Name	**Advait Ravi Sapkal**
UID No.	**2021700055**
Experiment No.	**3**

AIM:	To simulate Strassen's Matrix Multiplication algorithm for square matrices of size 2 x 2
Program 1	
PROBLEM STATEMENT:	Given 2 matrices of order 2 x 2 each, find their product with an algorithm that has a complexity less than $\Theta(N^2)$. The algorithm to be used is Strassen's Matrix Multiplication algorithm.
ALGORITHM:	**Strassen's Matrix Multiplication Algorithm** **Invariant:** 1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE. 2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time. 3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$. 4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time. **Algorithm:**

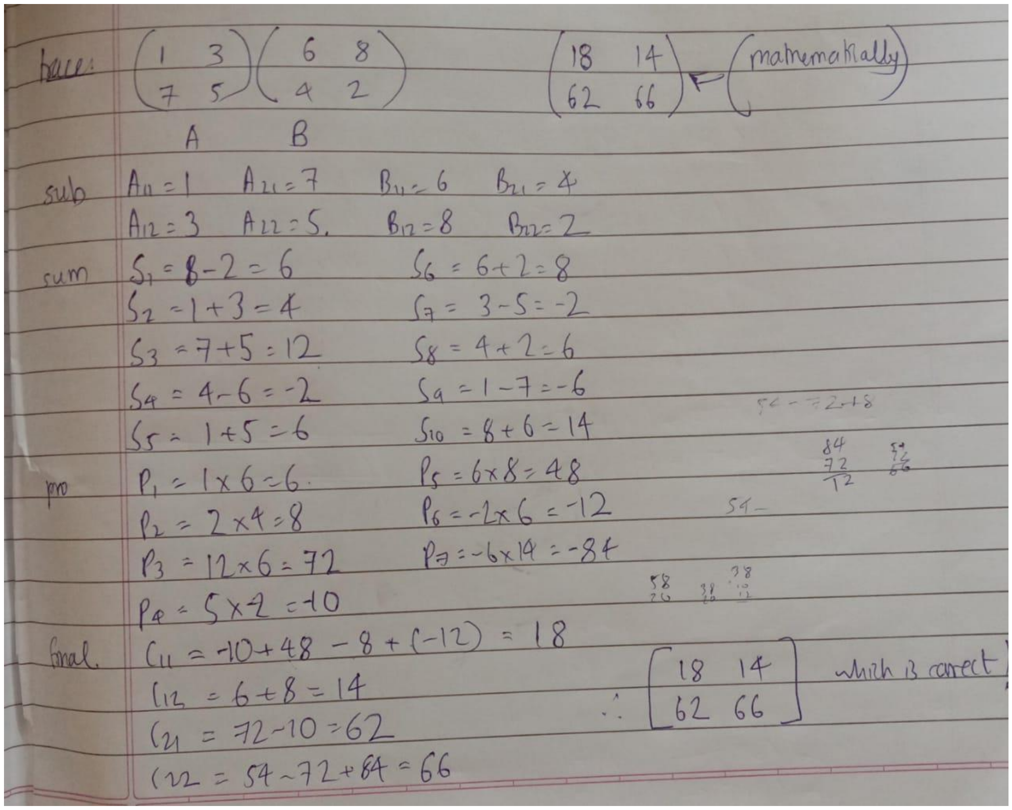
```
SQUARE-MATRIX-MULTIPLY-STRASSEN (A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C
    let S1, S2, ..., and S10 be 10 new n/2 × n/2 matrices
    let P1, P2, ..., and P7 be 7 new n/2 × n/2 matrices

    // calculate the sum matrices
    S1 = B12 − B22
    S2 = A11 + A12
    S3 = A21 + A22
    S4 = B21 − B11
    S5 = A11 + A22
    S6 = B11 + B22
    S7 = A12 − A22
    S8 = B21 + B22
    S9 = A11 − A21
    S10 = B11 + B12

    // calculate the product matrices
    P1 = SQUARE-MATRIX-MULTIPLY-STRASSEN(A11, S1)
    P2 = SQUARE-MATRIX-MULTIPLY-STRASSEN(S2, B22)
    P3 = SQUARE-MATRIX-MULTIPLY-STRASSEN(S3, B11)
    P4 = SQUARE-MATRIX-MULTIPLY-STRASSEN(A22, S4)
    P5 = SQUARE-MATRIX-MULTIPLY-STRASSEN(S5, S6)
    P6 = SQUARE-MATRIX-MULTIPLY-STRASSEN(S7, S8)
    P7 = SQUARE-MATRIX-MULTIPLY-STRASSEN(S9, S10)

    // calculate the final product sub matrices
    C11 = P4 + P5 + P6 − P2
    C12 = P1 + P2
    C21 = P3 + P4
    C22 = P1 + P5 − P3 − P7
return C
```

Trace:



Code:

```
//strassens matrix multiplication algorithm demo
#include<stdio.h>

//for printing a 2 x 2 matrix
void print_matrix(int mat1[2][2]){
    for(int i=0;i<2;i++){
    for(int j=0;j<2;j++){
        printf(" %d",mat1[i][j]);
     }
        printf(" \n");
    }
}

int main(){

    //initializing
    int A[2][2]={    {3,4},
                     {1,2}   };

    int B[2][2]={    {2,0},
```

```
                    {0,2}   };

    //display
    print_matrix(A);   printf("  X\n");   print_matrix(B);

    //sum matrices from sub matrices
    int S[11];
    S[ 1]=B[0][1]-B[1][1];  S[ 6]=B[0][0]+B[1][1];
    S[ 2]=A[0][0]-A[0][1];  S[ 7]=A[0][1]-A[1][1];
    S[ 3]=A[1][0]-A[1][1];  S[ 8]=B[1][0]+B[1][1];
    S[ 4]=B[1][0]-B[0][0];  S[ 9]=A[0][0]-A[1][0];
    S[ 5]=A[0][0]-A[1][1];  S[10]=B[0][0]+B[1][1];

    //product matrices from sum matrices
    int P[8];
    P[1]=A[0][0]*S[1];    P[5]=S[5]*S[6];
    P[2]=B[1][1]*S[2];    P[6]=S[7]*S[8];
    P[3]=B[0][0]*S[3];    P[7]=S[9]*S[10];
    P[4]=A[1][1]*S[4];

    //final matrices from product matrices
    int ret[2][2];
    ret[0][0]=P[5]+P[4]-P[2]+P[6];
    ret[0][1]=P[1]+P[2];
    ret[1][0]=P[3]+P[4];
    ret[1][1]=P[1]+P[5]-P[3]-P[7];

    //display
    printf("  =\n");
    print_matrix(ret);

}
```

Analysis:

analysis:

the recurrence relation is:

$$T(N) = 7T(N/2) + \theta(n^2)$$

now, by master method,

$$n^{\log_b a - \varepsilon} = n^{\log_2 7 - 0} = n^{\lg 7}$$

$$f(n) = n^2$$

now, n^2 is polynomically slower than $n^{\lg 7} \approx (n^{2.81})$

hence $T(n) = \theta(n^{\lg 7}) = \theta(n^{2.81})$

which is less than the naive method $\theta(n^3)$

RESULT:

Outputs

```
d:\DAA\expt3>
  3   4
  1   2
 X
  2   0
  0   2
 =
  6   8
  2   4
```

```
d:\DAA\expt3>
  1   3
  7   5
 X
  6   8
  4   2
 =
 18  14
 62  66
```

```
d:\DAA\expt3>
  3   4
  1   2
 X
  0   0
  0   0
 =
  0   0
  0   0
```

```
d:\DAA\expt3>
  3   4
  1   2
 X
  1   0
  0   1
 =
  3   4
  1   2
```

Conclusions:

1. There are 7 effective matrix multiplications compared to the 8 in naïve multiplication. This is because we converted the cost of the 8^{th} multiplication from a recurrence cost to an asymptotic cost. Due to this, the coefficient on the recurrence relation was reduced, and the one in the asymptotic cost increased, but it will anyways get subsumed.
2. The algorithm is only slightly better than the naïve one, if we compare lg7 with 3, the difference is only 0.20.
3. The algorithm is not much effective for practical use due to various reasons like accumulating memory of the submatrices, higher constant factor, ineffectiveness on sparse matrices, etc.
4. The rounding errors in SMM get accumulated and result in not a very precise

	answer. 5. Although useful for larger matrices, for sparse or small matrices, the naïve method performs better. 6. The code is too complex, requiring >50 lines even in languages like Python.