Universitat Autònoma de Barcelona

Facultat de ciències

Bachelor's Thesis in Mathematics

MATRIX MULTIPLICATION ALGORITHMS WITH TENSORS

Laura Cara Sala

Supervised by Joaquim Roé Vellvé

2023-2024

Abstract

My bachelor's thesis is a first approach to tensors used in optimal matrix multiplication algorithms. The project discusses Strassen's algorithm, the first improvement in matrix multiplication speed, and introduces the concept of tensor in the context of matrix multiplication. Using this structure, new matrix multiplication algorithms were created by an artificial intelligence agent called AlphaTensor in 2022.

The project aims to study the theoretical complexity of these algorithms applying the so-called "Master Method" for running times of recursive algorithms and compare it to real simulations by programming these tensor algorithms in C. The simulation results of threshold and times of computation obtained are a good approximation given the memory constraints for this work and allow suggesting other lines of research and optimizations of the algorithm. Finally, an optimal tensor algorithm is proposed as a final program, which is a faster alternative to the ordinary method.

Acknowledgments

Firstly, I would like to thank my tutor Joaquim Roé for his guidance in this complex project, both with the theory that the subject required and the programs needed -they have not been few-; and also for helping me by executing long computations in his department computer that ended up taking more than 55 hours per multiplication.

I need to make special recognition for my little brother, who learned the topic of the project faster than me, to be able to help me make any needed computations with his larger-memory-than-mine computer.

Thank you to Marc, my partner in life, for his daily support, even on the gray days.

Also, I want to thank my Mom and sister for their encouragement and aim to help in every way possible, as well as my degree-mates, Luismi, Rocío and Montse, because without them, I would not be here.

And acknowledgments to my friends and family for their encouragement and for always being willing to hear me complain.

Thank you to all of you.

And this is for you, Dad.

Contents

1 Introduction

In the era of the information and big data, where people want more quantity and quality, the matrices are the optimal solution to store and analyse information, and it is only logical that to move and use this information, a simple computation such as matrix multiplication is the perfect option to use.

The multiplication of matrices is used in a lot of areas. In computer graphics and processing images, matrix multiplications are used to deal with images: to rotate, scale and translate them. In machine learning, many algorithms, such as neural networks and optimization techniques like modeling economic relationships and performing statistical analysis, rely on matrix operations to analyze large datasets. The matrix multiplications are also included in structural analysis, electrical circuits, and quantum mechanics problems. Also, matrices are also used in encryption algorithms to secure data; this process involves matrix transformations to encrypt and decrypt messages. [4]

Therefore, for more than five decades, there has been an ongoing search to find even a slight improvement in the multiplication process. In this project, we will introduce the first improved algorithm for multiplications of 2×2 matrices using the algebraic concept of "tensors" and after, we will replicate the study in the general case, considering the new artificial intelligence tool to find optimal matrix multiplication algorithms, "AlphaTensor" presented less than two years ago.

After the theoretical research, we will put this knowledge into practice through simulations I programmed to find the size of matrices in which these new algorithms begin to be useful, and to study if these new algorithms are truly an advancement in the search.

2 Strassen's algorithm

In 1969, [5] Volker Strassen marked the start of the search of finding the optimal algorithm for matrix multiplication. His breakout was achieved by discovering a faster-than-ordinary algorithm for multiplying 2×2 matrices based on the divide-and-conquer strategy. This algorithm only needs 7 multiplications, while the standard method's needs 8.

2.1 Strassen's method

For his algorithm, he required two matrices $A_{n\times n}$ and $B_{n\times n}$, where n is a power of 2, and a result matrix $C = A \cdot B$ of the same size. The algorithm aims to divide all three matrices into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$ and A_{ij}, B_{ij}, and C_{ij} represent the submatrices of A, B, and C, respectively, with $i, j \in 1, 2$ as shown next:

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = C \tag{1}$$

Next, seven multiplications of linear combinations of A and B are computed in the following manner:

$$\begin{aligned}
P_1 &= A_{11} \cdot (B_{12} - B_{22}) = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= (A_{11} + A_{12}) \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= (A_{21} + A_{22}) \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot (B_{21} - B_{11}) = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}$$

Lastly, the P_i matrices are added or subtracted from the needed submatrices of the four that compose C:

$$C_{11} = P_5 + P_4 - P_2 + P_6 \tag{2}$$
$$C_{12} = P_1 + P_2 \tag{3}$$
$$C_{21} = P_3 + P_4 \tag{4}$$
$$C_{22} = P_5 + P_1 - P_3 - P_7 \tag{5}$$

Dividing the matrices into submatrices allows applying the algorithm recursively until the multiplication is a simple product between two numbers.

Proof. Recall that the expression for the ordinary method of multiplication is:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \tag{6}$$

Following Strassen's algorithm and replacing in C_{ij} its respective P_i expressions, we see that we obtain the same result as in the ordinary method.For instance, in the case $i, j = 1, 1$ in which $C_{11} = P_5 + P_4 - P_2 + P_6$, we get:

$$\begin{aligned}
C_{11} = \; & \mathbf{A_{11} \cdot B_{11}} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
& - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\
& - A_{11} \cdot B_{22} - A_{12} \cdot B_{22} \\
& - A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + \mathbf{A_{12} \cdot B_{21}} \\
\hline
= \; & \mathbf{A_{11} \cdot B_{11}} + \mathbf{A_{12} \cdot B_{21}}
\end{aligned}$$

Resulting in the same expression seen in (6).

□

2.2 Strassen's algorithm with tensors

The concept of tensors in algebra is a very general idea that allows working with vector spaces such as linear, bilinear or multilinear maps and the relationships between them.

The specific type of tensors we will be using corresponds to the bilinear map of matrices $M_{r\times s}$ of size $r \times s$:

$$M_{r\times s} \times M_{s,t} \longrightarrow M_{r\times t}$$

An example of this type of bilinear map would be the matrix multiplication, meaning that we can identify it with a tensor in this space.

Inside this type of tensor, there is a subtype called "elemental tensors", analogous to the concept of linear map of rank 1, and their definition is given by using the dual space definition. The dual space $M^*_{r\times s}$ in our specific case is the set of linear maps $M_{r\times s} \longrightarrow \mathbb{R}$. Its dimension is $r \times s$, matching the matrix size. A basis of this set is formed by a collection of maps $v_{i,j}$ such that for a given matrix A, $v_{i,j}(A)$ is the coefficient (i, j) of A.

If we consider two elements v, w of the dual $M^*_{r\times s}$ and an element C of $M_{r,t}$, we can define a bilinear map:

$$\begin{aligned} f : M_{r\times s} \times M_{s,t} \longrightarrow & M_{r,t} \\ (A, B) \longrightarrow & v(A)w(B)C \end{aligned}$$

Notice that $v(A)$ and $w(B)$ are numbers multiplied by C, therefore the image of any (A, B) is a multiple of the fixed matrix C; and for that reason, f is considered to be "elemental".

Given that the rank k of a tensor f is defined as the minimum k such that f can be written as the sum of k elementary tensors, any bilinear map can be written as a finite sum of k elemental maps, which are the "elemental tensors" named above.

The tensors we are going to work with are just a decomposition of k "elemental tensors" of a tensor of a map $(A, B) \longrightarrow A \cdot B$. Therefore, the tensor will have k rows of elemental tensors, each one constituted of three vectors, as explained before: In the first column, there is a first element v of the dual $M^*_{r\times s}$, in the second column, an element w of the dual $M^*_{s\times t}$ and in the third column, an element C of the space $M_{r\times t}$.

Proposition 1. *The tensor for Strassen's algorithm would be the following:*

$$P = \begin{bmatrix} [1,0,0,0] & [0,1,0,-1] & [0,1,0,1] \\ [1,1,0,0] & [0,0,0,1] & [-1,1,0,0] \\ [0,0,1,1] & [1,0,0,0] & [0,0,1,-1] \\ [0,0,0,1] & [-1,0,1,0] & [1,0,1,0] \\ [1,0,0,1] & [1,0,0,1] & [1,0,0,1] \\ [0,1,0,-1] & [0,0,1,1] & [1,0,0,0] \\ [1,0,-1,0] & [1,1,0,0] & [0,0,0,-1] \end{bmatrix} \tag{7}$$

Proof. Following the previous tensor explanation, for C_{11} the only rows needed are the ones in which the first position of the third column isn't null. Therefore, rows 2, 4, 5 and 6.

- Let's explain in detail the process in row 2:

  The first column has the coefficients of the linear combination needed for matrix A, i.e:

  $$1 \cdot A_{11} + 1 \cdot A_{12} + 0 \cdot A_{21} + 0 \cdot A_{22} = A_{11} + A_{12}$$

  And the same way for matrix B:

  $$0 \cdot B_{11} + 0 \cdot B_{12} + 0 \cdot B_{21} + 1 \cdot B_{22} = B_{22}$$

We multiply these two expressions and save them in an auxiliary matrix:

$$P_{2,3} = (A_{11} + A_{12}) \cdot B_{22}$$

And this already looks a lot like Strassen's method shown in the previous section. We follow the same process for the other rows needed:

- Row 4:

$$0 \cdot A_{11} + 0 \cdot A_{12} + 0 \cdot A_{21} + 1 \cdot A_{22} = A_{22}$$
$$-1 \cdot B_{11} + 0 \cdot B_{12} + 1 \cdot B21 + 0 \cdot B_{22} = -B_{11} + B_{21}$$
$$P_{4,3} = A_{22} \cdot (-B_{11} + B_{21})$$

- Row 5:

$$1 \cdot A_{11} + 0 \cdot A_{12} + 0 \cdot A_{21} + 1 \cdot A_{22} = A_{11} + A_{22}$$
$$1 \cdot B_{11} + 0 \cdot B_{12} + 0 \cdot B21 + 1 \cdot B_{22} = B_{11} + B_{22}$$
$$P_{5,3} = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

- Row 6:

$$0 \cdot A_{11} + 1 \cdot A_{12} + 0 \cdot A_{21} - 1 \cdot A_{22} = A_{12} - A_{22}$$
$$0 \cdot B_{11} + 0 \cdot B_{12} + 1 \cdot B21 + 1 \cdot B_{22} = B_{12} + B_{21}$$
$$P_{6,3} = (A_{12} - A_{22}) \cdot (B_{12} + B_{21})$$

Now, the third column of the tensor for its corresponding row tells us where we add this auxiliary P_i. In the case of C_{11}, the rows that conforms it are:

$$C_{11} = 0 \cdot P_1 - 1 \cdot P_2 + 0 \cdot P_3 + 1 \cdot P_4 + 1 \cdot P_5 + 1 \cdot P_6 + 0 \cdot P_7 = -P_2 + P_4 + P_5 + P_6$$

Matching the result obtained with the manual Strassen's method in (2).

The same process is followed to obtain C_{12}, C_{21} and C_{22} $\square$

3 AlphaTensor for matrix multiplications

3.1 AlphaTensor

On October 5, 2022, a paper presenting a new tool for finding new algorithms for matrix multiplication through artificial intelligence was published by Alhussein Fawzi, Matej Balog, Bernardino Romera-Paredes, Demis Hassabis and Pushmeet Kohli. [2]

The paper introduced AlphaTensor, a system built upon AlphaZero, an agent renowned for surpassing human-level performances in board games such as chess, among others. AlphaTensor uses the perspective of AlphaZero to convert the matrix multiplication problem to a single-player game on a three-dimensional board (representing a tensor), which quantifies the distance of a given current state from the optimal solution. The agent uses reinforcement learning to play the game, in which a rank 1 tensor or, equivalently, a triplet of vectors, is considered and penalizes each step to help find the shortest route. It improves over time, discovering the fastest algorithms ever known, which are beyond human capability. [1]

For example, for the multiplication of two matrices:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \end{pmatrix} = \tag{8}$$

$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} & A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} & A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33} & A_{11}B_{14} + A_{12}B_{24} + A_{13}B_{34} \\ A_{12}B_{11} + A_{22}B_{21} + A_{23}B_{31} & A_{12}B_{12} + A_{22}B_{22} + A_{23}B_{32} & A_{12}B_{13} + A_{22}B_{23} + A_{23}B_{33} & A_{12}B_{14} + A_{22}B_{24} + A_{23}B_{34} \end{pmatrix} \tag{9}$$

This ordinary multiplication takes $2 \cdot 3 \cdot 4 = 24$ multiplications, while the following tensor discovered with AlphaTensor takes only 20:

$$P = \begin{bmatrix}
[0,1,0,0,0,0] & [0,1,0,0,0,-1,0,0,0,-1,0,0] & [0,-1,0,1,0,0,0,0] \\
[0,1,-1,0,0,0] & [0,1,0,0,0,0,0,0,0,-1,0,0] & [0,1,0,0,0,1,0,0] \\
[0,1,-1,0,0,1] & [0,1,0,0,0,-1,0,-1,0,-1,0,0] & [0,0,0,-1,0,-1,0,0] \\
[0,1,-1,0,-1,1] & [0,1,0,0,0,-1,0,-1,1,0,0,0] & [0,0,0,0,0,1,0,0] \\
[0,0,0,0,1,0] & [0,0,0,0,0,0,0,1,0,0,0,0] & [0,0,0,0,0,-1,0,1] \\
[0,1,-1,-1,-1,0] & [0,1,0,0,0,0,0,0,1,0,0,0] & [-1,0,0,0,0,-1,0,0] \\
[0,0,1,0,0,-1] & [0,1,0,0,0,-1,0,-1,0,-1,-1,-1] & [0,0,0,-1,0,0,0,0] \\
[1,0,0,0,0,1] & [0,0,0,1,0,0,0,0,0,0,1,0] & [0,0,0,1,0,0,1,0] \\
[1,0,1,0,0,0] & [0,1,0,0,0,0,0,0,0,0,-1,0] & [0,0,-1,1,0,0,0,0] \\
[0,0,0,1,1,0] & [1,0,0,0,0,0,0,0,-1,0,0,0] & [1,0,0,0,1,0,0,0] \\
[0,0,0,1,0,1] & [0,0,0,1,0,0,0,0,1,0,0,0] & [0,0,0,0,0,0,-1,1] \\
[0,0,0,0,0,1] & [0,0,0,1,0,0,0,0,0,0,0,-1] & [0,0,0,-1,0,0,0,-1] \\
[1,0,0,-1,-1,0] & [1,0,0,0,-1,0,-1,0,-1,0,0,0] & [1,0,0,0,0,0,1,0] \\
[1,1,0,-1,-1,0] & [0,1,0,0,1,0,1,0,1,0,0,0] & [1,0,0,0,0,0,0,0] \\
[0,1,0,0,0,0] & [0,0,0,0,0,0,1,0,0,0,0,0] & [-1,0,1,0,0,0,0,0] \\
[0,0,0,0,1,0] & [1,0,0,0,-1,0,0,0,-1,0,0,0] & [0,0,0,0,-1,0,1,0] \\
[1,0,0,0,0,0] & [0,0,1,0,0,0,0,0,0,0,-1,0] & [0,0,1,0,0,0,1,0] \\
[1,0,1,0,0,0] & [0,1,0,0,0,0,0,0,0,0,0,0] & [-1,1,1,-1,0,0,0,0] \\
[0,0,0,1,0,1] & [0,0,0,0,0,0,0,0,1,0,0,0] & [0,0,0,0,1,-1,1,-1] \\
[1,0,0,-1,0,0] & [1,0,1,1,-1,0,-1,0,-1,0,0,0] & [0,0,0,0,0,0,-1,0]
\end{bmatrix} \tag{10}$$

Proof. In the tensor structure, to calculate C_{ij} we will only need the rows for which the corresponding position isn't null. Meaning:

$$C_{1,1} = - P_{6,3} + P_{10,3} + P_{13,3} + P_{14,3} - P_{15,3} - P_{18,3} \tag{11}$$
$$C_{1,2} = - P_{1,3} + P_{2,3} + P_{18,3} \tag{12}$$
$$C_{1,3} = - P_{9,3} + P_{15,3} + P - 17,3 + P_{18,3} \tag{13}$$
$$C_{1,4} = P_{1,3} - P_{3,3} - P_{7,3} + P_{8,3} + P_{9,3} - P_{12,3} - P_{18,3} \tag{14}$$
$$C_{2,1} = P_{10,3} - P_{16,3} + P_{19,3} \tag{15}$$
$$C_{2,2} = P_{2,3} - P_{3,3} + P_{4,3} - P_{5,3} - P_{6,3} - P_{19,3} \tag{16}$$
$$C_{2,3} = P_{8,3} - P_{11,3} + P_{13,3} + P_{16,3} + P_{17,3} + P_{19,3} - P_{20,3} \tag{17}$$
$$C_{2,4} = P_{5,3} + P_{11,3} - P_{12,3} - P_{19,3} \tag{18}$$

Therefore, to calculate $C_{1,1}$ would be needed:

$$\begin{aligned}
P_{6,3} =& (A_{12} - A_{13} - A_{21} - A_{22}) \cdot (B_{12} + B_{31}) = A_{12} \cdot B_{12} - A_{13} \cdot B_{12} - A_{21} \cdot B_{12} - A_{22} \cdot B_{12} + A_{12} \cdot B_{31} \\
& - A_{13} \cdot B_{31} - A_{21} \cdot B_{31} - A_{22} \cdot B_{31} \\
P_{10,3} =& (A_{21} + A_{22}) \cdot (B_{11} - B_{31}) = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} - A_{21} \cdot B_{31} - A_{22} \cdot B_{31} \\
P_{13,3} =& (A_{11} - A_{21} - A_{22}) \cdot (B_{11} - B_{21} - B_{23} - B_{31}) = A_{11} \cdot B_{11} - A_{11} \cdot B_{21} - A_{11} \cdot B_{23} - A_{11} \cdot B_{31} \\
& + A_{21} \cdot B_{21} + A_{21} \cdot B_{23} + A_{21} \cdot B_{31} - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} + A_{22} \cdot B_{23} + A_{22} \cdot B_{31} \\
P_{14,3} =& (A_{11} + A_{12} - A_{21} - A_{22}) \cdot (B_{12} + B_{21} + B_{23} + B_{31}) = A_{11} \cdot B_{12} + A_{11} \cdot B_{21} + A_{11} \cdot B_{23} + A_{11} \cdot B_{31} \\
& + A_{12} \cdot B_{12} + A_{12} \cdot B_{21} + A_{12} \cdot B_{23} + A_{12} \cdot B_{31} - A_{21} \cdot B_{12} - A_{21} \cdot B_{21} - A_{21} \cdot B_{23} - A_{21} \cdot B_{31} \\
& - A_{22} \cdot B_{12} - A_{22} \cdot B_{21} - A_{22} \cdot B_{23} - A_{22} \cdot B_{31} \\
P_{15,3} =& A_{12} \cdot B_{23} = A_{12} \cdot B_{23} \\
P_{18,3} =& (A_{11} + A_{13}) \cdot B_{12} = A_{11} \cdot B_{12} + A_{13} \cdot B_{12}
\end{aligned}$$

After computing the additions and subtractions indicated in (12), the result is the following:

$$\begin{aligned}
C_{1,1} =& \\
& - A_{12} \cdot B_{12} + A_{13} \cdot B_{12} + A_{21} \cdot B_{12} + A_{22} \cdot B_{12} - A_{12} \cdot B_{31} + \mathbf{A_{13} \cdot B_{31}} + A_{21} \cdot B_{31} + A_{22} \cdot B_{31} \\
& + A_{21} \cdot B_{11} + A_{22} \cdot B_{11} - A_{21} \cdot B_{31} - A_{22} \cdot B_{31} \\
& + \mathbf{A_{11} \cdot B_{11}} - A_{11} \cdot B_{21} - A_{11} \cdot B_{23} - A_{11} \cdot B_{31} + A_{21} \cdot B_{21} + A_{21} \cdot B_{23} + A_{21} \cdot B_{31} - A_{22} \cdot B_{11} \\
& \quad + A_{22} \cdot B_{21} + A_{22} \cdot B_{23} + A_{22} \cdot B_{31} \\
& + A_{11} \cdot B_{12} + A_{11} \cdot B_{21} + A_{11} \cdot B_{23} + A_{11} \cdot B_{31} + A_{12} \cdot B_{12} + \mathbf{A_{12} \cdot B_{21}} + A_{12} \cdot B_{23} + A_{12} \cdot B_{31} \\
& - A_{21} \cdot B_{12} - A_{21} \cdot B_{21} - A_{21} \cdot B_{23} - A_{21} \cdot B_{31} - A_{22} \cdot B_{12} - A_{22} \cdot B_{21} - A_{22} \cdot B_{23} - A_{22} \cdot B_{31} \\
& - A_{12} \cdot B_{23} \\
& - A_{11} \cdot B_{12} - A_{13} \cdot B_{12} \\
\hline
=& \mathbf{A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31}}
\end{aligned}$$

And we observe that all the remaining factors simplify, and the result is indeed the same as in the ordinary product seen in (9).

By repeating the same procedure for the remaining three cases, we can establish that the result is correct for all the values of the matrix C. □

3.2 Master Method for asymptotic behaviour

We want to study the speed of the newfound algorithms for multiplying matrices to prove that asymptotically, they are better than the ordinary method. To do so, in this section we will study the asymptotic runtime of the Divide and Conquer algorithm, an algorithm that does what its name implies: divides a certain problem of size n into several sub-problems of a smaller size. The problems are resolved recursively; therefore, what we are looking for is a theorem that solves a recurrence of the form: $T(n) = \alpha T(n/\beta) + f(n)$ in which n is the size of the problem and n/β is the size of the α sub-problems. This theorem is called the master theorem and was presented by Jon Bentley, Dorothea Blostein and James B. Saxe in 1980 as a method to unify solving recurrences of the previous form [7].

Notation 1 (O-notation)**.** *[3.1, pg 47 [6]] A certain function $f(n)$ belongs to the set $O(g(n))$ if there exists a certain positive constant c that makes $c \cdot g(n)$ an **upper** bond for $f(n)$ for a large enough n. Meaning:*

$$O(g(n)) = \{f(n) : f(n) < c \cdot g(n), \forall n > n_0\}$$

Therefore, we will be using the O-notation to make reference to the worst-case scenario runtime of a function.

Notation 2 (Ω-notation)**.** *[3.1, pg 48 [6]] A certain function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a certain positive constant c that makes $c \cdot g(n)$ a **lower** bond for $f(n)$ for a large enough n. Meaning:*

$$\Omega(g(n)) = \{f(n) : c \cdot g(n) < f(n), \forall n > n_0\}$$

Therefore, will be using the Ω-notation to make reference to the best case-scenario runtime of a function.

Notation 3 (Θ-notation)**.** *[3.1, pg 44 [6]] A certain function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist certain positive constants c_1 and c_2 such that $f(n)$ is larger than $c_1 \cdot g(n)$ but smaller than $c_2 \cdot g(n)$, for a large enough n. Meaning:*

$$\Theta(g(n)) = \{f(n) : c_1 \cdot g(n) < f(n) < c_2 \cdot g(n), \forall n > n_0\}$$

Therefore, we will be using the Θ-notation to consider the average or optimal runtime of a function.

Theorem 1 (Master theorem)**.** *[Theorem 4.1 [6]] Considering $\alpha \geq 1$ and $\beta > 1$ constants and $f(n)$ a certain function, we define $T(n)$ as the non-negative recurrence of integers:*

$$T(n) = \alpha T(n/\beta) + f(n),$$

where if n is not a β multiple, we can interpret n/β to its nearest integer, either up or down.

Then $T(n)$ has the following asymptotic limits:

1. *If $f(n) = O(n^{\log_\beta \alpha - \epsilon})$ for a constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_\beta \alpha})$.*
2. *If $f(n) = \Theta(n^{\log_\beta \alpha})$, then $T(n) = \Theta(n^{\log_\beta \alpha \log n})$.*
3. *If $f(n) = \Omega(n^{\log_\beta \alpha + \epsilon})$ for $\epsilon > 0$ a constant, and if $\alpha f(n/\beta) \leq \gamma f(n)$ for some constant $\gamma < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

3.2.1 Master Method for Strassen's Algorithm

In Strassen's algorithm, for the multiplication of two matrices of size n, each iteration of the recurrence computes the operations needed for 7 multiplications of size $n/2$ and the 18 additions of the matrices of size $(n/2, n/2)$. Hence, the recurrence for the complexity of the method is given by:

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \tag{19}$$

In this particular case, the Master Theorem can be applied with the following values:

$$\begin{aligned} \alpha =& 7 \\ \beta =& 2 \\ f(n) =& 18\left(\frac{n}{2}\right)^2 \approx \Theta(n^2) \approx \Theta(n^{\log_2 7-\epsilon}) \end{aligned}$$

Applying case (1) of the theorem, we have:

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807\dots})$$

And for the ordinary method, n^3 multiplications and $n^2(n-1)$ additions are needed:

$$T_{ord}(n) = n^3 + n^2(n-1) = \Theta(n^3)$$

Therefore, the theorem allows us to affirm that the tensor method is asymptotically better than the ordinary method.

3.2.2 Master Method applied in the general case

In the general case of the algorithm for the multiplications of two matrices with sizes $m \times n$ and $n \times p$ where

$$m = r^l, n = s^l, p = t^l \tag{20}$$

for fixed values of r, s and t, a certain l and the corresponding tensor of rank k, an iteration of the process involves:

k multiplications
a additions of size $r \times s$
b additions of size $s \times t$
c additions of size $r \times t$

Then, each iteration of the method will need k multiplications of the matrices of sizes $\frac{m}{r} \times \frac{n}{s}$ and $\frac{n}{s} \times \frac{p}{t}$, reducing the dimensions by a factor, and also three sets of additions, each corresponding to the matrix size additions used in the iteration of the algorithm. Therefore, the recurrence for the complexity of the method is:

$$T(m,n,p) = kT\Big(\frac{m}{r}, \frac{n}{s}, \frac{p}{t}\Big) + a\frac{m}{r}\frac{n}{s} + b\frac{n}{s}\frac{p}{t} + c\frac{m}{r}\frac{p}{t} \tag{21}$$

Now, to unify all the variables into one and have something more similar to Strassen's recurrence, we can fix p as the main variable of the recurrence, and taking into consideration (20) the recurrence will only depend on p and the fixed values l, r, s, t. Applying this, the recurrence will be:

$$T(p) = kT\Big(\frac{p}{t}\Big) + ar^{l-1}s^{l-1} + bs^{l-1}t^{l-1} + cr^{l-1}t^{l-1}$$

Therefore, in the general case of tensor algorithm multiplication, the Master Theorem can be applied with the following values:

$$\begin{aligned} \alpha &= k \\ \beta &= t \\ f(p) &= ar^{l-1}s^{l-1} + bs^{l-1}t^{l-1} + cr^{l-1}t^{l-1} \end{aligned}$$

And substituting $l = \log_t p$ due to the relation in (20) we see that:

$$\begin{aligned} \Theta(r^{l-1}s^{l-1}) &= \Theta((rs)^{\log_t p - 1}) \stackrel{1}{=} \Theta\Big((t^{\log_t rs})^{\log_t p-1} = \Theta(t^{\log_t rs \log_t p - \log_t rs})) = \Theta\Big(\frac{p^{\log_t rs}}{rs}\Big) \\ \Theta(s^{l-1}t^{l-1}) &= \Theta((st)^{\log_t p - 1}) \stackrel{1}{=} \Theta\Big((t^{\log_t st})^{\log_t p-1}) = \Theta(t^{\log_t st \log_t p - \log_t st}) = \Theta\Big(\frac{p^{\log_t st}}{st}\Big) \\ \Theta(r^{l-1}t^{l-1}) &= \Theta((rt)^{\log_t p - 1}) \stackrel{1}{=} \Theta((t^{\log_t rt})^{\log_t p-1}) = \Theta(t^{\log_t rt \log_t p - \log_t rt}) = \Theta\Big(\frac{p^{\log_t rt}}{rt}\Big) \end{aligned}$$

We know that $f(p) = \Theta\Big(p^{\max\left(\frac{p^{\log_t rs}}{rs}, \frac{p^{\log_t st}}{st}, \frac{p^{\log_t rt}}{rt}\right)}\Big) \approx \Theta(p^{\log_t k - \epsilon})$ as long as $k > rs, k > st, k > rt$.

And the Master Theorem tells us that

$$T(p) = \Theta(p^{\log_t k})$$

If $k < rst$ (in the found tensors it is always true due to the fact that this new algorithm is supposed to be an improvement), then this result is better than the ordinary product, in which are needed nmp multiplications and $mp(n-1)$ additions, and so its complexity is:

$$T_{ord}(p) = mnp + mp(n-1) = p^{\log_t r}p^{\log_t s}p + p^{\log_t r}p(p^{\log_t s} - 1) = \Theta(p^{\log_t r + \log_t s + 1}) = \Theta(p^{log_t rst}) \tag{22}$$

---

[1] $t^{\log_t a} = a$

3.3 Efficiency thresholds

We have seen that, asymptotically, the tensor methods are much better than the ordinary ones, but that doesn't apply to matrices of all proportions. In small sizes, the additional steps required by the tensor methods and their recursive nature make the ordinary method optimal. Therefore, the use of the tensor method will only be helpful for a matrix of a certain magnitude. The Master Theorem cannot help us with this threshold, so. in this section, we calculate it manually, solving the recurrence of the complexity of the method.

Note that this section only takes into account mathematical operations. In actual implementations, other instructions (e.g. function calls in the recursive methods) need to be run and may increment the actual threshold.

3.3.1 Efficiency Threshold between Ordinary and Tensor Methods

We have seen in section (3.2.1) that the complexity for Strassen's method is given by (19). To be able to work with it and compare it to the ordinary method, we first need to solve this recurrence:

Proposition 2 (Complexity in Strassen's algorithm)**.** *The solution to the recurrence (19) is:*

$$T(n) = 7^l + \frac{18}{3}(7^l - 4^l)$$

Proof. To solve the recurrence

$$T(n) = 7T\Big(\frac{n}{2}\Big) + 18\Big(\frac{n}{2}\Big)^2$$

We consider $T(n) = P(n) + S(n)$, where $P(n)$ are the multiplications of the algorithm and $S(n)$ the additions.

Knowing that $n = 2^l$, the number of multiplications is trivial to prove.

$$P(n) = 7P\Big(\frac{n}{2}\Big) = 7 \cdot 7P\Big(\frac{n}{4}\Big) = ... = 7^l P(1) = 7^l$$

For the additions, to compute $S(n)$ we need the 18 additions of Strassen's algorithm of size $\frac{n}{2} \times \frac{n}{2}$ plus the additions needed for each matrix of size $\frac{n}{2}$ in each of the 7 multiplications. That is:

$$S(n) = 18\Big(\frac{n}{2}\Big)^2 + 7S\Big(\frac{n}{2}\Big)$$

If we develop this recurrence, we will have:

$$\begin{aligned}
S(n) =& 18\Big(\frac{n}{2}\Big)^2 + 7S\Big(\frac{n}{2}\Big) = 18\Big(\frac{n}{2}\Big)^2 + 7 \cdot 18\Big(\frac{n}{4}\Big)^2 + 7^2 \cdot 18\Big(\frac{n}{8}\Big)^2 + ... + 7^{l-1} \cdot 18\Big(\frac{n}{2^l}\Big)^2 \\
=& 18\Big(\frac{n}{2}\Big)^2\left(1 + 7\frac{1}{2} + 7^2\frac{1}{2^2} + 7^3\frac{1}{2}^3 + ... + 7^{l-1}\frac{1}{2^{l-1}}\right) = 18\Big(\frac{n}{2}\Big)^2 \sum_{i=0}^{l-1} 7^k\Big(\frac{1}{2^k}\Big)^2 = 18\Big(\frac{n}{2}\Big)^2 \sum_{i=0}^{l-1} \frac{7^k}{4^k} \\
\overset{2}{=}& 18\Big(\frac{n}{2}\Big)^2 \frac{\left(\frac{7}{4}\right)^l - 1}{\frac{7}{4} - 1} = 18(2^l)^2 \frac{\left(\frac{7}{4}\right)^l - 1}{7 - 4} = \frac{18}{3}4^l\Big(\Big(\frac{7}{4}\Big)^l - 1\Big) = \frac{18}{3}(7^l - 4^l)
\end{aligned}$$

$\square$

Therefore, the threshold at which Strassen's method becomes optimal will be determined by the inequality:

$$T_{ord}(n) = n^3 + n^2(n-1) = 2^{3l} + 2^{2l}(2^l - 1) > 7^l + \frac{18}{3}(7^l - 4^l) = T(n) \tag{23}$$

Given its complexity, I wrote a C program that checks, starting from $l = 1$, if the integer satisfies the inequality. The first integer at which the ordinary method becomes more computationally complex is $l = 10$.

[2] Geometric progression

Now that we have found the threshold for Strassen's algorithm, we can focus on the global case of general matrices of sizes $m \times n$ and $n \times p$, which we have introduced in (3.2.2) and deduced its complexity recurrence in the equation (21). In the following proposition, we solve it:

Proposition 3 (Threshold in General algorithm)**.** *The solution to the recurrence (21) is*

$$T(p) = k^l + a(rs)^{l-1}\frac{(k/rs)^l - 1}{(k/rs) - 1} + b(st)^{l-1}\frac{(k/st)^l - 1}{(k/st) - 1} + c(rt)^{l-1}\frac{(k/rt)^l - 1}{(k/rt) - 1}$$

Proof. To solve the recurrence

$$T(m,n,p) = kT\Big(\frac{m}{r}, \frac{n}{s}, \frac{p}{t}\Big) + a\frac{m}{r}\frac{n}{s} + b\frac{n}{s}\frac{p}{t} + c\frac{m}{r}\frac{p}{t}$$

We consider again $T(m,n,p) = P(m,n,p) + S(m,n,p)$, where $P(m,n,p)$ are the multiplications needed in the algorithm and $S(m,n,p)$ the additions.

For the multiplications, it is clear that

$$P(m,n,p) = P(p) = kP(\frac{p}{t}) = k^l$$

For the additions, we need to solve:

$$S(m,n,p) = a\Big(\frac{m}{r}\frac{n}{s}\Big) + b\Big(\frac{n}{s}\frac{p}{t}\Big) + c\Big(\frac{m}{r}\frac{p}{t}\Big) + kS\Big(\frac{m}{r}, \frac{n}{s}, \frac{p}{t}\Big)$$

Given that $p = t^l$, we generalize the main variable to p, and we link m and n to this main variable by using the relation $log_t p = l$:

$$\begin{aligned} m &= r^{\log_t p} \\ n &= s^{\log_t p} \end{aligned}$$

Therefore, the recurrence becomes the following:

$$S(p) = a\Big(\frac{r^{\log_t p}}{r}\frac{s^{\log_t p}}{s}\Big) + b\Big(\frac{s^{\log_t p}}{s}\frac{p}{t}\Big) + c\Big(\frac{r^{\log_t p}}{r}\frac{p}{t}\Big) + kS\Big(\frac{p}{t}\Big)$$

And by simplifying the expressions and expanding the recurrence, we follow these calculations:

$$\begin{aligned} S(p) &= a(rs)^{\log_t p-1} + bs^{\log_t p-1}\frac{p}{t} + cr^{\log_t p-1}\frac{p}{t} + kS\Big(\frac{p}{t}\Big) \\ &= a(rs)^{\log_t p-1} + bs^{\log_t p-1}\frac{p}{t} + cr^{\log_t p-1}\frac{p}{t} + k\left(a(rs)^{\log_t \frac{p}{t}-1} + bs^{\log\frac{p}{t}-1}\frac{p}{t^2} + ct^{\log\frac{p}{t}-1}\frac{p}{t^2} + kS\Big(\frac{p}{t^2}\Big)\right) \\ {}^{3} &= a(rs)^{\log_t p-1} + ka(rs)^{\log_t p-2} + k^2a(rs)^{\log_t p-3} + ... + k^{l-1}a(rs)^{\log_t p-l} \\ &\quad + bs^{\log_t p-1}\frac{p}{t} + kbs^{\log_t p-2}\frac{p}{t^2} + k^2bs^{\log_t p-3}\frac{p}{t^3} + ... + k^{l-1}bs^{\log_t p-l}\frac{p}{t^l} \\ &\quad + cr^{\log_t p-1}\frac{p}{t} + kcr^{\log_t p-2}\frac{p}{t^2} + k^2cr^{\log_t p-3}\frac{p}{t^3} + ... + k^{l-1}cr^{\log_t p-l}\frac{p}{t^l} + k^lS(1) \end{aligned}$$

[3] $log_t \frac{p}{t} - 1 = \log_t - \log_t t - 1 = \log_t p - 1 - 1 = \log_t p - 2$

$$\overset{4}{=} a(rs)^{\log_t p-1}\left(1+\frac{k}{rs}+\left(\frac{k}{rs}\right)^2+...+\left(\frac{k}{rs}\right)^{l-1}\right)$$
$$+ bs^{\log_t p-1}\frac{p}{t}\left(1+\frac{k}{st}+\left(\frac{k}{st}\right)^2+...+\left(\frac{k}{st}\right)^{l-1}\right)$$
$$+ cr^{\log_t p-1}\frac{p}{t}\left(1+\frac{k}{rt}+\left(\frac{k}{rt}\right)^2+...+\left(\frac{k}{rt}\right)^{l-1}\right)$$
$$= a(rs)^{l-1}\sum_{i=0}^{l-1}\left(\frac{k}{rs}\right)^i + b(st)^{l-1}\sum_{i=0}^{l-1}\left(\frac{k}{st}\right)^i + c(rt)^{l-1}\sum_{i=0}^{l-1}\left(\frac{k}{rt}\right)^i$$
$$\overset{5}{=} a(rs)^{l-1}\frac{(k/rs)^l-1}{(k/rs)-1} + b(st)^{l-1}\frac{(k/st)^l-1}{(k/st)-1} + c(rt)^{l-1}\frac{(k/rt)^l-1}{(k/rt)-1}$$

□

Thus, taking into consideration the ordinary method complexity seen in (22), to find the efficiency threshold between methods we need to solve the inequality:

$$T_{ord}(p) = r^l s^l t^l + r^l t^l (s^l - 1) > k^l + a(rs)^{l-1}\frac{(k/rs)^l-1}{(k/rs)-1} + b(st)^{l-1}\frac{(k/st)^l-1}{(k/st)-1} + c(rt)^{l-1}\frac{(k/rt)^l-1}{(k/rt)-1} = T(p) \quad (24)$$

That finds the l by which the ordinary method has more computing complexity than the tensor method.

[4] $S(1) = 0$, because additions are not needed for a simple number multiplication
[5] Formula for finite geometric progression

3.3.2 Optimal efficiency threshold combining Ordinary and Tensor Methods

Until now, we have totally separated the ordinary method from the tensor method, and the threshold we have obtained is the l value for which the extra complexity in the tensor method is worth it compared to the number of multiplications the ordinary method needs.

In this section, we consider and an afterthought: The tensor method reduces the size of the matrix until the simple multiplication of numeric values, meaning that the tensor method also needs to compute smaller matrix multiplications, where it is not anymore the optimal method.

To deal with this inefficiency, what we can do is study the limit iteration inside the tensor method, that is, find the internal threshold between the tensor method and the ordinary method.

We have seen in section (3.2.2) that in a certain iteration were $p = t^l$, the complexity of the tensor method is the following:

$$T_{ten}(p) = kT_{ten}\left(\frac{p}{t}\right) + ar^{l-1}s^{l-1} + bs^{l-1}t^{l-1} + cr^{l-1}t^{l-1}$$

And we know, by using (20), the upper expression is equally to:

$$T_{ten}(t^l) = kT_{ten}(t^{l-1}) + ar^{l-1}s^{l-1} + bs^{l-1}t^{l-1} + cr^{l-1}t^{l-1}$$

Where $T_{ten}(t^{l-1})$ is the complexity of the multiplication of the next iteration of matrices with sizes $r^{l-1} \times s^{l-1}$ and $s^{l-1} \times t^{l-1}$.

If we focus on the iteration $l = L_0$ where L_0 is optimal for the tensor method, but $l = L_0 - 1$ is optimal for the ordinary method, the complexity of the iteration would be the same expression as before but taking into account that the following multiplication of the recurrence will be an ordinary one, as shown next:

$$T_{ten}(t^{L_0}) = kT_{ord}(t^{L_0-1}) + ar^{L_0-1}s^{L_0-1} + bs^{L_0-1}t^{L_0-1} + cr^{L_0-1}t^{L_0-1}$$

And considering the ordinary method complexity calculated in (22):

$$T_{ord}(p) = mnp + mp(n-1)$$

We compare the ordinary method complexity for $l = L_0$:

$$T_{ord}(t^{L_0}) = r^{L_0}s^{L_0}t^{L_0} + r^{L_0}t^{L_0}(s^{L_0} - 1) = r^{L_0}t^{L_0}(2s^{L_0} - 1)$$

To the optimal method for $l = L_0$ as well:

$$T_{op} = kr^{L_0-1}t^{L_0-1}(2s^{L_0-1} - 1) + ar^{L_0-1}s^{L_0-1} + bs^{L_0-1}t^{L_0-1} + cr^{L_0-1}t^{L_0-1}$$

Resulting in the inequality:

$$T_{op} = kr^{L_0}t^{L_0}(2s^{L_0} - 1) + ar^{L_0-1}s^{L_0-1} + bs^{L_0-1}t^{L_0-1} + cr^{L_0-1}t^{L_0-1} < r^{L_0}t^{L_0}(2s^{L_0} - 1) = T_{ord} \tag{25}$$

3.3.3 Threshold results

Due to the complexity of finding the threshold of the inequalities (24) and (25) manually, I developed a C program to compute its numeric solution, finding the immediate upper integer for which the inequality is true. In tables (1) and (2) are written, for each of the 93 given tensors, the following results:

- r, s and t of the tensor
- Its rank.
- The general threshold between methods, l, found by finding the lower l that satisfies the inequality (24).
- and the corresponding number of matrix multiplications for the ordinary method at which the tensor method becomes more effective.
- The optimal threshold, L_0, found by finding the lower L_0 that satisfies the inequality (25)
- and the corresponding number of matrix multiplications for the ordinary method at which the optimal method becomes more effective.

We can easily see that the optimal threshold outperforms the general case by far, improving it so much that the optimal threshold does not exceed 4 in any tensor, and, in most cases, is reduced to 3.

Comparing the columns of the expression $r^l s^l t^l$ between l and L_0 we notice that the improvement is done in the line of far fewer multiplications than the ordinary method, meaning that the optimal method that considers the threshold L_0 upgrades the multiplication time taken on a smaller scale than for the tensor method.

For example, in the case of the tensor $9, 9, 11$ with matrix of sizes $9^8, 9^8, 11^8$, making around $3 \cdot 10^{23}$ multiplications with the ordinary method is faster than making $576^8 \approx 1.2 \cdot 10^{22}$ multiplications with the tensor method, showing how inadequate this method is. Instead, the optimal method becomes a better option around $7 \cdot 10^8$ multiplications of the ordinary method, reducing by far the sizes of matrices to which we can apply the tensor algorithm.

r,s,t	rank	l	$r^l s^l t^l$	L_0	$r^{L_o} s^{L_o} t^{L_o}$
2,2,2	7	11	8589934592	5	32768
2,2,3	11	12	8916100448256	4	20736
2,2,4	14	11	17592186044416	4	65536
2,2,5	18	14	16384000000000000	5	3200000
2,2,6	21	9	2641807540224	4	331776
2,2,7	25	12	232218265089212420	4	614656
2,2,8	28	11	36028797018963968	4	1048576
2,3,3	15	8	11019960576	4	104976
2,3,4	20	9	2641807540224	4	331776
2,3,5	25	9	19683000000000	4	810000
2,4,4	26	8	1099511627776	4	1048576
2,4,5	33	10	10485760000000000	4	2560000
2,5,5	40	9	1953125000000000	4	6250000
3,3,3	23	10	205891132094649	4	531441
3,3,4	29	8	2821109907456	3	46656
3,3,5	36	7	373669453125	3	91125
3,4,11	103	7	698260569735168	3	2299968
3,4,4	38	7	587068342272	3	110592
3,4,5	47	7	2799360000000	3	216000
3,5,5	58	7	13348388671875	3	421875
3,5,9	105	7	817215093984375	3	2460375
3,9,11	225	9	1798075998222050300000	4	7780827681
4,4,4	49	8	281474976710656	3	262144
4,4,5	63	8	1677721600000000	3	512000
4,5,10	152	8	2560000000000000000	3	8000000
4,5,11	169	7	24943578880000000	3	10648000
4,5,5	76	7	100000000000000	3	1000000
4,5,9	139	7	6122200320000000	3	5832000
4,9,10	255	7	783641640960000000	3	46656000
4,9,11	280	7	1527095866010812400	3	62099136
4,11,11	343	7	6221821273427820500	3	113379904
4,11,12	366	7	11440301174540993000	3	147197952
5,5,5	98	7	476837158203125	3	1953125
5,5,7	134	7	5026507568359375	3	5359375
5,7,10	257	7	643392968750000000	3	42875000
5,7,11	280	9	1858431532210379700000000	3	57066625
5,7,9	234	7	307732862434921860	3	31255875
5,8,10	287	7	1638400000000000000	3	64000000
5,8,11	317	7	3192778096640000000	3	85184000
5,8,9	262	7	783641640960000000	3	46656000
5,9,10	323	7	3736694531249999900	3	91125000
5,9,11	358	7	7281760530523359200	3	121287375
5,9,12	381	7	13389252099840000000	3	157464000
5,9,9	296	7	1787249410543828200	3	66430125
6,7,10	296	7	2305393332480000000	3	74088000
6,7,11	322	8	2075562447064149800000	3	98611128

Table 1: Threshold between Ordinary and Tensor Methods

r,s,t	rank	l	$r^l s^l t^l$	L_0	$r^{L_0} s^{L_0} t^{L_0}$
6,7,9	270	7	1102662484205853300	3	54010152
6,8,10	329	6	12230590464000000	3	110592000
6,8,11	365	7	11440301174540993000	3	147197952
6,9,10	373	6	24794911296000000	3	157464000
6,9,11	411	7	26091864523169116000	3	209584584
6,9,9	342	7	6404037772671962100	3	114791256
7,7,10	350	7	6782230728490000400	3	117649000
7,7,11	384	7	13216648996753920000	3	156590819
7,7,9	318	7	3243919932521508900	3	85766121
7,8,10	393	7	17270948495360000000	3	175616000
7,8,11	432	7	33656192666127303000	3	233744896
7,8,12	462	6	92090671199944704	3	303464448
7,8,9	354	7	8260641125390352400	3	128024064
7,9,10	441	7	39389806391669998000	3	250047000
7,9,11	481	8	5319439537182768100000	3	332812557
7,9,12	510	8	10670244327163202000000	3	432081216
7,9,9	399	7	18840022288735949000	3	182284263
7,10,10	478	6	117649000000000000	3	343000000
7,10,11	536	7	160485232668530020000	3	456533000
7,11,11	582	7	312740317198643040000	3	607645423
8,8,10	441	7	43980465111040000000	3	262144000
8,8,11	489	6	121740744925904900	3	348913664
8,9,10	489	6	139314069504000000	3	373248000
8,9,11	533	8	15481087051271210000000	3	496793088
8,9,12	560	8	31053455938824548000000	3	644972544
8,10,10	532	6	262144000000000000	3	512000000
8,10,11	596	7	408675596369920000000	3	681472000
8,10,12	636	6	782757789696000000	3	884736000
8,11,11	649	7	796393122998761030000	3	907039232
8,11,12	691	7	1464358550341247000000	3	1177583616
9,9,10	534	8	18530201888518411000000	3	531441000
9,9,11	576	8	39721133415268930000000	3	707347971
9,9,9	498	9	5814973700304006400000000	3	387420489
9,10,10	606	6	531441000000000000	3	729000000
9,10,11	657	7	932065347906989980000	3	970299000
9,10,12	696	7	1713824268779520000000	3	1259712000
9,11,11	725	8	19779852014625586000000	3	1291467969
9,11,12	760	8	39676332868511877000000	3	1676676672
10,10,10	682	6	1000000000000000000	3	1000000000
10,10,11	746	7	1948717100000000100000	3	1331000000
10,10,12	798	7	3583180800000000000000	3	1728000000
10,11,11	821	7	3797498335832409900000	3	1771561000
10,11,12	874	7	6982605697351680000000	3	2299968000
10,12,12	928	7	12839184645488640000000	3	2985984000
11,11,11	896	8	98497326758076103000000	3	2357947691
11,11,12	941	8	197575427774806080000000	3	3061257408
11,12,12	990	8	396315828512542980000000	3	3974344704

Table 2: Threshold between Ordinary and Tensor Methods

4 From Theory to Practice: Implementing the Algorithm in C

To assess the accuracy of my theoretical results, I have developed a series of programs that allow us to compare the ordinary algorithm with the tensor's algorithm. This following part has a linear structure that embodies practically all the theoretical results studied in the previous sections.

4.1 Tensors reading

The 93 tensors found by AlphaTensor were presented on their GitHub [3] in a zipped archive ".npz" with 93 files that contained each a tensor written with the library NumPy from Python. On the other hand, to optimize efficiency, the programming language I used to program the matrix multiplication simulations was C.

Therefore, to be able to use these structures, I wrote a NumPy program (A) with Google Colab that reads each tensor column, transposes the third one for our convenience of notation and then converts the tensor in each file into a text file saved in a My Drive folder with the name "tensor_rxsxt.txt", where r, s and t are the sizes of the corresponding tensor. Inside the text files, there is a first line with the rank of the tensor, and in the following, each line of the tensor with its values separated by blank spaces.

4.2 Tensor algorithm

The first implementation of the tensor algorithm I wrote was a function (B) that follows the tensor algorithm recurrence from l of the initial sizes $m = r^l, n = s^l, p = t^l$ until $l = 0$, that is, when $m = n = p = 1$. The process is the following:

(a) The algorithm starts by being given as variables: the matrices A, B and their corresponding sizes m, n, p , the matrix C where we will input the result of the multiplication, the sizes of the chosen tensor r, s, t and its rank k, and lastly, the tensor used.

(b) The function starts by checking if we are in the simple final step where $m = n = p = 1$. In that case, the function will compute the number multiplication $A \cdot B$ and if not, the algorithm will start.

(c) The first step of the algorithm is to initialize the matrix of results C by filling it with zeros. After that, the function allocates the memory for the matrices $A1, B1$ and $P1$. The matrices $A1$ and $B1$ are the submatrices of sizes $\frac{m}{r} \times \frac{n}{s}$ and $\frac{n}{s} \times \frac{p}{t}$ corresponding to the division of the original matrices into $r \times s$ and $s \times t$ blocs. The matrix $P1$ is the one where the result of the multiplication of each row of the tensor is saved.

(d) Naturally, the following step in the algorithm is to enter a loop of k iterations to compute the k multiplications given by the tensor length. For each iteration of the loop, the algorithm calculates the linear combination of the matrix A and B indicated by the first and second columns of the tensor and saves them into the matrices $A1$ and $B1$ respectively.

(e) Having the two linear combinations, the algorithm calls itself recursively with the new submatrices $A1$ and $B1$ to compute the needed multiplication and save it into $P1$. The algorithm will start again with these new variables and sizes until it arrives to the final case explained in *(b)* step.

(f) The following step, still inside the loop, is to add the result saved in $P1$ into the corresponding C_{ij} position of the final matrix C. This is similar to a general process of the one explained in (12)-(18), but instead of searching for the P_i that goes in each C_{ij} position, the code iterates on P_i and adds them to the C_{ij} no null positions indicated by the third row of the tensor.

(g) Lastly, outside the loop, the memory of the three auxiliary matrices $A1, B1$ and $P1$ is freed.

4.3 Optimal tensor algorithm

After the studies explained in (3.3.2) I programmed a second improved version of the tensor algorithm (C) taking into consideration the optimal threshold shown in (1) and (2).

The difference between this new algorithm and the previous one is in steps (a) and (b). The function has an additional new variable, L_0: the optimal threshold of the corresponding tensor in the algorithm. This variable is used in the conditional at the begging of the function: instead of checking if we are on the last step of the recursion, the conditional controls if the size of the rows of the matrix A (m) is smaller or bigger than the size r^{L_0} where the tensor method starts to become the best option. If $m < r^{L_0}$, the function calls the ordinary algorithm for matrix multiplications, given that is the faster option. If not, the algorithm starts as explained in (c)-(g).

5 Simulation and results

In this chapter, we will put into practice the theory seen until now. It is important to be aware that the method to calculate the times of the algorithms is not entirely accurate, given that it uses the real execution time of the program and can be sensible to other open programs running at the same time.

Also, the theoretical results only consider the number of operations in each method, and because I programmed a general algorithm for tensors, it is not specially optimized and takes more time to perform certain processes, such as checking if each tensor coefficient is or is not null or the memory allocation in each iteration, while a personalized function for each particular tensor would be able to obviate these procedures.

5.1 Threshold in practice

In this subsection, we want to compare the theoretical results shown in tables (1) and (2) with the real computational time that the algorithm takes to solve the multiplication for a fixed l. For that, I have developed a program (D) using the tensor algorithm explained in (4.2):

(a) The program starts by asking the sizes r, s, t of the tensor and the power l we want to work with.

(b) It then reads the chosen tensor and considering that $m = r^l, n = s^l$ and $p = t^l$, reserves the needed memory for the two matrices $A_{m\times n}$ and $B_{n\times p}$, and for the result matrix $C_{m\times p}$.

(c) Afterward, the program fills the matrices A and B with positive and negative random floating point values and calls the function for the tensor multiplication.

(d) The program also computes the ordinary multiplication, and in the case of tensor 2,2,2 the multiplication with Strassen's tensor too.

(e) Lastly, the program prints the time taken by the two methods and frees up the memory allocated for the matrices.

This process is replicated four times for different random matrices to consider the average time of the process.

I executed this program for three not-too-large-to-make-calculations representative tensors (2,2,2; 2,3,4; 4,4,4) raised to a fixed power l, and I saved the time for each of the methods. Considering the four times for each method the program returns, I calculated its mean time and collected the logarithm of the final time. With this data, I was able to construct a table of information depending on the method and l, and I created regressions to estimate the real thresholds between methods. The results are the following:

5.1.1 Tensor 2,3,4

For this tensor, the results obtained are quite accurate compared to the theoretical case. The data summarized in the graphic (1) allows confirming that the slope of the trend line in the ordinary method is bigger than the one in the tensor method, meaning that the threshold between methods exists near 7.

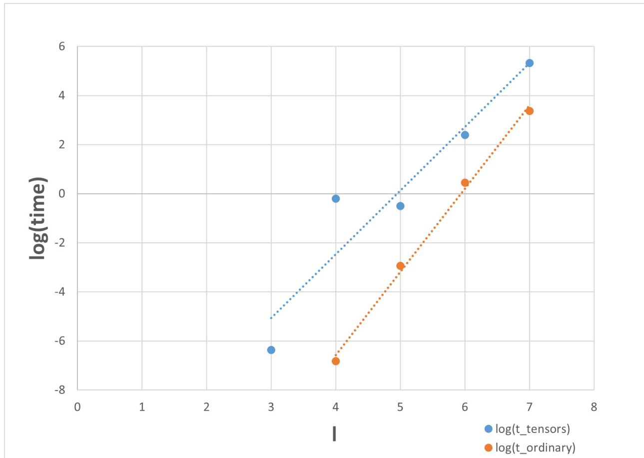


Figure 1: Observed times for tensor 2,3,4

To find this experimental threshold, we construct a linear regression by finding the slope and intersection of the lines for the two sets of data. In this case, the formulas for the linear regressions are:

$$y_{ten} = 2.597274071 \cdot l - 12.85422398$$
$$y_{ord} = 3.397486674 \cdot l - 20.17158166$$

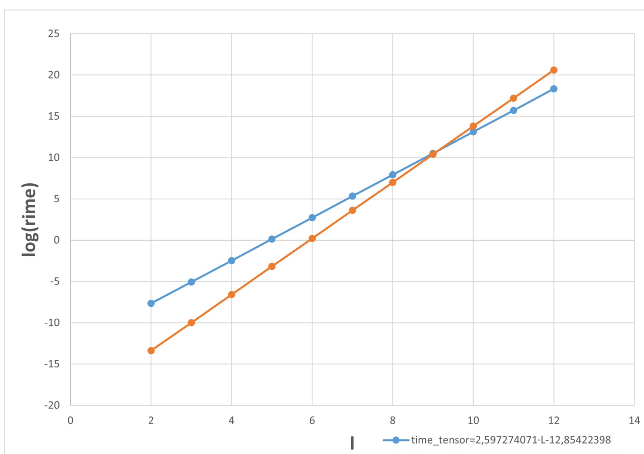
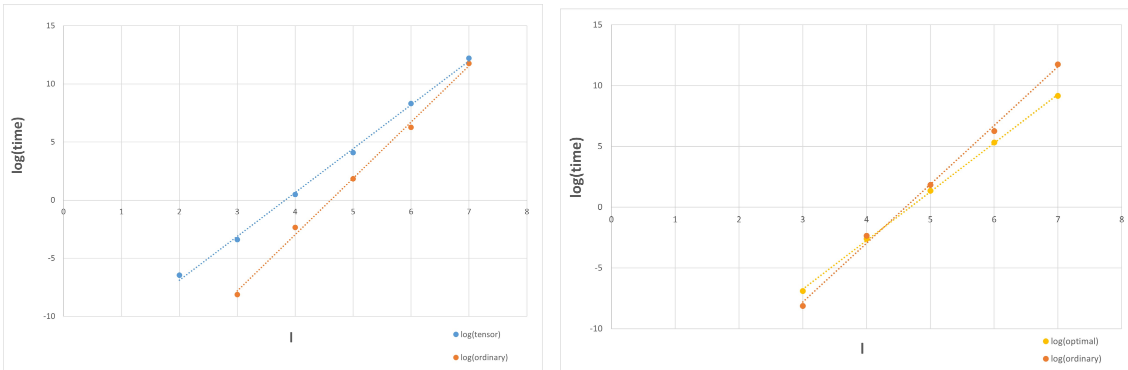


Figure 2: Linear regression for tensor 2,3,4

And their intersection happens in $l = 9.144266972$, a very close value to the theoretical threshold $l = 9$ found in (1).

5.1.2 Tensor 4,4,4

For the tensor 4,4,4, we will study the threshold between the tensor and the ordinary methods, which in this case the theory says it is $l = 8$, but we will also study the optimal algorithm threshold, theoretically $l = 4$.



(a) Methods: Ordinary and Tensor

(b) Methods: Optimal and Tensor

Figure 3: Observed times for tensor 4,4,4

With the data on the graphics (3a) and (3b) we can already corroborate that these thresholds in practice are similar to the theoretical ones, but if we write their corresponding linear regressions, we obtain the equations:

$$
\begin{aligned}
t_{ten} =& 3.77279696728277 \cdot l - 14.4381658631134 \\
t_{ord} =& 4.83812858263816 \cdot l - 22.3193159612635 \\
t_{op} =& 4.00664618275651 \cdot l - 18.7676769293097
\end{aligned}
$$

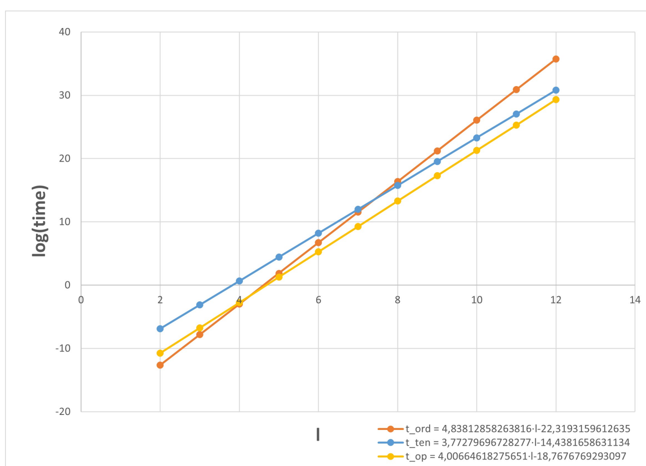


Figure 4: Linear regression for tensor 2,3,4

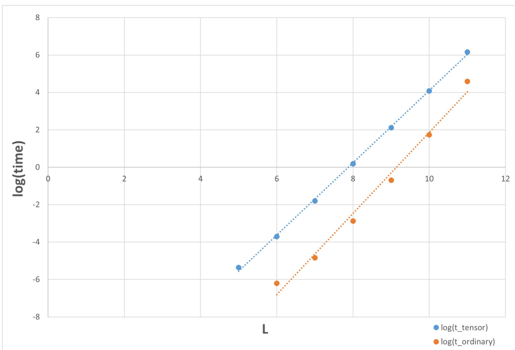
The intersection between the tensor and ordinary algorithms happens in $l = 7.397837429$ which, considering only $l \in \mathbb{N}$ as done in the theory section (3.3.3), would mean that $l = 8$. On the other hand, the intersection between the ordinary and the optimal algorithms happens in $l = 4.271454251$, a less favorable result compared to the theoretical $l = 4$.

Nevertheless, we need to keep in mind that this tensor gives rise to extremely large matrices already for small exponents l, and the data with which I worked was limited, so it makes sense that the results are less accurate.

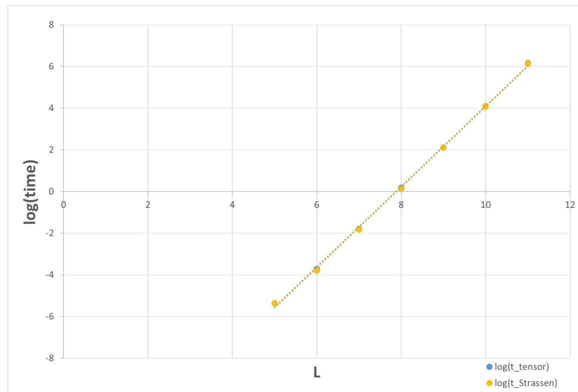
5.1.3 Tensor 2,2,2

For this tensor, we have more data and for higher l powers (from $l = 5$ to $l = 11$) thanks to the fact that the powers are lower. In the graphic (5a) we can already see that the methods are not near their intersection around $l = 11$, contrary to the theoretical result found in (1).

And in the graphic (5b) we can see that Strassen's method and the tensor 2,2,2 found by AlphaTensor behave equally, making it almost impossible to differentiate them in the graphic. This makes sense because the two algorithms have the same amount of multiplications, and what AlphaTensor managed to do was decrease the number of additions, and with small matrices like these, it's logical that the difference cannot be appreciated.



(a) Methods: Ordinary and Tensor



(b) Methods: Strassen's and Tensor

Figure 5: Observed times for tensor 2,2,2

If we create with this data a linear regression, we obtain the following results:

$$
\begin{aligned}
y_{ten} &= 1.9308 \cdot l - 15.20737752 \\
y_{ord} &= 2.1675458 \cdot l - 19.8057 \\
y_{Strass} &= 1.932407842 \cdot l - 15.24834468
\end{aligned}
$$

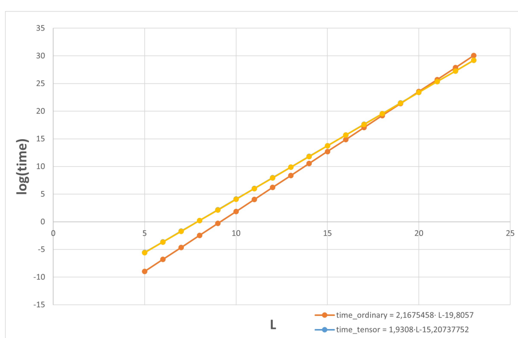


Figure 6: Second-degree polynomial regression of of tensor 2,2,2

Equalizing the tensors and the ordinary equations, these linear regressions tells us that the practical threshold between methods is $l = 19.42330424$, a surprisingly bad outcome compared to the previous tensors.

Given that in this specific tensor, $n = m = p = 2^l$, we can study the slopes of the data and compare the obtained values with the following theoretical results:

- According to the theory about the theoretical complexity studied in (3.2.1), the slope should be theoretically:
$$\log(t_{ordinary}) = \log(n^3) = 3 \cdot \log(2^l) = 3 \cdot \log(2) \cdot l \approx 2.08 \tag{26}$$
And the first degree incrementing rate found in the linear regression for the ordinary method is $\log(t_{ordinary}) = 2.167545821$, a tolerable error.

- The theoretical complexity of the method tensor we found in (3.2.2) shows us that:
$$\log(t_{tensor}) = \log(n^{\log_2 7} = \log(n^{\frac{log7}{\log 2}}) = \log\left((2^l)^{\frac{log7}{\log 2}}\right) = l\frac{log7}{\log 2}\log 2 = \log 7 \cdot l \approx 1.95 \tag{27}$$
The first degree incrementing rate in the linear regression for the tensor method is $\log(t_{tensor}) = 1.930803088$, a largely accurate approximation of the theoretical result.

Hence, the results obtained matches the theory.

This leads to think that because the amount of data we have from this tensor, maybe a linear regression is not good enough to simulate the tensors behavior, and we need a better-adjusting regression. Our first thought can be a second-degree polynomial regression, but it is not correct asymptotically, given that its slope never stops increasing, and by the results of the Master Theorem, it should be a straight line. Therefore, the correct approach would be to consider a type $ax + b + \frac{c}{x}$ hyperbole model, because this function does have an asymptote.

The equations of this new regression are:

$$\begin{aligned}
y_{ten} =& 2.14786032775748 \cdot l - 18.5953662105896 + \frac{12.3440172299102}{l} \\
y_{ord} =& 3.53136727167757 \cdot l - 42.6397942710677 + \frac{91.5758579420386}{l} \\
y_{Strass} =& 2.17980785686383 \cdot l - 19.1099456709033 + \frac{14.0696069328208}{l}
\end{aligned}$$

And we can see their behavior in the following graphic:

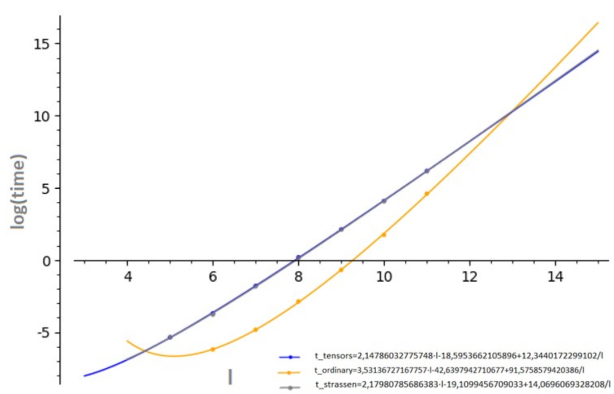


Figure 7: Hyperbole of tensor 2,2,2

The threshold between the tensor and the ordinary method obtained from these equations is $l = 12.96066912425335$, a closer value to the theoretical threshold.

But, if we study the slopes: ≈ 3.53 for the ordinary equation and ≈ 2.14 for the tensor equation, we see that they do not match the theoretical results explained in (26) and (27), and it would be necessary to deepen more into these studies to be able to give a final explanation.

However, in both linear and non-linear regressions, there is a difference between the practical and theoretical thresholds, and the main factor we can blame for this difference is memory management. In this case, the l threshold is quite large, and therefore more iterations of the algorithm method have to be made; this makes the algorithm lose time, and also the program has to allocate memory for the matrix for each needed size, which also takes extra time.

Meanwhile, a comparison we can do for this tensor is to relate it to the tensor 4,4,4. Notice that $4^l, 4^l, 4^l = 2^{2l}, 2^{2l}, 2^{2l}$, therefore, we can plot the data of the tensor $2, 2, 2$ in function of l alongside the data of the tensor $4, 4, 4$ in function of $2l$. And the obtained grafted data is the following:

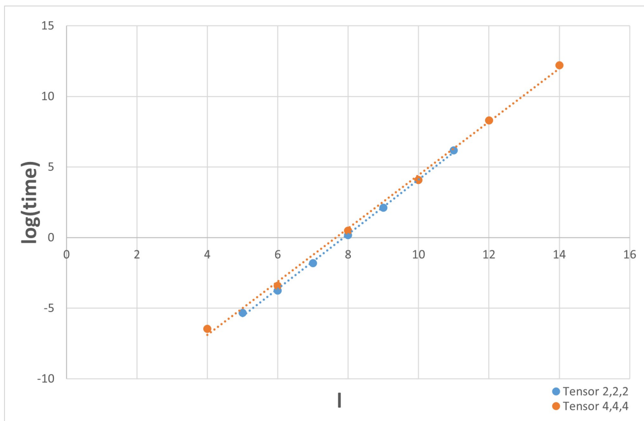


Figure 8: Comparision of tensors $2, 2, 2$ and $4, 4, 4$

In this graphic, we observe that the slopes are practically identical. In fact, if we calculate the trend line, we obtain:

$$y_{4,4,4} = 1.88639848364139 \cdot l - 14.4381658631134$$
$$y_{2,2,2} = 1.9331782860516 \cdot l - 15.2310398432455$$

A slope of $1,88639848364139$ for the tensor $4, 4, 4$ and a slope of 1.9331782860516 for the tensor $2, 2, 2$. This means that the tensor $4, 4, 4$ is not very useful given that all its multiplications can be made as fast with the tensor $2, 2, 2$. This makes sense with the essence of the tensors, because we can see in the table (1) that the tensor $2, 2, 2$ needs 7 multiplications while the tensor $4, 4, 4$ needs $7^2 = 49$ multiplications. Therefore, they are naturally direct multiples.

Anyway, even though I have not done more research on other tensors due to lack of time, it is important to note that this does not happen with all multiple tensors. For example, the tensor $3, 3, 3$ needs 23 multiplications, while the tensor $9, 9, 9$ needs 498 multiplications. Given that $498 < 23^2 = 529$, the tensor $9, 9, 9$ is a better option than the tensor $3, 3, 3$ for the multiplications where it can be used.

5.2 Matrices of Arbitrary Dimension

Applying this multiplication algorithm to arbitrary-size matrices is a more delicate issue: the size of the matrices can be unfitting for all 93 tensors, so first, we need to convert the matrices to a size that allows us to work with the given tensors. That is done by adding the necessary number of rows and columns, with the positions filled with zeros. But, which tensor size is preferable to match?

5.2.1 Suitable tensor for arbitrary dimension matrices

For this problem, I wrote a program (E) that asks for text files with the matrices A and B, and repeats the following process for each of the 93 tensors: First, it computes what would be the needed l to match the near power of the values r, s, t of the tensors. If the found l is smaller than the optimal threshold, the program will end, given that the optimal method for that matrix size would be the ordinary one. If not, the program adds the needed rows and columns of zeros to match the compatible size. After that, the program computes the multiplication of the two adapted matrices and prints the following:

- Z: The sum of the number of zeros added in the adapted matrices A and B and in the result matrix C
- l: The power of the tensor needed
- L_0: The corresponding threshold of the optimal method
- t: The time the multiplication has taken

The point of this program was to find a connection between the variables (number of zeros, tensor power, threshold) and the time required to create an easy formula to make the tensor choice. Due to the fact that this process takes a large amount of time, I have mainly focused on the multiplication of two random matrices $A_{130\times 80}$ and $B_{80\times 290}$ to analyze its behavior. The results of the observations are the following:

If we focus on the tensors that take less than one second for the multiplication, we note that there is a clear direct correlation between the number of zeros needed to add to the matrices and the computing time, as we can see in (9).

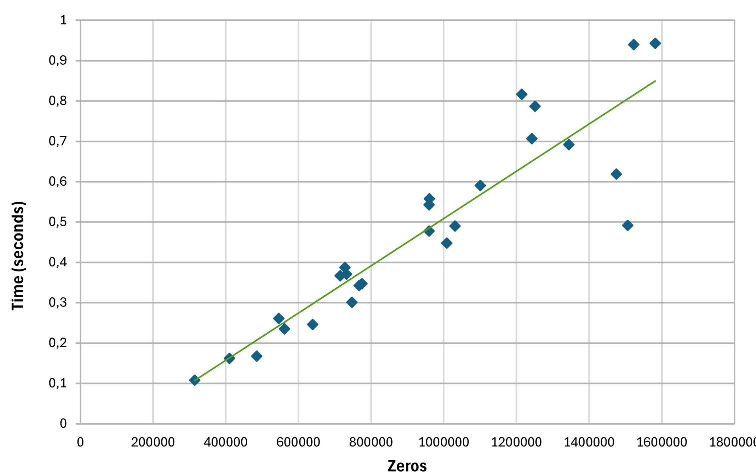


Figure 9: Relation between zeros added and computing time

On the other hand, in the cases of tensors with bigger multiplication times, this correlation isn't as clear as before and appears to have a certain amount of scattering when the numbers of zeros are above 2 million and the computing time is bigger. We can notice this in (10), the same graphic as before but with a limit of time for the multiplication of 600 seconds.

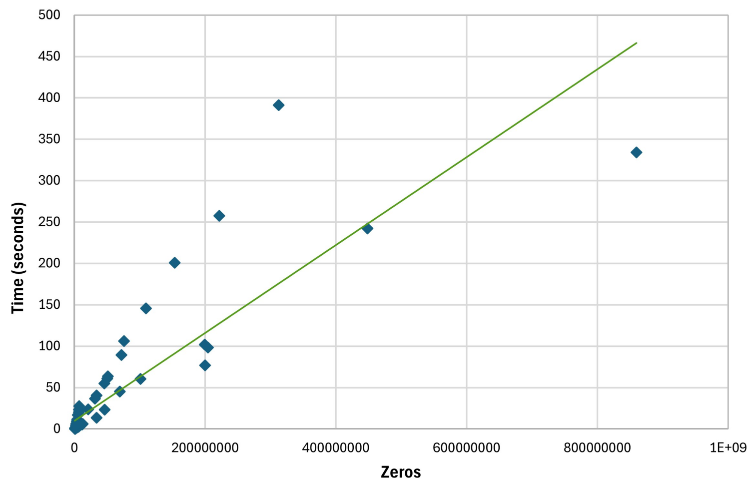


Figure 10: Relation between zeros added and computing time

This dispersion in situations with more zeros can be associated with the differences between tensors, such as how much we are improving the ordinary multiplication in each step computed by using the tensor method ($\frac{rst}{k}$) and the number of tensor algorithm iterations required ($l - L_0 + 1$). The figure (11) is an attempt to find a formula for the dispersion, taking into consideration the factors explained in the previous paragraph.

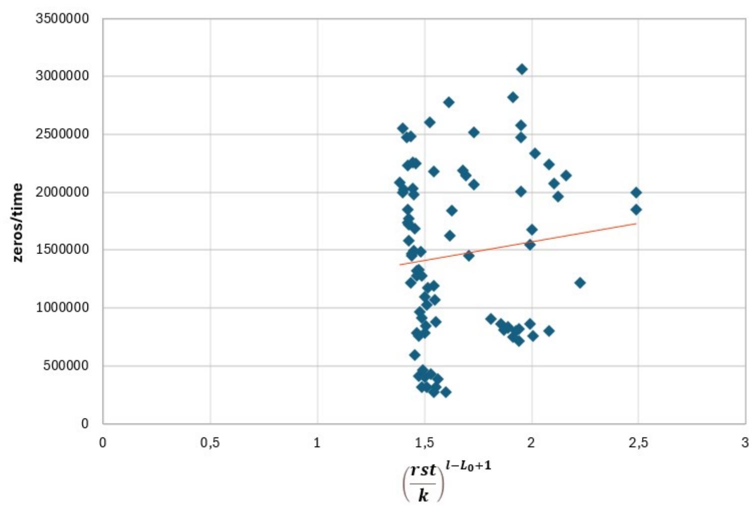


Figure 11: Dispersion

Since the cases that concern us are the faster ones, and as we recall, the correlation between time taken and the number of zeros is highly significant and greater than between any other factor, we put the focus on the smaller cases and therefore take into account only the number of zeros to add for each tensor and select the minimum.

5.2.2 Final program

Considering what we have seen in the previous sections, I developed a final program (F) with the following structure:

- Firstly, the program reads the two original matrices from a text file of choice and saves their dimensions.
- Secondly, the program calculates how many zeros are needed to add to the matrices for each tensor, as we have seen in (5.2.1). It picks the tensor that needs the least number of zeros to be added to the matrices.
- Next, the program checks that the exponent of the power of the chosen tensor l is bigger than the optimal threshold L_0. If not, the program automatically computes the ordinary multiplication with the original matrices, given that we have demonstrated that below this threshold, the faster algorithm is the ordinary.
- If $l > L_0$, the program creates matrices of the new chosen sizes, adding the needed zeros.
- Lastly, the program computes the multiplication of the two matrices using the optimal tensor method and the ordinary method and returns their corresponding times.

The results of this final program for some significant matrices are the following:

Matrices	**Tensor**	**l**	**Tensor's Time**	**Ordinary Time**
2000,3000,4000	2,2,2	12	176.712	2213.478
780,480,1740	4,4,5	5	7.46	1.964
1000,1000,1000	10,10,10	3	3.13	3.391
2000,2000,2000	2,2,2	11	23.283	73.478
2175,2175,12000	3,3,4	7	230	978
2180,2180,2180	3,3,3	7	27.941	66.371
250,6560,6560	2,3,3	8	27.558	105.219
130,80,290	4,4,5	4	0.223	0.016
510,510,19680	2,2,3	9	20.412	29.526
1024,1024,59000	2,2,3	10	236.249	375.833
6560,6560,6560	3,3,3	8	623.831	3147.037
10000,10000,10000	10,10,10	4	3669.350	11573.807

Table 3: Tensor and Ordinary Times

As we can see in the previous table (3), for most of the cases, the tensor is a clear improvement, but when the l stands too close to the threshold between methods, the times can be incorrect and very opposites. This is what happens in the red-painted cells, where the matrices have small sizes and therefore the thresholds are small too.

On the other hand, in cases where l is bigger, the difference is really notable. In the case of the multiplication of matrices of sizes $m = 2000, n = 3000, p = 4000$ and using the tensor $2, 2, 2$, the final tensor method takes only around 3 minutes, while the ordinary method takes around 36 minutes, twelve times more.

This also happens in the multiplication of matrices of sizes $m = 10, n = 10, p = 10$ where using the tensor $10, 10, 10$, the final tensor algorithm takes one hour while the ordinary method takes three long hours. This last improvement is not as good as the previous, but in here, we are already talking about hours, and the time saved is much bigger.

Nevertheless, this tensor algorithm will always be a bigger improvement for matrices with sizes near a power of a tensor size, because if the matrix is too far from any tensor, a lot of added zeros will be needed, and besides, it is very possible that there will be struggles with memory allocation for operations that are totally unnecessary.

6 Enhancements

A future work to improve the final program that I developed would be to adapt each iteration of the matrices to a different tensor. As is done now, the tensor that the program chooses is the more optimal for the sizes m, n, p, but once the algorithm selects the new size of the submatrices, we have sizes $m' = \frac{m}{r}, n' = \frac{n}{s}, p' = \frac{p}{t}$, and these numbers can be a more approximated power of another tensor; therefore, the improvement would be to consider the first better tensor to initiate the algorithm, and after each iteration, find the appropriate r, s, t for m', n', p'.

Another thing to consider in this general program is that when adding rows and columns of zeros, the program spends certain time considering submatrices full of zeros and calling each process even though none real operation will be made, because the result will be zero. A good improvement would be to be able to detect these submatrices and save the program from calling the recursion by just returning a zero.

It is important to keep in mind the main impediment that I faced executing these simulations: the memory limit of my computer. Even though my tutor has helped me with some computations on the department computer that has no memory problems, making this study on a more powerful computer at my full disposal would have allowed me to compare much bigger matrices, where the tensor algorithm is the leading one by far. But as the relations in the tensors are exponential, the sizes increase excessively fast.

Also, the simulation results are not as good as they could be because, in this work, I have considered a global algorithm for all the tensors, as I have explained in the introduction of section (5). Therefore, a good improvement would be to modify the general function for each tensor and personalize it with the literal linear combinations and the final positions in each case. This upgrade would make the algorithm work much less than in the general algorithm, and the program would be more optimal.

7 Conclusions

In this project, I have studied the first layer of tensor algorithm multiplications, and I'm delighted to have been able to create simulations that satisfy all the previous proven theory. In the beginning, I was not confident that I would match the theory with the practice, but I'm happy with the simulation results I have obtained and the justifications I have been able to deduce.

I'm aware that there is a lot to deepen into, starting with the enhancements I have suggested in the previous section. And even though if I had had more time, the program would be a better optimization, I believe the final program I have presented in this project is an initial step to produce very optimal new algorithms with these 93 tensors found by AlphaTensor.

There is a lot to contribute to matrix multiplication optimization, and this project has been my bit for the cause.

References

[1] Aja Huang Thomas Hubert Bernardino Romera-Paredes Mohammadamin Barekatain Alexander Novikov Francisco J. R. Ruiz1 Julian Schrittwieser Grzegorz Swirszcz1 David Silver Demis Hassabis Pushmeet Kohli Alhussein Fawzi, Matej Balog. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, october 2022.

[2] Bernardino Romera-Paredes Demis Hassabis Pushmeet Kohli Alhussein Fawzi, Matej Balog. Discovering novel algorithms with alphatensor, october 2022.

[3] Matej Balog. Alphatensor, october 2022.

[4] Martin Solomon. Matrix applications in data science machine learning, october 2023.

[5] V. Strassen. Gaussian elimination is not optimal. *Springer*, 13:354–356, 1969.

[6] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to algorithms.* Mit Press, 2009.

[7] wikipedia. Master theorem (analysis of algorithms), may 2024.

A Tensors reading program

```
import numpy as np
from google.colab import files

uploaded = files.upload()
filename = list(uploaded.keys())[0]
with open(filename, 'rb') as f:
  factorizations = dict(np.load(f, allow_pickle=True))

from google.colab import drive
drive.mount('/content/drive')

# Print available factorizations and their shapes.
for key in factorizations:
  u, v, w = factorizations[key]
  rank = u.shape[-1]
  assert rank == v.shape[-1] and rank == w.shape[-1]

  r, s, t = map(int, key.split(','))
  Trst = factorizations[key]
  file = f'/content/drive/My Drive/tensors/tensor_{r}x{s}x{t}.txt'
  with open(file, 'w') as f:
    print(rank, end='\n', file=f)
    for k in range(rank):
      A = [Trst[0][i,k] for i in range(r*s)]
      B = [Trst[1][i,k] for i in range(s*t)]
      C = [Trst[2][i,k] for i in range(r*t)]
      # Transposar C:
      CTransp = [C[n+m*r] for n in range(r) for m in range(t)]
      print(*A,sep=' ', end=' ', file=f)
      print(*B,sep=' ', end=' ', file=f)
      print(*CTransp,sep=' ', end='\n', file=f)
```

B Tensor algorithm

```
                    //m: rows A, n: columns A i rows B, p: columns B
void algorithm_rxsxt(int m, int n, int p, double **A, double **B, double **C,int r, int s, int t, int K, double *P[K][3]){
    if(m==1 && m==n && n==p)
        C[0][0]= A[0][0]*B[0][0];

    else{
        matrix_of_zeros(m,p,C);
        double **A1;
        double **B1;
        double **P1;

        allocate_matrix_memory(m/r,n/s,&A1);  //m=r*m1, n=s*n1:  size A1: m1 x n1
        allocate_matrix_memory(n/s,p/t,&B1);  //n=s*n1, p=t*p1   size B1: n1 x p1
        allocate_matrix_memory(m/r,p/t,&P1);  //m=r*m1, p=t*p1   size A1: m1 x p1
        for(int i=0;i<K;i++){
            combination_submatrices(m,n,A,r,s,A1,P[i][0]); //A1: lineal combination of A
            combination_submatrices(n,p,B,s,t,B1,P[i][1]);  //B1: lineal combination of B
            algorithm_rxsxt(m/r,n/s,p/t,A1,B1,P1,r,s,t,K,P); //multiplication: A1*B1
            recombination_submatrice(m,p,C,r,t,P1,P[i][2]); //Add P1 in the corresponding C position
        }
        free_matrix(m/r,A1);
        free_matrix(n/s,B1);
        free_matrix(m/r,P1);
    }
}

void combination_submatrices(int m, int n, double **A, int f, int c, double **A1, double *coef){
    int m1=m/f;
    int n1=n/c;
    matrix_of_zeros(m1,n1,A1);
    for(int i=0;i<f;i++){   //matrix rows
        for(int j=0;j<c;j++){  //matrix columns
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //eahc matrix rows
                    for(int l=0;l<n1;l++){   //each matrix columns
                        A1[k][l]=A1[k][l]+A[i*m1+k][j*n1+l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}
                                                        //rows columns
void recombination_submatrice(int m, int n, double **C, int f, int c, double **A1, double *coef){
    int m1=m/f;
    int n1=n/c;
    for(int i=0;i<f;i++){   //rows of matrices
        for(int j=0;j<c;j++){  //columns of matrices
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //rows of eaxh matrix
                    for(int l=0;l<n1;l++){   //columns of each matrix
                        C[i*m1+k][j*n1+l]+=A1[k][l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}

void matrix_of_zeros(int m, int n, double **M){
```

```
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=0;
        }
    }
}

void random_matrix(int m, int n, double **M){
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=(double)rand()+(double)rand()/RAND_MAX;
            if(rand()%2==1){
                M[i][j]=-M[i][j];
            }
        }
    }
}

void allocate_matrix_memory(int m, int n, double ***M){
    (*M) = (double **)malloc(m*sizeof(double *));
    for(int i=0;i<m;i++){
        (*M)[i]=(double *)malloc(n*sizeof(double));
    }

    if ((*M) == NULL) {
        printf("Memory allocation failed\n");
    }
}

void free_matrix(int m, double **M) {
    for (int i = 0; i < m; i++) {
        free(M[i]);
    }
    free(M);
}
```

C Optimal algorithm

```
                    //m: rows A, n: columns A i rows B, p: columns B
void algorithm_rxsxt_optim(int m, int n, int p, double **A, double **B, double **C,int r, int s
    , int t, int K, double *P[K][3],int L0){
    if(m<=pow(r,L0)){
        ordinary_product(m,n,p,A,B,C);
    }
    else{
        matrix_of_zeros(m,p,C);
        double **A1;
        double **B1;
        double **P1;

        allocate_matrix_memory(m/r,n/s,&A1);  //m=r*m1, n=s*n1:  size A1: m1 x n1
        allocate_matrix_memory(n/s,p/t,&B1);  //n=s*n1, p=t*p1   size B1: n1 x p1
        allocate_matrix_memory(m/r,p/t,&P1);  //m=r*m1, p=t*p1   size A1: m1 x p1
        for(int i=0;i<K;i++){
            combination_submatrices(m,n,A,r,s,A1,P[i][0]); //A1: lineal combination of A
            combination_submatrices(n,p,B,s,t,B1,P[i][1]);  //B1: lineal combination of B
            algorithm_rxsxt(m/r,n/s,p/t,A1,B1,P1,r,s,t,K,P); //multiplication: A1*B1
            recombination_submatrice(m,p,C,r,t,P1,P[i][2]); //Add P1 in the corresponding C
                position
        }
        free_matrix(m/r,A1);
        free_matrix(n/s,B1);
        free_matrix(m/r,P1);
    }
}

void combination_submatrices(int m, int n, double **A, int f, int c, double **A1, double *coef)
    {
    int m1=m/f;
    int n1=n/c;
    matrix_of_zeros(m1,n1,A1);
    for(int i=0;i<f;i++){   //matrix rows
        for(int j=0;j<c;j++){  //matrix columns
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //eahc matrix rows
                    for(int l=0;l<n1;l++){   //each matrix columns
                        A1[k][l]=A1[k][l]+A[i*m1+k][j*n1+l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}
                                                       //rows columns
void recombination_submatrice(int m, int n, double **C, int f, int c, double **A1, double *coef
    ){
    int m1=m/f;
    int n1=n/c;
    for(int i=0;i<f;i++){   //rows of matrices
        for(int j=0;j<c;j++){  //columns of matrices
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //rows of eaxh matrix
                    for(int l=0;l<n1;l++){   //columns of each matrix
                        C[i*m1+k][j*n1+l]+=A1[k][l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}

void matrix_of_zeros(int m, int n, double **M){
```

```c
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=0;
        }
    }
}

void random_matrix(int m, int n, double **M){
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=(double)rand()+(double)rand()/RAND_MAX;
            if(rand()%2==1){
                M[i][j]=-M[i][j];
            }
        }
    }
}

void allocate_matrix_memory(int m, int n, double ***M){
    (*M) = (double **)malloc(m*sizeof(double *));
    for(int i=0;i<m;i++){
        (*M)[i]=(double *)malloc(n*sizeof(double));
    }

    if ((*M) == NULL) {
        printf("Memory allocation failed\n");
    }
}

void ordinary_product(int m, int n, int p, double **A, double **B, double **C){
    matriu_zeros(m,p,C);
    for(int m_ite=0;m_ite<m;m_ite++){
        for(int p_ite=0;p_ite<p;p_ite++){
            for(int j=0;j<n;j++){
                C[m_ite][p_ite] = C[m_ite][p_ite] + A[m_ite][j]*B[j][p_ite];
            }
        }
    }
}

void free_matrix(int m, double **M) {
    for (int i = 0; i < m; i++) {
        free(M[i]);
    }
    free(M);
}
```

D Tensor - ordinary times program

```c
#include<stdio.h>
#include<math.h>
#include <stdlib.h>
#include <time.h>
#include <locale.h>

void matrix_of_zeros(int m, int n, double **M);
void random_matrix(int m, int n,double **M);
void allocate_matrix_memory(int m, int n, double ***M);
void ordinary_product(int m, int n, int p, double **A, double **B, double **C);
void combination_submatrices(int m, int n, double **A, int f, int c, double **A1, double *coef);
void recombination_submatrice(int m, int n, double **C, int f, int c, double **A1, double *coef);
void algorithm_rxsxt(int m, int n, int p, double **A, double **B, double **C,int r, int s, int t, int K, double *P[K][3]);
void free_matrix(int m, double **M);

///TENSOR STRASSEN
double S00[] = {1,0,0,0};
double S01[] = {0,1,0,-1};
double S02[] = {0,1,0,1};

double S10[] = {1,1,0,0};
double S11[] = {0,0,0,1};
double S12[] = {-1,1,0,0};

double S20[] = {0,0,1,1};
double S21[] = {1,0,0,0};
double S22[] = {0,0,1,-1};

double S30[] = {0,0,0,1};
double S31[] = {-1,0,1,0};
double S32[] = {1,0,1,0};

double S40[] = {1,0,0,1};
double S41[] = {1,0,0,1};
double S42[] = {1,0,0,1};

double S50[] = {0,1,0,-1};
double S51[] = {0,0,1,1};
double S52[] = {1,0,0,0};

double S60[] = {1,0,-1,0};
double S61[] = {1,1,0,0};
double S62[] = {0,0,0,-1};

double *S[7][3] = {{S00,S01,S02},{S10,S11,S12},{S20,S21,S22},{S30,S31,S32},{S40,S41,S42},{S50,S51,S52},{S60,S61,S62}};


int min_int;
int main(){
    srand (time(NULL));

    ///ASK SIZE
    char tensor_file[20];
    printf("Dimensions r s t: ");
    int r, s, t, K;
    scanf("%d %d %d", &r, &s, &t);
    sprintf(tensor_file, "tensor_%dx%dx%d.txt", r, s, t);
    ///READ TENSOR

    FILE *tensor;
    tensor = fopen(tensor_file, "r");
```

```
    if(tensor == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    fscanf(tensor,"%d",&K);

    double *P[K][3];

    for(int i=0;i<K;i++){
        P[i][0]=(double*)malloc((r*s)*sizeof(double));
        P[i][1]=(double*)malloc((s*t)*sizeof(double));
        P[i][2]=(double*)malloc((r*t)*sizeof(double));
        if(P[i][0] == NULL || P[i][1] == NULL || P[i][2] == NULL){
            printf("Memory allocation failed\n");
            return 1;
        }
    }

    for (int i=0;i<K;i++) {
        for (int k=0;k<r*s;k++)
            fscanf(tensor, "%lf", &P[i][0][k]);
        for(int k=0;k<s*t;k++)
            fscanf(tensor, "%lf", &P[i][1][k]);
        for (int k=0;k<r*t;k++)
            fscanf(tensor, "%lf", &P[i][2][k]);
    }
    fclose(tensor);

    printf("TENSOR %dx%dx%d\n",r,s,t);
    int l;
    ///FOR EACH L
    for(l=2;l<11;l++){
        int m=pow(r,l);
        int n=pow(s,l);
        int p=pow(t,l);
        struct timeval start, stop;
        ///THE PROCEDURE IS REPEATED 4 TIMES
        for(int i=0;i<4;i++){
            ///ALLOCATE THE MEMORY
            double **A;
            allocate_matrix_memory(m,n,&A);
            random_matrix(m,n,A);

            double **B;
            allocate_matrix_memory(n,p,&B);
            random_matrix(n,p,B);
            double**C;
            allocate_matrix_memory(m,p,&C);

            ///TENSOR ALGORITHM
            mingw_gettimeofday(&start, NULL);
            algorithm_rxsxt(m,n,p,A,B,C,r,s,t,K,P);
            mingw_gettimeofday(&stop, NULL);
            printf("Tensor: %6.8lf\n",(stop.tv_sec - start.tv_sec) + (stop.tv_usec - start.
                tv_usec) / 1000000.0);

            ///ORDINARY ALGORITHM
            mingw_gettimeofday(&start, NULL);
            ordinary_product(m,n,p,A,B,C);
            mingw_gettimeofday(&stop, NULL);
            printf("Ordinary: %6.8lf\n",(stop.tv_sec - start.tv_sec) + (stop.tv_usec - start.
                tv_usec) / 1000000.0);

            if(r==s && s==t && r==2){ //Fem el tensor de Strassen tambe!
                mingw_gettimeofday(&start, NULL);
                algorithm_rxsxt(m,n,p,A,B,C,r,s,t,7,S);
                mingw_gettimeofday(&stop, NULL);
```

```c
                printf("Strassen: %6.8lf\n",(stop.tv_sec - start.tv_sec) + (stop.tv_usec -
                    start.tv_usec) / 1000000.0);

            }
            printf("\n");
            ///FREE THE MATRICES
            free_matrix(m,A);
            free_matrix(n,B);
            free_matrix(m,C);

        }
    }

    return 0;
}
                                                        //rows columns
void combination_submatrices(int m, int n, double **A, int f, int c, double **A1, double *coef)
    {
    int m1=m/f;
    int n1=n/c;
    matrix_of_zeros(m1,n1,A1);
    for(int i=0;i<f;i++){   //matrix rows
        for(int j=0;j<c;j++){  //matrix columns
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //eahc matrix rows
                    for(int l=0;l<n1;l++){   //each matrix columns
                        A1[k][l]=A1[k][l]+A[i*m1+k][j*n1+l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}
                                                         //rows columns
void recombination_submatrice(int m, int n, double **C, int f, int c, double **A1, double *coef
    ){
    int m1=m/f;
    int n1=n/c;
    for(int i=0;i<f;i++){   //rows de matrices
        for(int j=0;j<c;j++){  //columns de matrices
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //rows de cada matriu
                    for(int l=0;l<n1;l++){   //columns de cada matriu
                        C[i*m1+k][j*n1+l]+=A1[k][l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}

//m: rows A, n: columns A i rows B, p: columns B
void algorithm_rxsxt(int m, int n, int p, double **A, double **B, double **C,int r, int s, int
    t, int K, double *P[K][3]){
    if(m==1 && m==n && n==p)
        C[0][0]= A[0][0]*B[0][0];
    else{
        matrix_of_zeros(m,p,C);
        double **A1;
        double **B1;
        double **P1;

        allocate_matrix_memory(m/r,n/s,&A1);  //m=r*m1, n=s*n1:  sizes A1: m1 x n1
        allocate_matrix_memory(n/s,p/t,&B1);  //n=s*n1, p=t*p1   sizes B1: n1 x p1
        allocate_matrix_memory(m/r,p/t,&P1);  //m=r*m1, p=t*p1   sizes A1: m1 x p1
        for(int i=0;i<K;i++){
            combination_submatrices(m,n,A,r,s,A1,P[i][0]); //lineal combination of A
            combination_submatrices(n,p,B,s,t,B1,P[i][1]);  //lineal combination of B
```

```
            algorithm_rxsxt(m/r,n/s,p/t,A1,B1,P1,r,s,t,K,P);   //multiplication: A*B
            recombination_submatrice(m,p,C,r,t,P1,P[i][2]);  //Add P_i in the correspondent C position
        }
        free_matrix(m/r,A1);
        free_matrix(n/s,B1);
        free_matrix(m/r,P1);
    }
}

void matrix_of_zeros(int m, int n, double **M){
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=0;
        }
    }
}

void random_matrix(int m, int n, double **M){
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=(double)rand()+(double)rand()/RAND_MAX;
            if(rand()%2==1){
                M[i][j]=-M[i][j];
            }
        }
    }
}

void allocate_matrix_memory(int m, int n, double ***M){
    (*M) = (double **)malloc(m*sizeof(double *));
    for(int i=0;i<m;i++){
        (*M)[i]=(double *)malloc(n*sizeof(double));
    }

    if ((*M) == NULL) {
        printf("Memory allocation failed\n");
    }
}
void free_matrix(int m, double **M) {
    for (int i = 0; i < m; i++) {
        free(M[i]);
    }
    free(M);
}

void ordinary_product(int m, int n, int p, double **A, double **B, double **C){
    matrix_of_zeros(m,p,C);
    for(int m_ite=0;m_ite<m;m_ite++){
        for(int p_ite=0;p_ite<p;p_ite++){
            for(int j=0;j<n;j++){
                C[m_ite][p_ite] = C[m_ite][p_ite] + A[m_ite][j]*B[j][p_ite];
            }
        }
    }
}
```

E Added zeros program

```c
#incluof<math.h>
#incluof <stdlib.h>
#incluof <stdbool.h>
#incluof <time.h>
int calculate_l(int m_ini,int n_ini,int p_ini,int Ti[3]);
int max3(int l1,int l2,int l3);
int find_l(double arr);
void fill_matrix(int m,int n, double **A,int m_ini,int n_ini,double **A_ini);

void matrix_of_zeros(int m, int n, double **M);
void allocate_matrix_memory(int m, int n, double ***M);
void ordinary_product(int m, int n, int p, double **A, double **B, double **C);
void combination_submatrices(int m, int n, double **A, int f, int c, double **A1, double *coef);
void recombination_submatrix(int m, int n, double **C, int f, int c, double **A1, double *coef);
void algorithm_rxsxt(int m, int n, int p, double **A, double **B, double **C,int r, int s, int t, int K, double *P[K][3],int L0);
void free_matrix(int m, double **M);

int additions_tensor_rxs(int r, int s, int K, double *P[K][3]);
int additions_tensor_sxt(int s, int t, int K, double *P[K][3]);
int additions_tensor_rxt(int r, int t, int K, double *P[K][3]);

double f_ord(int r, int s, int t, double l);
double f_op(int r, int s, int t, double l, int K, int a, int b, int c);

int min_integer_optimal(int r, int s, int t, int K, int a, int b, int c, double *P[K][3]);


void optimal_threshold(char *tensor_names[93],int L0_op[93]);

char *tensor_names[93] = {"tensor_2x2x2.txt",
                            "tensor_2x2x3.txt",
                            "tensor_2x2x4.txt",
                            "tensor_2x2x5.txt",
                            "tensor_2x2x6.txt",
                            "tensor_2x2x7.txt",
                            "tensor_2x2x8.txt",
                            "tensor_2x3x3.txt",
                            "tensor_2x3x4.txt",
                            "tensor_2x3x5.txt",
                            "tensor_2x4x4.txt",
                            "tensor_2x4x5.txt",
                            "tensor_2x5x5.txt",
                            "tensor_3x3x3.txt",
                            "tensor_3x3x4.txt",
                            "tensor_3x3x5.txt",
                            "tensor_3x4x11.txt",
                            "tensor_3x4x4.txt",
                            "tensor_3x4x5.txt",
                            "tensor_3x5x5.txt",
                            "tensor_3x5x9.txt",
                            "tensor_3x9x11.txt",
                            "tensor_4x4x4.txt",
                            "tensor_4x4x5.txt",
                            "tensor_4x5x10.txt",
                            "tensor_4x5x11.txt",
                            "tensor_4x5x5.txt",
                            "tensor_4x5x9.txt",
                            "tensor_4x9x10.txt",
                            "tensor_4x9x11.txt",
                            "tensor_4x11x11.txt",
                            "tensor_4x11x12.txt",
                            "tensor_5x5x5.txt",
```

```c
                                "tensor_5x5x7.txt",
                                "tensor_5x7x10.txt",
                                "tensor_5x7x11.txt",
                                "tensor_5x7x9.txt",
                                "tensor_5x8x10.txt",
                                "tensor_5x8x11.txt",
                                "tensor_5x8x9.txt",
                                "tensor_5x9x10.txt",
                                "tensor_5x9x11.txt",
                                "tensor_5x9x12.txt",
                                "tensor_5x9x9.txt",
                                "tensor_6x7x10.txt",
                                "tensor_6x7x11.txt",
                                "tensor_6x7x9.txt",
                                "tensor_6x8x10.txt",
                                "tensor_6x8x11.txt",
                                "tensor_6x9x10.txt",
                                "tensor_6x9x11.txt",
                                "tensor_6x9x9.txt",
                                "tensor_7x7x10.txt",
                                "tensor_7x7x11.txt",
                                "tensor_7x7x9.txt",
                                "tensor_7x8x10.txt",
                                "tensor_7x8x11.txt",
                                "tensor_7x8x12.txt",
                                "tensor_7x8x9.txt",
                                "tensor_7x9x10.txt",
                                "tensor_7x9x11.txt",
                                "tensor_7x9x12.txt",
                                "tensor_7x9x9.txt",
                                "tensor_7x10x10.txt",
                                "tensor_7x10x11.txt",
                                "tensor_7x11x11.txt",
                                "tensor_8x8x10.txt",
                                "tensor_8x8x11.txt",
                                "tensor_8x9x10.txt",
                                "tensor_8x9x11.txt",
                                "tensor_8x9x12.txt",
                                "tensor_8x10x10.txt",
                                "tensor_8x10x11.txt",
                                "tensor_8x10x12.txt",
                                "tensor_8x11x11.txt",
                                "tensor_8x11x12.txt",
                                "tensor_9x9x10.txt",
                                "tensor_9x9x11.txt",
                                "tensor_9x9x9.txt",
                                "tensor_9x10x10.txt",
                                "tensor_9x10x11.txt",
                                "tensor_9x10x12.txt",
                                "tensor_9x11x11.txt",
                                "tensor_9x11x12.txt",
                                "tensor_10x10x10.txt",
                                "tensor_10x10x11.txt",
                                "tensor_10x10x12.txt",
                                "tensor_10x11x11.txt",
                                "tensor_10x11x12.txt",
                                "tensor_10x12x12.txt",
                                "tensor_11x11x11.txt",
                                "tensor_11x11x12.txt",
                                "tensor_11x12x12.txt"};


int main(){

    ///READING OF A MATRIX
    FILE *A_txt;

    char A_file[20];
```

```c
    printf("Matrix A: ");
    scanf("%s",A_file);
    A_txt = fopen(A_file, "r");

    if(A_txt == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    int m_ini=0;
    int n_ini=0;
    char c;
    while((c = fgetc(A_txt)) != EOF) {
        if(c == '\n')
            m_ini++; //each \n
    }
    rewind(A_txt);
    while((c = fgetc(A_txt)) != '\n'){
        if(c == ' ') // Counts the black spaces
            n_ini++;
    }
    rewind(A_txt);

    double **A_ini;
    allocate_matrix_memory(m_ini,n_ini,&A_ini);

    for (int i=0;i<m_ini;i++) {
        for (int j=0;j<n_ini;j++)
            fscanf(A_txt, "%lf", &A_ini[i][j]);
    }
    fclose(A_txt);

    ///READING OF B MATRIX
    FILE *B_txt;

    char B_file[20];

    printf("Matrix B: ");
    scanf("%s",B_file);
    B_txt = fopen(B_file, "r");

    if(B_txt == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    int n_ini_B=0;
    int p_ini=0;
    while((c = fgetc(B_txt)) != EOF) {
        if(c == '\n')
            n_ini_B++; // Each \n
    }
    if(n_ini != n_ini_B){
        printf("The number of columns of A does not match the number of rows if B\n");
        return 1;
    }
    rewind(B_txt);
    while((c = fgetc(B_txt)) != '\n'){
        if(c == ' ') //Conuts the blank spaces
            p_ini++;
    }
    rewind(B_txt);

    double **B_ini;
    allocate_matrix_memory(n_ini,p_ini,&B_ini);

    for (int i=0;i<n_ini;i++) {
        for (int j=0;j<p_ini;j++)
            fscanf(B_txt, "%lf", &B_ini[i][j]);
```

```
    }
    fclose(B_txt);

    ///OPTIMAL THRESHOLD OF EACH TENSOR
     int L0_op[93];
    optimal_threshold(tensor_names,L0_op);

    int rst[4];
    int T[93][3];
    for(int i=0;i<93;i++)
        sscanf(tensor_names[i],"tensor_%dx%dx%d.txt", &T[i][0], &T[i][1], &T[i][2]);

    ///ZEROS OF EACH TENSOR
    int Z[93];
    int l[93];
    int z1,z2,z3;
    clock_t start, stop;

    for(int i=0;i<93;i++){
        ///CALCULATION OF l
        l[i]=calculate_l(m_ini,n_ini,p_ini,T[i]);
        int m,n,p,r,s,t;
        //T[i][1,2,3] are the corresponding r s i t fixated of the 93 tensors
        r=T[i][0];
        s=T[i][1];
        t=T[i][2];

        m=pow(r,l[i]);
        n=pow(s,l[i]);
        p=pow(t,l[i]);

        ///ZEROS TO ADD
        z1=pow(T[i][0],l[i])-m_ini;
        z2=pow(T[i][1],l[i])-n_ini;
        z3=pow(T[i][2],l[i])-p_ini;

        //Each row of zeros counted is multiplied by the number of columns that need to be filled of zeros and vice versa
                //rows A    columns A       rows B    columns B       rows C    columns C
        Z[i]=z1*n_ini + z2*(m_ini+z1)+ z2*p_ini + z3*(n_ini+z2) + z1*p_ini+ z3*(m_ini+z1);

        if(l[i]>=L0_op[i] && Z[i]>=0){  //if l is abvobe the threshold          ///ADD TO THE NEW MATRIX THE CORRESPONDING ZEROS

            double **A;
            allocate_matrix_memory(m,n,&A);
            fill_matrix(m,n,A,m_ini,n_ini,A_ini);

            double **B;
            allocate_matrix_memory(n,p,&B);
            fill_matrix(n,p,B,n_ini,p_ini,B_ini);

            double**C;
            allocate_matrix_memory(m,p,&C);

            ///MULTIPLICATION
            FILE *tensor;
            tensor = fopen(tensor_names[i], "r");

            if(tensor == NULL) {
                printf("Error opening file\n");
                return 1;
            }
            int K;
            fscanf(tensor,"%d",&K);

            double *P[K][3];

```

```
            for(int i=0;i<K;i++){
                P[i][0]=(double*)malloc((r*s)*sizeof(double));
                P[i][1]=(double*)malloc((s*t)*sizeof(double));
                P[i][2]=(double*)malloc((r*t)*sizeof(double));
                if(P[i][0] == NULL || P[i][1] == NULL || P[i][2] == NULL){
                    printf("Memory allocation failed\n");
                    return 1;
                }
            }

            for (int i=0;i<K;i++) {
                for (int k=0;k<r*s;k++)
                    fscanf(tensor, "%lf", &P[i][0][k]);
                for(int k=0;k<s*t;k++)
                    fscanf(tensor, "%lf", &P[i][1][k]);
                for (int k=0;k<r*t;k++)
                    fscanf(tensor, "%lf", &P[i][2][k]);
            }

            fclose(tensor);
            start = clock();
            algorithm_rxsxt(m,n,p,A,B,C,r,s,t,K,P,L0_op[i],start);
            stop = clock();
            if((double)((stop - start) / CLOCKS_PER_SEC)>0.5)
                printf("Tensor %d,%d,%d:\n  - zeros: %d\n  - needed l: %d\n  - L0(threshold): %d\n  - time: more than 0.5 seconds\n\n",r,s,t,Z[i],l[i],L0_op[i]);
            else
                printf("Tensor %d,%d,%d:\n  - zeros: %d\n  - needed l: %d\n  - L0(threshold): %d\n  - time: %6.3lf seconds\n\n",r,s,t,Z[i],l[i],L0_op[i],(double)(stop - start) / CLOCKS_PER_SEC);
            free_matrix(m,A);
            free_matrix(n,B);
            free_matrix(m,C);

            for(int i=0;i<K;i++){
                    free(P[i][0]);
                    free(P[i][1]);
                    free(P[i][2]);
            }
        }
        else
            printf("Tensor %d,%d,%d:\n  - Too many zeros, no memory\n  - needed l: %d\n  - L0(threshold): %d\n\n",r,s,t,l[i],L0_op[i]);
    }
}
                                                                         //rows columns
void combination_submatrices(int m, int n, double **A, int f, int c, double **A1, double *coef){
    int m1=m/f;
    int n1=n/c;
    matrix_of_zeros(m1,n1,A1);
    for(int i=0;i<f;i++){   //rows of matrices
        for(int j=0;j<c;j++){  //columns of matrices
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //rows of each matrix
                    for(int l=0;l<n1;l++){   //columns of each matrix
                        A1[k][l]=A1[k][l]+A[i*m1+k][j*n1+l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}
                                                                      //rows columns
void recombination_submatrix(int m, int n, double **C, int f, int c, double **A1, double *coef){
    int m1=m/f;
    int n1=n/c;
```

```c
    for(int i=0;i<f;i++){   //rows of matrices
        for(int j=0;j<c;j++){  //columns of matrices
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //rows of each matrix
                    for(int l=0;l<n1;l++){   //columns of each matrix
                        C[i*m1+k][j*n1+l]+=A1[k][l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}

 //m: rows A, n: columns A i rows B, p: columns B
void algorithm_rxsxt(int m, int n, int p, double **A, double **B, double **C,int r, int s, int
    t, int K, double *P[K][3],int L0){
    if(m<=pow(r,L0))
        ordinary_product(m,n,p,A,B,C);
    else{
        matrix_of_zeros(m,p,C);
        double **A1;
        double **B1;
        double **P1;

        //Els apunts diuen que perds temps reservant les matrices?
        allocate_matrix_memory(m/r,n/s,&A1);  //m=r*m1, n=s*n1:  sizes A1: m1 x n1
        allocate_matrix_memory(n/s,p/t,&B1);  //n=s*n1, p=t*p1   sizes B1: n1 x p1
        allocate_matrix_memory(m/r,p/t,&P1);  //m=r*m1, p=t*p1   sizes A1: m1 x p1
        for(int i=0;i<K;i++){
            combination_submatrices(m,n,A,r,s,A1,P[i][0]); //lineal combination of A saved in
                A1
            combination_submatrices(n,p,B,s,t,B1,P[i][1]);  //lineal combination of B saved in
                B1
            algorithm_rxsxt(m/r,n/s,p/t,A1,B1,P1,r,s,t,K,P,L0); //multiplication: A1*B1
            recombination_submatrice(m,p,C,r,t,P1,P[i][2]); //Add P1 ("P_i") in the
                corresponding C position
        }
        free_matrix(m/r,A1);
        free_matrix(n/s,B1);
        free_matrix(m/r,P1);
    }
}

void matrix_of_zeros(int m, int n, double **M){
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=0;
        }
    }
}

void fill_matrix(int m,int n, double **A,int m_ini,int n_ini,double **A_ini){
    if(m<m_ini || n<n_ini){
        printf("The new matrix is smaller than the original!!!\n");
    }
    for(int i=0;i<m_ini;i++){
        for(int j=0;j<n_ini;j++){
            A[i][j]=A_ini[i][j];
        }
    }
    for(int i=m_ini;i<m;i++){
        for(int j=n_ini;j<n;j++){
            A[i][j]=0;
        }
    }
}

void allocate_matrix_memory(int m, int n, double ***M){
```

```c
    (*M) = (double **)malloc(m*sizeof(double *));
    for(int i=0;i<m;i++){
        (*M)[i]=(double *)malloc(n*sizeof(double));
    }
    if ((*M) == NULL) {
        printf("Memory allocation failed\n");
    }
}

void free_matrix(int m, double **M) {
    for (int i = 0; i < m; i++) {
        free(M[i]);
    }
    free(M);
}

void ordinary_product(int m, int n, int p, double **A, double **B, double **C){
    matrix_of_zeros(m,p,C);
    for(int m_ite=0;m_ite<m;m_ite++){
        for(int p_ite=0;p_ite<p;p_ite++){
            for(int j=0;j<n;j++){
                C[m_ite][p_ite] = C[m_ite][p_ite] + A[m_ite][j]*B[j][p_ite];
            }
        }
    }
}

int calculate_l(int m_ini,int n_ini,int p_ini,int Ti[3]){
    double arr1,arr2,arr3;

    arr1=(log(m_ini)/log(Ti[0]));
    arr2=(log(n_ini)/log(Ti[1]));
    arr3=(log(p_ini)/log(Ti[2]));

    int l1,l2,l3;
    //With the power arr is not enough, it is smaller than the original number. We need the following potence
    l1=find_l(arr1);
    l2=find_l(arr2);
    l3=find_l(arr3);

    //We grab the bigger l to find the coloums and rows of zeros to add. We need the matrix to be bigger than all of the original
    return max3(l1,l2,l3);
}

int find_l(double arr){
    if((arr-(int)arr)==0) //Is an exact power
        return (int)arr;
    else
        return (int)(arr+1);
}
int max3(int l1,int l2,int l3){
    if(l1>=l2)
        if(l1>=l3)
            return l1;
        else
            return l3;
    else
        if(l2>=l3)
            return l2;
        else
            return l3;
}

///threshold FUNCTIONS


```

```c
void optimal_threshold(char *tensor_names[93],int L0_op[93]){

    int cnt = 0;
    while(cnt<93) {
        // Open the file for reading
        FILE *tensor = fopen(tensor_names[cnt], "r");
        if(tensor == NULL) {
            printf("Error opening file\n");
            //return 1;
            continue;
        }
        int r,s,t;
        sscanf(tensor_names[cnt],"tensor_%dx%dx%d.txt", &r, &s, &t);
        int K;
        fscanf(tensor,"%d",&K);
        double *P[K][3];
        for(int i=0;i<K;i++){
            P[i][0]=(double*)malloc((r*s)*sizeof(double));
            P[i][1]=(double*)malloc((s*t)*sizeof(double));
            P[i][2]=(double*)malloc((r*t)*sizeof(double));
            if(P[i][0] == NULL || P[i][1] == NULL || P[i][2] == NULL){
                printf("Memory allocation failed\n");
            }
        }

        for (int i=0;i<K;i++) {
            for (int k=0;k<r*s;k++)
                fscanf(tensor,"%lf", &P[i][0][k]);
            for (int k=0;k<s*t;k++)
                fscanf(tensor,"%lf", &P[i][1][k]);
            for (int k=0;k<r*t;k++)
                fscanf(tensor,"%lf", &P[i][2][k]);
        }
        fclose(tensor);

        int a,b,c;

        a=additions_tensor_rxs(r,s,K,P);
        b=additions_tensor_sxt(s,t,K,P);
        c=additions_tensor_rxt(r,t,K,P);
        L0_op[cnt]=min_integer_optimal(r,s,t,K,a,b,c,P);

        // Free memory
        for (int i = 0; i < K; i++) {
            for (int j = 0; j < 3; j++) {
                free(P[i][j]);
            }
        }
        cnt++;
    }
}

int min_integer_optimal(int r, int s, int t, int K, int a, int b, int c, double *P[K][3]){
    double x1,x2;
    for(int i=0;i<15;i++){
        x1=f_ord(r,s,t,(double)i);
        x2=f_op(r,s,t,(double)i,K,a,b,c);
        if(x1>x2){
            return i;
        }
    }
    return -100;
}

double f_ord(int r, int s, int t, double l){
    return pow(r,l)*pow(t,l)*(2*pow(s,l)-1);
}

```

```c
double f_op(int r, int s, int t, double l, int K, int a, int b, int c){
    double sumaord,sumaa,sumab,sumac;
    double k_aux=K;
    sumaord=k_aux*pow(r,l-1)*pow(t,l-1)*(2*pow(s,l-1)-1);
    sumaa=a*pow(r,l-1)*pow(s,l-1);
    sumab=b*pow(s,l-1)*pow(t,l-1);
    sumac=c*pow(r,l-1)*pow(t,l-1);
    return(sumaord+sumaa+sumab+sumac);
}

int additions_tensor_rxs(int r, int s, int K, double *P[K][3]){
    int rxs=0;  //additions of matrices r x s
    int sum;
    //additions r x s:
    for(int k=0;k<K;k++){
        sum=0;
        for(int i=0;i<(r*s);i++){
            if(P[k][0][i]!=0)
                sum++;
        }
        rxs=rxs+(sum-1);  // A each matriu hi ha el valor of uns menys 1.
    }
    return rxs;
}

int additions_tensor_sxt(int s, int t, int K, double *P[K][3]){
    int sxt=0;  //additions of matrices s x t
    int sum;
    //additions s x t:
    for(int k=0;k<K;k++){
        sum=0;
        for(int i=0;i<(s*t);i++){
            if(P[k][1][i]!=0)
                sum++;
        }
        sxt=sxt+(sum-1);  // A each matriu hi ha el valor of uns menys 1.
    }
    return sxt;
}

int additions_tensor_rxt(int r, int t, int K, double *P[K][3]){
    int rxt=0;  //additions of matrices r x t
    int sum;
    //additions r x t:
    for(int i=0;i<(r*t);i++){
            sum=0;
        for(int k=0;k<K;k++){
            if(P[k][2][i]!=0)
                sum++;
        }
        rxt=rxt+(sum-1);
    }
    return rxt;
}
```

F Final Program

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <malloc.h>
int calculate_l(int m_ini,int n_ini,int p_ini,int Ti[3]);
int max3(int l1,int l2,int l3);
int find_l(double arr);
void random_matrix_to_tensor_matrix(int m_ini,int n_ini,int p_ini,int T[93][3], int rst[4]);

void matrix_of_zeros(int m, int n, double **M);
void allocate_matrix_memory(int m, int n, double ***M);
void matrix_redimension(int m_ini, int n_ini, int m, int n, double ***M);
void ordinary_product(int m, int n, int p, double **A, double **B, double **C);
void submatrices_combination(int m, int n, double **A, int f, int c, double **A1, double *coef);
void submatrix_recombination(int m, int n, double **C, int f, int c, double **A1, double *coef);
void algorithm_rxsxt(int m, int n, int p, double **A, double **B, double **C,int r, int s, int t, int K, double *P[K][3],int L0);
void free_matrix(int m, double **M);

int additions_tensor_rxs(int r, int s, int K, double *P[K][3]);
int additions_tensor_sxt(int s, int t, int K, double *P[K][3]);
int additions_tensor_rxt(int r, int t, int K, double *P[K][3]);

double f_ord(int r, int s, int t, double l);
double f_op(int r, int s, int t, double l, int K, int a, int b, int c);

int min_integer_optimal(int r, int s, int t, int K, int a, int b, int c, double *P[K][3]);
int minimZero(int m_ini,int n_ini,int p_ini,int L0_op[93],int aux[5]);
void optimal_threshold(char *tensor_names[93],int L0_op[93]);

char *tensor_names[93] = {"tensor_2x2x2.txt",
                           "tensor_2x2x3.txt",
                           "tensor_2x2x4.txt",
                           "tensor_2x2x5.txt",
                           "tensor_2x2x6.txt",
                           "tensor_2x2x7.txt",
                           "tensor_2x2x8.txt",
                           "tensor_2x3x3.txt",
                           "tensor_2x3x4.txt",
                           "tensor_2x3x5.txt",
                           "tensor_2x4x4.txt",
                           "tensor_2x4x5.txt",
                           "tensor_2x5x5.txt",
                           "tensor_3x3x3.txt",
                           "tensor_3x3x4.txt",
                           "tensor_3x3x5.txt",
                           "tensor_3x4x11.txt",
                           "tensor_3x4x4.txt",
                           "tensor_3x4x5.txt",
                           "tensor_3x5x5.txt",
                           "tensor_3x5x9.txt",
                           "tensor_3x9x11.txt",
                           "tensor_4x4x4.txt",
                           "tensor_4x4x5.txt",
                           "tensor_4x5x10.txt",
                           "tensor_4x5x11.txt",
                           "tensor_4x5x5.txt",
                           "tensor_4x5x9.txt",
                           "tensor_4x9x10.txt",
                           "tensor_4x9x11.txt",
                           "tensor_4x11x11.txt",
```

```
                                "tensor_4x11x12.txt",
                                "tensor_5x5x5.txt",
                                "tensor_5x5x7.txt",
                                "tensor_5x7x10.txt",
                                "tensor_5x7x11.txt",
                                "tensor_5x7x9.txt",
                                "tensor_5x8x10.txt",
                                "tensor_5x8x11.txt",
                                "tensor_5x8x9.txt",
                                "tensor_5x9x10.txt",
                                "tensor_5x9x11.txt",
                                "tensor_5x9x12.txt",
                                "tensor_5x9x9.txt",
                                "tensor_6x7x10.txt",
                                "tensor_6x7x11.txt",
                                "tensor_6x7x9.txt",
                                "tensor_6x8x10.txt",
                                "tensor_6x8x11.txt",
                                "tensor_6x9x10.txt",
                                "tensor_6x9x11.txt",
                                "tensor_6x9x9.txt",
                                "tensor_7x7x10.txt",
                                "tensor_7x7x11.txt",
                                "tensor_7x7x9.txt",
                                "tensor_7x8x10.txt",
                                "tensor_7x8x11.txt",
                                "tensor_7x8x12.txt",
                                "tensor_7x8x9.txt",
                                "tensor_7x9x10.txt",
                                "tensor_7x9x11.txt",
                                "tensor_7x9x12.txt",
                                "tensor_7x9x9.txt",
                                "tensor_7x10x10.txt",
                                "tensor_7x10x11.txt",
                                "tensor_7x11x11.txt",
                                "tensor_8x8x10.txt",
                                "tensor_8x8x11.txt",
                                "tensor_8x9x10.txt",
                                "tensor_8x9x11.txt",
                                "tensor_8x9x12.txt",
                                "tensor_8x10x10.txt",
                                "tensor_8x10x11.txt",
                                "tensor_8x10x12.txt",
                                "tensor_8x11x11.txt",
                                "tensor_8x11x12.txt",
                                "tensor_9x9x10.txt",
                                "tensor_9x9x11.txt",
                                "tensor_9x9x9.txt",
                                "tensor_9x10x10.txt",
                                "tensor_9x10x11.txt",
                                "tensor_9x10x12.txt",
                                "tensor_9x11x11.txt",
                                "tensor_9x11x12.txt",
                                "tensor_10x10x10.txt",
                                "tensor_10x10x11.txt",
                                "tensor_10x10x12.txt",
                                "tensor_10x11x11.txt",
                                "tensor_10x11x12.txt",
                                "tensor_10x12x12.txt",
                                "tensor_11x11x11.txt",
                                "tensor_11x11x12.txt",
                                "tensor_11x12x12.txt"};


int main(){

    ///READING OF MATRIX A
    FILE *A_txt;
```

```c

    char A_file[20];

    printf("Matrix A: ");
    scanf("%s",A_file);
    A_txt = fopen(A_file, "r");

    if(A_txt == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    int m_ini=0;
    int n_ini=0;
    char c;
    while((c = fgetc(A_txt)) != EOF) {
        if(c == '\n')
            m_ini++; // Counts each \n
    }
    rewind(A_txt);
    while((c = fgetc(A_txt)) != '\n'){
        if(c == ' ') // Counts the black spaces
            n_ini++;
    }
    rewind(A_txt);

    double **A;
    allocate_matrix_memory(m_ini,n_ini,&A);

    for (int i=0;i<m_ini;i++) {
        for (int j=0;j<n_ini;j++)
            fscanf(A_txt, "%lf", &A[i][j]);
    }
    fclose(A_txt);

    ///READING MATRIX B
    FILE *B_txt;

    char B_file[20];

    printf("Matrix B: ");
    scanf("%s",B_file);
    B_txt = fopen(B_file, "r");

    if(B_txt == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    int n_ini_B=0;
    int p_ini=0;
    while((c = fgetc(B_txt)) != EOF) {
        if(c == '\n')
            n_ini_B++; // counts each \n
    }
    if(n_ini != n_ini_B){
        printf("The number of columns of A does not match the number of rows if B\n");
        return 1;
    }
    rewind(B_txt);
    while((c = fgetc(B_txt)) != '\n'){
        if(c == ' ') //Conuts the blank spaces
            p_ini++;
    }
    rewind(B_txt);

    double **B;
    allocate_matrix_memory(n_ini,p_ini,&B);

    for (int i=0;i<n_ini;i++) {
```

```
        for (int j=0;j<p_ini;j++)
            fscanf(B_txt, "%lf", &B[i][j]);
    }
    fclose(B_txt);

    printf("Initial matrix multiplication of size %dx%dx%d\n",m_ini,n_ini,p_ini);

    clock_t start, stop;

    ///OPTIMAL THRESHOLD OF EACH TENSOR
    int L0_op[93]={5,4,4,5,4,4,4,4,4,4,4,4,4,4,3,3,3,3,3,3,3,4,3,3,3,3,3,3,
                   3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
                   3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
                   3,3,3,3,3,3,3,3,3};
    ///ZEROS OF EACH TENSOR
    // Choses the tensor with the minimal number of zeros
    int aux[5];
    minimZero(m_ini,n_ini,p_ini,L0_op,aux);
    int r,s,t,l,L0;
    r=aux[0];
    s=aux[1];
    t=aux[2];
    l=aux[3];
    L0=aux[4];

    if(l<L0){ //if the power needed l is smaller that the threshold: ordinary multiplication
        double**C;
        allocate_matrix_memory(m_ini,p_ini,&C);
        start=clock();
        ordinary_product(m_ini,n_ini,p_ini,A,B,C);
        stop=clock();
        printf("The best option is the ordinary product. Time:%6.3lf\n",(double)(stop - start) / CLOCKS_PER_SEC);
    }
    else{ // If l>threshold, tensor product
        int m=pow(r,l);
        int n=pow(s,l);
        int p=pow(t,l);

        ///ORDINARY PRODUCT
        double**C_ord;
        allocate_matrix_memory(m_ini,p_ini,&C_ord);

        start = clock();
        ordinary_product(m_ini,n_ini,p_ini,A,B,C_ord);
        stop = clock();
        printf("Ordinary multiplcation time: %6.3lf\n",(double)(stop - start) / CLOCKS_PER_SEC);
        free_matrix(m_ini,C_ord);

        ///ADD ZEROS TO THE NEW MATRIX

       matrix_redimension(m_ini,n_ini,m,n,&A);
       matrix_redimension(n_ini,p_ini,n,p,&B);

        double**C;
        allocate_matrix_memory(m,p,&C);

        /// TENSOR READING
        FILE *tensor;

        char tensor_file[20];
        sprintf(tensor_file, "tensor_%dx%dx%d.txt", r, s, t);
        tensor = fopen(tensor_file, "r");

        if(tensor == NULL) {
            printf("Error opening file\n");
            return 1;
```

```
        }
        int K;
        fscanf(tensor,"%d",&K);

        double *P[K][3];

        for(int i=0;i<K;i++){
            P[i][0]=(double*)malloc((r*s)*sizeof(double));
            P[i][1]=(double*)malloc((s*t)*sizeof(double));
            P[i][2]=(double*)malloc((r*t)*sizeof(double));
            if(P[i][0] == NULL || P[i][1] == NULL || P[i][2] == NULL){
                printf("Memory allocation failed\n");
                return 1;
            }
        }

        for (int i=0;i<K;i++) {
            for (int k=0;k<r*s;k++)
                fscanf(tensor, "%lf", &P[i][0][k]);
            for(int k=0;k<s*t;k++)
                fscanf(tensor, "%lf", &P[i][1][k]);
            for (int k=0;k<r*t;k++)
                fscanf(tensor, "%lf", &P[i][2][k]);
        }
        fclose(tensor);

        /// COMPUTE THE TENSOR MULTIPLICATION
        start = clock();
        algorithm_rxsxt(m,n,p,A,B,C,r,s,t,K,P,L0);
        stop = clock();
        printf("Tensor multiplication time: %6.3lf\n",(double)(stop - start) / CLOCKS_PER_SEC);

        free_matrix(m,A);
        free_matrix(n,B);
        free_matrix(m,C);

        for(int i=0;i<K;i++){
            free(P[i][0]);
            free(P[i][1]);
            free(P[i][2]);
        }
    }

    return 0;
}
int minimZero(int m_ini,int n_ini,int p_ini,int L0_op[93],int aux[5]){
    //T[i][1,2,3] are the corresponding r s i t fixated of the 93 tensors
    int T[93][3];
    for(int i=0;i<93;i++)
        sscanf(tensor_names[i],"tensor_%dx%dx%d.txt", &T[i][0], &T[i][1], &T[i][2]);

    long int Z[93];
    int l[93];
    int z1,z2,z3;
    ///CALCULATION OF l AND THE NUMBER OF ADDED ZEROS NEEDED
    for(int i=0;i<93;i++){
        l[i]=calculate_l(m_ini,n_ini,p_ini,T[i]);

        int m_aux=pow(T[i][0],l[i]);
        int n_aux=pow(T[i][1],l[i]);
        int p_aux=pow(T[i][2],l[i]);
        if(m_aux<0 || n_aux<0 || p_aux<0){ //IF m n p <0 the power is too big
            Z[i]=-1;
        }
        else{
            z1=pow(T[i][0],l[i])-m_ini;
            z2=pow(T[i][1],l[i])-n_ini;
            z3=pow(T[i][2],l[i])-p_ini;
```

```c
            //Each row of zeros counted is multiplied by the number of columns that need to be
                filled of zeros and vice versa
                //rows A   columns A      rows B   columns B      rows C   columns C
            Z[i]=z1*n_ini + z2*(m_ini+z1)+ z2*p_ini + z3*(n_ini+z2) + z1*p_ini+ z3*(m_ini+z1);
        }
    }
    ///FIND THE POSITION OF THE TENSOR WITH THE MINIMUM NUMBER OF NEEDED ZEROS
    int row=0;
    while(Z[row]<0){
        row=row+1;
    }

    for(int i=0;i<93;i++){
        if(Z[i]>=0){
            if(Z[i]<Z[row]){
                row=i;
            }
        }
    }
    printf("The optimal tensor to choose for %d,%d,%d is %d,%d,%d and l=%d\n",m_ini,n_ini,p_ini
        ,T[row][0],T[row][1],T[row][2],l[row]);
    aux[0]=T[row][0];
    aux[1]=T[row][1];
    aux[2]=T[row][2];
    aux[3]=l[row];
    aux[4]=L0_op[row];
}

int calculate_l(int m_ini,int n_ini,int p_ini,int Ti[3]){
    double arr1,arr2,arr3;

    arr1=(log(m_ini)/log(Ti[0]));
    arr2=(log(n_ini)/log(Ti[1]));
    arr3=(log(p_ini)/log(Ti[2]));

    int l,l1,l2,l3;
    //With the power arr in't enough, because it's smaller than the original number. We need
        the following power
    l1=find_l(arr1);
    l2=find_l(arr2);
    l3=find_l(arr3);

    //We grab the bigger l to find the columns and rows of zeros to add
    //The bigger because is the minimum for r,s or t, and we need the matrix to be bigger than
        all of the original
    return max3(l1,l2,l3);
}

int max3(int l1,int l2,int l3){
    if(l1>=l2)
        if(l1>=l3)
            return l1;
        else
            return l3;
    else
        if(l2>=l3)
            return l2;
        else
            return l3;
}

int find_l(double arr){
    if((arr-(int)arr)<0.00001) //Is an exact power
        return (int)arr;
    else
        return (int)(arr+1);

}
```

```
                                                                    //rows columns
void submatrices_combination(int m, int n, double **A, int f, int c, double **A1, double *coef){
    int m1=m/f;
    int n1=n/c;
    matrix_of_zeros(m1,n1,A1);
    for(int i=0;i<f;i++){   //rows of matrices
        for(int j=0;j<c;j++){  //columns of matrices
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //rows of each matrix
                    for(int l=0;l<n1;l++){   //columns of each matrix
                        A1[k][l]=A1[k][l]+A[i*m1+k][j*n1+l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}
                                                                      //rows columns
void submatrix_recombination(int m, int n, double **C, int f, int c, double **A1, double *coef){
    int m1=m/f;
    int n1=n/c;
    for(int i=0;i<f;i++){   //rows of matrices
        for(int j=0;j<c;j++){  //columns of matrices
            if(coef[c*i+j]!=0){
                for(int k=0;k<m1;k++){   //rows of each matrix
                    for(int l=0;l<n1;l++){   //columns of each matrix
                        C[i*m1+k][j*n1+l]+=A1[k][l]*coef[c*i+j];
                    }
                }
            }
        }
    }
}



 //m: rows A, n: columns A i rows B, p: columns B
void algorithm_rxsxt(int m, int n, int p, double **A, double **B, double **C,int r, int s, int t, int K, double *P[K][3],int L0){
    if(m<=pow(r,L0)){
        ordinary_product(m,n,p,A,B,C);
    }
    else{
        matrix_of_zeros(m,p,C);
        double **A1;
        double **B1;
        double **P1;

        allocate_matrix_memory(m/r,n/s,&A1);  //m=r*m1, n=s*n1:  sizes A1: m1 x n1
        allocate_matrix_memory(n/s,p/t,&B1);  //n=s*n1, p=t*p1   sizes B1: n1 x p1
        allocate_matrix_memory(m/r,p/t,&P1);  //m=r*m1, p=t*p1   sizes A1: m1 x p1
        for(int i=0;i<K;i++){
            submatrices_combination(m,n,A,r,s,A1,P[i][0]); //lineal combination of A saved in A1
            submatrices_combination(n,p,B,s,t,B1,P[i][1]);  //lineal combination of B saved in B1
            algorithm_rxsxt(m/r,n/s,p/t,A1,B1,P1,r,s,t,K,P,L0);   //multiplication: A1*B1
            submatrix_recombination(m,p,C,r,t,P1,P[i][2]);  //Add P1 ("P_i") in the corresponding C position
        }
        free_matrix(m/r,A1);
        free_matrix(n/s,B1);
        free_matrix(m/r,P1);
```

```c
    }
}


void matrix_of_zeros(int m, int n, double **M){
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            M[i][j]=0;
        }
    }
}


void allocate_matrix_memory(int m, int n, double ***M){
    (*M) = (double **)malloc(m*sizeof(double *));
    if (*M == NULL) {
        printf("Memory reallocation failed\n");
        exit(1);
    }
    for(int i=0;i<m;i++){
        (*M)[i]=(double *)malloc(n*sizeof(double));
    }

    if ((*M) == NULL) {
        printf("Memory allocation failed\n");
    }
}

void matrix_redimension(int m_ini, int n_ini, int m, int n, double ***M) {
    // Redimensionar les files
    *M = (double **)realloc(*M, m * sizeof(double *));
    if (*M == NULL) {
        printf("Memory reallocation failed\n");
        exit(1);
    }

    // Redimensionar cada fila
    for (int i = 0; i < m; i++) {
        if (i < m_ini) {
            (*M)[i] = (double *)realloc((*M)[i], n*sizeof(double));
        } else {
            (*M)[i] = (double *)malloc(n*sizeof(double));
        }
        if ((*M)[i] == NULL) {
            printf("Memory reallocation failed\n");
            exit(1);
        }
    }

    // Inicializar los nuevos elementos a 0
    for (int i = 0; i < m; i++) {
        for (int j = n_ini; j < n; j++) {
            (*M)[i][j] = 0.0;
        }
    }
    for (int i = m_ini; i < m; i++) {
        for (int j = 0; j < n; j++) {  //podria ser fins n_ini, pero no ens arrisquem...
            (*M)[i][j] = 0.0;
        }
    }
}
                //free
void free_matrix(int m, double **M) {
    for (int i = 0; i < m; i++) {
        free(M[i]);
    }
    free(M);
}
```

```c

void imprimir_matriu(int m, int n, double **M){
    for(int i=0;i<m;i++){
        for(int j=0;j<n;j++){
            printf("%lf    ",M[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}


void ordinary_product(int m, int n, int p, double **A, double **B, double **C){
    matrix_of_zeros(m,p,C);
    for(int m_ite=0;m_ite<m;m_ite++){
        for(int p_ite=0;p_ite<p;p_ite++){
            for(int j=0;j<n;j++){
                C[m_ite][p_ite] = C[m_ite][p_ite] + A[m_ite][j]*B[j][p_ite];
            }
        }
    }
}


///THRESHOLD FUNCTIONS


void optimal_threshold(char *tensor_names[93],int L0_op[93]){

    int cnt = 0;
    while(cnt<93) {
        // Open the file for reading
        FILE *tensor = fopen(tensor_names[cnt], "r");
        if(tensor == NULL) {
            printf("Error opening file\n");
            //return 1;
            continue;
        }
        int r,s,t;
        sscanf(tensor_names[cnt],"tensor_%dx%dx%d.txt", &r, &s, &t);
        int K;
        fscanf(tensor,"%d",&K);
        double *P[K][3];
        for(int i=0;i<K;i++){
            P[i][0]=(double*)malloc((r*s)*sizeof(double));
            P[i][1]=(double*)malloc((s*t)*sizeof(double));
            P[i][2]=(double*)malloc((r*t)*sizeof(double));
            if(P[i][0] == NULL || P[i][1] == NULL || P[i][2] == NULL){
                printf("Memory allocation failed\n");
            }
        }

        for (int i=0;i<K;i++) {
            for (int k=0;k<r*s;k++)
                fscanf(tensor,"%lf", &P[i][0][k]);
            for (int k=0;k<s*t;k++)
                fscanf(tensor,"%lf", &P[i][1][k]);
            for (int k=0;k<r*t;k++)
                fscanf(tensor,"%lf", &P[i][2][k]);
        }
        fclose(tensor);

        int a,b,c;

        a=additions_tensor_rxs(r,s,K,P);
        b=additions_tensor_sxt(s,t,K,P);
        c=additions_tensor_rxt(r,t,K,P);
        L0_op[cnt]=min_integer_optimal(r,s,t,K,a,b,c,P);
```

```
        // Free memory
        for (int i = 0; i < K; i++) {
            for (int j = 0; j < 3; j++) {
                free(P[i][j]);
            }
        }
        cnt++;
    }
}

int min_integer_optimal(int r, int s, int t, int K, int a, int b, int c, double *P[K][3]){
    double x1,x2;
    for(int i=0;i<15;i++){
        x1=f_ord(r,s,t,(double)i);
        x2=f_op(r,s,t,(double)i,K,a,b,c);
        //printf("%d ord:%lf >? optim:%lf\n\n",i,x1,x2);
        if(x1>x2){
            return i;
        }
    }
    return -100;
}

double f_ord(int r, int s, int t, double l){
    return pow(r,l)*pow(t,l)*(2*pow(s,l)-1);
}

double f_op(int r, int s, int t, double l, int K, int a, int b, int c){
    double sumaord,sumaa,sumab,sumac;
    double k_aux=K;
    sumaord=k_aux*pow(r,l-1)*pow(t,l-1)*(2*pow(s,l-1)-1);
    sumaa=a*pow(r,l-1)*pow(s,l-1);
    sumab=b*pow(s,l-1)*pow(t,l-1);
    sumac=c*pow(r,l-1)*pow(t,l-1);
    return(sumaord+sumaa+sumab+sumac);
}

int additions_tensor_rxs(int r, int s, int K, double *P[K][3]){
    int rxs=0;  //additions de matrius r x s
    int sum;
    //additions r x s:
    for(int k=0;k<K;k++){
        sum=0;
        for(int i=0;i<(r*s);i++){
            if(P[k][0][i]!=0)
                sum++;
        }
        rxs=rxs+(sum-1);  // A cada matriu hi ha el valor de uns menys 1.
    }
    return rxs;
}

int additions_tensor_sxt(int s, int t, int K, double *P[K][3]){
    int sxt=0;  //additions de matrius s x t
    int sum;
    //additions s x t:
    for(int k=0;k<K;k++){
        sum=0;
        for(int i=0;i<(s*t);i++){
            if(P[k][1][i]!=0)
                sum++;
        }
        sxt=sxt+(sum-1);  // A cada matriu hi ha el valor de uns menys 1.
    }
    return sxt;
}
```

```
int additions_tensor_rxt(int r, int t, int K, double *P[K][3]){
    int rxt=0;  //additions de matrius r x t
    int sum;
    //additions r x t:
    for(int i=0;i<(r*t);i++){
            sum=0;
        for(int k=0;k<K;k++){
            if(P[k][2][i]!=0)
                sum++;
        }
        rxt=rxt+(sum-1);
    }
    return rxt;
}
```