DJILLALI LIABES UNIVERSITY OF SIDI BEL ABBES
FACULTY OF EXACT SCIENCES
DEPARTMENT OF COMPUTER SCIENCES

Module : Algorithmique et Complexité
1ST YEAR OF MASTER'S DEGEREE IN
NETWORKS,INFORMATION SYSTEMS & SECURITY(RSSI)
2021/2022

Comparaison entre 2 algorithmes pour la multiplication matricielle

---

Author:
HADJAZI M.Hisham
AMUER Wassim Malik
Group: 01/RSSI

Supervisor:
Dr. MME. BELKHODJA
ZENAIDI Lamia

A paper submitted in fulfilment of the requirements for the
TP-03

November 16, 2021

Contents

Chapter 1

Solutions of Fiche TP-03

Notes regarding this solution :
This solution and the executions of the code in it was done in the following machine :

- *Machine*: Lenovo Ideapad S210
- *CPU*: Intel Celeron 1037U 1800 MHz
- *RAM*: 8GB DDR3l
- *OS* : Linux Mint 20.2 Cinnamon Kernel v.5.4.0-88
- *IDE* : Eclipse IDE for Java Developers Version: 2019-12 (4.14.0)
- *Java version*: 11.0.11

Le but de ce TP est la Comparaison entre l'algorithme cubique classique et l'algorithme de Strassen pour la multiplication matricielle. Soit 3 matrice A, B, C de taille n*n :

Description de l'algorithme classique :

le calcul de C=A*B pour n=2 se fait selon la formule suivante :

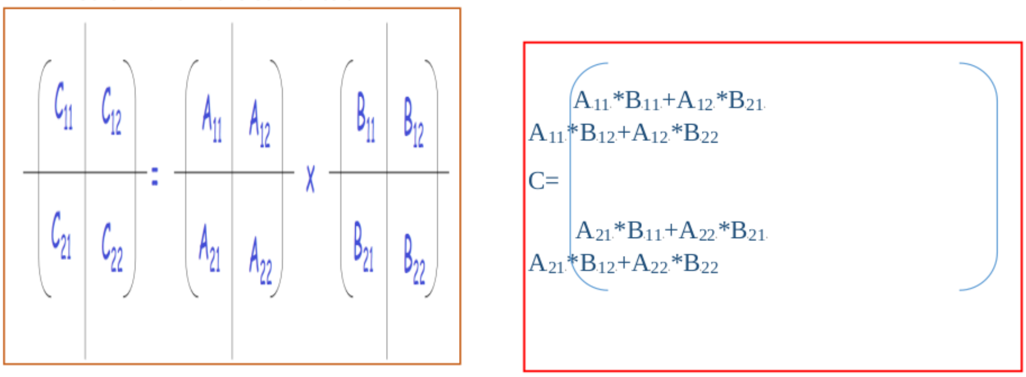

FIGURE 1.1: Classic Matrix multiplication

Description de l'algorithme de Strassen :

L'algorithme de Strassen est un algorithme de type « diviser pour régner » dont l'objectif est de minimiser le nombre de multiplications. Le produit de deux matrices 2×2 peut être effectué avec seulement 7 multiplications au lieu de 8 avec la méthode classique. Cet algorithme ne s'applique que sur les matrices dont la taille est une puissance de 2. L'algorithme de Strassen est récursif : à chaque étape la matrice est divisée en quatre sous matrices. Le cas d'arrêt de la récursivité est celui où les matrices sont de taille 1x1. Les calculs se font selon les formules suivantes :

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

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

$$C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

FIGURE 1.2: Strassen's Matrix multiplication

Ici les additions et soustractions sont des additions et soustractions de matrices. L'algorithme est le suivant :

```
int [][] Strassen(int [][] A, int [][] B, int n)
{
// n : nombre de ligne, de colonnes
// Si n n'est pas une puissance de 2 alors   on ajoute des lignes et des colonnes de
0 afin d'accéder à la puissance de 2 la plus proche supérieurement
if ( n==0)   C[0] [0]=A[0][0]*B[0][0];
 else
 {
Décomposer chacune des matrices A et B en 4 sous matrices de taille n/2 × n/2 ;
M1=Strassen(A11+A22,B11 + B22,n/2);
M2=Strassen(A21 + A22,B11 ,n/2);
M3=Strassen(A11 ,B12- B22,n/2);
M4=Strassen(A22,B21-B11,n/2);
M5=Strassen(A11+A12,B22,N/2);
M6=Strassen(A21-A11,B11+B12,n/2);
M7=Strassen(A12 - A22,B21+B22,n/2);
C11=M1 + M4 – M5 + M7;
C12=M3 + M5;
C21=M2+ M4;
C22=M1 + M3 –M2 +M6;
Composer la matrice C à partir de C11, C12, C21, C22;
Return C;
 }
```

FIGURE 1.3: Strassen's Algorithm

1.1 1. Lire attentivement puis compléter le code java de la classe StrassenMult, en tenant compte de toutes les indications qui y sont données.

The Source code can viewed either in **AppendixA** or with the included **StrassenMult.java**.

1.2 Afficher le nombre de multiplications exécutées par chaque algorithme.

In order to calculate the number of multiplications we need to add a counter in right place.

For the classical method we put inside the 3rd loop.

```
        for (int i = 0; i < aRows; i++) {
            for (int j = 0; j < bColumns; j++) {
                for (int k = 0; k < aColumns; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                    cpt1++;
                }
            }
        }
```

While in Strassen's method we put it inside after the condition of the smallest decomposition where the size of the matrix is **n=1**.

```
        if (n == 1) {
            resultat[0][0] = A[0][0] * B[0][0];
            cpt2++;
        } else {
```

1.2.1 When the matrix size is multiple of 2

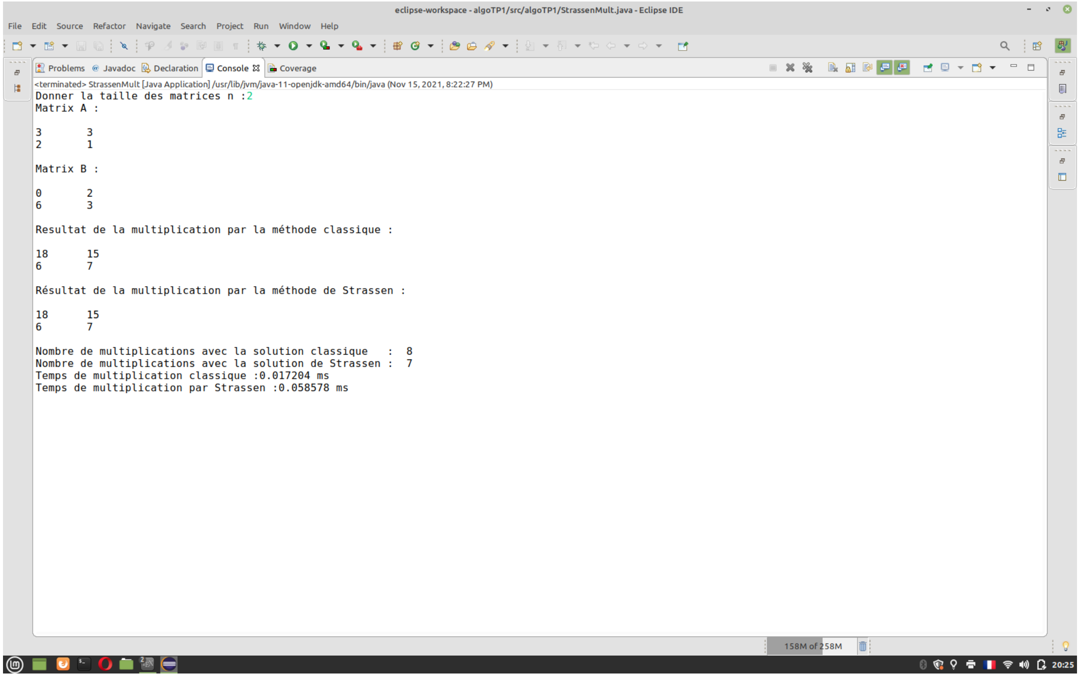

FIGURE 1.4: Excution of n=2

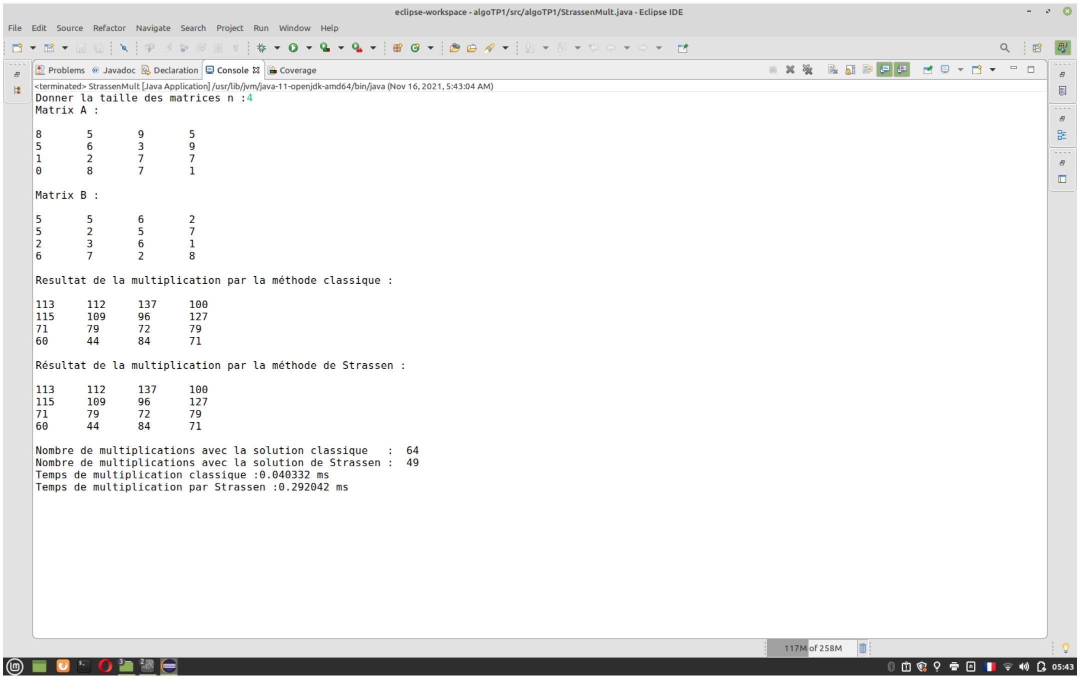

FIGURE 1.5: Excution of n=4

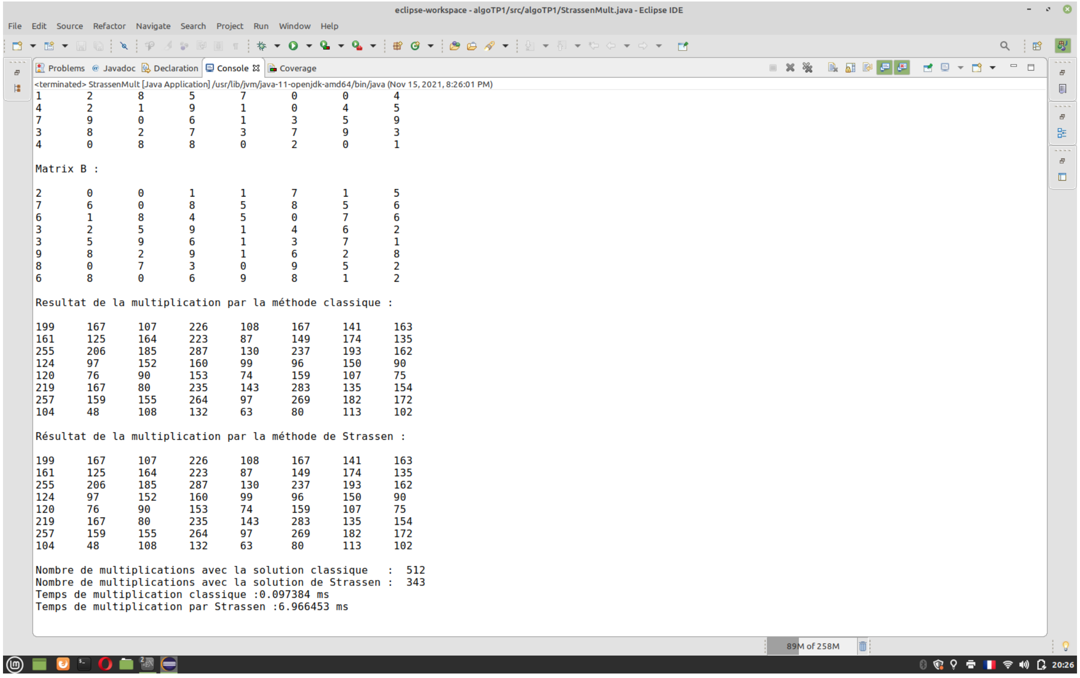

FIGURE 1.6: Excution of n=8

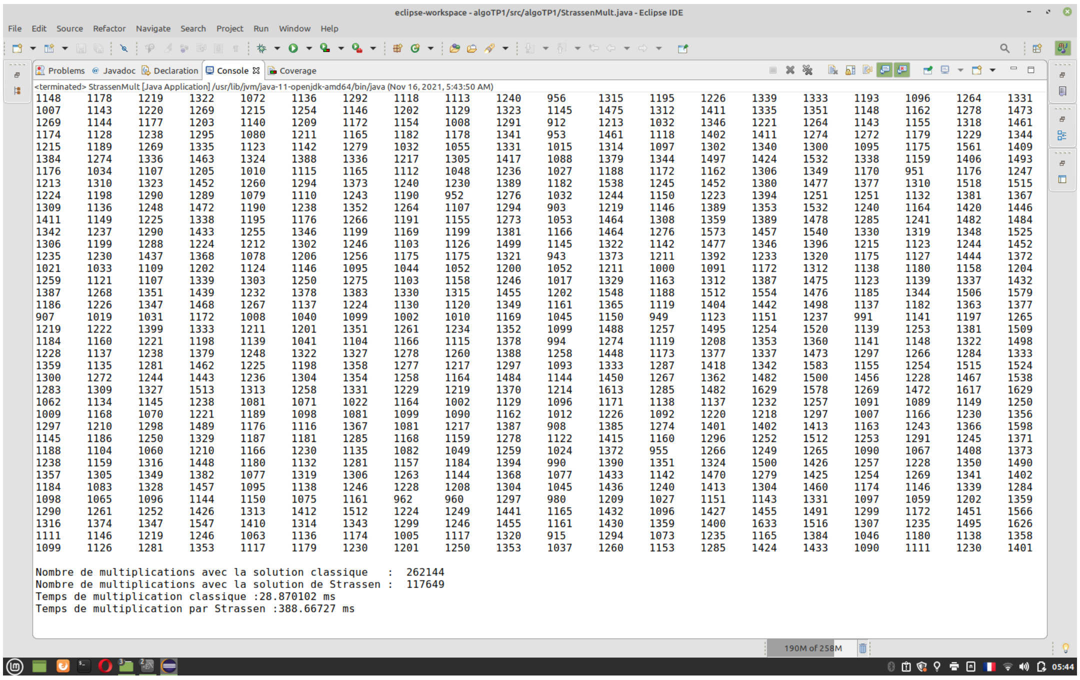

FIGURE 1.7: Excution of n=64

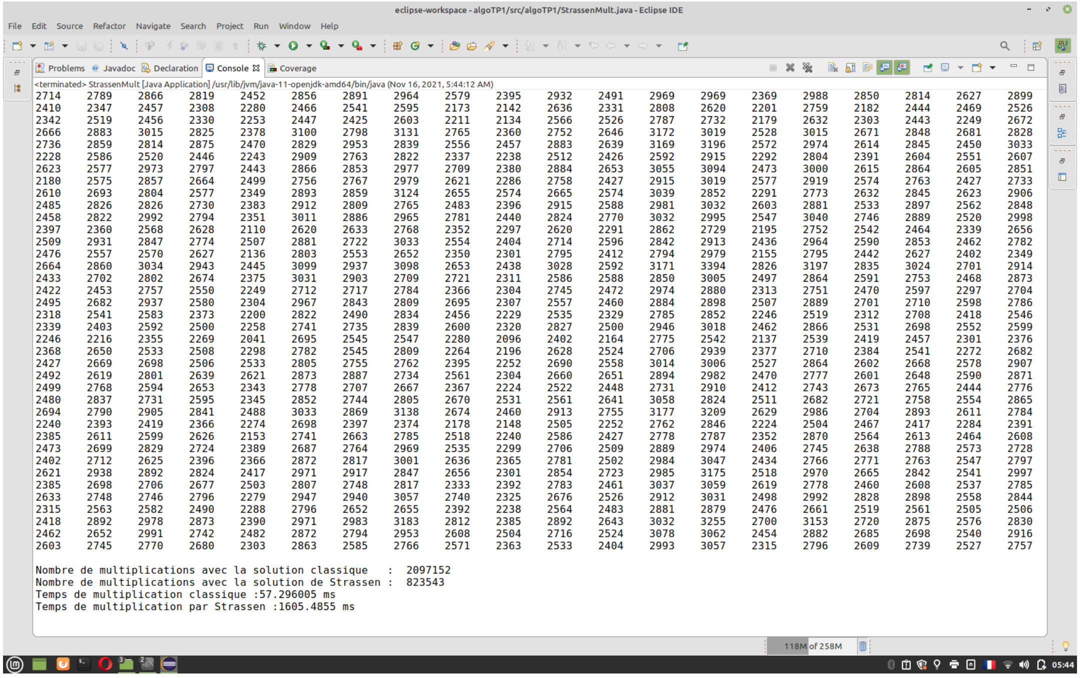

FIGURE 1.8: Excution of n=128

As we can see when the size of the matrix is of size 2^k where **k is multiple of 2** powers of 2 : $2^k, 2^k + 1, 2^k + 2$. the Strassen's performs at its best by reducing the number of multiplications dramaticly specially as the size increases as in **n=128** we see there are **823543** multiplication operations only in Strassen's algorithm while the Classical method requires a staggering **2097152** multiplication operations.

1.2.2 When the matrix size is EVEN

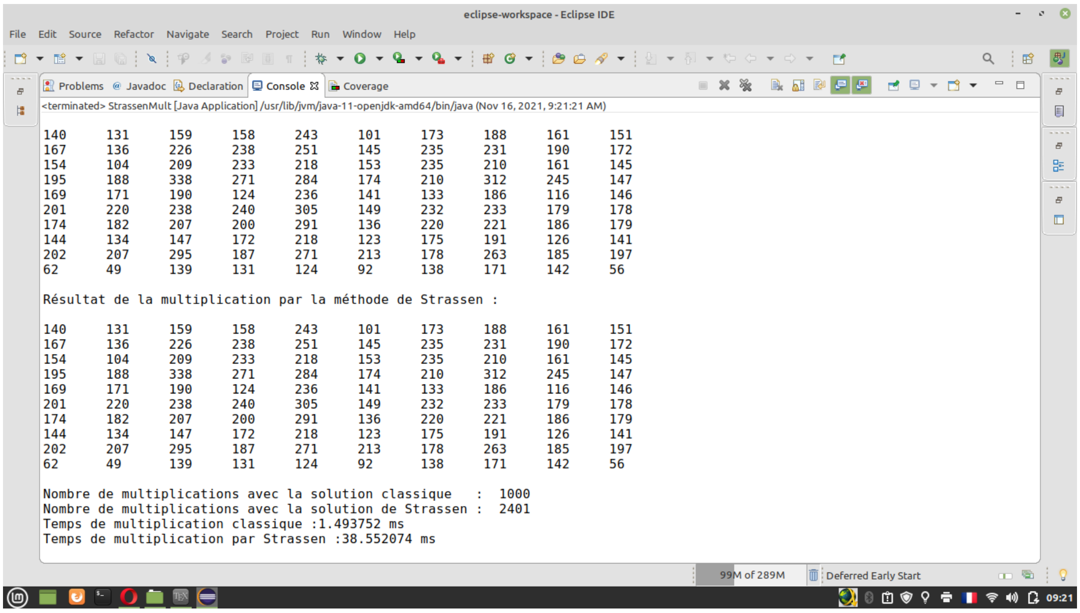

FIGURE 1.9: Excution of n=10

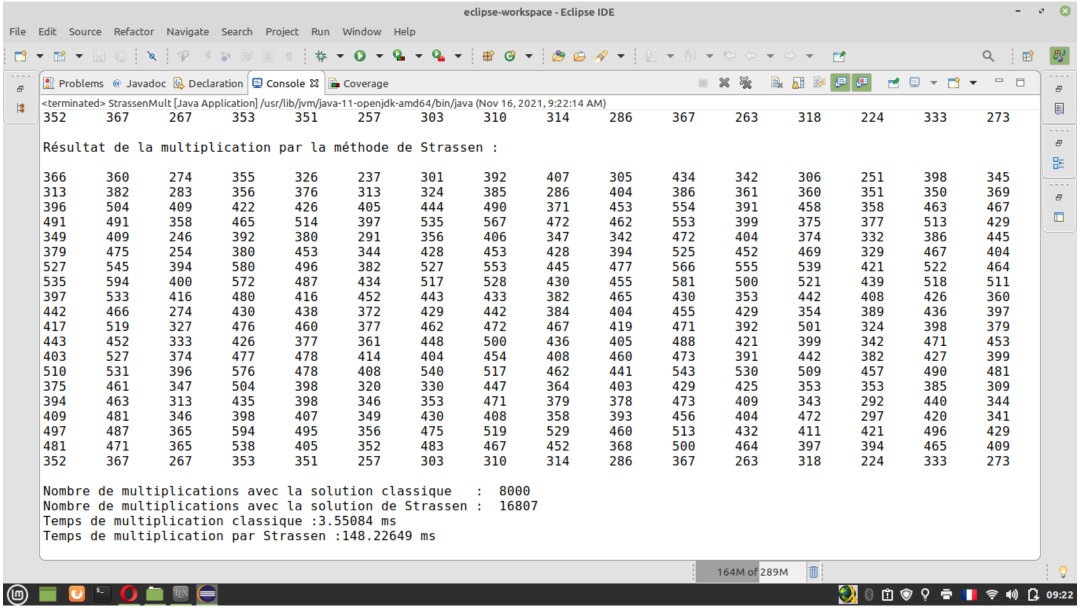

FIGURE 1.10: Excution of n=20

In even numbers but not powers of 2 Strassen's Algorithms loses its efficiency and the number of multiplications increase and becomes worse than the classical method as we can see in **n=20** the total number of multiplications in Strassen's method is **16807** where the classical method takes only **8000** multiplications which is half, so the loss is large. as the Strassen's method requires twice the number of multiplications not forgetting the large constant of Strassen's algorithm that consists of additions and subtractions as we will see later on.

1.2.3 When the matrix size is ODD

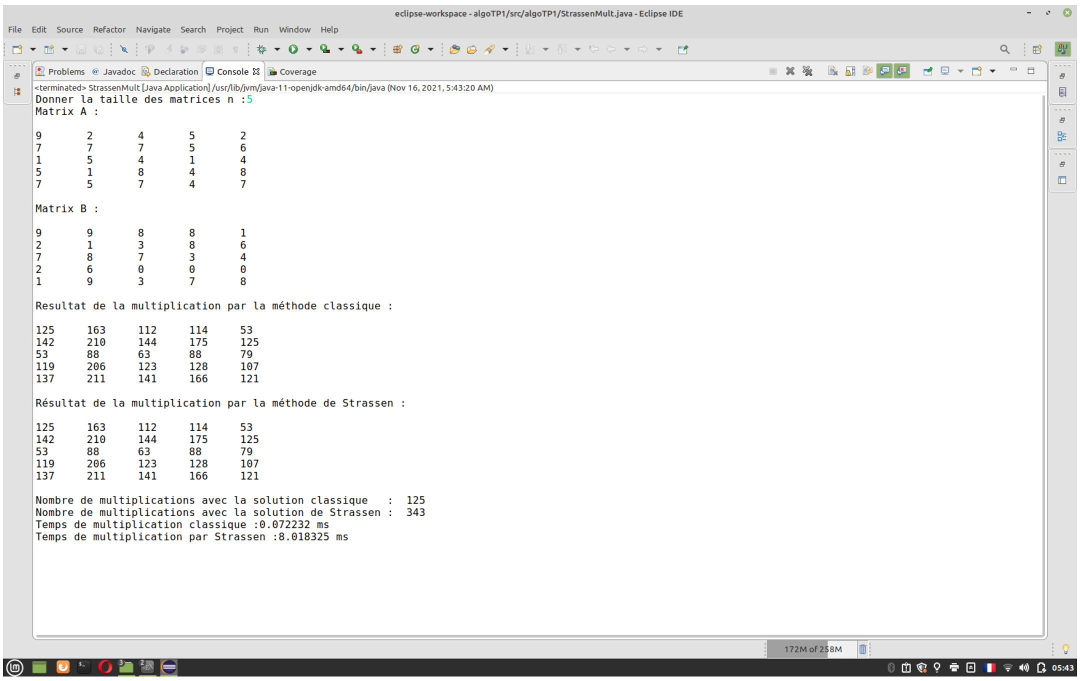

FIGURE 1.11: Excution of n=5

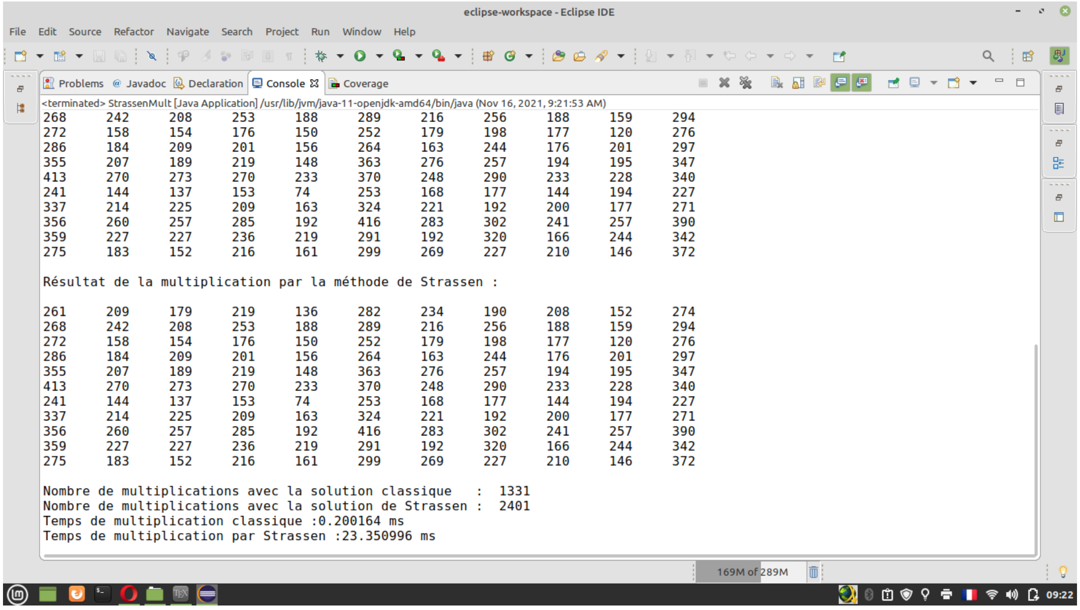

FIGURE 1.12: Excution of n=11

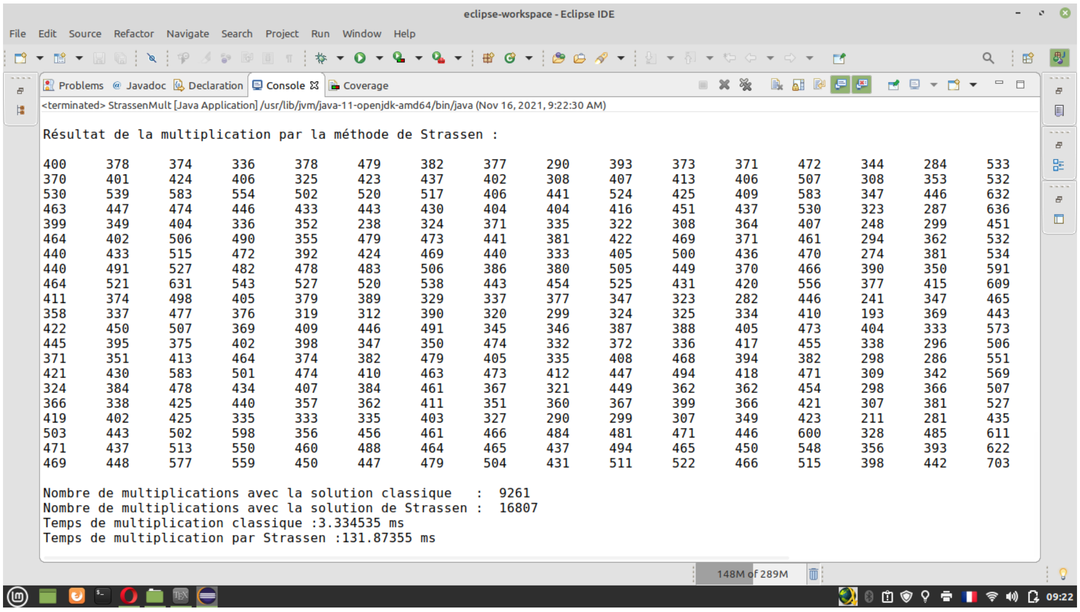

FIGURE 1.13: Excution of n=21

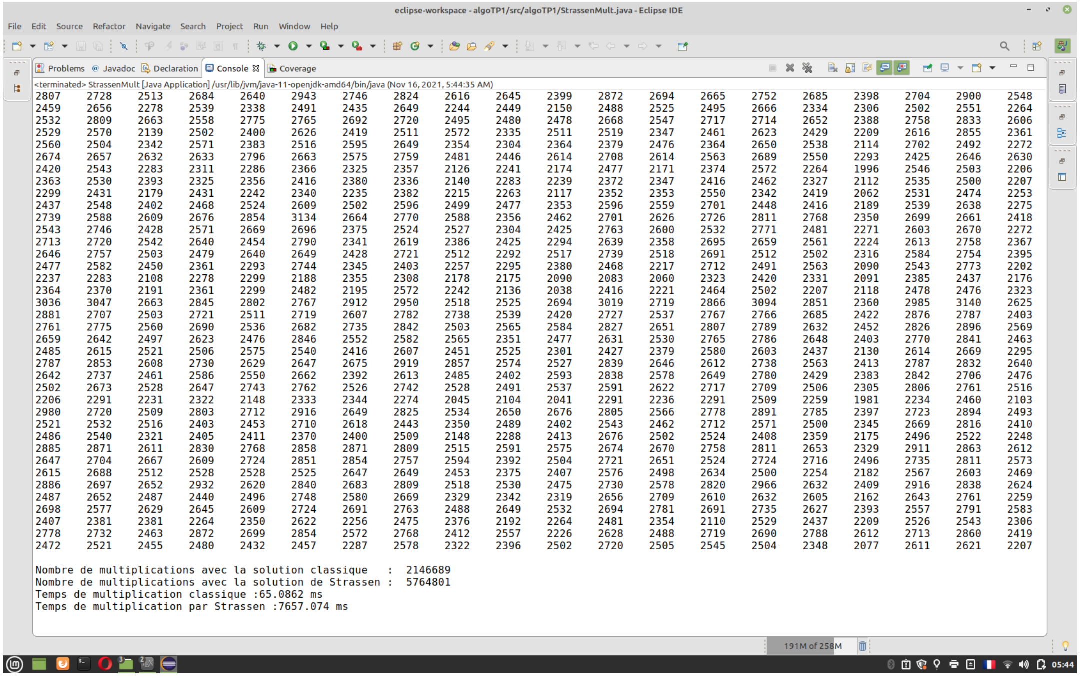

FIGURE 1.14: Excution of n=129

As we can see when size of the multiplications increases when size of the matrix is odd, and it is as bad as pairs for example when **n=21** we see the number of multiplications in the classical method are **9261** multiplications while in the Strassen's method it is much higher at **16807** which is twice as much as the classical method. when the size increases it shows even more for example when **n=129** the classic method does **2146689** multiplications while the Strassen's method requires **5764801** multiplications again over twice the number.

In fact if we have a machine where we can test for very large numbers we will see a huge difference that keeps incrementing.

1.3 Afficher le temps d'exécution pour chaque algorithme.

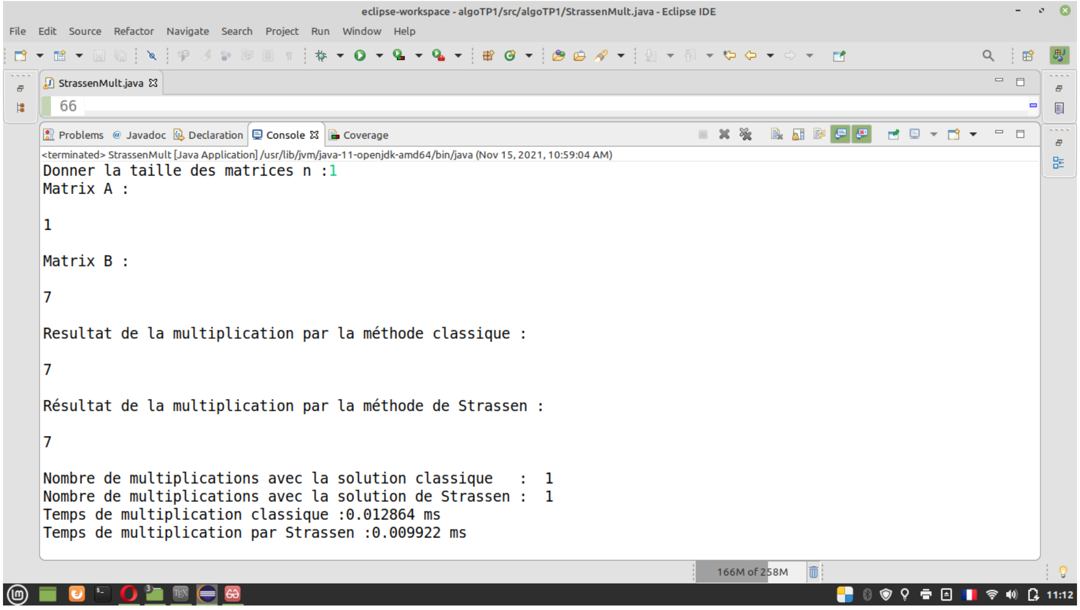

FIGURE 1.15: Excution of n=1

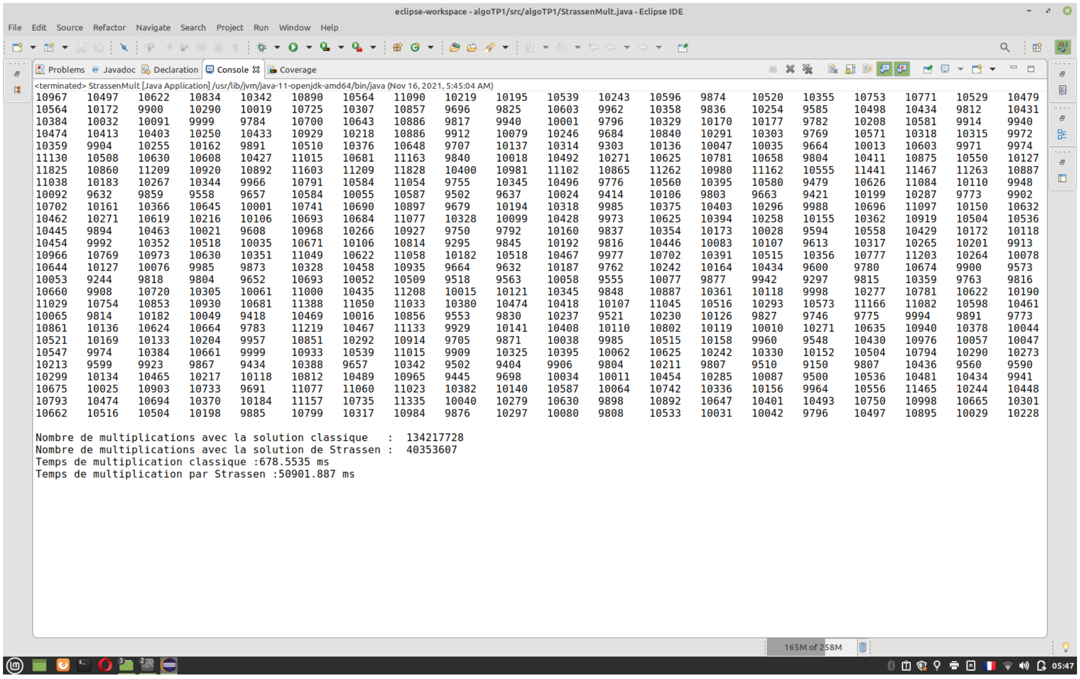

FIGURE 1.16: Excution of n=512

Considering we will use it only on matrices of size of multiples of 2 the Strassen's method still requires much higher time than classic algorithm. for many reasons. firs Strassen's method has a large constant of addition of subtraction of 20 operations per a single iteration which adds up quickly as the matrix gets bigger, and the benefits will be only in really large matrices. as seen when size of **n=512** the classic method requires **678 ms** while the Strassen's method requires **50901 ms** which is a lot of time.

Another reason is that Strassen's method is not optimized for small sizes so a better way would be **IF n<1024 use classic() else use Strassen()**. like this we would use Strassen's method at its optimum sizes.

Large integer multiplication is a notorious example of this:

- *Classic Multiplication*: ON^2 optimal for $<$ 100 digits
- *Karatsuba Multiplication*: $O(N^{1.585})$ faster than above at 100 digits
- *Toom-Cook 3-way*: $O(N^{1.465})$ faster than Karatsuba at 3000 digits
- *Floating-point FFT* : $O(> Nlog(N))$ faster than Karatsuba/Toom-3 at 700 digits
- *Schönhage–Strassen algorithm (SSA* : $O(Nlog(n)loglog(n))$ faster than FFT at a billion digits
- *Fixed-width Number-Theoretic Transform*: $O(Nlog(n))$ faster than SSA at a few billion digits

Input : $Matrices\ A, B$

Phase 1

$T_1 = A_{11} + A_{22}$ $\quad T_6 = B_{11} + B_{22}$
$T_2 = A_{21} + A_{22}$ $\quad T_7 = B_{12} - B_{22}$
$T_3 = A_{11} + A_{12}$ $\quad T_8 = B_{21} - B_{11}$
$T_4 = A_{21} - A_{11}$ $\quad T_9 = B_{11} + B_{12}$
$T_5 = A_{12} - A_{22}$ $\quad T_{10} = B_{21} + B_{22}$

Phase 2

$Q_1 = T_1 * T_6$ $\quad Q_5 = T_3 * B_{22}$
$Q_2 = T_2 * B_{11}$ $\quad Q_6 = T_4 * T_9$
$Q_3 = A_{11} * T_7$ $\quad Q_7 = T_5 * T_{10}$
$Q_4 = A_{22} * T_8$

Phase 3

$U_1 = Q_1 + Q_4$ $\quad U_2 = Q_5 - Q_7$
$U_3 = Q_3 + Q_1$ $\quad U_4 = Q_2 - Q_6$
$C_{11} = U_1 - U_2$ $\quad C_{12} = Q_3 + Q_5$
$C_{21} = Q_2 + Q_4$ $\quad C_{22} = U_3 - U_4$

Output : $C = (C_{ij})$

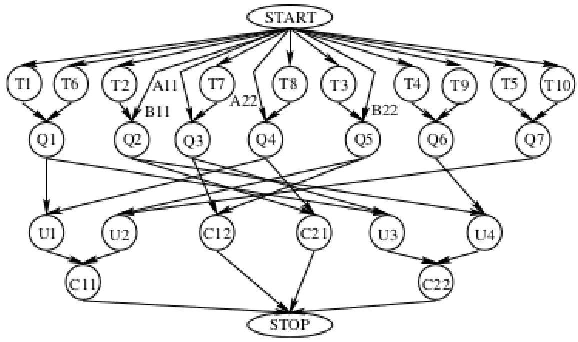

FIGURE 1.17: First level of recursion of the Strassen algorithm and its MDG representation.

One last caveat specific to Strassen's Algorithm is that in practice, the $\theta(n^2)$ term requires $20 \cdot n^2$ operations, which is quite a large constant to hide. If our data is large enough that it must be distributed across machines in order to store it all, then really we can often only afford to pass through the entire data set one time. If each matrix-multiply requires twenty passes through the data, we're in big trouble. Big θ notation is great to get you started, and tells us to throw away egregiously inefficient algorithms. But once we get down to comparing two reasonable algorithms, we often have to look at the algorithms more closely.[3]

If we're actually in the PRAM model, i.e. we have a shared memory cluster, then Strassen's algorithm tends to be advantageous only if $n \geq 1,000$, assuming no communication costs. Higher communication costs drive up the n at which

Strassen's becomes useful very quickly. Even at $n = 1,000$, naive matrix-multiply requires $1e9$ operations; we can't really do much more than this with a single processor. Strassen's is mainly interesting as a theoretical idea. For more on Strassen in distributed models[3]

1.4 Faire plusieurs exécutions en modifiant la taille des matrices, en prenant des tailles avec des valeurs paires et impaires. Vérifier que les matrices C obtenues par les deux algorithmes sont les mêmes.

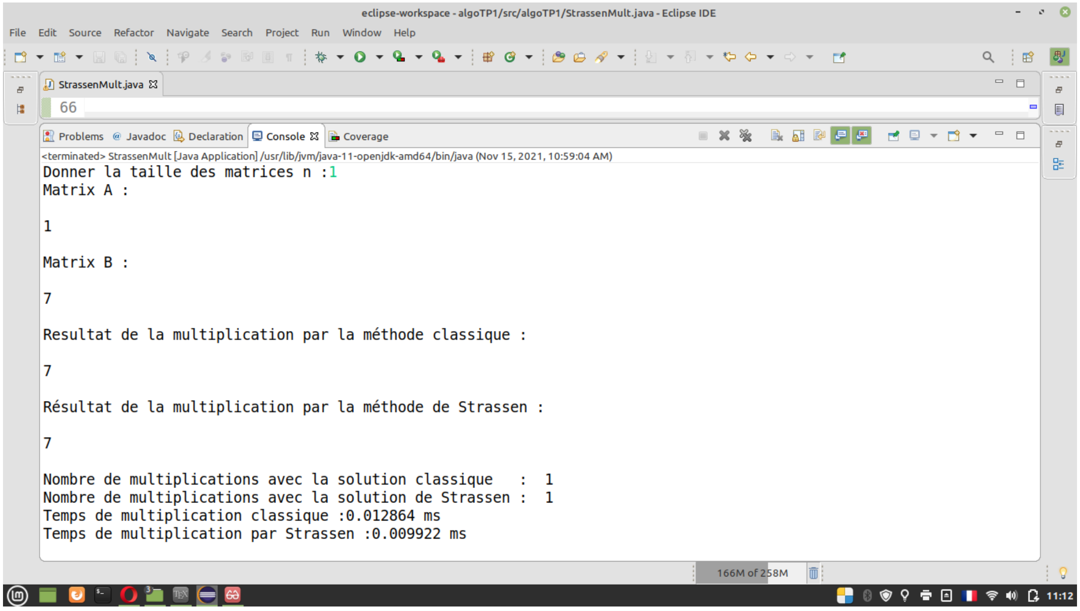

FIGURE 1.18: Excution of n=1

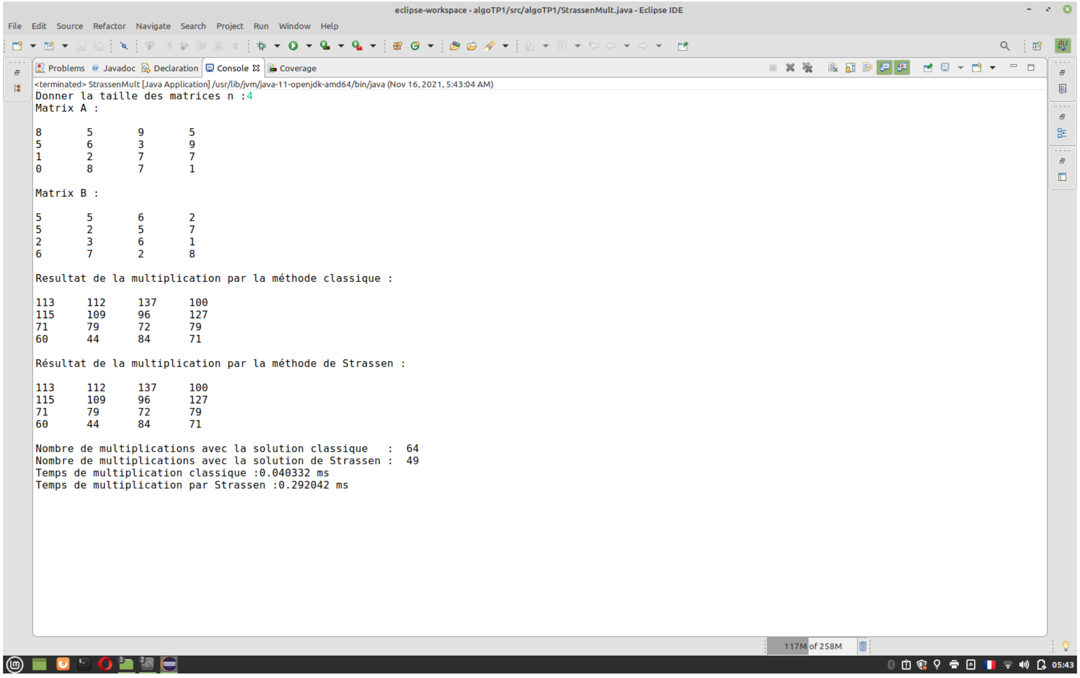

FIGURE 1.19: Excution of n=4

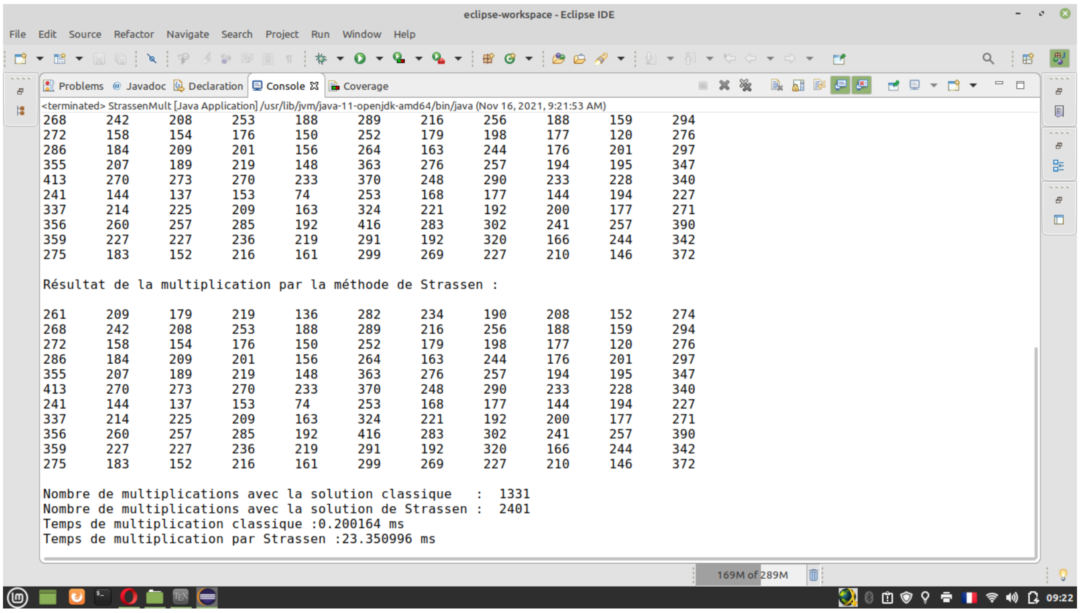

FIGURE 1.20: Excution of n=11

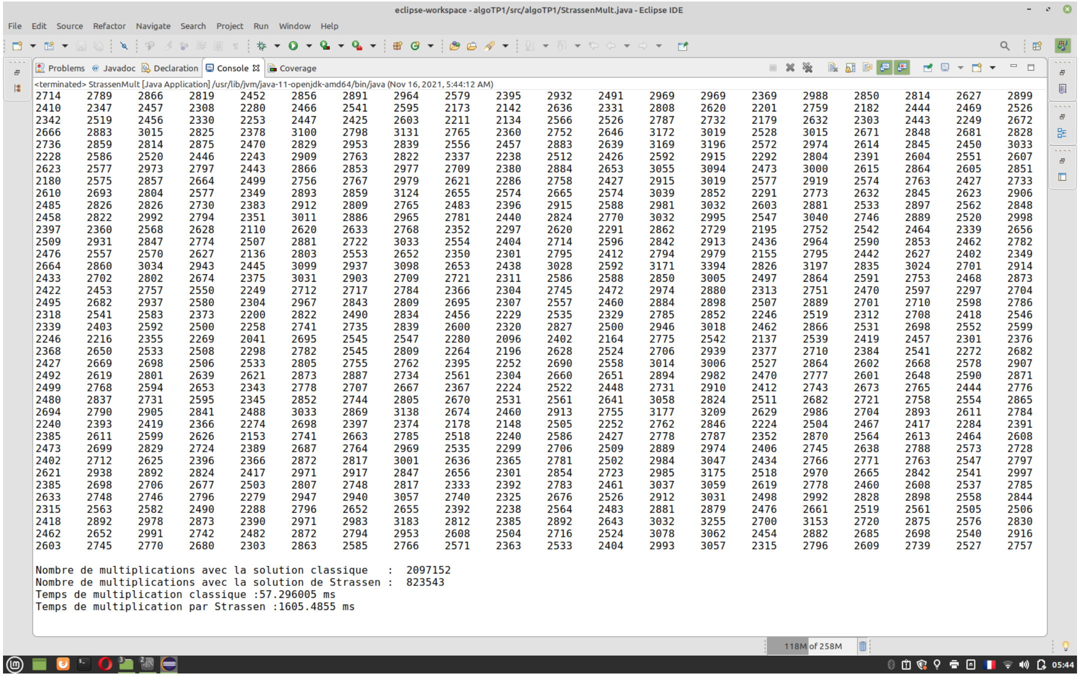

FIGURE 1.21: Excution of n=128

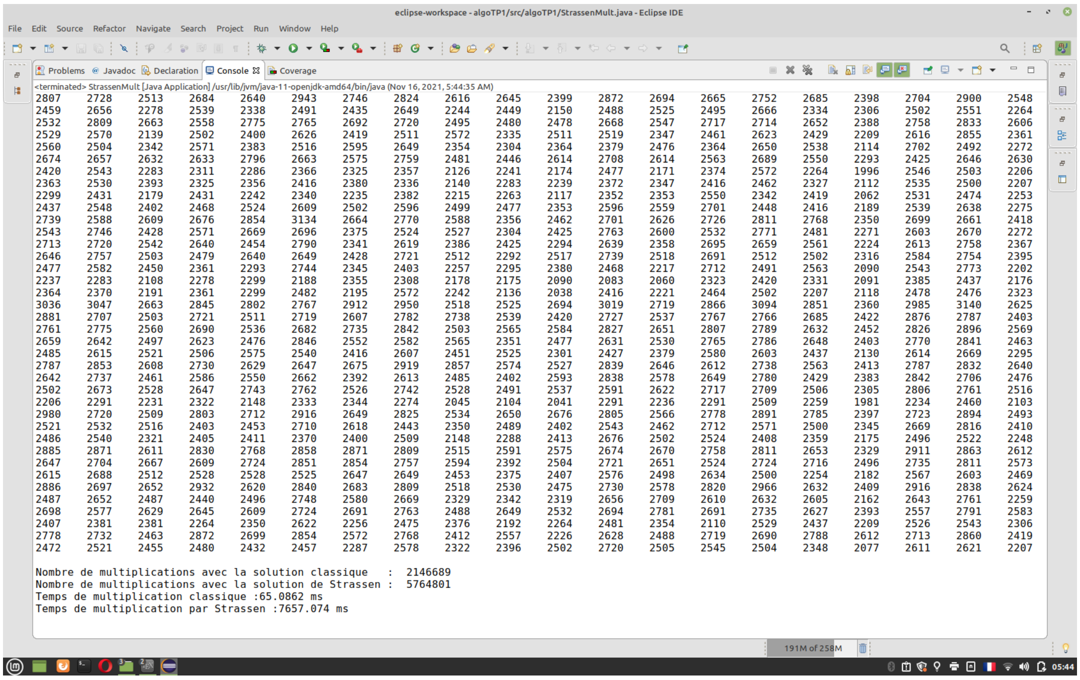

FIGURE 1.22: Excution of n=129

1.5 Faire des captures d'écran pour n=2 ; n=7 ; n=8.

1.5.1 n=2

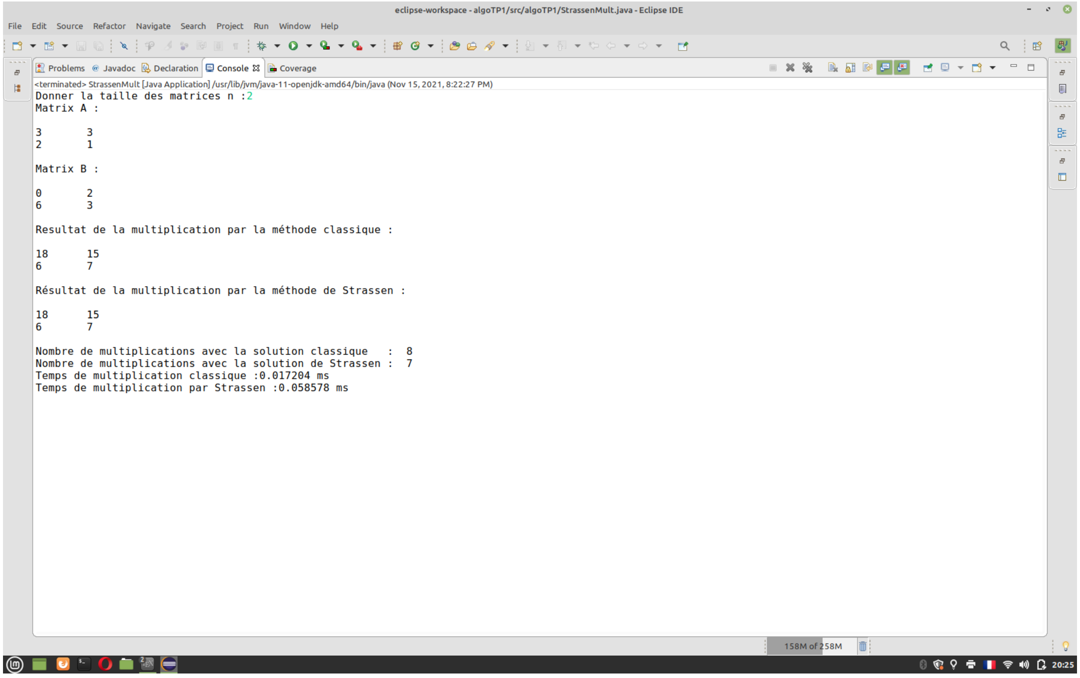

FIGURE 1.23: Excution of n=2

1.5.2 n=7

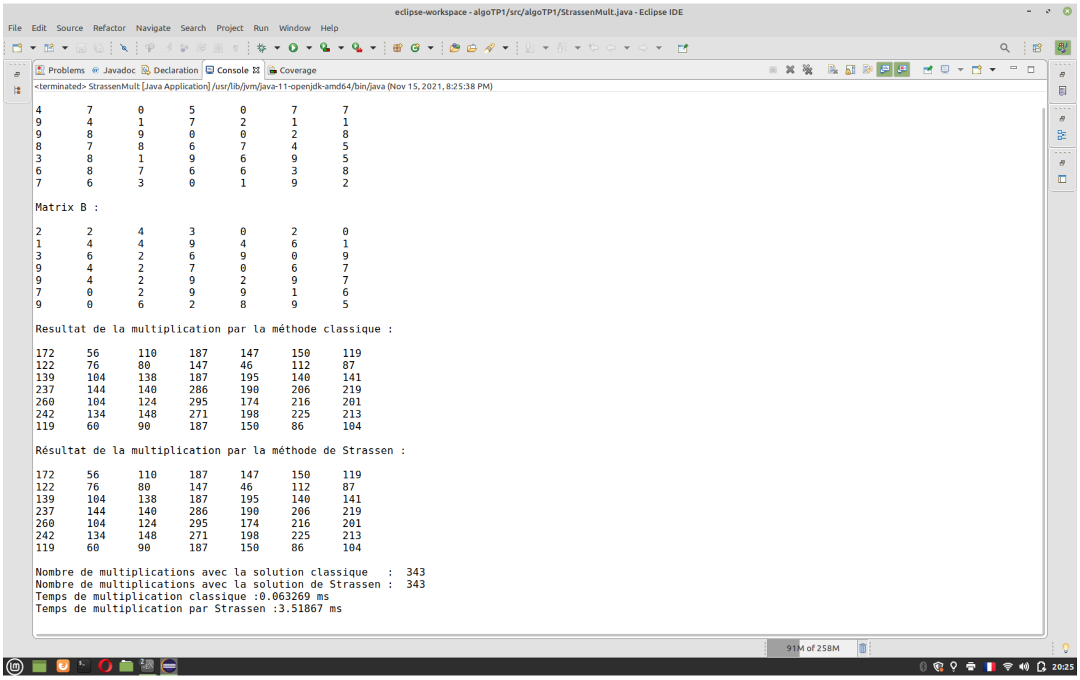

FIGURE 1.24: Excution of n=7

1.5.3 n=8

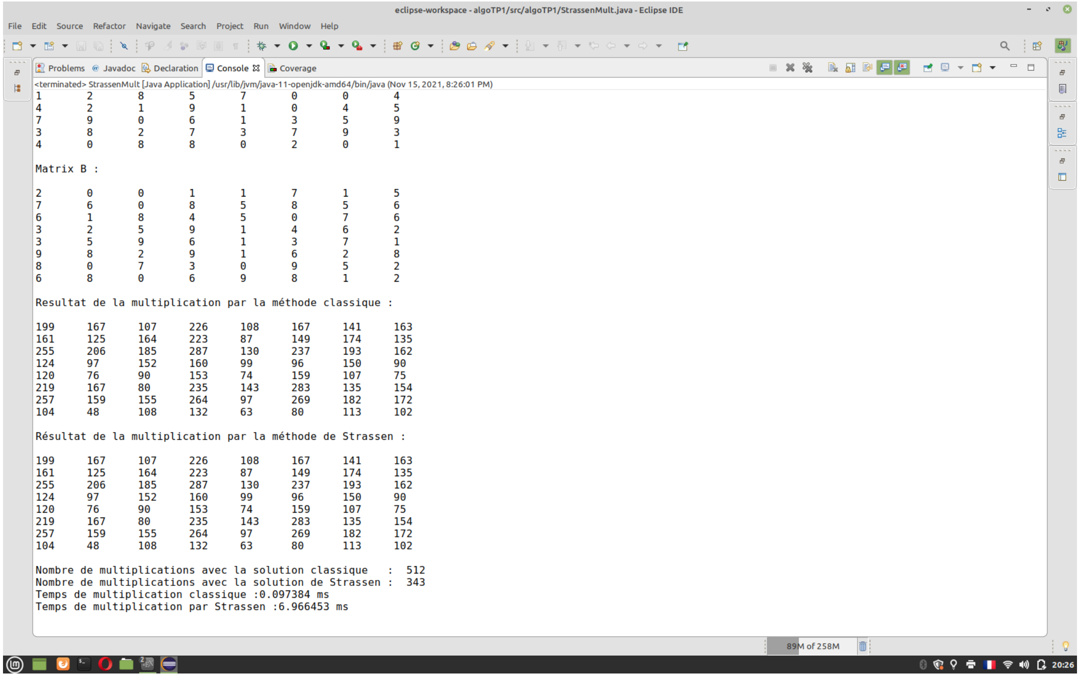

FIGURE 1.25: Excution of n=8

1.6 Remplir le tableau suivant :

n		Algorithme classique	Algorithme de Strassen
2	Nombre de multiplications	8	7
	Temps d'exécution	0.018081 ms	0.063568 ms
7	Nombre de multiplications	343	343
	Temps d'exécution	0.07704 ms	2.416576 ms
8	Nombre de multiplications	512	343
	Temps d'exécution	0.093842 ms	4.719538 ms
50	Nombre de multiplications	125000	117649
	Temps d'exécution	11.445735 ms	637.6434 ms
64	Nombre de multiplications	262144	117649
	Temps d'exécution	31.544764 ms	629.3083 ms
	Complexité temporelle	$T(n) = \theta(n^3)$	$T(n) = \theta(n^{2.8074})$

1.6.1 Les calculs des complexités temporelles doivent être justifiés, pour les deux algorithmes. Pour l'algorithme de Strassen, il faut donner l'équation de récurrence et la résoudre avec une méthode de votre choix.

Time Complexity of Classic Matrix Multiplication

```
for (int i = 0; i < aRows; i++) { --------------- n
    for (int j = 0; j < bColumns; j++) { -------- n*n
        for (int k = 0; k < aColumns; k++) { ---- n*n*n
            C[i][j] += A[i][k] * B[k][j]; ------- n*n*n+1
        }
    }
}
```

Therfore Time Complecity = $2n^3 + n^2 + n + 1 \simeq \Theta(n^3)$

Time Complexity of Strassen's Algorithm

Solving using Recurrence

Lets consider our base case as when **n = power of 2**:

$$\begin{cases} T(n) = 7T(\frac{n}{2}) & n > 1 \\ T(1) = 1 \end{cases}$$

$$\begin{aligned} T(2) &= 7T(\tfrac{2}{2}) = 7T(1) = 7 \\ T(4) &= 7T(\tfrac{4}{2}) = 7T(1) = 7^2 \\ T(8) &= 7T(\tfrac{8}{2}) = 7T(1) = 7^3 \\ T(16) &= 7T(\tfrac{16}{2}) = 7T(1) = 7^4 \end{aligned}$$

Therefore we reach :

$$T(n) = 7^{logn}$$

To prove that this is correct :

$$T(1) = 7^{log1} = 7^0 = 1$$

Assume, for an arbitrary **n > 0** and **n a power of 2**, that :

$$T(2n) = 7^{log2n}$$

Finally, because :

$$7^{logn} = n^{log7}$$

this recurrence is usually given as :

$$T(n) = n^{log7} \simeq n^{2.81}$$

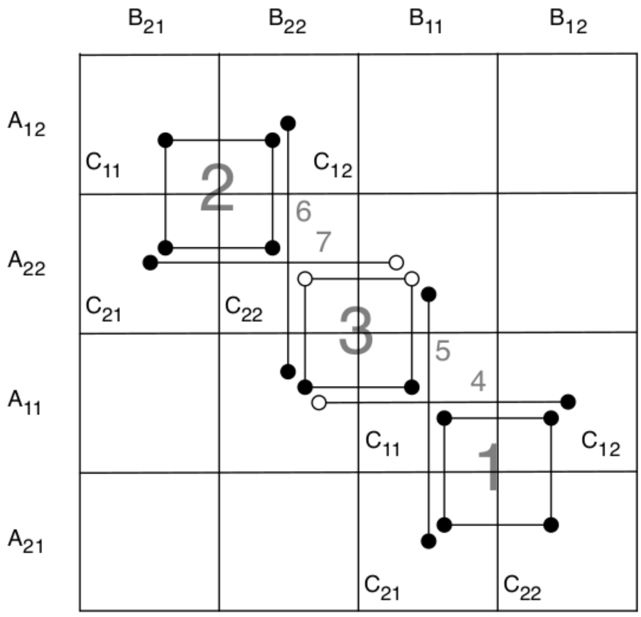

FIGURE 1.26: Strassen's algorithm

We now turn toward Strassen's algorithm, such that we will be able to reduce the number of sub-calls to matrix multiplies to 7, using just a bit of algebra. In this way, we bring the work down to $O(n^{log_2^7})$. How do we do this? We use the following factoring scheme. We write down C_{ij}'s in terms of block matrices M_k's. Each M k may be calculated simply from products and sums of sub-blocks of A and B.[7]

That is, we let

$M1 = (A11 + A22)(B11 + B22)$
$M2 = (A21 + A22)B11$
$M3 = A11(B12 - B22)$
$M4 = A22(B21 - B11)$
$M5 = (A11 + A12)B22$
$M6 = (A21 - A11)(B11 + B12)$
$M7 = (A2 - A22)(B21 + B22)$

Crucially, each of the above factors can be evaluated using exactly one matrix multiplication. And yet, since each of the M_k's expands by the distributive property of matrix multiplication, they capture additional information. Also important, is that these matrices M_k may be computed independently of one another, i.e. this is where the parallelization of our algorithm occurs.[7]

It can be verified that

$C11 = M1 + M4 - M5 + M7$
$C12 = M3 + M5$
$C21 = M2 + M4$
$C22 = M1 - M2 + M3 + M6$

Realize that our algorithm requires quite a few summations, however, this number is a constant independent of the size of our matrix multiples. Hence, the work is given by a recurrence of the form

$$T(n) = 7T(\frac{n}{2}) + O(n^2) \Rightarrow T(n) = O(n^{log_2^7})$$

and if we apply **Masters Theorem** we find that :

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

1.7 Que peut –on conclure ?

	Algorithme classique	Algorithme de Strassen
Faster at Size of matrix	**n < 1024**	**n > 1024**
Works on	**All sized Matrices**	**Squared Matrices only**
Number of Multiplications	**8**	**7**
Number of Additions	**4**	**18**
Odd Matrices	**same size**	**additional 0's row or colon**
Complexité temporelle	$T(n) = \theta(n^3)$	$T(n) = \theta(n^{2.8074})$

In conclusion according to theoretical analysis we see that Strassen's algorithm is a little bit faster than the classical 3 loop matrix multiplication method. However in practice at least with our way of implementing Strassen's algorithm and the size of our matrices we see the opposite.

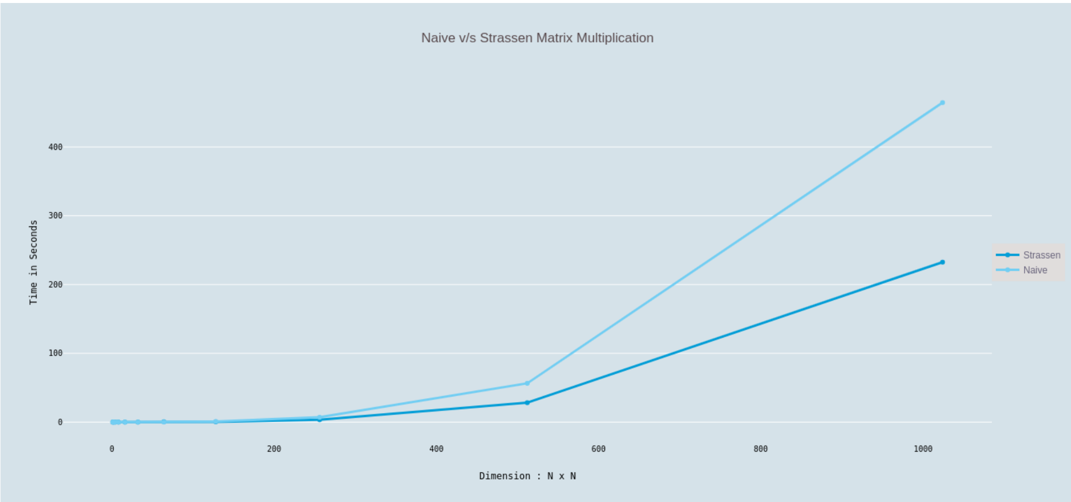

FIGURE 1.27: Strassen's algorithm vs Classic Method

However If implemented with a little modifications we can get better results, For example if we change the minimum size of our base matrix from **n=1** to **n<1024** where we switch to the classic method at that size we get better results, it is also noted that this algorithm is good with matrices of sizes **n = power of 2** while it performs poorly on **ODD** or **EVEN** matrices.

Therefore the Time complexity of Strassen's algorithm of $T(n) = \theta(n^{2.8074})$ is only good at really large matrices and preferably powers of 2.

Appendix A

Appendix A

A.1 Java Code

```java
//TP3 Algorithmique et Complexite 2021-2022
//LIRE ET BIEN COMPRENDRE LE CODE AVANT DE LE COMPLETER
//Respecter la demarche a suivre donnee en commentaires

//Nom:HADJAZI
//Prenom: Mohammed Hisham
//Specialite:   RSSI      Groupe: 01

//Nom:Ameur
//Prenom: Wassim Malik
//Specialite:   RSSI      Groupe: 01

import java.util.*;

public class StrassenMult {

    static long cpt1 = 0;
    static long cpt2 = 0;

public static int [][]  multiplication(int[][] A, int[][] B)
 {
    int aRows = A.length;
    int aColumns = A[0].length;
    int bRows = B.length;
    int bColumns = B[0].length;

    if (aColumns != bRows) {
        throw new IllegalArgumentException("A:Rows: " + aColumns
                + " did not match B:Columns " + bRows + ".");
    }

    int[][] C = new int[aRows][bColumns];
    for (int i = 0; i < aRows; i++) {
        for (int j = 0; j < bColumns; j++) {
            C[i][j] = 0;
        }
    }

    for (int i = 0; i < aRows; i++) {
        for (int j = 0; j < bColumns; j++) {
            for (int k = 0; k < aColumns; k++) {
                C[i][j] += A[i][k] * B[k][j];
```

```
                    cpt1++;
                }
            }
        }

        return C;

}

 public static int [][] strassen(int [][] A, int [][] B)
     {
         cpt2++;
         int n = A.length;
         int [][] resultat = new int[n][n];

         if((n%2 != 0 ) && (n !=1))
         {
     /*Ajouter une ligne de 0 et une colonne de 0 pour A et B :
     definir 3 nouvelles matrices temporaires A1,B1,C1
     de taille n+1,  ensuite A1=A et B1=B         */
             int[][] A1, B1, C1;
             int n1 = n+1;
             A1 = new int[n1][n1];
             B1 = new int[n1][n1];

             for(int i=0;i<n;i++)
             for(int j=0;j<n;j++)
             A1[i][j]=A[i][j];
             for(int i=0;i<n;i++)
             for(int j=0;j<n;j++)
             B1[i][j]=B[i][j];


               C1 = strassen(A1, B1);
             for(int i=0; i<n; i++)
                 for(int j=0; j<n; j++)
                     resultat[i][j] =C1[i][j];
             return resultat;
         }


         if(n == 1)
         {
             resultat[0][0] = A[0][0] * B[0][0];
         }
         else
         {

//Creation de 4 sous matrices A11,A12,A21,A22 (n/2 x n/2)
          int [][] A11 = new int[n/2][n/2];
          int [][] A12 = new int[n/2][n/2];
          int [][] A21 = new int[n/2][n/2];
          int [][] A22 = new int[n/2][n/2];



//Creation de 4 sous matrices B11,B12,B21,B22 (n/2 x n/2)
```

```
        int [][] B11 = new int[n/2][n/2];
        int [][] B12 = new int[n/2][n/2];
        int [][] B21 = new int[n/2][n/2];
        int [][] B22 = new int[n/2][n/2];



//Decomposition de A en 4 sous matrices A11,A12,A21,A22

            decomposer(A, A11, 0 , 0);
            decomposer(A, A12, 0 , n/2);
            decomposer(A, A21, n/2, 0);
            decomposer(A, A22, n/2, n/2);


//Decomposition de B en 4 sous matrices B11,B12,B21,B22

            decomposer(B, B11, 0 , 0);
            decomposer(B, B12, 0 , n/2);
            decomposer(B, B21, n/2, 0);
            decomposer(B, B22, n/2, n/2);

//les 7 appels recursifs M1,...M7 :
        int [][] M1 = strassen(add(A11, A22), add(B11, B22));
        int [][] M3 = strassen(A11, sub(B12, B22));
        int [][] M2 = strassen(add(A21, A22), B11);
        int [][] M4 = strassen(A22, sub(B21, B11));
        int [][] M5 = strassen(add(A11, A12), B22);
        int [][] M6 = strassen(sub(A21, A11), add(B11, B12));
        int [][] M7 = strassen(sub(A12, A22), add(B21, B22));

// calcul de C11,C12,C21,C22 :
        int [][] C11 = add(sub(add(M1, M4), M5), M7);
        int [][] C12 = add(M3, M5);
        int [][] C21 = add(M2, M4);
        int [][] C22 = add(sub(add(M1, M3), M2), M6);




/* Composition de la matrice C  a partir de C11,C12,C21,C22 */
            composer(C11, resultat, 0 , 0);
            composer(C12, resultat, 0 , n/2);
            composer(C21, resultat, n/2, 0);
            composer(C22, resultat, n/2, n/2);

}

        return resultat;
    }


    public static int [][] add(int [][] A, int [][] B)
    {
        int n = A.length;
        int[][] C = new int[n][n];
        for (int i = 0; i < n; i++)
```

```
                for (int j = 0; j < n; j++)
                    C[i][j] = A[i][j] + B[i][j];
            return C;

        }

        public static int [][] sub(int [][] A, int [][] B)
        {
               int n = A.length;
                int[][] C = new int[n][n];
                for (int i = 0; i < n; i++)
                    for (int j = 0; j < n; j++)
                        C[i][j] = A[i][j] - B[i][j];
                return C;

        }

        public static void decomposer (int[][] p1, int[][] c1, int iB,
            int jB)
        {
/* decomposition de p1: resultat dans c1
c1(n/2x n/2) doit contenir la partie de p1(nxn) a partir
de la ligne iB  et de la colonne jB de p1 */


            for(int i1 = 0, i2=iB; i1<c1.length; i1++, i2++)
            for(int j1 = 0, j2=jB; j1<c1.length; j1++, j2++)

            c1[i1][j1] = p1[i2][j2];



}

        public static void composer(int[][] c1, int[][] p1, int iB, int
            jB)
        {
/* Composition de p1( nxn)  a partir de  c1(n/2 x n/2) :
    affectation de c1 a la partie de p1 commencant a la ligne iB  et
     de la colonne jB de p1 */

            for(int i1 = 0, i2 = iB; i1 < c1.length; i1++, i2++)
            for(int j1 = 0, j2 = jB; j1 < c1.length; j1++, j2++)
                    p1[i2][j2] = c1[i1][j1];

        }


         public static void affiche(int [][] tab)
        {
            int n = tab.length;

            System.out.println();
            for(int i=0; i<n; i++)
            {
                for(int j=0; j<n; j++)
                {
                    System.out.print(tab[i][j] + "\t");
```

```
            }
            System.out.println();
        }
        System.out.println();
    }

     public static void lire(int [][] A,int [][] B)
{
         Random r = new Random();

int i,j;
        int N = A.length;

    for(i=0;i<N;i++)
     {
         for(j =0; j<N;j++)
         {
             A[i][j] = r.nextInt(10);
             B[i][j] = r.nextInt(10);

         }
     }
}

 public static void main(String[] args) {
        long startTime, endTime;
        float res1,res2;
     Scanner scan = new Scanner(System.in);
     System.out.print("Donner la taille des matrices n :");
         int N = scan.nextInt();
         int[][] A = new int[N][N];
         int[][] B = new int[N][N];
         int[][] C = new int[N][N];
         int[][] D = new int[N][N];


         lire(A,B);


         System.out.println("Matrix A : ");

            affiche(A);
            System.out.println("Matrix B : ");

            affiche(B);


            startTime = System.nanoTime();
            C=multiplication(A,B);
            endTime = System.nanoTime();
            res1 = (float) (endTime - startTime) / 1000000;


System.out.println("Resultat de la multiplication par la methode
   classique : ");

        affiche(C);
```

```java

        startTime = System.nanoTime();
        D = strassen(A,B);
        endTime = System.nanoTime();
        res2 = (float) (endTime - startTime) / 1000000;

System.out.println("Resultat de la multiplication par la methode de
     Strassen : ");

        affiche(D);

System.out.println("Nombre de multiplications avec la solution
    classique   :  "+cpt1);
System.out.println("Nombre de multiplications avec la solution de
    Strassen :  "+cpt2);
System.out.println("Temps de multiplication classique :"+ res1 + "
    ms");
System.out.println("Temps de multiplication par Strassen :"+ res2 +
     " ms");



 }

}
```

Bibliography

[1] Brice Boyer et al. **?**Memory efficient scheduling of Strassen-Winograds matrix multiplication algorithm**?** In: *Proceedings of the 2009 international symposium on Symbolic and algebraic computation - ISSAC 09* (2009). DOI: `10.1145/1576702.1576713`.

[2] Radu Ciucanu et al. **?**Secure Strassen-Winograd Matrix Multiplication with MapReduce**?** In: *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications* (2019). DOI: `10.5220/0007916302200227`.

[3] F. Desprez and F. Suter. **?**Impact of mixed-parallelism on parallel implementations of the Strassen and Winograd matrix multiplication algorithms**?** In: *Concurrency and Computation: Practice and Experience* 16.8 (2004), pp. 771–797. DOI: `10.1002/cpe.791`.

[4] Henrymorco. *Fastest Java Matrix Multiplication: NM DEV*. Aug. 2015. URL: `https://nm.dev/2015/08/07/accelerating-matrix-multiplication/`.

[5] Pai-Wei Lai et al. **?**Accelerating Strassen-Winograds matrix multiplication algorithm on GPUs**?** In: *20th Annual International Conference on High Performance Computing* (2013). DOI: `10.1109/hipc.2013.6799109`.

[6] O.m. Makarov. **?**The connection between algorithms of the fast Fourier and Hadamard transformations and the algorithms of Karatsuba, Strassen, and Winograd**?** In: *USSR Computational Mathematics and Mathematical Physics* 15.5 (1975), pp. 1–11. DOI: `10.1016/0041-5553(75)90099-3`.

[7] Richard E. Neapolitan. *Foundations of algorithms*. Jones and Bartlett Learning, 2015.