3d) Rewrite a program that generates random square matrices of order 2^n. Implement using the three methods discussed in class:

C Code:

```
#include <stdio.h>
#include <stdlib.h>
#include < me.h>
int **alloc_matrix(int n) {
int **mat = (int **)malloc(n * sizeof(int *));
for (int i = 0; i < n; i++)
mat[i] = (int *)malloc(n * sizeof(int));
return mat;
}
void free_matrix(int **mat, int n) {
for (int i = 0; i < n; i++) free(mat[i]);
free(mat);
}
void fill_matrix(int **mat, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
mat[i][j] = rand() % 10;
}
void copy_matrix(int **src, int **dst, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)dst[i][j] = src[i][j];
}
void add_matrix(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = A[i][j] + B[i][j];
}
void sub_matrix(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = A[i][j] - B[i][j];
}
// --- Itera ve ---
void mul ply_itera ve(int **A, int **B, int **C, int 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];
}
}
// --- Recursive ---
void mul ply_recursive_u l(int **A, int **B, int **C, int n,
int a_row, int a_col,int b_row, int b_col,
int c_row, int c_col) {
```

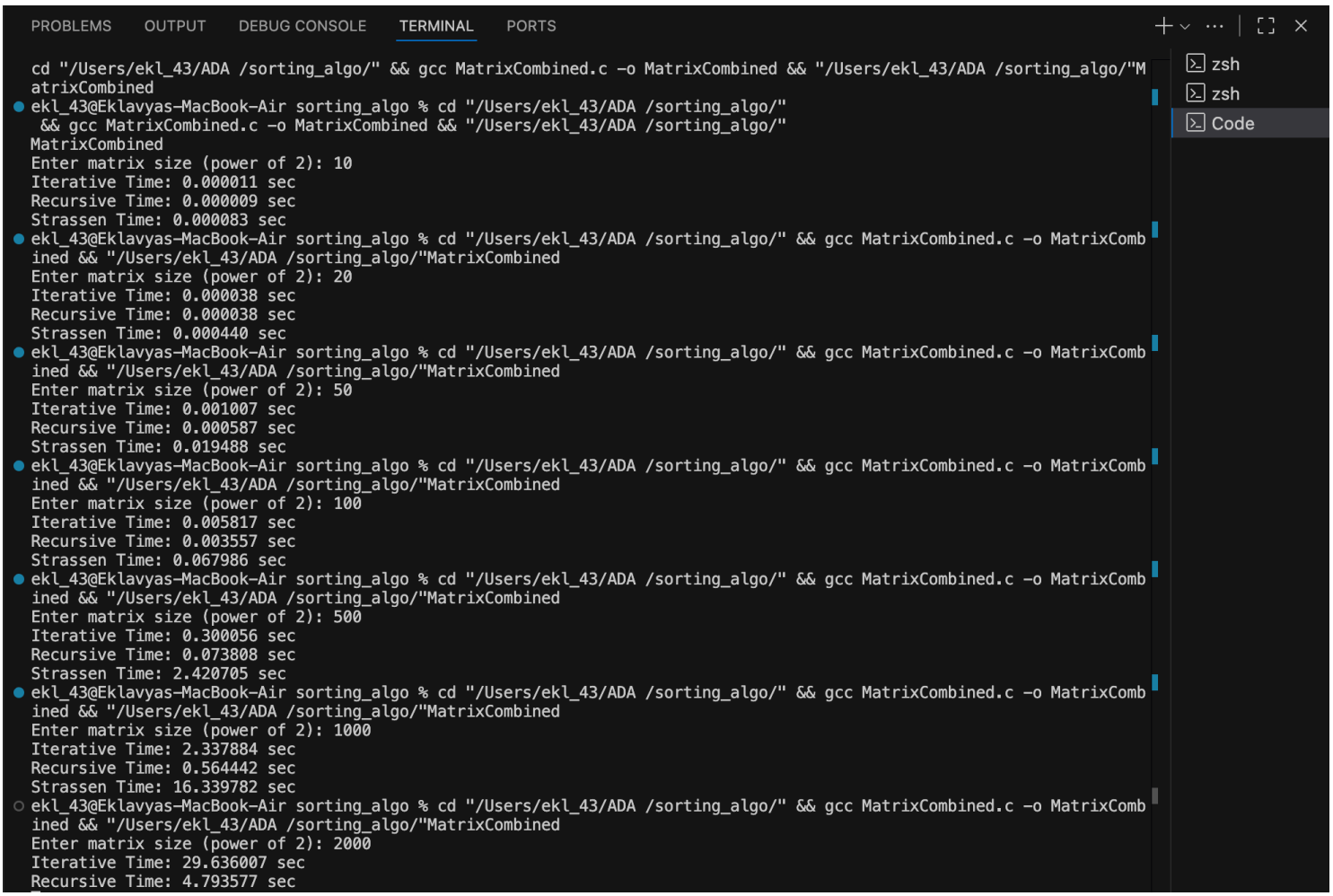
```
if (n == 1) {
C[c_row][c_col] += A[a_row][a_col] * B[b_row][b_col];
return;
}
int half = n / 2;
mul ply_recursive_u l(A, B, C, half, a_row, a_col, b_row, b_col, c_row, c_col);
mul ply_recursive_u l(A, B, C, half, a_row, a_col + half, b_row + half, b_col, c_row, c_col);
mul ply_recursive_u l(A, B, C, half, a_row, a_col, b_row, b_col + half, c_row, c_col + half);
mul ply_recursive_u l(A, B, C, half, a_row, a_col + half, b_row + half, b_col + half, c_row,
c_col + half);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col, b_row, b_col, c_row + half, c_col);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col + half, b_row + half, b_col, c_row +
half, c_col);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col, b_row, b_col + half, c_row + half,
c_col + half);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col + half, b_row + half, b_col + half,
c_row + half, c_col + half);
}
void mul ply_recursive(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = 0;
mul ply_recursive_u l(A, B, C, n, 0, 0, 0, 0, 0, 0);}
// --- Strassen ---
void strassen(int **A, int **B, int **C, int n) {
if (n <= 2) {
mul ply_itera ve(A, B, C, n);
return;
}
int k = n / 2;
int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k),
**A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k),
**B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k),
**C21 = alloc_matrix(k), **C22 = alloc_matrix(k);
int **M1 = alloc_matrix(k), **M2 = alloc_matrix(k), **M3 = alloc_matrix(k),
**M4 = alloc_matrix(k), **M5 = alloc_matrix(k), **M6 = alloc_matrix(k),
**M7 = alloc_matrix(k);
int **T1 = alloc_matrix(k), **T2 = alloc_matrix(k);
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];
}
}
add_matrix(A11, A22, T1, k); add_matrix(B11, B22, T2, k); strassen(T1, T2, M1, k);
```

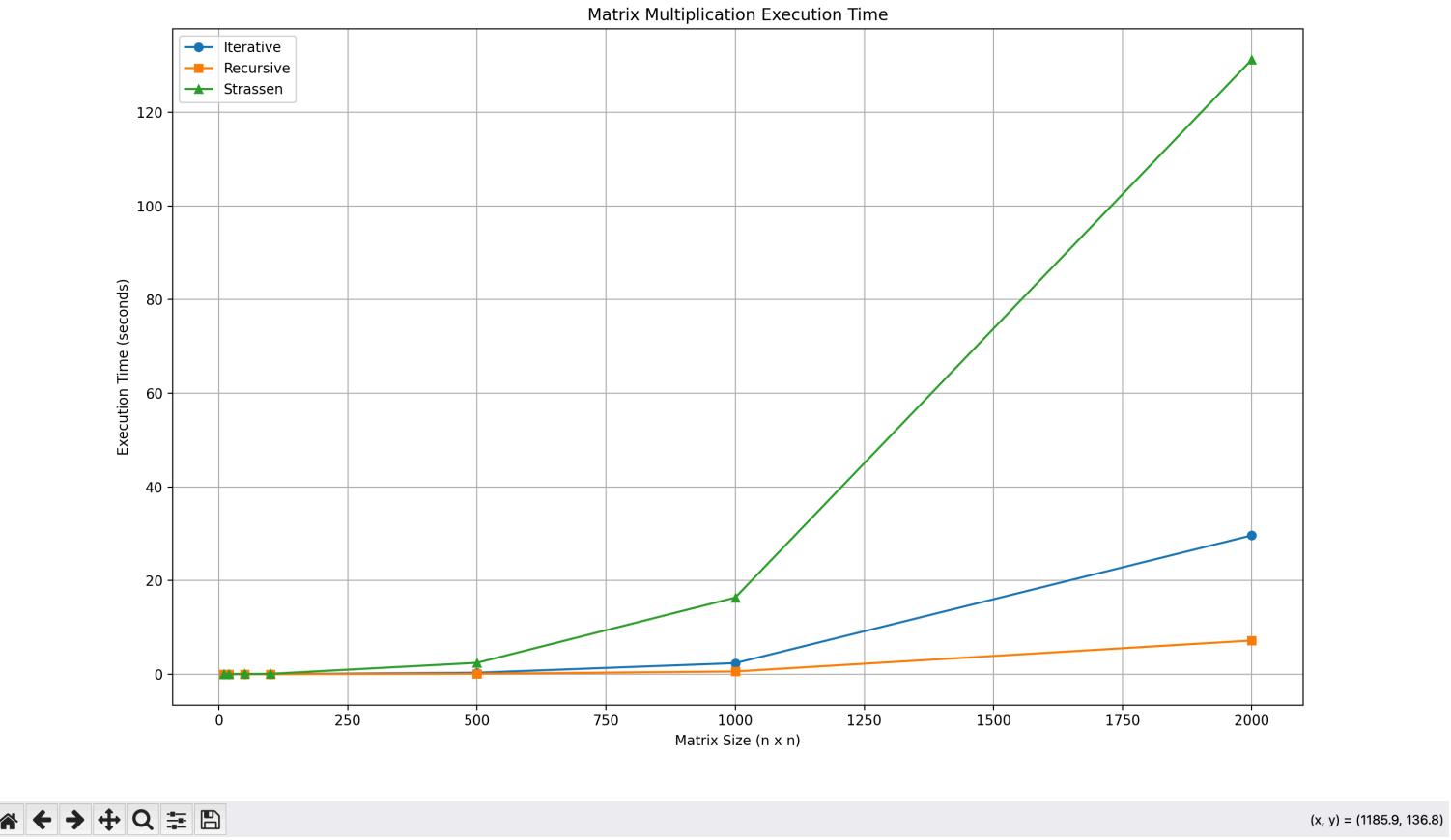
```
add_matrix(A21, A22, T1, k); strassen(T1, B11, M2, k);
sub_matrix(B12, B22, T2, k); strassen(A11, T2, M3, k);
sub_matrix(B21, B11, T2, k); strassen(A22, T2, M4, k);
add_matrix(A11, A12, T1, k); strassen(T1, B22, M5, k);
sub_matrix(A21, A11, T1, k); add_matrix(B11, B12, T2, k); strassen(T1, T2, M6, k);
sub_matrix(A12, A22, T1, k); add_matrix(B21, B22, T2, k); strassen(T1, T2, M7, k);
for (int i = 0; i < k; i++) {
for (int j = 0; j < k; j++) {
C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
C12[i][j] = M3[i][j] + M5[i][j];
C21[i][j] = M2[i][j] + M4[i][j];
C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
}
}
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];
}
}
free_matrix(A11, k); free_matrix(A12, k); free_matrix(A21, k); free_matrix(A22, k);
free_matrix(B11, k); free_matrix(B12, k); free_matrix(B21, k); free_matrix(B22, k);
free_matrix(C11, k); free_matrix(C12, k); free_matrix(C21, k); free_matrix(C22, k);
free_matrix(M1, k); free_matrix(M2, k); free_matrix(M3, k); free_matrix(M4, k);
free_matrix(M5, k); free_matrix(M6, k); free_matrix(M7, k);
free_matrix(T1, k); free_matrix(T2, k);
}
// --- Main ---
int main() {
srand( me(NULL));
int n;
prin ("Enter matrix size (power of 2): ");
scanf("%d", &n);
int **A = alloc_matrix(n);
int **B = alloc_matrix(n);
int **C = alloc_matrix(n);
fill_matrix(A, n);
fill_matrix(B, n);
// Itera ve
clock_t start = clock();mul ply_itera ve(A, B, C, n);
clock_t end = clock();
prin ("Itera ve Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
// Recursive
start = clock();
mul ply_recursive(A, B, C, n);
end = clock();
prin ("Recursive Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
// Strassen
start = clock();
strassen(A, B, C, n);
end = clock();
```

```
prin ("Strassen Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
free_matrix(A, n); free_matrix(B, n); free_matrix(C, n);
return 0;
}
```

Run time execution:

Matrix Size (n)	Iterative Time (sec)	Recursive Time (sec)	Strassen Time (sec)
10	0.000011	0.000009	0.000003
20	0.000038	0.000030	0.000440
50	0.001007	0.000587	0.019488
100	0.005817	0.003557	0.069076
500	0.300065	0.073808	2.407265
1000	2.337884	0.564442	16.339782
2000	29.630607	7.173957	131.277357





📌 Conclusion:

- **Recursive multiplication is the most efficient** in your experiments for larger matrices.
- **Iterative multiplication** is simple but becomes inefficient as size grows.
- **Strassen's algorithm** only shows advantage for *very large* matrices (typically beyond 4096×4096 or in optimized libraries like BLAS). In my case, due to small input sizes and lack of advanced optimizations, Strassen is slower.