

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.

Code:

```
C matrixit.c > ...
1 #include <stdio.h>
2 #include <time.h>
3
4 #define MAX 200 // maximum size of matrix
5
6 int main() {
7     int n;
8     int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
9     clock_t start, end;
10    double cpu_time_used;
11
12    printf("Enter matrix size (n x n, n <= %d): ", MAX);
13    scanf("%d", &n);
14
15    if (n > MAX) {
16        printf("Matrix size too large! Use <= %d\n", MAX);
17        return 1;
18    }
19
20    // Initialize matrices with some values
21    printf("\nFilling matrices A and B with sample values...\n");
22    for (int i = 0; i < n; i++) {
23        for (int j = 0; j < n; j++) {
24            A[i][j] = i + j;           // example values
25            B[i][j] = i - j;           // example values
26            C[i][j] = 0;                // initialize result
27        }
28    }
29
30    // Measure start time
31    start = clock();
32
33    // Matrix multiplication (iterative)
34    for (int i = 0; i < n; i++) {
35        for (int j = 0; j < n; j++) {
36            for (int k = 0; k < n; k++) {
37                C[i][j] += A[i][k] * B[k][j];
38            }
39        }
40    }
41
42    // Measure end time
43    end = clock();
44
45    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
46
```

```

6 int main() {
46
47     printf("\nMatrix multiplication completed.\n");
48     printf("Execution time for %dx%d matrix: %f seconds\n", n, n, cpu_time_used);
49
50     // Print small result for verification
51     if (n <= 5) {
52         printf("\nResultant Matrix C:\n");
53         for (int i = 0; i < n; i++) {
54             for (int j = 0; j < n; j++) {
55                 printf("%5d ", C[i][j]);
56             }
57             printf("\n");
58         }
59     } else {
60         printf("Result not printed (too large).\n");
61     }
62
63     return 0;
64 }
65

```

Output:

```

PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "c:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix\" ; if ($?) { gcc matrixit.c -o matrixit } ; if ($?)
{ .\matrixit }
Enter matrix size (n x n, n <= 200): 2
Filling matrices A and B with sample values...

Matrix multiplication completed.
Execution time for 2x2 matrix: 0.000000 seconds

Resultant Matrix C:
 1   0
 2  -1
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "c:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix\" ; if ($?) { gcc matrixit.c -o matrixit } ; if ($?)
{ .\matrixit }
Enter matrix size (n x n, n <= 200): 3
Filling matrices A and B with sample values...

Matrix multiplication completed.
Execution time for 3x3 matrix: 0.000000 seconds

Resultant Matrix C:
 5   2   -1
 8   2   -4
11   2   -7

```

Python code:

```
tricky / ...
import time
import matplotlib.pyplot as plt
import numpy as np

def multiply_matrices(n):
    # Initialize matrices
    A = np.fromfunction(lambda i, j: i + j, (n, n), dtype=int)
    B = np.fromfunction(lambda i, j: i - j, (n, n), dtype=int)
    C = np.zeros((n, n), dtype=int)

    # Measure time
    start = time.time()
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    end = time.time()

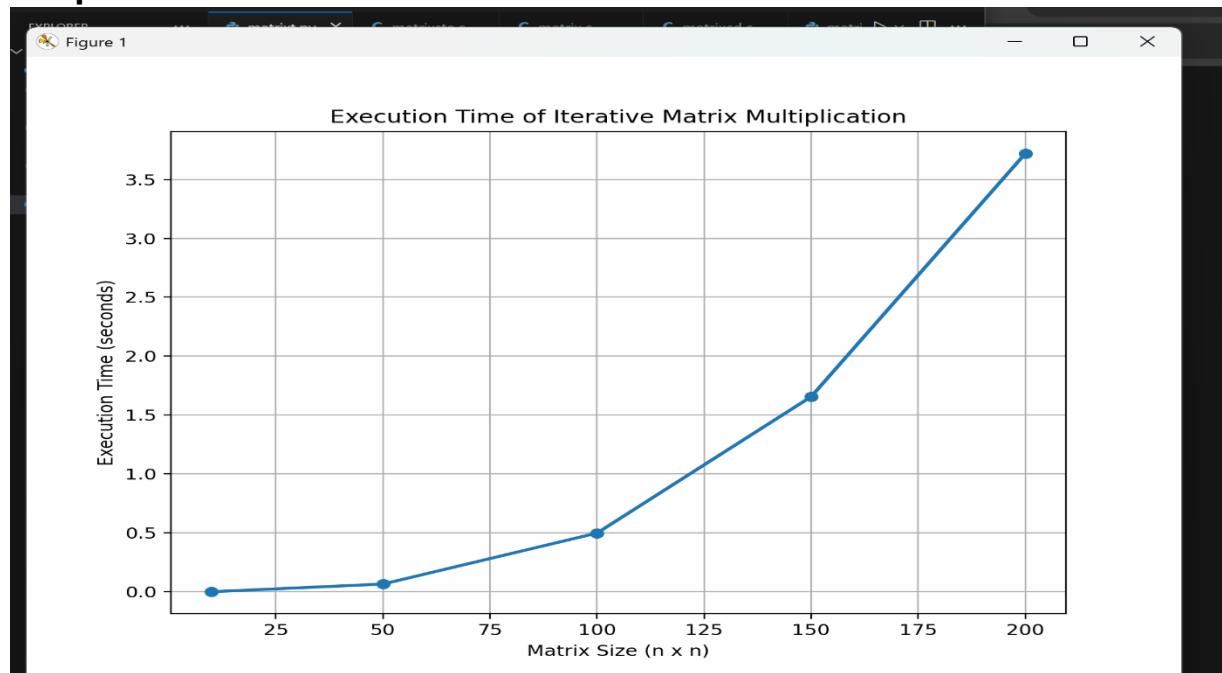
    return end - start

# Matrix sizes to test (adjust as needed)
sizes = [10, 50, 100, 150, 200]
times = []

for n in sizes:
    print(f"Running for {n}x{n}...")
    t = multiply_matrices(n)
    times.append(t)

# Plot results
plt.figure(figsize=(8, 6))
plt.plot(sizes, times, marker='o', linestyle='-', linewidth=2)
plt.title("Execution Time of Iterative Matrix Multiplication")
plt.xlabel("Matrix Size (n x n)")
plt.ylabel("Execution Time (seconds)")
plt.grid(True)
plt.show()
```

Graph:



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

Code:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5
6 void add(int n, int A[n][n], int B[n][n], int C[n][n]) {
7     for (int i = 0; i < n; i++) {
8         for (int j = 0; j < n; j++) {
9             C[i][j] = A[i][j] + B[i][j];
10    }
11
12    |
13    void multiply(int n, int A[n][n], int B[n][n], int C[n][n]) {
14        if (n == 2) { // Base case
15            C[0][0] = A[0][0]*B[0][0] + A[0][1]*B[1][0];
16            C[0][1] = A[0][0]*B[0][1] + A[0][1]*B[1][1];
17            C[1][0] = A[1][0]*B[0][0] + A[1][1]*B[1][0];
18            C[1][1] = A[1][0]*B[0][1] + A[1][1]*B[1][1];
19            return;
20        }
21
22        int k = n/2;
23
24        int A11[k][k], A12[k][k], A21[k][k], A22[k][k];
25        int B11[k][k], B12[k][k], B21[k][k], B22[k][k];
26        int C11[k][k], C12[k][k], C21[k][k], C22[k][k];
27        int M1[k][k], M2[k][k];
28
29        // Initialize result submatrices
30        for (int i=0; i<k; i++) {
31            for (int j=0; j<k; j++) {
32                C11[i][j]=C12[i][j]=C21[i][j]=C22[i][j]=0;
33
34        // Divide A and B into submatrices
35        for (int i=0; i<k; i++) {
36            for (int j=0; j<k; j++) {
37                A11[i][j] = A[i][j];

```

```
matrix.c > ...
3 void multiply(int n, int A[n][n], int B[n][n], int C[n][n]) {
5     for (int i=0; i<k; i++) {
6         for (int j=0; j<k; j++) {
8             A12[i][j] = A[i][j+k];
9             A21[i][j] = A[i+k][j];
10            A22[i][j] = A[i+k][j+k];
12
13            B11[i][j] = B[i][j];
14            B12[i][j] = B[i][j+k];
15            B21[i][j] = B[i+k][j];
16            B22[i][j] = B[i+k][j+k];
17        }
18    }
19
20    // C11 = A11*B11 + A12*B21
21    multiply(k, A11, B11, M1);
22    multiply(k, A12, B21, M2);
23    add(k, M1, M2, C11);
24
25    // C12 = A11*B12 + A12*B22
26    multiply(k, A11, B12, M1);
27    multiply(k, A12, B22, M2);
28    add(k, M1, M2, C12);
29
30    // C21 = A21*B11 + A22*B21
31    multiply(k, A21, B11, M1);
32    multiply(k, A22, B21, M2);
33    add(k, M1, M2, C21);
34
35    // C22 = A21*B12 + A22*B22
36    multiply(k, A21, B12, M1);
37    multiply(k, A22, B22, M2);
38    add(k, M1, M2, C22);
39
40    // Assemble result matrix C
41    for (int i=0; i<k; i++) {
42        for (int j=0; j<k; j++) {
43            C[i][j] = A11[i][j] * B11[j][i] + A12[i][j] * B21[j][i] + A21[i][j] * B11[j][i] + A22[i][j] * B21[j][i];
44        }
45    }
46}
```

```

13 void multiply(int n, int A[n][n], int B[n][n], int C[n][n]) {
70     for (int i=0; i<k; i++) {
71         for (int j=0; j<k; j++) {
73             C[i][j+k] = C12[i][j];
74             C[i+k][j] = C21[i][j];
75             C[i+k][j+k] = C22[i][j];
76         }
77     }
78 }
79
80 // Function to print a matrix
81 void printMatrix(int n, int M[n][n]) {
82     for (int i=0; i<n; i++) {
83         for (int j=0; j<n; j++)
84             printf("%d ", M[i][j]);
85         printf("\n");
86     }
87 }
88
89 int main() {
90     int sizes[] = {2, 4, 8}; // test different sizes
91     int numSizes = 3;
92
93     for (int s = 0; s < numSizes; s++) {
94         int n = sizes[s];
95         int A[n][n], B[n][n], C[n][n];
96
97         // Fill matrices with sample values
98         for (int i=0; i<n; i++) {
99             for (int j=0; j<n; j++) {
100                 A[i][j] = i + j;
101                 B[i][j] = i - j;
102                 C[i][j] = 0;
103             }
104         }
105
106         clock_t start = clock();
107
108         int main() {
109             for (int s = 0; s < numSizes; s++) {
110                 multiply(n, A, B, C);
111                 clock_t end = clock();
112
113                 double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
114
115                 printf("\nMatrix size: %dx%d\n", n, n);
116                 if (n <= 4) { // print small matrices only
117                     printf("Matrix A:\n"); printMatrix(n, A);
118                     printf("Matrix B:\n"); printMatrix(n, B);
119                     printf("Result Matrix C:\n"); printMatrix(n, C);
120                 } else {
121                     printf("Matrices too large to print.\n");
122                 }
123                 printf("Execution time: %f seconds\n", time_taken);
124             }
125
126             return 0;
127         }
128     }

```

Output:

```
Matrix size: 2x2
Matrix A:
0 1
1 2
Matrix B:
0 -1
1 0
Result Matrix C:
1 0
2 -1
Execution time: 0.000000 seconds

Matrix size: 4x4
Matrix A:
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
Matrix B:
0 -1 -2 -3
1 0 -1 -2
2 1 0 -1
3 2 1 0
Result Matrix C:
14 8 2 -4
20 10 0 -10
26 12 -2 -16
32 14 -4 -22
Execution time: 0.000000 seconds

Matrix size: 8x8
Matrices too large to print.
Execution time: 0.000000 seconds
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix>
```

Python code:

```
import matplotlib.pyplot as plt

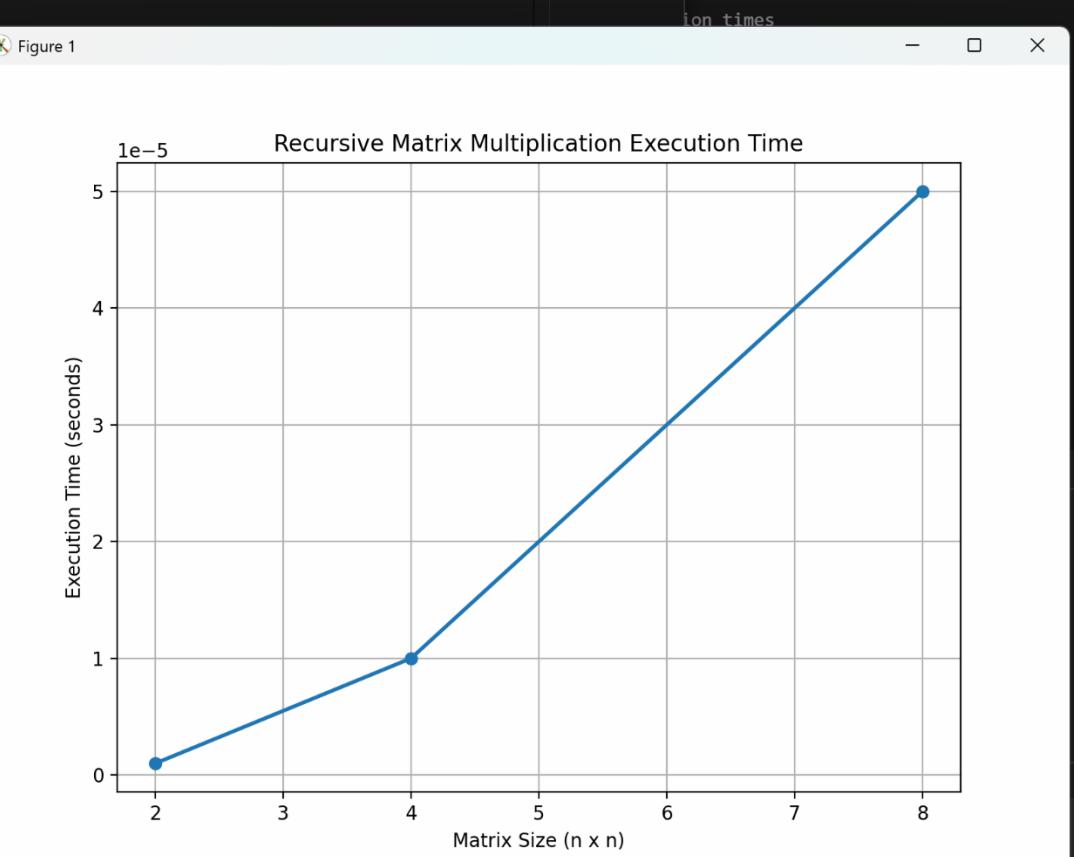
# Matrix sizes tested in the C code
sizes = [2, 4, 8]

# Replace these times with the actual execution times
# printed by your C program
execution_times = [0.000001, 0.00001, 0.00005]

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(sizes, execution_times, marker='o', linestyle='-', linewidth=2)

plt.title("Recursive Matrix Multiplication Execution Time")
plt.xlabel("Matrix Size (n x n)")
plt.ylabel("Execution Time (seconds)")
plt.grid(True)
plt.show()
```

Graph:



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.

Code

:

```

C matrixsta.c > ...
1  #include <stdio.h>
2  #include <time.h>
3
4  #define MAX 64 // maximum matrix size (must be power of 2, e.g., 2,4,8,16,32,64)
5
6  // Function to add two matrices
7  void add(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
8      for (int i = 0; i < n; i++) {
9          for (int j = 0; j < n; j++) {
10             C[i][j] = A[i][j] + B[i][j];
11         }
12     }
13
14 // Function to subtract two matrices
15 void subtract(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
16     for (int i = 0; i < n; i++) {
17         for (int j = 0; j < n; j++) {
18             C[i][j] = A[i][j] - B[i][j];
19         }
20     }
21
22 // Recursive Strassen's algorithm
23 void strassen(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
24     if (n == 1) { // base case: single element multiplication
25         C[0][0] = A[0][0] * B[0][0];
26         return;
27     }
28
29     int half = n / 2;
30     int A11[MAX][MAX], A12[MAX][MAX], A21[MAX][MAX], A22[MAX][MAX];
31     int B11[MAX][MAX], B12[MAX][MAX], B21[MAX][MAX], B22[MAX][MAX];
32     int C11[MAX][MAX], C12[MAX][MAX], C21[MAX][MAX], C22[MAX][MAX];
33     int M1[MAX][MAX], M2[MAX][MAX], M3[MAX][MAX], M4[MAX][MAX], M5[MAX][MAX], M6[MAX][MAX], M7[MAX][MAX];
34     int temp1[MAX][MAX], temp2[MAX][MAX];
35
36     // Split matrices into submatrices
37     for (int i = 0; i < half; i++) {
38         for (int j = 0; j < half; j++) {
39             A11[i][j] = A[i][j];
40             A12[i][j] = A[i][j + half];
41             A21[i][j] = A[i + half][j];
42             A22[i][j] = A[i + half][j + half];
43
44             B11[i][j] = B[i][j];
45             B12[i][j] = B[i][j + half];
46             B21[i][j] = B[i + half][j];
47             B22[i][j] = B[i + half][j + half];
48         }
49     }
50
51     M1 = add(A11, B11);
52     M2 = add(A11, B21);
53     M3 = add(A12, B11);
54     M4 = add(A12, B21);
55     M5 = add(A21, B11);
56     M6 = add(A21, B21);
57     M7 = add(A22, B11);
58
59     C11 = subtract(M2, M3);
60     C12 = add(C11, M4);
61     C21 = add(C11, M5);
62     C22 = add(C12, C21);
63
64     for (int i = 0; i < half; i++) {
65         for (int j = 0; j < half; j++) {
66             C[i][j] = C22[i][j];
67         }
68     }
69
70     strassen(half, A11, B11, M1);
71     strassen(half, A11, B21, M2);
72     strassen(half, A12, B11, M3);
73     strassen(half, A12, B21, M4);
74     strassen(half, A21, B11, M5);
75     strassen(half, A21, B21, M6);
76     strassen(half, A22, B11, M7);
77
78     for (int i = 0; i < half; i++) {
79         for (int j = 0; j < half; j++) {
80             C[i][j] = C[i][j] + M7[i][j];
81         }
82     }
83
84     free(A11);
85     free(A12);
86     free(A21);
87     free(A22);
88     free(B11);
89     free(B12);
90     free(B21);
91     free(B22);
92     free(M1);
93     free(M2);
94     free(M3);
95     free(M4);
96     free(M5);
97     free(M6);
98     free(M7);
99 }
```

```
trixsta.c > ⚙ strassen(int, int [MAX][MAX], int [MAX][MAX], int [MAX][MAX])
void strassen(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
    for (int i = 0; i < half; i++) {
        for (int j = 0; j < half; j++) {
    }

    // M1 = (A11 + A22)(B11 + B22)
    add(half, A11, A22, temp1);
    add(half, B11, B22, temp2);
    strassen(half, temp1, temp2, M1);

    // M2 = (A21 + A22)B11
    add(half, A21, A22, temp1);
    strassen(half, temp1, B11, M2);

    // M3 = A11(B12 - B22)
    subtract(half, B12, B22, temp1);
    strassen(half, A11, temp1, M3);

    // M4 = A22(B21 - B11)
    subtract(half, B21, B11, temp1);
    strassen(half, A22, temp1, M4);

    // M5 = (A11 + A12)B22
    add(half, A11, A12, temp1);
    strassen(half, temp1, B22, M5);

    // M6 = (A21 - A11)(B11 + B12)
    subtract(half, A21, A11, temp1);
    add(half, B11, B12, temp2);
    strassen(half, temp1, temp2, M6);

    // M7 = (A12 - A22)(B21 + B22)
    subtract(half, A12, A22, temp1);
    add(half, B21, B22, temp2);
    strassen(half, temp1, temp2, M7);

    // C11 = M1 + M4 - M5 + M7
    add(half, M1, M4, temp1);
    subtract(half, temp1, M5, temp2);
    add(half, temp2, M7, C11);

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

    // C21 = M2 + M4
    add(half, M2, M4, C21);
```

```

C matrixsta.c > strassen(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX])
21 void strassen(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
94     add(half, temp2, M6, C22);
95
96     // Combine results into C
97     for (int i = 0; i < half; i++) {
98         for (int j = 0; j < half; j++) {
99             C[i][j] = C11[i][j];
100            C[i][j + half] = C12[i][j];
101            C[i + half][j] = C21[i][j];
102            C[i + half][j + half] = C22[i][j];
103        }
104    }
105 }
106
107 int main() {
108     int n;
109     int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
110     clock_t start, end;
111     double cpu_time_used;
112
113     printf("Enter matrix size (power of 2, <= %d): ", MAX);
114     scanf("%d", &n);
115
116     // Fill matrices with sample values
117     for (int i = 0; i < n; i++) {
118         for (int j = 0; j < n; j++) {
119             A[i][j] = i + j; // sample values
120             B[i][j] = i - j;
121             C[i][j] = 0;
122         }
123     }
124
125     start = clock();
126     strassen(n, A, B, C);
127     end = clock();
128
129     cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
130
131     printf("\nResultant Matrix C:\n");
132     if (n <= 4) { // print small matrix only
133         for (int i = 0; i < n; i++) {
134             for (int j = 0; j < n; j++) {
135                 printf("%5d ", C[i][j]);
136             }
137             printf("\n");
138         }
139     } else {
140         printf("Matrix too large to display.\n");
141     }
142
143     printf("\nExecution time for %dx%d matrix: %f seconds\n", n, n, cpu_time_used);
144
145     return 0;
146 }
147

```

Output:

```
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix" & if ($?) { gcc matrixsta.c -o matrixsta } ; if ($?) { .\matrixsta }

Resultant Matrix C:
 1   0
 2  -1

Execution time for 2x2 matrix: 0.001000 seconds
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix" & if ($?) { gcc matrixsta.c -o matrixsta } ; if ($?) { .\matrixsta }

Enter matrix size (power of 2, <= 64): 4

Resultant Matrix C:
 14    8    2   -4
 20   10    0  -10
 26   12   -2  -16
 32   14   -4  -22

Execution time for 4x4 matrix: 0.001000 seconds
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix" & if ($?) { gcc matrixsta.c -o matrixsta } ; if ($?) { .\matrixsta }
```

Python code:

```
1 import time
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 # Matrix addition
6 def add(A, B):
7     return A + B
8
9 # Matrix subtraction
10 def subtract(A, B):
11     return A - B
12
13 # Strassen's Algorithm (recursive)
14 def strassen(A, B):
15     n = A.shape[0]
16     if n == 1:
17         return A * B
18
19     mid = n // 2
20
21     A11, A12, A21, A22 = A[:mid, :mid], A[:mid, mid:], A[mid:, :mid], A[mid:, mid:]
22     B11, B12, B21, B22 = B[:mid, :mid], B[:mid, mid:], B[mid:, :mid], B[mid:, mid:]
23
24     M1 = strassen(add(A11, A22), add(B11, B22))
25     M2 = strassen(add(A21, A22), B11)
26     M3 = strassen(A11, subtract(B12, B22))
27     M4 = strassen(A22, subtract(B21, B11))
28     M5 = strassen(add(A11, A12), B22)
29     M6 = strassen(subtract(A21, A11), add(B11, B12))
30     M7 = strassen(subtract(A12, A22), add(B21, B22))
31
32     C11 = add(subtract(add(M1, M4), M5), M7)
33     C12 = add(M3, M5)
34     C21 = add(M2, M4)
35     C22 = add(add(subtract(M1, M2), M3), M6)
36
37     # Combine results
```

```

def strassen(A, B):
    """Combines C11, C12, C21, C22 into C
    top = np.hstack((C11, C12))
    bottom = np.hstack((C21, C22))
    return np.vstack((top, bottom))

# Test different matrix sizes
sizes = [2, 4, 8, 16, 32, 64]
times = []

for n in sizes:
    A = np.fromfunction(lambda i, j: i + j, (n, n), dtype=int)
    B = np.fromfunction(lambda i, j: i - j, (n, n), dtype=int)

    start = time.time()
    C = strassen(A, B)
    end = time.time()

    elapsed = end - start
    times.append(elapsed)

    print(f"Size {n}x{n} -> {elapsed:.6f} seconds")

# Plot graph
plt.figure(figsize=(8,6))
plt.plot(sizes, times, marker='o', linestyle='--', linewidth=2)
plt.title("Strassen's Matrix Multiplication Execution Time")
plt.xlabel("Matrix Size (n x n)")
plt.ylabel("Execution Time (seconds)")
plt.grid(True)
plt.show()

```

Graph:

