DESIGN AND ANALYSIS OF ALGORITHMS LAB

NAME: Mohammed Saad Belgi

UID: 2021700005

BATCH: A

BRANCH: CSE DS

EXPT. NO.: 3

AIM: Experiment based on divide and conquer approach: Strassen's Matrix Multiplication

ALGORITHM:

STRASSENS-MULTIPLICATION (A, B):

1. n = A.rows
2. Let C be a new n X n matrix
3. if n == 2:
    a. P1 ← A11 x (B12 – B22)
    b. P2 ← (A11 + A12) x B22
    c. P3 ← (A21 + A22) x B21
    d. P4 ← A22 x (B21 – B11)
    e. P5 ← (A11 + A22) x (B11 + B22)
    f. P6 ← (A12 – A22) x (B21 + B22)
    g. P7 ← (A11 – A21) x (B11 + B12)
    h. C11 ← P5 + P4 – P2 + P6
    i. C12 ← P1 + P2
    j. C21 ← P3 + P4
    k. C22 ← P5 + P1 – P3 – P7
    l. return C
4. Divide input matrices A and B and output matrix C into 4 submatrices of size n/2 X n/2 each as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

5. P1 ← STRASSENS-MULTIPLICATION(A11, (B12 – B22))
6. P2 ← STRASSENS-MULTIPLICATION(A11 + A12, B22)
7. P3 ← STRASSENS-MULTIPLICATION(A21 + A22, B21)
8. P4 ← STRASSENS-MULTIPLICATION(A22, B21 – B11)
9. P5 ← STRASSENS-MULTIPLICATION(A11 + A22, B11 + B22)
10. P6 ← STRASSENS-MULTIPLICATION(A12 – A22, B21 + B22)
11. P7 ← STRASSENS-MULTIPLICATION(A11 – A21, B11 + B12)
12. C11 ← P5 + P4 – P2 + P6
13. C12 ← P1 + P2
14. C21 ← P3 + P4

15. C22 ← P5 + P1 – P3 – P7
16. return C

CODE:

```
#include <stdio.h>
#include <stdlib.h>

// prototypes
int **addSquareMatrices(int **a, int **b, int n, int a_p, int a_q, int
b_p, int b_q);
int **subtractSquareMatrices(int **a, int **b, int n, int a_p, int a_q,
int b_p, int b_q);
int **strassensMultiplication(int **a, int **b, int n);
int **actualStrassensMultiplication(int **a, int **b, int n, int a_p, int
a_q, int b_p, int b_q);
int **mallocSqaureMatrix(int n);
void freeSquareMatrix(int **mat, int n);

int **mallocSqaureMatrix(int n)
{
    int **new = malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++)
        new[i] = malloc(n * sizeof(int));
    return new;
}

void freeSquareMatrix(int **mat, int n)
{
    for (int i = 0; i < n; i++)
        free(mat[i]);
    free(mat);
}

int **addSquareMatrices(int **a, int **b, int n, int a_p, int a_q, int
b_p, int b_q)
{
    int **sum = mallocSqaureMatrix(n);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            sum[i][j] = a[a_p + i][a_q + j] + b[b_p + i][b_q + j];
    }
    return sum;
}

int **subtractSquareMatrices(int **a, int **b, int n, int a_p, int a_q,
int b_p, int b_q)
{
    int **diff = mallocSqaureMatrix(n);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            diff[i][j] = a[a_p + i][a_q + j] - b[b_p + i][b_q + j];
```

```c
    }
    return diff;
}

int **strassensMultiplication(int **a, int **b, int n)
{
    return actualStrassensMultiplication(a, b, n, 0, 0, 0, 0);
}

int **actualStrassensMultiplication(int **a, int **b, int n, int a_p, int
a_q, int b_p, int b_q)
{
    int **prod = mallocSqaureMatrix(n);
    if (n == 2)
    {
        int p1 = a[a_p][a_q] * (b[b_p][b_q + 1] - b[b_p + 1][b_q + 1]);
        int p2 = (a[a_p][a_q] + a[a_p][a_q + 1]) * b[b_p + 1][b_q + 1];
        int p3 = (a[a_p + 1][a_q] + a[a_p + 1][a_q + 1]) * b[b_p][b_q];
        int p4 = a[a_p + 1][a_q + 1] * (b[b_p + 1][b_q] - b[b_p][b_q]);
        int p5 = (a[a_p][a_q] + a[a_p + 1][a_q + 1]) * (b[b_p][b_q] +
b[b_p + 1][b_q + 1]);
        int p6 = (a[a_p][a_q + 1] - a[a_p + 1][a_q + 1]) * (b[b_p +
1][b_q] + b[b_p + 1][b_q + 1]);
        int p7 = (a[a_p][a_q] - a[a_p + 1][a_q]) * (b[b_p][b_q] +
b[b_p][b_q + 1]);
        prod[0][0] = p5 + p4 - p2 + p6;
        prod[0][1] = p1 + p2;
        prod[1][0] = p3 + p4;
        prod[1][1] = p5 + p1 - p3 - p7;
    }
    else
    {
        int x = n / 2;
        int **temp = subtractSquareMatrices(b, b, x, b_p, b_q + x, b_p +
x, b_q + x);
        int **p1 = actualStrassensMultiplication(a, temp, x, a_p, a_q, 0,
0);
        freeSquareMatrix(temp, x);
        temp = addSquareMatrices(a, a, x, a_p, a_q, a_p, a_q + x);
        int **p2 = actualStrassensMultiplication(temp, b, x, 0, 0, b_p +
x, b_q + x);
        freeSquareMatrix(temp, x);
        temp = addSquareMatrices(a, a, x, a_p + x, a_q, a_p + x, a_q + x);
        int **p3 = actualStrassensMultiplication(temp, b, x, 0, 0, b_p,
b_q);
        freeSquareMatrix(temp, x);
        temp = subtractSquareMatrices(b, b, x, b_p + x, b_q, b_p, b_q);
        int **p4 = actualStrassensMultiplication(a, temp, x, a_p + x, a_q
+ x, 0, 0);
        freeSquareMatrix(temp, x);
        temp = addSquareMatrices(a, a, x, a_p, a_q, a_p + x, a_q + x);
        int **temp2 = addSquareMatrices(b, b, x, b_p, b_q, b_p + x, b_q +
x);
```

```
        int **p5 = actualStrassensMultiplication(temp, temp2, x, 0, 0, 0,
0);
        freeSquareMatrix(temp, x);
        freeSquareMatrix(temp2, x);
        temp = subtractSquareMatrices(a, a, x, a_p, a_q + x, a_p + x, a_q
+ x);
        temp2 = addSquareMatrices(b, b, x, b_p + x, b_q, b_p + x, b_q +
x);
        int **p6 = actualStrassensMultiplication(temp, temp2, x, 0, 0, 0,
0);
        freeSquareMatrix(temp, x);
        freeSquareMatrix(temp2, x);
        temp = subtractSquareMatrices(a, a, x, a_p, a_q, a_p + x, a_q);
        temp2 = addSquareMatrices(b, b, x, b_p, b_q, b_p, b_q + x);
        int **p7 = actualStrassensMultiplication(temp, temp2, x, 0, 0, 0,
0);
        freeSquareMatrix(temp, x);
        freeSquareMatrix(temp2, x);

        temp = addSquareMatrices(p5, p4, x, 0, 0, 0, 0);
        temp2 = addSquareMatrices(temp, p6, x, 0, 0, 0, 0);
        freeSquareMatrix(temp, x);
        temp = subtractSquareMatrices(temp2, p2, x, 0, 0, 0, 0);
        freeSquareMatrix(temp2, x);
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < x; j++)
                prod[i][j] = temp[i][j];
        }
        freeSquareMatrix(temp, x);
        temp = addSquareMatrices(p1, p2, x, 0, 0, 0, 0);
        for (int i = 0; i < x; i++)
        {
            for (int j = x; j < n; j++)
                prod[i][j] = temp[i][j - x];
        }
        freeSquareMatrix(temp, x);
        temp = addSquareMatrices(p3, p4, x, 0, 0, 0, 0);
        for (int i = x; i < n; i++)
        {
            for (int j = 0; j < x; j++)
                prod[i][j] = temp[i - x][j];
        }
        freeSquareMatrix(temp, x);
        temp = addSquareMatrices(p5, p1, x, 0, 0, 0, 0);
        temp2 = subtractSquareMatrices(temp, p3, x, 0, 0, 0, 0);
        freeSquareMatrix(temp, x);
        temp = subtractSquareMatrices(temp2, p7, x, 0, 0, 0, 0);
        for (int i = x; i < n; i++)
        {
            for (int j = x; j < n; j++)
                prod[i][j] = temp[i - x][j - x];
        }
        freeSquareMatrix(temp, x);
```

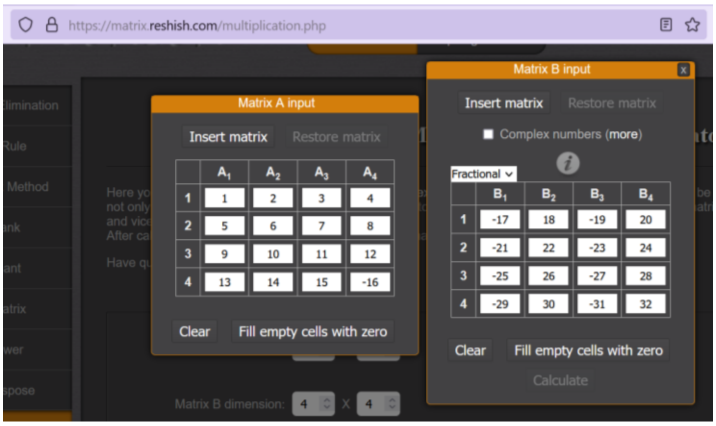
```c
        freeSquareMatrix(temp2, x);
    }
    return prod;
}

int main()
{
    printf("Enter order of input matrices (should be a power of 2): ");
    int n;
    scanf("%d", &n);
    int **a = mallocSqaureMatrix(n);
    int **b = mallocSqaureMatrix(n);
    printf("Enter matrix elements of first matrix in row major order: ");
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
    }
    printf("Enter matrix elements of second matrix in row major order: ");
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            scanf("%d", &b[i][j]);
    }
    printf("Product of first and second matrices:\n");
    int **prod = strassensMultiplication(a, b, n);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            printf("%15d", prod[i][j]);
        printf("\n");
    }
}
```

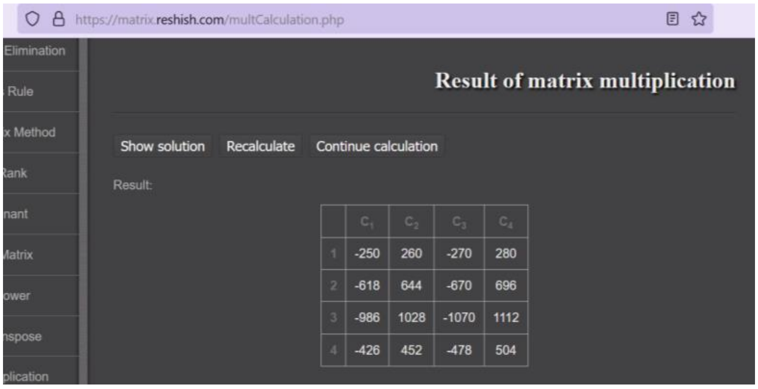
OUTPUT:

Multiplication of two 4x4 matrices:

```
Enter order of input matrices (should be a power of 2): 4
Enter matrix elements of first matrix in row major order: 1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 -16
Enter matrix elements of second matrix in row major order: -17 18 -19 20
-21 22 -23 24
-25 26 -27 28
-29 30 -31 32
Product of first and second matrices:
           -250            260           -270            280
           -618            644           -670            696
           -986           1028          -1070           1112
           -426            452           -478            504
(base) PS C:\Users\arifa\Desktop\sem4 work\daa lab\exp3>
```

Verifying result using this matrix multiplier website:



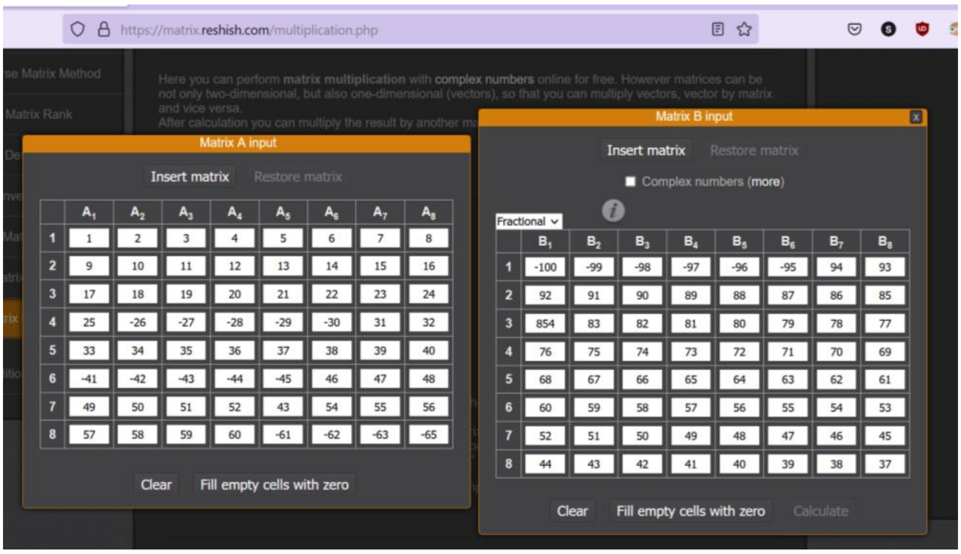


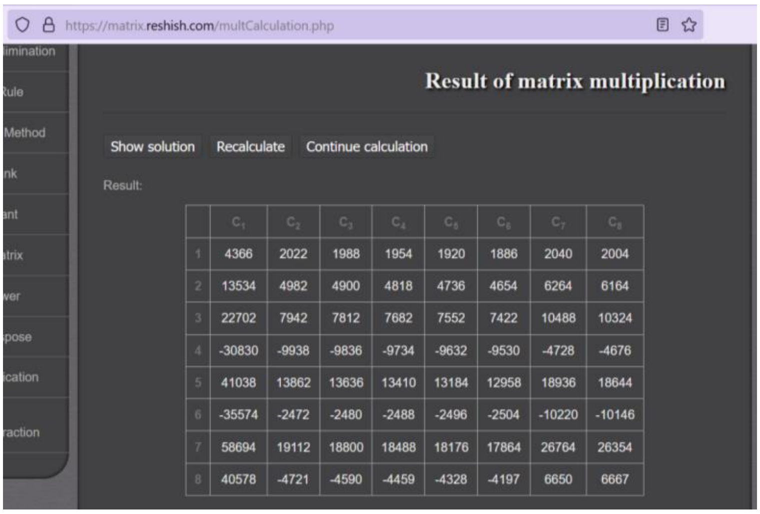
	C_1	C_2	C_3	C_4
1	-250	260	-270	280
2	-618	644	-670	696
3	-986	1028	-1070	1112
4	-426	452	-478	504

Multiplication of two 8x8 matrices:

```
Enter order of input matrices (should be a power of 2): 8
Enter matrix elements of first matrix in row major order: 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
25 -26 -27 -28 -29 -30 31 32
33 34 35 36 37 38 39 40
-41 -42 -43 -44 -45 46 47 48
49 50 51 52 43 54 55 56
57 58 59 60 -61 -62 -63 -65
Enter matrix elements of second matrix in row major order: -100 -99 -98 -97 -96 -95 94 93
92 91 90 89 88 87 86 85
854 83 82 81 80 79 78 77
76 75 74 73 72 71 70 69
68 67 66 65 64 63 62 61
60 59 58 57 56 55 54 53
52 51 50 49 48 47 46 45
44 43 42 41 40 39 38 37
Product of first and second matrices:
         4366          2022          1988          1954          1920          1886          2040          2004
        13534          4982          4900          4818          4736          4654          6264          6164
        22702          7942          7812          7682          7552          7422         10488         10324
       -30830         -9938         -9836         -9734         -9632         -9530         -4728         -4676
        41038         13862         13636         13410         13184         12958         18936         18644
       -35574         -2472         -2480         -2488         -2496         -2504        -10220        -10146
        58694         19112         18800         18488         18176         17864         26764         26354
        40578         -4721         -4590         -4459         -4328         -4197          6650          6667
(base) PS C:\Users\arifa\Desktop\sem4 work\daa lab\exp3>
```

Verification of result:





CONCLUSION:

Strassen's matrix multiplication is a divide and conquer algorithm, which uses 7 recursive calls to itself to perform matrix multiplication in asymptotically less time than normal matrix multiplication.