National Polytechnic Institute
Superior School of Computer Sciences

Algorithm Analysis.

Practice 5 - Strassen's Algorithm.

Hernandez Martinez Carlos David.
Burciaga Ornelas Rodrigo Andres.

davestring@outlook.com.
andii_burciaga@live.com.

Group: 2cv3.

October 12, 2017

Contents

1 Introduction

If you have seen matrices before, then you probably know how to multiply them. If A = $a_{i\ j}$ and B = $b_{i\ j}$ are square ***n x n*** matrices, then in the product C = $A \bullet B$ we define the entry $c_{i\ j}$ for i, j = 1, 2, ..., n, by

$$c_{i\ j} = \sum_{k\ =\ 1}^{n} a_{ik} \bullet b_{kj} \tag{1}$$

We must compute n^2 matrix entries, and each is the sum of n values. The following procedure takes ***n x n*** matrices A and B and multiplies them, returning their ***n x n*** product C . We assume that each matrix has an attribute rows, giving the number of rows in the matrix.

```
1  def SQUARE–MATRIX–MULTIPLY ( A, B ):
2      n = A.rows
3      let C be a new n x n matrix
4      for i = 1 to n do:
5          for j = 1 to n do:
6              Cij = 0
7              for k = 1 to n do:
8                  Cij = Cij + aik * bkj
9      return C
```

The **SQUARE-MATRIX-MULTIPLY** procedure works as follows. The ***for*** loop of lines 4 − 8 computes the entries of each row ***i***, and within a given row ***i***, the ***for*** loop of lines 5 − 8 computes each of the entries c_{ij} for each column ***j***. Line 6 initializes c_{ij} to 0 as we start computing the sum given in equation 1, and each iteration of the ***for*** loop of lines 7 − 8 adds in one more term of equation 1.

Because each of the triply-nested for loops runs exactly n iterations, and each execution of line 8 takes constant time, the **SQUARE-MATRIX-MULTIPLY** procedure takes θ (n^3) time.

You might at first think that any matrix multiplication algorithm must take Ω (n^3) time, since the natural definition of matrix multiplication requires that many multiplications. You would be incorrect, however: we have a way to multiply matrices in O (n^3). In this practice, we shall see Strassens remarkable recursive algorithm for multiplying ***n x n*** matrices. It runs in Ω ($n^{log_2(7)}$). Since $log_2(7)$ lies between 2.80 and 2.81, Strassens algorithm runs in O ($n^{2.81}$). Which is asymptotically better than the simple **SQUARE-MATRIX-MULTIPLY** procedure.

2 Basic Concepts:

The ***Strassen's*** algorithm, implements the divide-and-conquer paradigm.

2.1 Divide-and-Conquer Paradigm:

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- ***Divide:*** Divide the problem into a number of sub-problems that are smaller instances of the same problem.
- ***Conquer:*** Conquer the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.
- ***Combine:*** Combine the solutions to the sub-problems into the solution for the original problem.

2.2 Strassen's Algorithm:

The key to ***Strassen's*** method is to make the recursion tree slightly less bushy. That is, instead of performing eight recursive multiplications of $\frac{n}{2}\ x\ \frac{n}{2}$ matrices, it performs only seven. The ***Strassen's*** has four steps:

1. Divide the input matrices A and B and output matrix C into $\frac{n}{2}\ x\ \frac{n}{2}$ sub-matrices. This steep takes θ (1) time by index calculation.

2. Create 10 matrices: $S_1\ +\ S_2\ +\ ...\ +\ S_{10}$ each of which is $\frac{n}{2}\ x\ \frac{n}{2}$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in θ (n^2) time.

3. Using the sub-matrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products: $P_1\ +\ P_2\ +\ ...\ +\ P_7$. Each matrix P_i it's also $\frac{n}{2}\ x\ \frac{n}{2}$.

4. Compute the desired sub-matrices: $C_{11},\ C_{12},\ C_{21},\ C_{22}$. Of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four sub-matrices in θ (n^2) time.

Let us assume that once the matrix ***size n*** gets down to 1, we perform a simple scalar multiplication. When $\mathbf{n} > \mathbf{1}$ steps 1, 2, and 4 take a total of θ (n^2) time, and step 3 requires us to perform seven multiplications of $\frac{n}{2}\ x\ \frac{n}{2}$. Hence, we obtain the following recurrence for the running time ***T (n)*** of ***Strassen's*** algorithm:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 7\ T\ (\frac{n}{2}) + \theta\ (\ n^2\ ) & \text{if } n > 1 \end{cases} \tag{2}$$

We have traded off one matrix multiplication for a constant number of matrix additions. Once we understand recurrences and their solutions, we shall see that this trade-off actually leads to a lower asymptotic running time.

We now proceed to describe the details. In step 2, we create the following 10 matrices:

$$\begin{aligned} S_1 &= B_{12} - B_{22} \\ S_2 &= A_{11} + A_{12} \\ S_3 &= A_{21} + A_{22} \\ S_4 &= B_{21} - B_{11} \\ S_5 &= A_{11} + A_{22} \\ S_6 &= B_{11} + B_{22} \\ S_7 &= A_{12} - A_{22} \\ S_8 &= B_{21} + B_{22} \\ S_9 &= A_{11} - A_{21} \\ S_{10} &= B_{11} + B_{12} \end{aligned}$$

Since we must add or subtract $\frac{n}{2}\ x\ \frac{n}{2}$ matrices 10 times, this step does indeed take $\theta\ (\ n^2\ )$ time.

In step 3, we recursively multiply $\frac{n}{2}\ x\ \frac{n}{2}$ matrices seven times to compute the following $\frac{n}{2}\ x\ \frac{n}{2}$ matrices, each of which is the sum or difference of products of A and B sub-matrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 = (\ A_{11} + A_{22}\ ) \cdot (\ B_{11} + B_{22}\ ) \\
P_6 &= S_7 \cdot S_8 = (\ A_{12} - A_{22}\ ) \cdot (\ B_{21} + B_{22}\ ) \\
P_7 &= S_9 \cdot S_{10} = (\ A_{11} - A_{21}\ ) \cdot (\ B_{11} + B_{12}\ )
\end{aligned}
$$

Note that the only multiplications we need to perform are those in the middle column of the above equations. The right-hand column just shows what these products equal in terms of the original sub-matrices created in step 1.

Step 4 adds and subtracts the P_i matrices created in step 3 to construct the four $\frac{n}{2}\ x\ \frac{n}{2}$ sub-matrices of the product C . We start with:

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

Thus, we see that ***Strassen's*** algorithm, comprising steps $1 - 4$, produces the correct matrix product and that recurrence (2) characterizes its running time. Finally we can conclude that ***Strassen's*** algorithm runs in $\theta\ (\ n^{2.81}\ )$.

3 Development:

Based on the procedure explained in sub-section 2.2, in the following sub-sections we will implement the ***Strassen's*** algorithm. I divided the program in 8 python modules, to have a better control of the code.

- ***main.py:*** Control the sequence of execution.
- ***strassen.py:*** This module implements the ***Strassen's*** algorithm.
- ***create_matrices.py:*** Creates the matrices **A** and **B**.
- ***sub_matrices.py:*** Divide the matrices **A** and **B** in sub-matrices (step 1 in Section 2.2).
- ***matrix_operations.py:*** Has the methods **add** and **subtract**.
- ***ijk_product.py:*** Implements the usual matrix product algorithm.
- ***plot.py:*** Plot the computational time of this algorithm.
- ***global_variables.py:*** As the name of the module, stores the global variables of the program.

Observation: *The code that we will show bellow doesn't include the counter in each line of the algorithm, this because to make notice only the essential parts.*

3.1 Main.py

This module has the main implementation, as we can see in the code bellow we only call 3 methods:

- create ( ): Create and return the matrices A and B.
- strassen ( ): Implements the strassen algorithm, return the resulting matrix and receive as parameters the matrices A and B.
- printer ( ): This method it's nested in this module, prints on screen the matrices A, B and C.

```
1 if ( __name__ == "__main__" ):
2     A, B = create ( )
3     C = strassen ( A, B )
4     printer ( A, B, C )
```

The module as we mention, has another method, ***printer (A, B, C)*** gives format to the matrices and display them in console.

```
1  def printer ( A, B, C ):
2      assert len ( A ) == len ( B ) == len ( C )
3      print ( "\n\tStrassen Algorithm:" )
4      print ( "\n\tMatrix A:\n" )
5      list ( map ( lambda x: print ( "\t{}".format ( x ) ), A ) )
6      print ( "\n\tMatrix B:\n" )
7      list ( map ( lambda x: print ( "\t{}".format ( x ) ), B ) )
8      print ( "\n\tProduct A * B:\n" )
9      list ( map ( lambda x: print ( "\t{}".format ( x ) ), C ) )
10     n = len ( C )
11     print ( "\n\tWhere: A, B, C     M [ {}x{} ] (    + )\n".format ( n, n ) )
```

Observation: *In line 2 of the code above, the keyword* ***assert*** *validates that both A, B and the product C has the same size, in case that the condition doesn't satisfy the program will stop the execution and throw an error.*

3.2 Strassen.py:

This module is probably the most important, implements the Strassen's algorithm and return the product of the matrices A and B. As we can see in the code bellow, the line 3 validates that both arrays and rows had the same size, this corroborates that A and B are **square matrices**. And line 4 validates that both A and B are of type **list**.

From line 7 to 39 the algorithm is implemented. In case that the condition is fulfilled, then the program will call the method ***usual_matrix_product (A, B)*** and will return the product of A and B. This method it's more like an usual scalar product between two vectors of size 1 because the matrices are of that size. In other words, the matrices only have one element stored.

Observation: *It's negligible which matrix is evaluated in the if condition since both are of the same size.*

In other case, in line 11 the program will calculate the size $\frac{n}{2}$ and will store it in the variable ***n*** then, according to sub-section's 2.2 step 1, will split the input matrices in 4 sub-matrices of size $\frac{n}{2}$ x $\frac{n}{2}$ each one.

From line 15 to 27 we can easily visualize the sub-subsection's 2.2 step 3, but nested in each recursive function we can see that the parameters call the methods ***add (...)*** and ***subtract (...)***, we can say that this nested step is actually sub-subsection's 2.2 step 2.

Observation: *The recursive functions may be in different order than the ones in sub-section 2.2, but are the same.*

Finally, from lines 31 to 39 we can see the sub-section's 2.2 step 4 implemented, and the algorithm will return the sub-matrices C_{11}, C_{12}, C_{21}, C_{22} joined in a resulting matrix **C**.

```
1  def strassen ( A, B ):
2      # Validates the condition of matrices of [ 2^n x 2^n ] size.
3      assert len ( A ) == len ( A [ 0 ] ) == len ( B ) == len ( B [ 0 ] )
4      assert type ( A ) == list and type ( B ) == list
5
6      # Usual matrix product.
7      if ( len ( A ) == 1 ):
8          return usual_matrix_product ( A, B )
9      else:
10     # Strassen algorithm.
11         n = int ( len ( A ) / 2 )
12         # Divide de matrices A and B in eigth sub-matrices of size 2^n/2.
13         A11, A12, A21, A22, B11, B12, B21, B22 = split_in_sub_matrices ( A, B, n )
14         # S1 = strassen ( ( A11 + A22 ), ( B11 + B22 ) )
15         S1 = strassen ( add ( A11, A22 ), add ( B11, B22 ) )
16         # S2 = strassen ( ( A21 + A22 ), ( B11 ) )
17         S2 = strassen ( add ( A21, A22 ), B11 )
18         # S3 = strassen ( ( A11 ), ( B12 - B22 ) )
19         S3 = strassen ( A11, subtract ( B12, B22 ) )
20         # S4 = strassen ( ( A22 ), ( B21 - B11 ) )
21         S4 = strassen ( A22, subtract ( B21, B11 ) )
22         # S5 = strassen ( ( A11 + A12 ), ( B22 ) )
23         S5 = strassen ( add ( A11, A12 ), B22 )
24         # S6 = strassen ( ( A21 - A11 ), ( B11 + B12 ) )
25         S6 = strassen ( subtract ( A21, A11 ), add ( B11, B12 ) )
26         # S7 = strassen ( ( A12 - A22 ), ( B21 + B22 ) )
27         S7 = strassen ( subtract ( A12, A22 ), add ( B21, B22 ) )
28
29         # Expressing Cij in terms of Sk, where C it's the resulting matrix.
30         # C11 = S1 + S4 - S5 + S7
31         C11 = subtract ( add ( add ( S1, S4 ), S7 ), S5 )
32         # C12 = S3 + S5
33         C12 = add ( S3, S5 )
34         # C21 = S2 + S4
35         C21 = add ( S2, S4 )
36         # C22 = S1 - S2 + S3 + S6
37         C22 = add ( subtract ( S1, S2 ), add ( S3, S6 ) )
38         # Joining all the resulting sub-matrices
39         return join_sub_matrices ( C11, C12, C21, C22, n * 2 )
```

3.3 Create_matrices.py

This module it's very simple, only has one method that its only function is create the input matrices A and B, for later apply the Strassen's algorithm and obtain the product of $\mathbf{A} \cdot \mathbf{B}$. In line 3 the program will ask to the user to enter the power of 2, and will store the integer in the variable *power*, then, will calculate 2^k where k = *power* and the result will be stored in the variable **n** where $n \ x \ n$ is the size of the matrices. In lines 5 and 6 the matrices will be created storing values from 0 to 9 randomly. Finally in line 7, the method will return A and B.

```
1 def create ( ):
2     # Size of the matrices: 2^n.
3     power = int ( input ( "\n\tAdd an nth-power: " ) )
4     n = int ( math.pow ( 2, power ) )
5     A = [ [ rnd.randint ( 0, 9 ) for i in range ( n ) ] for j in range ( n ) ]
6     B = [ [ rnd.randint ( 0, 9 ) for i in range ( n ) ] for j in range ( n ) ]
7     return A, B
```

Observation: *In line 5 and 6 the matrices are created using a method called* ***List-Compression****.*

3.4 Matrix_operations.py

This module has 2 methods:

- **add (...).**
- **subtract (...).**

Each method realize a very simply operation, **add (A, B)** as its name says, add/sum the matrices **A** and **B** and **subtract (A, B)**, subtract both matrices and both methods return the resulting matrix **C**.

```
1  def add ( A, B ):
2      n = len ( A )
3      C = [ [ A [ i ] [ j ] + B [ i ] [ j ] for j in range ( n ) ] for i in range ( n ) ]
4      # Return statement.
5      return C
6
7  def subtract ( A, B ):
8      n = len ( A )
9      C = [ [ A [ i ] [ j ] - B [ i ] [ j ] for j in range ( n ) ] for i in range ( n ) ]
10     # Return statement.
11     return C
```

3.5 Sub_matrices.py

This module has two methods:

- ***split_in_sub_matrices (...).***
- ***join_sub_matrices (...).***

According to sub-section's 2.2 step 1, first it's necessary to split the input matrices in sub-matrices of size $\frac{n}{2}\ x\ \frac{n}{2}$. The sub-matrices are declared in lines 2 and 3, from lines 6 to 14 are filled.

This method return the sub-matrices: $A_{11},\ A_{12},\ A_{21},\ A_{22},\ B_{11},\ B_{12},\ B_{21},\ B_{22}$.

```
1  def split_in_sub_matrices ( A, B, n ):
2      # Initialize the sub-matrices.
3      A11, A12, A21, A22 = [ ], [ ], [ ], [ ]
4      B11, B12, B21, B22 = [ ], [ ], [ ], [ ]
5      # Fill the sub-matrices.
6      for i in range ( n ):
7          A11.append ( A [ i ] [ :n ] )
8          A12.append ( A [ i ] [ n: ] )
9          A21.append ( A [ n + i ] [ :n ] )
10         A22.append ( A [ n + i ] [ n: ] )
11         B11.append ( B [ i ] [ :n ] )
12         B12.append ( B [ i ] [ n: ] )
13         B21.append ( B [ n + i ] [ :n ] )
14         B22.append ( B [ n + i ] [ n: ] )
15     # Return statement.
16     return A11, A12, A21, A22, B11, B12, B21, B22
```

The next method, as its name says, join the sub-matrices $C_{11},\ C_{12}.\ C_{21},\ C_{22}$ in a resulting matrix $\mathbf{C}$. When the recursive process finished this matrix of size $n\ x\ n$ it's the product of $\mathbf{A} \cdot \mathbf{B}$. This method it's basically the implementation of sub-section's 2.2 step 4.

```
1  def join_sub_matrices ( C11, C12, C21, C22, n ):
2      sub_size = int ( n / 2 )
3      C = [ [ None for i in range ( n ) ] for j in range ( n ) ]
4      for i in range ( sub_size ):
5          for j in range ( sub_size ):
6              C [ i ] [ j ] = C11 [ i ] [ j ]
7              C [ i ] [ j + sub_size ] = C12 [ i ] [ j ]
8              C [ i + sub_size ] [ j ] = C21 [ i ] [ j ]
9              C [ i + sub_size ] [ j + sub_size ] = C22 [ i ] [ j ]
10     # Return statement.
11     return C
```

Observation: ***None*** *in Python is similar to* ***NULL*** *in C or* ***null*** *in java.*

3.6 ijk_Algorithm.py

This module implements the usual matrix product also know as ***ijk-algorithm***. This algorithms implements 3 **for** loops, and its complexity it's $\theta\ (\ n^3\ )$. In our code this method it's only called when the matrices are bigger or equal than $\mathbf{n} = 2^8$, when this condition it's fulfilled, then the program will compare the complexity of both algorithms by making the product of matrices **A** and **B**.

In lines 3 and 4 the code validates that the input matrices are of the same size, in case that don't accomplish the condition the program will throw an error, same as in the Strassen's algorithm implementation. Then, in line 7 we initialize our resulting matrix and from line 8 to 11 the ***ijk-algorithm*** it's implemented.

```
1  def ijk_product ( A, B ):
2      # Validates the condition of matrices of [ 2^n x 2^n ] size.
3      assert len ( A ) == len ( A [ 0 ] ) == len ( B ) == len ( B [ 0 ] )
4      assert type ( A ) == list and type ( B ) == list
5
6      n = len ( A )
7      C = [ [ 0 for i in range ( n ) ] for j in range ( n ) ]
8      for i in range ( n ):
9          for j in range ( n ):
10             for k in range ( n ):
11                 C [ i ] [ j ] += A [ i ] [ k ] * B [ k ] [ j ]
12     # Return statement.
13     return C
```

3.7 Plot.py

This method, as its name says, plot the Strassen's algorithm complexity for a given 2^n size. In line 13 we can see an ***if*** sentence, as we mention in sub-section 3.6, in case that the input matrices have a size bigger than 2^8 then the program will compare the complexities of both algorithms, in other case, we propose an asymptotic function $g\ (\ n\ ) = \frac{3}{2}\ \cdot\ n^{2.81}$.

```
1  def plot ( ):
2      global function, _function
3      # Window title.
4      plt.figure ( "Strassen's Algorithm", figsize = ( 14, 7 ) )
5      # Graph title.
6      plt.title ( "Strassen ( " + str ( gb.parameters [ -1 ] [ 0 ] ) + ", "
7          + str ( gb.parameters [ -1 ] [ 1 ] ) + " ):", color = ( 0.3, 0.4, 0.6 ),
8              weight = "bold" )
9      # Parameters S ( n ) -size- of the graph.
10     s = list ( map ( lambda x: x [ 0 ], gb.parameters ) )
11     # Parameters T ( t ) -time- of the graph for Strassen.
12     t = list ( map ( lambda x: x [ 1 ], gb.parameters ) )
13     if ( gb.flag == True ):
14         # Compares the complexities of Strassen against ijk algorithms.
15         # Parameters T ( t ) -time- of the graph for ijk.
16         _t = list ( map ( lambda x: x [ 1 ], gb._parameters ) )
17         _function = "ijk-Algorithm function: g ( n ) = n^3"
18     else:
19         # In other case, we propose an asymptotic function g ( n ) = 3/2 n^2.81
20         # Parameters T ( t ) -time- of the graph.
21         _t = list ( map ( lambda x: ( 3/2 ) * x [ 1 ], gb.parameters ) )
22         _function = "Proposed asymptotic function: g ( n ) = 3/2 ( n^2.81 )"
23     # Axes names.
24     plt.xlabel ( "Size ( n )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",
25         size = "large" )
26     plt.ylabel ( "Time ( t )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",
27         size = "large" )
28     # Plot.
29     plt.plot ( s, t, "#778899", linewidth = 3, label = function )
30     plt.plot ( s, _t, "#800000", linestyle = "--", label = _function )
31     plt.legend ( loc = "upper left" )
32     plt.show ( )
```

3.8 Calculating The Temporal Complexity:

For calculate the temporal complexity of Strassen's or IJK algorithms it's necessary to put a counter in each line of the codes and store the results in a list, where each element it's a tuple that in its first element stores the size of the matrices and in the second element stores the counter. In ***global_variables.py*** are declared the variables that we will use for this purpose:

(i) **parameters**: List that store the parameters of the points to plot for Strassens algorithm. Each element it's a tuple that stores the length of the matrices, and the time that the algorithm takes to calculate the product.

(ii) **_parameters**: List that store the parameters of the points to plot for ijk-Algorithm. Each element it's a tuple that stores the length of the matrices, and the time that the algorithm takes to calculate the product.

(iii) **time**: Counter that stores the computational time that Strassens Algorithm takes to calculate the product of two matrices.

(iv) **_time**: Counter that stores the computational time that the ijk-Algorithm takes to calculate the product of two matrices.

As we mention, the variable (iv) stores the time that IJK-Algorithm takes to accomplish it process, then, putting the counter in each line:

```
1  def ijk_product ( A, B ):
2      # Validates the condition of matrices of [ 2^n x 2^n ] size.
3      assert len ( A ) == len ( A [ 0 ] ) == len ( B ) == len ( B [ 0 ] )
4      assert type ( A ) == list and type ( B ) == list
5
6      gb._time += 1
7      n = len ( A )
8      gb._time += 1
9      C = [ [ 0 for i in range ( n ) ] for j in range ( n ) ]
10     gb._time += 1
11     for i in range ( n ):
12         gb._time += 1
13         for j in range ( n ):
14             gb._time += 1
15             for k in range ( n ):
16                 gb._time += 1
17                 C [ i ] [ j ] += A [ i ] [ k ] * B [ k ] [ j ]
18                 gb._time += 1
19             gb._time += 1
20         gb._time += 1
21     gb._time += 1
22     # Return statement.
23     gb._time += 1
24     return C
```

Very similar with the variable (iii):

```
def strassen ( A, B ):
    # Validates the condition of matrices of [ 2^n x 2^n ] size.
    assert len ( A ) == len ( A [ 0 ] ) == len ( B ) == len ( B [ 0 ] )
    assert type ( A ) == list and type ( B ) == list

    # Usual matrix product.
    gb.time += 1
    if ( len ( A ) == 1 ):
        gb.time += 1
        return [ [ A [ 0 ] [ 0 ] * B [ 0 ] [ 0 ] ] ]
    else:
    # Strassen algorithm.
        gb.time += 1
        n = int ( len ( A ) / 2 )
        gb.time += 1
        # Divide de matrices A and B in eigth sub-matrices of size 2^n/2.
        A11, A12, A21, A22, B11, B12, B21, B22 = split_in_sub_matrices ( A, B, n )
        gb.time += 1
        # S1 = strassen ( ( A11 + A22 ), ( B11 + B22 ) )
        S1 = strassen ( add ( A11, A22 ), add ( B11, B22 ) )
        gb.time += 1
        # S2 = strassen ( ( A21 + A22 ), ( B11 ) )
        S2 = strassen ( add ( A21, A22 ), B11 )
        gb.time += 1
        # S3 = strassen ( ( A11 ), ( B12 - B22 ) )
        S3 = strassen ( A11, subtrack ( B12, B22 ) )
        gb.time += 1
        # S4 = strassen ( ( A22 ), ( B21 - B11 ) )
        S4 = strassen ( A22, subtrack ( B21, B11 ) )
        gb.time += 1
        # S5 = strassen ( ( A11 + A12 ), ( B22 ) )
        S5 = strassen ( add ( A11, A12 ), B22 )
        gb.time += 1
        # S6 = strassen ( ( A21 - A11 ), ( B11 + B12 ) )
        S6 = strassen ( subtrack ( A21, A11 ), add ( B11, B12 ) )
        gb.time += 1
        # S7 = strassen ( ( A12 - A22 ), ( B21 + B22 ) )
        S7 = strassen ( subtrack ( A12, A22 ), add ( B21, B22 ) )

        # Expressing Cij in terms of Sk, where C it's the resulting matrix.
        gb.time += 1
        # C11 = S1 + S4 - S5 + S7
        C11 = subtrack ( add ( add ( S1, S4 ), S7 ), S5 )
        gb.time += 1
        # C12 = S3 + S5
        C12 = add ( S3, S5 )
        gb.time += 1
        # C21 = S2 + S4
        C21 = add ( S2, S4 )
        gb.time += 1
        # C22 = S1 - S2 + S3 + S6
        C22 = add ( subtrack ( S1, S2 ), add ( S3, S6 ) )
        # Joining all the resulting sub-matrices
        gb.time += 1
        return join_sub_matrices ( C11, C12, C21, C22, n * 2 )
```

With the counters now set, running the program we can visualize that the value of them for matrices of size $2^0, 2^1, 2^2, 2^3, \ldots, 2^n$ will always be the same, then, if we want to calculate the time that the algorithm takes to solve a product of matrices of size 2^n we send first the previous sizes until reaching the expected one, for example: If we want the time for matrices of size 2^4, first we calculate the complexity for matrices of size 2^0, 2^1, 2^2, 2^3 and finally 2^4. For each iteration we settle the counters in 0, and the size of the resulting matrix and the time that the algorithms takes to solve the product will be stored in the lists ***parameters*** and ***_parameters*** as a tuple [(size, time)]. Finally with this process we will get the point for plotting both Strassen's and IJK algorithms.

The process explained above will be executed by modifying the main method, from lines 5 to 16 we can see that the for loop iterates from 1 to the value of the variable **power**, where *power* will be the nth-power of 2. In lines 10 and 11 we store the tuple in the variable *parameters* and restart the counter *time*, as we mention, this variables belongs to Strassen's algorithm. The same process it's repeated in lines 15 and 16 for IJK-Algorithm in case of satisfying the condition.

```
1  if ( __name__ == "__main__" ):
2      power = int ( input ( "\n\tAdd an nth-power: " ) )
3      if ( power >= 8 ):
4          gb.flag = True
5      for i in range ( 1, power + 1 ):
6          # Strassen algorithm.
7          n = int ( math.pow ( 2, i ) )
8          A, B = create ( n )
9          C = strassen ( A, B )
10         gb.parameters.append ( ( n, gb.time ) )
11         gb.time = 0
12         # For big matrices compare the Strassen's algorithm with the ijk algorithm.
13         if ( gb.flag == True ):
14             _C = ijk_product ( A, B )
15             gb._parameters.append ( ( n, gb._time ) )
16             gb._time = 0
17     printer ( A, B, C )
18     plot ( )
```

The stored values in the variables **parameters** and **_parameters** can be visualized in Figures 4.2.0 and 4.2.2.

4 Results:

All the code shown above doesn't have significance if its operation is not shown. This section will show you the ***console*** output and the graphic of the temporal complexity of the algorithms previously mentioned. Also, I attach a table with the plot points for each test.

4.1 Strassen's Algorithm Results:

First test for ***Strassen's Algorithm***. The program will plot the ***time*** that the algorithm takes to make the product of matrices of size $n = 2^4$.

> ***Observation:*** *This algorithm has the same complexity always, that is because doesn't has worst or best case.*

```
Add an nth-power: 4

Strassen Algorithm

Matrix A:

[9, 8, 3, 6, 7, 3, 5, 3, 2, 8, 7, 8, 0, 7, 1, 8]
[1, 3, 1, 0, 0, 0, 9, 9, 7, 3, 7, 0, 6, 1, 2, 7]
[6, 4, 4, 2, 4, 6, 1, 4, 2, 7, 2, 1, 1, 9, 9, 0]
[8, 8, 4, 1, 0, 1, 5, 9, 7, 3, 2, 1, 4, 6, 1, 0]
[8, 8, 1, 7, 4, 0, 8, 9, 8, 9, 0, 6, 7, 0, 4, 2]
[0, 8, 7, 5, 0, 4, 6, 9, 7, 1, 0, 9, 0, 1, 9, 5]
[5, 0, 8, 7, 9, 7, 9, 0, 7, 2, 7, 3, 3, 2, 1, 2]
[9, 1, 2, 5, 9, 6, 6, 3, 1, 4, 2, 0, 9, 5, 6, 1]
[4, 6, 8, 4, 5, 0, 1, 9, 1, 1, 9, 5, 4, 1, 2, 4]
[0, 6, 1, 3, 9, 5, 6, 4, 4, 6, 1, 9, 3, 4, 1, 9]
[7, 7, 0, 1, 2, 1, 7, 1, 4, 9, 5, 4, 8, 4, 5, 7]
[1, 9, 3, 6, 0, 1, 5, 3, 2, 8, 6, 4, 5, 1, 9, 2]
[3, 9, 6, 1, 4, 2, 4, 6, 0, 5, 6, 2, 8, 1, 4, 3]
[0, 0, 5, 1, 7, 8, 8, 2, 7, 6, 7, 3, 2, 2, 5, 9]
[3, 3, 4, 4, 7, 3, 5, 0, 1, 6, 0, 1, 9, 2, 9, 4]
[0, 0, 1, 6, 8, 4, 7, 7, 0, 4, 6, 2, 9, 4, 0, 6]
```

Figure 4.1.0: Matrix A of size n = 2^4.

```
Matrix B:

[9, 1, 7, 0, 2, 4, 3, 4, 0, 7, 3, 7, 1, 4, 7, 4]
[0, 8, 9, 5, 9, 0, 2, 3, 2, 1, 0, 4, 8, 9, 7, 7]
[6, 2, 6, 2, 7, 5, 5, 1, 6, 3, 3, 0, 6, 5, 2, 5]
[9, 2, 7, 3, 6, 1, 6, 7, 8, 5, 3, 0, 7, 7, 2, 5]
[2, 9, 4, 0, 6, 9, 3, 6, 1, 9, 6, 9, 6, 3, 1, 5]
[8, 5, 8, 1, 3, 5, 7, 8, 9, 4, 2, 4, 8, 9, 4, 6]
[1, 5, 4, 4, 8, 7, 0, 2, 5, 4, 9, 0, 9, 1, 7, 2]
[0, 6, 5, 4, 2, 1, 0, 0, 4, 8, 5, 0, 8, 7, 7, 4]
[2, 2, 1, 5, 4, 1, 7, 3, 8, 3, 6, 7, 5, 3, 8, 1]
[2, 9, 3, 7, 3, 8, 1, 4, 9, 5, 7, 3, 5, 1, 3, 2]
[3, 4, 4, 2, 1, 8, 4, 3, 9, 8, 5, 5, 7, 1, 1, 7]
[9, 5, 3, 4, 4, 8, 6, 6, 8, 2, 0, 4, 9, 7, 0, 9]
[9, 4, 4, 8, 8, 8, 2, 6, 7, 3, 5, 6, 8, 9, 4, 4]
[6, 1, 9, 8, 6, 7, 5, 9, 3, 4, 4, 7, 8, 8, 0, 5]
[9, 3, 2, 5, 9, 7, 7, 0, 9, 8, 9, 9, 1, 9, 0, 4]
[8, 6, 9, 1, 5, 6, 0, 8, 7, 1, 0, 3, 1, 1, 7, 1]
```

Figure 4.1.1: Matrix B of size n = 2^4.

```
Product A * B:

[424, 414, 497, 280, 406, 463, 276, 415, 454, 391, 302, 357, 504, 389, 315, 398]
[199, 268, 265, 232, 277, 279, 125, 187, 351, 258, 282, 194, 348, 226, 301, 190]
[330, 265, 341, 252, 324, 339, 257, 261, 347, 334, 294, 317, 346, 373, 188, 280]
[234, 245, 331, 257, 303, 238, 184, 208, 282, 275, 257, 239, 382, 334, 307, 251]
[360, 397, 383, 336, 428, 369, 242, 300, 450, 378, 371, 315, 496, 417, 384, 326]
[349, 318, 359, 269, 408, 298, 281, 238, 456, 291, 277, 239, 442, 426, 295, 324]
[369, 300, 360, 202, 363, 406, 293, 325, 432, 359, 332, 293, 457, 313, 260, 315]
[399, 301, 365, 246, 376, 411, 242, 322, 360, 383, 346, 347, 402, 383, 245, 292]
[299, 299, 347, 202, 310, 319, 204, 231, 353, 325, 232, 233, 403, 331, 233, 325]
[333, 400, 385, 256, 377, 405, 222, 365, 414, 291, 261, 294, 459, 345, 275, 314]
[365, 351, 372, 308, 388, 419, 206, 317, 424, 311, 319, 337, 405, 337, 307, 288]
[316, 319, 318, 291, 380, 336, 227, 229, 437, 298, 300, 255, 397, 361, 228, 299]
[284, 333, 344, 252, 359, 344, 179, 231, 359, 301, 269, 254, 406, 350, 254, 300]
[346, 357, 350, 230, 358, 441, 258, 320, 490, 339, 340, 307, 418, 290, 273, 279]
[359, 296, 302, 246, 387, 379, 211, 266, 369, 297, 308, 298, 335, 338, 202, 244]
[312, 329, 343, 238, 325, 391, 171, 321, 384, 326, 294, 238, 437, 306, 232, 271]

Where: A, B, C ∈ M[ 16x16 ] ( Z+ )

Strassen's algorithm parameters: [(2, 29), (4, 218), (8, 1541), (16, 10802)]
```

Figure 4.1.2: Product of A · B..

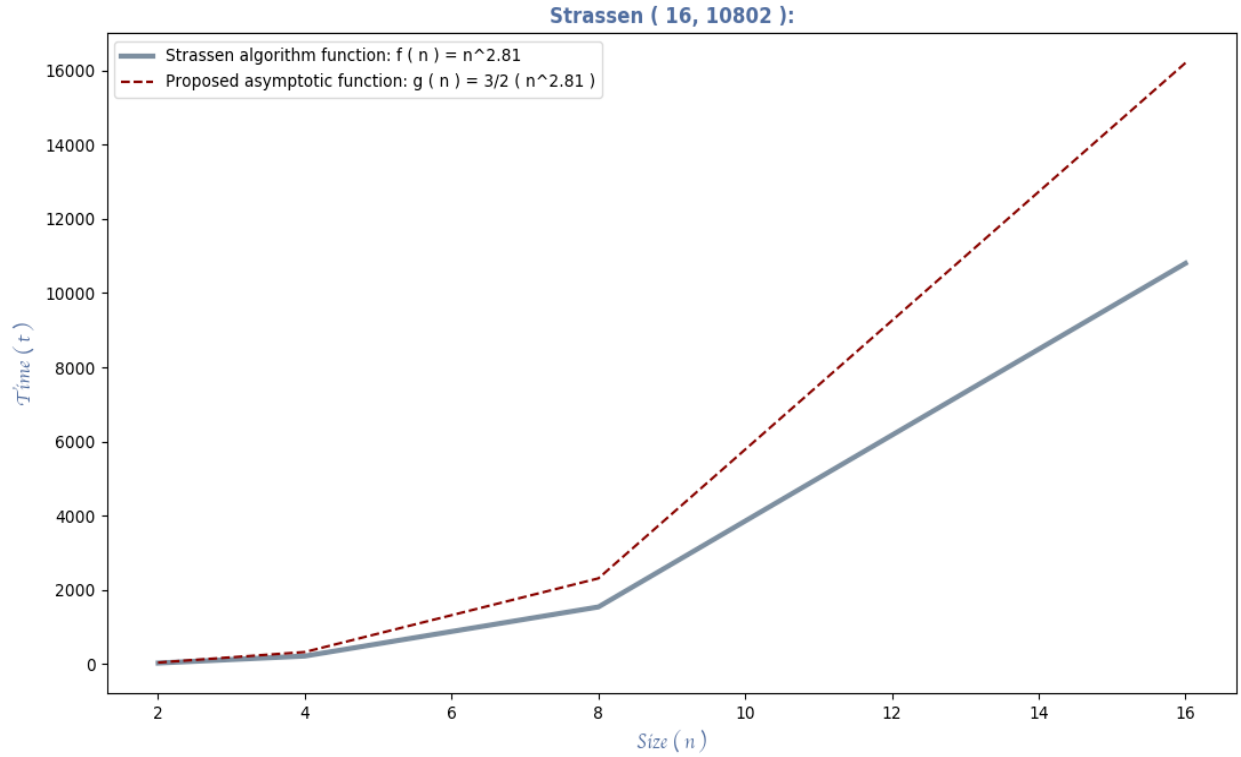

Figure 4.1.3: Graph for Figure 4.1.2.

Observation: *The red function it's the proposed one:* $g\ (\ n\ ) = \frac{3}{2} \cdot n^{2.81}$ *and the blue it's the Stranssen's Algorithm complexity:* $f\ (\ n\ ) = n^{2.81}$.

Size (2^n)	Time (t)
2	29
4	218
8	1541
16	10802

Table 1: Plot point for Strassen's Algorithm complexity of Figure 4.1.3.

Second test for ***Strassen's Algorithm***. The program will plot the ***time*** that the algorithm takes to make the product of matrices of size $n = 2^5$.

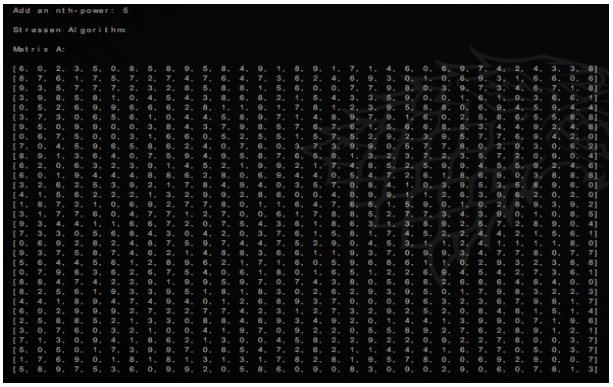

Figure 4.1.4: Matrix A of size n = 2^5.

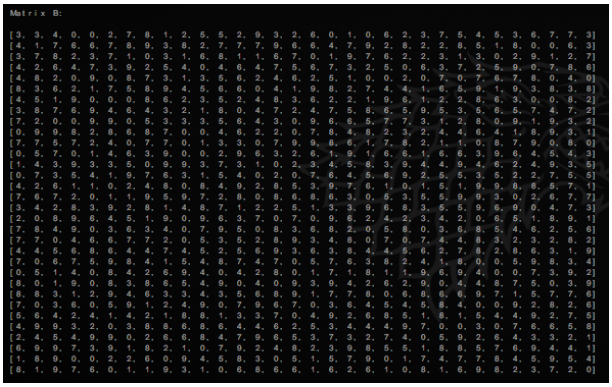

Figure 4.1.5: Matrix B of size n = 2^5.

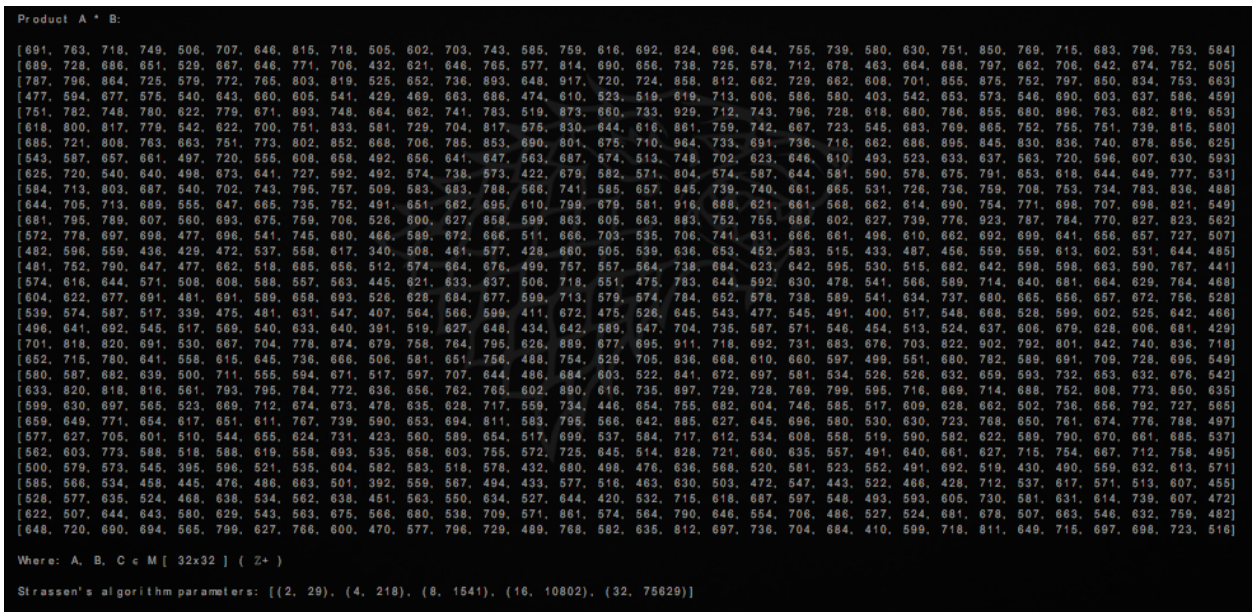

Figure 4.1.6: Product of A · B..

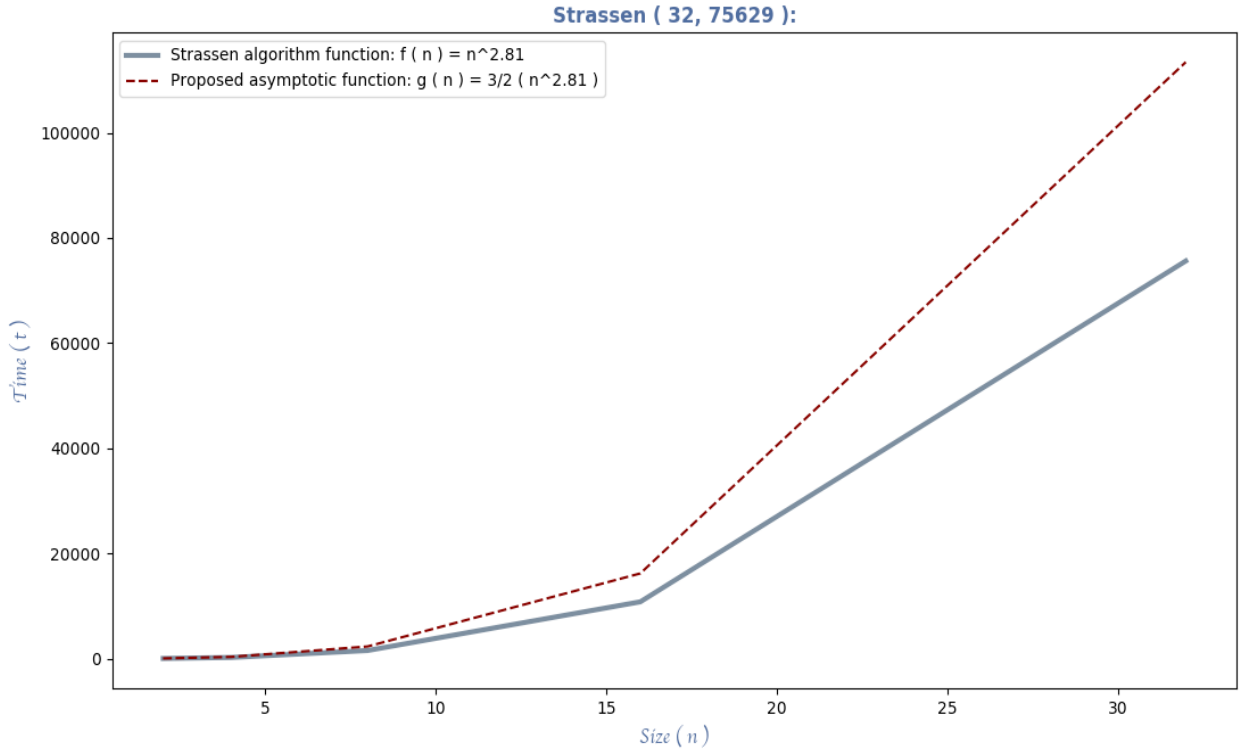

Figure 4.1.7: Graph for Figure 4.1.6.

Observation: *The red function it's the proposed one:* $g\ (\ n\ ) = \frac{3}{2} \cdot n^{2.81}$ *and the blue it's the Stranssen's Algorithm complexity:* $f\ (\ n\ ) = n^{2.81}$.

Size (2^n)	Time (t)
2	29
4	218
8	1541
16	10802
32	75629

Table 2: Plot point for Strassen's Algorithm complexity of Figure 4.1.7.

4.2 Strassen's vs IJK-Algorithm:

First versus of ***Strassen's Algorithm*** against ***IJK-Algorithm***. The program will plot the ***time*** that both algorithm's takes to make the product of matrices of size $n = 2^8$.

Observation: *Both matrices A and B are to big to put a screen-shot of the console output, same for the resulting matrix. In compensation we will put the algorithms parameters in the console output for plotting both complexities.*

```
Where: A, B, C ∈ M[ 256x256 ] ( Z+ )
Strassen's algorithm parameters: [(2, 29), (4, 218), (8, 1541), (16, 10802), (32, 75629), (64, 529418), (128, 3705941), (256, 25941602)]
IJK algorithm parameters: [(2, 33), (4, 173), (8, 1173), (16, 8741), (32, 67653), (64, 532613), (128, 4227333), (256, 33686021)]
```

Figure 4.2.0: Parameters of Strassen's and IJK algorithms for matrices of size n = 2^8.

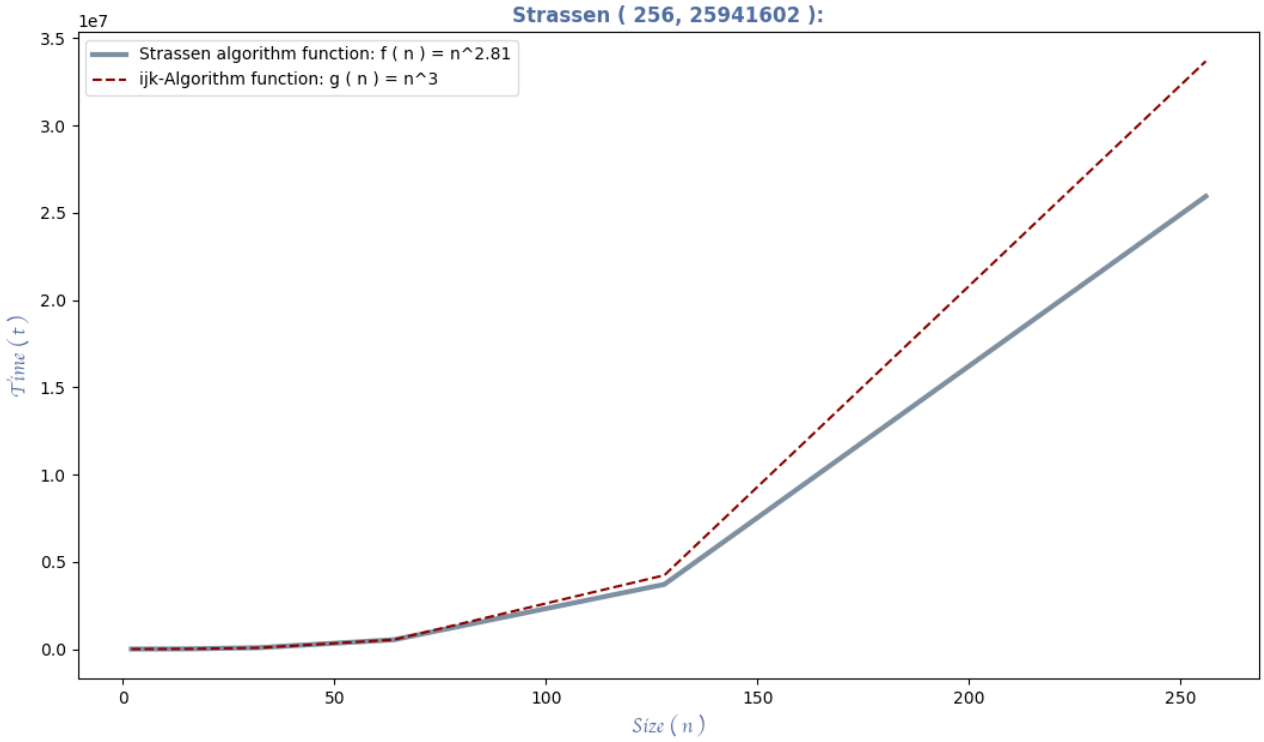

Figure 4.2.1: Graph for Figure 4.2.0.

Size (2^n)	Strassen's Algorithm Time (t)	IJK-Algorithm Time (t)
2	29	33
4	218	173
8	1541	1173
16	10802	8741
32	75629	67653
64	529418	532613
128	3705941	4227333
256	25941602	33686021

Table 3: Plot point for Strassen's and IJK Algorithms complexity of Figure 4.2.1.

Second versus of ***Strassen's Algorithm*** against ***IJK-Algorithm***. The program will plot the ***time*** that both algorithm's takes to make the product of matrices of size $n = 2^9$.

```
Where: A, B, C ∈ M[ 512x512 ] ( Z+ )
Strassen's algorithm parameters: [(2, 29), (4, 218), (8, 1541), (16, 10802), (32, 75629), (64, 529418), (128, 3705941), (256, 25941602), (512, 181591229)]
IJK algorithm parameters: [(2, 33), (4, 173), (8, 1173), (16, 8741), (32, 67653), (64, 532613), (128, 4227333), (256, 33686021), (512, 268960773)]
```

Figure 4.2.2: Parameters of Strassen's and IJK algorithms for matrices of size n = 2^9.

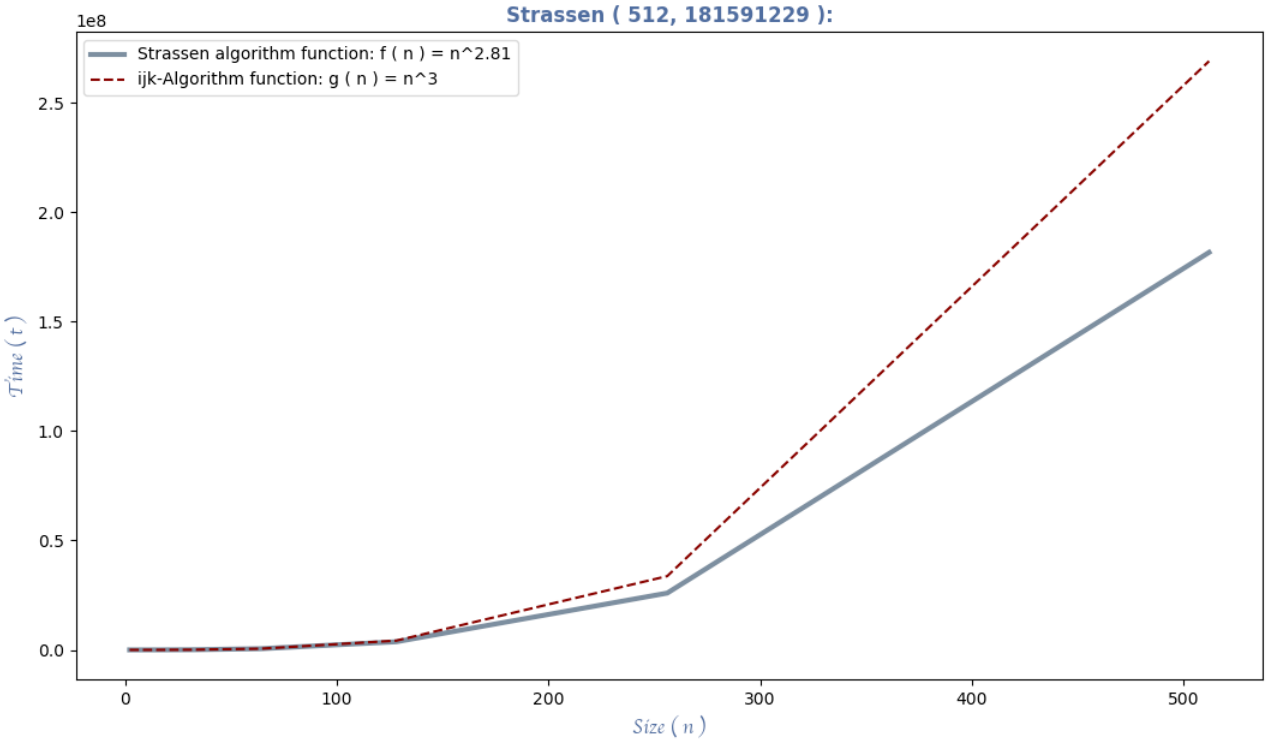

Figure 4.2.3: Graph for Figure 4.2.2.

Size (2^n)	Strassen's Algorithm Time (t)	IJK-Algorithm Time (t)
2	29	33
4	218	173
8	1541	1173
16	10802	8741
32	75629	67653
64	529418	532613
128	3705941	4227333
256	25941602	33686021
512	181591229	268960773

Table 4: Plot point for Strassen's and IJK Algorithms complexity of Figure 4.2.3.

Observation: *The red function it's the IJK-Algorithm complexity: $g\ (\ n\ ) = n^3$ and the blue it's the Stranssen's Algorithm complexity: $f\ (\ n\ ) = n^{2.81}$.*

5 Conclusion:

I never realize the importance of the ***divide-and-conquer*** paradigm, I realize that maybe I applied it many times, but never take conscience what I was really doing, an example it's the ***Binary-Search***, this algorithm was maybe the first that I programmed, but up to this point, I didn't know that uses this paradigm. Now, I'm quite intrigued what other applications will have ***divide-and-conquer***. The Strassen's algorithm was something very new for me, and it's very interesting how with this paradigm and recursion we can have a better algorithm than the usual ijk-algorithm.

- Hernandez Martinez Carlos David.

This time we could use the algorithms properties to demonstrate the complexity of some algorithms, we increase in complexity about the programming task, for instance, the programming level was a bit interesting. The new issue was the recursion of the algorithm, I think this time the algorithm was very useful than other many programs that we were analyzing.

- Burciaga Ornelas Rodrigo Andres.

6 Bibliography References:

[1] Baase and Van Gelder. "Computer Algorithms: Introduction to Design and Analysis". Addison-Wesley.

[2] Thomas H. Cormen. "Introduction to Algorithms". The MIT press.