

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



*Module : Algorithmique et Complexité*  
1ST YEAR OF MASTER'S DEGREE 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  
ZENAIIDI Lamia

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

November 16, 2021

# Contents

<b>1</b>	<b>Solutions of Fiche TP-03</b>	<b>1</b>
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. . . . .	3
1.2	Afficher le nombre de multiplications exécutées par chaque algorithme. . . . .	3
1.2.1	When the matrix size is multiple of 2 . . . . .	4
1.2.2	When the matrix size is EVEN . . . . .	6
1.2.3	When the matrix size is ODD . . . . .	7
1.3	Afficher le temps d'exécution pour chaque algorithme. . . . .	10
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. . . . .	12
1.5	Faire des captures d'écran pour n=2 ; n=7 ; n=8. . . . .	15
1.5.1	n=2 . . . . .	15
1.5.2	n=7 . . . . .	15
1.5.3	n=8 . . . . .	16
1.6	Remplir le tableau suivant : . . . . .	16
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. . . . .	17
	Time Complexity of Classic Matrix Multiplication . . . . .	17
	Time Complexity of Strassen's Algorithm . . . . .	17
1.7	Que peut-on conclure ? . . . . .	19
<b>A</b>	<b>Appendix A</b>	<b>20</b>
A.1	Java Code . . . . .	20

## Chapter 1

# Solutions of Fiche TP-03

### Notes regarding this solution :

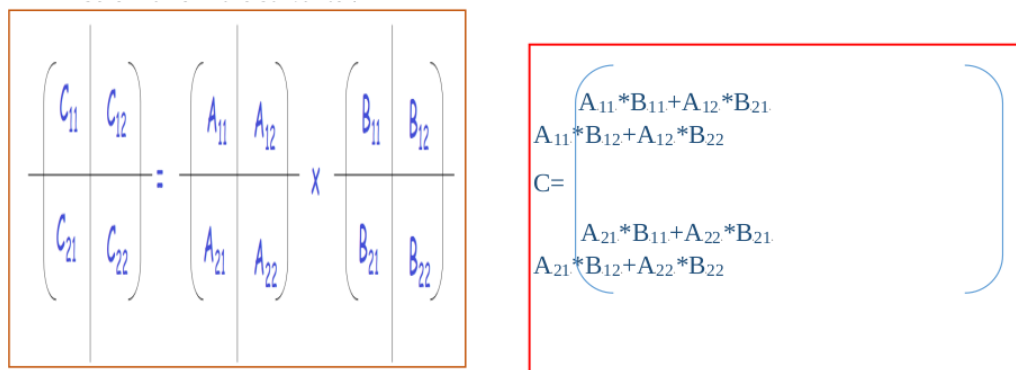
This solution and the executions of the code in it was done in the following machine :

- *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 :



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

$$C = \begin{pmatrix} A_{11} * B_{11} + A_{12} * B_{21} & A_{11} * B_{12} + A_{12} * B_{22} \\ A_{21} * B_{11} + A_{22} * B_{21} & A_{21} * B_{12} + A_{22} * B_{22} \end{pmatrix}$$

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  $2 \times 2$  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 :

The diagram illustrates the recursive step of Strassen's algorithm. It shows three main components:

- Matrix Partitioning:** Matrix  $C$  is partitioned into four sub-matrices  $C_{11}, C_{12}, C_{21}, C_{22}$ . Matrix  $A$  is partitioned into  $A_{11}, A_{12}, A_{21}, A_{22}$ . Matrix  $B$  is partitioned into  $B_{11}, B_{12}, B_{21}, B_{22}$ . The operation is  $C = A \times B$ .
- Intermediate Matrices:** Seven intermediate matrices are calculated using the following formulas:
 
$$\begin{aligned} M_1 &= (A_{11} + A_{22}) (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) B_{11} \\ M_3 &= A_{11} (B_{12} - B_{22}) \\ M_4 &= A_{22} (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) B_{22} \\ M_6 &= (A_{21} - A_{11}) (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) (B_{21} + B_{22}) \end{aligned}$$
- Final Assembly:** The result matrix  $C$  is assembled from the intermediate matrices:
 
$$C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

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.

```
1      for (int i = 0; i < aRows; i++) {
2          for (int j = 0; j < bColumns; j++) {
3              for (int k = 0; k < aColumns; k++) {
4                  C[i][j] += A[i][k] * B[k][j];
5                  cpt1++;
6              }
7          }
8      }
```

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**.

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

## 1.2.1 When the matrix size is multiple of 2

```

eclipse-workspace - algoTP1/src/algoTP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 15, 2021, 8:22:27 PM)
Donner la taille des matrices n :2
Matrix A :
3 3
2 1
Matrix B :
0 2
6 3
Resultat de la multiplication par la méthode classique :
18 15
6 7
Résultat de la multiplication par la méthode de Strassen :
18 15
6 7
Nombre de multiplications avec la solution classique : 8
Nombre de multiplications avec la solution de Strassen : 7
Temps de multiplication classique :0.017204 ms
Temps de multiplication par Strassen :0.058578 ms
158M of 258M

```

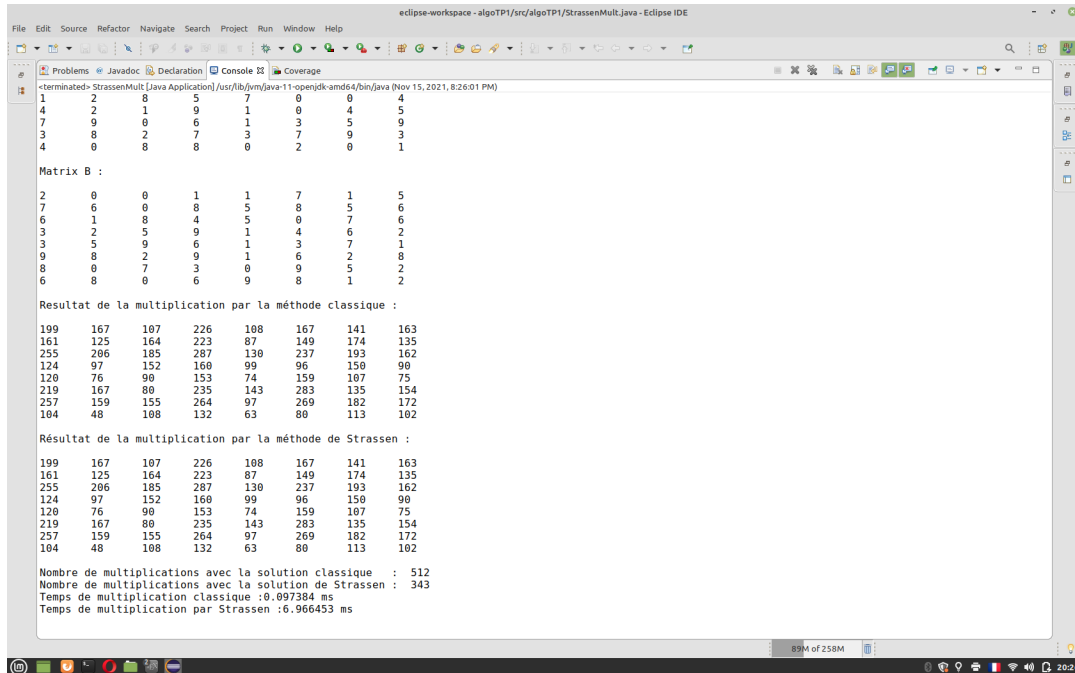
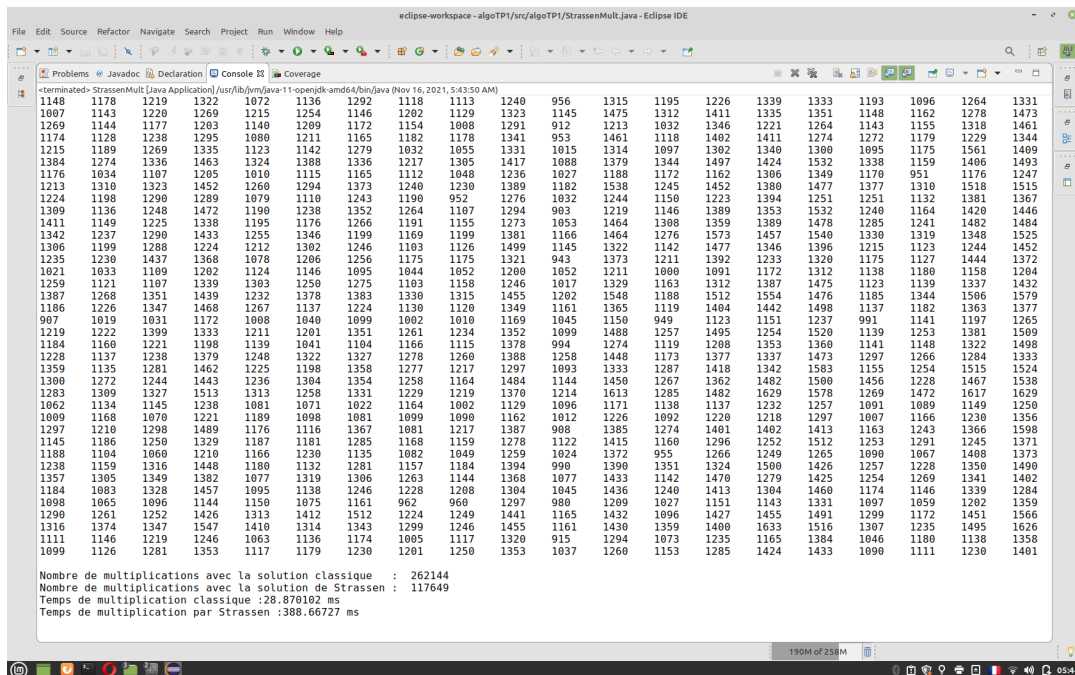
FIGURE 1.4: Execution of n=2

```

eclipse-workspace - algoTP1/src/algoTP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 5:43:04 AM)
Donner la taille des matrices n :4
Matrix A :
8 5 9 5
5 6 3 9
1 2 7 7
0 8 7 1
Matrix B :
5 5 6 2
5 2 5 7
2 3 6 1
6 7 2 8
Resultat de la multiplication par la méthode classique :
113 112 137 100
115 109 96 127
71 79 72 79
60 44 84 71
Résultat de la multiplication par la méthode de Strassen :
113 112 137 100
115 109 96 127
71 79 72 79
60 44 84 71
Nombre de multiplications avec la solution classique : 64
Nombre de multiplications avec la solution de Strassen : 49
Temps de multiplication classique :0.040332 ms
Temps de multiplication par Strassen :0.292042 ms
117M of 258M

```

FIGURE 1.5: Execution of n=4

FIGURE 1.6: Execution of  $n=8$ FIGURE 1.7: Execution of  $n=64$

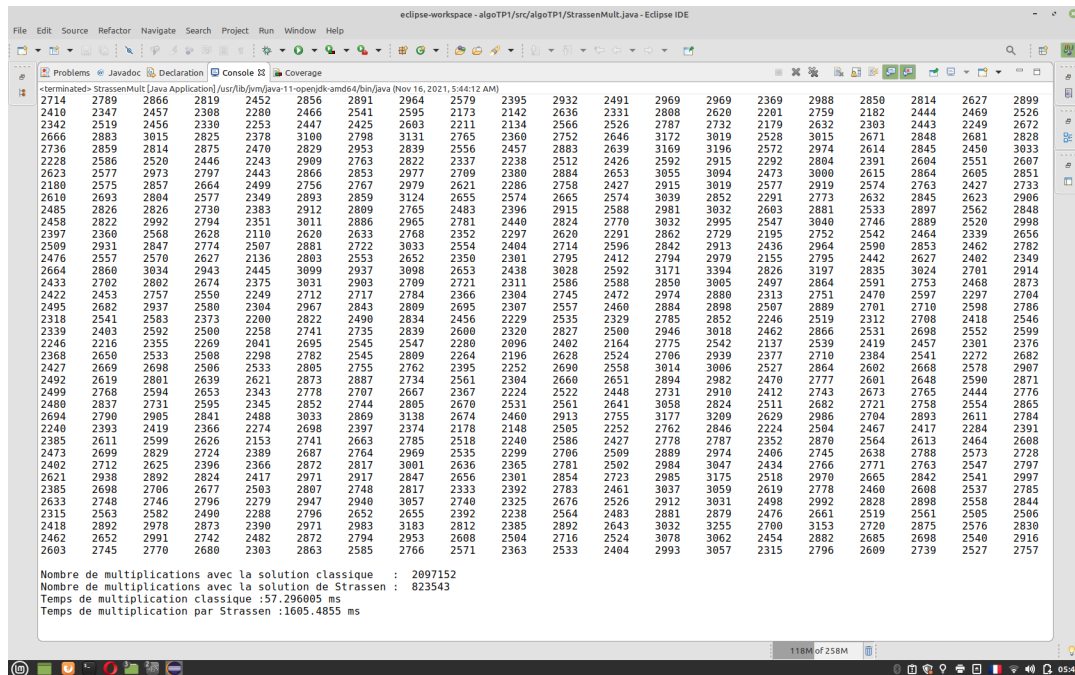


FIGURE 1.8: Execution of n=128

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 dramatically 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

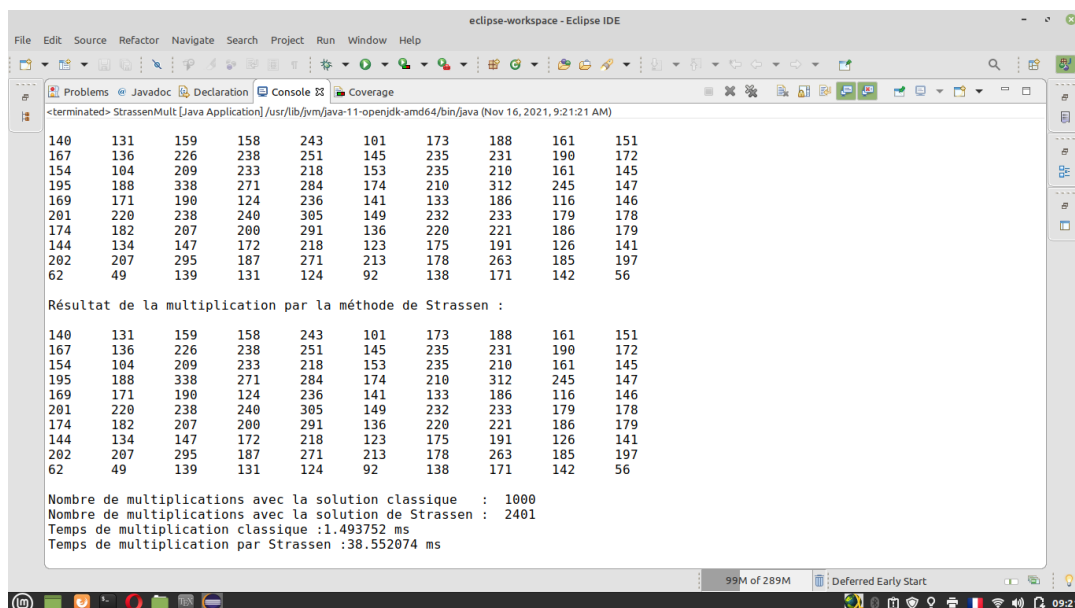
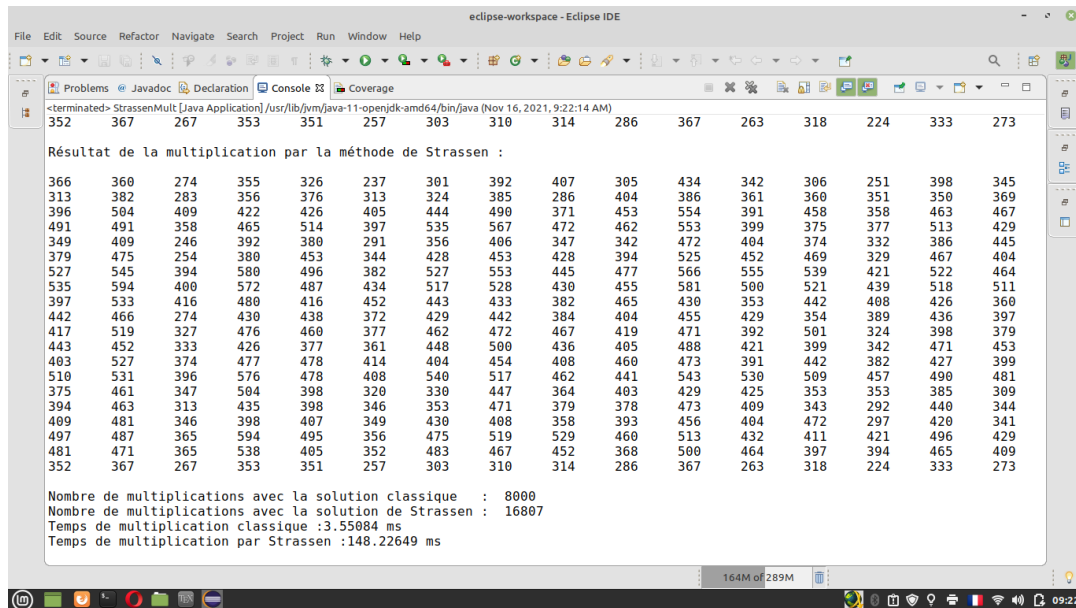


FIGURE 1.9: Execution of n=10





```

eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 9:22:14 AM)
352 367 267 353 351 257 303 310 314 286 367 263 318 224 333 273

Résultat de la multiplication par la méthode de Strassen :

366 360 274 355 326 237 301 392 407 305 434 342 306 251 398 345
313 382 283 356 376 313 324 385 286 404 386 361 360 351 350 369
396 504 409 422 426 405 444 490 371 453 554 391 458 358 463 467
491 491 358 465 514 397 535 567 472 462 553 399 375 377 513 429
349 409 246 392 380 291 356 406 347 342 472 404 374 332 386 445
379 475 254 380 453 344 428 453 428 394 525 452 469 329 467 404
527 545 394 580 496 382 527 553 445 477 566 555 539 421 522 464
535 594 400 572 487 434 517 528 430 455 581 500 521 439 518 511
397 533 416 480 416 452 443 433 382 465 430 353 442 408 426 360
442 466 274 430 438 372 429 442 384 404 455 429 354 389 436 397
417 519 327 476 460 377 462 472 467 419 392 501 324 398 379
443 452 333 426 377 361 448 500 436 405 488 421 399 342 471 453
403 527 374 477 478 414 404 454 408 460 473 391 442 382 427 399
510 531 396 576 478 408 540 517 462 441 543 530 509 457 490 481
375 461 347 504 398 320 330 447 364 403 429 425 353 353 385 309
394 463 313 435 398 346 353 471 379 378 473 409 343 292 440 344
409 481 346 398 407 349 430 408 358 393 456 404 472 297 420 341
497 487 365 594 495 356 475 519 529 460 513 432 411 421 496 429
481 471 365 538 405 352 483 467 452 368 500 464 397 394 465 409
352 367 267 353 351 257 303 310 314 286 367 263 318 224 333 273

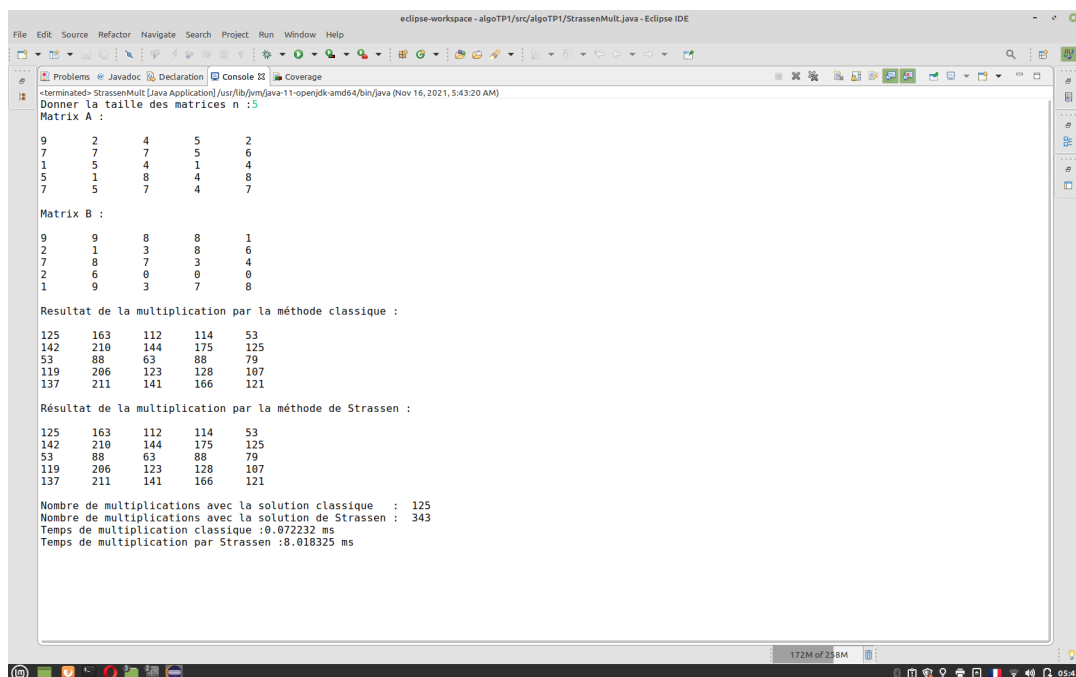
Nombre de multiplications avec la solution classique : 8000
Nombre de multiplications avec la solution de Strassen : 16807
Temps de multiplication classique :3.55084 ms
Temps de multiplication par Strassen :148.22649 ms
164M of 289M
09:22

```

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



```

eclipse-workspace - algoTP1/src/algotP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 5:43:20 AM)
Donner la taille des matrices n :5
Matrix A :
9 2 4 5 2
7 7 7 5 6
1 5 4 1 4
5 1 8 4 8
7 5 7 4 7

Matrix B :
9 9 8 8 1
2 1 3 8 6
7 8 7 3 4
2 6 0 0 0
1 9 3 7 8

Résultat de la multiplication par la méthode classique :
125 163 112 114 53
142 210 144 175 125
53 88 63 88 79
119 206 123 128 107
137 211 141 166 121

Résultat de la multiplication par la méthode de Strassen :
125 163 112 114 53
142 210 144 175 125
53 88 63 88 79
119 206 123 128 107
137 211 141 166 121

Nombre de multiplications avec la solution classique : 125
Nombre de multiplications avec la solution de Strassen : 343
Temps de multiplication classique :0.072232 ms
Temps de multiplication par Strassen :0.018325 ms
172M of 288M
05:43

```

FIGURE 1.11: Excution of n=5

```

eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 9:21:53 AM)
268 242 208 253 188 289 216 256 188 159 294
272 158 154 176 150 252 179 198 177 120 276
286 184 209 201 156 264 163 244 176 201 297
355 207 189 219 148 363 276 257 194 195 347
413 270 273 270 233 370 248 290 233 228 340
241 144 137 153 74 253 168 177 144 194 227
337 214 225 209 163 324 221 192 200 177 271
356 260 257 285 192 416 283 302 241 257 390
359 227 227 236 219 291 192 320 166 244 342
275 183 152 216 161 299 269 227 210 146 372

Résultat de la multiplication par la méthode de Strassen :

261 209 179 219 136 282 234 190 208 152 274
268 242 208 253 188 289 216 256 188 159 294
272 158 154 176 150 252 179 198 177 120 276
286 184 209 201 156 264 163 244 176 201 297
355 207 189 219 148 363 276 257 194 195 347
413 270 273 270 233 370 248 290 233 228 340
241 144 137 153 74 253 168 177 144 194 227
337 214 225 209 163 324 221 192 200 177 271
356 260 257 285 192 416 283 302 241 257 390
359 227 227 236 219 291 192 320 166 244 342
275 183 152 216 161 299 269 227 210 146 372

Nombre de multiplications avec la solution classique : 1331
Nombre de multiplications avec la solution de Strassen : 2401
Temps de multiplication classique :0.200164 ms
Temps de multiplication par Strassen :23.350996 ms
169M of 289M
09:22

```

FIGURE 1.12: Excution of n=11

```

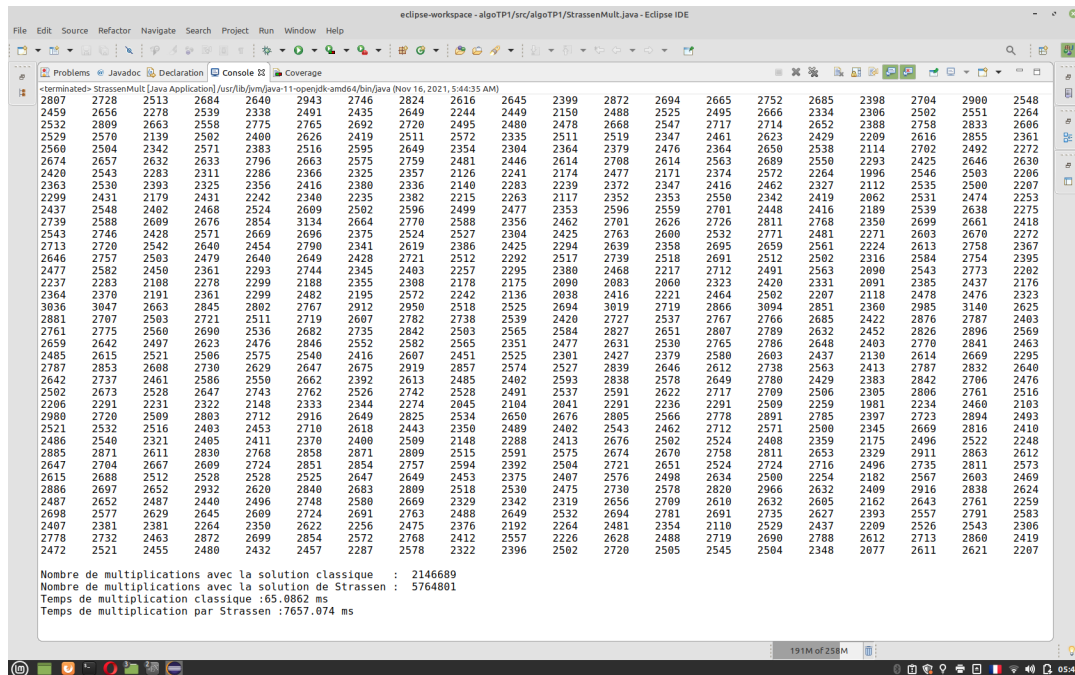
eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 9:22:30 AM)
Résultat de la multiplication par la méthode de Strassen :

400 378 374 336 378 479 382 377 290 393 373 371 472 344 284 533
370 401 424 406 325 423 437 402 308 407 413 406 507 308 353 532
530 539 583 554 502 520 517 406 441 524 425 409 583 347 446 632
463 447 474 446 433 443 430 404 404 416 451 437 530 323 287 636
399 349 404 336 352 238 324 371 335 322 308 364 407 248 299 451
464 402 506 490 355 479 473 441 381 422 469 371 461 294 362 532
440 433 515 472 392 424 469 440 333 405 500 436 470 274 381 534
440 491 527 482 478 483 506 386 380 505 449 370 466 390 350 591
464 521 631 543 527 520 538 443 454 525 431 420 556 377 415 609
411 374 498 405 379 389 329 337 377 347 323 282 446 241 347 465
358 337 477 376 319 312 390 320 299 324 325 334 410 193 369 443
422 450 507 369 409 446 491 345 346 387 388 405 473 404 333 573
445 395 375 402 398 347 350 474 332 372 336 417 455 338 296 506
371 351 413 464 374 382 479 405 335 408 468 394 382 298 286 551
421 430 583 501 474 410 463 473 412 447 494 418 471 309 342 569
324 384 478 434 407 384 461 367 321 449 362 362 454 298 366 507
366 338 425 440 357 362 411 351 360 367 399 366 421 307 381 527
419 402 425 335 333 335 403 327 290 299 307 349 423 211 281 435
503 443 502 598 356 456 461 466 484 481 471 446 600 328 485 611
471 437 513 550 460 488 464 465 437 494 465 450 548 356 393 622
469 448 577 559 450 447 479 504 431 511 522 466 515 398 442 703

Nombre de multiplications avec la solution classique : 9261
Nombre de multiplications avec la solution de Strassen : 16807
Temps de multiplication classique :3.334535 ms
Temps de multiplication par Strassen :131.87355 ms
148M of 289M
09:22

```

FIGURE 1.13: Excution of n=21

FIGURE 1.14: Exccution 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.

```

eclipse-workspace - algoTP1/src/TP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
StrassenMult.java 66
Problems Javadoc Declaration Console Coverage
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 15, 2021, 10:59:04 AM)
Donner la taille des matrices n : 1
Matrix A :
1
Matrix B :
7
Resultat de la multiplication par la méthode classique :
7
Résultat de la multiplication par la méthode de Strassen :
7
Nombre de multiplications avec la solution classique : 1
Nombre de multiplications avec la solution de Strassen : 1
Temps de multiplication classique : 0.012864 ms
Temps de multiplication par Strassen : 0.009922 ms
166M of 258M

```

FIGURE 1.15: Exccution of n=1

```

eclipse-workspace - algoTP1/src/TP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
StrassenMult.java 66
Problems Javadoc Declaration Console Coverage
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 5:45:04 AM)
10195 10539 10243 10596 9874 10520 10355 10753 10771 10529 10479
10564 10172 9900 10290 10019 10725 10307 10857 9696 9825 10603 9962 10358 9836 10254 9585 10498 10434 9812 10431
10384 10032 10091 9999 9784 10700 10643 10886 9817 9940 10001 9796 10329 10170 10177 9782 10208 10581 9914 9940
10474 10413 10403 10259 10433 10929 10218 10886 9912 10679 10246 9684 10840 10291 10303 9769 10571 10318 10315 9972
10359 9904 10255 10162 9891 10510 10376 10640 9707 10137 10314 9303 10136 10047 10035 9664 10013 10603 9971 9974
11130 10508 10630 10608 10427 11015 10681 11163 9840 10018 10492 10271 10625 10781 10658 9804 10411 10875 10550 10127
11825 10860 11209 10920 10892 11603 11209 11828 10400 10981 11102 10865 11262 10980 11162 10555 11441 11467 11263 10887
11038 10183 10267 10344 9966 10791 10584 11054 9755 10345 10496 9776 10560 10395 10580 9479 10626 11084 10110 9948
10992 9632 9859 9558 9657 10584 10955 10587 9582 9637 10024 9414 10106 9803 9663 9421 10199 10287 9773 9902
10702 10161 10366 10645 10001 10741 10690 10897 9679 10194 10318 9985 10375 10403 10296 9988 10696 11097 10150 10632
10462 10271 10619 10216 10106 10693 10684 11077 10328 10099 10428 9973 10625 10394 10258 10155 10362 10919 10504 10536
10445 9894 10463 10021 9608 10968 10266 10927 9750 9792 10160 9837 10354 10173 10028 9594 10558 10429 10172 10118
10454 9992 10352 10518 10035 10671 10106 10814 9295 9845 10192 9816 10446 10083 10107 9613 10317 10265 10201 9913
10966 10769 10973 10630 10351 11049 10622 11058 10182 10518 10467 9977 10702 10391 10515 10356 10777 11203 10264 10078
10644 10127 10076 9985 9873 10328 10458 10935 9664 9632 10187 9762 10242 10164 10434 9600 9780 10674 9908 9573
10053 9244 9818 9804 9652 10693 10952 10509 9518 9563 10058 9555 10077 9877 9942 9297 9815 10359 9763 9816
10660 9908 10720 10305 10061 11080 10435 11208 10015 10121 10345 9848 10087 10361 10118 9998 10277 10781 10622 10190
11029 10754 10853 10930 10601 11308 11050 11033 10380 10474 10107 11045 10516 10293 10573 11166 11082 10598 10461
10065 9814 10182 10049 9418 10469 10016 10856 9553 9830 10237 9521 10230 10126 9827 9746 9775 9994 9891 9773
10061 10136 10624 10664 9783 11219 10467 11133 9929 10141 10408 10110 10002 10119 10010 10271 10635 10940 10378 10044
10521 10109 10133 10204 9957 10851 10292 10914 9705 9871 10038 9985 10515 10158 9968 9540 10430 10976 10057 10047
10547 9074 10304 10661 9999 10933 10539 11015 9909 10325 10395 10062 10025 10242 10330 10152 10504 10704 10290 10273
10213 9599 9923 9867 9434 10388 9657 10342 9502 9404 9906 9804 10211 9807 9510 9150 9807 10436 9560 9590
10299 10134 10465 10217 10118 10812 10489 10965 9445 9698 10034 10011 10454 10285 10087 9500 10536 10481 10434 9941
10675 10025 10903 10733 9691 11077 11060 11023 10382 10140 10587 10064 10742 10336 10156 9964 10556 11465 10244 10448
10793 10474 10604 10370 10184 11157 10735 11335 10040 10279 10630 9098 10092 10647 10401 10493 10750 10998 10665 10301
10662 10516 10504 10198 9885 10799 10317 10984 9876 10297 10080 9808 10533 10031 10042 9796 10497 10895 10029 10228
Nombre de multiplications avec la solution classique : 134217728
Nombre de multiplications avec la solution de Strassen : 40353607
Temps de multiplication classique : 678.5535 ms
Temps de multiplication par Strassen : 50901.887 ms
165M of 258M

```

FIGURE 1.16: Execution 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. first 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*:  $O(N^2)$  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*:  $O(> N \log(N))$  faster than Karatsuba/Toom-3 at 700 digits
- *Schönhage–Strassen algorithm (SSA)*:  $O(N \log(n) \log \log(n))$  faster than FFT at a billion digits
- *Fixed-width Number-Theoretic Transform*:  $O(N \log(n))$  faster than SSA at a few billion digits

**Input : Matrices  $A, B$**

**Phase 1**

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

**Phase 2**

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

**Phase 3**

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

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

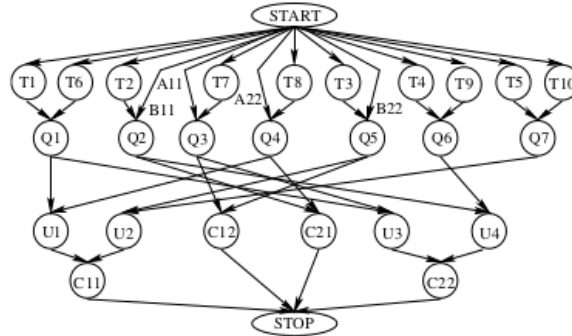


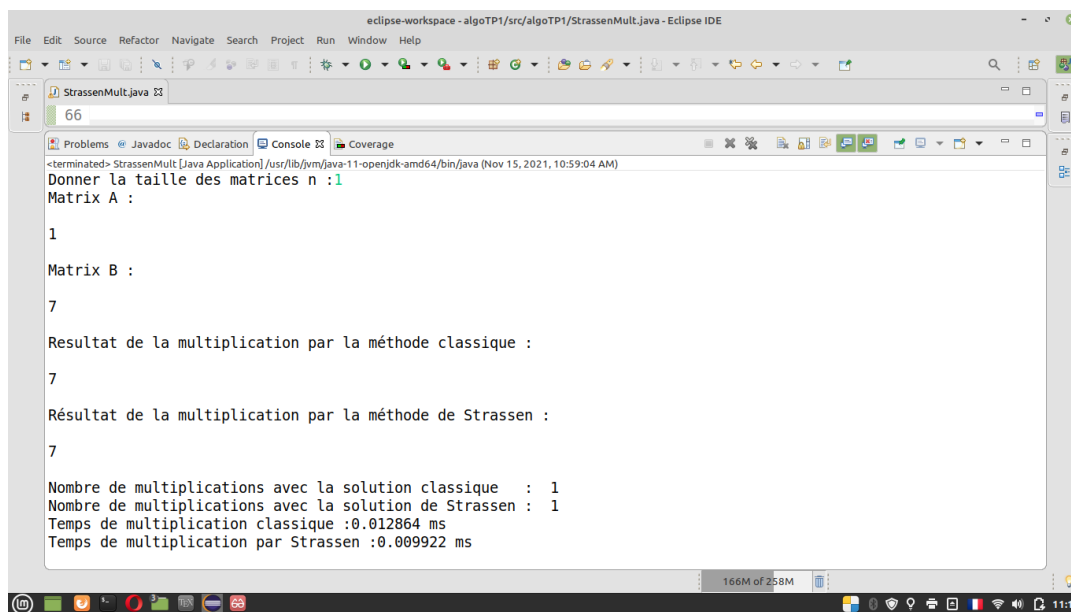
FIGURE 1.17: First level of recursion of the Strassen algorithm and its MDG representation.

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  $\theta$  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.



```
eclipse-workspace - algoTP1/src/algTP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
66
StrassenMult.java
66
Problems Javadoc Declaration Console Coverage
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 15, 2021, 10:59:04 AM)
Donner la taille des matrices n :1
Matrix A :
1
Matrix B :
7
Resultat de la multiplication par la méthode classique :
7
Résultat de la multiplication par la méthode de Strassen :
7
Nombre de multiplications avec la solution classique : 1
Nombre de multiplications avec la solution de Strassen : 1
Temps de multiplication classique :0.012864 ms
Temps de multiplication par Strassen :0.009922 ms
166M of 258M
11:12
```

FIGURE 1.18: Exécution de  $n=1$

```

eclipse-workspace - algoTP1/src/TP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 5:43:04 AM)
Donner la taille des matrices n :4
Matrix A :
8 5 9 5
5 6 3 9
1 2 7 7
0 8 7 1
Matrix B :
5 5 6 2
5 2 5 7
2 3 6 1
6 7 2 8
Resultat de la multiplication par la méthode classique :
113 112 137 100
115 109 96 127
71 79 72 79
60 44 84 71
Résultat de la multiplication par la méthode de Strassen :
113 112 137 100
115 109 96 127
71 79 72 79
60 44 84 71
Nombre de multiplications avec la solution classique : 64
Nombre de multiplications avec la solution de Strassen : 49
Temps de multiplication classique :0.048332 ms
Temps de multiplication par Strassen :0.292042 ms
117M of 258M
05:43

```

FIGURE 1.19: Exécution de  $n=4$ 

```

eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 16, 2021, 9:21:53 AM)
268 242 208 253 188 289 216 256 188 159 294
272 158 154 176 150 252 179 198 177 120 276
286 184 209 201 156 264 163 244 176 201 297
355 207 189 219 148 363 276 257 194 195 347
413 270 273 270 233 370 248 290 233 228 340
241 144 137 153 74 253 168 177 144 194 227
337 214 225 209 163 324 221 192 200 177 271
356 260 257 285 192 416 283 302 241 257 390
359 227 227 236 219 291 192 320 166 244 342
275 183 152 216 161 299 269 227 210 146 372
Résultat de la multiplication par la méthode de Strassen :
261 209 179 219 136 282 234 190 208 152 274
268 242 208 253 188 289 216 256 188 159 294
272 158 154 176 150 252 179 198 177 120 276
286 184 209 201 156 264 163 244 176 201 297
355 207 189 219 148 363 276 257 194 195 347
413 270 273 270 233 370 248 290 233 228 340
241 144 137 153 74 253 168 177 144 194 227
337 214 225 209 163 324 221 192 200 177 271
356 260 257 285 192 416 283 302 241 257 390
359 227 227 236 219 291 192 320 166 244 342
275 183 152 216 161 299 269 227 210 146 372
Nombre de multiplications avec la solution classique : 1331
Nombre de multiplications avec la solution de Strassen : 2401
Temps de multiplication classique :0.200164 ms
Temps de multiplication par Strassen :23.350996 ms
169M of 289M
09:22

```

FIGURE 1.20: Exécution de  $n=11$



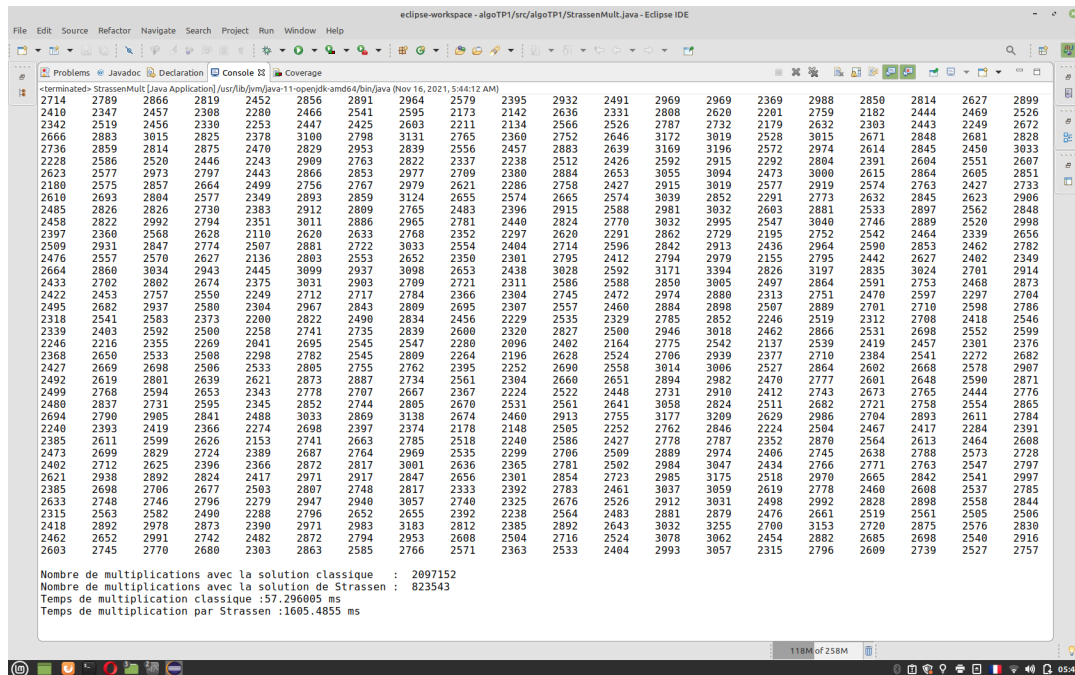


FIGURE 1.21: Execution of n=128

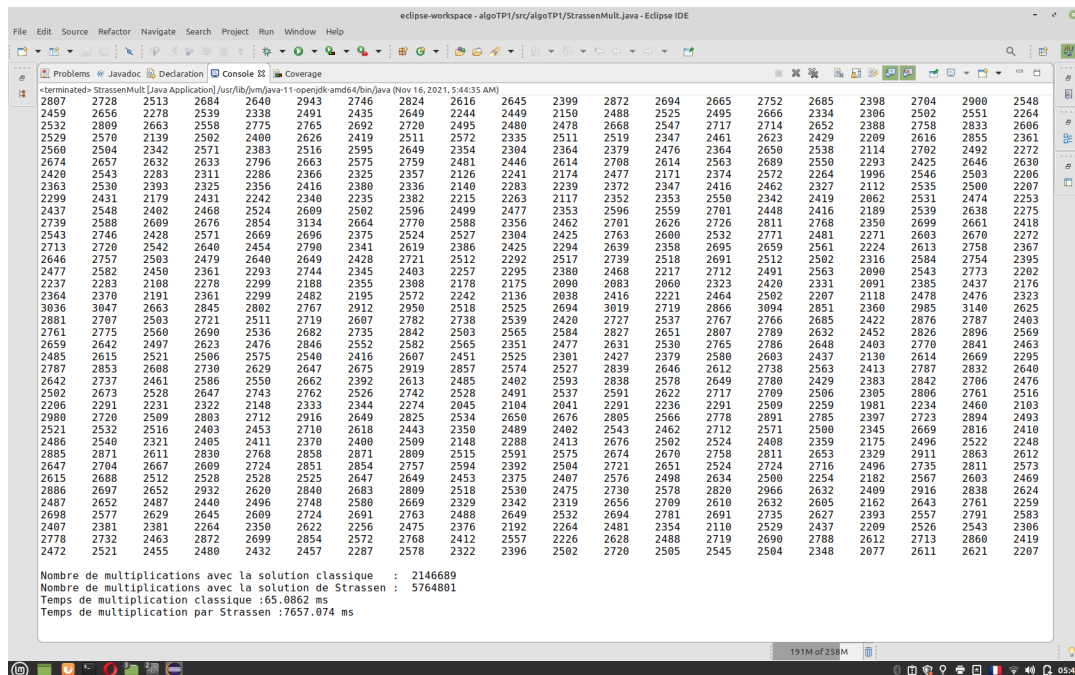


FIGURE 1.22: Execution of n=129



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

### 1.5.1 $n=2$

```

eclipse-workspace - algoTP1/src/algotP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
-terminated- StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 15, 2021, 8:22:27 PM)
Donner la taille des matrices n : 2
Matrix A :
3 3
2 1
Matrix B :
0 2
6 3
Resultat de la multiplication par la méthode classique :
18 15
6 7
Résultat de la multiplication par la méthode de Strassen :
18 15
6 7
Nombre de multiplications avec la solution classique : 8
Nombre de multiplications avec la solution de Strassen : 7
Temps de multiplication classique : 0.017284 ms
Temps de multiplication par Strassen : 0.058578 ms
158M of 258M
2021

```

FIGURE 1.23: Exécution de  $n=2$ 

### 1.5.2 $n=7$

```

eclipse-workspace - algoTP1/src/algotP1/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
-terminated- StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 15, 2021, 8:25:38 PM)
4 7 0 5 0 7 7
9 4 1 7 2 1 1
9 8 9 0 0 2 8
8 7 8 6 7 4 5
3 8 1 9 6 9 5
6 8 7 6 6 3 8
7 6 3 0 1 9 2
Matrix B :
2 2 4 3 0 2 0
1 4 4 9 4 6 1
3 6 2 6 9 0 9
9 4 2 7 0 6 7
9 4 2 9 2 9 7
7 0 2 9 9 1 6
9 0 6 2 8 9 5
Resultat de la multiplication par la méthode classique :
172 56 110 187 147 150 119
122 76 80 147 46 112 87
139 104 138 187 195 140 141
237 144 140 286 190 206 219
260 104 124 295 174 216 201
242 134 148 271 198 225 213
119 60 90 187 150 86 104
Résultat de la multiplication par la méthode de Strassen :
172 56 110 187 147 150 119
122 76 80 147 46 112 87
139 104 138 187 195 140 141
237 144 140 286 190 206 219
260 104 124 295 174 216 201
242 134 148 271 198 225 213
119 60 90 187 150 86 104
Nombre de multiplications avec la solution classique : 343
Nombre de multiplications avec la solution de Strassen : 343
Temps de multiplication classique : 0.063269 ms
Temps de multiplication par Strassen : 3.51867 ms
91M of 258M
2021

```

FIGURE 1.24: Exécution de  $n=7$

## 1.5.3 n=8

```

eclipse-workspace - algoTP1/src/StrassenMult.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> StrassenMult [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Nov 15, 2021, 8:26:01 PM)
1 2 8 5 7 0 0 4
4 2 1 9 1 0 4 5
7 9 0 6 1 3 5 9
3 8 2 7 3 7 9 3
4 0 8 8 0 2 0 1

Matrix B :
2 0 0 1 1 7 1 5
7 6 0 8 5 8 5 6
6 1 8 4 5 0 7 6
3 2 5 9 1 4 6 2
3 5 9 6 1 3 7 1
9 8 2 9 1 6 2 8
8 0 7 3 0 9 5 2
6 8 0 6 9 8 1 2

Résultat de la multiplication par la méthode classique :
199 167 107 226 108 167 141 163
161 125 164 223 87 149 174 135
255 206 185 287 130 237 193 162
124 97 152 160 99 96 150 90
120 76 90 153 74 159 107 75
219 167 80 235 143 283 135 154
257 159 155 264 97 269 182 172
104 48 108 132 63 80 113 102

Résultat de la multiplication par la méthode de Strassen :
199 167 107 226 108 167 141 163
161 125 164 223 87 149 174 135
255 206 185 287 130 237 193 162
124 97 152 160 99 96 150 90
120 76 90 153 74 159 107 75
219 167 80 235 143 283 135 154
257 159 155 264 97 269 182 172
104 48 108 132 63 80 113 102

Nombre de multiplications avec la solution classique : 512
Nombre de multiplications avec la solution de Strassen : 343
Temps de multiplication classique : 0.097384 ms
Temps de multiplication par Strassen : 6.966453 ms

```

FIGURE 1.25: Execution 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**

```

13 for (int i = 0; i < aRows; i++) { ----- n
14     for (int j = 0; j < bColumns; j++) { ----- n*n
15         for (int k = 0; k < aColumns; k++) { ---- n*n*n
16             C[i][j] += A[i][k] * B[k][j]; ----- n*n*n+1
17         }
18     }
19 }

```

Therefore 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**:

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

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

Therefore we reach :

$$T(n) = 7^{\log n}$$

To prove that this is correct :

$$T(1) = 7^{\log 1} = 7^0 = 1$$

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

$$T(2n) = 7^{\log 2n}$$

Finally, because :

$$7^{\log n} = n^{\log 7}$$

this recurrence is usually given as :

$$T(n) = n^{\log 7} \simeq n^{2.81}$$

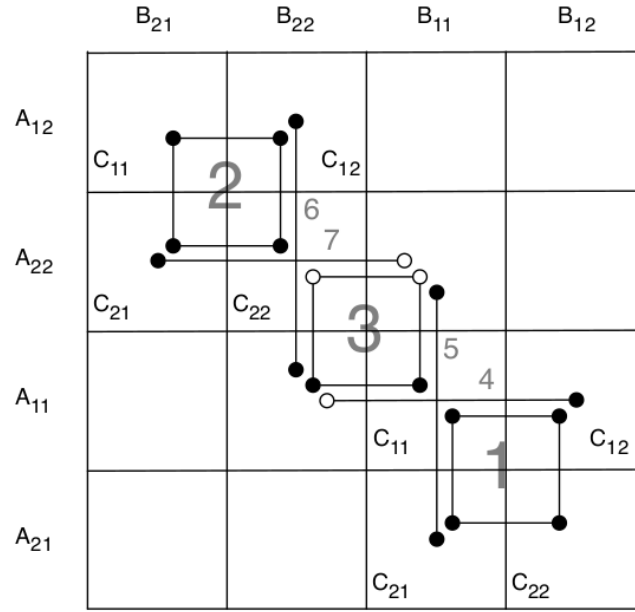


FIGURE 1.26: Strassen's algorithm

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 multiplies. Hence, the work is

given by a recurrence of the form

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

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

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

## 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.

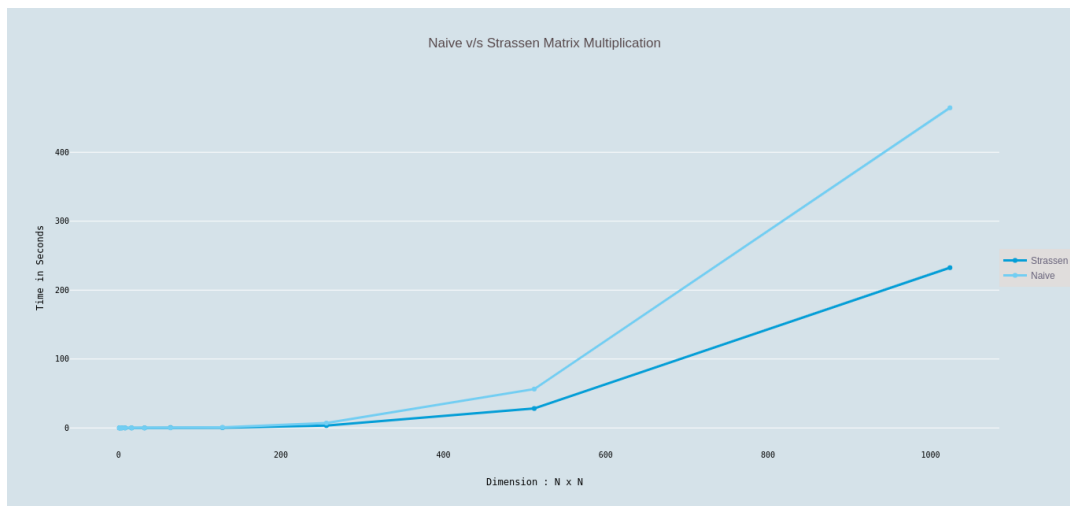


FIGURE 1.27: Strassen's algorithm vs Classic Method

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  $n = \text{power of } 2$  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

```

20 //TP3 Algorithmique et Complexite 2021-2022
21 //LIRE ET BIEN COMPRENDRE LE CODE AVANT DE LE COMPLETER
22 //Respecter la demarche a suivre donnee en commentaires
23
24 //Nom:HADJAZI
25 //Prenom: Mohammed Hisham
26 //Specialite:  RSSI      Groupe: 01
27
28 //Nom:Ameur
29 //Prenom: Wassim Malik
30 //Specialite:  RSSI      Groupe: 01
31
32 import java.util.*;
33
34 public class StrassenMult {
35
36     static long cpt1 = 0;
37     static long cpt2 = 0;
38
39     public static int [][] multiplication(int[][] A, int[][] B)
40     {
41         int aRows = A.length;
42         int aColumns = A[0].length;
43         int bRows = B.length;
44         int bColumns = B[0].length;
45
46         if (aColumns != bRows) {
47             throw new IllegalArgumentException("A:Rows: " + aColumns
48                 + " did not match B:Columns " + bRows + ".");
49         }
50
51         int[][] C = new int[aRows][bColumns];
52         for (int i = 0; i < aRows; i++) {
53             for (int j = 0; j < bColumns; j++) {
54                 C[i][j] = 0;
55             }
56         }
57
58         for (int i = 0; i < aRows; i++) {
59             for (int j = 0; j < bColumns; j++) {
60                 for (int k = 0; k < aColumns; k++) {
61                     C[i][j] += A[i][k] * B[k][j];

```

```

62         cpt1++;
63     }
64 }
65 }
66
67 return C;
68
69 }
70
71 public static int [][] strassen(int [][] A, int [][] B)
72 {
73     cpt2++;
74     int n = A.length;
75     int [][] resultat = new int[n][n];
76
77     if((n%2 != 0 ) && (n !=1))
78     {
79         /*Ajouter une ligne de 0 et une colonne de 0 pour A et B :
80         definir 3 nouvelles matrices temporaires A1,B1,C1
81         de taille n+1, ensuite A1=A et B1=B          */
82         int [][] A1, B1, C1;
83         int n1 = n+1;
84         A1 = new int[n1][n1];
85         B1 = new int[n1][n1];
86
87         for(int i=0;i<n;i++)
88             for(int j=0;j<n;j++)
89                 A1[i][j]=A[i][j];
90         for(int i=0;i<n;i++)
91             for(int j=0;j<n;j++)
92                 B1[i][j]=B[i][j];
93
94
95         C1 = strassen(A1, B1);
96         for(int i=0; i<n; i++)
97             for(int j=0; j<n; j++)
98                 resultat[i][j] =C1[i][j];
99         return resultat;
100     }
101
102
103     if(n == 1)
104     {
105         resultat[0][0] = A[0][0] * B[0][0];
106     }
107     else
108     {
109
110         //Creation de 4 sous matrices A11,A12,A21,A22 (n/2 x n/2)
111         int [][] A11 = new int[n/2][n/2];
112         int [][] A12 = new int[n/2][n/2];
113         int [][] A21 = new int[n/2][n/2];
114         int [][] A22 = new int[n/2][n/2];
115
116
117
118         //Creation de 4 sous matrices B11,B12,B21,B22 (n/2 x n/2)

```



```

119         int [][] B11 = new int[n/2][n/2];
120         int [][] B12 = new int[n/2][n/2];
121         int [][] B21 = new int[n/2][n/2];
122         int [][] B22 = new int[n/2][n/2];
123
124
125
126         //Decomposition de A en 4 sous matrices A11,A12,A21,A22
127
128         decomposer(A, A11, 0 , 0);
129         decomposer(A, A12, 0 , n/2);
130         decomposer(A, A21, n/2, 0);
131         decomposer(A, A22, n/2, n/2);
132
133
134         //Decomposition de B en 4 sous matrices B11,B12,B21,B22
135
136         decomposer(B, B11, 0 , 0);
137         decomposer(B, B12, 0 , n/2);
138         decomposer(B, B21, n/2, 0);
139         decomposer(B, B22, n/2, n/2);
140
141         //les 7 appels recursifs M1,...M7 :
142         int [][] M1 = strassen(add(A11, A22), add(B11, B22));
143         int [][] M3 = strassen(A11, sub(B12, B22));
144         int [][] M2 = strassen(add(A21, A22), B11);
145         int [][] M4 = strassen(A22, sub(B21, B11));
146         int [][] M5 = strassen(add(A11, A12), B22);
147         int [][] M6 = strassen(sub(A21, A11), add(B11, B12));
148         int [][] M7 = strassen(sub(A12, A22), add(B21, B22));
149
150         // calcul de C11,C12,C21,C22 :
151         int [][] C11 = add(sub(add(M1, M4), M5), M7);
152         int [][] C12 = add(M3, M5);
153         int [][] C21 = add(M2, M4);
154         int [][] C22 = add(sub(add(M1, M3), M2), M6);
155
156
157
158
159         /* Composition de la matrice C a partir de C11,C12,C21,C22 */
160         composer(C11, resultat, 0 , 0);
161         composer(C12, resultat, 0 , n/2);
162         composer(C21, resultat, n/2, 0);
163         composer(C22, resultat, n/2, n/2);
164
165     }
166
167     return resultat;
168 }
169
170
171 public static int [][] add(int [][] A, int [][] B)
172 {
173     int n = A.length;
174     int [][] C = new int[n][n];
175     for (int i = 0; i < n; i++)

```

```

176         for (int j = 0; j < n; j++)
177             C[i][j] = A[i][j] + B[i][j];
178     return C;
179
180 }
181
182 public static int [][] sub(int [][] A, int [][] B)
183 {
184     int n = A.length;
185     int[][] C = new int[n][n];
186     for (int i = 0; i < n; i++)
187         for (int j = 0; j < n; j++)
188             C[i][j] = A[i][j] - B[i][j];
189     return C;
190
191 }
192
193 public static void decomposer (int[][] p1, int[][] c1, int iB,
194     int jB)
195 {
196     /* decomposition de p1: resultat dans c1
197     c1(n/2x n/2) doit contenir la partie de p1(nxn) a partir
198     de la ligne iB et de la colonne jB de p1 */
199
200     for(int i1 = 0, i2=iB; i1<c1.length; i1++, i2++)
201         for(int j1 = 0, j2=jB; j1<c1.length; j1++, j2++)
202
203             c1[i1][j1] = p1[i2][j2];
204
205 }
206
207
208 public static void composer(int[][] c1, int[][] p1, int iB, int
209     jB)
210 {
211     /* Composition de p1( nxn) a partir de c1(n/2 x n/2) :
212     affectation de c1 a la partie de p1 commençant a la ligne iB et
213     de la colonne jB de p1 */
214
215     for(int i1 = 0, i2 = iB; i1 < c1.length; i1++, i2++)
216         for(int j1 = 0, j2 = jB; j1 < c1.length; j1++, j2++)
217             p1[i2][j2] = c1[i1][j1];
218
219 }
220
221 public static void affiche(int [][] tab)
222 {
223     int n = tab.length;
224
225     System.out.println();
226     for(int i=0; i<n; i++)
227     {
228         for(int j=0; j<n; j++)
229         {
230             System.out.print(tab[i][j] + "\t");

```

```
229         }
230         System.out.println();
231     }
232     System.out.println();
233 }
234
235     public static void lire(int [][] A,int [][] B)
236 {
237     Random r = new Random();
238
239     int i,j;
240     int N = A.length;
241
242     for(i=0;i<N;i++)
243     {
244         for(j =0; j<N;j++)
245         {
246             A[i][j] = r.nextInt(10);
247             B[i][j] = r.nextInt(10);
248         }
249     }
250 }
251 }
252
253     public static void main(String[] args) {
254         long startTime, endTime;
255         float res1,res2;
256         Scanner scan = new Scanner(System.in);
257         System.out.print("Donner la taille des matrices n :");
258         int N = scan.nextInt();
259         int [][] A = new int[N][N];
260         int [][] B = new int[N][N];
261         int [][] C = new int[N][N];
262         int [][] D = new int[N][N];
263
264
265         lire(A,B);
266
267
268         System.out.println("Matrix A : ");
269
270         affiche(A);
271         System.out.println("Matrix B : ");
272
273         affiche(B);
274
275
276         startTime = System.nanoTime();
277         C=multiplication(A,B);
278         endTime = System.nanoTime();
279         res1 = (float) (endTime - startTime) / 1000000;
280
281
282         System.out.println("Resultat de la multiplication par la methode
283             classique : ");
284
285         affiche(C);
```

```
285         startTime = System.nanoTime();
286         D = strassen(A,B);
287         endTime = System.nanoTime();
288         res2 = (float) (endTime - startTime) / 1000000;
289
290
291 System.out.println("Resultat de la multiplication par la methode de
    Strassen : ");
292
293         affiche(D);
294
295 System.out.println("Nombre de multiplications avec la solution
    classique : "+cpt1);
296 System.out.println("Nombre de multiplications avec la solution de
    Strassen : "+cpt2);
297 System.out.println("Temps de multiplication classique :"+ res1 + "
    ms");
298 System.out.println("Temps de multiplication par Strassen :"+ res2 +
    " ms");
299
300
301
302 }
303
304 }
```

# 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.