Name	Aryaman Agarwal
UID no.	2021700002
Experiment No.	3

AIM:	Experiment based on Strassen’s Matrix Multiplication
Program 1	
PROBLEM STATEMENT :	Write C code to perform matrix multiplication using strassen’s matrix multiplication.
ALGORITHM/ THEORY	Let us consider two matrices X and Y. We want to calculate the resultant matrix Z by multiplying X and Y. Naïve Method First, we will discuss naïve method and its complexity. Here, we are calculating Z = X × Y. Using Naïve method, two matrices (X and Y) can be multiplied if the order of these matrices are p × q and q × r. Following is the algorithm. Algorithm: Matrix-Multiplication (X, Y, Z) for i = 1 to p do for j = 1 to r do Z[i,j] := 0 for k = 1 to q do Z[i,j] := Z[i,j] + X[i,k] × Y[k,j] Complexity Here, we assume that integer operations take O(1) time. There are three for loops in this algorithm and one is nested in other. Hence, the algorithm takes O(n3) time to execute. Strassen’s Matrix Multiplication Algorithm In this context, using Strassen’s Matrix multiplication algorithm, the time consumption can be improved a little bit. Strassen’s Matrix multiplication can be performed only on square matrices where n is a power of 2. Order of both of the matrices are n × n. Divide X, Y and Z into four (n/2)×(n/2) matrices as represented below − Z=[IKJL] X=[ACBD] and Y=[EGFH]

	Using Strassen’s Algorithm compute the following − M1:=(A+C)×(E+F) M2:=(B+D)×(G+H) M3:=(A−D)×(E+H) M4:=A×(F−H) M5:=(C+D)×(E) M6:=(A+B)×(H) M7:=D×(G−E) Then, I:=M2+M3−M6−M7 J:=M4+M6 K:=M5+M7 L:=M1−M3−M4−M5 Analysis T(n)={c7xT(n2)+dxn2ifn=1otherwise where c and d are constants Using this recurrence relation, we get T(n)=O(nlog7) Hence, the complexity of Strassen’s matrix multiplication algorithm is O(nlog7

* Strassen's Matrix Multiplication

$$\Rightarrow A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 \end{bmatrix}$$

$$A_{11} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}, \quad A_{12} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 3 & 3 \\ 2 & 2 \end{bmatrix}, \quad A_{22} = \begin{bmatrix} 3 & 3 \\ 2 & 2 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}, \quad B_{12} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$$

$$B_{21} = \begin{bmatrix} 3 & 3 \\ 2 & 2 \end{bmatrix}, \quad B_{22} = \begin{bmatrix} 3 & 3 \\ 2 & 2 \end{bmatrix}$$

$$P = (A_{11} + A_{22}) * (B_{11} + B_{22})$$

$$= \left(\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 3 \\ 2 & 2 \end{bmatrix} \right) * \left(\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 3 \\ 2 & 2 \end{bmatrix} \right)$$

$$= \underset{A}{\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}} * \underset{B}{\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}}$$

$$P' = (A_{11} + A_{22}) * (B_{11} + B_{22})$$
$$= 8 * 8$$
$$= \underline{\underline{64}}$$

$$Q' = (A_{21} + A_{22}) * B_{11}$$
$$(4 + 4) * 4 = \underline{\underline{32}}$$

$$S' = A_{22} * (B_{21} - B_{11})$$
$$= 4 * (4 - 4) = 0$$

$$T' = (A_{11} + A_{12}) * B_{22}$$
$$= 32.$$

$$U' = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$= 0.$$

$$V' = (A_{12} - A_{22}) * (A_{21} + A_{22})$$
$$= 0.$$

$$C_{11}' = P' + S' - T' + V'$$
$$= 64 + 0 - 32 + 0$$
$$= 32.$$

$$C_{12}' = R' + T'$$
$$= 0 + 32 = 32.$$

$$C_{21}' = Q' + S'$$
$$= 32 + 0$$
$$= 32$$

$$C_{22}' = P' + R' - Q' - U'$$
$$= 64 + 0 - 32 - 0$$
$$= 32$$

Similarly, by dividing & using recursion, we can use matrix multiplication.

* Recursion relation for strassen's multiplication:

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T(n/2) + n^2 & n \geq 2. \end{cases}$$

$$T(n) = 7T(n/2) + n^2$$

$a = 7$, $b = 2$, $k = 2$, $p = 0$.

$$\log_b a = \log_2 7 = 2.81 > k.$$

By case 1,

$$O(n^{\log_2 7}) \text{ or } O(n^{2.81})$$

PROGRAM:	(code below)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 8

void add(int **a, int **b, int size, int **c)
{
	int i, j;
	for (i = 0; i < size; i++)
	{
		for (j = 0; j < size; j++)
		{
			c[i][j] = a[i][j] + b[i][j];
		}
	}
}

void sub(int **a, int **b, int size, int **c)
{
	int i, j;
	for (i = 0; i < size; i++)
	{
		for (j = 0; j < size; j++)
		{
			c[i][j] = a[i][j] - b[i][j];
		}
	}
}
void multiply(int **c, int **d, int size, int size2, int **new)
{
	if (size == 1)
	{
	//if there is only one element in matrixnew[0][0] =
		c[0][0] * d[0][0];
	}
	else
	{
		int i, j;
		//dividing the matrix into 4 matrixint nsize =
		size / 2;

		//space for part-1 of the matrix A
		int **c11 = malloc(nsize * sizeof(int *));for (i = 0; i <
		nsize; i++)
		{
```

| | ```
        c11[i] = malloc(nsize * sizeof(int));
}

//space for part-2 of the matrix A
int **c12 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        c12[i] = malloc(nsize * sizeof(int));
}

//space for part-3 of the matrix A
int **c21 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        c21[i] = malloc(nsize * sizeof(int));
}

//space for part-4 of the matrix A
int **c22 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        c22[i] = malloc(nsize * sizeof(int));
}

//space for part-1 of the matrix B
int **d11 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        d11[i] = malloc(nsize * sizeof(int));
}

//space for part-2 of the matrix B
int **d12 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        d12[i] = malloc(nsize * sizeof(int));
}

//space for part-3 of the matrix B
int **d21 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        d21[i] = malloc(nsize * sizeof(int));
}
``` |
```

```
//space for part-4 of the matrix B
int **d22 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        d22[i] = malloc(nsize * sizeof(int));
}

//allocating spaces for strassens formulas of matrixmultiplication
int **m1 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        m1[i] = malloc(nsize * sizeof(int));
}
int **m2 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        m2[i] = malloc(nsize * sizeof(int));
}
int **m3 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        m3[i] = malloc(nsize * sizeof(int));
}
int **m4 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        m4[i] = malloc(nsize * sizeof(int));
}
int **m5 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        m5[i] = malloc(nsize * sizeof(int));
}
int **m6 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        m6[i] = malloc(nsize * sizeof(int));
}
int **m7 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        m7[i] = malloc(nsize * sizeof(int));
```

	(code below)

```
}


for (i = 0; i < nsize; i++)
{
        for (j = 0; j < nsize; j++)
        {
                //top-left half of the matrixc11[i][j] =
                c[i][j];

                //top-right half of the matrixc12[i][j] =
                c[i][j + nsize];

                //bottom-left half of the matrixc21[i][j] =
                c[i + nsize][j];

                //bottom-right half of the matrix c22[i][j] = c[i
                + nsize][j + nsize];

                d11[i][j] = d[i][j];
                d12[i][j] = d[i][j + nsize];
                d21[i][j] = d[i + nsize][j]; d22[i][j] = d[i +
                nsize][j + nsize];
        }
}

//creating 2 matrix for temporary storage int **temp1 =
malloc(nsize * sizeof(int *));for (i = 0; i < nsize; i++)
{
        temp1[i] = malloc(nsize * sizeof(int));
}
int **temp2 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
        temp2[i] = malloc(nsize * sizeof(int));
}


//first formula
//P = (A11 + A22)*(B11 + B22)
add(c11, c22, nsize, temp1);
add(d11, d22, nsize, temp2);
multiply(temp1, temp2, nsize, size, m1);
```

```
free(temp1);
free(temp2);


int **temp3 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp3[i] = malloc(nsize * sizeof(int));
}
//Q = (A21 + A22) * B11
add(c21, c22, nsize, temp3); multiply(temp3,
d11, nsize, size, m2);free(temp3);

int **temp4 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp4[i] = malloc(nsize * sizeof(int));
}
//R = A11 * (B12 - B22)
sub(d12, d22, nsize, temp4); multiply(c11,
temp4, nsize, size, m3);free(temp4);

int **temp5 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp5[i] = malloc(nsize * sizeof(int));
}
//S = A22 * (B21 - B11)
sub(d21, d11, nsize, temp5); multiply(c22,
temp5, nsize, size, m4);free(temp5);

int **temp6 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp6[i] = malloc(nsize * sizeof(int));
}
//T = (A11 + A12) * B22
add(c11, c12, nsize, temp6); multiply(temp6,
d22, nsize, size, m5);free(temp6);
```

```
int **temp7 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp7[i] = malloc(nsize * sizeof(int));
}
int **temp8 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp8[i] = malloc(nsize * sizeof(int));
}
//U = (A21 - A11) * (B11 + B12)
sub(c21, c11, nsize, temp7);
add(d11, d12, nsize, temp8);
multiply(temp7, temp8, nsize, size, m6);
free(temp7);
free(temp8);

int **temp9 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp9[i] = malloc(nsize * sizeof(int));
}
int **temp10 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	temp10[i] = malloc(nsize * sizeof(int));
}
//V = (A12 - A22)*(B21 + B22)
sub(c12, c22, nsize, temp9);
add(d21, d22, nsize, temp10);
multiply(temp9, temp10, nsize, size, m7);
free(temp9);
free(temp10);


//matrices for temporary storage
int **te1 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
	te1[i] = malloc(nsize * sizeof(int));
}
int **te2 = malloc(nsize * sizeof(int *));for (i = 0; i <
nsize; i++)
{
```

	`` te2[i] = malloc(nsize * sizeof(int)); } int **te3 = malloc(nsize * sizeof(int *));for (i = 0; i < nsize; i++) { te3[i] = malloc(nsize * sizeof(int)); } int **te4 = malloc(nsize * sizeof(int *));for (i = 0; i < nsize; i++) { te4[i] = malloc(nsize * sizeof(int)); } int **te5 = malloc(nsize * sizeof(int *));for (i = 0; i < nsize; i++) { te5[i] = malloc(nsize * sizeof(int)); } int **te6 = malloc(nsize * sizeof(int *));for (i = 0; i < nsize; i++) { te6[i] = malloc(nsize * sizeof(int)); } int **te7 = malloc(nsize * sizeof(int *));for (i = 0; i < nsize; i++) { te7[i] = malloc(nsize * sizeof(int)); } int **te8 = malloc(nsize * sizeof(int *));for (i = 0; i < nsize; i++) { te8[i] = malloc(nsize * sizeof(int)); } add(m1, m7, nsize, te1); sub(m4, m5, nsize, te2); add(te1, te2, nsize, te3); // C11 add(m3, m5, nsize, te4); // C12 add(m2, m4, nsize, te5); // C21 add(m3, m6, nsize, te6); sub(m1, m2, nsize, te7); add(te6, te7, nsize, te8); // C22 ```

```
int a = 0;
int b = 0;
int c = 0;
int d = 0;
int e = 0;
int nsize2 = 2 * nsize;
for (i = 0; i < nsize2; i++)
{
	for (j = 0; j < nsize2; j++)
	{
		//C11
		if (j >= 0 && j < nsize && i >= 0 && i < nsize)
		{
			new[i][j] = te3[i][j];
		}

		//C12
			if (j >= nsize && j < nsize2 && i >= 0 && i <
nsize)
		{
			a = j - nsize;
			new[i][j] = te4[i][a];
		}

		//C21
			if (j >= 0 && j < nsize && i >= nsize && i <
nsize2)
		{
			c = i - nsize;
			new[i][j] = te5[c][j];
		}

		//C22
		if (j >= nsize && j < nsize2 && i >= nsize && i <
nsize2)
		{
			d = i - nsize;
			e = j - nsize;
			new[i][j] = te8[d][e];
		}
	}
}
```

```
		//deallocating all the spaces
		free(m1);
		free(m2);
		free(m3);
		free(m4);
		free(m5);
		free(m6);
		free(m7);
		free(te1);
		free(te2);
		free(te3);
		free(te4);
		free(te5);
		free(te6);
		free(te7);
		free(te8);
		free(c11);
		free(c12);
		free(c21);
		free(c22);
		free(d11);
		free(d12);
		free(d21);
		free(d22);
	}
}

void main()
{
	int  size,  itr,  itr1,  i,  j,  nsize;printf("Enter
	Size of matrix\n");scanf("%d", &size);

	int tempS = size;

	//allocating space for matrix A
	int **a = malloc(size * sizeof(int *));for (i = 0; i <
	size; i++)
	{
		a[i] = malloc(size * sizeof(int));
	}

	//allocating space for matrix B
	int **b = malloc(size * sizeof(int *));
```

| | ```
for (i = 0; i < size; i++)
{
        b[i] = malloc(size * sizeof(int));
}

//Taking inputs for matrix A printf("Enter elements
of matrix A:\n");for (itr = 0; itr < size; itr++)
{
        for (itr1 = 0; itr1 < size; itr1++)
        {
                scanf("%d", &a[itr][itr1]);
        }
}

//Taking inputs for matrix B printf("Enter elements
of matrix B:\n");for (itr = 0; itr < size; itr++)
{
        for (itr1 = 0; itr1 < size; itr1++)
        {
                scanf("%d", &b[itr][itr1]);
        }
}

//allocating space to store result of multiplicationint **new =
malloc(size * sizeof(int *));
for (i = 0; i < size; i++)
{
        new[i] = malloc(size * sizeof(int));
}

//strassens multiplication multiply(a, b,
size, size, new);

printf("Matrix   C:\n"); for (i = 0; i
< size; i++)
{
        for (j = 0; j < size; j++)
        {
                printf("%d          ", new[i][j]);
        }
        printf("\n");
}
``` |
```

```
}
```

RESULT:

```
PS D:\BTECH\SEM-4\DAA\Practicals\Exp3> ./main.exe
Enter Size of matrix
4
Enter elements of matrix A:
1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
Enter elements of matrix B:
1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
Matrix C:
10   10   10   10
20   20   20   20
30   30   30   30
40   40   40   40
PS D:\BTECH\SEM-4\DAA\Practicals\Exp3>
```

CONCLUSION :	• Implemented Strassen's Matrix multiplication using C language. • Time Complexity(all cases): O(n^2.81)