

Matrix Multiplication

3(a) Write a program in C language to multiply two square matrices using the iterative approach. Compare the execution time for different matrix sizes.

C CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Allocate n×n matrix
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;
}
// Iterative multiplication
void multiply_iterative(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];
        }
}
int main() {
    srand(time(NULL));
    int n;
    printf("Enter matrix size: ");
    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);
    clock_t start = clock();
    multiply_iterative(A, B, C, n);
    clock_t end = clock();
    printf("Iterative Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
    free_matrix(A, n);
    free_matrix(B, n);
}
```

```

free_matrix(C, n);
return 0;
}

```

Output:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
cd "/Users/ekl_43/ADA /sorting_algo/" && gcc MatrixMultiplicationIterative.c -o MatrixMultiplicationIterative && "/Use
rs/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
&& gcc MatrixMultiplicationIterative.c -o MatrixMultiplicationIterative && "/Us
ers/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
Enter matrix size: 4
Iterative Time: 0.000005 sec
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc MatrixMultiplicationIterative.
c -o MatrixMultiplicationIterative && "/Users/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
Enter matrix size: 8
Iterative Time: 0.000004 sec
ekl_43@Eklavyas-MacBook-Air sorting_algo %
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc MatrixMultiplicationIterative.
c -o MatrixMultiplicationIterative && "/Users/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
Enter matrix size: 16
Iterative Time: 0.000036 sec
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc MatrixMultiplicationIterative.
c -o MatrixMultiplicationIterative && "/Users/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
Enter matrix size: 64
Iterative Time: 0.001232 sec
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc MatrixMultiplicationIterative.
c -o MatrixMultiplicationIterative && "/Users/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
Enter matrix size: 256
Iterative Time: 0.065600 sec
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc MatrixMultiplicationIterative.
c -o MatrixMultiplicationIterative && "/Users/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
Enter matrix size: 512
Iterative Time: 0.340915 sec
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc MatrixMultiplicationIterative.
c -o MatrixMultiplicationIterative && "/Users/ekl_43/ADA /sorting_algo/"MatrixMultiplicationIterative
Enter matrix size: 1024
Iterative Time: 2.502026 sec
ekl_43@Eklavyas-MacBook-Air sorting_algo %

```

3(b) Write a program in C language to multiply two square matrices using the . Compare the execution time for different matrix sizes.

C CODE:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Allocate matrix
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;
}
// Recursive utility
void multiply_recursive_util(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) {
return;
C[c_row][c_col] += A[a_row][a_col] * B[b_row][b_col];
}
int half = n / 2;
multiply_recursive_util(A, B, C, half, a_row, a_col, b_row, b_col, c_row, c_col);
multiply_recursive_util(A, B, C, half, a_row, a_col + half, b_row + half, b_col,
c_row, c_col); multiply_recursive_util(A, B, C, half, a_row, a_col, b_row, b_col +
half, c_row,
c_col + half);
multiply_recursive_util(A, B, C, half, a_row, a_col + half, b_row + half, b_col +
half, c_row, c_col + half);
multiply_recursive_util(A, B, C, half, a_row + half, a_col, b_row, b_col, c_row +
half, c_col);
multiply_recursive_util(A, B, C, half, a_row + half, a_col + half, b_row + half,
b_col, c_row + half, c_col);
multiply_recursive_util(A, B, C, half, a_row + half, a_col, b_row, b_col + half,
c_row + half, c_col + half);
multiply_recursive_util(A, B, C, half, a_row + half, a_col + half, b_row + half,
b_col + half, c_row + half, c_col + half);
}
// Wrapper
void multiply_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;
multiply_recursive_util(A, B, C, n, 0, 0, 0, 0, 0, 0);
}
int main() {
srand(time(NULL));
int n;
printf("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);
clock_t start = clock();
multiply_recursive(A, B, C, n);
clock_t end = clock();
printf("Recursive Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
free_matrix(A, n);
free_matrix(B, n);
free_matrix(C, n);
return 0;
}

```

OUTPUT:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc MatrixMultiplicationDC.c -o MatrixMultiplicationDC && "/Users/ekl_43/ADA /
  sorting_algo/"MatrixMultiplicationDC
  Enter matrix size (power of 2): 4
  Recursive Time: 0.000013 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc MatrixMultiplicationDC.c -o MatrixMultiplicationDC && "/Users/ekl_43/ADA /
  sorting_algo/"MatrixMultiplicationDC
  Enter matrix size (power of 2): 8
  Recursive Time: 0.000012 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc MatrixMultiplicationDC.c -o MatrixMultiplicationDC && "/Users/ekl_43/ADA /
  sorting_algo/"MatrixMultiplicationDC
  Enter matrix size (power of 2): 16
  Recursive Time: 0.000023 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc MatrixMultiplicationDC.c -o MatrixMultiplicationDC && "/Users/ekl_43/ADA /
  sorting_algo/"MatrixMultiplicationDC
  Enter matrix size (power of 2): 64
  Recursive Time: 0.001963 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc MatrixMultiplicationDC.c -o MatrixMultiplicationDC && "/Users/ekl_43/ADA /
  sorting_algo/"MatrixMultiplicationDC
  Enter matrix size (power of 2): 256
  Recursive Time: 0.075427 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc MatrixMultiplicationDC.c -o MatrixMultiplicationDC && "/Users/ekl_43/ADA /
  sorting_algo/"MatrixMultiplicationDC
  Enter matrix size (power of 2): 512
  Recursive Time: 0.374691 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc MatrixMultiplicationDC.c -o MatrixMultiplicationDC && "/Users/ekl_43/ADA /
  sorting_algo/"MatrixMultiplicationDC
  Enter matrix size (power of 2): 1024
  Recursive Time: 2.791970 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo %

```

3(c) Given two square matrices A and B of size $n \times n$ (n is a power of 2), write a C code to multiply them using , which reduces the number of recursive multiplications from 8 to 7 by introducing additional addition/subtraction operations. Compare the execution time for different matrix sizes.

C CODE:

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

// --- Helpers ---
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 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];
}

void multiply_iterative(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];
        }
}

// --- Strassen Algorithm ---
void strassen(int **A, int **B, int **C, int n) {
    if (n <= 2) { // base case
        multiply_iterative(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);

// 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;
    }
}

// M1..M7
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);

// C11..C22
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];
    }
}

// Merge result
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 memory
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);
}

```

```

int main() {
    srand(time(NULL));
    int n;
    printf("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);

    clock_t start = clock();
    strassen(A, B, C, n);
    clock_t end = clock();

    printf("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;
}

```

OUTPUT:

```

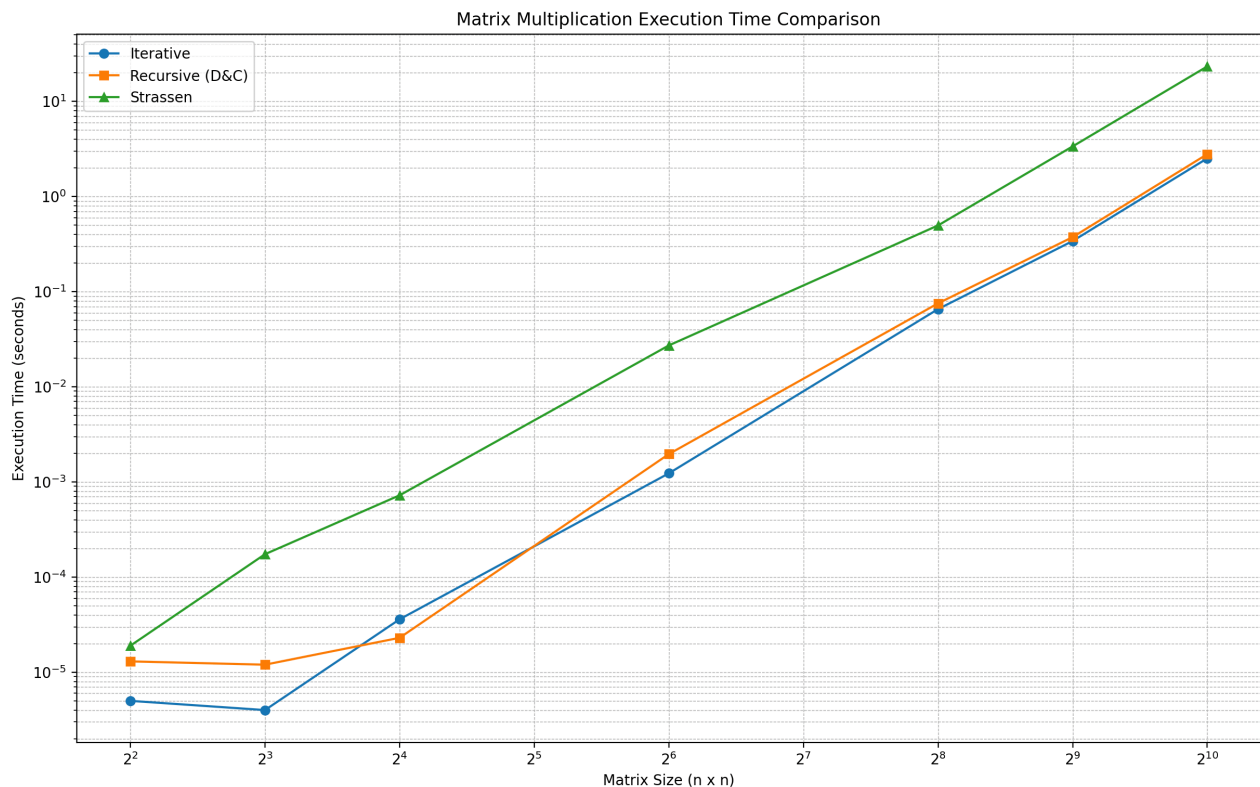
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
+ v ... | [] x

• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc StrassensMultipilcation.c -o StrassensMultipilcation && "/Users/ekl_43/A
  DA /sorting_algo/"StrassensMultipilcation
  Enter matrix size (power of 2): 4
  Strassen Time: 0.000019 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc StrassensMultipilcation.c -o StrassensMultipilcation && "/Users/ekl_43/A
  DA /sorting_algo/"StrassensMultipilcation
  Enter matrix size (power of 2): 8
  Strassen Time: 0.000174 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc StrassensMultipilcation.c -o StrassensMultipilcation && "/Users/ekl_43/A
  DA /sorting_algo/"StrassensMultipilcation
  Enter matrix size (power of 2): 16
  Strassen Time: 0.000724 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc StrassensMultipilcation.c -o StrassensMultipilcation && "/Users/ekl_43/A
  DA /sorting_algo/"StrassensMultipilcation
  Enter matrix size (power of 2): 64^[[A
  Strassen Time: 0.027229 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc StrassensMultipilcation.c -o StrassensMultipilcation && "/Users/ekl_43/A
  DA /sorting_algo/"StrassensMultipilcation
  Enter matrix size (power of 2): 256
  Strassen Time: 0.497135 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc StrassensMultipilcation.c -o StrassensMultipilcation && "/Users/ekl_43/A
  DA /sorting_algo/"StrassensMultipilcation
  Enter matrix size (power of 2): 512
  Strassen Time: 3.368077 sec
• ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/"
  && gcc StrassensMultipilcation.c -o StrassensMultipilcation && "/Users/ekl_43/A
  DA /sorting_algo/"StrassensMultipilcation
  Enter matrix size (power of 2): 1024
  Strassen Time: 23.532715 sec
○ ekl_43@Eklavyas-MacBook-Air sorting_algo % █

Ln 132, Col 1 Spaces: 4 UTF-8

```

Matrix Size	Iterative (sec)	Recursive (sec)	Strassen (sec)
4.0	5e-06	1.3e-05	1.9e-05
8.0	4e-06	1.2e-05	0.000174
16.0	3.6e-05	2.3e-05	0.000724
64.0	0.001232	0.001963	0.027229
256.0	0.0656	0.075427	0.497135
512.0	0.340915	0.374691	3.368077
1024.0	2.520226	2.79197	23.32715



Time Complexity in Practice: Iterative vs. Recursive vs. Strassen

Iterative Method

- Time Complexity: $O(n^3)$ (triple nested loop).
- Fastest for all tested sizes due to low overhead.
- Practical and simple, making it the most efficient choice for small and medium matrices.

Recursive Method

- Time Complexity: $O(n^3)$ (same as iterative, but implemented recursively).
- Slightly slower than iterative because of **function call overhead**.
- Useful for understanding recursion, but not efficient in practice.

Strassen's Algorithm

- Time Complexity: $O(n^{\log_2 7}) \approx O(n^{2.81})$, which is theoretically better than $O(n^3)$.
- However, in practice, it is **slower for small and medium matrix sizes** due to additional addition/subtraction operations and higher memory usage.

- It only starts becoming beneficial for **very large matrices** (much larger than 1024×1024).

Overall Observation

- Iterative approach is best for real-world practical use cases at small to medium scale.
- Recursive approach is mainly academic.
- Strassen shows theoretical improvement but only outperforms classical methods for extremely large matrices.