$, then multiplying by powers of them is equivalent to cyclic shifts in binary, which is very efficient computationally.

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

1. Split x and y into r parts, each of length m, which can be thought of as coefficients of two r degree polynomials.

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

1. Split x and y into r parts, each of length m, which can be thought of as coefficients of two r degree polynomials.
2. Weight both coefficient vectors with the powers of θ.

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

1. Split x and y into r parts, each of length m, which can be thought of as coefficients of two r degree polynomials.
2. Weight both coefficient vectors with the powers of θ.
3. Perform FFT on the coefficient vectors.

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

1. Split x and y into r parts, each of length m, which can be thought of as coefficients of two r degree polynomials.
2. Weight both coefficient vectors with the powers of θ.
3. Perform FFT on the coefficient vectors.
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.

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

1. Split x and y into r parts, each of length m, which can be thought of as coefficients of two r degree polynomials.
2. Weight both coefficient vectors with the powers of θ.
3. Perform FFT on the coefficient vectors.
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 FFT^{-1} on the c coefficient vector.

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

1. Split x and y into r parts, each of length m, which can be thought of as coefficients of two r degree polynomials.
2. Weight both coefficient vectors with the powers of θ.
3. Perform FFT on the coefficient vectors.
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 FFT^{-1} on the c coefficient vector.
6. Apply the 'counterweight', θ^{-k}.

Schönhage-Strassen Algorithm

To multiply $x \cdot y$ we apply FFT in the field $\mathbb{Z}_{2^n+1}$. Let $n = r \cdot m$, where $r = 2^k$ is the largest power of two dividing n. The algorithm is as follows:

1. Split x and y into r parts, each of length m, which can be thought of as coefficients of two r degree polynomials.
2. Weight both coefficient vectors with the powers of θ.
3. Perform FFT on the coefficient vectors.
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 FFT^{-1} on the c coefficient vector.
6. Apply the 'counterweight', θ^{-k}.
7. Reconstruct the integer product from the coefficient vector by adding and carrying; in other words, evaluate $C(x)$ at $x = 10$.

Matrix Multiplication

Strassen Algorithm

$$\mathbf{C} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix} \tag{2.1}$$

requires **8** multiplications. However, if we define, with **7** multiplications,

$$\begin{aligned} X_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\ X_2 &= B_{11}(A_{21} + A_{22}), \\ X_3 &= A_{11}(B_{12} - B_{22}), \\ X_4 &= A_{22}(B_{21} - B_{11}), \\ X_5 &= B_{22}(A_{11} + A_{12}), \\ X_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\ X_7 &= (A_{12} - A_{22})(B_{21} + B_{22}), \text{then} \end{aligned} \tag{2.2}$$

$$\mathbf{C} = \begin{bmatrix} X_1 + X_4 - X_5 + X_7 & X_3 + X_5 \\ X_2 + X_4 & X_1 - X_2 + X_3 + X_6 \end{bmatrix} \tag{2.3}$$

Strassen Algorithm Continued

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. The computational complexity of this is $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.807})$. However, note that for small enough matrices the $\mathcal{O}(n^3)$ approach is more efficient; the cutoff point between these two algorithms depends on numerous factors though, including hardware efficiency and code optimization.

Strassen Algorithm Continued

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. The computational complexity of this is $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.807})$. However, note that for small enough matrices the $\mathcal{O}(n^3)$ approach is more efficient; the cutoff point between these two algorithms depends on numerous factors though, including hardware efficiency and code optimization.

Winograd also discovered a modified version of Strassen's algorithm that requires the same number of multiplications but fewer additions/subtractions.

Tensors

We can represent the multiplication of two $n \times n$ matrices as a tensor in its trilinear form:

$$T = \langle n, n, n \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} x_{ij} y_{jk} z_{ki}, \tag{2.4}$$

where x_{ij} is in matrix $\mathbf{A}$ and y_{jk} is in matrix $\mathbf{B}$. Then the coefficient of z_{ki} is just the entry at position (i, k) in matrix $\mathbf{C}$.

Tensors

We can represent the multiplication of two $n \times n$ matrices as a tensor in its trilinear form:

$$T = \langle n, n, n \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} x_{ij} y_{jk} z_{ki}, \tag{2.4}$$

where x_{ij} is in matrix $\mathbf{A}$ and y_{jk} is in matrix $\mathbf{B}$. Then the coefficient of z_{ki} is just the entry at position (i, k) in matrix $\mathbf{C}$.

The *tensor rank* of T, $R(T)$, is defined as the minimum number of rank one tensors which sum to T. A rank one tensor is one that can be expressed as the product of linear polynomials:

$$S = (\sum_{i=1}^{n} a_i x_i)(\sum_{j=1}^{n} b_j y_j)(\sum_{k=1}^{n} c_k z_k) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} a_i b_j c_k x_i y_j z_k. \tag{2.5}$$

A Lemma

Lemma

If $R(\langle n, n, n\rangle) = r$, then $\omega \leq \log_n r$ via a recursive algorithm.

Proof.

Express $\langle n, n, n\rangle$ as a sum of r rank one tensors:

$$\langle n, n, n\rangle = \sum_{p=1}^{r}((\sum_{i',j} a_{pi'j}x_{i'j})(\sum_{j',k'} b_{pj'k'}x_{j'k'})(\sum_{k,i} c_{pki}z_{ki})).$$

Split the 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, being done easily in linear time. The recursive steps give us the $\mathcal{O}(\log_n r)$ time. ■

Components of the Laser Method

1. Zeroing out a tensor. This involves setting variables in a certain tensor to 0, effectively canceling some terms and turning the tensor into another, simpler tensor.
2. Direct sums of a tensor. This just the sum of copies of a single tensor but each copy has independent variable sets. If A is the direct sum of k copies of tensor T, and if $R(A) = r$, then by the lemma we have $\omega \leq \log_n \frac{r}{k}$.
3. $R(T^{\otimes m}) \leq R(T)^m$, where we define, if $S = \sum p_{ijk} x_i y_j z_k$ and $T = \sum q_{tuv} x_t y_u z_v$, $S \otimes T = \sum p_{ijk} q_{tuv} x_{i,t} y_{j,u} z_{k,v}$ (think of this as just a normal multiplication of the trilinear polynomials).
4. Partitioning a tensor. If we disjointly split the variable sets into d subsets each, then define $T_{ijk} = \sum_{x_f \in X_i} \sum_{y_g \in Y_j} \sum_{z_h \in Z_k} p_{fgh} x_f y_g z_h$. So then $T = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k=1}^{d} T_{ijk}$.

The Main Idea

We take our matrix multiplication tensor $\langle n, n, n \rangle$ and write it as a direct sum of other matrix multiplication tensors (going between these two results in our upper bound of $\omega \leq \log_n \frac{r}{k}$ from before). Then, we express this direct sum as a power of a tensor via partitioning and zeroing out variables (more details in paper). Finally, the rank of the power of this tensor is bounded by the power of the rank of the tensor. Therefore, if we can simply bound the rank of this tensor we obtained (or do the previous steps to get to a tensor of known rank bound), then we can ultimately obtain a bound on ω via these three steps.

Thank you for listening! Any questions?