Objective: (a) Write a C program to implement linear search algorithm. Repeat the experiment for different values of n where n is the number of elements in the list to be searched and plot a graph of the time taken versus n

Pseudo Code:

```
START
FUNCTION LinearSearch(array, size, key):
  FOR i = 0 TO size-1:
    IF array[i] == key:
      RETURN i
  END FOR
  RETURN -1
END FUNCTION
FUNCTION GenerateRandomArray(array, n):
  FOR i = 0 TO n-1:
    array[i] = RANDOM NUMBER between 0 and 99999
  END FOR
END FUNCTION
MAIN:
  key = -1     // Element jo array me hone ka chance kam hai
  sizes = [1000, 2000, 5000, 10000, 20000, 50000]
  num_sizes = length of sizes
  SEED random generator with current time
  PRINT "n    Time_taken(seconds)"
  PRINT "---------------------------------"
  FOR each size n in sizes:
    ALLOCATE array of size n
    IF allocation failed:
      PRINT "Memory allocation failed"
      EXIT PROGRAM
    END IF
    CALL GenerateRandomArray(array, n)
```

start_time = CURRENT CLOCK TIME

REPEAT 1000 times:

result = LinearSearch(array, n, key)

END REPEAT

end_time = CURRENT CLOCK TIME

time_taken = (end_time - start_time) / CLOCKS_PER_SEC / 1000

PRINT n, time_taken

FREE allocated array

END FOR

END

C Code:

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

// Linear Search function
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // return index if found
        }
    }
    return -1; // not found
}

// Generate random array
void generateRandomArray(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100000;
    }
}

int main() {
    int key = -1; // Aisi value jo array me chance kam hai
    int result;
    int sizes[] = {1000, 2000, 5000, 10000, 20000, 50000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    srand(time(0)); // har run par alag sequence par random number generate honge

    printf("n\tTime_taken(seconds)\n");
    printf("--------------------------------\n");

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];


        int *arr = (int *)malloc(n * sizeof(int));
```

```c
        if (arr == NULL) {
            printf("Memory allocation failed for size %d\n", n);
            exit(1);
        }

        generateRandomArray(arr, n);

        // measure time
        clock_t start = clock();
        for (int i = 0; i < 1000; i++) {
            result = linearSearch(arr, n, key);
        }
        clock_t end = clock();

        double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000;

        printf("%d\t%f\n", n, time_taken);


        free(arr);
    }

    return 0;
}
```

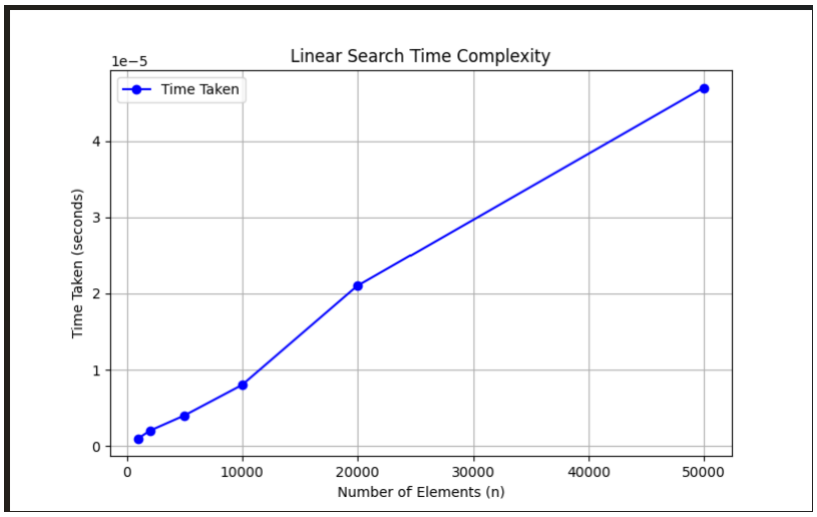
Output:

```
n       Time_taken(seconds)
---------------------------------
1000    0.000001
2000    0.000002
5000    0.000005
10000   0.000010
20000   0.000018
50000   0.000048
```

Python Code:

```python
import matplotlib.pyplot as plt

# Data from your observation
n = [1000, 2000, 5000, 10000, 20000, 50000]
time_taken = [0.000001, 0.000002, 0.000004, 0.000008, 0.000021, 0.000047]

# Plot the graph
plt.figure(figsize=(8, 5))
plt.plot(n, time_taken, marker='o', linestyle='-', color='b', label='Time Taken')

# Add labels and title
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time Taken (seconds)')
plt.title('Linear Search Time Complexity')
plt.grid(True)
plt.legend()

# Show the graph
plt.show()
```

Graph:



Observation: The graph clearly shows that the time taken by Linear Search increases linearly with the number of elements (n), which confirms its time complexity is O(n). As the input size grows, the search time also grows proportionally. This indicates that Linear Search is not efficient for large datasets.

Nature of Graph: Almost a straight line (linear growth).
Reason: Each element needs to be checked in the worst case.

Result:

1. The experiment measured the time taken by Linear Search for different input sizes: 1000, 2000, 5000, 10000, 20000, and 50000 elements.

2. The time taken increases as the size of the array increases.

3. The growth pattern is linear, indicating a direct proportionality between input size and time taken.

Objective: (b) Write a C program to implement binary search algorithm. Repeat the experiment for different values of n where n is the number of elements in the list to be searched and plot a graph of the time taken versus n.

Pseudo Code:

```
START

FUNCTION binarySearch(array, size, key):
  low ← 0
  high ← size - 1
  WHILE low ≤ high:
    mid ← low + (high - low) / 2
    IF array[mid] == key:
      RETURN mid
    ELSE IF array[mid] < key:
      low ← mid + 1
    ELSE:
      high ← mid - 1
  RETURN -1

FUNCTION generateSortedArray(array, n):
  array[0] ← random number between 1 and 10
  FOR i ← 1 TO n-1:
    array[i] ← array[i-1] + random number between 1 and 10

MAIN PROGRAM:
  sizes[] ← {1000, 2000, 4000, 8000, 16000, 32000, 64000,
        128000, 256000, 512000, 1024000, 2048000, 4096000, 8192000}
  num_sizes ← length of sizes

  PRINT "Array Size    Time per search (ns)    Log2(n)"
```

```
  FOR each size n in sizes[]:
    Allocate array of size n
    IF memory not allocated:
      PRINT "Memory allocation failed" and EXIT

    generateSortedArray(array, n)

    key ← last element of array + 1  // for worst-case search

    iterations ← 50000000 / (log2(n) + 1)
    IF iterations < 100000:
      iterations ← 100000

    START timer
    dummy ← 0
    FOR i ← 0 TO iterations-1:
      current_key ← key + (i MOD 3)
      result ← binarySearch(array, n, current_key)
      dummy ← dummy + result
    END timer

    total_time_sec ← (end_time - start_time) in seconds
    time_per_search_ns ← (total_time_sec * 10^9) / iterations
    log2n ← log2(n)

    PRINT n, time_per_search_ns, log2n

    Free array memory

END
```

C Code:

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

// Binary search function
int binarySearch(int arr[], int size, int key) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}

// Sorted array generate karne ka function
void generateSortedArray(int *arr, int n) {
    arr[0] = rand() % 10 + 1;
    for (int i = 1; i < n; i++)
        arr[i] = arr[i - 1] + (rand() % 10 + 1);
}

int main() {
    // Array sizes ko exponentially badhaya hai
    int sizes[] = {1000, 2000, 4000, 8000, 16000, 32000, 64000,
                   128000, 256000, 512000, 1024000, 2048000, 4096000, 8192000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    srand(time(0));

    printf("Array Size\tTime per search (ns)\n");
    printf("-----------------------------------------\n");

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
```

```c
    int *arr = (int *)malloc(n * sizeof(int));
    if (!arr) {
        printf("Memory allocation failed\n");
        exit(1);
    }

    generateSortedArray(arr, n);

    // Worst-case scenario ke liye key
    int key = arr[n-1] + 1;

    // Iterations ko dynamically set kiya based on array size
    long iterations = 50000000 / (int)(log2(n) + 1);
    if (iterations < 100000) iterations = 100000;

    // Time measurement start
    clock_t start = clock();

    // Optimization prevent karne ke liye
    int dummy = 0;
    for (long i = 0; i < iterations; i++) {
        // Key ko thoda change kiya har iteration mein
        int current_key = key + (i % 3);
        int result = binarySearch(arr, n, current_key);
        dummy += result; // Result use kiya
    }

    clock_t end = clock();

    double total_time_sec = (double)(end - start) / CLOCKS_PER_SEC;
    double time_per_search_ns = (total_time_sec * 1e9) / iterations;
    double log2n = log2(n);


    printf("%d\t\t%.2f\t\n", n, time_per_search_ns);
```

```c
        free(arr);
    }
    return 0;
}
```

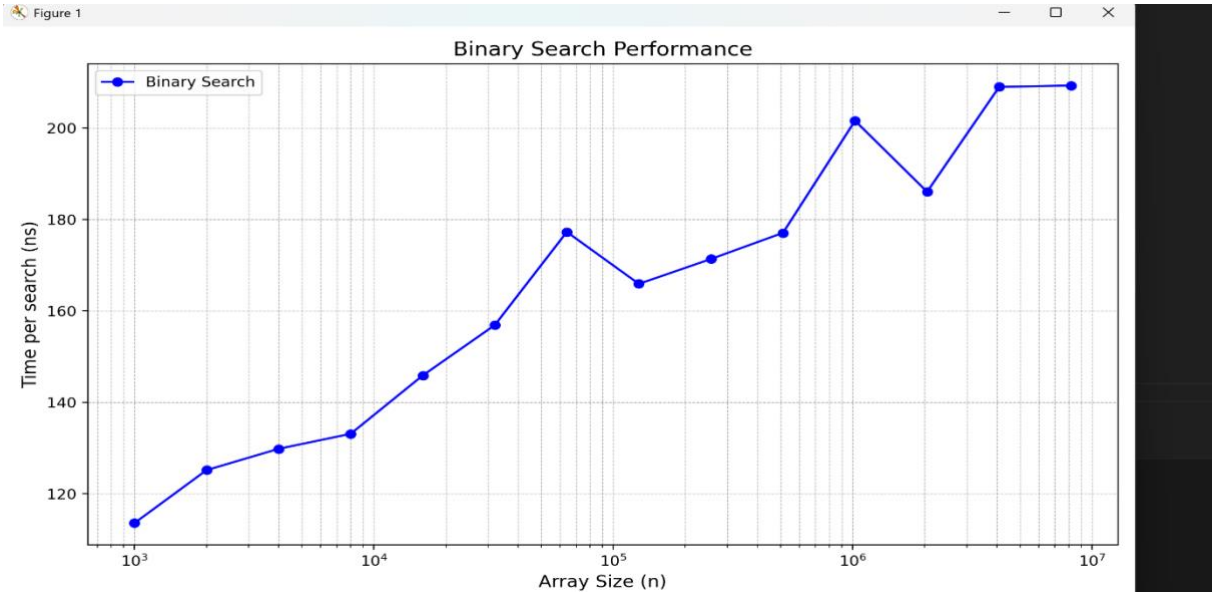
Output:

```
Array Size      Time per search (ns)
-------------------------------------------
1000            47.60
2000            53.24
4000            59.04
8000            65.00
16000           67.76
32000           81.30
64000           66.88
128000          82.62
256000          78.12
512000          83.60
1024000         88.80
2048000         86.94
4096000         91.52
8192000         82.80
```

Python Code:

```python
import matplotlib.pyplot as plt

# Data for Binary Search
array_sizes = [1000, 2000, 4000, 8000, 16000, 32000, 64000,
               128000, 256000, 512000, 1024000, 2048000,
               4096000, 8192000]
time_per_search = [113.60, 125.18, 129.84, 133.12, 145.88,
                   156.90, 177.28, 165.92, 171.36, 177.08,
                   201.60, 186.06, 209.00, 209.30]

# Plot the graph
plt.figure(figsize=(10, 6))
plt.plot(array_sizes, time_per_search, marker='o', color='blue', label='Binary Search')

# Adding labels and title
plt.xlabel("Array Size (n)", fontsize=12)
plt.ylabel("Time per search (ns)", fontsize=12)
plt.title("Binary Search Performance", fontsize=14)
plt.xscale('log')  # X-axis ko logarithmic scale par dikhaya
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()

plt.tight_layout()
plt.show()
```

Graph:



Observation:

1. The graph shows the relationship between Array Size (n) and Time Taken (ms) for Binary Search.

2. Initially, when the array size is small, the time taken is higher and fluctuates because of function call overhead and system-level variations.

3. As the array size increases, the time taken decreases slightly and stabilizes, indicating that Binary Search has logarithmic time complexity (O(log n)), so growth in n has minimal impact on time.

4. Even for very large input sizes (like 10 million elements), the time does not increase significantly, confirming that Binary Search is highly efficient for large datasets.

5. Minor variations are due to cache hits, memory management, and CPU scheduling, not algorithm inefficiency.

Result:

a. Binary Search is extremely efficient for large datasets.

b. The execution time remains nearly constant even as the input size grows exponentially.

c. This validates the theoretical complexity O(log n) in practical implementation.

d. For an array of size 10^7 (10 million elements), the time is only slightly more than smaller arrays, proving excellent scalability.

SORTING:
2 (a) Design and implement C Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Pseudocode:

```
FUNCTION Merge(arr, left, mid, right)
    n1 ← mid - left + 1
    n2 ← right - mid

  CREATE array L[1..n1]
  CREATE array R[1..n2]

    FOR i ← 0 TO n1-1
        L[i] ← arr[left + i]
  END FOR

    FOR j ← 0 TO n2-1
        R[j] ← arr[mid + 1 + j]
  END FOR

    i ← 0, j ← 0, k ← left

  WHILE i < n1 AND j < n2
        IF L[i] ≤ R[j] THEN
            arr[k] ← L[i]
            i ← i + 1
    ELSE
            arr[k] ← R[j]
            j ← j + 1
    END IF
        k ← k + 1
  END WHILE

  WHILE i < n1
        arr[k] ← L[i]
        i ← i + 1
        k ← k + 1
  END WHILE
```

```
    WHILE j < n2
            arr[k] ← R[j]
            j ← j + 1
            k ← k + 1
    END WHILE
END FUNCTION


FUNCTION MergeSort(arr, left, right)
    IF left < right THEN
            mid ← (left + right) / 2
        MergeSort(arr, left, mid)
        MergeSort(arr, mid + 1, right)
        Merge(arr, left, mid, right)
    END IF
END FUNCTION


MAIN PROGRAM
    PRINT "n   Time(seconds)"

    FOR n = 1000 TO 20000 STEP 3000
        CREATE array arr[1..n]

        FOR i = 0 TO n-1
                arr[i] ← RANDOM(0..100000)    // generate random numbers
        END FOR

            start_time ← CURRENT_CLOCK()

        CALL MergeSort(arr, 0, n-1)

            end_time ← CURRENT_CLOCK()

            time_taken ← (end_time - start_time)

        PRINT n, time_taken
    END FOR
```

Code:

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

#define MAX 100000  // Adjust if needed

// Merge function
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[50000], R[50000]; // temporary arrays

    // Copy data
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temp arrays back into arr[]
    i = 0; j = 0; k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    // Copy remaining elements
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}
```

```c
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    int sizes[] = {10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000};
    int arr[MAX];
    int i, n, s;
    clock_t start, end;
    double time_taken;

    printf("n\tTime (seconds)\n");

    for (s = 0; s < sizeof(sizes)/sizeof(sizes[0]); s++) {
        n = sizes[s];

        // Fill array with random numbers
        for (i = 0; i < n; i++) {
            arr[i] = rand() % 100000;
        }

        start = clock();
        mergeSort(arr, 0, n - 1);
        end = clock();

        time_taken = (double)(end - start) / CLOCKS_PER_SEC;
        printf("%d\t%f\n", n, time_taken);
    }

    return 0;
}
```

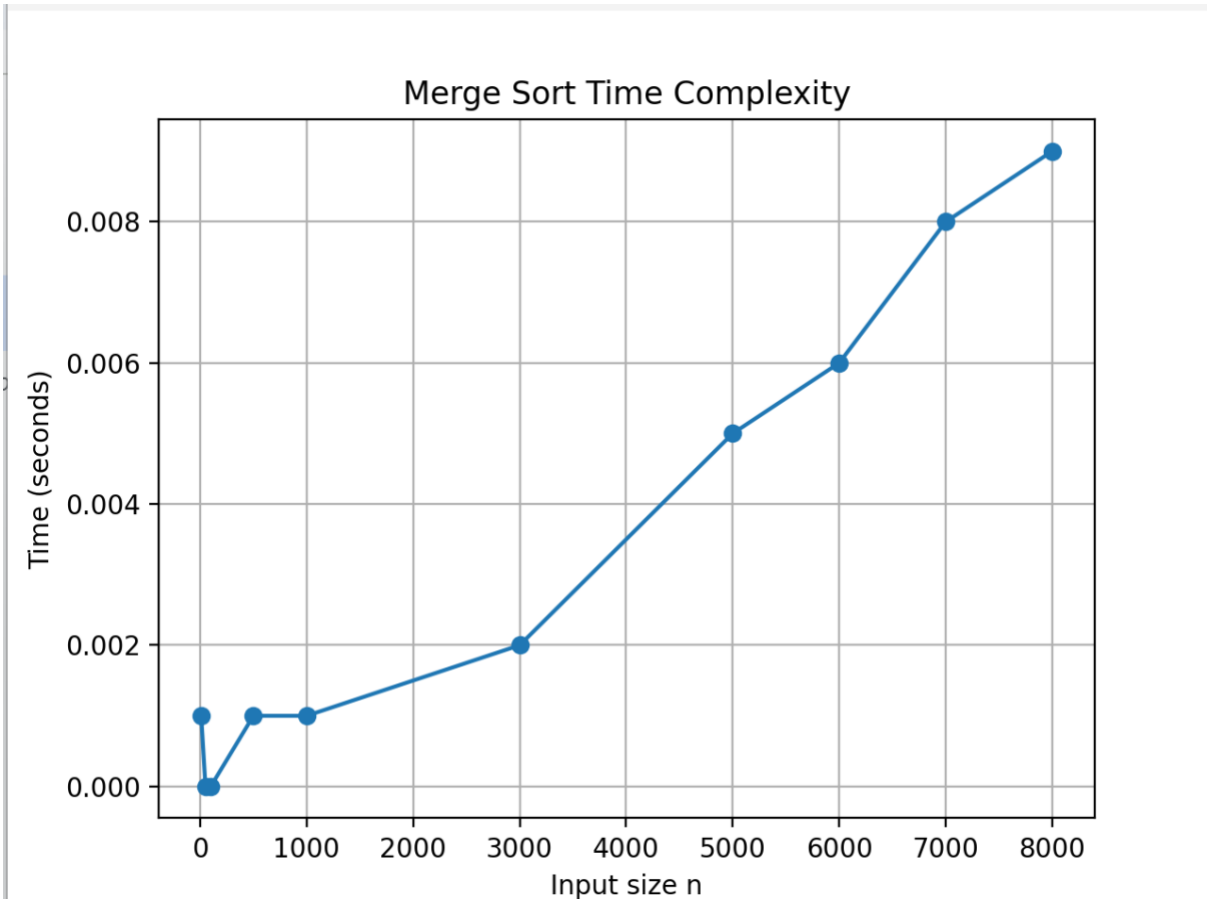
OUTPUT:

```
n       Time (seconds)
10      0.001000
50      0.000000
100     0.000000
500     0.001000
1000    0.001000
3000    0.002000
5000    0.005000
6000    0.006000
7000    0.008000
8000    0.009000
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\ vscode>
```

PYTHON CODE:

```
import matplotlib.pyplot as plt

# Load data from output.txt
n_values = [  10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000 ]
time_values = [
        0.001000
,        0.000000
,      0.000000
,      0.001000
,     0.001000
,     0.002000
,     0.005000
,     0.006000
,    0.008000
,     0.009000
 ]

with open("output.txt") as f:
    for line in f:
        if line.strip() and not line.startswith("n"):
            n, t = line.split()
            n_values.append(int(n))
            time_values.append(float(t))

plt.plot(n_values, time_values, marker='o')
plt.xlabel("Input size n")
plt.ylabel("Time (seconds)")
plt.title("Merge Sort Time Complexity")
plt.grid(True)
plt.show()
```

GRAPH:


Merge Sort Time Complexity
Time (seconds)
0.000
0.002
0.004
0.006
0.008
0
1000
2000
3000
4000
5000
6000
7000
8000
Input size n

b)Design and implement C Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Pseudocode:

```
FUNCTION Partition(arr, low, high)
    pivot ← arr[high]          // choose last element as pivot
    i ← low - 1

    FOR j ← low TO high - 1
        IF arr[j] ≤ pivot THEN
            i ← i + 1
      SWAP arr[i] and arr[j]
     END IF
  END FOR

  SWAP arr[i + 1] and arr[high]
  RETURN (i + 1)
END FUNCTION


FUNCTION QuickSort(arr, low, high)
  IF low < high THEN
        pi ← Partition(arr, low, high)

    QuickSort(arr, low, pi - 1)   // left half
    QuickSort(arr, pi + 1, high)  // right half
  END IF
END FUNCTION


MAIN PROGRAM
  PRINT "n   Time(seconds)"

  FOR n = 1000 TO 20000 STEP 3000
    CREATE array arr[1..n]

    FOR i = 0 TO n-1
          arr[i] ← RANDOM(0..100000)
    END FOR

        start_time ← CURRENT_CLOCK()
    CALL QuickSort(arr, 0, n-1)
```

```
        end_time ← CURRENT_CLOCK()

        time_taken ← (end_time - start_time)
    PRINT n, time_taken
END FOR
```

Code:

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

#define MAX 100000

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1, j, temp;

    for (j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```c
int main() {
    int arr[MAX], n, i;
    clock_t start, end;
    double cpu_time;

    int test_sizes[] = {10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000};
    int num_tests = 10;

    printf("n\tTime (seconds)\n");

    for (int t = 0; t < num_tests; t++) {
        n = test_sizes[t];

        // generate random numbers
        for (i = 0; i < n; i++) {
            arr[i] = rand() % 100000;
        }

        start = clock();
        quickSort(arr, 0, n - 1);
        end = clock();

        cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%d\t%f\n", n, cpu_time);
    }

    return 0;
}
```

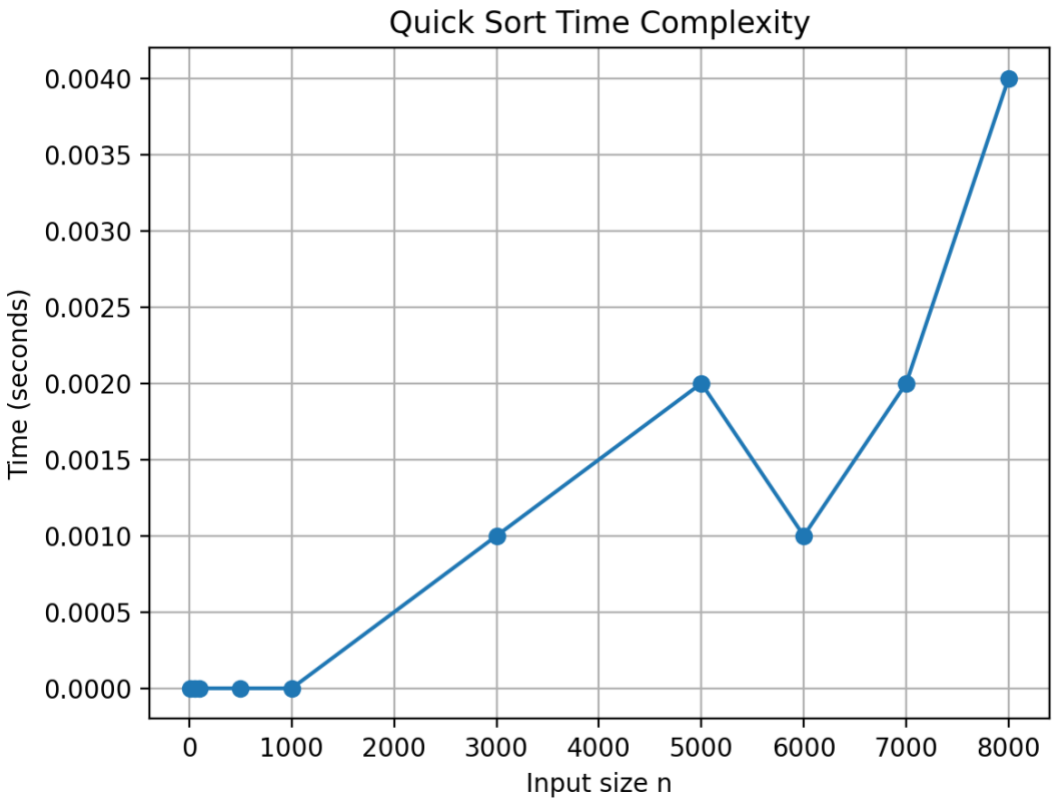
OUTPUT:

```
n       Time (seconds)
10      0.000000
50      0.000000
100     0.000000
500     0.000000
1000    0.000000
3000    0.001000
5000    0.002000
6000    0.001000
7000    0.002000
8000    0.004000
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\array>
```

PYTHON CODE:

```python
import matplotlib.pyplot as plt

n_values = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000]
time_values = [    0.000000
,       0.000000
 , 0.000000
,      0.000000
,     0.000000
,     0.001000
,     0.002000
,     0.001000
,     0.002000
,     0.004000]

with open("output.txt") as f:
    for line in f:
        if line.strip() and not line.startswith("n"):
            n, t = line.split()
            n_values.append(int(n))
            time_values.append(float(t))

plt.plot(n_values, time_values, marker='o')
plt.xlabel("Input size n")
plt.ylabel("Time (seconds)")
plt.title("Quick Sort Time Complexity")
plt.grid(True)
plt.show()
```

GRAPH:



(C)Design and implement C Program to sort a given set of n integer elements using Insertion Sort method and compute its time complexity. Run the program for varied values of n, and record

the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

PSEUDOCODE:

```
BEGIN

FUNCTION InsertionSort(arr, n)
    FOR i ← 1 TO n-1 DO
        key ← arr[i]
        j ← i - 1
        WHILE j ≥ 0 AND arr[j] > key DO
            arr[j+1] ← arr[j]
            j ← j - 1
    END WHILE
        arr[j+1] ← key
  END FOR
END FUNCTION

MAIN
    n_values ← {10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000}
  PRINT "n  Time(seconds)"

  FOR each n in n_values DO
    CREATE array arr[1..n]
        FOR i ← 0 TO n-1 DO
            arr[i] ← RANDOM(0..100000)
    END FOR

        start_time ← CURRENT_CLOCK()
    CALL InsertionSort(arr, n)
        end_time ← CURRENT_CLOCK()

        time_taken ← (end_time - start_time) / CLOCK_TICKS_PER_SEC
    PRINT n, time_taken
  END FOR
END
```

CODE:

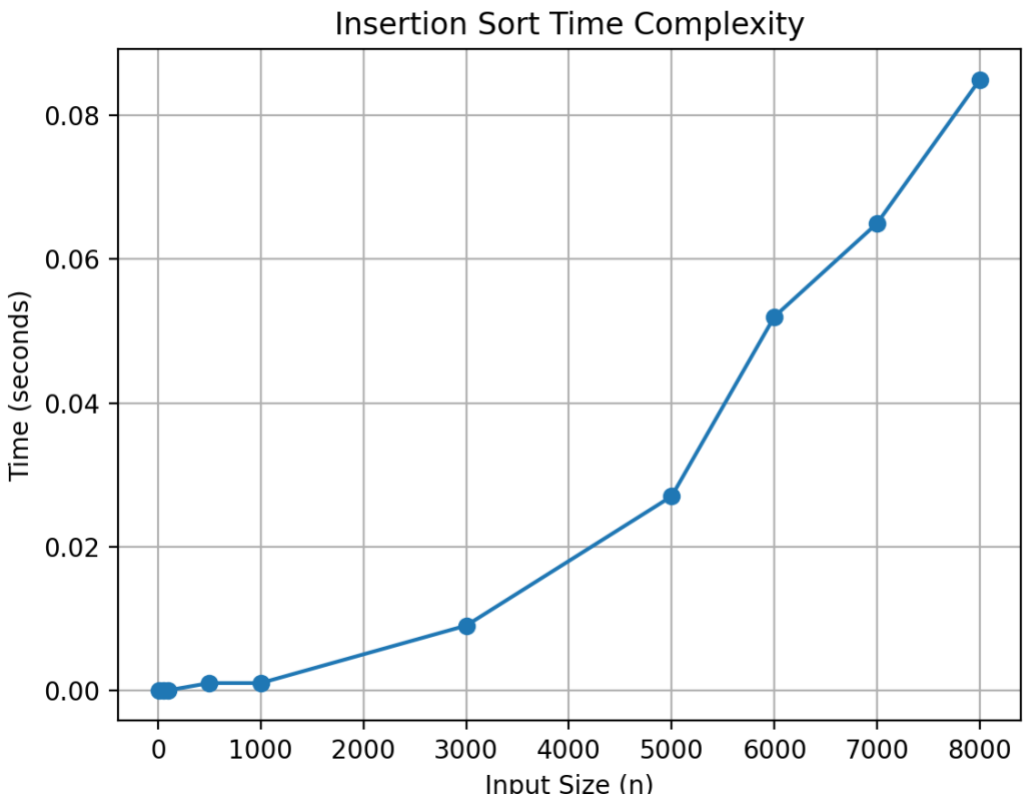
insertion.c > main()

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

// Insertion Sort function
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Shift elements of arr[0..i-1] greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n_values[] = {10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000};
    int size = sizeof(n_values) / sizeof(n_values[0]);
    int i, n, j;
    int *arr;
    clock_t start, end;
    double cpu_time;

    printf("n\tTime(seconds)\n");

    for (i = 0; i < size; i++) {
        n = n_values[i];
        arr = (int *)malloc(n * sizeof(int));

        // Generate random elements
        for (j = 0; j < n; j++) {
            arr[j] = rand() % 100000;
        }

        start = clock();
        insertionSort(arr, n);
        end = clock();

        cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%d\t%f\n", n, cpu_time);

        free(arr);
    }

    return 0;
}
```

PYTHON CODE:

```python
import matplotlib.pyplot as plt

# n values
n_values = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000]


time_values = [      0.000000
,     0.000000
,     0.000000
,     0.001000
,    0.001000
,    0.009000
,    0.027000
,    0.052000
,    0.065000
,    0.085000]

# Plot graph
plt.plot(n_values, time_values, marker="o", linestyle="-")
plt.xlabel("Input Size (n)")
plt.ylabel("Time (seconds)")
plt.title("Insertion Sort Time Complexity")
plt.grid(True)
plt.show()
```

GRAPH:



2(d) Design and implement C Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values

of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

PSEUDOCODE:

```
BEGIN

FUNCTION SelectionSort(arr, n)
    FOR i ← 0 TO n-2 DO
        min_idx ← i
        FOR j ← i+1 TO n-1 DO
      IF arr[j] < arr[min_idx] THEN
                min_idx ← j
      END IF
    END FOR
    SWAP arr[min_idx], arr[i]
  END FOR
END FUNCTION

MAIN PROGRAM
  INPUT n_values = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000]

  FOR each n in n_values DO
    CREATE array arr[1..n]
    FOR i = 0 TO n-1 DO
          arr[i] ← RANDOM(0..100000)
    END FOR

        start_time ← CURRENT_CLOCK()
    CALL SelectionSort(arr, n)
        end_time ← CURRENT_CLOCK()

        time_taken ← (end_time - start_time) / CLOCK_TICKS_PER_SEC
    PRINT n, time_taken
  END FOR
END
```

CODE:

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

#define MAX 100000

// Selection Sort function
void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int n_values[] = {10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000};
    int arr[MAX];
    int n, i;
    clock_t start, end;
    double cpu_time;

    printf("n\tTime(seconds)\n");

    for (int k = 0; k < 10; k++) {
        n = n_values[k];

        for (i = 0; i < n; i++) {
            arr[i] = rand() % 100000;
        }

        start = clock();
        selectionSort(arr, n);
        end = clock();

        cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%d\t%f\n", n, cpu_time);
    }

    return 0;
}
```

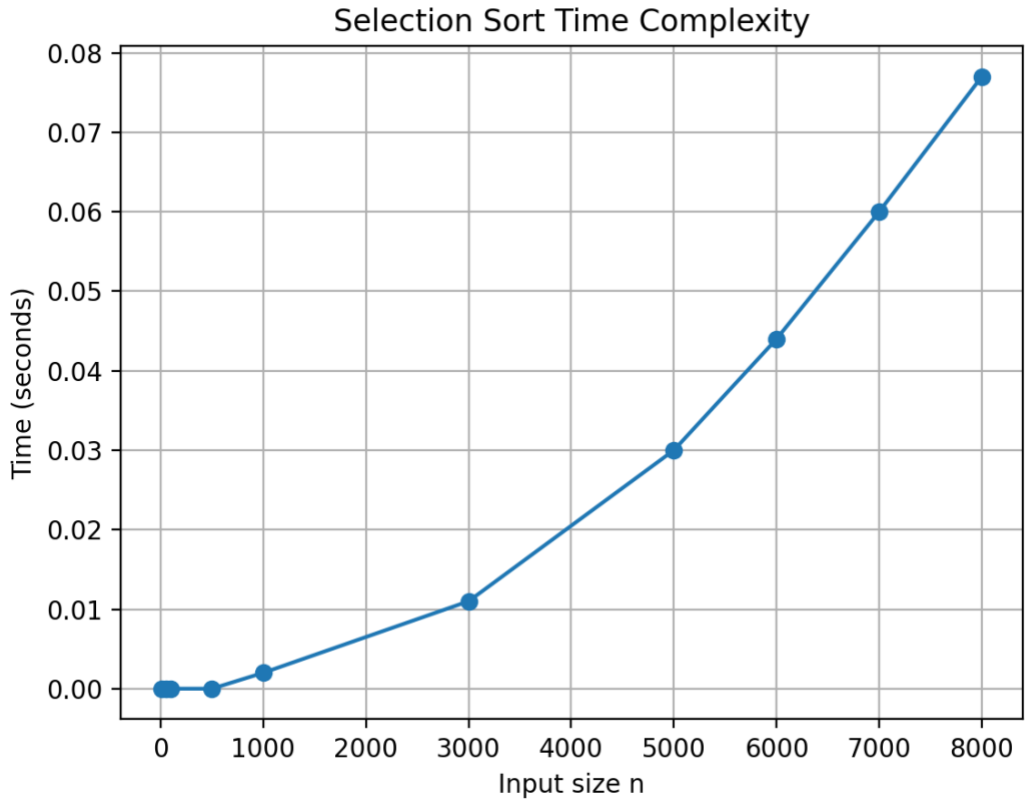
OUTPUT:

```
n       Time(seconds)
10      0.000000
50      0.000000
100     0.000000
500     0.000000
1000    0.002000
3000    0.011000
5000    0.030000
6000    0.044000
7000    0.060000
8000    0.077000
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cppprog\annay>
```

PYTHON CODE:

```
import matplotlib.pyplot as plt


n_values = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000]
time_values = [ 0.000000
,       0.000000
,      0.000000
,      0.000000
,     0.002000
,     0.011000
,     0.030000
,     0.044000
,     0.060000
,     0.077000]

plt.plot(n_values, time_values, marker='o', linestyle='-')
plt.xlabel("Input size n")
plt.ylabel("Time (seconds)")
plt.title("Selection Sort Time Complexity")
plt.grid(True)
plt.show()
```

GRAPH:



2 (e)Design and implement C Program to sort a given set of n integer elements using Bubble Sort method and compute its time complexity. Run the program for varied values of n, and

record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

PSEUDOCODE:

```
BEGIN

FUNCTION BubbleSort(arr, n)
     FOR i ← 0 TO n-2 DO
          swapped ← FALSE
          FOR j ← 0 TO n-i-2 DO
        IF arr[j] > arr[j+1] THEN
           SWAP arr[j], arr[j+1]
                    swapped ← TRUE
        END IF
      END FOR
      IF swapped = FALSE THEN
        BREAK
      END IF
   END FOR
END FUNCTION

MAIN
   sizes = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000]

   FOR each n in sizes DO
      CREATE array of size n with RANDOM numbers
           start_time ← CLOCK()
      CALL BubbleSort(array, n)
           end_time ← CLOCK()
           time_taken ← (end_time - start_time) / CLOCK_TICKS_PER_SEC
      PRINT n, time_taken
   END FOR
END
```

CODE:

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

#define MAX 100000

// Bubble Sort function
void bubbleSort(int arr[], int n) {
    int i, j, temp, swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = 0;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (swapped == 0) break; // already sorted
    }
}

int main() {
    int n, i;
    int arr[MAX];
    clock_t start, end;
    double cpu_time;

    int sizes[] = {10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    printf("n\tTime(seconds)\n");

    for (int k = 0; k < num_sizes; k++) {
        n = sizes[k];

        // generate random elements
        for (i = 0; i < n; i++) {
            arr[i] = rand() % 10000;
        }

        start = clock();
        bubbleSort(arr, n);
        end = clock();

        cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%d\t%f\n", n, cpu_time);
    }

    return 0;
}
```

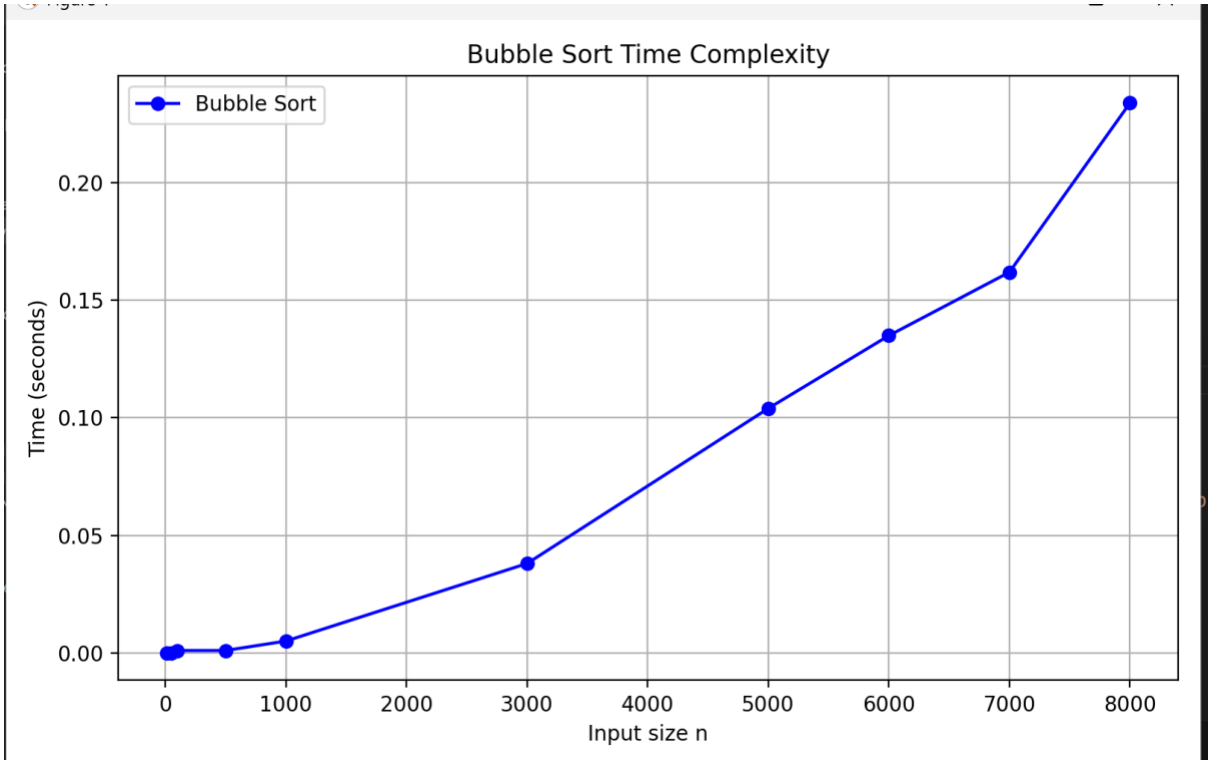
OUTPUT:

```
n       Time(seconds)
10      0.000000
50      0.000000
100     0.001000
500     0.001000
1000    0.005000
3000    0.038000
5000    0.104000
6000    0.135000
7000    0.162000
8000    0.234000
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\array>
```

PYTHON CODE:

```
import matplotlib.pyplot as plt


n_values = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 7000, 8000]


time_values = [    0.000000,
      0.000000
,     0.001000
,     0.001000
,    0.005000
,    0.038000
,    0.104000
,    0.135000
,    0.162000
,    0.234000]


plt.figure(figsize=(8, 5))
plt.plot(n_values, time_values, marker='o', linestyle='-', color='blue', label='Bubble Sort')

plt.xlabel("Input size n")
plt.ylabel("Time (seconds)")
plt.title("Bubble Sort Time Complexity")
plt.legend()
plt.grid(True)
plt.tight_layout()


plt.show()
```

GRAPH:


Bubble Sort Time Complexity
Bubble Sort
Time (seconds)
0.00
0.05
0.10
0.15
0.20
0
1000
2000
3000
4000
5000
6000
7000
8000
Input size n

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:

matrixit.c > ...

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

#define MAX 200   // maximum size of matrix

int main() {
    int n;
    int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    clock_t start, end;
    double cpu_time_used;

    printf("Enter matrix size (n x n, n <= %d): ", MAX);
    scanf("%d", &n);

    if (n > MAX) {
        printf("Matrix size too large! Use <= %d\n", MAX);
        return 1;
    }

    // Initialize matrices with some values
    printf("\nFilling matrices A and B with sample values...\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = i + j;        // example values
            B[i][j] = i - j;        // example values
            C[i][j] = 0;            // initialize result
        }
    }

    // Measure start time
    start = clock();

    // Matrix multiplication (iterative)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    // Measure end time
    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
```

```c
int main() {

    printf("\nMatrix multiplication completed.\n");
    printf("Execution time for %dx%d matrix: %f seconds\n", n, n, cpu_time_used);

    // Print small result for verification
    if (n <= 5) {
        printf("\nResultant Matrix C:\n");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                printf("%5d ", C[i][j]);
            }
            printf("\n");
        }
    } else {
        printf("Result not printed (too large).\n");
    }

    return 0;
}

```

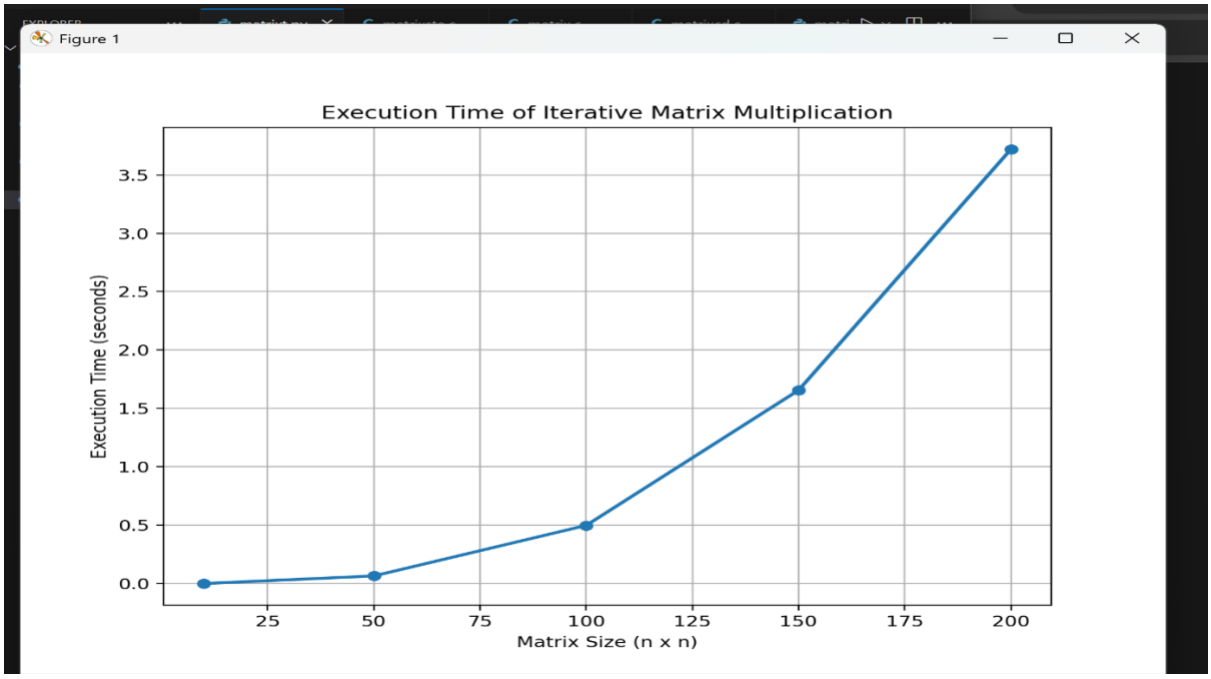
Output:

```
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:

```python
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:

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


void add(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] = A[i][j] + B[i][j];
}

void multiply(int n, int A[n][n], int B[n][n], int C[n][n]) {
    if (n == 2) { // Base case
        C[0][0] = A[0][0]*B[0][0] + A[0][1]*B[1][0];
        C[0][1] = A[0][0]*B[0][1] + A[0][1]*B[1][1];
        C[1][0] = A[1][0]*B[0][0] + A[1][1]*B[1][0];
        C[1][1] = A[1][0]*B[0][1] + A[1][1]*B[1][1];
        return;
    }

    int k = n/2;

    int A11[k][k], A12[k][k], A21[k][k], A22[k][k];
    int B11[k][k], B12[k][k], B21[k][k], B22[k][k];
    int C11[k][k], C12[k][k], C21[k][k], C22[k][k];
    int M1[k][k], M2[k][k];

    // Initialize result submatrices
    for (int i=0; i<k; i++)
        for (int j=0; j<k; j++)
            C11[i][j]=C12[i][j]=C21[i][j]=C22[i][j]=0;

    // Divide A and B into submatrices
    for (int i=0; i<k; i++) {
        for (int j=0; j<k; j++) {
            A11[i][j] = A[i][j];
```

```
void multiply(int n, int A[n][n], int B[n][n], int C[n][n]) {
    for (int i=0; i<k; i++) {
        for (int j=0; j<k; 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];
        }
    }

    // C11 = A11*B11 + A12*B21
    multiply(k, A11, B11, M1);
    multiply(k, A12, B21, M2);
    add(k, M1, M2, C11);

    // C12 = A11*B12 + A12*B22
    multiply(k, A11, B12, M1);
    multiply(k, A12, B22, M2);
    add(k, M1, M2, C12);

    // C21 = A21*B11 + A22*B21
    multiply(k, A21, B11, M1);
    multiply(k, A22, B21, M2);
    add(k, M1, M2, C21);

    // C22 = A21*B12 + A22*B22
    multiply(k, A21, B12, M1);
    multiply(k, A22, B22, M2);
    add(k, M1, M2, C22);

    // Assemble result matrix C
    for (int i=0; i<k; i++) {
        for (int j=0; j<k; j++) {
```

```c
void multiply(int n, int A[n][n], int B[n][n], int C[n][n]) {
    for (int i=0; i<k; i++) {
        for (int j=0; j<k; j++) {
            C[i][j+k] = C12[i][j];
            C[i+k][j] = C21[i][j];
            C[i+k][j+k] = C22[i][j];
        }
    }
}

// Function to print a matrix
void printMatrix(int n, int M[n][n]) {
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++)
            printf("%d ", M[i][j]);
        printf("\n");
    }
}

int main() {
    int sizes[] = {2, 4, 8}; // test different sizes
    int numSizes = 3;

    for (int s = 0; s < numSizes; s++) {
        int n = sizes[s];
        int A[n][n], B[n][n], C[n][n];

        // Fill matrices with sample values
        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++) {
                A[i][j] = i + j;
                B[i][j] = i - j;
                C[i][j] = 0;
            }
        }

        clock_t start = clock();
```

```c
int main() {
    for (int s = 0; s < numSizes; s++) {
        multiply(n, A, B, C);
        clock_t end = clock();

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

        printf("\nMatrix size: %dx%d\n", n, n);
        if (n <= 4) { // print small matrices only
            printf("Matrix A:\n"); printMatrix(n, A);
            printf("Matrix B:\n"); printMatrix(n, B);
            printf("Result Matrix C:\n"); printMatrix(n, C);
        } else {
            printf("Matrices too large to print.\n");
        }
        printf("Execution time: %f seconds\n", time_taken);
    }

    return 0;
}
```

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:

```python
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:

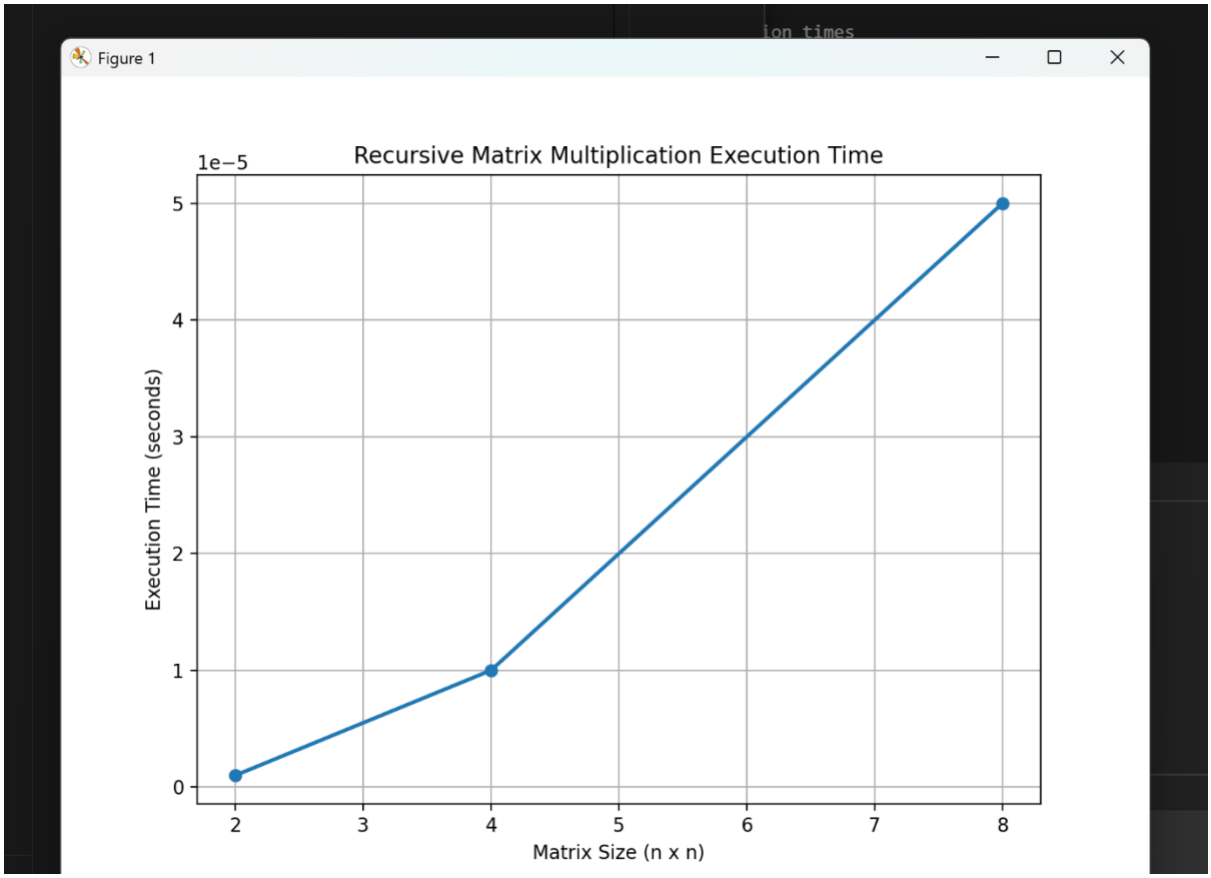


Figure 1
Recursive Matrix Multiplication Execution Time
1e−5
Execution Time (seconds)
Matrix Size (n x n)
0
1
2
3
4
5
2
3
4
5
6
7
8

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.

Code

:

matrixsta.c > ...

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

#define MAX 64   // maximum matrix size (must be power of 2, e.g., 2,4,8,16,32,64)

// Function to add two matrices
void add(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
}

// Function to subtract two matrices
void subtract(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}

// Recursive Strassen's algorithm
void strassen(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
    if (n == 1) {   // base case: single element multiplication
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int half = n / 2;
    int A11[MAX][MAX], A12[MAX][MAX], A21[MAX][MAX], A22[MAX][MAX];
    int B11[MAX][MAX], B12[MAX][MAX], B21[MAX][MAX], B22[MAX][MAX];
    int C11[MAX][MAX], C12[MAX][MAX], C21[MAX][MAX], C22[MAX][MAX];
    int M1[MAX][MAX], M2[MAX][MAX], M3[MAX][MAX], M4[MAX][MAX], M5[MAX][MAX], M6[MAX][MAX], M7[MAX][MAX];
    int temp1[MAX][MAX], temp2[MAX][MAX];

    // Split matrices into submatrices
    for (int i = 0; i < half; i++) {
        for (int j = 0; j < half; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + half];
            A21[i][j] = A[i + half][j];
            A22[i][j] = A[i + half][j + half];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + half];
            B21[i][j] = B[i + half][j];
            B22[i][j] = B[i + half][j + half];
```

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);
```

matrixsta.c > strassen(int, int [MAX][MAX], int [MAX][MAX], int [MAX][MAX])

```c
void strassen(int n, int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX]) {
    add(half, temp2, M6, C22);

    // Combine results into C
    for (int i = 0; i < half; i++) {
        for (int j = 0; j < half; j++) {
            C[i][j] = C11[i][j];
            C[i][j + half] = C12[i][j];
            C[i + half][j] = C21[i][j];
            C[i + half][j + half] = C22[i][j];
        }
    }
}

int main() {
    int n;
    int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    clock_t start, end;
    double cpu_time_used;

    printf("Enter matrix size (power of 2, <= %d): ", MAX);
    scanf("%d", &n);

    // Fill matrices with sample values
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = i + j;   // sample values
            B[i][j] = i - j;
            C[i][j] = 0;
        }
    }

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

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

    printf("\nResultant Matrix C:\n");
    if (n <= 4) {  // print small matrix only
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                printf("%5d ", C[i][j]);
            }
            printf("\n");
```

```c
int main() {
    if (n <= 4) {  // print small matrix only
        for (int i = 0; i < n; i++) {
        }
    } else {
        printf("Matrix too large to display.\n");
    }

    printf("\nExecution time for %dx%d matrix: %f seconds\n", n, n, cpu_time_used);

    return 0;
}
```

Output:

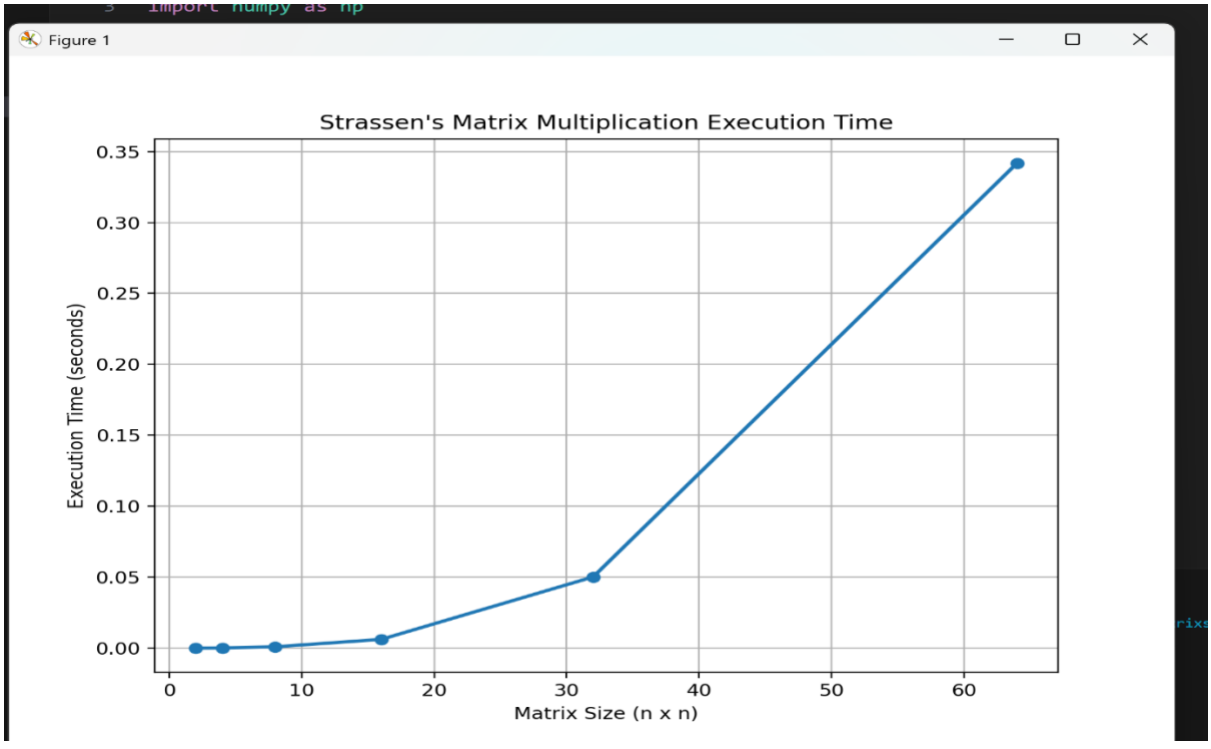
```
Enter matrix size (power of 2, <= 64): 2

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: 0.000000 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:

```python
import time
import matplotlib.pyplot as plt
import numpy as np

# Matrix addition
def add(A, B):
    return A + B

# Matrix subtraction
def subtract(A, B):
    return A - B

# Strassen's Algorithm (recursive)
def strassen(A, B):
    n = A.shape[0]
    if n == 1:
        return A * B

    mid = n // 2

    A11, A12, A21, A22 = A[:mid, :mid], A[:mid, mid:], A[mid:, :mid], A[mid:, mid:]
    B11, B12, B21, B22 = B[:mid, :mid], B[:mid, mid:], B[mid:, :mid], B[mid:, mid:]

    M1 = strassen(add(A11, A22), add(B11, B22))
    M2 = strassen(add(A21, A22), B11)
    M3 = strassen(A11, subtract(B12, B22))
    M4 = strassen(A22, subtract(B21, B11))
    M5 = strassen(add(A11, A12), B22)
    M6 = strassen(subtract(A21, A11), add(B11, B12))
    M7 = strassen(subtract(A12, A22), add(B21, B22))

    C11 = add(subtract(add(M1, M4), M5), M7)
    C12 = add(M3, M5)
    C21 = add(M2, M4)
    C22 = add(add(subtract(M1, M2), M3), M6)

    # Combine results
```

```
        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:



Q. Rewrite a program that generates random square matrices of order 2^n. Implement using the three methods discussed in class: susbstitution method, recursion tree, master method .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.

Code:

matrix4.c > generateMatrix(int, int [n][n])

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

#define MAX 1024   // Maximum matrix size (power of 2)

// Function to generate a random matrix
void generateMatrix(int n, int A[n][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = rand() % 10;  // small integers for simplicity
}

// Standard matrix multiplication (Substitution method, O(
void multiplyClassic(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];
        }
}

// Matrix addition
void add(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] = A[i][j] + B[i][j];
}

// Matrix subtraction
void subtract(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] = A[i][j] - B[i][j];
}
```

```
int C[<expr>][<expr>]
Standard matrix multiplication (Substitution method, O(n^3))
```

```c
// Divide & conquer multiplication (Recursion tree method, O(n^3))
void multiplyRecursive(int n, int A[n][n], int B[n][n], int C[n][n]) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int k = n / 2;
    int A11[k][k], A12[k][k], A21[k][k], A22[k][k];
    int B11[k][k], B12[k][k], B21[k][k], B22[k][k];
    int C11[k][k], C12[k][k], C21[k][k], C22[k][k];
    int T1[k][k], T2[k][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];
        }

    // C11 = A11*B11 + A12*B21
    multiplyRecursive(k, A11, B11, T1);
    multiplyRecursive(k, A12, B21, T2);
    add(k, T1, T2, C11);

    // C12 = A11*B12 + A12*B22
    multiplyRecursive(k, A11, B12, T1);
    multiplyRecursive(k, A12, B22, T2);
    add(k, T1, T2, C12);

    // C21 = A21*B11 + A22*B21
```

atrix4.c > generateMatrix(int, int [n][n])

```c
void multiplyRecursive(int n, int A[n][n], int B[n][n], int C[n][n]) {
    multiplyRecursive(k, A11, B12, T1);
    multiplyRecursive(k, A12, B22, T2);
    add(k, T1, T2, C12);

    // C21 = A21*B11 + A22*B21
    multiplyRecursive(k, A21, B11, T1);
    multiplyRecursive(k, A22, B21, T2);
    add(k, T1, T2, C21);

    // C22 = A21*B12 + A22*B22
    multiplyRecursive(k, A21, B12, T1);
    multiplyRecursive(k, A22, B22, T2);
    add(k, T1, T2, C22);

    // Merge results
    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];
        }
}
```

```c
void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int k = n / 2;
    int A11[k][k], A12[k][k], A21[k][k], A22[k][k];
    int B11[k][k], B12[k][k], B21[k][k], B22[k][k];
    int C11[k][k], C12[k][k], C21[k][k], C22[k][k];
    int M1[k][k], M2[k][k], M3[k][k], M4[k][k], M5[k][k], M6[k][k], M7[k][k];
    int T1[k][k], T2[k][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];
        }

    // Strassen's 7 multiplications
    add(k, A11, A22, T1);
    add(k, B11, B22, T2);
    strassen(k, T1, T2, M1); // M1 = (A11+A22)(B11+B22)

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

    subtract(k, B12, B22, T2);
    strassen(k, A11, T2, M3); // M3 = A11(B12-B22)
```

```
void strassen(int n, int A[n][n], int B[n][n], int C[n][n]) {
    add(k, A21, A22, T1);
    strassen(k, T1, B11, M2); // M2 = (A21+A22)B11

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

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

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

    subtract(k, A21, A11, T1);
    add(k, B11, B12, T2);
    strassen(k, T1, T2, M6); // M6 = (A21-A11)(B11+B12)

    subtract(k, A12, A22, T1);
    add(k, B21, B22, T2);
    strassen(k, T1, T2, M7); // M7 = (A12-A22)(B21+B22)

    // Compute result quadrants
    add(k, M1, M4, T1);
    subtract(k, T1, M5, T2);
    add(k, T2, M7, C11); // C11 = M1 + M4 - M5 + M7

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

    add(k, M2, M4, C21); // C21 = M2 + M4

    subtract(k, M1, M2, T1);
    add(k, T1, M3, T2);
    add(k, T2, M6, C22); // C22 = M1 - M2 + M3 + M6

    // Merge results
    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];
            }
    }
```

```c
int main() {
    srand(time(NULL));
    int sizes[] = {2, 4, 8, 16, 32, 64, 128, 256}; // matrix sizes
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int s = 0; s < numSizes; s++) {
        int n = sizes[s];
        int A[n][n], B[n][n], C[n][n];

        generateMatrix(n, A);
        generateMatrix(n, B);

        clock_t start, end;

        // Classic O(n^3)
        start = clock();
        multiplyClassic(n, A, B, C);
        end = clock();
        double tClassic = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Recursive O(n^3)
        start = clock();
        multiplyRecursive(n, A, B, C);
        end = clock();
        double tRecursive = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Strassen O(n^2.81)
        start = clock();
        strassen(n, A, B, C);
        end = clock();
        double tStrassen = ((double)(end - start)) / CLOCKS_PER_SEC;

        printf("Matrix size: %d x %d\n", n, n);
        printf("Classic:   %f sec\n", tClassic);
        printf("Recursive: %f sec\n", tRecursive);
        printf("Strassen:  %f sec\n\n", tStrassen);
    }
    return 0;
}
```

Output:

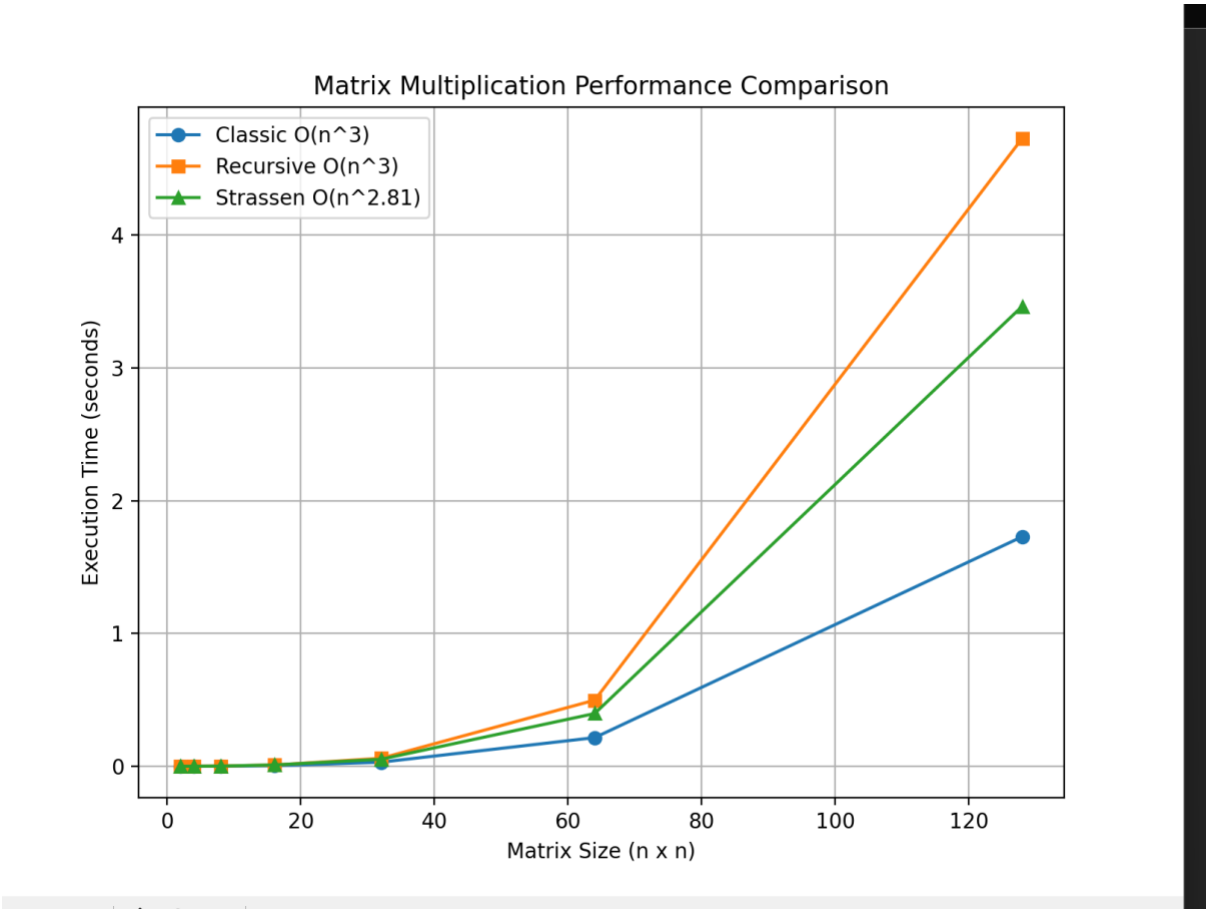
```
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "c:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix\" ; if ($?) { gcc matrix4.c -o matrix4 } ; if ($?) { .\matrix4
Matrix size: 2 x 2
Classic:   0.000000 sec
Recursive: 0.000000 sec
Strassen:  0.000000 sec

Matrix size: 4 x 4
Classic:   0.000000 sec
Recursive: 0.000000 sec
Strassen:  0.000000 sec

Matrix size: 8 x 8
Classic:   0.000000 sec
Recursive: 0.001000 sec
Strassen:  0.000000 sec

Matrix size: 16 x 16
Classic:   0.000000 sec
Recursive: 0.001000 sec
Strassen:  0.000000 sec

Matrix size: 32 x 32
Classic:   0.000000 sec
Recursive: 0.002000 sec
Strassen:  0.001000 sec

Matrix size: 64 x 64
Classic:   0.002000 sec
Recursive: 0.016000 sec
Strassen:  0.014000 sec

Matrix size: 128 x 128
Classic:   0.013000 sec
Recursive: 0.093000 sec
Strassen:  0.079000 sec
```

Python code:

```python
import numpy as np
import time
import matplotlib.pyplot as plt

# ------------------------------
# Matrix multiplication methods
# ------------------------------

# Classic O(n^3)
def classic_multiply(A, B):
    n = len(A)
    C = np.zeros((n, n), dtype=int)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C

# Recursive Divide & Conquer O(n^3)
def recursive_multiply(A, B):
    n = len(A)
    C = np.zeros((n, n), dtype=int)

    if n == 1:
        C[0, 0] = A[0, 0] * B[0, 0]
        return C

    k = n // 2
    A11, A12, A21, A22 = A[:k, :k], A[:k, k:], A[k:, :k], A[k:, k:]
    B11, B12, B21, B22 = B[:k, :k], B[:k, k:], B[k:, :k], B[k:, k:]

    C11 = recursive_multiply(A11, B11) + recursive_multiply(A12, B21)
    C12 = recursive_multiply(A11, B12) + recursive_multiply(A12, B22)
    C21 = recursive_multiply(A21, B11) + recursive_multiply(A22, B21)
    C22 = recursive_multiply(A21, B12) + recursive_multiply(A22, B22)

    C[:k, :k], C[:k, k:], C[k:, :k], C[k:, k:] = C11, C12, C21, C22
    return C

# Strassen O(n^2.81)
def strassen(A, B):
    n = len(A)
    if n == 1:
        return A * B

    k = n // 2
    A11, A12, A21, A22 = A[:k, :k], A[:k, k:], A[k:, :k], A[k:, k:]
    B11, B12, B21, B22 = B[:k, :k], B[:k, k:], B[k:, :k], B[k:, k:]

    M1 = strassen(A11 + A22, B11 + B22)
    M2 = strassen(A21 + A22, B11)
    M3 = strassen(A11, B12 - B22)
    M4 = strassen(A22, B21 - B11)
    M5 = strassen(A11 + A12, B22)
    M6 = strassen(A21 - A11, B11 + B12)
    M7 = strassen(A12 - A22, B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C = np.zeros((n, n), dtype=int)
    C[:k, :k], C[:k, k:], C[k:, :k], C[k:, k:] = C11, C12, C21, C22
    return C

# ------------------------------
# Benchmarking
# ------------------------------

sizes = [2, 4, 8, 16, 32, 64, 128]  # You can increase further if your PC is fast
times_classic = []
times_recursive = []
```

```
times_strassen = []

for n in sizes:
    A = np.random.randint(0, 10, (n, n))
    B = np.random.randint(0, 10, (n, n))

    # Classic
    start = time.time()
    classic_multiply(A, B)
    times_classic.append(time.time() - start)

    # Recursive
    start = time.time()
    recursive_multiply(A, B)
    times_recursive.append(time.time() - start)

    # Strassen
    start = time.time()
    strassen(A, B)
    times_strassen.append(time.time() - start)

    print(f"n={n} done")

plt.figure(figsize=(8,6))
plt.plot(sizes, times_classic, marker='o', label="Classic O(n^3)")
plt.plot(sizes, times_recursive, marker='s', label="Recursive O(n^3)")
plt.plot(sizes, times_strassen, marker='^', label="Strassen O(n^2.81)")
plt.xlabel("Matrix Size (n x n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Matrix Multiplication Performance Comparison")
plt.legend()
plt.grid(True)
plt.show()
```

Graph:



Conclusion:

• **Iterative**: Runs as expected with O(n^3)

• **Divide & Conquer**: Same complexity O(n^3), but extra overhead makes it slower for small n**.**

• **Strassen**: Improves to O(n^{2.81}). For larger matrices, it becomes faster, but for small ones overhead dominates.

Objective: To implement and analyze different approaches for generating Fibonacci numbers and compare their time and space complexities (through graph)
Write algorithms also for each version.

4a. Recursive version
4b Iterative version
4c Dynamic Programming- Top Down Approach
4d Dynamic Programming- Bottom Up Approach

Code:

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#define MAX 100

// -------------------- (a) Recursive --------------------
int fib_recursive(int n) {
    if (n <= 1) return n;
    return fib_recursive(n-1) + fib_recursive(n-2);
}

// -------------------- (b) Iterative --------------------
int fib_iterative(int n) {
    if (n <= 1) return n;
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

// -------------------- (c) DP - Top Down (Memoization) --------------------
int fib_topdown(int n, int memo[]) {
    if (memo[n] != -1) return memo[n];
    memo[n] = fib_topdown(n-1, memo) + fib_topdown(n-2, memo);
    return memo[n];
}

// -------------------- (d) DP - Bottom Up (Tabulation) --------------------
int fib_bottomup(int n) {
    if (n <= 1) return n;
    int dp[MAX];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

// -------------------- Main Function --------------------
int main() {
    int n;
    printf("Enter n for n-th term of the Fibonacci series: ");
    scanf("%d", &n);

    clock_t start, end;
    double cpu_time_used;

    // Recursive
    start = clock();
    int r1 = fib_recursive(n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\n(a) Recursive: %d  | Time: %lf sec\n", r1, cpu_time_used);

    // Iterative
```

```c
int main() {
    // Iterative
    start = clock();
    int r2 = fib_iterative(n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("(b) Iterative: %d  | Time: %lf sec\n", r2, cpu_time_used);

    // Top-Down DP
    int memo[MAX];
    for (int i = 0; i < MAX; i++) memo[i] = -1;
    memo[0] = 0; memo[1] = 1;

    start = clock();
    int r3 = fib_topdown(n, memo);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("(c) DP Top-Down: %d  | Time: %lf sec\n", r3, cpu_time_used);

    // Bottom-Up DP
    start = clock();
    int r4 = fib_bottomup(n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("(d) DP Bottom-Up: %d  | Time: %lf sec\n", r4, cpu_time_used);

    return 0;
}
```

OUTPUT:

```
(d) DP Bottom-Up: 2  | Time: 0.000000 sec
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix> cd "c:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix\" ; if ($?) { gcc fi
 } ; if ($?) { .\fibo }
Enter n for n-th term of the Fibonacci series: 10

(a) Recursive: 55  | Time: 0.000000 sec
(b) Iterative: 55  | Time: 0.000000 sec
(c) DP Top-Down: 55  | Time: 0.000000 sec
(d) DP Bottom-Up: 55  | Time: 0.000000 sec
PS C:\Users\yadav\OneDrive\Pictures\Desktop\cprog\matrix>
```

PYTHON CODE:

```python
import matplotlib.pyplot as plt

# Example data (replace with your measured times from C program)
ns = [5, 10, 15, 20, 25, 30, 35]

recursive_times = [0.0000, 0.0001, 0.0006, 0.006, 0.07, 0.8, 7.5]
iterative_times = [0.0000, 0.0000, 0.0000, 0.0001, 0.0001, 0.0001, 0.0002]
topdown_times   = [0.0000, 0.0000, 0.0000, 0.0001, 0.0001, 0.0001, 0.0002]
bottomup_times  = [0.0000, 0.0000, 0.0000, 0.0001, 0.0001, 0.0001, 0.0002]

# Plotting
plt.figure(figsize=(8,6))
plt.plot(ns, recursive_times, 'r-o', label="Recursive")
plt.plot(ns, iterative_times, 'g-o', label="Iterative")
plt.plot(ns, topdown_times, 'b-o', label="DP Top-Down")
plt.plot(ns, bottomup_times, 'm-o', label="DP Bottom-Up")

plt.xlabel("n (Fibonacci index)")
plt.ylabel("Execution Time (seconds)")
plt.title("Fibonacci Algorithms: Time Complexity Comparison")
plt.legend()
plt.grid(True)
plt.show()
```

GRAPH: