**PRACTICAL = 3 (d)**

MATRIX MULTIPLICATION COMPARISION:

**C CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void printMatrix(int rows, int cols, int \*\*matrix) {

    if (rows <= 10 && cols <= 10) {

        for (int i = 0; i < rows; i++) {

            for (int j = 0; j < cols; j++) {

                printf("%d ", matrix[i][j]);

            }

            printf("\n");

        }

    } else {

        printf("Matrix is too large to display.\n");

    }

}

int\*\* allocateMatrix(int rows, int cols) {

    int\*\* matrix = (int\*\*)malloc(rows \* sizeof(int));

    for (int i = 0; i < rows; i++) {

        matrix[i] = (int\*)malloc(cols \* sizeof(int));

    }

    return matrix;

}

void freeMatrix(int\*\* matrix, int rows) {

    if (matrix == NULL) return;

    for (int i = 0; i < rows; i++) {

        free(matrix[i]);

    }

    free(matrix);

}

void generateRandomMatrix(int rows, int cols, int\*\* matrix) {

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            matrix[i][j] = rand() % 10;

        }

    }

}

void addMatrix(int\*\* A, int\*\* B, int\*\* result, int size) {

    for (int i = 0; i < size; i++) {

        for (int j = 0; j < size; j++) {

            result[i][j] = A[i][j] + B[i][j];

        }

    }

}

void subMatrix(int\*\* A, int\*\* B, int\*\* result, int size) {

    for (int i = 0; i < size; i++) {

        for (int j = 0; j < size; j++) {

            result[i][j] = A[i][j] - B[i][j];

        }

    }

}

// Iterative Multiplication

void standardMultiply(int n, int m, int \*\*arr1, int n2, int m2, int \*\*arr2, int \*\*arr\_result) {

    if (m != n2) {

        printf("Error: Matrices cannot be multiplied.\n");

        return;

    }

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < m2; j++) {

            arr\_result[i][j] = 0;

            for (int k = 0; k < m; k++) {

                arr\_result[i][j] += arr1[i][k] \* arr2[k][j];

            }

        }

    }

}

// Divide and Conquer

void divideAndConquerMultiply(int size, int \*\*A, int \*\*B, int \*\*C) {

    if (size == 1) {

        C[0][0] = A[0][0] \* B[0][0];

        return;

    }

    int newSize = size / 2;

    int \*\*A11 = allocateMatrix(newSize, newSize); int \*\*A12 = allocateMatrix(newSize, newSize);

    int \*\*A21 = allocateMatrix(newSize, newSize); int \*\*A22 = allocateMatrix(newSize, newSize);

    int \*\*B11 = allocateMatrix(newSize, newSize); int \*\*B12 = allocateMatrix(newSize, newSize);

    int \*\*B21 = allocateMatrix(newSize, newSize); int \*\*B22 = allocateMatrix(newSize, newSize);

    int \*\*C11 = allocateMatrix(newSize, newSize); int \*\*C12 = allocateMatrix(newSize, newSize);

    int \*\*C21 = allocateMatrix(newSize, newSize); int \*\*C22 = allocateMatrix(newSize, newSize);

    int \*\*temp1 = allocateMatrix(newSize, newSize); int \*\*temp2 = allocateMatrix(newSize, newSize);

    for (int i = 0; i < newSize; i++) {

        for (int j = 0; j < newSize; j++) {

            A11[i][j] = A[i][j]; A12[i][j] = A[i][j + newSize];

            A21[i][j] = A[i + newSize][j]; A22[i][j] = A[i + newSize][j + newSize];

            B11[i][j] = B[i][j]; B12[i][j] = B[i][j + newSize];

            B21[i][j] = B[i + newSize][j]; B22[i][j] = B[i + newSize][j + newSize];

        }

    }

    divideAndConquerMultiply(newSize, A11, B11, temp1);

    divideAndConquerMultiply(newSize, A12, B21, temp2);

    addMatrix(temp1, temp2, C11, newSize);

    divideAndConquerMultiply(newSize, A11, B12, temp1);

    divideAndConquerMultiply(newSize, A12, B22, temp2);

    addMatrix(temp1, temp2, C12, newSize);

    divideAndConquerMultiply(newSize, A21, B11, temp1);

    divideAndConquerMultiply(newSize, A22, B21, temp2);

    addMatrix(temp1, temp2, C21, newSize);

    divideAndConquerMultiply(newSize, A21, B12, temp1);

    divideAndConquerMultiply(newSize, A22, B22, temp2);

    addMatrix(temp1, temp2, C22, newSize);

    for (int i = 0; i < newSize; i++) {

        for (int j = 0; j < newSize; j++) {

            C[i][j] = C11[i][j];

            C[i][j + newSize] = C12[i][j];

            C[i + newSize][j] = C21[i][j];

            C[i + newSize][j + newSize] = C22[i][j];

        }

    }

    freeMatrix(A11, newSize); freeMatrix(A12, newSize); freeMatrix(A21, newSize); freeMatrix(A22, newSize);

    freeMatrix(B11, newSize); freeMatrix(B12, newSize); freeMatrix(B21, newSize); freeMatrix(B22, newSize);

    freeMatrix(C11, newSize); freeMatrix(C12, newSize); freeMatrix(C21, newSize); freeMatrix(C22, newSize);

    freeMatrix(temp1, newSize); freeMatrix(temp2, newSize);

}

// Strassen's Algorithm

void strassenMultiply(int\*\* A, int\*\* B, int\*\* C, int size) {

    if (size == 1) {

        C[0][0] = A[0][0] \* B[0][0];

        return;

    }

    int newSize = size / 2;

    int\*\* A11 = allocateMatrix(newSize, newSize); int\*\* A12 = allocateMatrix(newSize, newSize);

    int\*\* A21 = allocateMatrix(newSize, newSize); int\*\* A22 = allocateMatrix(newSize, newSize);

    int\*\* B11 = allocateMatrix(newSize, newSize); int\*\* B12 = allocateMatrix(newSize, newSize);

    int\*\* B21 = allocateMatrix(newSize, newSize); int\*\* B22 = allocateMatrix(newSize, newSize);

    for (int i = 0; i < newSize; i++) {

        for (int j = 0; j < newSize; j++) {

            A11[i][j] = A[i][j]; A12[i][j] = A[i][j + newSize];

            A21[i][j] = A[i + newSize][j]; A22[i][j] = A[i + newSize][j + newSize];

            B11[i][j] = B[i][j]; B12[i][j] = B[i][j + newSize];

            B21[i][j] = B[i + newSize][j]; B22[i][j] = B[i + newSize][j + newSize];

        }

    }

    int\*\* P = allocateMatrix(newSize, newSize); int\*\* Q = allocateMatrix(newSize, newSize);

    int\*\* R = allocateMatrix(newSize, newSize); int\*\* S = allocateMatrix(newSize, newSize);

    int\*\* T = allocateMatrix(newSize, newSize); int\*\* U = allocateMatrix(newSize, newSize);

    int\*\* V = allocateMatrix(newSize, newSize);

    int\*\* temp1 = allocateMatrix(newSize, newSize); int\*\* temp2 = allocateMatrix(newSize, newSize);

    addMatrix(A11, A22, temp1, newSize);

    addMatrix(B11, B22, temp2, newSize);

    strassenMultiply(temp1, temp2, P, newSize);

    addMatrix(A21, A22, temp1, newSize);

    strassenMultiply(temp1, B11, Q, newSize);

    subMatrix(B12, B22, temp1, newSize);

    strassenMultiply(A11, temp1, R, newSize);

    subMatrix(B21, B11, temp1, newSize);

    strassenMultiply(A22, temp1, S, newSize);

    addMatrix(A11, A12, temp1, newSize);

    strassenMultiply(temp1, B22, T, newSize);

    subMatrix(A21, A11, temp1, newSize);

    addMatrix(B11, B12, temp2, newSize);

    strassenMultiply(temp1, temp2, U, newSize);

    subMatrix(A12, A22, temp1, newSize);

    addMatrix(B21, B22, temp2, newSize);

    strassenMultiply(temp1, temp2, V, newSize);

    int\*\* C11 = allocateMatrix(newSize, newSize); int\*\* C12 = allocateMatrix(newSize, newSize);

    int\*\* C21 = allocateMatrix(newSize, newSize); int\*\* C22 = allocateMatrix(newSize, newSize);

    addMatrix(P, S, temp1, newSize);

    subMatrix(temp1, T, temp2, newSize);

    addMatrix(temp2, V, C11, newSize);

    addMatrix(R, T, C12, newSize);

    addMatrix(Q, S, C21, newSize);

    addMatrix(P, R, temp1, newSize);

    subMatrix(temp1, Q, temp2, newSize);

    addMatrix(temp2, U, C22, newSize);

    for (int i = 0; i < newSize; i++) {

        for (int j = 0; j < newSize; j++) {

            C[i][j] = C11[i][j];

            C[i][j + newSize] = C12[i][j];

            C[i + newSize][j] = C21[i][j];

            C[i + newSize][j + newSize] = C22[i][j];

        }

    }

    freeMatrix(A11, newSize); freeMatrix(A12, newSize); freeMatrix(A21, newSize); freeMatrix(A22, newSize);

    freeMatrix(B11, newSize); freeMatrix(B12, newSize); freeMatrix(B21, newSize); freeMatrix(B22, newSize);

    freeMatrix(C11, newSize); freeMatrix(C12, newSize); freeMatrix(C21, newSize); freeMatrix(C22, newSize);

    freeMatrix(P, newSize); freeMatrix(Q, newSize); freeMatrix(R, newSize); freeMatrix(S, newSize);

    freeMatrix(T, newSize); freeMatrix(U, newSize); freeMatrix(V, newSize);

    freeMatrix(temp1, newSize); freeMatrix(temp2, newSize);

}

// Main Function

int main() {

    int size;

    clock\_t start, end;

    double time\_taken\_iterative, time\_taken\_dc, time\_taken\_strassen;

    srand(time(NULL));

    printf("Enter size of square matrices (must be a power of 2 for D&C and Strassen): ");

    scanf("%d", &size);

    int\*\* A = allocateMatrix(size, size);

    int\*\* B = allocateMatrix(size, size);

    int\*\* C = allocateMatrix(size, size);

    generateRandomMatrix(size, size, A);

    generateRandomMatrix(size, size, B);

    printf("\n--- Iterative Matrix Multiplication ---\n");

    start = clock();

        standardMultiply(size, size, A, size, size, B, C);

    end = clock();

    time\_taken\_iterative = (double)(end - start) / CLOCKS\_PER\_SEC / 1000;

    printf("Resultant Matrix:\n");

    printMatrix(size, size, C);

    printf("Time taken for iterative multiplication: %f seconds\n", time\_taken\_iterative);

    // Ensure the size is a power of 2 for the other two algorithms

    if ((size & (size - 1)) != 0 && size != 1) {

        printf("\nWarning: Matrix size is not a power of 2. Cannot run Divide and Conquer or Strassen's algorithm.\n");

        freeMatrix(A, size);

        freeMatrix(B, size);

        freeMatrix(C, size);

        return 0;

    }

    printf("\n--- Divide and Conquer Matrix Multiplication ---\n");

    start = clock();

        divideAndConquerMultiply(size, A, B, C);

    end = clock();

    time\_taken\_dc = (double)(end - start) / CLOCKS\_PER\_SEC / 1000;

    printf("Resultant Matrix:\n");

    printMatrix(size, size, C);

    printf("Time taken for Divide and Conquer: %f seconds\n", time\_taken\_dc);

    printf("\n--- Strassen's Matrix Multiplication ---\n");

    start = clock();

        strassenMultiply(A, B, C, size);

    end = clock();

    time\_taken\_strassen = (double)(end - start) / CLOCKS\_PER\_SEC / 1000;

    printf("Resultant Matrix:\n");

    printMatrix(size, size, C);

    printf("Time taken for Strassen's algorithm: %f seconds\n", time\_taken\_strassen);

    freeMatrix(A, size);

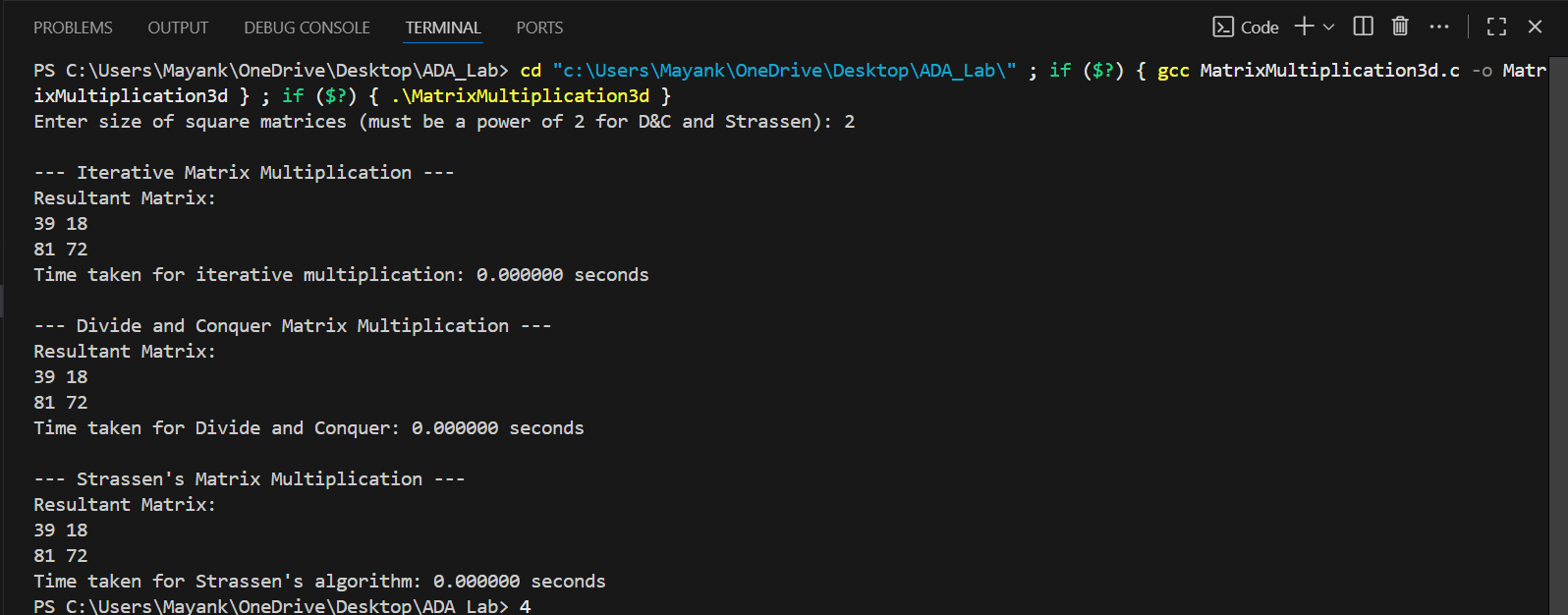
    freeMatrix(B, size);

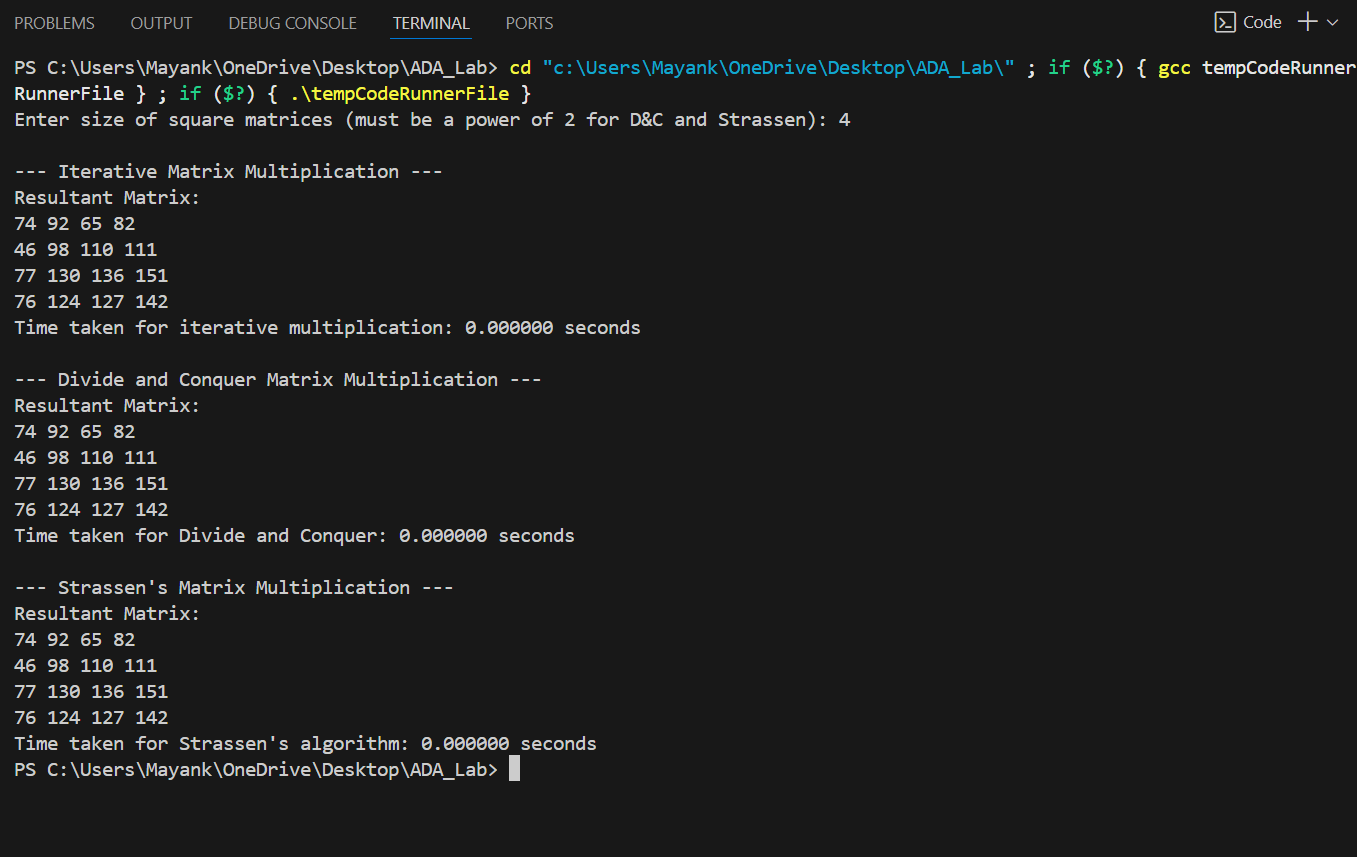
    freeMatrix(C, size);

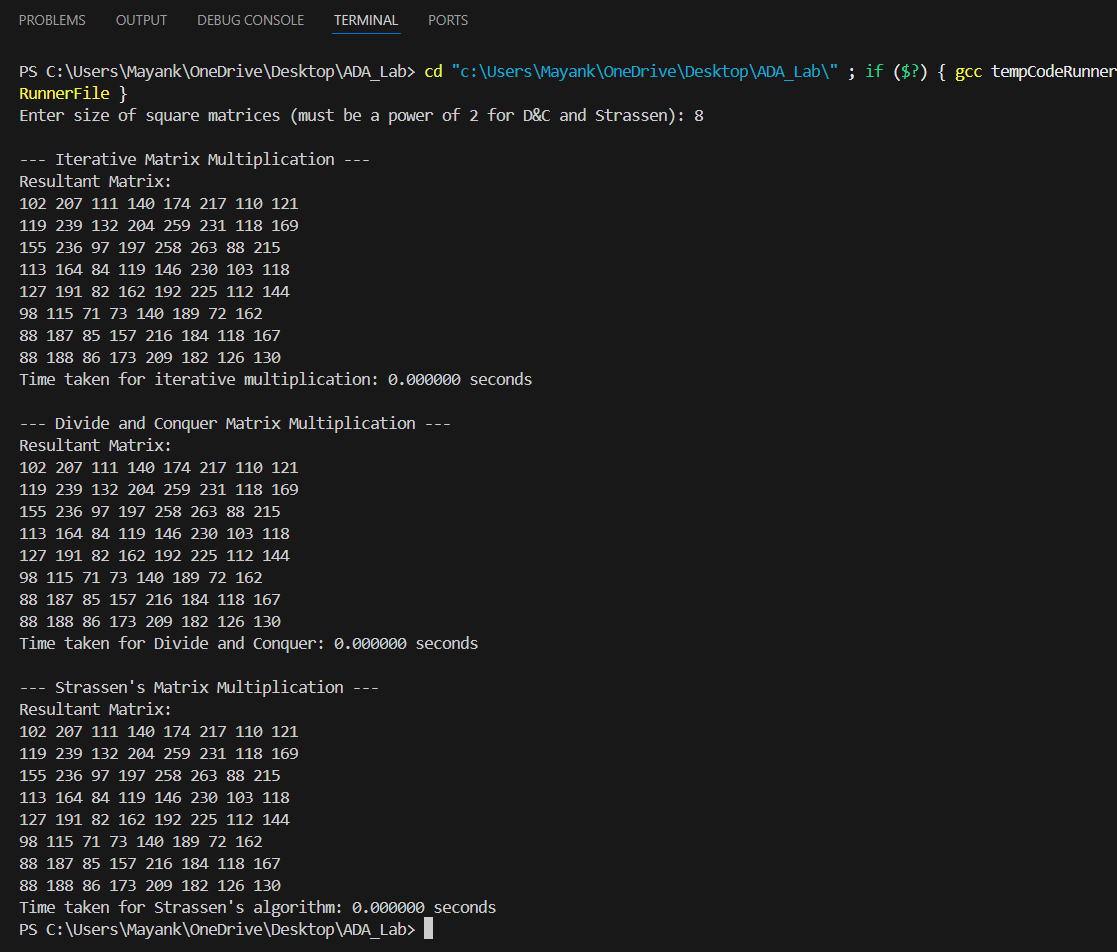
    return 0;

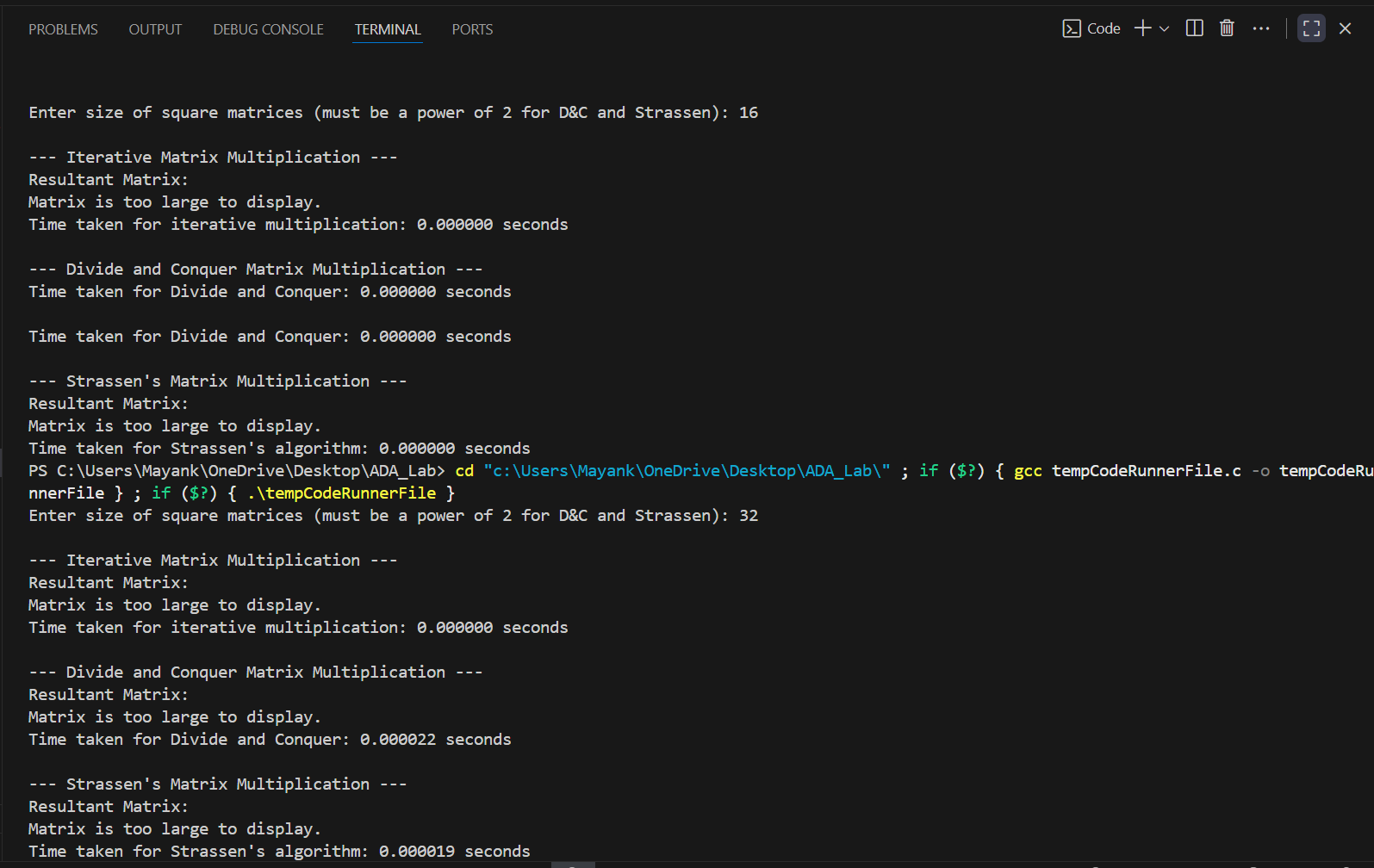
}

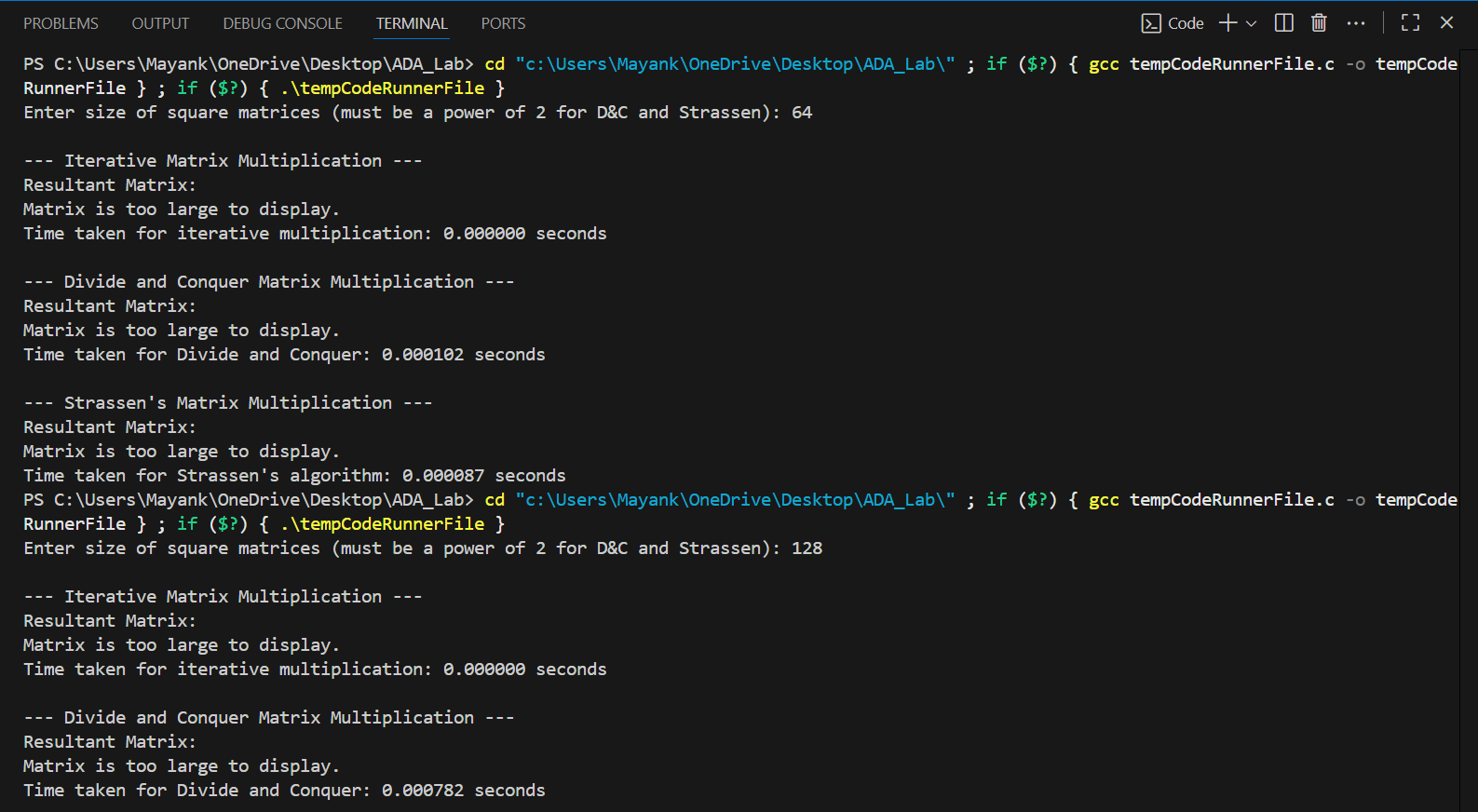
**OUTPUT:**

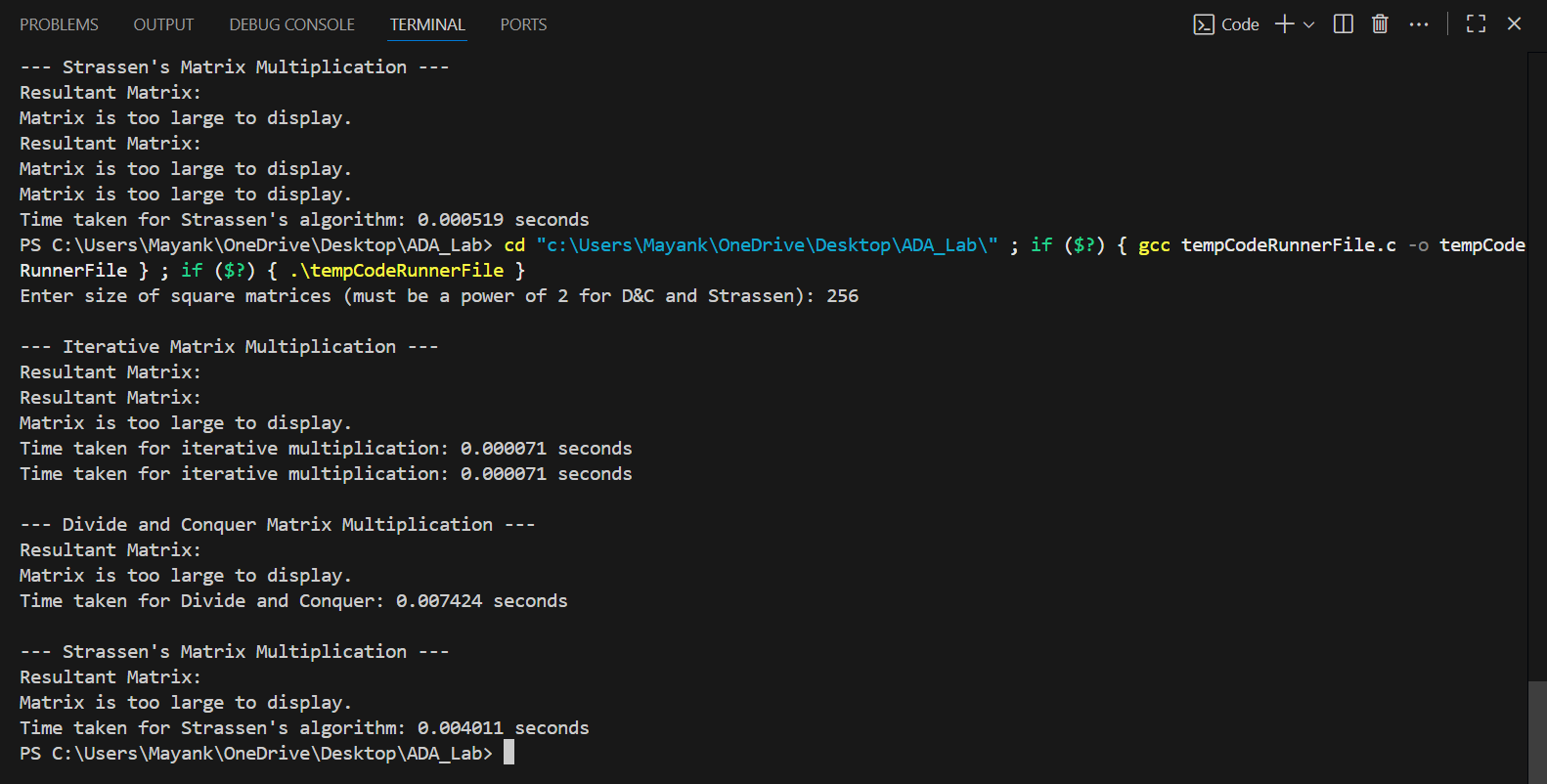












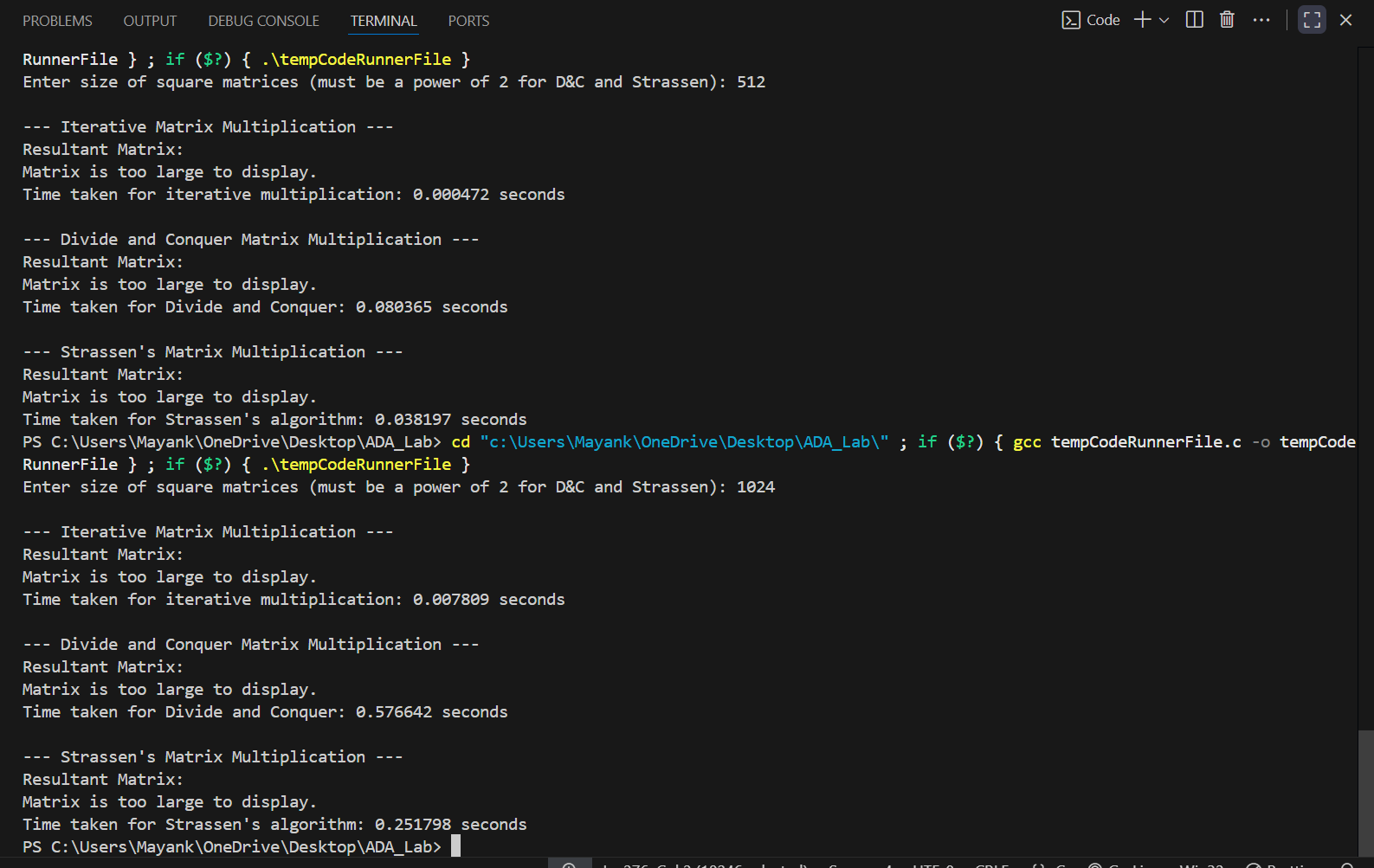


Table of run time taken:

|  |  |  |  |
| --- | --- | --- | --- |
| Matrix size | Iterative approach | Divide and conquer approach | Strassen’s Approach |
| 2\*2 | 0.000000 | 0.000000 | 0.000000 |
| 4\*4 | 0.000000 | 0.000000 | 0.000000 |
| 8\*8 | 0.000000 | 0.000000 | 0.000000 |
| 16\*16 | 0.000000 | 0.000000 | 0.000000 |
| 32\*32 | 0.000000 | 0.000022 | 0.000019 |
| 64\*64 | 0.000000 | 0.000102 | 0.000087 |
| 128\*128 | 0.000000 | 0.000782 | 0.000519 |
| 256\*256 | 0.000071 | 0.007424 | 0.004011 |
| 512\*512 | 0.000472 | 0.080365 | 0.038197 |
| 1024\*1024 | 0.007809 | 0.576642 | 0.251798 |
| **Theoretical time complexities** | **O(n^3)** | **O(n^3)** | **O(n^2.81)** |

PYTHON CODE:

import matplotlib.pyplot as plt

# Sample data

x = ['2x2','4x4','8X8','16X16','32X32','64X64','128X128','256X256','512X512','1024X1024']

y1 = [0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000071,0.000472,0.007809]

y2 =[0.000000,0.000000,0.000000,0.000000, 0.000022,0.000102,0.000782,0.007424,0.080365,0.576642]

y3=[0.000000,0.000000,0.000000,0.000000,0.000019,0.000087,0.000519,0.004011,0.038197,0.251798]

# Create the plot

plt.plot(x, y1, marker='o', linestyle='-', color='c', label='ITERATIVE APPROACH')

plt.plot(x, y2, marker='o', linestyle='-', color='y', label='DIVIDE AND CONQUER APPROACH')

plt.plot(x, y3, marker='o', linestyle='-', color='b', label="STRASSEN'S APPROACH")

# Add labels and title

plt.xlabel('MATRIX SIZES : N\*N')

plt.ylabel('TIME TAKEN TO COMPUTE')

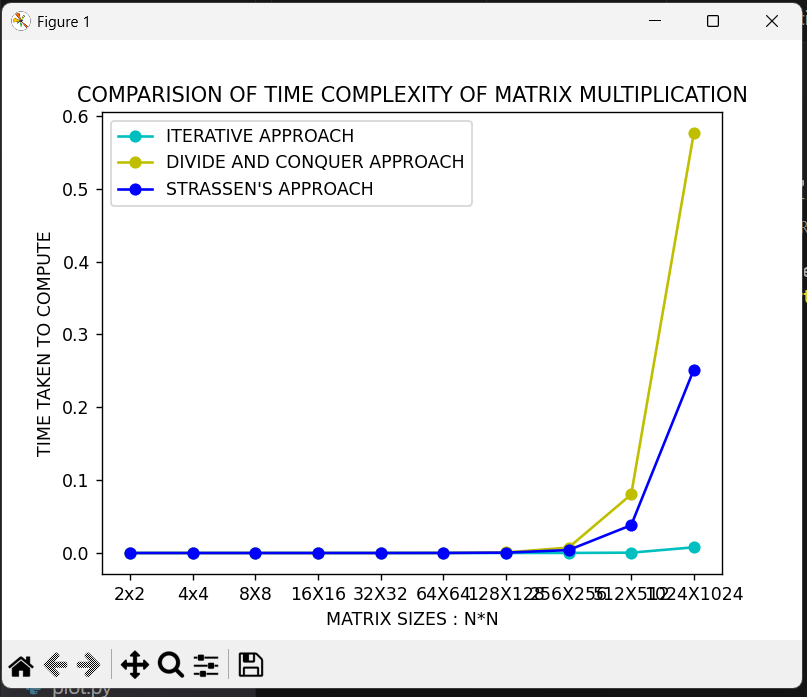
plt.title('COMPARISION OF TIME COMPLEXITY OF MATRIX MULTIPLICATION')

plt.legend()

# Show the plot

plt.show()

OUTPUT:



Practical conclusion:

FOR SMALL SIZE OF MATRICES ----------Time complexity : Divide & Conquer > Strassen >Iterative

FOR LARGE SIZE OF MATRICES ----------Time Complexity:

Divide & conquer > Iterative > Strassen