

# 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  $C = A \times B$

```

1: for  $i \leftarrow 0$  to  $n - 1$  do
2:   for  $j \leftarrow 0$  to  $n - 1$  do
3:      $C[i][j] \leftarrow 0$ 
4:     for  $k \leftarrow 0$  to  $n - 1$  do
5:        $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$ 
6:     end for
7:   end for
8: end for

```

---

*C Code:*

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  int mat_mul(int n, int A[n][n], int B[n][n], int C[n][n])
5  {
6      for (int i = 0; i < n; i++)
7      {
8          for (int j = 0; j < n; j++)
9          {
10             C[i][j] = 0;
11             for (int k = 0; k < n; k++)
12             {
13                 C[i][j] += A[i][k] * B[k][j];
14             }
15         }
16     }
17 }
18 void fill_mat(int n, int Mat[n][n])
19 {
20     for (int i = 0; i < n; i++)
21     {
22         for (int j = 0; j < n; j++)
23         {
24             Mat[i][j] = rand() % 10;
25         }
26     }

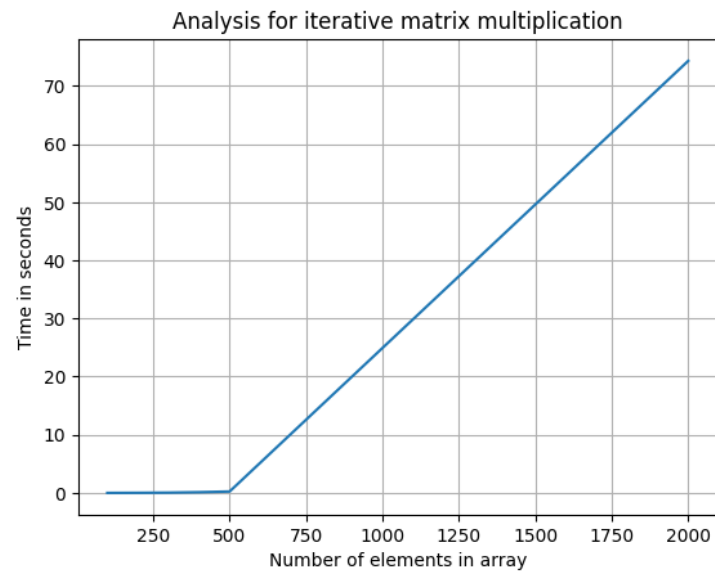
```

```

27 }
28 int main()
29 {
30     srand(time(NULL));
31     int size[] = {100, 200, 300, 400, 500};
32     int num = sizeof(size) / sizeof(size[0]);
33     for (int i = 0; i < num; i++)
34     {
35         int n = size[i];
36         int (*A)[n] = malloc(sizeof(int[n][n]));
37         int (*B)[n] = malloc(sizeof(int[n][n]));
38         int (*C)[n] = malloc(sizeof(int[n][n]));
39
40         if (A == NULL || B == NULL || C == NULL)
41         {
42             printf("Memory allocation failed for size %d\n", n);
43             return(1);
44         }
45         fill_mat(n, A);
46         fill_mat(n, B);
47
48         clock_t start = clock();
49         mat_mul(n, A, B, C);
50         clock_t end = clock();
51
52         double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
53
54         printf("Matrix size: %d x %d / Execution time: %f seconds\n", n, n,
55
56             free(A);
57             free(B);
58             free(C);
59     }
60
61     return 0;
62 }

```

**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

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

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:*

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  void mat_mul(int n, int A[n][n], int B[n][n], int C[n][n])
5  {
6      if (n < 2) {
7          C[0][0] += A[0][0] * B[0][0];
8          return;
9      }
10
11     int k = n / 2;
12
13     int (*A11)[k] = malloc(sizeof(int[k][k]));
14     int (*A12)[k] = malloc(sizeof(int[k][k]));
15     int (*A21)[k] = malloc(sizeof(int[k][k]));
16     int (*A22)[k] = malloc(sizeof(int[k][k]));
17
18     int (*B11)[k] = malloc(sizeof(int[k][k]));
19     int (*B12)[k] = malloc(sizeof(int[k][k]));
20     int (*B21)[k] = malloc(sizeof(int[k][k]));
21     int (*B22)[k] = malloc(sizeof(int[k][k]));
22
23     int (*C11)[k] = malloc(sizeof(int[k][k]));
24     int (*C12)[k] = malloc(sizeof(int[k][k]));
25     int (*C21)[k] = malloc(sizeof(int[k][k]));
26     int (*C22)[k] = malloc(sizeof(int[k][k]));
27

```

```

28
29     for (int i = 0; i < k; i++)
30         for (int j = 0; j < k; j++) {
31             C11[i][j] = 0;
32             C12[i][j] = 0;
33             C21[i][j] = 0;
34             C22[i][j] = 0;
35         }
36
37     // Divide A and B into 4 parts
38     for (int i = 0; i < k; i++) {
39         for (int j = 0; j < k; j++) {
40             A11[i][j] = A[i][j];
41             A12[i][j] = A[i][j + k];
42             A21[i][j] = A[i + k][j];
43             A22[i][j] = A[i + k][j + k];
44
45             B11[i][j] = B[i][j];
46             B12[i][j] = B[i][j + k];
47             B21[i][j] = B[i + k][j];
48             B22[i][j] = B[i + k][j + k];
49         }
50     }
51
52
53     mat_mul(k, A11, B11, C11);
54     mat_mul(k, A12, B21, C11);
55
56     mat_mul(k, A11, B12, C12);
57     mat_mul(k, A12, B22, C12);
58
59     mat_mul(k, A21, B11, C21);
60     mat_mul(k, A22, B21, C21);
61
62     mat_mul(k, A21, B12, C22);
63     mat_mul(k, A22, B22, C22);
64
65     for (int i = 0; i < k; i++) {
66         for (int j = 0; j < k; j++) {
67             C[i][j] = C11[i][j];
68             C[i][j + k] = C12[i][j];
69             C[i + k][j] = C21[i][j];
70             C[i + k][j + k] = C22[i][j];
71         }
72     }
73
74     free(A11); free(A12); free(A21); free(A22);
75     free(B11); free(B12); free(B21); free(B22);
76     free(C11); free(C12); free(C21); free(C22);
77 }
78

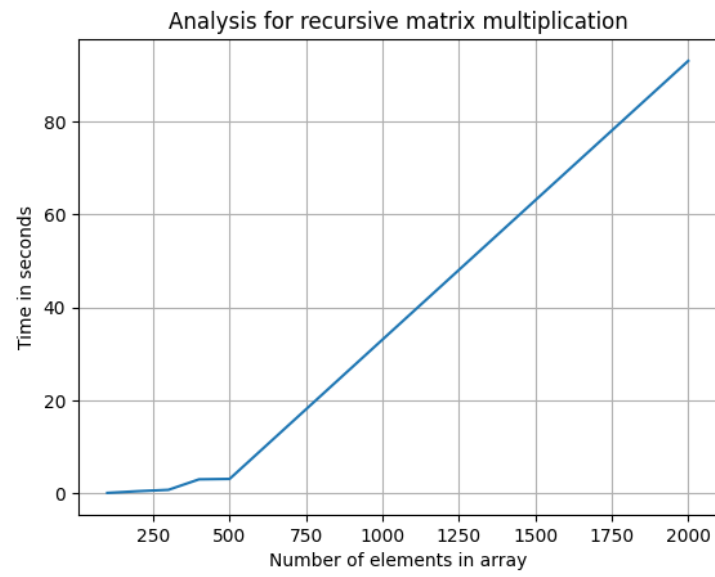
```

```

79
80 void fill_mat(int n, int Mat[n][n])
81 {
82     for (int i = 0; i < n; i++)
83     {
84         for (int j = 0; j < n; j++)
85         {
86             Mat[i][j] = rand() % 10;
87         }
88     }
89 }
90 int main()
91 {
92     srand(time(NULL));
93     int size[] = {100, 200, 300, 400, 500};
94     int num = sizeof(size) / sizeof(size[0]);
95     for (int i = 0; i < num; i++)
96     {
97         int n = size[i];
98         int (*A)[n] = malloc(sizeof(int[n][n]));
99         int (*B)[n] = malloc(sizeof(int[n][n]));
100        int (*C)[n] = malloc(sizeof(int[n][n]));
101
102        if (A == NULL || B == NULL || C == NULL)
103        {
104            printf("Memory allocation failed for size %d\n", n);
105            return(1);
106        }
107        fill_mat(n, A);
108        fill_mat(n, B);
109
110        clock_t start = clock();
111        mat_mul(n, A, B, C);
112        clock_t end = clock();
113
114        double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
115
116        printf("Matrix size: %d x %d | Execution time: %f seconds\n", n, n,
117
118        free(A);
119        free(B);
120        free(C);
121    }
122
123    return 0;
124 }

```

**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  $n \times n$  ( $n$  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

### 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  $M_1, M_2, \dots, M_7$ 
    Form  $C_{11}, C_{12}, C_{21}, C_{22}$ 
end if

```

---

**C Code:**

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void fill_matrix(int n, int **M) {
6      for (int i = 0; i < n; i++)
7          for (int j = 0; j < n; j++)
8              M[i][j] = rand() % 10;
9  }
10
11 void print_matrix(int n, int **M) {
12     for (int i = 0; i < n; i++) {
13         for (int j = 0; j < n; j++)
14             printf("%4d ", M[i][j]);
15         printf("\n");
16     }
17     printf("\n");
18 }
19
20 int **alloc_matrix(int n) {
21     int **M = malloc(n * sizeof(int *));
22     for (int i = 0; i < n; i++)
23         M[i] = calloc(n, sizeof(int)); // initialise to 0

```

```

24     return M;
25 }
26
27 void free_matrix(int n, int **M) {
28     for (int i = 0; i < n; i++) free(M[i]);
29     free(M);
30 }
31
32 void add_matrix(int n, int **A, int **B, int **C) {
33     for (int i = 0; i < n; i++)
34         for (int j = 0; j < n; j++)
35             C[i][j] = A[i][j] + B[i][j];
36 }
37
38 void sub_matrix(int n, int **A, int **B, int **C) {
39     for (int i = 0; i < n; i++)
40         for (int j = 0; j < n; j++)
41             C[i][j] = A[i][j] - B[i][j];
42 }
43
44 void naive_mul(int n, int **A, int **B, int **C) {
45     for (int i = 0; i < n; i++)
46         for (int j = 0; j < n; j++) {
47             C[i][j] = 0;
48             for (int k = 0; k < n; k++)
49                 C[i][j] += A[i][k] * B[k][j];
50         }
51 }
52
53
54 void strassen(int n, int **A, int **B, int **C) {
55     if (n <= 2) { // base case: use naive multiplication
56         naive_mul(n, A, B, C);
57         return;
58     }
59
60     int k = n / 2;
61
62
63     int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k);
64     int **A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
65     int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k);
66     int **B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
67     int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k);
68     int **C21 = alloc_matrix(k), **C22 = alloc_matrix(k);
69
70     int **M1 = alloc_matrix(k), **M2 = alloc_matrix(k), **M3 = alloc_matrix(k);
71     int **M4 = alloc_matrix(k), **M5 = alloc_matrix(k), **M6 = alloc_matrix(k);
72     int **T1 = alloc_matrix(k), **T2 = alloc_matrix(k);
73
74     // split A and B into 4 parts

```

```

75     for (int i = 0; i < k; i++) {
76         for (int j = 0; j < k; j++) {
77             A11[i][j] = A[i][j];
78             A12[i][j] = A[i][j + k];
79             A21[i][j] = A[i + k][j];
80             A22[i][j] = A[i + k][j + k];
81
82             B11[i][j] = B[i][j];
83             B12[i][j] = B[i][j + k];
84             B21[i][j] = B[i + k][j];
85             B22[i][j] = B[i + k][j + k];
86         }
87     }
88
89     // M1 = (A11 + A22) * (B11 + B22)
90     add_matrix(k, A11, A22, T1);
91     add_matrix(k, B11, B22, T2);
92     strassen(k, T1, T2, M1);
93
94     // M2 = (A21 + A22) * B11
95     add_matrix(k, A21, A22, T1);
96     strassen(k, T1, B11, M2);
97
98     // M3 = A11 * (B12 - B22)
99     sub_matrix(k, B12, B22, T2);
100    strassen(k, A11, T2, M3);
101
102    // M4 = A22 * (B21 - B11)
103    sub_matrix(k, B21, B11, T2);
104    strassen(k, A22, T2, M4);
105
106    // M5 = (A11 + A12) * B22
107    add_matrix(k, A11, A12, T1);
108    strassen(k, T1, B22, M5);
109
110    // M6 = (A21 - A11) * (B11 + B12)
111    sub_matrix(k, A21, A11, T1);
112    add_matrix(k, B11, B12, T2);
113    strassen(k, T1, T2, M6);
114
115    // M7 = (A12 - A22) * (B21 + B22)
116    sub_matrix(k, A12, A22, T1);
117    add_matrix(k, B21, B22, T2);
118    strassen(k, T1, T2, M7);
119
120    // C11 = M1 + M4 - M5 + M7
121    for (int i = 0; i < k; i++)
122        for (int j = 0; j < k; j++)
123            C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
124
125    // C12 = M3 + M5

```

```

126     for (int i = 0; i < k; i++)
127         for (int j = 0; j < k; j++)
128             C12[i][j] = M3[i][j] + M5[i][j];
129
130     // C21 = M2 + M4
131     for (int i = 0; i < k; i++)
132         for (int j = 0; j < k; j++)
133             C21[i][j] = M2[i][j] + M4[i][j];
134
135     // C22 = M1 - M2 + M3 + M6
136     for (int i = 0; i < k; i++)
137         for (int j = 0; j < k; j++)
138             C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
139
140     // join C
141     for (int i = 0; i < k; i++) {
142         for (int j = 0; j < k; j++) {
143             C[i][j] = C11[i][j];
144             C[i][j + k] = C12[i][j];
145             C[i + k][j] = C21[i][j];
146             C[i + k][j+k] = C22[i][j];
147         }
148     }
149
150     // free memory
151     free_matrix(k, A11); free_matrix(k, A12); free_matrix(k, A21); free_matr
152     free_matrix(k, B11); free_matrix(k, B12); free_matrix(k, B21); free_matr
153     free_matrix(k, C11); free_matrix(k, C12); free_matrix(k, C21); free_matr
154     free_matrix(k, M1); free_matrix(k, M2); free_matrix(k, M3);
155     free_matrix(k, M4);
156     free_matrix(k, M5); free_matrix(k, M6); free_matrix(k, M7);
157     free_matrix(k, T1); free_matrix(k, T2);
158 }
159
160 int main() {
161     int size[] = {100, 200, 300, 400, 500};
162     int num = sizeof(size) / sizeof(size[0]);
163     for (int i = 0; i < num; i++)
164     {
165         srand(time(NULL));
166         int n = size[i];
167
168         int **A = alloc_matrix(n);
169         int **B = alloc_matrix(n);
170         int **C = alloc_matrix(n);
171
172         fill_matrix(n, A);
173         fill_matrix(n, B);
174
175         clock_t start = clock();

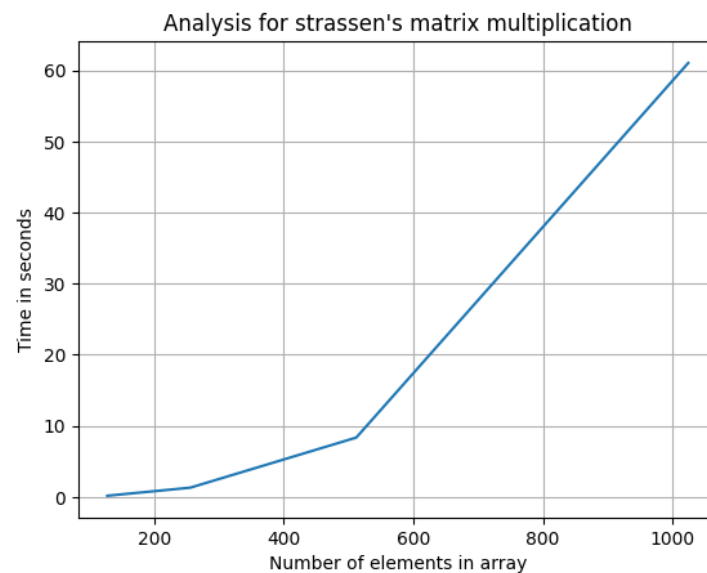
```

```

176     strassen(n, A, B, C);
177     clock_t end = clock();
178
179     double secs = (double)(end - start) / CLOCKS_PER_SEC;
180     printf("Matrix size %d x %d / Time taken: %f seconds\n", n, n, secs);
181
182     // print_matrix(n, C);
183
184     free_matrix(n, A);
185     free_matrix(n, B);
186     free_matrix(n, C);
187 }
188     return 0;
189 }

```

**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:**

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5
6 int **alloc_matrix(int n) {
7     int **M = malloc(n * sizeof(int *));
8     for (int i = 0; i < n; i++)
9         M[i] = calloc(n, sizeof(int));
10    return M;
11 }
12
13 void free_matrix(int n, int **M) {
14     for (int i = 0; i < n; i++)
15         free(M[i]);
16     free(M);
17 }
18
19 void fill_matrix(int n, int **M) {
20     for (int i = 0; i < n; i++)
21         for (int j = 0; j < n; j++)
22             M[i][j] = rand() % 10;
23 }
24
25 void iterative_mul(int n, int **A, int **B, int **C) {
26     for (int i = 0; i < n; i++)
27         for (int j = 0; j < n; j++) {
28             C[i][j] = 0;
29             for (int k = 0; k < n; k++)
30                 C[i][j] += A[i][k] * B[k][j];
31         }
32 }
33
34 void recursive_mul(int n, int **A, int **B, int **C) {
35     if (n == 1) {
36         C[0][0] += A[0][0] * B[0][0];
37         return;
38     }
39     int k = n / 2;
40     int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k);
41     int **A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
42     int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k);
43     int **B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
44     int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k);
45     int **C21 = alloc_matrix(k), **C22 = alloc_matrix(k);
46     for (int i = 0; i < k; i++) {

```

```

47     for (int j = 0; j < k; j++) {
48         A11[i][j] = A[i][j];
49         A12[i][j] = A[i][j + k];
50         A21[i][j] = A[i + k][j];
51         A22[i][j] = A[i + k][j + k];
52         B11[i][j] = B[i][j];
53         B12[i][j] = B[i][j + k];
54         B21[i][j] = B[i + k][j];
55         B22[i][j] = B[i + k][j + k];
56     }
57 }
58 recursive_mul(k, A11, B11, C11);
59 recursive_mul(k, A12, B21, C11);
60 recursive_mul(k, A11, B12, C12);
61 recursive_mul(k, A12, B22, C12);
62 recursive_mul(k, A21, B11, C21);
63 recursive_mul(k, A22, B21, C21);
64 recursive_mul(k, A21, B12, C22);
65 recursive_mul(k, A22, B22, C22);
66 for (int i = 0; i < k; i++) {
67     for (int j = 0; j < k; j++) {
68         C[i][j] = C11[i][j];
69         C[i][j + k] = C12[i][j];
70         C[i + k][j] = C21[i][j];
71         C[i + k][j + k] = C22[i][j];
72     }
73 }
74 free_matrix(k, A11);
75 free_matrix(k, A12);
76 free_matrix(k, A21);
77 free_matrix(k, A22);
78 free_matrix(k, B11);
79 free_matrix(k, B12);
80 free_matrix(k, B21);
81 free_matrix(k, B22);
82 free_matrix(k, C11);
83 free_matrix(k, C12);
84 free_matrix(k, C21);
85 free_matrix(k, C22);
86 }
87
88 void add_matrix(int n, int **A, int **B, int **C) {
89     for (int i = 0; i < n; i++)
90         for (int j = 0; j < n; j++)
91             C[i][j] = A[i][j] + B[i][j];
92 }
93
94 void sub_matrix(int n, int **A, int **B, int **C) {
95     for (int i = 0; i < n; i++)
96         for (int j = 0; j < n; j++)
97             C[i][j] = A[i][j] - B[i][j];

```

```

98 }
99
100 void naive_mul(int n, int **A, int **B, int **C) {
101     for (int i = 0; i < n; i++)
102         for (int j = 0; j < n; j++) {
103             C[i][j] = 0;
104             for (int k = 0; k < n; k++)
105                 C[i][j] += A[i][k] * B[k][j];
106         }
107 }
108
109 void strassen(int n, int **A, int **B, int **C) {
110     if (n <= 2) {
111         naive_mul(n, A, B, C);
112         return;
113     }
114     int k = n / 2;
115     int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k);
116     int **A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
117     int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k);
118     int **B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
119     int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k);
120     int **C21 = alloc_matrix(k), **C22 = alloc_matrix(k);
121     int **M1 = alloc_matrix(k), **M2 = alloc_matrix(k),
122     **M3 = alloc_matrix(k);
123     int **M4 = alloc_matrix(k), **M5 = alloc_matrix(k),
124     **M6 = alloc_matrix(k), **M7 = alloc_matrix(k);
125     int **T1 = alloc_matrix(k), **T2 = alloc_matrix(k);
126     for (int i = 0; i < k; i++) {
127         for (int j = 0; j < k; j++) {
128             A11[i][j] = A[i][j];
129             A12[i][j] = A[i][j + k];
130             A21[i][j] = A[i + k][j];
131             A22[i][j] = A[i + k][j + k];
132             B11[i][j] = B[i][j];
133             B12[i][j] = B[i][j + k];
134             B21[i][j] = B[i + k][j];
135             B22[i][j] = B[i + k][j + k];
136         }
137     }
138     add_matrix(k, A11, A22, T1);
139     add_matrix(k, B11, B22, T2);
140     strassen(k, T1, T2, M1);
141     add_matrix(k, A21, A22, T1);
142     strassen(k, T1, B11, M2);
143     sub_matrix(k, B12, B22, T2);
144     strassen(k, A11, T2, M3);
145     sub_matrix(k, B21, B11, T2);
146     strassen(k, A22, T2, M4);
147     add_matrix(k, A11, A12, T1);
148     strassen(k, T1, B22, M5);

```



```

149     sub_matrix(k, A21, A11, T1);
150     add_matrix(k, B11, B12, T2);
151     strassen(k, T1, T2, M6);
152     sub_matrix(k, A12, A22, T1);
153     add_matrix(k, B21, B22, T2);
154     strassen(k, T1, T2, M7);
155     for (int i = 0; i < k; i++)
156         for (int j = 0; j < k; j++)
157             C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
158     for (int i = 0; i < k; i++)
159         for (int j = 0; j < k; j++)
160             C12[i][j] = M3[i][j] + M5[i][j];
161     for (int i = 0; i < k; i++)
162         for (int j = 0; j < k; j++)
163             C21[i][j] = M2[i][j] + M4[i][j];
164     for (int i = 0; i < k; i++)
165         for (int j = 0; j < k; j++)
166             C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
167     for (int i = 0; i < k; i++) {
168         for (int j = 0; j < k; j++) {
169             C[i][j] = C11[i][j];
170             C[i][j + k] = C12[i][j];
171             C[i + k][j] = C21[i][j];
172             C[i + k][j + k] = C22[i][j];
173         }
174     }
175     free_matrix(k, A11); free_matrix(k, A12); free_matrix(k, A21); free_matr
176     free_matrix(k, B11); free_matrix(k, B12); free_matrix(k, B21); free_matr
177     free_matrix(k, C11); free_matrix(k, C12); free_matrix(k, C21); free_matr
178     free_matrix(k, M1); free_matrix(k, M2); free_matrix(k, M3); free_matrix(
179     free_matrix(k, M5); free_matrix(k, M6); free_matrix(k, M7);
180     free_matrix(k, T1); free_matrix(k, T2);
181 }
182
183 int main() {
184     int sizes[] = {128, 256, 512, 1024};
185     int num = sizeof(sizes) / sizeof(sizes[0]);
186     printf("%10s %15s %15s %15s\n", "n", "Iterative", "Recursive", "Strassen
187     for (int i = 0; i < num; i++) {
188         int n = sizes[i];
189         int **A = alloc_matrix(n);
190         int **B = alloc_matrix(n);
191         int **C = alloc_matrix(n);
192         fill_matrix(n, A);
193         fill_matrix(n, B);
194
195         clock_t start, end;
196         double t_iter, t_rec, t_strassen;
197         start = clock();
198         iterative_mul(n, A, B, C);
199         end = clock();

```

```

200     t_iter = (double)(end - start) / CLOCKS_PER_SEC;
201
202     start = clock();
203     recursive_mul(n, A, B, C);
204     end = clock();
205     t_rec = (double)(end - start) / CLOCKS_PER_SEC;
206
207     start = clock();
208     strassen(n, A, B, C);
209     end = clock();
210     t_strassen = (double)(end - start) / CLOCKS_PER_SEC;
211
212     printf("%10d %15.6f %15.6f %15.6f\n", n, t_iter, t_rec, t_strassen);
213     free_matrix(n, A);
214     free_matrix(n, B);
215     free_matrix(n, C);
216 }
217 return 0;
218 }

```

### 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.