24293916070-ADA-Sourish Bandaru

INDEX

1A- Write a C program to implement a 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-

```
LinearSearch(Array, Target)
    Input: Array of n elements, and a Target value
    Output: Index of Target if found, otherwise -1

    for i ← 0 to n-1 do
        if Array[i] = Target then
            return i   // Target found at index i
        end if
    end for

    return -1   // Target not found
```

CODE-

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

// Linear Search with light artificial delay
int linear_search(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        // small artificial delay (keeps graph smooth but not too slow)
        for (volatile int j = 0; j < 20; j++);
        if (arr[i] == target) return i;
    }
    return -1;
}

int main() {
    FILE *fp = fopen("linear_search_results.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "size,avg_time\n");
    printf(" size\t avg_time (seconds)\n");
```

```
    printf("------------------------------\n");

    for (int n = 1000; n <= 100000; n += 5000) {
        int *arr = malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) arr[i] = i;

        int target = n - 1; // worst case (last element)

        // Adjust number of iterations depending on n (to prevent lag)
        int iterations = (n <= 20000) ? 5000 : 500;

        clock_t start = clock();
        for (int i = 0; i < iterations; i++) {
            linear_search(arr, n, target);
        }
        clock_t end = clock();

        double total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
        double avg_time = total_time / iterations;

        fprintf(fp, "%d,%f\n", n, avg_time);
        printf("%6d\t %f\n", n, avg_time);

        free(arr);
    }

    fclose(fp);
    printf("\nResults written to linear_search_results.txt\n");
    return 0;
}
```

OUTPUT-

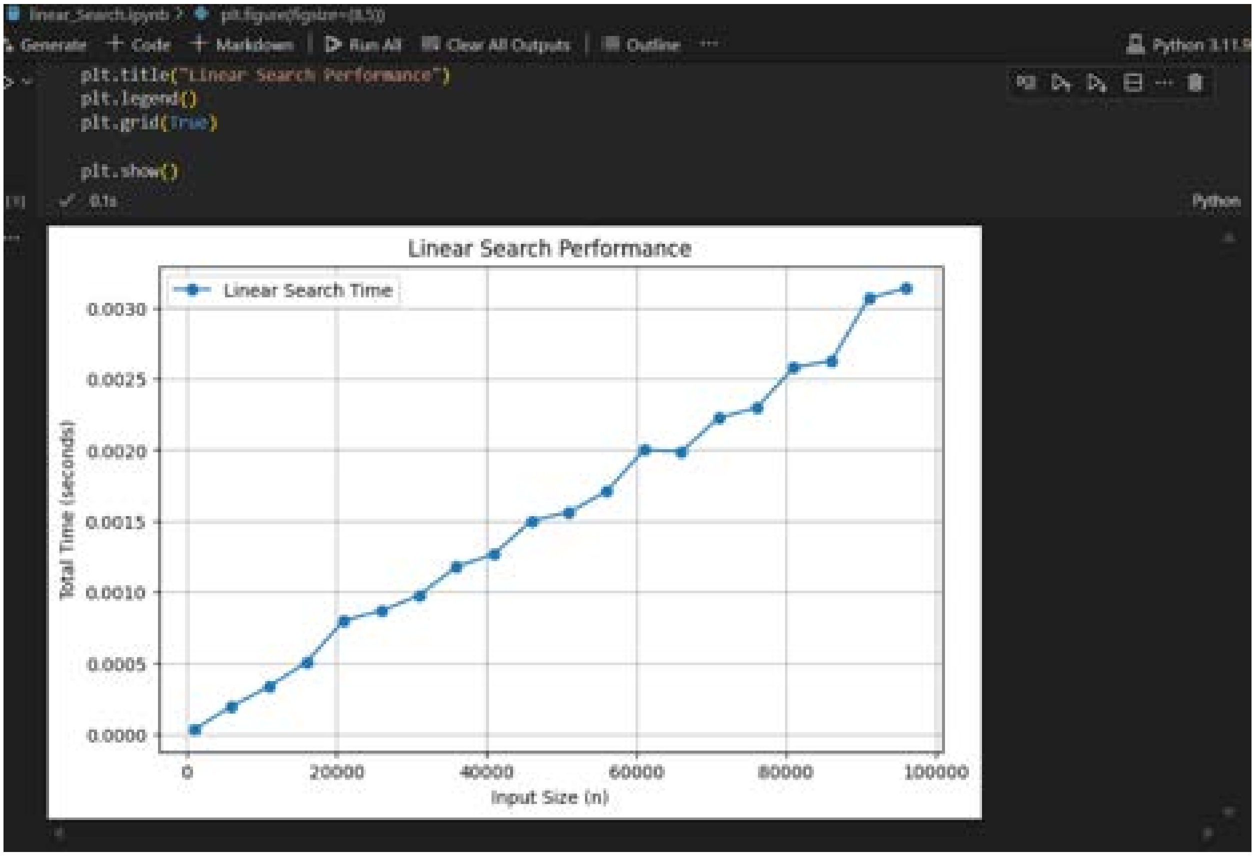
```
souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/linear search
$ gcc linear_search.c

souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/linear search
$ ./a.exe
 size    avg_time (seconds)
------------------------------
  1000   0.000033
  6000   0.000197
 11000   0.000334
 16000   0.000507
 21000   0.000802
 26000   0.000868
 31000   0.000976
 36000   0.001180
 41000   0.001266
 46000   0.001502
 51000   0.001562
 56000   0.001708
 61000   0.002000
 66000   0.001998
 71000   0.002226
 76000   0.002294
 81000   0.002580
 86000   0.002626
 91000   0.003068
 96000   0.003138

Results written to linear_search_results.txt

souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/linear search
$
```

PYTHON CODE-

```python
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

```python
df= pd.read_csv(r'C:\Users\souri\OneDrive\Desktop\Sourish\College\ADA\linear search\linear_search_results.txt')
```

```python
plt.figure(figsize=(8,5))

plt.plot(df["size"], df["avg_time"], marker='o', linestyle='-', label="Linear Search Time")

plt.xlabel("Input Size (n)")
plt.ylabel("Total Time (seconds)")
plt.title("Linear Search Performance")
plt.legend()
plt.grid(True)

plt.show()
```



OBSERVATION AND CONCLUSION-

The provided data demonstrates that a linear search algorithm's efficiency is directly tied to the size of the dataset. Its theoretical time complexity of O(n) was reflected in the practical results, where the search time would ideally increase as the array size grew. .

1(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-

```
BinarySearch(Array, Target)
   Input: Sorted Array of n elements, and a Target value
   Output: Index of Target if found, otherwise -1

   left ← 0
   right ← n - 1

   while left ≤ right do
      mid ← (left + right) // 2   // integer division

      if Array[mid] = Target then
         return mid   // Target found at index mid

      else if Array[mid] < Target then
         left ← mid + 1   // Search right half

      else
         right ← mid - 1  // Search left half
      end if
   end while

   return -1   // Target not found
```

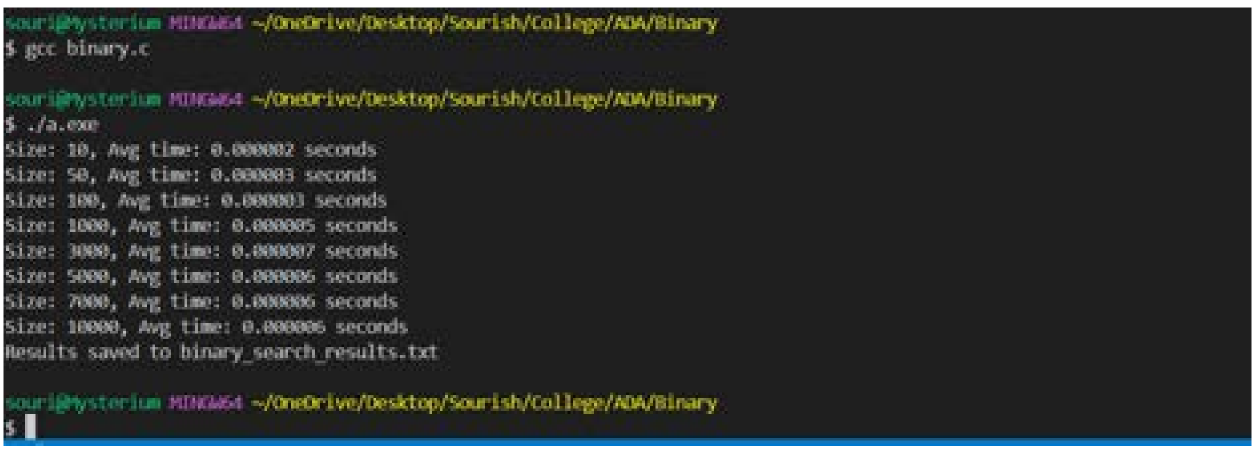
CODE-

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

// Binary search with artificial delay
int binary_search(int arr[], int size, int target) {
   int low = 0;
   int high = size - 1;

   while (low <= high) {
      int mid = low + (high - low) / 2;

      // Artificial delay loop (does nothing)
      for (volatile int j = 0; j < 1000; j++);

      if (arr[mid] == target) {
         return mid;
      } else if (arr[mid] < target) {
         low = mid + 1;
```

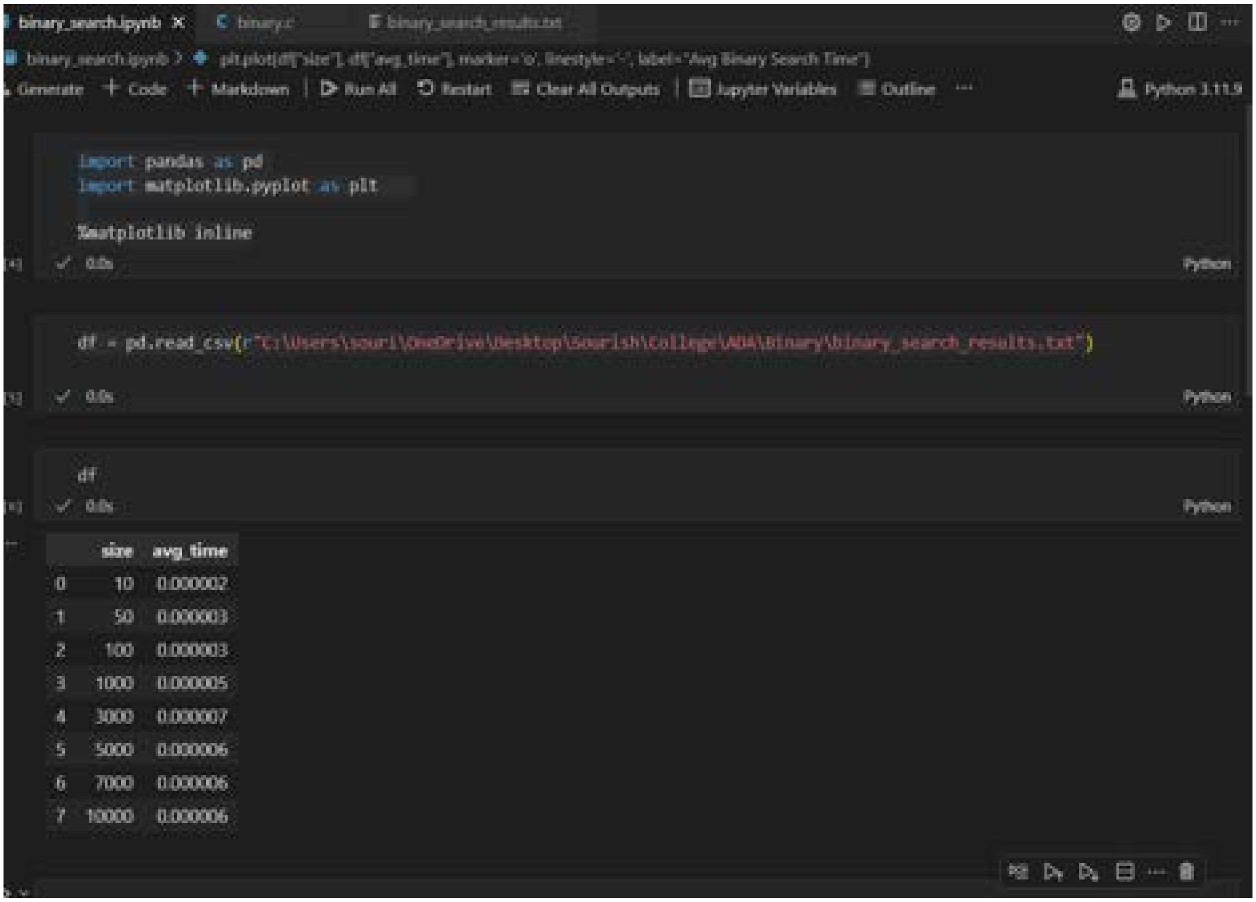
```c
        } else {
            high = mid - 1;
        }
    }

    return -1;
}

void generateRandomArray(int array[], int size){
    for(int i = 0; i < size; i++)
        array[i] = rand() % 100000;
}

int cmpfunc(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

int main() {
    int input_size[8] = {10, 50, 100, 1000, 3000, 5000, 7000, 10000};

    // Open file to write results
    FILE *fp = fopen("binary_search_results.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "size,avg_time\n"); // CSV header

    for(int i = 0; i < 8; i++){
        int size = input_size[i];
        int array[size];

        generateRandomArray(array, size);
        qsort(array, size, sizeof(int), cmpfunc);

        int target = array[size-1] + 1; // worst-case: not in array

        double total_time = 0;
        int iterations = 100000; // batch size

        clock_t start = clock();
        for(int iter = 0; iter < iterations; iter++) {
            binary_search(array, size, target);
```

```
        }
        clock_t end = clock();

        total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
        double avg_time = total_time / iterations;

        printf("Size: %d, Avg time: %lf seconds\n", size, avg_time);
        fprintf(fp, "%d,%lf\n", size, avg_time);
    }

    fclose(fp);
    printf("Results saved to binary_search_results.txt\n");
    return 0;
}
```

OUTPUT-



PYTHON CODE-

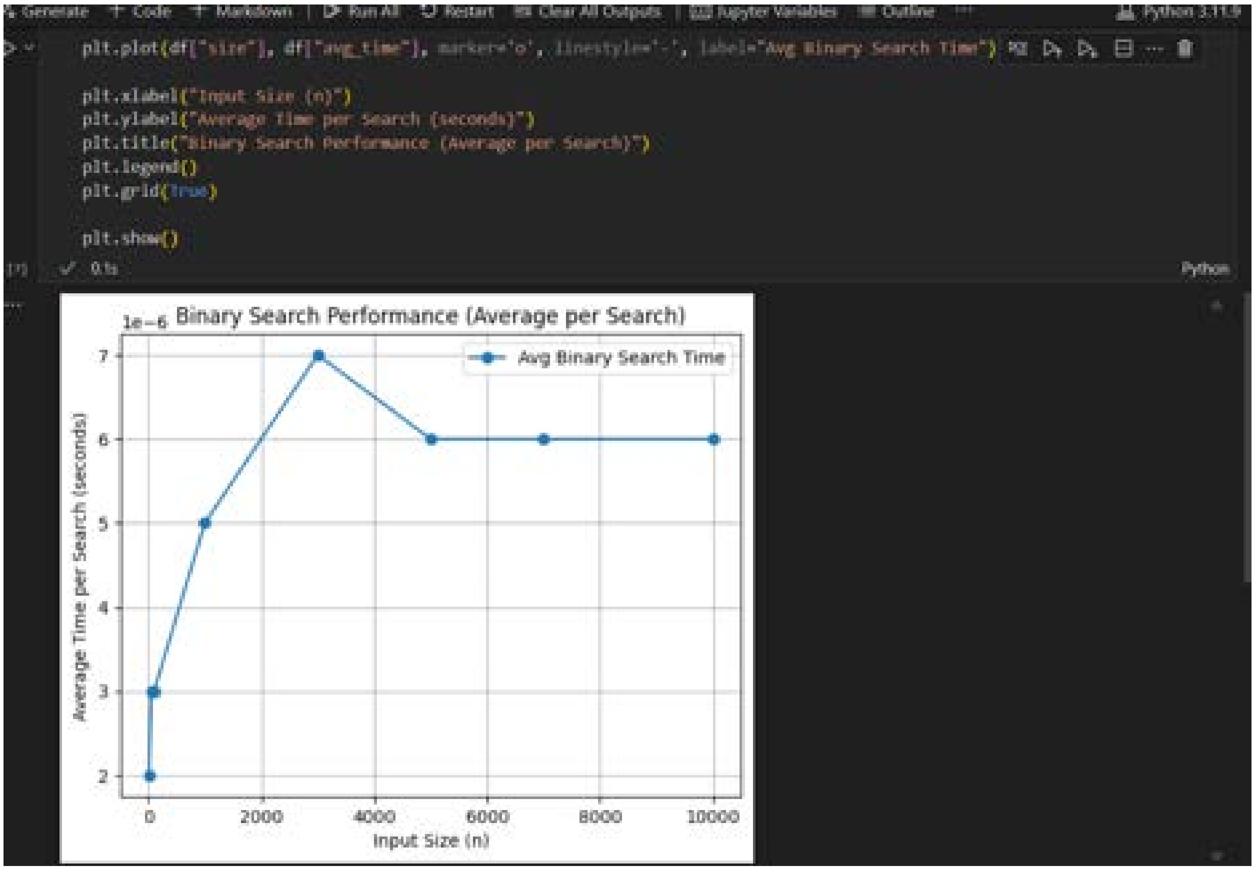


```python
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline
```

```python
df = pd.read_csv(r"C:\Users\souri\OneDrive\Desktop\Sourish\College\ADA\Binary\binary_search_results.txt")
```

```python
df
```

	size	avg_time
0	10	0.000002
1	50	0.000003
2	100	0.000003
3	1000	0.000005
4	3000	0.000007
5	5000	0.000006
6	7000	0.000006
7	10000	0.000006



OBSERVATION AND CONCLUSION-

Binary search proved to be an extremely efficient algorithm for searching sorted arrays, even when dealing with a large number of elements. The time complexity of O(logn) demonstrates its superior performance over linear search, where the search space is halved in each step. The minor fluctuations in execution time observed across different runs were not indicative of any algorithm flaw but rather a reflection of system timing limitations and randomness, which can slightly affect the precise measurements in a real-world computing environment.

Github Link-https://github.com/sourishbandaru/24293916070-CseA-ADA1-Sorting.git

Q1-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.

Pseudo Code-

```
MERGE_SORT(A):
   if length(A) <= 1:
      return A

   mid ← length(A) / 2
   left ← MERGE_SORT(A[0 .. mid-1])
   right ← MERGE_SORT(A[mid .. end])

   return MERGE(left, right)

MERGE(left, right):
   result ← empty list
   i ← 0, j ← 0

   while i < length(left) and j < length(right):
      if left[i] ≤ right[j]:
         append left[i] to result
         i ← i + 1
      else:
         append right[j] to result
         j ← j + 1

   // add remaining elements
   while i < length(left):
      append left[i] to result
      i ← i + 1

   while j < length(right):
```

```
        append right[j] to result
        j ← j + 1

    return result
```

Code-

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

// Merge function
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

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

    free(L);
```

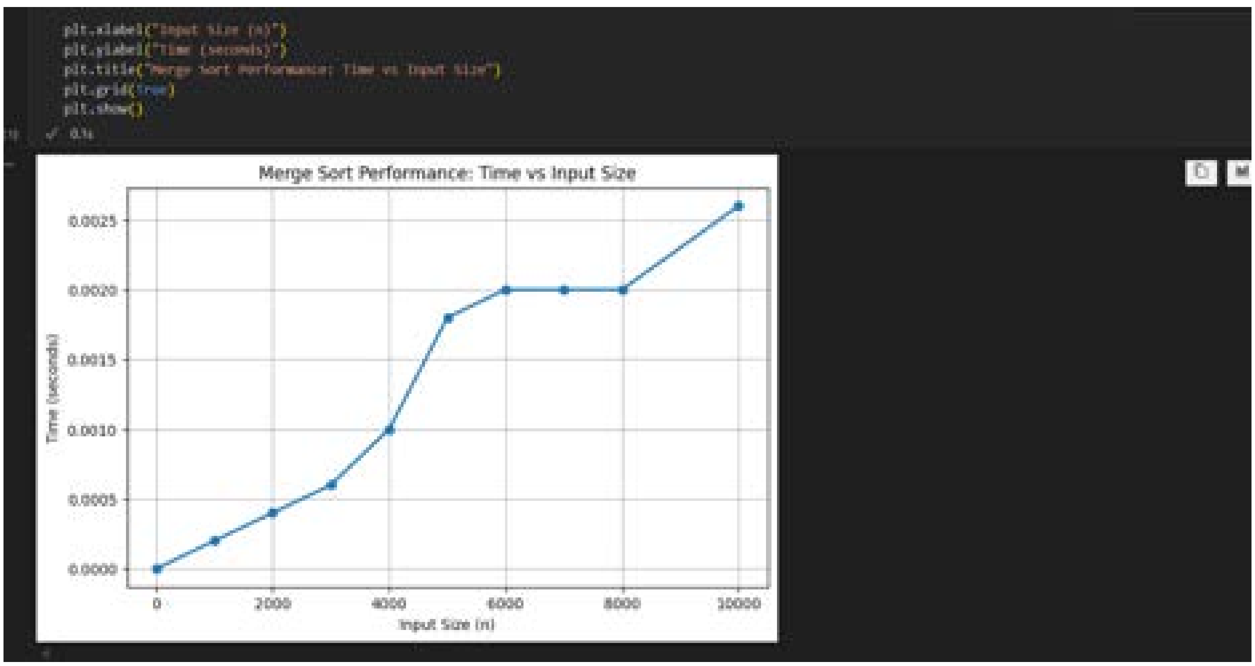
```
    free(R);
}

// Merge Sort
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

int main() {
    srand(time(NULL));
    FILE *fp;

    // Predefined array of sizes
    int sizes[10] = {0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 10000};

    fp = fopen("merge.txt", "w"); // overwrite each run
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    int trials = 5; // average over 5 runs

    for (int k = 0; k < 10; k++) {
        int n = sizes[k];
        if (n == 0) {
            fprintf(fp, "0 0.0\n");
            printf("0 0.0\n");
            continue;
        }

        double total_time = 0.0;

        for (int t = 0; t < trials; t++) {
```

```
        int *arr = (int*)malloc(n * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed for size %d!\n", n);
            continue;
        }

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

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

        total_time += ((double)(end - start)) / CLOCKS_PER_SEC;

        free(arr);
    }

    double avg_time = total_time / trials;

    // Print result
    printf("%d %.6f\n", n, avg_time);

    // Save to file
    fprintf(fp, "%d %.6f\n", n, avg_time);
}

fclose(fp);
return 0;
}
```

Output-

```
$ cd "C:\Users\souri\OneDrive\Desktop\Sourish\College\ADA\Sorting\Merge"

souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/Sorting/Merge
$ gcc mergesort.c

souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/Sorting/Merge
$ ./a.exe
0 0.0
1000 0.000200
2000 0.000400
3000 0.000600
4000 0.001000
5000 0.001800
6000 0.002000
7000 0.002000
8000 0.002000
10000 0.002600

souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/Sorting/Merge
$
```

Python Graph-



Q2-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.

Pseudo Code-

```
QUICK_SORT(A, low, high):
   if low < high:
      pivotIndex ← PARTITION(A, low, high)
      QUICK_SORT(A, low, pivotIndex - 1)   // sort left side
      QUICK_SORT(A, pivotIndex + 1, high) // sort right side

PARTITION(A, low, high):
   pivot ← A[high]         // choose last element as pivot
   i ← low - 1             // index of smaller element

   for j ← low to high - 1:
      if A[j] ≤ pivot:
         i ← i + 1
         swap A[i] with A[j]

   swap A[i + 1] with A[high]
   return i + 1            // pivot position
```

Code-

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

// Swap two elements
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
```

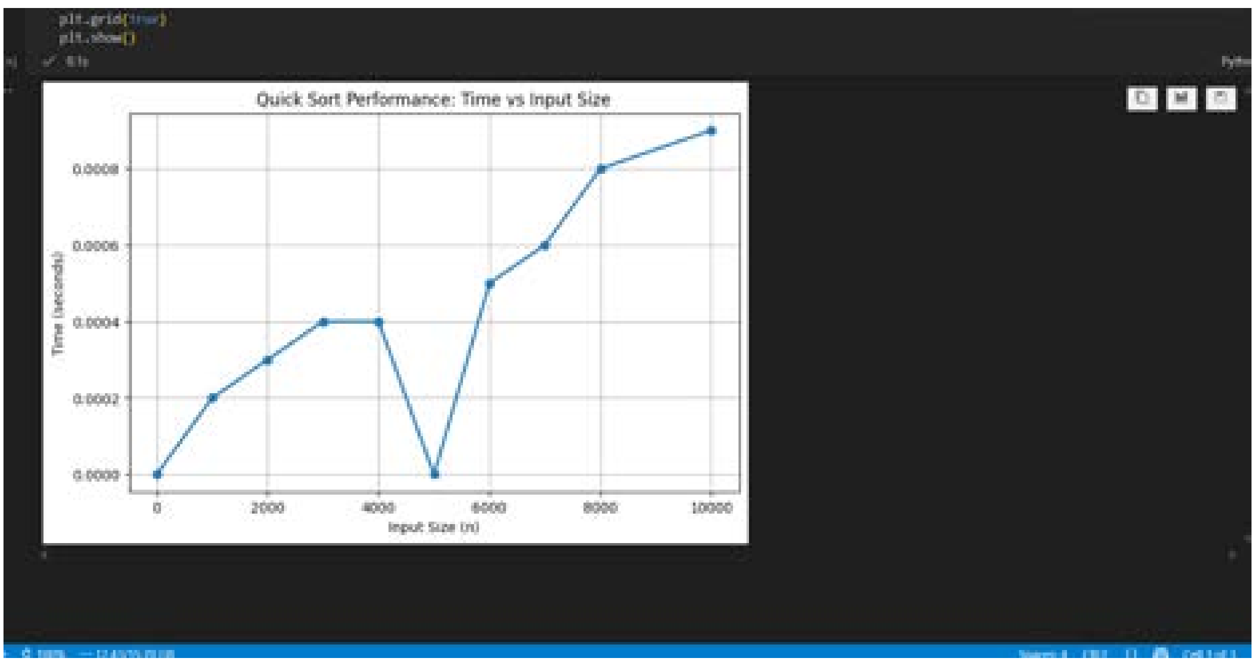
```
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// QuickSort
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);
    }
}

int main() {
    srand(time(NULL));
    FILE *fp;

    // Predefined array of sizes
    int sizes[10] = {0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 10000};

    fp = fopen("quicksort_data.txt", "w"); // overwrite each run
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    int trials = 10; // number of repetitions for averaging

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

```
    if (n == 0) {
        fprintf(fp, "0 0.0\n");
        printf("0 0.0\n");
        continue;
    }

    double total_time = 0.0;

    for (int t = 0; t < trials; t++) {
        int *arr = (int*)malloc(n * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed for size %d!\n", n);
            continue;
        }

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

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

        total_time += ((double)(end - start)) / CLOCKS_PER_SEC;

        free(arr);
    }

    double avg_time = total_time / trials;

    // Print result
    printf("%d %.6f\n", n, avg_time);

    // Save to file
    fprintf(fp, "%d %.6f\n", n, avg_time);
}
```

```
    fclose(fp);
    return 0;
}
```

Output-

Python Output-



Q3-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

Pseudo Code-

```
INSERTION_SORT(A):
    for i ← 1 to length(A) - 1:
        key ← A[i]
        j ← i - 1

        // Move elements greater than key to one position ahead
        while j ≥ 0 and A[j] > key:
            A[j + 1] ← A[j]
            j ← j - 1

        A[j + 1] ← key
```

Code-

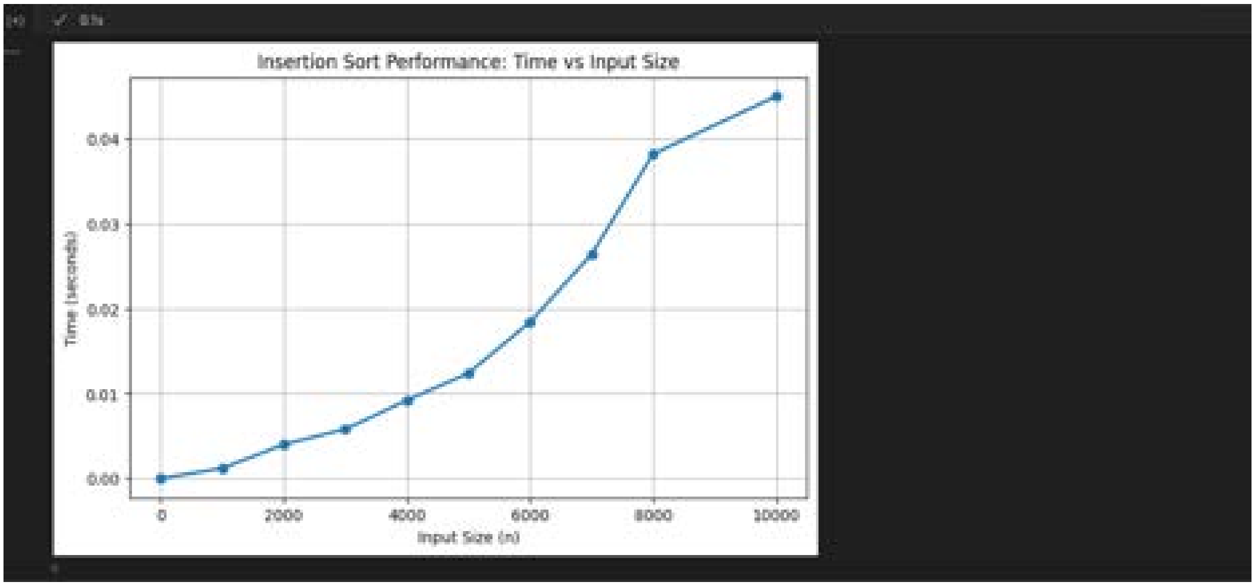
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Insertion Sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
```

```
srand(time(NULL));
FILE *fp;

// Predefined array of sizes
int sizes[10] = {0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 10000};

fp = fopen("insertion_data.txt", "w"); // overwrite each run
if (fp == NULL) {
    printf("Error opening file!\n");
    return 1;
}

int trials = 5; // average across multiple runs for accuracy

for (int k = 0; k < 10; k++) {
    int n = sizes[k];
    if (n == 0) {
        fprintf(fp, "0 0.0\n");
        printf("0 0.0\n");
        continue;
    }

    double total_time = 0.0;

    for (int t = 0; t < trials; t++) {
        int *arr = (int*)malloc(n * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed for size %d!\n", n);
            continue;
        }

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

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

```c
            end = clock();

            total_time += ((double)(end - start)) / CLOCKS_PER_SEC;

            free(arr);
        }

        double avg_time = total_time / trials;

        // Print result
        printf("%d %.6f\n", n, avg_time);

        // Save to file
        fprintf(fp, "%d %.6f\n", n, avg_time);
    }

    fclose(fp);
    return 0;
}
```

Output-

Python Output-



Insertion Sort Performance: Time vs Input Size
Time (seconds)
0.04
0.03
0.02
0.01
0.00
0
2000
4000
6000
8000
10000
Input Size (n)

Q4-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

Pseudo Code-

```
SELECTION_SORT(A):
   n ← length(A)

   for i ← 0 to n - 2:
      minIndex ← i

      // Find index of smallest element in remaining array
      for j ← i + 1 to n - 1:
         if A[j] < A[minIndex]:
            minIndex ← j

      // Swap smallest element with A[i]
      if minIndex ≠ i:
         swap A[i] ↔ A[minIndex]
```

Code-

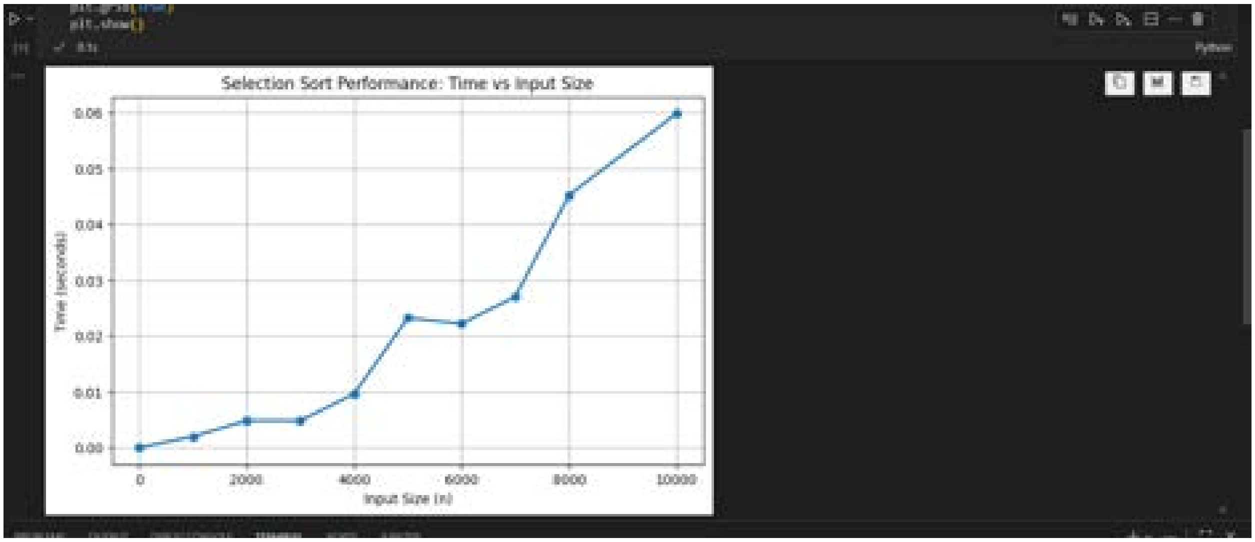
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

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

```
        arr[i] = temp;
    }
}

int main() {
    srand(time(NULL));
    FILE *fp;

    // Predefined array of sizes
    int sizes[10] = {0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 10000};

    fp = fopen("selection.txt", "w"); // overwrite each run
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    int trials = 5; // run multiple times and average

    for (int k = 0; k < 10; k++) {
        int n = sizes[k];
        if (n == 0) {
            fprintf(fp, "0 0.0\n");
            printf("0 0.0\n");
            continue;
        }

        double total_time = 0.0;

        for (int t = 0; t < trials; t++) {
            int *arr = (int*)malloc(n * sizeof(int));
            if (arr == NULL) {
                printf("Memory allocation failed for size %d!\n", n);
                continue;
            }

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

```
            }

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

            total_time += ((double)(end - start)) / CLOCKS_PER_SEC;

            free(arr);
        }

        double avg_time = total_time / trials;

        // Print result
        printf("%d %.6f\n", n, avg_time);

        // Save to file
        fprintf(fp, "%d %.6f\n", n, avg_time);
    }

    fclose(fp);
    return 0;
}
```

Output-

Python Output-



Q5-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

Pseudo Code-

```
BUBBLE_SORT(A):
    n ← length(A)

    for i ← 0 to n - 1:
        swapped ← false

        // Last i elements are already in place
        for j ← 0 to n - i - 2:
            if A[j] > A[j + 1]:
                swap A[j] ↔ A[j + 1]
                swapped ← true

        // If no elements were swapped, array is sorted
        if swapped = false:
            break
```

Code-

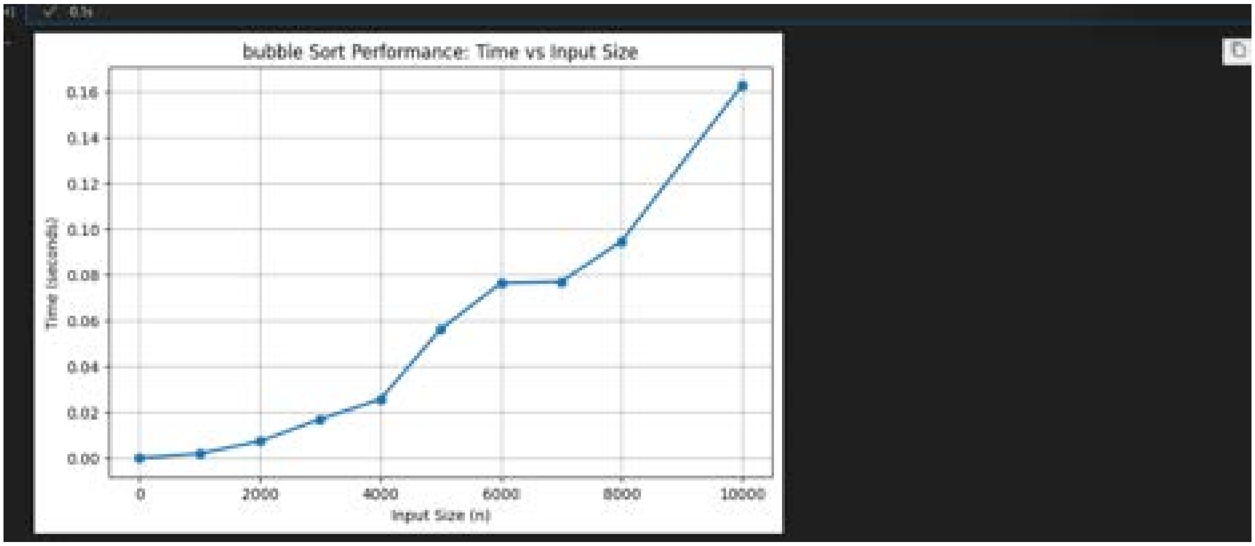
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

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

```
    }
}

int main() {
    srand(time(NULL));
    FILE *fp;

    // Predefined array of sizes
    int sizes[10] = {0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 10000};

    fp = fopen("bubble.txt", "w"); // overwrite each run
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    int trials = 3; // fewer trials, bubble sort is slow!

    for (int k = 0; k < 10; k++) {
        int n = sizes[k];
        if (n == 0) {
            fprintf(fp, "0 0.0\n");
            printf("0 0.0\n");
            continue;
        }

        double total_time = 0.0;

        for (int t = 0; t < trials; t++) {
            int *arr = (int*)malloc(n * sizeof(int));
            if (arr == NULL) {
                printf("Memory allocation failed for size %d!\n", n);
                continue;
            }

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

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

            total_time += ((double)(end - start)) / CLOCKS_PER_SEC;

            free(arr);
        }

        double avg_time = total_time / trials;

        // Print result
        printf("%d %.6f\n", n, avg_time);

        // Save to file
        fprintf(fp, "%d %.6f\n", n, avg_time);
    }

    fclose(fp);
    return 0;
}
```

Output-

Python Output-



Q1-Write a program in C language to multiply two square matrices using the iterative approach. Compare the execution time for different matrix sizes.

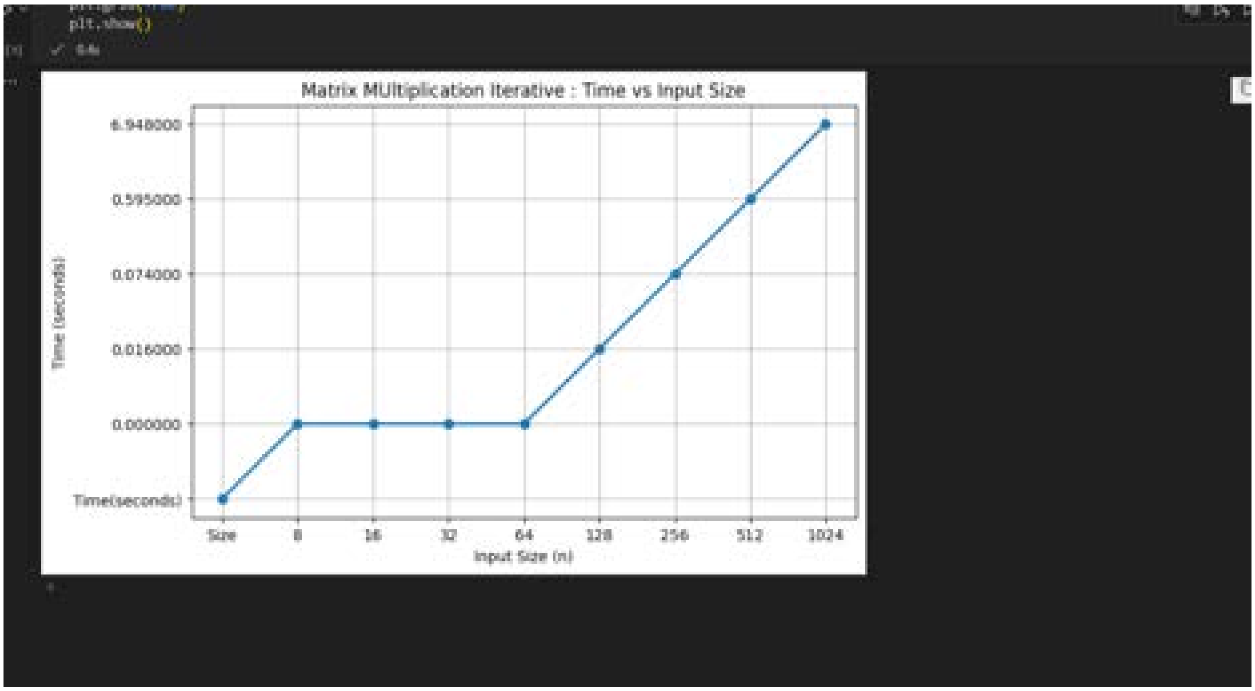
Code-

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

void multiplyMatrices(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];
            }
        }
    }
}

int main() {
    int sizes[] = { 8, 16,32, 64,128,256,512,1024};
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    srand(time(NULL));


    FILE *fp = fopen("results.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "Size Time(seconds)\n");

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


        int (*A)[n] = malloc(sizeof(int[n][n]));
        int (*B)[n] = malloc(sizeof(int[n][n]));
```

```
        int (*C)[n] = malloc(sizeof(int[n][n]));


        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = rand() % 10;
                B[i][j] = rand() % 10;
            }
        }

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

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

        printf(" %dx%d => : %.6f \n", n, n, time_taken);
        fprintf(fp, "%d %.6f\n", n, time_taken);


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

    fclose(fp);
    printf("Results stored in results.txt\n");

    return 0;
}
```

Output-

Python Output-



Q2- Write a program in C language to multiply two square matrices using the . Compare the execution time for different matrix sizes.

Code-

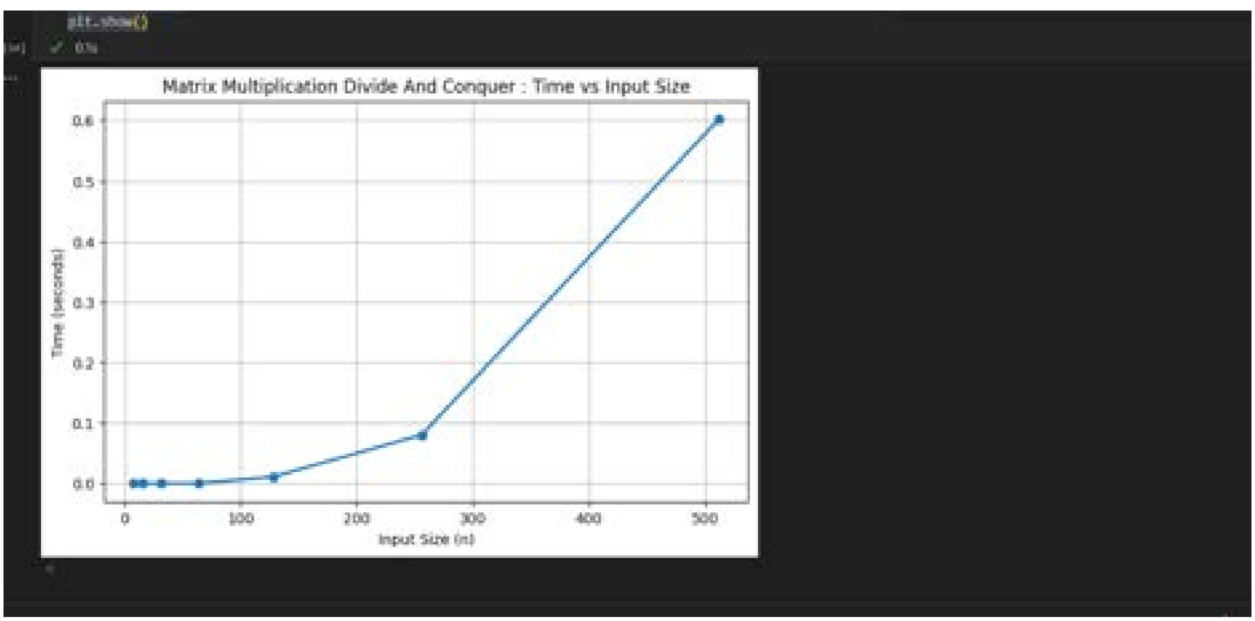
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Standard matrix multiplication
void multiply(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Allocate 2D array dynamically
int **allocate_matrix(int n) {
    int **mat = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        mat[i] = (int *)malloc(n * sizeof(int));
    }
    return mat;
}

// Free 2D array
void free_matrix(int **mat, int n) {
    for (int i = 0; i < n; i++) {
        free(mat[i]);
    }
    free(mat);
}

// Fill matrix with random numbers
void fill_matrix(int **mat, int n) {
```

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            mat[i][j] = rand() % 10; // values 0–9
}

int main() {
    srand(time(NULL));

    int sizes[] = {8, 16, 32, 64, 128, 256, 512};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    FILE *fp = fopen("Data.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

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

        // Allocate matrices
        int **A = allocate_matrix(n);
        int **B = allocate_matrix(n);
        int **C = allocate_matrix(n);

        // Fill A and B with random numbers
        fill_matrix(A, n);
        fill_matrix(B, n);

        // Start timing
        clock_t start = clock();
        multiply(A, B, C, n);
        clock_t end = clock();

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

        printf("%d => %.6f\n", n, elapsed);
        fprintf(fp, "%d %.6f\n", n, elapsed);
```

```
        // Free memory
        free_matrix(A, n);
        free_matrix(B, n);
        free_matrix(C, n);
    }

    fclose(fp);

    printf("Results stored in Data.txt\n");
    return 0;
}
```

Output-

Python Output-



Q3-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-

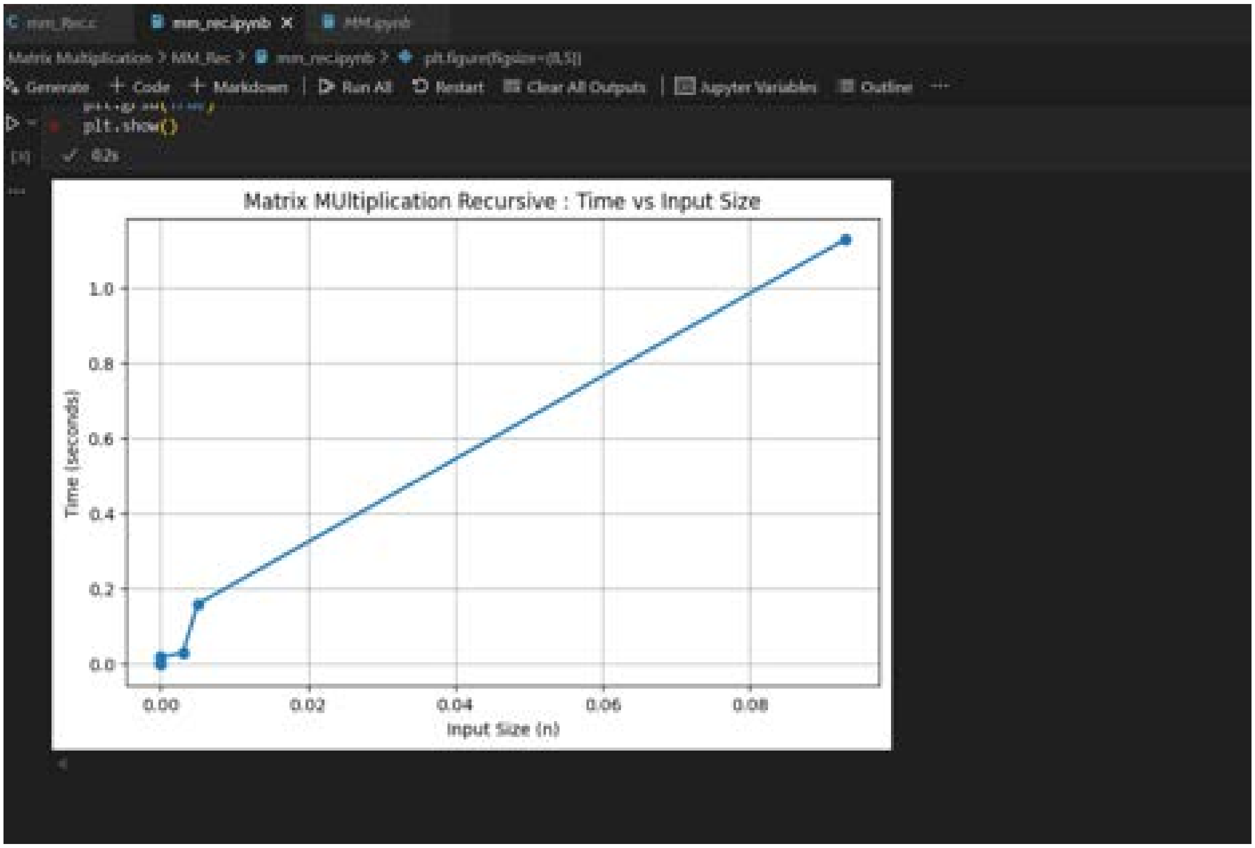
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Allocate memory for a matrix
int** allocate_matrix(int n) {
    int** matrix = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++)
        matrix[i] = (int*)malloc(n * sizeof(int));
    return matrix;
}

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

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

// Subtract two matrices
void sub_matrix(int** A, int** B, int** C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}

// Standard matrix multiplication (O(n^3))
void normal_multiply(int** A, int** B, int** C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
```

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

// Strassen's Algorithm
void strassen(int** A, int** B, int** C, int n) {
    if (n <= 2) {
        normal_multiply(A, B, C, n); // base case
        return;
    }

    int k = n / 2;

    // Allocate submatrices
    int** A11 = allocate_matrix(k); int** A12 = allocate_matrix(k);
    int** A21 = allocate_matrix(k); int** A22 = allocate_matrix(k);
    int** B11 = allocate_matrix(k); int** B12 = allocate_matrix(k);
    int** B21 = allocate_matrix(k); int** B22 = allocate_matrix(k);
    int** C11 = allocate_matrix(k); int** C12 = allocate_matrix(k);
    int** C21 = allocate_matrix(k); int** C22 = allocate_matrix(k);

    // Temporary matrices
    int** M1 = allocate_matrix(k); int** M2 = allocate_matrix(k);
    int** M3 = allocate_matrix(k); int** M4 = allocate_matrix(k);
    int** M5 = allocate_matrix(k); int** M6 = allocate_matrix(k);
    int** M7 = allocate_matrix(k);
    int** T1 = allocate_matrix(k); int** T2 = allocate_matrix(k);

    // Split matrices into submatrices
    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_matrix(A11, A22, T1, k);  add_matrix(B11, B22, T2, k);  strassen(T1, T2, M1, k);
    add_matrix(A21, A22, T1, k);  strassen(T1, B11, M2, k);
    sub_matrix(B12, B22, T2, k);  strassen(A11, T2, M3, k);
    sub_matrix(B21, B11, T2, k);  strassen(A22, T2, M4, k);
    add_matrix(A11, A12, T1, k);  strassen(T1, B22, M5, k);
    sub_matrix(A21, A11, T1, k);  add_matrix(B11, B12, T2, k);  strassen(T1, T2, M6, k);
    sub_matrix(A12, A22, T1, k);  add_matrix(B21, B22, T2, k);  strassen(T1, T2, M7, k);

    // Compute C11, C12, C21, C22
    add_matrix(M1, M4, T1, k);  sub_matrix(T1, M5, T2, k);  add_matrix(T2, M7, C11, k);
    add_matrix(M3, M5, C12, k);
    add_matrix(M2, M4, C21, k);
    sub_matrix(M1, M2, T1, k);  add_matrix(T1, M3, T2, k);  add_matrix(T2, M6, C22, k);

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

    // Free memory
    free_matrix(A11, k); free_matrix(A12, k); free_matrix(A21, k); free_matrix(A22, k);
    free_matrix(B11, k); free_matrix(B12, k); free_matrix(B21, k); free_matrix(B22, k);
    free_matrix(C11, k); free_matrix(C12, k); free_matrix(C21, k); free_matrix(C22, k);
    free_matrix(M1, k); free_matrix(M2, k); free_matrix(M3, k); free_matrix(M4, k);
    free_matrix(M5, k); free_matrix(M6, k); free_matrix(M7, k);
    free_matrix(T1, k); free_matrix(T2, k);
}

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

    FILE *fp = fopen("Data.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
```

```
    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        int** A = allocate_matrix(n);
        int** B = allocate_matrix(n);
        int** C = allocate_matrix(n);

        // Fill A and B with random numbers
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) {
                A[i][j] = rand() % 10;
                B[i][j] = rand() % 10;
            }

        clock_t start, end;

        // Standard multiplication
        start = clock();
        normal_multiply(A, B, C, n);
        end = clock();
        double time_normal = (double)(end - start) / CLOCKS_PER_SEC;

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

        printf("   Normal multiply:   %f sec\n", time_normal);
        printf("   Strassen multiply: %f sec\n", time_strassen);
        printf("--------------------------\n");

        // Write results into file (clean format)
        fprintf(fp, "%d %.6f %.6f\n", n, time_normal, time_strassen);

        free_matrix(A, n);
        free_matrix(B, n);
        free_matrix(C, n);
    }

    fclose(fp);
    printf("Results stored in Data.txt\n");

    return 0;
}
```

Output-



Matrix MUltiplication Recursive : Time vs Input Size
Time (seconds)
Input Size (n)
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.02
0.04
0.06
0.08
plt.show()

Q1-Write a program in C language to multiply two square matrices using the iterative approach. Compare the execution time for different matrix sizes.

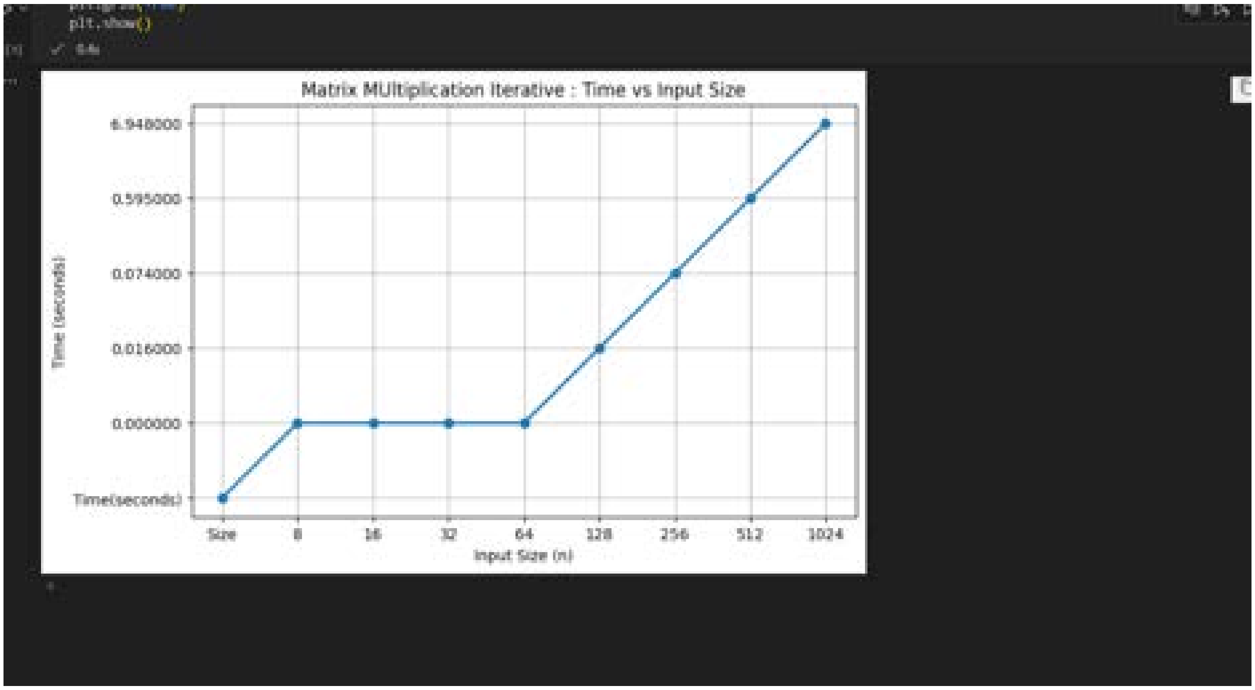
Code-

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

void multiplyMatrices(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];
            }
        }
    }
}

int main() {
    int sizes[] = { 8, 16,32, 64,128,256,512,1024};
    int numSizes = sizeof(sizes) / sizeof(sizes[0]);

    srand(time(NULL));


    FILE *fp = fopen("results.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "Size Time(seconds)\n");

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


        int (*A)[n] = malloc(sizeof(int[n][n]));
        int (*B)[n] = malloc(sizeof(int[n][n]));
```

```
        int (*C)[n] = malloc(sizeof(int[n][n]));


        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = rand() % 10;
                B[i][j] = rand() % 10;
            }
        }

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

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

        printf(" %dx%d => : %.6f \n", n, n, time_taken);
        fprintf(fp, "%d %.6f\n", n, time_taken);


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

    fclose(fp);
    printf("Results stored in results.txt\n");

    return 0;
}
```

Output-

Python Output-



Q2- Write a program in C language to multiply two square matrices using the . Compare the execution time for different matrix sizes.

Code-

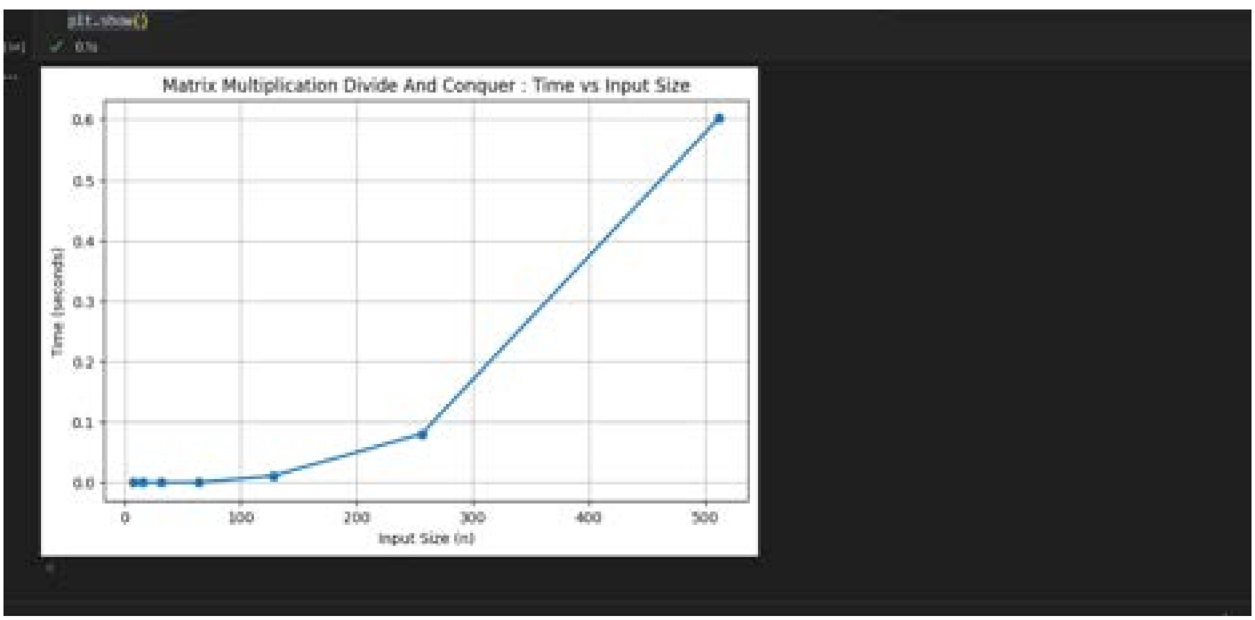
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Standard matrix multiplication
void multiply(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Allocate 2D array dynamically
int **allocate_matrix(int n) {
    int **mat = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        mat[i] = (int *)malloc(n * sizeof(int));
    }
    return mat;
}

// Free 2D array
void free_matrix(int **mat, int n) {
    for (int i = 0; i < n; i++) {
        free(mat[i]);
    }
    free(mat);
}

// Fill matrix with random numbers
void fill_matrix(int **mat, int n) {
```

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            mat[i][j] = rand() % 10; // values 0–9
}

int main() {
    srand(time(NULL));

    int sizes[] = {8, 16, 32, 64, 128, 256, 512};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    FILE *fp = fopen("Data.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

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

        // Allocate matrices
        int **A = allocate_matrix(n);
        int **B = allocate_matrix(n);
        int **C = allocate_matrix(n);

        // Fill A and B with random numbers
        fill_matrix(A, n);
        fill_matrix(B, n);

        // Start timing
        clock_t start = clock();
        multiply(A, B, C, n);
        clock_t end = clock();

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

        printf("%d => %.6f\n", n, elapsed);
        fprintf(fp, "%d %.6f\n", n, elapsed);
```

```
        // Free memory
        free_matrix(A, n);
        free_matrix(B, n);
        free_matrix(C, n);
    }

    fclose(fp);

    printf("Results stored in Data.txt\n");
    return 0;
}
```

Output-

Python Output-



Q3-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-

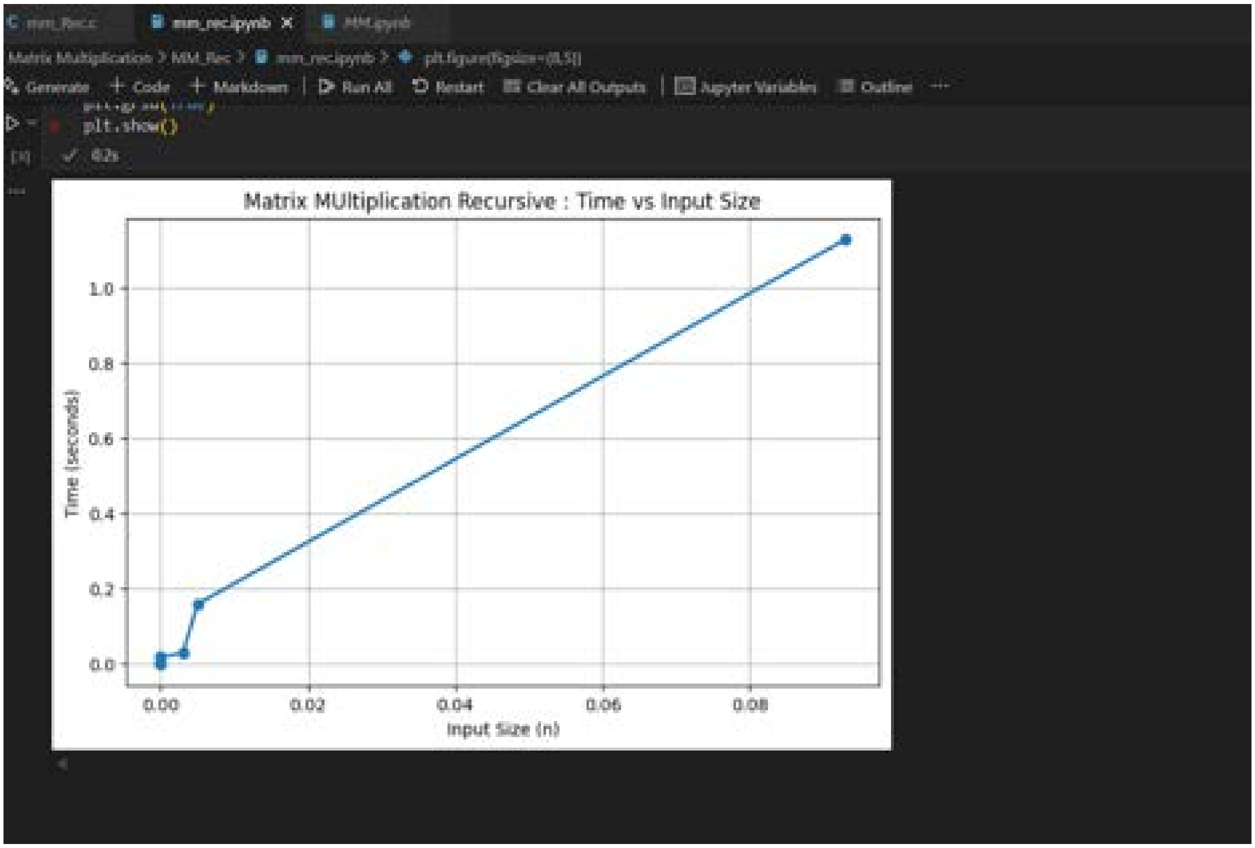
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Allocate memory for a matrix
int** allocate_matrix(int n) {
    int** matrix = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++)
        matrix[i] = (int*)malloc(n * sizeof(int));
    return matrix;
}

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

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

// Subtract two matrices
void sub_matrix(int** A, int** B, int** C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}

// Standard matrix multiplication (O(n^3))
void normal_multiply(int** A, int** B, int** C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
```

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

// Strassen's Algorithm
void strassen(int** A, int** B, int** C, int n) {
    if (n <= 2) {
        normal_multiply(A, B, C, n); // base case
        return;
    }

    int k = n / 2;

    // Allocate submatrices
    int** A11 = allocate_matrix(k); int** A12 = allocate_matrix(k);
    int** A21 = allocate_matrix(k); int** A22 = allocate_matrix(k);
    int** B11 = allocate_matrix(k); int** B12 = allocate_matrix(k);
    int** B21 = allocate_matrix(k); int** B22 = allocate_matrix(k);
    int** C11 = allocate_matrix(k); int** C12 = allocate_matrix(k);
    int** C21 = allocate_matrix(k); int** C22 = allocate_matrix(k);

    // Temporary matrices
    int** M1 = allocate_matrix(k); int** M2 = allocate_matrix(k);
    int** M3 = allocate_matrix(k); int** M4 = allocate_matrix(k);
    int** M5 = allocate_matrix(k); int** M6 = allocate_matrix(k);
    int** M7 = allocate_matrix(k);
    int** T1 = allocate_matrix(k); int** T2 = allocate_matrix(k);

    // Split matrices into submatrices
    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_matrix(A11, A22, T1, k);  add_matrix(B11, B22, T2, k);  strassen(T1, T2, M1, k);
    add_matrix(A21, A22, T1, k);  strassen(T1, B11, M2, k);
    sub_matrix(B12, B22, T2, k);  strassen(A11, T2, M3, k);
    sub_matrix(B21, B11, T2, k);  strassen(A22, T2, M4, k);
    add_matrix(A11, A12, T1, k);  strassen(T1, B22, M5, k);
    sub_matrix(A21, A11, T1, k);  add_matrix(B11, B12, T2, k);  strassen(T1, T2, M6, k);
    sub_matrix(A12, A22, T1, k);  add_matrix(B21, B22, T2, k);  strassen(T1, T2, M7, k);

    // Compute C11, C12, C21, C22
    add_matrix(M1, M4, T1, k);  sub_matrix(T1, M5, T2, k);  add_matrix(T2, M7, C11, k);
    add_matrix(M3, M5, C12, k);
    add_matrix(M2, M4, C21, k);
    sub_matrix(M1, M2, T1, k);  add_matrix(T1, M3, T2, k);  add_matrix(T2, M6, C22, k);

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

    // Free memory
    free_matrix(A11, k); free_matrix(A12, k); free_matrix(A21, k); free_matrix(A22, k);
    free_matrix(B11, k); free_matrix(B12, k); free_matrix(B21, k); free_matrix(B22, k);
    free_matrix(C11, k); free_matrix(C12, k); free_matrix(C21, k); free_matrix(C22, k);
    free_matrix(M1, k); free_matrix(M2, k); free_matrix(M3, k); free_matrix(M4, k);
    free_matrix(M5, k); free_matrix(M6, k); free_matrix(M7, k);
    free_matrix(T1, k); free_matrix(T2, k);
}

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

    FILE *fp = fopen("Data.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
```

```
    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        int** A = allocate_matrix(n);
        int** B = allocate_matrix(n);
        int** C = allocate_matrix(n);

        // Fill A and B with random numbers
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) {
                A[i][j] = rand() % 10;
                B[i][j] = rand() % 10;
            }

        clock_t start, end;

        // Standard multiplication
        start = clock();
        normal_multiply(A, B, C, n);
        end = clock();
        double time_normal = (double)(end - start) / CLOCKS_PER_SEC;

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

        printf("   Normal multiply:   %f sec\n", time_normal);
        printf("   Strassen multiply: %f sec\n", time_strassen);
        printf("---------------------------\n");

        // Write results into file (clean format)
        fprintf(fp, "%d %.6f %.6f\n", n, time_normal, time_strassen);

        free_matrix(A, n);
        free_matrix(B, n);
        free_matrix(C, n);
    }

    fclose(fp);
    printf("Results stored in Data.txt\n");

    return 0;
}
```

Output-



mm_rec.ipynb
MM.ipynb
Generate
Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
plt.show()
Matrix MUltiplication Recursive : Time vs Input Size
Time (seconds)
Input Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
0.00
0.02
0.04
0.06
0.08

Q4-
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// ------------------ Utility Functions ------------------ //

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

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

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

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

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

// ------------------ Iterative Multiplication ------------------ //
```

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

// ------------------ Divide and Conquer ------------------ //

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

    int k = n / 2;
    int **A11 = allocate_matrix(k), **A12 = allocate_matrix(k),
        **A21 = allocate_matrix(k), **A22 = allocate_matrix(k),
        **B11 = allocate_matrix(k), **B12 = allocate_matrix(k),
        **B21 = allocate_matrix(k), **B22 = allocate_matrix(k),
        **C11 = allocate_matrix(k), **C12 = allocate_matrix(k),
        **C21 = allocate_matrix(k), **C22 = allocate_matrix(k),
        **T = allocate_matrix(k);

    for (int i = 0; i < k; i++) {
        for (int j = 0; j < k; j++) {
            A11[i][j] = A[i][j];     A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j]; A22[i][j] = A[i + k][j + k];
            B11[i][j] = B[i][j];     B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j]; B22[i][j] = B[i + k][j + k];
        }
    }

    divide_and_conquer_mult(A11, B11, C11, k);
    divide_and_conquer_mult(A12, B21, T, k);
    add_matrix(C11, T, C11, k);

    divide_and_conquer_mult(A11, B12, C12, k);
    divide_and_conquer_mult(A12, B22, T, k);
```

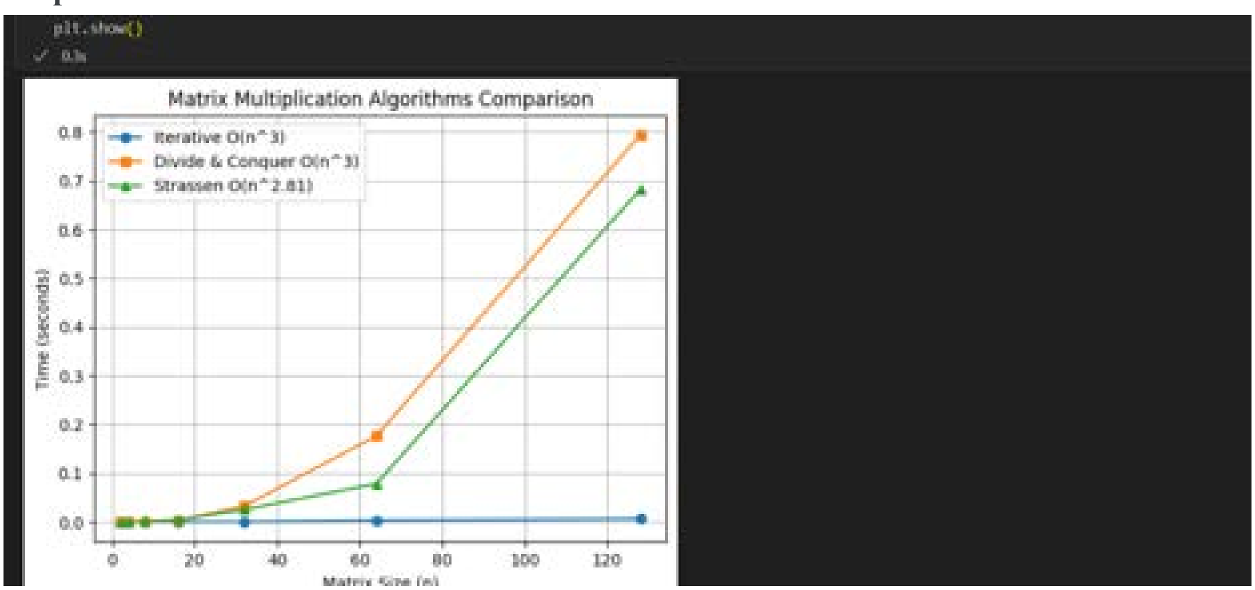
```
    add_matrix(C12, T, C12, k);

    divide_and_conquer_mult(A21, B11, C21, k);
    divide_and_conquer_mult(A22, B21, T, k);
    add_matrix(C21, T, C21, k);

    divide_and_conquer_mult(A21, B12, C22, k);
    divide_and_conquer_mult(A22, B22, T, k);
    add_matrix(C22, T, C22, k);

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

    free_matrix(A11, k); free_matrix(A12, k); free_matrix(A21, k); free_matrix(A22, k);
    free_matrix(B11, k); free_matrix(B12, k); free_matrix(B21, k); free_matrix(B22, k);
    free_matrix(C11, k); free_matrix(C12, k); free_matrix(C21, k); free_matrix(C22, k);
    free_matrix(T, k);
}

// ------------------ Strassen ------------------ //

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

    int k = n / 2;
    int **A11 = allocate_matrix(k), **A12 = allocate_matrix(k),
        **A21 = allocate_matrix(k), **A22 = allocate_matrix(k),
        **B11 = allocate_matrix(k), **B12 = allocate_matrix(k),
        **B21 = allocate_matrix(k), **B22 = allocate_matrix(k);

    for (int i = 0; i < k; i++) {
```

```
        for (int j = 0; j < k; j++) {
            A11[i][j] = A[i][j];     A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j]; A22[i][j] = A[i + k][j + k];
            B11[i][j] = B[i][j];     B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j]; B22[i][j] = B[i + k][j + k];
        }
    }

    int **M1 = allocate_matrix(k), **M2 = allocate_matrix(k), **M3 = allocate_matrix(k),
        **M4 = allocate_matrix(k), **M5 = allocate_matrix(k), **M6 = allocate_matrix(k),
        **M7 = allocate_matrix(k);
    int **T1 = allocate_matrix(k), **T2 = allocate_matrix(k);

    add_matrix(A11, A22, T1, k); add_matrix(B11, B22, T2, k); strassen_mult(T1, T2, M1,
k);
    add_matrix(A21, A22, T1, k); strassen_mult(T1, B11, M2, k);
    sub_matrix(B12, B22, T1, k); strassen_mult(A11, T1, M3, k);
    sub_matrix(B21, B11, T1, k); strassen_mult(A22, T1, M4, k);
    add_matrix(A11, A12, T1, k); strassen_mult(T1, B22, M5, k);
    sub_matrix(A21, A11, T1, k); add_matrix(B11, B12, T2, k); strassen_mult(T1, T2, M6,
k);
    sub_matrix(A12, A22, T1, k); add_matrix(B21, B22, T2, k); strassen_mult(T1, T2, M7,
k);

    int **C11 = allocate_matrix(k), **C12 = allocate_matrix(k),
        **C21 = allocate_matrix(k), **C22 = allocate_matrix(k);

    add_matrix(M1, M4, T1, k); sub_matrix(T1, M5, T2, k); add_matrix(T2, M7, C11, k);
    add_matrix(M3, M5, C12, k);
    add_matrix(M2, M4, C21, k);
    sub_matrix(M1, M2, T1, k); add_matrix(T1, M3, T2, k); add_matrix(T2, M6, C22, k);

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

```
    free_matrix(A11, k); free_matrix(A12, k); free_matrix(A21, k); free_matrix(A22, k);
    free_matrix(B11, k); free_matrix(B12, k); free_matrix(B21, k); free_matrix(B22, k);
    free_matrix(M1, k); free_matrix(M2, k); free_matrix(M3, k);
    free_matrix(M4, k); free_matrix(M5, k); free_matrix(M6, k); free_matrix(M7, k);
    free_matrix(T1, k); free_matrix(T2, k);
    free_matrix(C11, k); free_matrix(C12, k); free_matrix(C21, k); free_matrix(C22, k);
}

// ------------------ Main ------------------ //

int main() {
    srand(time(NULL));

    FILE *fp = fopen("matrix_times.csv", "w");
    fprintf(fp, "Size,Iterative,DivideAndConquer,Strassen\n");

    for (int exp = 1; exp <= 7; exp++) { // up to 128x128
        int n = 1 << exp;
        printf("\nMatrix Size: %d x %d\n", n, n);

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

        fill_random(A, n);
        fill_random(B, n);

        clock_t start, end;
        double t1, t2, t3;

        start = clock();
        iterative_mult(A, B, C, n);
        end = clock();
        t1 = (double)(end - start) / CLOCKS_PER_SEC;
        printf("Iterative: %.6f sec\n", t1);

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

```
    t2 = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Divide & Conquer: %.6f sec\n", t2);

    start = clock();
    strassen_mult(A, B, C, n);
    end = clock();
    t3 = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Strassen: %.6f sec\n", t3);

    fprintf(fp, "%d,%.6f,%.6f,%.6f\n", n, t1, t2, t3);

    free_matrix(A, n); free_matrix(B, n); free_matrix(C, n);
  }

  fclose(fp);
  printf("\nResults written to matrix_times.csv\n");

  return 0;
}
```

Output-

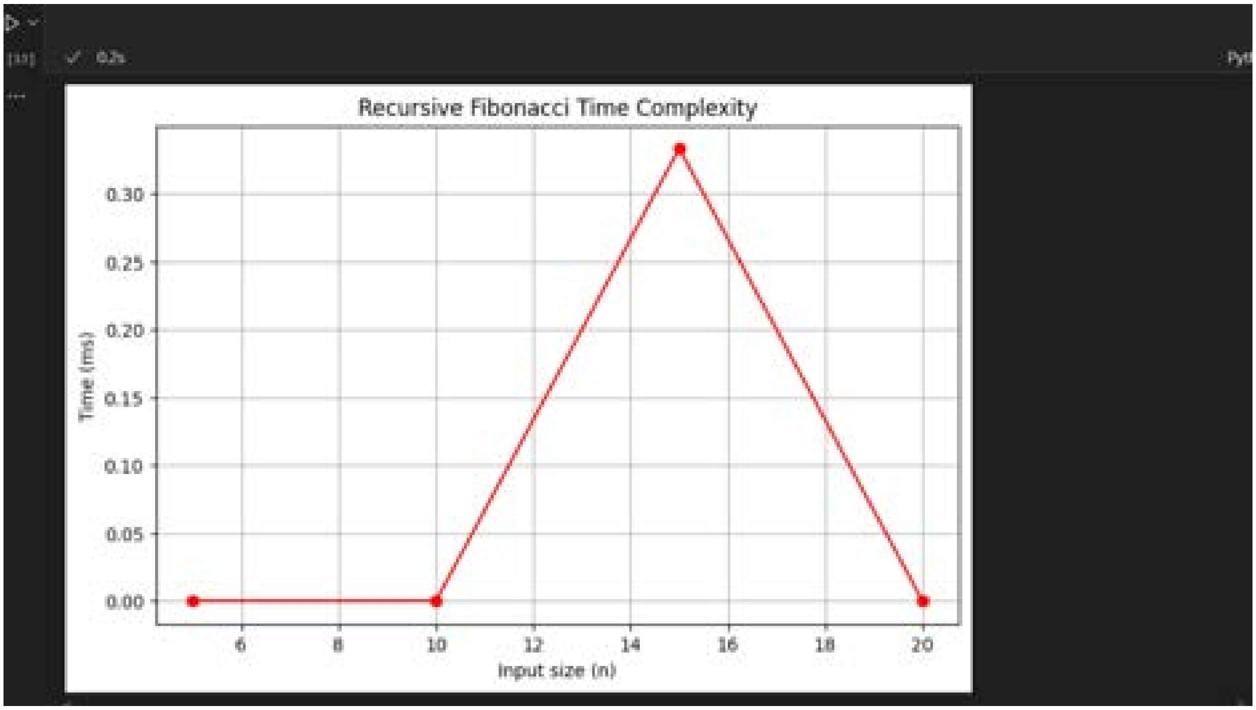


4a. Recursive version

Code-

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

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

int main() {
    // Use smaller n for recursive to keep times reasonable
    int n_values[] = {5, 10, 15, 20};
    int num_values = sizeof(n_values)/sizeof(n_values[0]);

    FILE *fp = fopen("recursive.csv", "w");
    fprintf(fp, "n,time_ms\n");

    for (int i = 0; i < num_values; i++) {
        int n = n_values[i];
        int repeats = 3; // very small repeat because recursive is slow

        clock_t start = clock();
        for (int j = 0; j < repeats; j++) {
            fib_recursive(n);
        }
        clock_t end = clock();

        double time_ms = ((double)(end - start)/CLOCKS_PER_SEC)*1000.0 / repeats;
        fprintf(fp, "%d,%.5f\n", n, time_ms);
    }

    fclose(fp);
    return 0;
}
```

Output-

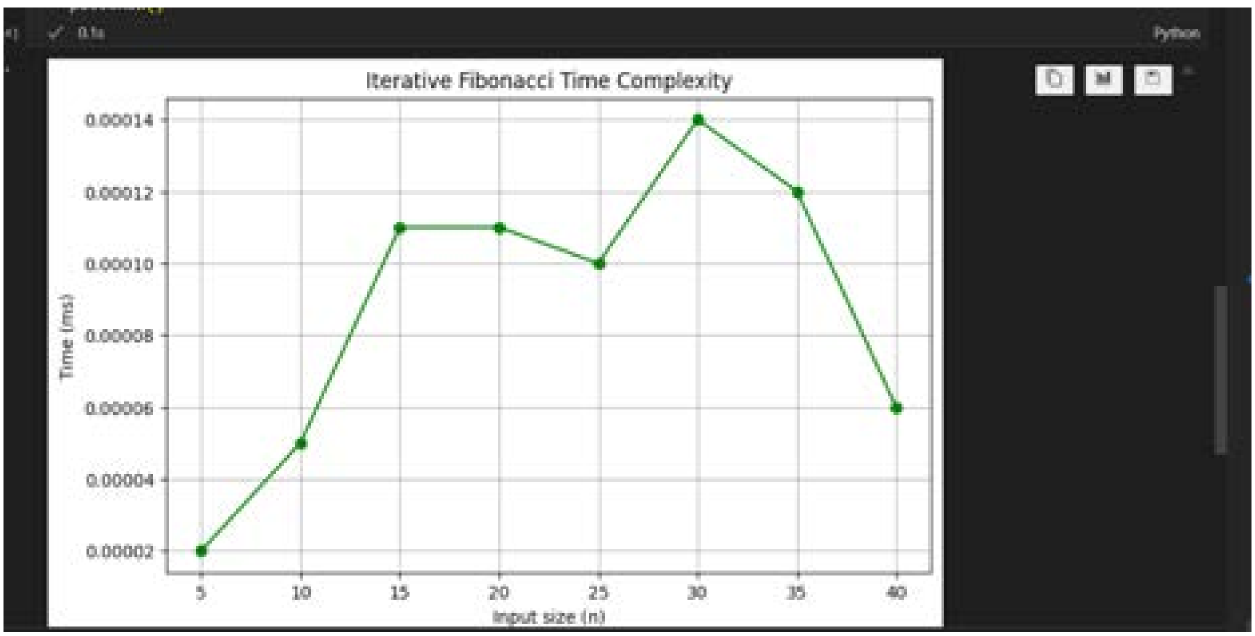


4b Iterative version
Code-

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

long long fib_iterative(int n) {
    if (n <= 1) return n;
    long long prev2 = 0, prev1 = 1, curr = 0;
    for (int i = 2; i <= n; i++) {
        curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return curr;
}

int main() {
    int n_values[] = {5, 10, 15, 20, 25, 30, 35, 40};
    int num_values = sizeof(n_values) / sizeof(n_values[0]);
    FILE *fp = fopen("iterative.csv", "w");
    fprintf(fp, "n,time_ms\n");

    for (int i = 0; i < num_values; i++) {
        int n = n_values[i];
```

```
        int repeats = 100000; // fast, so high repeats
        clock_t start = clock();
        for (int j = 0; j < repeats; j++) {
            fib_iterative(n);
        }
        clock_t end = clock();
        double time_ms = ((double)(end - start) / CLOCKS_PER_SEC) * 1000.0 / repeats;
        fprintf(fp, "%d,%.8f\n", n, time_ms);
    }

    fclose(fp);
    return 0;
}
```

Output-



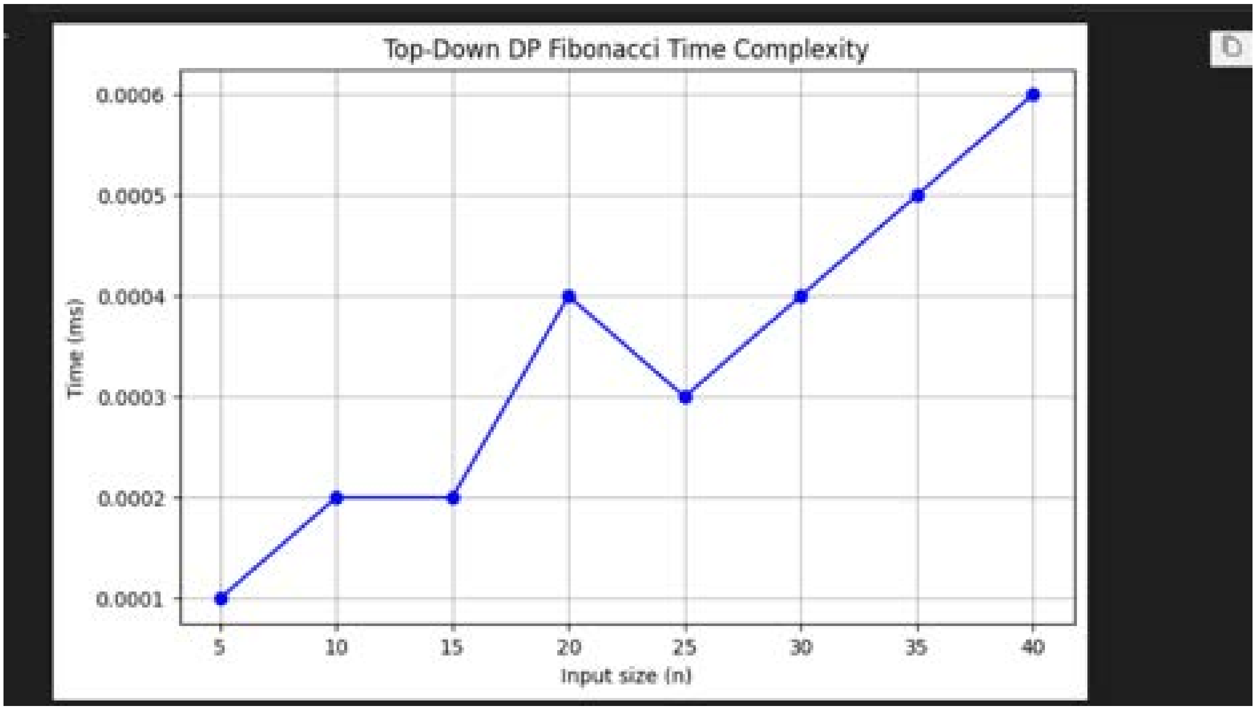
4c Dynamic Programming- Top Down Approach

Code-

```
#include <stdio.h>
#include <time.h>
#define MAX_N 100

long long memo[MAX_N];

long long fib_topdown(int n) {
    if (n <= 1) return n;
```

```
    if (memo[n] != -1) return memo[n];
    memo[n] = fib_topdown(n-1) + fib_topdown(n-2);
    return memo[n];
}

int main() {
    int n_values[] = {5, 10, 15, 20, 25, 30, 35, 40};
    int num_values = sizeof(n_values)/sizeof(n_values[0]);

    FILE *fp = fopen("topdown.csv", "w");
    fprintf(fp, "n,time_ms\n");

    for (int i = 0; i < num_values; i++) {
        int n = n_values[i];
        int repeats = 10000;

        clock_t start = clock();
        for (int j = 0; j < repeats; j++) {
            for (int k = 0; k <= n; k++) memo[k] = -1;
            fib_topdown(n);
        }
        clock_t end = clock();
        double time_ms = ((double)(end - start)/CLOCKS_PER_SEC)*1000.0 / repeats;
        fprintf(fp, "%d,%.8f\n", n, time_ms);
    }

    fclose(fp);
    return 0;
}
```

Output-

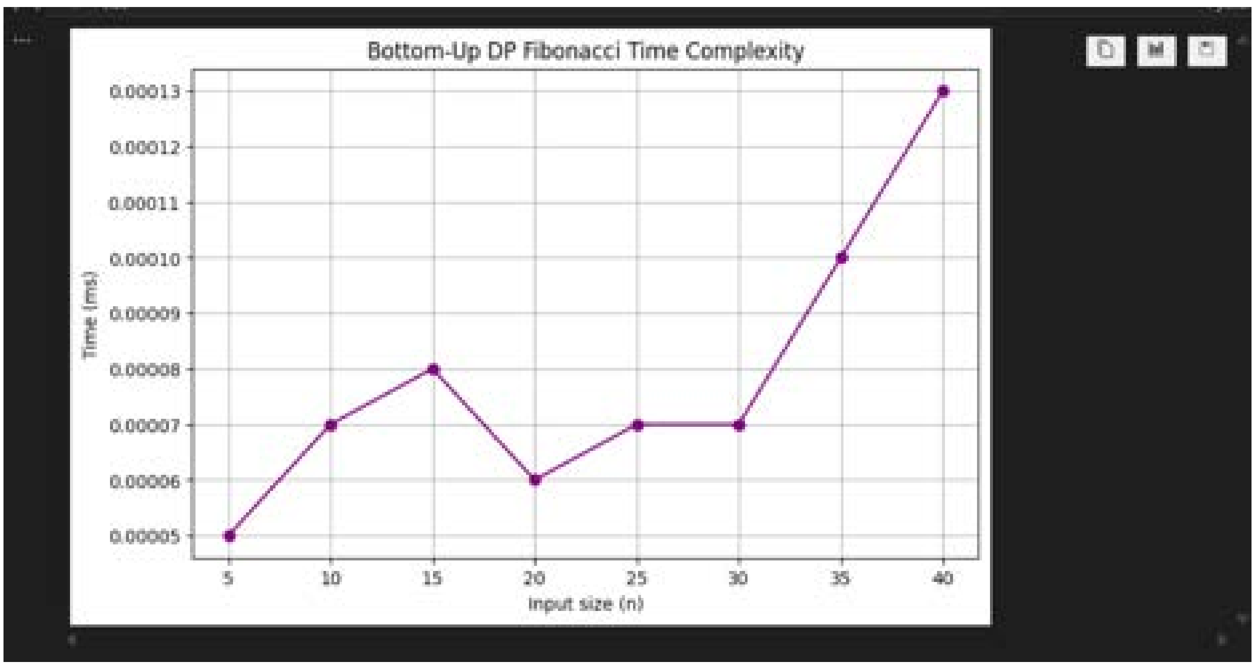


4d Dynamic Programming- Bottom Up Approach
Code-

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

long long fib_bottomup(int n) {
    if (n <= 1) return n;
    long long dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}

int main() {
    int n_values[] = {5, 10, 15, 20, 25, 30, 35, 40};
    int num_values = sizeof(n_values)/sizeof(n_values[0]);

    FILE *fp = fopen("bottomup.csv", "w");
    fprintf(fp, "n,time_ms\n");

    for (int i = 0; i < num_values; i++) {
```

```
        int n = n_values[i];
        int repeats = 100000; // repeat many times
        clock_t start = clock();
        for (int j = 0; j < repeats; j++) {
            fib_bottomup(n);
        }
        clock_t end = clock();
        double time_ms = ((double)(end - start) / CLOCKS_PER_SEC) * 1000.0 / repeats;
        fprintf(fp, "%d,%.8f\n", n, time_ms);
    }

    fclose(fp);
    return 0;
}
```

Output-



1-. Fractional knapsack Problem

Code-

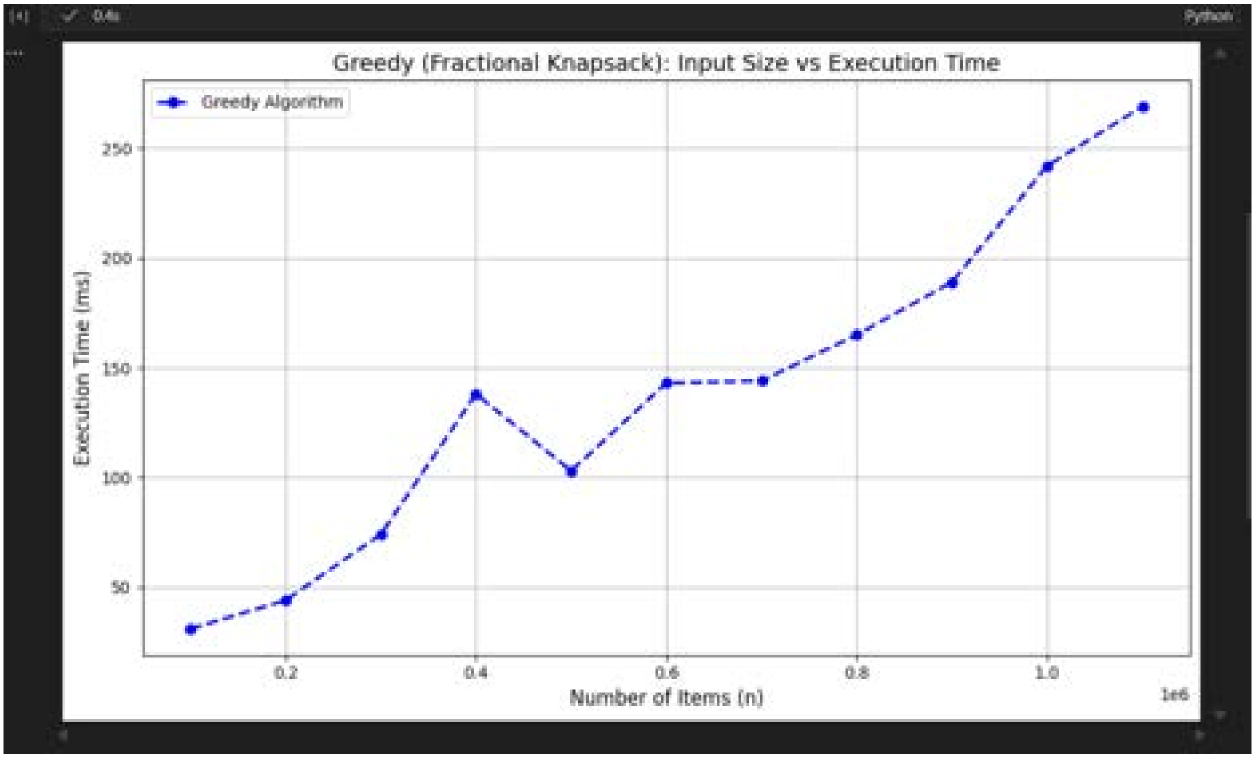
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct {
    int weight;
    int value;
    float ratio;
} Item;

int cmp(const void *a, const void *b) {
    Item *i1 = (Item*)a;
    Item *i2 = (Item*)b;
    if (i2->ratio > i1->ratio) return 1;
    else if (i2->ratio < i1->ratio) return -1;
    else return 0;
}

float fractionalKnapsack(int W, Item items[], int n) {
    qsort(items, n, sizeof(Item), cmp);
    int curWeight = 0;
    float finalValue = 0.0;

    for (int i = 0; i < n; i++) {
        if (curWeight + items[i].weight <= W) {
            curWeight += items[i].weight;
            finalValue += items[i].value;
        } else {
            int remain = W - curWeight;
            finalValue += items[i].value * ((float)remain / items[i].weight);
            break;
        }
    }
```

```
    return finalValue;
}

int main() {
    int n_values[11] =
{100000,200000,300000,400000,500000,600000,700000,800000,900000,100000
0,1100000};
    FILE *fp = fopen("fractional_results.txt", "w");
    if (!fp) {
        printf("Error opening file!\n");
        return 1;
    }

    srand(time(NULL));
    int capacity = 5000;

    fprintf(fp, "n\tGreedyTime(ms)\n");

    for (int t = 0; t < 11; t++) {
        int n = n_values[t];
        Item *items = malloc(n * sizeof(Item));

        for (int i = 0; i < n; i++) {
            items[i].value = rand() % 100 + 1;
            items[i].weight = rand() % 50 + 1;
            items[i].ratio = (float)items[i].value / items[i].weight;
        }

        clock_t start = clock();
        fractionalKnapsack(capacity, items, n);
        clock_t end = clock();

        double timeTaken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;

        printf("n=%d  Greedy=%.2f ms\n", n, timeTaken);
        fprintf(fp, "%d\t%.2f\n", n, timeTaken);
```

```
        free(items);
    }

    fclose(fp);
    printf("\nFractional knapsack results saved in fractional_results.txt\n");
    return 0;
}
```

Output-

```
souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/Knapsack
$ gcc fractional.c

souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/Knapsack
$ ./a.exe
n=100000  Greedy=31.00 ms
n=200000  Greedy=44.00 ms
n=300000  Greedy=74.00 ms
n=400000  Greedy=138.00 ms
n=500000  Greedy=101.00 ms
n=600000  Greedy=143.00 ms
n=700000  Greedy=144.00 ms
n=800000  Greedy=165.00 ms
n=900000  Greedy=189.00 ms
n=1000000  Greedy=242.00 ms
n=1100000  Greedy=269.00 ms

Fractional knapsack results saved in fractional_results.txt
```



2-0/1 Knapsack Problem

Code-

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

typedef struct {
    int weight;
    int value;
```

```
} Item;

int zeroOneKnapsack(int W, Item items[], int n) {
    int **dp = (int**)malloc((n+1) * sizeof(int*));
    for(int i=0; i<=n; i++)
        dp[i] = (int*)malloc((W+1) * sizeof(int));

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (items[i-1].weight <= w)
                dp[i][w] = (items[i-1].value + dp[i-1][w-items[i-1].weight] > dp[i-1][w])
                           ? items[i-1].value + dp[i-1][w-items[i-1].weight]
                           : dp[i-1][w];
            else
                dp[i][w] = dp[i-1][w];
        }
    }

    int result = dp[n][W];

    for(int i=0; i<=n; i++) free(dp[i]);
    free(dp);

    return result;
}

int main() {
    int n_values[11] =
{1000,2000,3000,4000,5000,6000,7000,8000,9000,10000,11000};
    FILE *fp = fopen("zeroone_results.txt", "w");
    if (!fp) {
        printf("Error opening file!\n");
        return 1;
    }
```

```
    srand(time(NULL));
    fprintf(fp, "n\tW\tDPTime(ms)\n");

    for (int t = 0; t < 11; t++) {
        int n = n_values[t];

        int capacity = n / 10;
        if (capacity < 1) capacity = 1;

        Item *items = malloc(n * sizeof(Item));

        for (int i = 0; i < n; i++) {
            items[i].value = rand() % 100 + 1;
            items[i].weight = rand() % capacity + 1;
        }

        clock_t start = clock();
        zeroOneKnapsack(capacity, items, n);
        clock_t end = clock();

        double timeTaken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;

        printf("n=%d W=%d  DP=%.2f ms\n", n, capacity, timeTaken);
        fprintf(fp, "%d\t%d\t%.2f\n", n, capacity, timeTaken);

        free(items);
    }

    fclose(fp);
    printf("\n0/1 knapsack results saved in zeroone_results.txt\n");
    return 0;
}
```

Output-

```
souri@Mysterium MINGW64 ~/OneDrive/Desktop/Sourish/College/ADA/Knapsack
$ ./a.exe
n=1000 W=100  DP=2.00 ms
n=2000 W=200  DP=2.00 ms
n=3000 W=300  DP=5.00 ms
n=4000 W=400  DP=14.00 ms
n=5000 W=500  DP=25.00 ms
n=6000 W=600  DP=28.00 ms
n=7000 W=700  DP=37.00 ms
n=8000 W=800  DP=56.00 ms
n=9000 W=900  DP=60.00 ms
n=10000 W=1000  DP=80.00 ms
n=11000 W=1100  DP=98.00 ms

0/1 knapsack results saved in zeroone_results.txt
```