Practical: Matrix Multiplication Algorithms

Student: Dhruv Sachdeva, `sachdevadhruv023@ce.du.ac.in`

Introduction

Matrix multiplication is a fundamental operation in computer science and numerical computing. We will study three approaches:

1. Classical Iterative Matrix Multiplication
2. Recursive Matrix Multiplication
3. Strassen's Algorithm

The time taken for multiplication will be measured for different matrix sizes n.

Problem 1. *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.*

1. Iterative Matrix Multiplication

The classical method runs three nested loops and computes:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]$$

Algorithm 1 Iterative Matrix Multiplication

Require: Matrices A, B of size $n \times n$
Ensure: Matrix $C = A \times B$

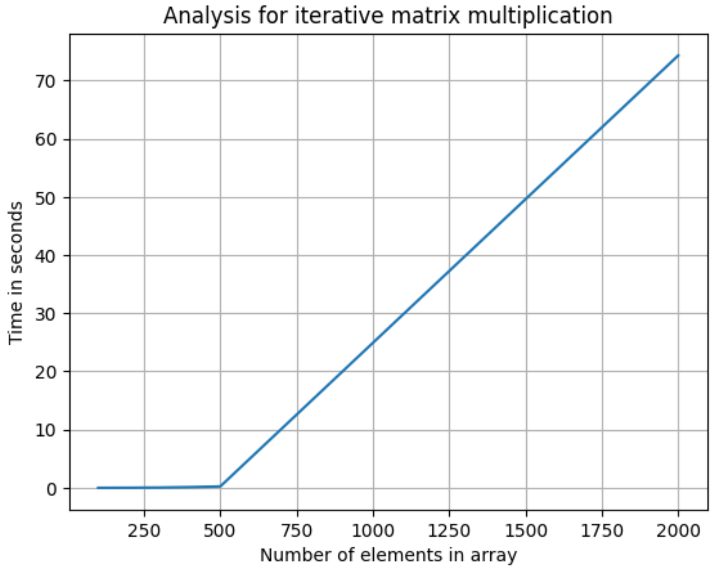
for $i \leftarrow 0$ to $n-1$ **do**
    for $j \leftarrow 0$ to $n-1$ **do**
        $C[i][j] \leftarrow 0$
        for $k \leftarrow 0$ to $n-1$ **do**
            $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
        end for
    end for
end for

C Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int mat_mul(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];
            }
        }
    }
}
void fill_mat(int n, int Mat[n][n])
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            Mat[i][j] = rand() % 10;
        }
    }
```

```c
}
int main()
{
    srand(time(NULL));
    int size[] = {100, 200, 300, 400,500};
    int num = sizeof(size) / sizeof(size[0]);
    for (int i = 0; i < num; i++)
    {
        int n = size[i];
        int (*A)[n] = malloc(sizeof(int[n][n]));
        int (*B)[n] = malloc(sizeof(int[n][n]));
        int (*C)[n] = malloc(sizeof(int[n][n]));

        if (A == NULL || B == NULL || C == NULL)
        {
            printf("Memory allocation failed for size %d\n", n);
            return(1);
        }
        fill_mat(n, A);
        fill_mat(n, B);

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

        double time_taken = (double)(end - start) / CLOCKS_PER_SEC;

        printf("Matrix size: %d x %d | Execution time: %f seconds\n", n, n,

        free(A);
        free(B);
        free(C);
    }

    return 0;
}
```

Graph:



Analysis

$$T(n) = O(n^3), \quad S(n) = O(n^2)$$

Problem 2. *3(b) Write a program in C language to multiply two square matrices using the . Compare the execution time for different matrix sizes.*

.

2. Recursive Matrix Multiplication

The matrix is divided into four submatrices and the formula

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \quad \ldots$$

is applied recursively.

Algorithm 2 Recursive Matrix Multiplication

Require: A, B of size $n \times n$, $n > 1$

Ensure: $C = A \times B$

  if $n = 1$ **then**

    $C[0][0] \leftarrow A[0][0] \times B[0][0]$

  else

    Split A and B into submatrices of size $n/2$

    Compute submatrices of C recursively

  end if

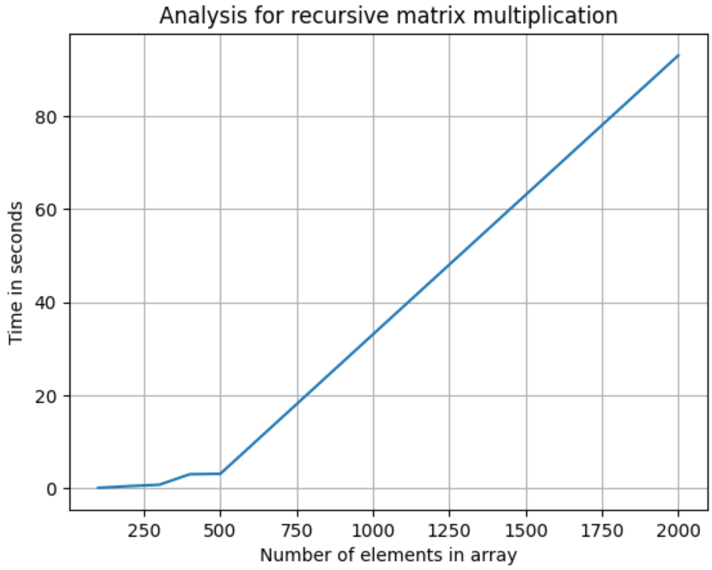
C Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void mat_mul(int n,int A[n][n],int B[n][n], int C[n][n])
{
    if (n < 2) {
            C[0][0] += A[0][0] * B[0][0];
            return;
      }

      int k = n / 2;

      int (*A11)[k] = malloc(sizeof(int[k][k]));
      int (*A12)[k] = malloc(sizeof(int[k][k]));
      int (*A21)[k] = malloc(sizeof(int[k][k]));
      int (*A22)[k] = malloc(sizeof(int[k][k]));

      int (*B11)[k] = malloc(sizeof(int[k][k]));
      int (*B12)[k] = malloc(sizeof(int[k][k]));
      int (*B21)[k] = malloc(sizeof(int[k][k]));
      int (*B22)[k] = malloc(sizeof(int[k][k]));

      int (*C11)[k] = malloc(sizeof(int[k][k]));
      int (*C12)[k] = malloc(sizeof(int[k][k]));
      int (*C21)[k] = malloc(sizeof(int[k][k]));
      int (*C22)[k] = malloc(sizeof(int[k][k]));

```

```c

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

    // Divide A and B into 4 parts
    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];
        }
    }


    mat_mul(k, A11, B11, C11);
    mat_mul(k, A12, B21, C11);

    mat_mul(k, A11, B12, C12);
    mat_mul(k, A12, B22, C12);

    mat_mul(k, A21, B11, C21);
    mat_mul(k, A22, B21, C21);

    mat_mul(k, A21, B12, C22);
    mat_mul(k, A22, B22, C22);

    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(A11); free(A12); free(A21); free(A22);
    free(B11); free(B12); free(B21); free(B22);
    free(C11); free(C12); free(C21); free(C22);
}

```

```c

void fill_mat(int n, int Mat[n][n])
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            Mat[i][j] = rand() % 10;
        }
    }
}
int main()
{
    srand(time(NULL));
    int size[] = {100, 200, 300, 400,500};
    int num = sizeof(size) / sizeof(size[0]);
    for (int i = 0; i < num; i++)
    {
        int n = size[i];
        int (*A)[n] = malloc(sizeof(int[n][n]));
        int (*B)[n] = malloc(sizeof(int[n][n]));
        int (*C)[n] = malloc(sizeof(int[n][n]));

        if (A == NULL || B == NULL || C == NULL)
        {
            printf("Memory allocation failed for size %d\n", n);
            return(1);
        }
        fill_mat(n, A);
        fill_mat(n, B);

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

        double time_taken = (double)(end - start) / CLOCKS_PER_SEC;

        printf("Matrix size: %d x %d | Execution time: %f seconds\n", n, n,

        free(A);
        free(B);
        free(C);
    }

    return 0;
}
```

Graph:



Analysis

$$T(n) = O(n^3), \quad S(n) = O(n^2)$$

Problem 3. *3(c) Given two square matrices A and B of size n × 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*

3. Strassen's Algorithm

Strassen reduces the number of recursive multiplications to 7 using clever combinations.

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

By Master Theorem:

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

Algorithm 3 Strassen's Algorithm

Require: Matrices A, B of size $n \times n$, n is power of 2
Ensure: $C = A \times B$
  if $n = 1$ **then**
    Multiply directly
  else
    Split A and B into quadrants
    Compute $M_1, M_2, \ldots, M_7$
    Form $C_{11}, C_{12}, C_{21}, C_{22}$
  end if

C Code:

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

void fill_matrix(int n, int **M) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            M[i][j] = rand() % 10;
}

void print_matrix(int n, int **M) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf("%4d ", M[i][j]);
        printf("\n");
    }
    printf("\n");
}

int **alloc_matrix(int n) {
    int **M = malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++)
        M[i] = calloc(n, sizeof(int));    // initialise to 0
```

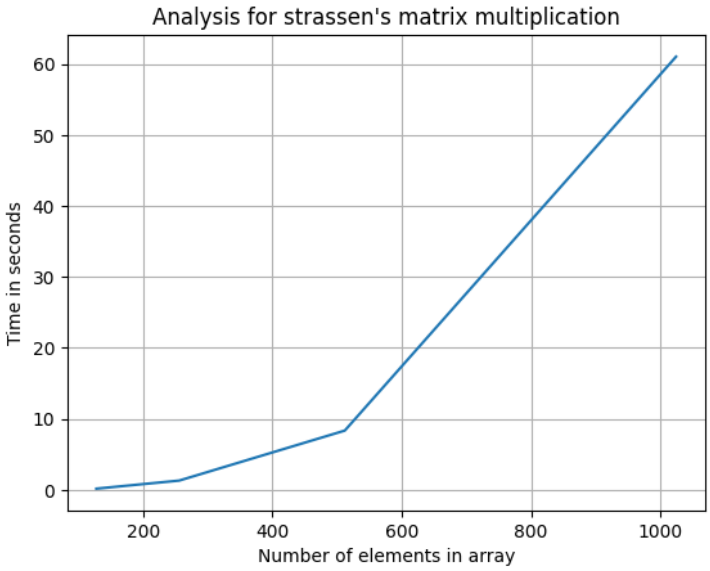
```c
    return M;
}

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

void add_matrix(int n, int **A, int **B, int **C) {
    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 n, int **A, int **B, int **C) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}

void naive_mul(int n, int **A, int **B, int **C) {
    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];
        }
}


void strassen(int n, int **A, int **B, int **C) {
    if (n <= 2) {  // base case: use naive multiplication
        naive_mul(n, A, B, C);
        return;
    }

    int k = n / 2;


    int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k);
    int **A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
    int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k);
    int **B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
    int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k);
    int **C21 = alloc_matrix(k), **C22 = alloc_matrix(k);

    int **M1 = alloc_matrix(k), **M2 = alloc_matrix(k), **M3 = alloc_matrix(
    int **M4 = alloc_matrix(k), **M5 = alloc_matrix(k), **M6 = alloc_matrix(
    int **T1 = alloc_matrix(k), **T2 = alloc_matrix(k);

    // split A and B into 4 parts
```

```
    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 = (A11 + A22) * (B11 + B22)
    add_matrix(k, A11, A22, T1);
    add_matrix(k, B11, B22, T2);
    strassen(k, T1, T2, M1);

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

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

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

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

    // M6 = (A21 - A11) * (B11 + B12)
    sub_matrix(k, A21, A11, T1);
    add_matrix(k, B11, B12, T2);
    strassen(k, T1, T2, M6);

    // M7 = (A12 - A22) * (B21 + B22)
    sub_matrix(k, A12, A22, T1);
    add_matrix(k, B21, B22, T2);
    strassen(k, T1, T2, M7);

    // C11 = M1 + M4 - M5 + M7
    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 = M3 + M5
```

```
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            C12[i][j] = M3[i][j] + M5[i][j];

    // C21 = M2 + M4
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            C21[i][j] = M2[i][j] + M4[i][j];

    // C22 = M1 - M2 + M3 + M6
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];

    // join C
    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(k, A11); free_matrix(k, A12); free_matrix(k, A21); free_matr
    free_matrix(k, B11); free_matrix(k, B12); free_matrix(k, B21); free_matr
    free_matrix(k, C11); free_matrix(k, C12); free_matrix(k, C21); free_matr
    free_matrix(k, M1);  free_matrix(k, M2);  free_matrix(k, M3);
free_matrix(k, M4);
    free_matrix(k, M5);  free_matrix(k, M6);  free_matrix(k, M7);
    free_matrix(k, T1);  free_matrix(k, T2);
}


int main() {
    int size[] = {100, 200, 300, 400,500};
    int num = sizeof(size) / sizeof(size[0]);
    for (int i = 0; i < num; i++)
    {
    srand(time(NULL));
    int n= size[i];

    int **A = alloc_matrix(n);
    int **B = alloc_matrix(n);
    int **C = alloc_matrix(n);

    fill_matrix(n, A);
    fill_matrix(n, B);

    clock_t start = clock();
```

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

        double secs = (double)(end - start) / CLOCKS_PER_SEC;
        printf("Matrix size %d x %d | Time taken: %f seconds\n", n, n, secs);

        // print_matrix(n, C);

        free_matrix(n, A);
        free_matrix(n, B);
        free_matrix(n, C);
    }
        return 0;
}
```

Graph:



Analysis

$$T(n) = O(n^{2.81}), \quad S(n) = O(n^2)$$

Problem 4. *Rewrite a program that generates random square matrices of order 2^n. Implement using the three methods discussed in class: 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.* ***C Code:***

```c

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

int **alloc_matrix(int n) {
    int **M = malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++)
        M[i] = calloc(n, sizeof(int));
    return M;
}

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

void fill_matrix(int n, int **M) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            M[i][j] = rand() % 10;
}

void iterative_mul(int n, int **A, int **B, int **C) {
    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];
        }
}

void recursive_mul(int n, int **A, int **B, int **C) {
    if (n == 1) {
        C[0][0] += A[0][0] * B[0][0];
        return;
    }
    int k = n / 2;
    int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k);
    int **A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
    int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k);
    int **B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
    int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k);
    int **C21 = alloc_matrix(k), **C22 = 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];
        }
    }
    recursive_mul(k, A11, B11, C11);
    recursive_mul(k, A12, B21, C11);
    recursive_mul(k, A11, B12, C12);
    recursive_mul(k, A12, B22, C12);
    recursive_mul(k, A21, B11, C21);
    recursive_mul(k, A22, B21, C21);
    recursive_mul(k, A21, B12, C22);
    recursive_mul(k, A22, B22, C22);
    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(k, A11);
    free_matrix(k, A12);
    free_matrix(k, A21);
    free_matrix(k,A22);
    free_matrix(k, B11);
    free_matrix(k, B12);
    free_matrix(k, B21);
    free_matrix(k,B22);
    free_matrix(k, C11);
    free_matrix(k, C12);
    free_matrix(k, C21);
    free_matrix(k,C22);
}

void add_matrix(int n, int **A, int **B, int **C) {
    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 n, int **A, int **B, int **C) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
```

```
}

void naive_mul(int n, int **A, int **B, int **C) {
    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];
        }
}

void strassen(int n, int **A, int **B, int **C) {
    if (n <= 2) {
        naive_mul(n, A, B, C);
        return;
    }
    int k = n / 2;
    int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k);
    int **A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
    int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k);
    int **B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
    int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k);
    int **C21 = alloc_matrix(k), **C22 = alloc_matrix(k);
    int **M1 = alloc_matrix(k), **M2 = alloc_matrix(k),
    **M3 = alloc_matrix(k);
    int **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(k, A11, A22, T1);
    add_matrix(k, B11, B22, T2);
    strassen(k, T1, T2, M1);
    add_matrix(k, A21, A22, T1);
    strassen(k, T1, B11, M2);
    sub_matrix(k, B12, B22, T2);
    strassen(k, A11, T2, M3);
    sub_matrix(k, B21, B11, T2);
    strassen(k, A22, T2, M4);
    add_matrix(k, A11, A12, T1);
    strassen(k, T1, B22, M5);
```

```
    sub_matrix(k, A21, A11, T1);
    add_matrix(k, B11, B12, T2);
    strassen(k, T1, T2, M6);
    sub_matrix(k, A12, A22, T1);
    add_matrix(k, B21, B22, T2);
    strassen(k, T1, T2, M7);
    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];
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            C12[i][j] = M3[i][j] + M5[i][j];
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            C21[i][j] = M2[i][j] + M4[i][j];
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; 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(k, A11); free_matrix(k, A12); free_matrix(k, A21); free_matr
    free_matrix(k, B11); free_matrix(k, B12); free_matrix(k, B21); free_matr
    free_matrix(k, C11); free_matrix(k, C12); free_matrix(k, C21); free_matr
    free_matrix(k, M1); free_matrix(k, M2); free_matrix(k, M3); free_matrix(
    free_matrix(k, M5); free_matrix(k, M6); free_matrix(k, M7);
    free_matrix(k, T1); free_matrix(k, T2);
}

int main() {
    int sizes[] = {128, 256, 512, 1024};
    int num = sizeof(sizes) / sizeof(sizes[0]);
    printf("%10s %15s %15s %15s\n", "n", "Iterative", "Recursive", "Strassen
    for (int i = 0; i < num; i++) {
        int n = sizes[i];
        int **A = alloc_matrix(n);
        int **B = alloc_matrix(n);
        int **C = alloc_matrix(n);
        fill_matrix(n, A);
        fill_matrix(n, B);

        clock_t start, end;
        double t_iter, t_rec, t_strassen;
        start = clock();
        iterative_mul(n, A, B, C);
        end = clock();
```

```c
        t_iter = (double)(end - start) / CLOCKS_PER_SEC;

        start = clock();
        recursive_mul(n, A, B, C);
        end = clock();
        t_rec = (double)(end - start) / CLOCKS_PER_SEC;

        start = clock();
        strassen(n, A, B, C);
        end = clock();
        t_strassen = (double)(end - start) / CLOCKS_PER_SEC;

        printf("%10d %15.6f %15.6f %15.6f\n", n, t_iter, t_rec, t_strassen);
        free_matrix(n, A);
        free_matrix(n, B);
        free_matrix(n, C);
    }
    return 0;
}
```

Comparison:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS D:\college\second year\ADA> ./a.exe
         n       Iterative       Recursive        Strassen
       128        0.011000        0.786000        0.174000
       256        0.082000        6.190000        2.009000
       512        1.637000      123.722000       22.311000
PS D:\college\second year\ADA>
```

Practical Notes on Strassen's Algorithm

While Strassen's algorithm has a better theoretical complexity ($O(n^{2.81})$ compared to $O(n^3)$), in practice we observed that it can be slower than the naïve and recursive methods for smaller input sizes. This is due to:

- *Overhead of recursive function calls.*
- *Extra memory allocations for temporary submatrices.*
- *Increased number of additions/subtractions, which outweigh the savings in multiplications for small n.*
- *Poorer cache locality compared to the simple iterative method.*

Therefore, Strassen's algorithm usually only becomes faster for ***very large*** *n (in the order of thousands). On typical laptops, the cross-over point where Strassen starts to outperform classical methods is usually between $n = 512$ and $n = 2048$.*

Conclusion

- *The classical iterative and recursive approaches both run in $O(n^3)$.*
- *Strassen's algorithm asymptotically improves runtime to $O(n^{2.81})$.*
- *For small n, iterative may outperform Strassen due to overhead.*