# FAST INTEGER AND MATRIX MULTIPLICATION ALGORITHMS

AADISH VERMA

Abstract. In this paper, we discuss algorithms for two major problems in theoretical computer science: fast integer multiplication and fast matrix multiplication. We begin with a brief introduction to theoretical computer science, including time complexity and common patterns in algorithms. We then provide explanations of a few major algorithms for each problem. For integer multiplication, we discuss the Karatsuba and Toom-Cook algorithms and have a brief treatment of the Harvey-van der Hoeven result. For matrix multiplication, we explain the tensor basis for all recent fast matrix multiplication results, with which we discuss Strassen's 1969 algorithm and Strassen's 1986 laser method.

## 1. Introduction

Theoretical computer science sits at the intersection of computer science and mathematics. One of the primary goals of computer science is to find fast *algorithms* – i.e., methods to implement some functionality. This involves a surprising amount of mathematics, both in developing the theoretical foundations needed to motivate, derive, and justify algorithms, but also to quantify and objectively analyze their speed for various inputs. In this paper, we discuss two major problems in theoretical computer science: fast multiplication of integers, and fast multiplication of matrices. While these seem like primitive operations, doing them quickly for massive inputs is, in fact, still an active field of study, and involves a lot more math than it may seem. We only focus a couple of major leaps in each respective field for the sake of space, and give brief motivations and summaries of the more complex recent algorithms. Furthermore, while some of the advancements described here are so complex so as to remain theoretical, many of the ideas of algorithms discussed have found practical applications in applied computer science to accelerate large-scale computation. We begin with a basic introduction to big-O notation, which is possibly one of the most important tools in theoretical computation. We then split the paper into two sections; one for integer multiplication, and one for matrix multiplication. Each section contains a formalized description of the problem, an explanation of the relevant mathematical background, and finally a discussion of actual algorithms. For integer multiplication, we cover the Karatsuba and Toom-Cook algorithms and briefly touch on the Harvey-van der Hoeven result. For matrix multiplication, we cover Strassen's algorithm, contexualize it and the broader field in terms of tensors, then finally conclude with a motivation for Strassen's laser method.

## 2. Background

2.1. **Big-O notation.** In order to quantify improvements in algorithms, we need some objective measure of the speed of an algorithm. Big-O notation is the standard way to do this:

*Date*: July 10, 2026.

**Definition 2.1.** For a given function $g(n)$, a function $f(n)$ is a member of the set of functions $O(g(n))$ if there exist a $c$ and $n_0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

The use of big-O notation informally is quite different. We say $f(n)$ is $O(g(n))$, or $f(n) = O(g(n))$, to mean $f(n) \in O(g(n))$. An *algorithm* has complexity $O(g(n))$ if its runtime is $O(g(n))$.

Some complexities are common enough to have (mostly self-explanatory names). $O(n^p)$ for some $p \geq 0$ is "polynomial" time, $O(n)$ is linear, $O(n^2)$ is quadratic, $O(n^3)$ is cubic, $O(\log n)$ is logarithmic, $O(n \log n)$ is linearithmic.

Finally, consider the implications of our definition of big-O notation. The primary reason we define it this way is to eliminate constant factors – an algorithm with $2n$ additions and an algorithm with $100n$ additions are both $O(n)$. If a function or complexity has multiple terms, the one that dominates in the long run is usually the only one that is included in its big-O complexity. If $f(n) = 3n^2 + 10n$, $f$ is $O(n^2)$. If $f(n) = 2^n + 100n^5$, $f$ is $O(2^n)$. This allows us to focus on *asymptotic complexity*; that is, what happens as $n \to \infty$. This is reasonable as most of these algorithms only do become useful above a relatively large $n$. However, it means that hidden constant factors can bite hard. If I have an $O(n^3)$ and an $O(\log n)$ algorithm, it seems like the easy choice is the $O(\log n)$ one. But that disregards the constant factor – the second algorithm may actually use $10^{100} \log n$ operations, which I wouldn't know solely from big-O algorithms.

This motivates the term *galactic algorithm*, used to describe an algorithm whose crossover point – the $n$ for which it becomes faster than the next fastest algorithm – is so astronomically high that it is completely useless for real-world practical use. Many galactic algorithms boast very low asymptotic big-O complexities but have a massive associated constant factor, meaning they only become faster than "slower" methods for very high $n$ which are physically impossible to encounter.

2.2. **Cost models.** What is the runtime of an algorithm, then? For our purposes, an algorithm's runtime can be thought of as consisting of the total complexity of the operations it does. However, how an operation is defined varies for integer and matrix multiplication.

For integer multiplication, the complexity we are attempting to measure directly depends on the number of digits of the input integers, so we let addition, subtraction, and multiplication by a constant scalar of $d$-digit integers be $O(d)$.

For matrix multiplication, however, we care about the size of the matrices being multiplied; integer multiplication here is comparatively constant-time, so we define addition, subtraction, and multiplication of integers as $O(1)$, regardless of size. However, the *matrix* addition, subtraction, and scalar multiplication used in fast matrix multiplication algorithms have to do one integer operation for each output entry, so matrix addition, subtraction, and multiplication by a scalar with $n \times n$ matrices has $O(n^2)$ complexity.

Similarly, what $n$ represents in either section differs. In integer multiplication, $n$ represents the number of digits of the inputs. In matrix multiplication, $n$ is the side length of the two square input matrices (i.e., the product is of two $n \times n$ matrices).

2.3. **Divide-and-conquer algorithms.** All of the algorithms discussed in this paper are *divide-and-conquer* algorithms, part of the greater class of *recursive* functions in computer science. This essentially means the algorithm calls itself with simpler or smaller inputs – in our algorithms, the input size is always a fraction of the size of the original, greater input – some number of times as part of its computation. Of course, to prevent an infinite loop,

some *base case* must be defined where the algorithm directly computes a result instead of calling itself. The textbook example of a divide-and-conquer algorithm is merge sort, which sorts an array of numbers by repeatedly subdividing it, recursively sorting the subarrays, then merging the sorted subarrays.

The complexity of most common divide-and-conquer algorithms can be defined using a recurrence. The recurrence is created as follows: let $T(n)$ represent the complexity of the algorithm, $a$ represent how many recursive calls it makes, $b$ represent how much the inputs are subdivided before being passed into the recursive calls, and $f(n)$ represent the time complexity of the constant (non-recursive) portion of the algorithm. Then, we have

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

The Master Theorem is a tool to analyze the complexity of such recurrences. It has several cases, but all of the algorithms we will discuss can be analyzed using the following case:

**Theorem 2.2** ( [BHS80]). *(Limited Master Theorem.) For some recurrence* $T(n) = aT\left(\frac{n}{b}\right) + f(n)$*, if there is an* $\epsilon > 0$ *such that*

$$f(n) = O\left(n^{\log_b(a)-\epsilon}\right),$$

*then*

$$T(n) = O(n^{\log_b a}).$$

The proof is omitted for the sake of space.

## 3. Fast integer multiplication

3.1. **Goal.** Integer multiplication is one of the most fundamental arithmetic operations. Fast integer multiplication deals with fast ways of multiplying (usually very large) integers, which has applications in cryptography, prime number research, computer algebra, and many other fields. In this section, we describe two divide-and-conquer strategies – the Karatsuba algorithm, and the Toom-Cook family of algorithms – for fast matrix multiplication and end with a brief overview of modern progress on the problem. We follow the convention used in most algorithms, which is that both input integers have the same length of digits (although the base used depends, and indeed is often crucial to the workings of the algorithms).

3.2. **The Karatsuba algorithm.** The Karatsuba algorithm [KO63] is a textbook example used for computational complexity analysis and one of the earliest results in fast integer multiplication algorithms. The algorithm is defined as follows to compute the product of two $n$-digit integers $X$ and $Y$.

First, the base case: if $n$ is 1, look up the product using a precomputed table of single-digit products and return it. Otherwise, split the inputs $X$ and $Y$ each into two $n/2$-digit halves, producing $X_1, X_2, Y_1, Y_2$. This is equivalent to changing the base of the two numbers to a power $b^k$ of the original base (say, $b = 10$), such that each number becomes two digits long. Equivalently $X_1, X_2, Y_1, Y_2$ are simply digits of the inputs in that high base, e.g., $X = X_1 b^k + X_2$ and $Y = Y_1 b^k + Y_2$. If $n$ is not even, pad $X$ and $Y$ with zeros on the left to produce inputs with even $n$ and run the algorithm.

Now, define the following intermediate products:

$$\begin{aligned} U &= X_1Y_1 \\ V &= X_2Y_2 \\ W &= (X_1 - X_2)(Y_1 - Y_2) \\ Z &= U + V - W = X_1Y_2 + X_2Y_1. \end{aligned}$$

For each multiplication done by the above computations, we can recursively use the Karatsuba algorithm to compute it, given this algorithm its divide-and-conquer nature. Then, we compute the product as

$$\begin{aligned} & Ub^{2k} + Zb^k + V \\ &= X_1Y_1b^{2k} + (X_1Y_2 + X_2Y_1)b^k + X_2Y_2 \\ &= (X_1b^k + X_2)(Y_1b^k + Y_2) \\ &= XY \end{aligned}$$

which indeed is equivalent to the product of $X$ and $Y$.

To model the time complexity of the Karatsuba algorithm, we simply apply the Master Theorem. Letting the algorithm be $T(n)$, the recurrence is

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

where $O(n)$ represents the time complexity of constant part of the algorithm. The coefficient of 3 indicates that the algorithm contains three recursive calls of itself (one for each integer multiplication done in computing $U$, $V$, and $W$, respectively). In our case, the constant part of the algorithm is a fixed number of additions and subtractions, each of which is $O(n)$ (and, since the number of those operations is fixed, it is a constant factor that need not be included in the time complexity).

To meet the Master Theorem's preconditions, we find some $\epsilon > 0$ such that $O(n) = O\left(n^{\log_2(3)-\epsilon}\right)$ for some $\epsilon > 0$. Since $\log_2 3 > 1$, we can simply let $\epsilon = \log_2(3) - 1$, such that

$$\begin{aligned} O\left(n^{\log_2(3)-\epsilon}\right) &= O\left(n^{\log_2(3)-(\log_2(3)-1)}\right) \\ &= O\left(n\right). \end{aligned}$$

Having met this condition, we can now apply the Master Theorem, finding

$$T(n) = O\left(n^{\log_2 3}\right) \approx O\left(n^{1.585}\right).$$

This is a significant improvement on the quadratic $O\left(n^2\right)$ textbook (long multiplication) algorithm.

3.3. **Toom-Cook algorithms.** Toom-$k$ multiplication ( [Too63, Bod07] and [Coo66, Chapter 3]) is a generalization of Karatsuba where the inputs are split into $k$ ($k \geq 2$) blocks each. This is equivalent to finding a high base $B$ which is a power of the original base $b$ (e.g., 10), i.e., $B = b^j$ for some $j$, such that each input can be represented in base-$B$ using $k$ digits or less (so each digit is one block). Informally, it is fine to treat each block as a chunk of digits. In either representation, multiplying each block by increasing powers of $B$ then summing recovers the original inputs. It's also possible for $k$ to be different for either input, but for the purposes of this explanation we assume that each input is split into the same number of chunks, which is always possible by padding the smaller input on the left with zeroes.

3.3.1. *The big idea.* Let the blocks of our inputs $X$ and $Y$ be $X_1, \ldots, X_k$ and $Y_1, \ldots, Y_k$. Construct two $(k-1)$-degree polynomials $p(x)$ and $q(x)$ as follows:

$$p(x) = X_1 x^{k-1} + X_2 x^{k-2} + \ldots + X_{k-1} x + X_k$$
$$q(x) = Y_1 x^{k-1} + Y_2 x^{k-2} + \ldots + Y_{k-1} x + Y_k.$$

By construction and as explained above, $p(B) = X$ and $q(B) = Y$. This is an intuitive result following from how we define the blocks.

The purpose of these polynomials is that we can then define $r(x) = p(x)q(x)$ as their product of the two polynomials. Since we know that $p(B) = X$ and $q(B) = Y$, $r(B) = p(B)q(B) = XY$ is one way to compute the product of $X$ and $Y$. Thus, if we can determine the coefficients of $r$ efficiently, we can simply compute $r(B)$. The big idea of Toom-Cook is that computing these coefficients and evaluating the polynomial can be done quickly for large $n$.

It is commonly known that a polynomial of degree $n$ requires $n+1$ points (e.g., input-output pairs) to uniquely determine its coefficients. We know that

$$\deg(r) = \deg(pq) = \deg(p) + \deg(q) = 2k - 2,$$

so we need to compute $2k-1$ points in order to pin down the coefficients of $r$.

3.3.2. *Evaluation.* It turns out, we can evaluate $r(x)$ relatively quickly for $2k-1$ inputs. Note that inputs are generally carefully chosen and hard-coded so as to make later steps faster. One common input that is unusual is $\infty$. Using infinity as an input is defined here (not generally) as $p(\infty) = \lim_{x\to\infty} \frac{p(x)}{x^{\deg(p)}}$ for some polynomial $p$. It can be shown and is intuitive that $p(\infty)$ is then simply the coefficient on the highest power term of $p$. Inputs can also be chosen such that outputs for one input can be computed partially in terms of the output for another input, saving a few more valuable operations. Finally, small inputs are often chosen so that, instead of having to do multiplication of each coefficient of $p$ and $q$ with the given input, repeated additions or subtractions can be done. Note that these improvements are genuine but don't improve the big-O asymptotic complexity, as they affect the constant-time, not recursive, part of the algorithm and don't transfer to the asymptotic complexity. Regardless, they *do* improve the hidden constant factor and thus are valid optimizations.

Now we have chosen some set of inputs $i_1, \ldots, i_{2k-1}$. After plugging a given input into $p(x)$ and $q(x)$ and computing the outputs, we can multiply the outputs to compute the output of $r$ at that input. Importantly, we can recursively use Toom-Cook to multiply the outputs of $p$ and $q$, giving the algorithm its divide-and-conquer nature. Of course, for small numbers, a fallback to a lookup table or long multiplication is wise (and necessary as a base case). In this way, $2k-1$ input-output pairs for $r$ are computed.

3.3.3. *Finding the coefficients.* Now, we can compute the coefficients $c_1, c_2, \ldots, c_{2k-1}$ of $r$. Our computed outputs $r(i_1), \ldots, r(i_{2k-1})$ can also be represented in terms of $r$'s coefficients using a matrix-vector multiplication:

$$\begin{bmatrix} i_1^0 & i_1^1 & \cdots & i_1^{2k-2} \\ \vdots & \vdots & \ddots & \vdots \\ i_{2k-1}^0 & i_{2k-1}^1 & \cdots & i_{2k-1}^{2k-2} \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_{2k-1} \end{bmatrix} = \begin{bmatrix} r(i_1) \\ \vdots \\ r(i_{2k-1}) \end{bmatrix}.$$

For the row of the matrix corresponding to the input of infinity (if there is such an input), the powers of the input are obviously not well-defined. For our purposes, we define the row

so that it matches the stated behavior of extracting the highest-order term's coefficient: all zeros except for a 1 in the farthest-right entry of the row.

We can, of course, substitute in the powers of the inputs (since our inputs are hard-coded and thus we can also precompute their powers), as well as the right-hand side product vector (since we've already computed the values of $r$ at $i_1, \ldots, i_{2k-1}$).

Assuming the inputs were chosen wisely – which, of course, they will be in any practical implementation of this algorithm – the matrix of the input powers is invertible. This means we can multiply on the left by the inverse of that matrix on the left on both sides. On the left-hand side, this cancels out the matrix-vector multiplication, leaving us with just the vector of $r$'s coefficients. The right-hand size becomes a matrix-vector multiplication of the aforementioned inverse matrix with the vector of $r$'s values at the chosen inputs:

$$\begin{bmatrix} c_1 \\ \vdots \\ c_{2k-1} \end{bmatrix} = \left( \begin{bmatrix} i_1^0 & i_1^1 & \cdots & i_1^{2k-2} \\ \vdots & \vdots & \ddots & \vdots \\ i_{2k-1}^0 & i_{2k-1}^1 & \cdots & i_{2k-1}^{2k-2} \end{bmatrix} \right)^{-1} \begin{bmatrix} r(i_1) \\ \vdots \\ r(i_{2k-1}) \end{bmatrix}.$$

By computing this matrix-vector multiplication, we can obtain the coefficients of $r$. In practice, the inverse matrix is precomputed. This step also contributes a constant factor, making it a consideration for choosing optimal hard-coded inputs; one optimization, for example, is to choose inputs such that some entries of the matrix-vector product (i.e., coefficients of $r$) can be computed in terms of other entries.

Now, we have the coefficients of $r$, we can simply compute $r(B)$ directly. This is part of the non-recursive portion of the algorithm, which may be surprising: shouldn't multiplications use Toom-Cook recursively? However, since these are multiplications by constant scalars, and particularly, powers of $B$, the multiplication can be done with a simple digit-shift. Recursively applying the algorithm is not needed.

3.3.4. *Time complexity.* We can compute time complexity again using the Master Theorem. Similarly to Karatsuba, all of the constant operations are ultimately addition, subtraction, or scalar multiplication; the matrix-vector multiplication expands to a combination of those operations as well. All of those operations are, of course, $O(n)$. The number of times the algorithm recursively calls itself is $2k - 1$ – once for input to compute the product of $p(x) \times q(x)$. The outputs of $p$ and $q$ are each $n/k$ digits, so the recurrence is as follows for fixed $k$:

$$T(n) = (2k-1)T\left(\frac{n}{k}\right) + O(n).$$

The rest of the proof maps perfectly to the time complexity proof for Karatsuba. The precondition given by Master Theorem is that we must be able to write the constant portion as

$$O(n) = O\left(n^{\log_k(2k-1)-\epsilon}\right).$$

This is of course trivial as $2k - 1 > k$ (recall we are given $k \geq 2$), so $\log_k(2k-1) > 1$ (1 being the exponent of the linear-time constant part) and we can simply set $\epsilon$ to the difference. Applying the Master Theorem, we finally compute the time complexity of an Toom-$k$ integer multiplication algorithm as $O(n^{\log_k(2k-1)})$.

Toom-Cook is perhaps the most useful algorithm described in this paper; it is implemented in several programming libraries for fast integer multiplication. The Karatsuba algorithm, too, can actually be recovered as a Toom-2 multiplication algorithm.

3.4. **Harvey-van der Hoeven and beyond.** In 2019, David Harvey and Joris van der Hoeven [HVDH21] derived a fast multiplication algorithm for integers with $O(n \log n)$ time complexity, using fast Fourier transforms and several other tricks to achieve a new time complexity record. This finally achieves the linearithmic algorithm whose existence was conjectured by Schonhage and Strassen in 1971. Schonage and Strassen also conjectured that such an $O(n \log n)$ algorithm would be the fastest possible runtime for an integer multiplication algorithm, although this has not been proven (so there may well be a faster algorithm than Harvey-van der Hoeven's). The algorithm itself is immensely complex so we forgo an explanation of it for the sake of space.

Despite the record-low big-O time complexity, this algorithm has an astronomical constant factor, making it all but useless for any practical use. The "crossover point" where it becomes faster than other multiplication algorithms (like Karatsuba and Toom-Cook above) is so large it is physically impossible to even represent such numbers.

## 4. Fast matrix multiplication

4.1. **Goal.** Matrix multiplication is a fundamental operation in linear algebra. Given a $n \times m$ matrix $A$ and $m \times p$ matrix $B$, their product $AB$ is an $n \times p$ matrix defined as

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}.$$

This algorithm can be shown to have a $O(nmp)$ runtime. Large matrix multiplications are an important tool used across applied disciplines, from machine learning to finance, so it is of particular interest whether a faster runtime is possible. In this section, we discuss progress made on fast matrix multiplication, starting with a treatment of tensor-based divide-and-conquer algorithms. This is followed by an explanation of Strassen's 1969 algorithm and a brief motivation for Strassen's laser method.

We follow the convention used in most research, which is to focus on the most part on *square* matrix multiplication, e.g., multiplying an $n \times n$ matrix $A$ with an $n \times n$ matrix $B$. The above textbook implementation of matrix multiplication (with the sum of $n$ products for each of the $n^2$ entries of the output matrix) has cubic – $O(n^3)$ – runtime. All relevant algorithms discussed here have polynomial runtime. We define $\omega$ as the smallest exponent where $n \times n$ matrix multiplication has runtime $O(n^\omega)$. The goal of fast matrix multiplication research is thus to decrease the exponent $\omega$.

4.2. **Background.** The primary background needed to understand the strategies discussed in this section are tensors. Tensors are effectively the generalization of matrices and vectors across higher dimensions; indeed, matrices are simply 2-dimensional tensors, and vectors are 1-dimensional tensors. The tensors used in this paper are at most 3-dimensional. We now establish some basic properties of, and operations on, tensors.

**Definition 4.1.** Given $N$ $k$-dimensional tensors $T_1, T_2, \ldots, T_N$, each of any size and shape, their direct sum $T_1 \oplus T_2 \oplus \cdots \oplus T_N$ is the $k$-dimensional tensor produced by stacking the tensors diagonally end-to-end, with empty blocks zeroed.

An example: take the following two matrices (2-dimensional tensors):

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$
$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

Their direct sum is computed as follows:

$$A \oplus B = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 5 & 6 \\ 0 & 0 & 7 & 8 \end{bmatrix}.$$

**Definition 4.2.** A *decomposition* of a tensor is a construction of a set of simpler tensors whose sum is the original tensor.

**Definition 4.3.** Given $N$ vectors $v_1, v_2, \ldots, v_N$, each of length $l_1, l_2, \ldots, l_N$, the *tensor product* $v_1 \otimes v_2 \otimes \ldots \otimes v_N$ is an $N$-dimensional tensor with size $l_1 \times l_2 \times \cdots \times l_N$ with entries defined as

$$(v_1 \otimes v_2 \otimes \ldots \otimes v_N)_{i_1,i_2,\ldots,i_N} = (v_1)_{i_1} \times (v_2)_{i_2} \times \ldots \times (v_N)_{i_N}.$$

**Definition 4.4.** If a tensor $T$ can be expressed as the tensor product of some set of vectors, then its rank, $R(T)$, is 1. A tensor's *rank* $R(T)$ is defined as the minimum number of rank-1 tensors needed to construct a decomposition of $T$.

**Definition 4.5.** Given an $n$-dimensional $T$ with dimensions $l_1 \times l_2 \times \ldots \times l_n$, and a vector $v$, *contraction* of $T$ with $v$ over axis $k$ (where $l_k$ is equal to the number of entries in $v$) produces an $(n-1)$-dimensional tensor $T'$ with entries defined as

$$T'_{i_1,i_2,\ldots,i_{k-1},i_{k+1},\ldots,i_n} = \sum_{j=1}^{l_k} T_{i_1,i_2,\ldots,i_{k-1},j,i_{k+1},\ldots,i_n} v_j.$$

Contraction can also be done with multiple vectors (each with a corresponding axis to contract over); in that case, simply contract with each vector and its respective axis one at a time.

For our purposes, the contraction done is much simpler than the definition makes it seem. Also note the following that will be useful for future proofs:

**Proposition 4.6.** *Contraction is distributive across tensor addition: contraction of $T_1 + T_2$ with some vectors is equivalent to the sum of $T_1$ and $T_2$ individually contracted with those vectors.*

*It is also distributive across the direct sum: contraction of $T_1 \oplus T_2$ with the vector produced by the concatenation of $v_1$ and $v_2$ is equivalent to the direct sum of $T_1$ and $T_2$ individually contracted with the respective vectors.*

4.3. **Divide-and-conquer matrix multiplication algorithms.** Like for fast integer multiplication, all recent progress in matrix multiplication algorithms has been in developing fast divide-and-conquer algorithms. An important property of such algorithms use is that a $N \times N$ matrix multiplication algorithm can be converted into a divide-and-conquer algorithm whose complexity can be analyzed using Theorem 2.2. The only requirement is that

the original algorithm be valid over noncommutative rings. Most common matrix multiplication algorithms operate over matrices whose entries are elements of the real numbers. The real numbers are a commutative ring, meaning $ab = ba$ for all $a, b \in \mathbb{R}$. However, the conversion step into a divide-and-conquer algorithm involves applying the algorithm to a matrix block (essentially a matrix of matrices); the entries are now matrices, and matrix multiplication is *not* commutative – $AB \neq BA$ for $A, B \in \mathbb{R}^{N \times N}$. Thus, the algorithm must be valid on inputs whose elements' multiplication is not commutative.

This validity condition is closely related to another property of matrix multiplication algorithms, bilinearity. Bilinear algorithms are defined as those which compute each output term as a sum of product terms, where each product term is the product of 1) a linear combination of elements of the first input matrix 2) a linear combination of elements of the second input matrix. Expanding bilinear algorithms results in monomials which always have elements of the first matrix on the left and elements of the second matrix on the right, so combining like terms to show equivalency to matrix multiplication does not require commutativity. For our purposes, the only algorithms valid over noncommutative rings that are useful are bilinearity. The tensor methods described below are the primary way used to represent bilinear algorithms.

We now prove how to derive a divide-and-conquer algorithm from an $N \times N$ matrix multiplication algorithm meeting the discussed preconditions, and prove its time complexity.

**Theorem 4.7.** *Given an algorithm for $N \times N$ matrix multiplication that is valid over noncommutative rings with $K$ bilinear multiplications, a general matrix multiplication algorithm can be derived with a time complexity of $O\left(M^{\log_N K}\right)$.*

*Proof.* The theorized algorithm is defined as follows. Let the input matrices $A$ and $B$ be $M \times M$, and let the $N \times N$ matrix multiplication algorithm be called the "original algorithm." First, the base case: if $M < N$, use naive matrix multiplication or another algorithm to evaluate them and directly return the result.

Now, the recursive case. To split $A$ and $B$ into equally sized submatrices, we require that $M = N^k$ for some whole number $k$. If this is not true automatically for $M$, we simply *pad* the input matrices with zeroes on the bottom and right (this will simply result in zeroes in the same positions).

We split the matrices into $N \times N$ matrix blocks whose entries are $N^{k-1} \times N^{k-1}$ matrices (each entry, of course, is a submatrix of the respective original input matrix).

We now simply apply the original $N \times N$ algorithm, passing as input the two matrix blocks. Wherever the original algorithm multiplies two entries, we recursively call this newly defined algorithm to compute their product, giving this new algorithm its recursive nature.

Since we have defined the algorithm such that it is valid over matrices whose entries are elements of a noncommutative ring – such as the set of square matrices of a given size – we can guarantee that it can be applied. Many matrix multiplications over the real numbers are valid this way, and as noted above, *all* bilinear matrix multiplication algorithms are valid.

Next, we prove its time complexity. We are given that the original algorithm makes $K$ bilinear multiplications. Because of how this new algorithm is defined, we also know that the entries being multiplied in those $K$ products are each $(N^{k-1})^2$, or equivalently, $\left(\frac{M}{N}\right)^2$.

It also, of course, involves operations like scalar multiplication and addition. The actual number of these non-bilinear operations is irrelevant since it ultimately is a constant factor. Since these operations are not recursive, we represent with them with a separate $f(M)$ term. The associated costs of these operations are directly proportional to the size of the

entry matrices. We note that the entry matrix size is $(N^{k-1})^2$, but by factoring out $\frac{1}{N^2}$ and rewriting $N^{k-1}$ as $\frac{M}{N}$, we have the much nicer $M^2$, concluding that $f(M) = O(M^2)$.

We thus have that, letting the new algorithm be $T(M)$, the recurrence is

$$T(M) = KT\left(\frac{M}{N}\right) + f(M).$$

By the Master Theorem, we must show that $f(M) = O\left(M^{\log_N(K)-\epsilon}\right)$ for some $\epsilon > 0$. First, note that $K > N^2$ for $N \geq 2$ (which is given); this is a proven lower bound on matrix multiplication algorithms [AS81], although we omit the proof for the sake of space. Then, we can define $\epsilon = \log_N(K) - 2$, which is positive due to the above result, and the condition simplifies as

$$\begin{aligned} f(M) &= O\left(M^{\log_N(K)-\epsilon}\right) \\ &= O\left(M^{\log_N(K)-(\log_N(K)-2)}\right) \\ &= O\left(M^2\right) \end{aligned}$$

which, of course, we have shown above. Having met this condition, we can now apply Theorem 2.2, finding

$$T(M) = O\left(M^{\log_N K}\right).$$

■

## 4.4. **Motivation for tensor-based methods.**

**Theorem 4.8.** *Multiplication of $N \times N$ matrices is encoded by a tensor. Decompositions of the tensor into $R$ rank-1 terms yields a $N \times N$ matrix multiplication algorithms containing $R$ multiplications.*

### 4.4.1. *The matrix multiplication tensor.*

**Lemma 4.9.** *A 3-dimensional tensor $T$ can be defined such that tensor contraction with two flattened square input matrices is equivalent to computing the product of those matrices.*

*Proof.* Let the two $N \times N$ input matrices be $A$ and $B$, and let their product be $C$. Then, by the definition of matrix multiplication, each element of $C$ can be computed as follows:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}.$$

This can be represented as a 3-dimensional $N^2 \times N^2 \times N^2$ tensor $T$. For the sake of this representation, we need to "flatten" the matrices, so that $A$, $B$, and $C$ can be treated as vectors. We can still index them using 2-dimensional indexes – $A_{ij}$ etc. are still valid – but they lose their 2-dimensional structure for the purposes of the tensor operations. Using this indexing scheme, $T$ is indexed using three 2-dimensional coordinates. We then define $T$ as

$$T_{(r,s),(t,u),(i,j)} = \delta_{ri}\delta_{st}\delta_{uj}$$

where the Kronecker delta is defined as

$$\delta_{xy} = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}.$$

Contracting this tensor $T$ with the vectors of the flattened input matrices $A$ and $B$ produces a vector $V$. We now prove that $V$ is equivalent to the flattened output matrix $C$. If

we contract $T$ with $A$ over the first axis, and with $B$ over the second axis, by the definition of tensor contraction, the corresponding entries in the output vector are

$$\begin{aligned} V_{(i,j)} &= \sum_{(r,s)} \sum_{(t,u)} T_{(r,s),(t,u),(i,j)} A_{(r,s)} B_{(t,u)} \\ &= \sum_{(r,s)} \sum_{(t,u)} \delta_{ri}\delta_{st}\delta_{uj} A_{(r,s)} B_{(t,u)}. \end{aligned} \tag{4.1}$$

Observing our earlier definition of the entries of $T$, we find that the only terms in this sum which survive are those in which all three Kronecker deltas evaluate to 1. This forces $r = i$, $s = t$, and $u = j$. With fixed $(i, j)$ (to compute one element of the output vector), only one free variable remains and the double sum collapses to a single sum over $s$. Substituting the forced equalities, the sum can thus be rewritten as

$$\begin{aligned} V_{(i,j)} &= \sum_{(r,s)} \sum_{(t,u)} \delta_{ri}\delta_{st}\delta_{uj} A_{rs} B_{tu} \\ &= \sum_{s=1}^{N} A_{is} B_{sj}. \end{aligned}$$

Letting $k = s$, we find

$$V_{(i,j)} = \sum_{k=1}^{N} A_{ik} B_{kj}.$$

This is the exact same as the definition for entries of the matrix product $C_{ij}$ given at the beginning of this proof, so we conclude $V = C$, and the tensor contraction operation is equivalent to matrix multiplication. ■

4.4.2. *Tensor decompositions as matrix multiplication algorithms.*

**Lemma 4.10.** *A decomposition of the tensor $T$ defined in Lemma 4.9 into $R$ rank-1 tensors is equivalent to a matrix multiplication algorithm containing $R$ bilinear multiplications.*

*Proof.* For the sake of this proof, let us assume we have found a decomposition of $T$ into $R$ rank-1 tensors $T_1, \ldots, T_R$ where $T = T_1 + \cdots + T_R$. As the tensors are rank-1, they can be also expressed as the tensor product of three vectors $u_l, v_l, w_l$ such that $T_l = u_l \otimes v_l \otimes w_l$. We can write the contraction performed in Lemma 4.9 as:

$$C = \text{contract}(T, A, B)$$

where contract contracts $A$ into the first axis, $B$ into the second axis, and leaves the third axis for output, as established in 4.9. Einstein summation notation would be appropriate for a more formal proof, but we omit it to avoid confusion between contraction and indexing.

Because contraction is distributive over tensor addition, we can substitute our decomposition of $T$:

$$C = \text{contract}(T_1, A, B) + \cdots + \text{contract}(T_R, A, B).$$

Take the $l$th term. The entries of the tensor product $T_l = u_l \otimes v_l \otimes w_l$ can be computed using the definition of the tensor product:

$$(u_l \otimes v_l \otimes w_l)_{(r,s),(t,u),(i,j)} = (u_l)_{(r,s)} (v_l)_{(t,u)} (w_l)_{(i,j)}$$

To compute the contraction of the $l$th term, we can substitute back into the formula for contraction given in (4.1), replacing the entries of $T$ with our computed entries of $T_l$ and collapsing sums into dot products:

$$\begin{aligned}
\text{contract}(T_l, A, B)_{(i,j)} &= \sum_{(r,s)} \sum_{(t,u)} (T_l)_{(r,s),(t,u),(i,j)} A_{(r,s)} B_{(t,u)} \\
&= (w_l)_{(i,j)} \sum_{(r,s)} (u_l)_{(r,s)} A_{(r,s)} \sum_{(t,u)} (v_l)_{(t,u)} B_{(t,u)} \\
&= (w_l)_{(i,j)} \times \sum_{(r,s)} (u_l)_{(r,s)} A_{(r,s)} \times \sum_{(t,u)} (v_l)_{(t,u)} B_{(t,u)} \\
&= (w_l)_{(i,j)} \times (u_l \cdot A) \times (v_l \cdot B) .
\end{aligned}$$

For the purposes of applying this contraction as part of a matrix multiplication algorithm, $u_l$, $v_l$, and $w_l$ are constant when applying $T$'s contraction with two (variable) matrix inputs. This means the dot products expand to scalar multiplications followed by an addition step. We aim to analyze *bilinear* multiplications: that is, multiplications where both operands are variable with respect to the algorithm inputs, not constant (since constant scalar multiplication, at least in complexity theory, is equivalent to addition). Thus, the only bilinear multiplication in the $l$th contraction is the multiplication of the dot products $(u_l \cdot A) \times (v_l \cdot B)$. The outer multiplication by $(w_l)_{(i,j)}$ is also not bilinear by a similar argument.

We have now established that any one contraction by a rank-1 subtensor of $T$ can be modeled as having the complexity of one bilinear multiplication. Finally, note that the reduction step $\sum_l \text{contract}(T_l, A, B)$ is obviously addition and does not any bilinear multiplications. Given that our decomposition is into $R$ rank-1 subtensors, we conclude that the computational complexity with respect to bilinear multiplications of $\text{contract}(T, A, B)$ is $R$. ■

4.4.3. *The big picture.*

*Proof of Theorem 4.8.* The result follows from Lemmas 4.9 and 4.10. ■

This theorem fits neatly into the big picture. We have just proven a way (tensor decompositions) to represent matrix multiplication algorithms, and we already know from Theorem 4.7 how to use bilinear fixed-size matrix multiplication algorithms to derive general-purpose matrix multiplication algorithms and prove their time complexity. It follows from these two conclusions that, given a $N^2 \times N^2 \times N^2$ tensor with a decomposition into $R$ rank-1 subtensors representing a fast fixed-size matrix multiplication algorithm, a corresponding divide-and-conquer algorithm can be computed with time complexity $O\left(M^{\log_N R}\right)$. This is derived directly from 4.7 with the number of bilinear multiplications being $K = R$ given by Lemma 4.10. One quite fruitful method to find matrix multiplication algorithms, then, effectively boils down to finding efficient decompositions of matrix multiplication tensors to then produce divide-and-conquer methods as outlined above.

4.5. **Strassen's 1969 algorithm.** Strassen's 1969 matrix multiplication algorithm [Str69] established the first subcubic algorithm (one with a time complexity faster than that of $O(n^3)$), proving that $\omega$ can indeed be lesser than 3. Similarly to how the Karatsuba algorithm is a member of the Toom-Cook family of algorithms, Strassen's algorithm can be expressed as a tensor decomposition (tying into Theorem 4.8).

The naive algorithm for divide-and-conquer matrix multiplication uses the result shown in Section 4.3 but, instead of using an optimized $N \times N$ matrix multiplication algorithm, simply uses the naive cubic algorithm for $2 \times 2$ matrices. If we subdivide the input matrices $A$ and $B$ (which, by construction, have been padded so each are $2^k \times 2^k$ square matries for some shared $k$) such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$
$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

then, by Theorem 4.7, their product can be computed using the textbook $2 \times 2$ matrix multiplication algorithm, simply applied to the matrix blocks:

$$A \times B = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

where the products of the submatrices are computed recursively via this algorithm, or via direct multiplication in the base case. It is obvious that this algorithm is bilinear (every term in the result has an $A$ submatrix on the left and a $B$ submatrix on the right, no commutativity assumption needed). Thus, Theorem 4.7 applies directly. The result clearly contains eight terms, each of which contains one bilinear multiplication of an $A$ submatrix with a $B$ submatrix, so the time complexity of this recurisve algorithm is $O\left(n^{\log_2 8}\right) = O\left(n^3\right)$ – no improvement upon direct textbook multiplication.

Strassen's 1969 algorithm follows a very similar process to the above method. The primary distinguishing feature is, instead of using the naive $2 \times 2$ matrix multiplication algorithm with 8 bilinear multiplications, it creates and manipulates intermediate products so that the result only needs 7 multiplications. Continuing with the definition of $A$ and $B$ above, Strassen first defines intermediate products

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) \times B_{11} \\ M_3 &= A_{11} \times (B_{12} - B_{22}) \\ M_4 &= A_{22} \times (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) \times B_{22} \\ M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) . \end{aligned}$$

Then, their product is simply

$$AB = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}.$$

Equivalence to (4.5) can be shown by substituting in the definitions of $M_1, \ldots, M_7$, expanding, and combining like terms.

Note that this is indeed bilinear – expanding the values of $M_1, \ldots, M_7$ using the distributive property (which holds across the ring of matrices as well), the terms only involve multiplications with $A$ entries on the left and $B$ entries on the right. This means the algorithm works for both matrices of real numbers and for matrix blocks. The latter allows us to again apply Theorem 4.7 to produce a new divide-and-conquer matrix multiplication

algorithm. Notably, compared to the naive cubic $2 \times 2$ matrix multiplication algorithm given in (4.5), this contains 7 bilinear multiplications instead of 8 (one for each $M_i$). Its time complexity is thus $O\left(n^{\log_2 7}\right) \approx O\left(n^{2.807}\right)$. This is indeed an improvement on the cubic runtime of the textbook algorithm, or the divide-and-conquer algorithm given above.

*Remark* 4.11. Perhaps a bit surprisingly, Strassen's algorithm can be reduced to an instance of a tensor decomposition-based algorithm given in Theorem 4.8. Recall that a tensor decomposition of the $N \times N$ matrix multiplication tensor $T$ into $R$ rank-1 tensors involves finding $u_i, v_i$, and $w_i$ such that

$$T = \sum_{i=1}^{R} \left(u_i \otimes v_i \otimes w_i\right).$$

Strassen's algorithm is equivalent to a construction of a decomposition of the $2 \times 2$ matrix multiplication tensor $T$ into 7 rank-1 tensors. Each $M_i$ exactly corresponds to one $u_i, v_i, w_i$. First, thinking of our input matrices $A$ and $B$ as flattened vectors of submatrices, each intermediate product $M_i$ can be viewed simply as a product of some linear combination of $A$'s elements with some linear combination of $B$'s elements. Linear combinations of a vector's elements can be expressing as the dot product with some constant vector; thus, define $u_i$ such that $u_i \cdot A$ produces the linear combination of $A$'s elements on the left-hand side of $M_i$, and $v_i$ such that $v_i \cdot B$ produces the linear combination of $B$'s elements on the right-hand side of $M_i$. Then, it follows by construction that $M_i = (u_i \cdot A) \times (v_i \cdot B)$. Finally, define $w_i$ such that it defines the coefficient of $M_i$ in each element of the flattened output vector of the product $AB$. For example, $M_2$ has a coefficient of 1 in $(AB)_{21}$ and a coefficient of $-1$ in $(AB)_{22}$, so $w_2$ would be defined such that $(w_i)_{21} = 1$, $(w_i)_{22} = -1$, and all other entries are 0. Thus, by summing $w_i \times M_i$ across all $i$, the flattened product vector $AB$ will be recovered.

Now, note that this operation is exactly equivalent to performing tensor contraction of each subtensor $u_i \otimes v_i \otimes w_i$! Following from the results shown in Theorem 4.8:

$$w_i \times M_i = w_i \times (u_i \cdot A) \times (v_i \cdot B) = \text{contract}(u_i \otimes v_i \otimes w_i, A, B).$$

It can be shown that the sum of $u_i \otimes v_i \otimes w_i$ is indeed $T$ given our constructions. Thus, this gives a decomposition of the $2 \times 2$ matrix multiplication into 7 rank-1 subtensors which encodes a matrix multiplication algorithm equivalent to the 7-multiplication algorithm Strassen used.

4.6. **Border rank and the asymptotic sum inequality.** We have now established two useful results:

(1) Finding decompositions of matrix multiplication tensors $T$ yields fixed-size bilinear matrix multiplication algorithms.
(2) Fixed-size bilinear matrix multiplications give rise to divide-and-conquer algorithms for general matrix multiplication with polynomial time complexity.

These motivate the next significant step in fast matrix multiplication, and the one which is the source of *all* modern progress: Strassen's 1986 laser method. Importantly, these motivate, but do not directly beget, the laser method, which is built on separate (but closely related) foundations.

Say we have a sequence of matrices $M_i$ which converges such that $\lim_{i \to \infty} M_i = M$ for some $M$. It can be shown that the rank of the sequence's elements are at most the rank of the matrix which the sequence converges to, that is, $\lim_{i \to \infty} R(M_i) = R(M)$. However,

this is *not* guaranteed for higher-dimensional tensors. We thus define border rank as the asymptotic equivalent of rank. It's closely related to rank, but provides us this guarantee of being "resistant" to limits:

**Definition 4.12.** The *border rank* of a tensor $T$ $\underline{R}(T)$ is defined as the small $r$ such that $T$ can be expressed as the limit of a sequence of tensors whose rank is at most $r$.

Now, we can define a result which is crucial for – indeed, justifies – the laser method:

**Theorem 4.13** ( [Sch81])**.** *(Asymptotic sum inequality). Take $K$ tensors $\langle n_i, m_i, p_i\rangle$ $(1 \leq i \leq K)$, each representing the matrix multiplication tensor for multiplying $n_i \times m_i$ with $m_i \times p_i$ matrices. Define the tensor $T$ as the direct sum of the tensors:*

$$T = \bigoplus_{i=1}^{K} \langle n_i, m_i, p_i\rangle.$$

*Then,*

$$\sum_{i=1}^{K} (n_i m_i p_i)^{\omega/3} \leq \underline{R}(T).$$

The proof is omitted for the sake of space.

*Remark* 4.14. This theorem is a little clunky in the context of our past simplified descriptions, since we always assume square matrix multiplication. For optimally multiplying two $N \times N$ matrices, the cost is simply $N^\omega$ (that's how the exponent $\omega$ is defined, after all), or equivalently $(N^3)^{\omega/3}$. Roughly think of $N^3$ as the "volume" of the matrix multiplication tensor $\langle N, N, N\rangle$ (the number of entries). Extending this notion of cost to non-square multiplication, we simply replace the volume $N^3$ with $nmp$, yielding $(n_i m_i p_i)^{\omega/3}$.

Next, note that the direct sum of the matrix multiplication tensors is defined such that individual matrix multiplications are preserved. It's fairly intuitive that the direct sum can be used to compute its component matrix multiplications "in parallel", by concatenating the flattened input matrices and then contracting as usual.

The result of the inequality then follows naturally. The border rank of the direct sum gives an upper bound on the total computational cost of its component matrix multiplications in terms of the exponent $\omega$. Importantly, the total computational cost is defined in terms of the exponent $\omega$. In fact, $\omega$ is the *only* free variable in the inequality. In other words, Theorem 4.13 can be viewed as extending Theorem 4.8 to direct sums (although it does not directly follow). This allows us to derive new upper bounds for $\omega$ and is the key mechanism used by the laser method.

4.7. **The laser method.** A comprehensive explanation of the laser method [Str86] is omitted as even a brief treatment would span several pages – indeed, several more expository papers could be written on the laser method and its extensions alone. For our purposes, explaining the goal of the laser method is much more realistic and fruitful. A useful note to keep in mind throughout the following explanation is this: *the ultimate goal of the laser method is to produce a tensor $T$ which possesses maximal utility when applied to the asymptotic sum inequality to tightly bound $\omega$.*

The laser method starts with a tensor which has a low border rank and converting it into a new tensor that Theorem 4.13 can be applied to. The setup is as follows: take a tensor $T$ with relatively low border rank, canonically, the Coppersmith–Winograd tensor

$CW_q$. $T$ most likely isn't equivalent to a direct sum of matrix multiplication tensors, which is the form it needs to be in to apply Theorem 4.13. For the purposes of the laser method, we need to show that the tensor $T$ decomposes into matrix multiplication tensors (this is proven for $CW_q$; proof omitted). However, this doesn't make it a border sum, as the sum of the decomposition will result in the subtensors overlapping. The laser method operates as follows: take a high power of $T$, then zero some terms, so that the resulting tensor is a valid direct sum of matrix multiplication tensors.

But which terms should we zero? Our goal is to produce a tensor which is of maximum utility to our ultimate goal, which is to tightly bound $\omega$. The border rank of our tensor does not increase when zeroing terms (proof omitted), so we can effectively let the right-hand side of the inequality be constant:

$$\sum_{i=1}^{K} (n_i m_i p_i)^{\omega/3} \leq (\text{constant}).$$

Then, the goal is to maximize the left-hand side sum, forcing $\omega$ to be smaller to fulfill the inequality. To achieve this, the laser method zeros terms of the tensor to make it a direct sum while maximizing the left-hand side. How it does this and optimizes that tradeoff is an extremely complex multi-step process which is the core of the laser method.

Note that there are two "strings" we can pull to change the usefulness of a laser method result. The first is the volume of the produced matrix; higher volumes on the left-hand side of the asymptotic sum inequality force $\omega$ lower. The second is the border rank of the tensor we apply the laser method (strictly, the border rank of the power of the tensor whose terms we zero); a lower border rank on the right-hand side of the inequality similarly forces $\omega$ lower. Choosing inputs and powers to use in the laser method thus requires balancing these two "strings."

Finally, we now can use the tensor produced by the laser method as described above, plugging in to find a new bound on $\omega$. Most modern progress in fast matrix multiplication boils down to improving the laser method or simply applying it to higher and higher powers of the Coppersmith–Winograd tensor.

## 5. Acknowledgements

I would like to thank my TA, Rajan Foster, for his helpful suggestions and resources throughout the writing process, and my classmates whom I peer reviewed and discussed our papers with. I would also like to thank Simon Rubinstein-Salzedo for providing this opportunity and his help in understanding how to format and write the paper.

## References

[AS81] A. Alder and V. Strassen. On the algorithmic complexity of associative algebras. *Theoretical Computer Science*, 15(2):201–211, 1981.

[BHS80] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, September 1980.

[Bod07] Marco Bodrato. Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, pages 116–133, Berlin, Heidelberg, 2007. Springer.

[Coo66] Stephen A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Department of Mathematics, Harvard University, 1966.

[HVDH21] David Harvey and Joris Van Der Hoeven. Integer multiplication in time $O(N\mathrm{log}\, n)$. *Annals of Mathematics*, 193(2), March 2021.

[KO63] Anatoly A. Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[Sch81] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, August 1981.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[Str86] V. Strassen. The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 49–54, October 1986. ISSN: 0272-5428.

[Too63] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.

*Email address*: aadishv0@gmail.com