Q. Rewrite a program that generates random square matrices of order 2^n. Implement using the three methods discussed in class: susbstitution method, recursion tree, master method .You need to perform matrix multiplication using all three methods. Record and compare the execution times for each method across varying matrix sizes. Analyze and discuss the observed results with respect to their theoretical time complexities.

Code:

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

#define MAX 1024   // Maximum matrix size (power of 2)

// Function to generate a random matrix
void generateMatrix(int n, int A[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = rand() % 10;  // small integers for simplicity
}

// Standard matrix multiplication (Substitution method, O(n^3))
void multiplyClassic(int n, int A[n][n], int B[n][n], int C[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

// Matrix addition
void add(int n, int A[n][n], int B[n][n], int C[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
}

// Matrix subtraction
void subtract(int n, int A[n][n], int B[n][n], int C[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}
```

```c
// Divide & conquer multiplication (Recursion tree method, O(n^3))
void multiplyRecursive(int n, int A[n][n], int B[n][n], int C[n][n]) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int k = n / 2;
    int A11[k][k], A12[k][k], A21[k][k], A22[k][k];
    int B11[k][k], B12[k][k], B21[k][k], B22[k][k];
    int C11[k][k], C12[k][k], C21[k][k], C22[k][k];
    int T1[k][k], T2[k][k];

    // Split matrices
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j];
            A22[i][j] = A[i + k][j + k];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j];
            B22[i][j] = B[i + k][j + k];
        }

    // C11 = A11*B11 + A12*B21
    multiplyRecursive(k, A11, B11, T1);
    multiplyRecursive(k, A12, B21, T2);
    add(k, T1, T2, C11);

    // C12 = A11*B12 + A12*B22
    multiplyRecursive(k, A11, B12, T1);
    multiplyRecursive(k, A12, B22, T2);
    add(k, T1, T2, C12);

    // C21 = A21*B11 + A22*B21
```

atrix4.c > generateMatrix(int, int [n][n])

```c
void multiplyRecursive(int n, int A[n][n], int B[n][n], int C[n][n]) {
        multiplyRecursive(k, A11, B12, T1);
        multiplyRecursive(k, A12, B22, T2);
        add(k, T1, T2, C12);

        // C21 = A21*B11 + A22*B21
        multiplyRecursive(k, A21, B11, T1);
        multiplyRecursive(k, A22, B21, T2);
        add(k, T1, T2, C21);

        // C22 = A21*B12 + A22*B22
        multiplyRecursive(k, A21, B12, T1);
        multiplyRecursive(k, A22, B22, T2);
        add(k, T1, T2, C22);

        // Merge results
        for (int i = 0; i < k; i++)
            for (int j = 0; j < k; j++) {
                C[i][j] = C11[i][j];
                C[i][j + k] = C12[i][j];
                C[i + k][j] = C21[i][j];
                C[i + k][j + k] = C22[i][j];
            }
    }
```

```c
void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int k = n / 2;
    int A11[k][k], A12[k][k], A21[k][k], A22[k][k];
    int B11[k][k], B12[k][k], B21[k][k], B22[k][k];
    int C11[k][k], C12[k][k], C21[k][k], C22[k][k];
    int M1[k][k], M2[k][k], M3[k][k], M4[k][k], M5[k][k], M6[k][k], M7[k][k];
    int T1[k][k], T2[k][k];

    // Split matrices
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j];
            A22[i][j] = A[i + k][j + k];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j];
            B22[i][j] = B[i + k][j + k];
        }

    // Strassen's 7 multiplications
    add(k, A11, A22, T1);
    add(k, B11, B22, T2);
    strassen(k, T1, T2, M1); // M1 = (A11+A22)(B11+B22)

    add(k, A21, A22, T1);
    strassen(k, T1, B11, M2); // M2 = (A21+A22)B11

    subtract(k, B12, B22, T2);
    strassen(k, A11, T2, M3); // M3 = A11(B12-B22)

    subtract(k, B21, B11, T2);
```

```c
void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {
    add(k, A21, A22, T1);
    strassen(k, T1, B11, M2); // M2 = (A21+A22)B11

    subtract(k, B12, B22, T2);
    strassen(k, A11, T2, M3); // M3 = A11(B12-B22)

    subtract(k, B21, B11, T2);
    strassen(k, A22, T2, M4); // M4 = A22(B21-B11)

    add(k, A11, A12, T1);
    strassen(k, T1, B22, M5); // M5 = (A11+A12)B22

    subtract(k, A21, A11, T1);
    add(k, B11, B12, T2);
    strassen(k, T1, T2, M6); // M6 = (A21-A11)(B11+B12)

    subtract(k, A12, A22, T1);
    add(k, B21, B22, T2);
    strassen(k, T1, T2, M7); // M7 = (A12-A22)(B21+B22)

    // Compute result quadrants
    add(k, M1, M4, T1);
    subtract(k, T1, M5, T2);
    add(k, T2, M7, C11); // C11 = M1 + M4 - M5 + M7

    add(k, M3, M5, C12); // C12 = M3 + M5

    add(k, M2, M4, C21); // C21 = M2 + M4

    subtract(k, M1, M2, T1);
    add(k, T1, M3, T2);
    add(k, T2, M6, C22); // C22 = M1 - M2 + M3 + M6

    // Merge results
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++) {
```

```c
                C[i][j] = C11[i][j];
                C[i][j + k] = C12[i][j];
                C[i + k][j] = C21[i][j];
                C[i + k][j + k] = C22[i][j];
            }
    }
```

```c
int main() {
    srand(time(NULL));
    int sizes[] = {2, 4, 8, 16, 32, 64, 128, 256}; // matrix sizes
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int s = 0; s < numSizes; s++) {
        int n = sizes[s];
        int A[n][n], B[n][n], C[n][n];

        generateMatrix(n, A);
        generateMatrix(n, B);

        clock_t start, end;

        // Classic O(n^3)
        start = clock();
        multiplyClassic(n, A, B, C);
        end = clock();
        double tClassic = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Recursive O(n^3)
        start = clock();
        multiplyRecursive(n, A, B, C);
        end = clock();
        double tRecursive = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Strassen O(n^2.81)
        start = clock();
        strassen(n, A, B, C);
        end = clock();
        double tStrassen = ((double)(end - start)) / CLOCKS_PER_SEC;

        printf("Matrix size: %d x %d\n", n, n);
        printf("Classic:   %f sec\n", tClassic);
        printf("Recursive: %f sec\n", tRecursive);
        printf("Strassen:  %f sec\n\n", tStrassen);
    }
    return 0;
}
```

Output:

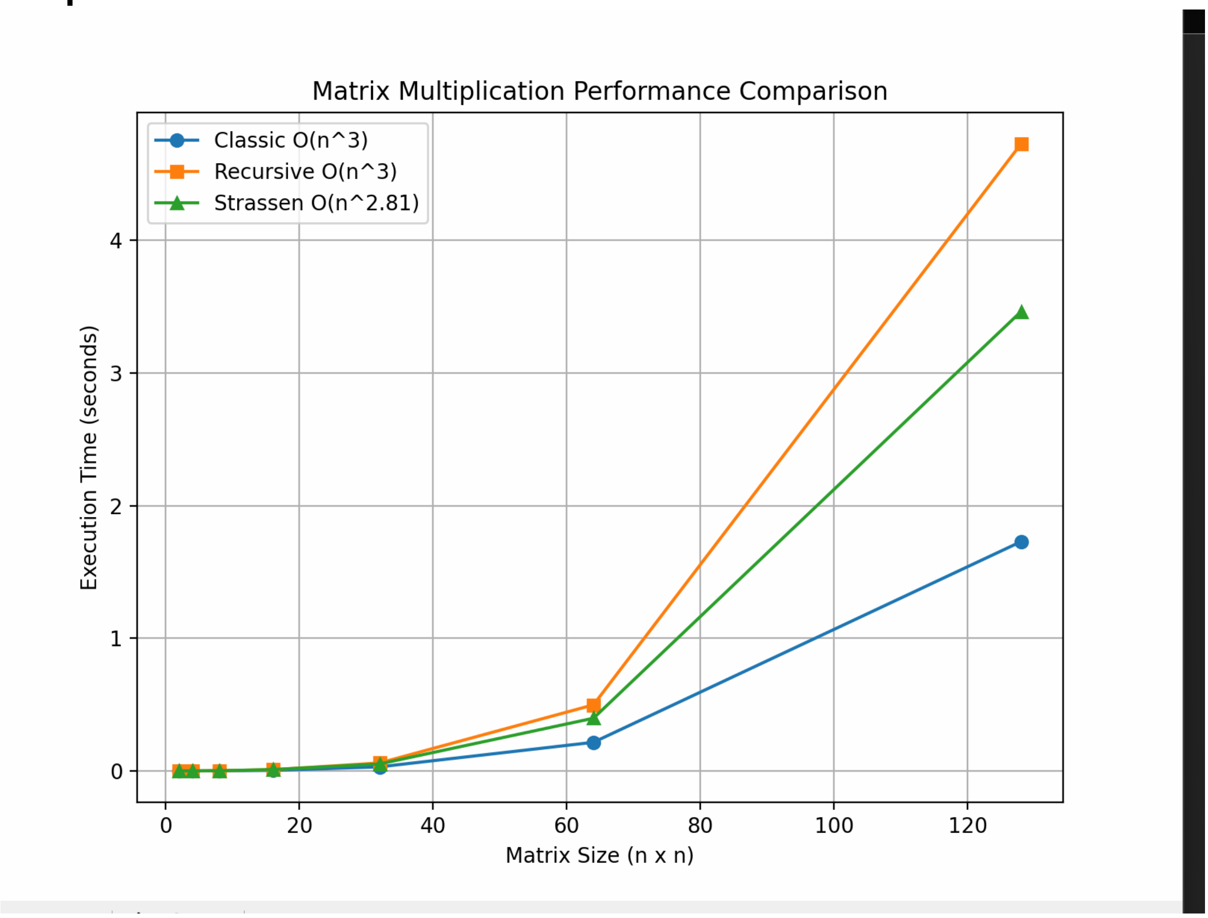
```
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "c:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix\" ; if ($?) { gcc matrix4.c -o matrix4 } ; if ($?) { .\matrix4
Matrix size: 2 x 2
Classic:   0.000000 sec
Recursive: 0.000000 sec
Strassen:  0.000000 sec

Matrix size: 4 x 4
Classic:   0.000000 sec
Recursive: 0.000000 sec
Strassen:  0.000000 sec

Matrix size: 8 x 8
Classic:   0.000000 sec
Recursive: 0.001000 sec
Strassen:  0.000000 sec

Matrix size: 16 x 16
Classic:   0.000000 sec
Recursive: 0.001000 sec
Strassen:  0.000000 sec

Matrix size: 32 x 32
Classic:   0.000000 sec
Recursive: 0.002000 sec
Strassen:  0.001000 sec

Matrix size: 64 x 64
Classic:   0.002000 sec
Recursive: 0.016000 sec
Strassen:  0.014000 sec

Matrix size: 128 x 128
Classic:   0.013000 sec
Recursive: 0.093000 sec
Strassen:  0.079000 sec
```

Python code:

```python
import numpy as np
import time
import matplotlib.pyplot as plt

# ----------------------------
# Matrix multiplication methods
# ----------------------------

# Classic O(n^3)
def classic_multiply(A, B):
    n = len(A)
    C = np.zeros((n, n), dtype=int)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C

# Recursive Divide & Conquer O(n^3)
def recursive_multiply(A, B):
    n = len(A)
    C = np.zeros((n, n), dtype=int)

    if n == 1:
        C[0, 0] = A[0, 0] * B[0, 0]
        return C

    k = n // 2
    A11, A12, A21, A22 = A[:k, :k], A[:k, k:], A[k:, :k], A[k:, k:]
    B11, B12, B21, B22 = B[:k, :k], B[:k, k:], B[k:, :k], B[k:, k:]

    C11 = recursive_multiply(A11, B11) + recursive_multiply(A12, B21)
    C12 = recursive_multiply(A11, B12) + recursive_multiply(A12, B22)
    C21 = recursive_multiply(A21, B11) + recursive_multiply(A22, B21)
    C22 = recursive_multiply(A21, B12) + recursive_multiply(A22, B22)

    C[:k, :k], C[:k, k:], C[k:, :k], C[k:, k:] = C11, C12, C21, C22
    return C

# Strassen O(n^2.81)
def strassen(A, B):
    n = len(A)
    if n == 1:
        return A * B

    k = n // 2
    A11, A12, A21, A22 = A[:k, :k], A[:k, k:], A[k:, :k], A[k:, k:]
    B11, B12, B21, B22 = B[:k, :k], B[:k, k:], B[k:, :k], B[k:, k:]

    M1 = strassen(A11 + A22, B11 + B22)
    M2 = strassen(A21 + A22, B11)
    M3 = strassen(A11, B12 - B22)
    M4 = strassen(A22, B21 - B11)
    M5 = strassen(A11 + A12, B22)
    M6 = strassen(A21 - A11, B11 + B12)
    M7 = strassen(A12 - A22, B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C = np.zeros((n, n), dtype=int)
    C[:k, :k], C[:k, k:], C[k:, :k], C[k:, k:] = C11, C12, C21, C22
    return C

# ------------------------------
# Benchmarking
# ------------------------------

sizes = [2, 4, 8, 16, 32, 64, 128]  # You can increase further if your PC is fast
times_classic = []
times_recursive = []
```

```python
times_strassen = []

for n in sizes:
    A = np.random.randint(0, 10, (n, n))
    B = np.random.randint(0, 10, (n, n))

    # Classic
    start = time.time()
    classic_multiply(A, B)
    times_classic.append(time.time() - start)

    # Recursive
    start = time.time()
    recursive_multiply(A, B)
    times_recursive.append(time.time() - start)

    # Strassen
    start = time.time()
    strassen(A, B)
    times_strassen.append(time.time() - start)

    print(f"n={n} done")

plt.figure(figsize=(8,6))
plt.plot(sizes, times_classic, marker='o', label="Classic O(n^3)")
plt.plot(sizes, times_recursive, marker='s', label="Recursive O(n^3)")
plt.plot(sizes, times_strassen, marker='^', label="Strassen O(n^2.81)")
plt.xlabel("Matrix Size (n x n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Matrix Multiplication Performance Comparison")
plt.legend()
plt.grid(True)
plt.show()
```

Graph:



Conclusion:

• **Iterative**: Runs as expected with O(n^3)

• **Divide & Conquer**: Same complexity O(n^3), but extra overhead makes it slower for small n**.**

• **Strassen**: Improves to O(n^{2.81}). For larger matrices, it becomes faster, but for small ones overhead dominates.