**Practical – 1(a)**

**Aim –**

1 (a) 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 :-**

 FUNCTION GENERATE-RANDOM-NUMBERS(A, n)

    for i ← 0 to n-1

        do A[i] ← random(0..99999)

FUNCTION LINEAR-SEARCH(A, n, key)

    for i ← 0 to n-1

        do if A[i] = key

              then return i

    return -1

MAIN()

    input n

    if n ≤ 5000

        then print "Enter a number greater than 5000"

    allocate array A[0..n-1]

    seed random generator

    call GENERATE-RANDOM-NUMBERS(A, n)

    key ← A[random(0..n-1)]

    start ← current clock time

    for i ← 1 to 10000

        do LINEAR-SEARCH(A, n, key)

    end ← current clock time

    time\_taken ← (end - start) / CLOCKS\_PER\_SEC / 100

    print "Time taken for linear search in n elements = time\_taken seconds"

    free memory

**CODE:-**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void random\_array(int \*arr,int n){

    for(int i=0;i<n;i++){

        arr[i]=rand()%100;

    }

}

int linearsearch(int \*arr, int n, int key)

{

    for (int i = 0; i < n; i++)

    {

        if (arr[i] == key)

        {

            return i;

        }

    }

    return -1;

}

int main()

{

    int n, key, pos;

    printf("Enter the number of elements you want in an array");

    scanf("%d", &n);

    if(n<10){

        return -1;

    }

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

    if(arr==NULL){

        printf("Memory is not allocated to array");

        return -1;

    }

    printf("Enter the element you want to search: ");

    scanf("%d", &key);

    random\_array(arr,n);

    clock\_t start = clock();

    for(int i=0;i<10000;i++){

        pos = linearsearch(arr, n, key);

    }

    clock\_t end = clock();

    if (pos != -1)

    {

        printf("Element is found at %d th index\n", pos);

    }

    else

    {

        printf("Element not found.\n");

    }

    double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC/100.0;

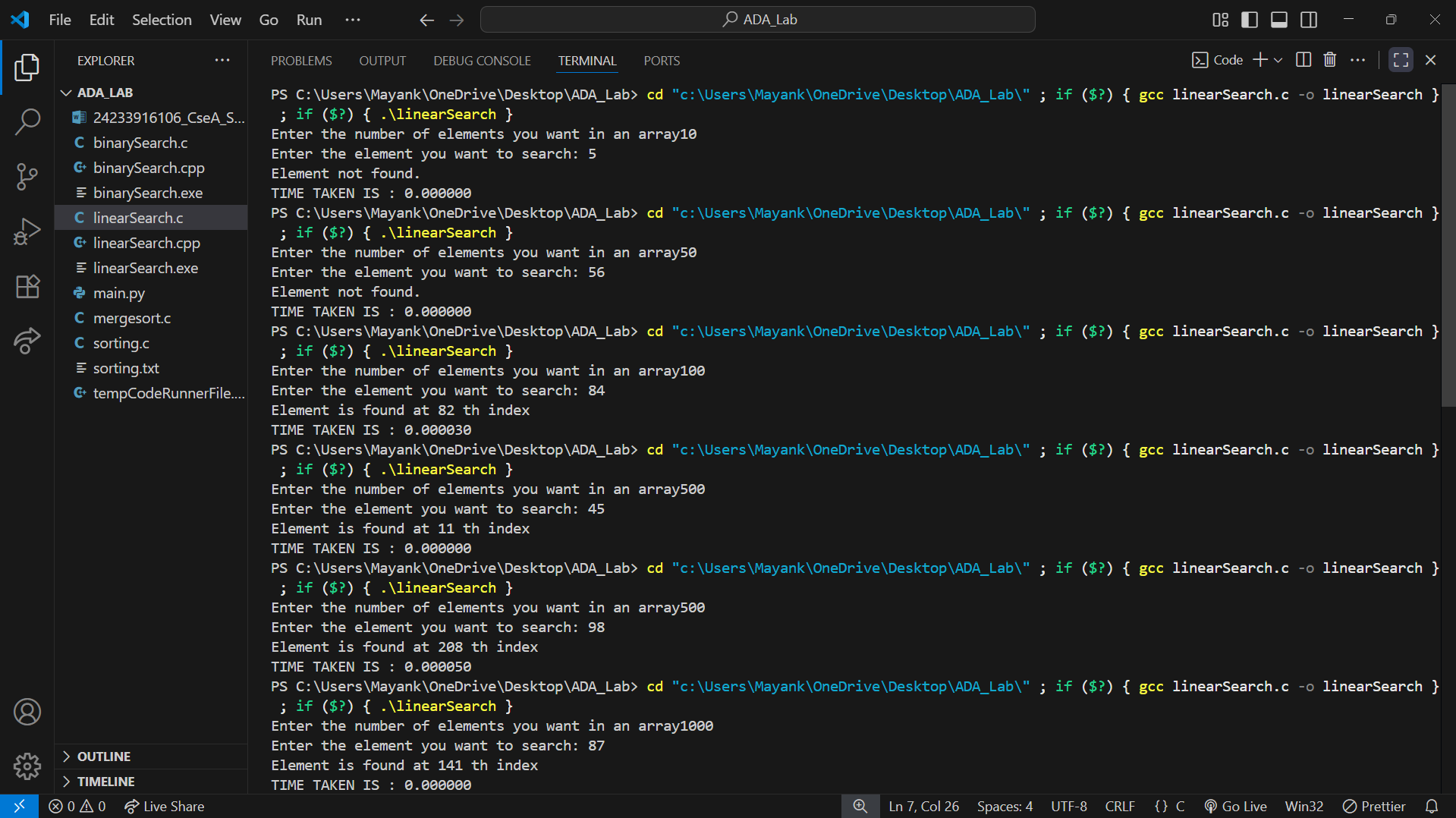
    printf("TIME TAKEN IS : %f", time\_taken);

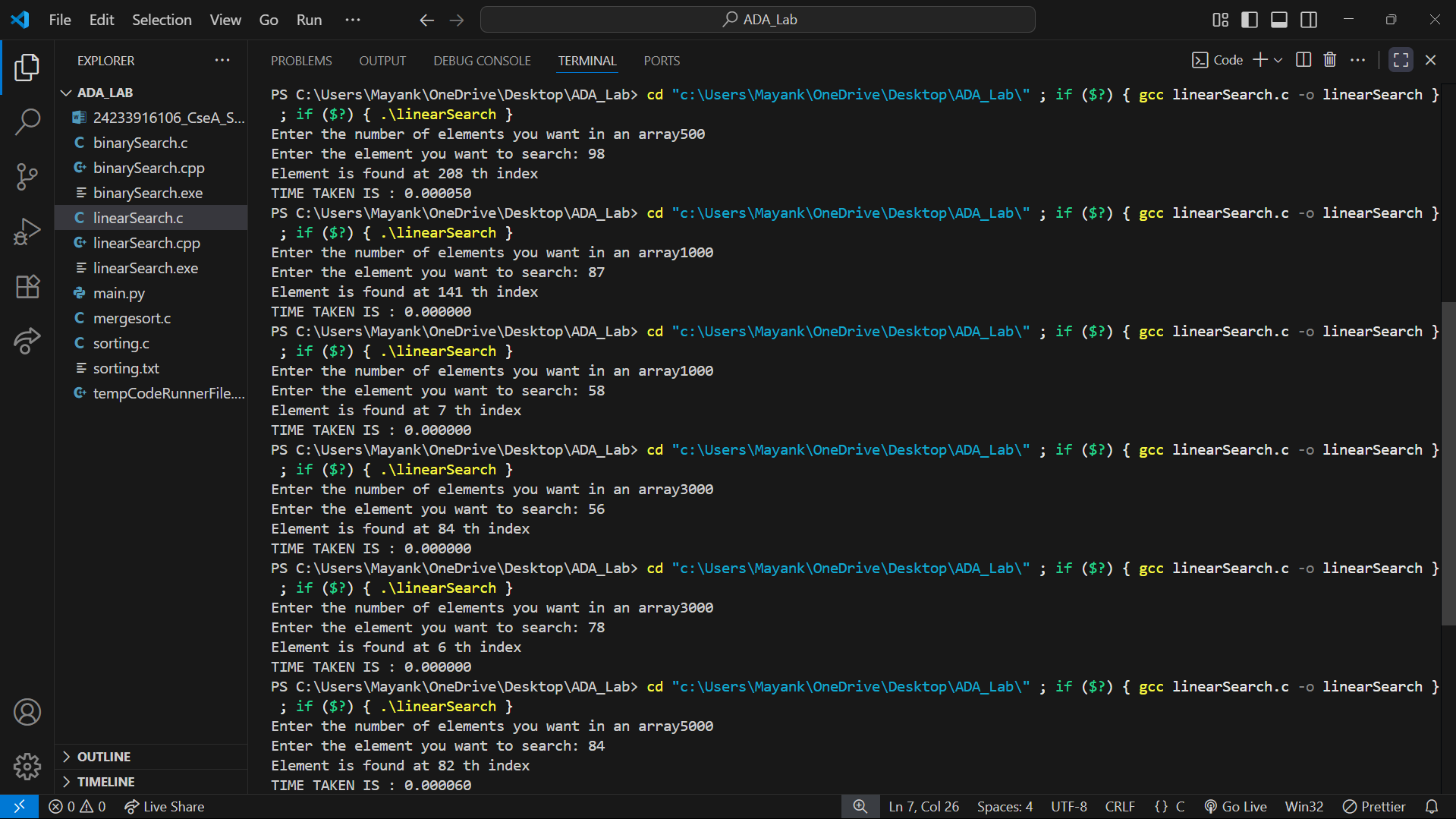
    free(arr);

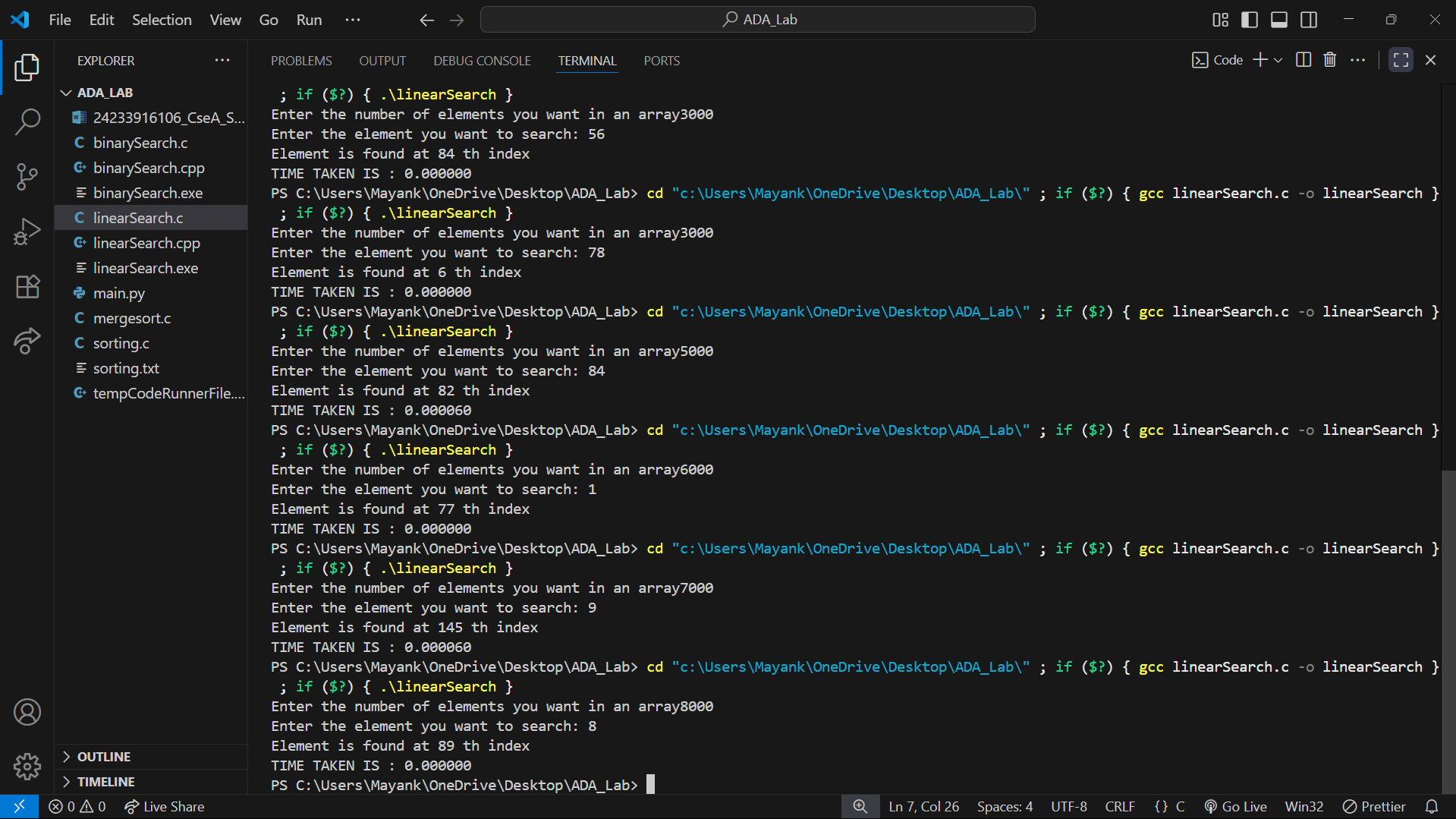
    return 0;

}

**OUTPUT:-**

****

****

****

**Python Code :-**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 8000]

time\_taken = [0.0, 0.0, 0.00003, 0.00005, 0.0, 0.0 , 0.00006, 0.0 , 0.0 ]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='--')

# Labels and Title

plt.xlabel("Input size")

plt.ylabel("Time Taken")

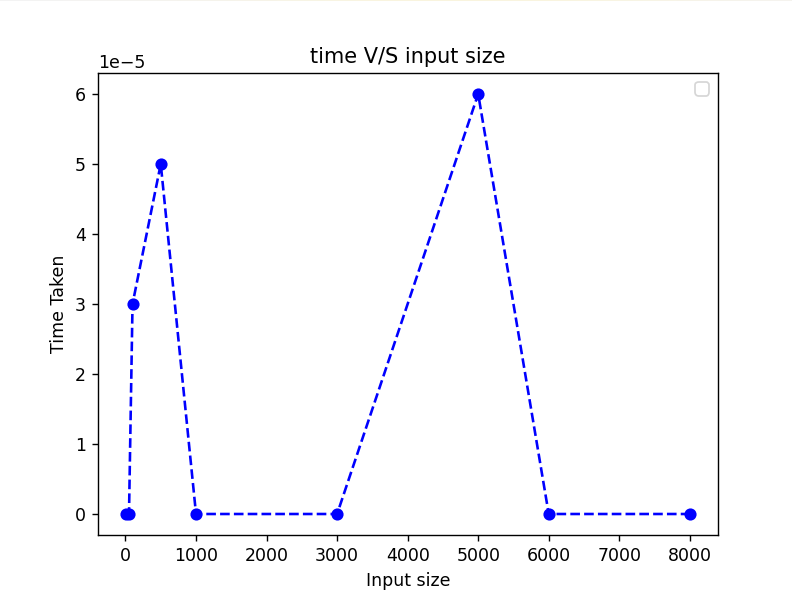
plt.title("time V/S input size")

plt.legend()

# Show the graph

plt.show()

**Graph :-**

****

**CONCLUSION:-** In this practical of linear search, I observed that the graph obtained did not match the ideal graph when the n increases.

Generally, the time complexity of linear search grows n, so the graph of growth of linear search resulting straight line.

Here, in my case ,it is zero because we take the random array so it might possible that if we enter the key is **not present in array** so it takes time zero(0) or sometime late that takes much time.

Therefore due to this variations, there is a difference between the graph of growth of linear search obtained and the ideal one.

**Practical – 1(b)**

**Aim –**

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 :-**

FUNCTION BubbleSort(array, size):

FOR i from 0 to size - 2:

FOR j from 0 to size - i - 2:

IF array[j] > array[j + 1]:

SWAP array[j] and array[j + 1]

FUNCTION BinarySearch(array, size, key):

SET low = 0

SET high = size - 1

WHILE low <= high:

SET mid = low + (high - low) / 2

IF array[mid] == key:

RETURN mid

ELSE IF key < array[mid]:

high = mid - 1

ELSE:

low = mid + 1

RETURN -1

FUNCTION GenerateRandomArray(array, size):

FOR i from 0 to size - 1:

array[i] = random number between 0 and 99

MAIN:

PROMPT user for number of elements (n)

READ n

ALLOCATE memory for array of size n

IF memory allocation fails:

PRINT "Memory is not allocated"

EXIT with error

CALL GenerateRandomArray(array, n)

PROMPT user for element to search (key)

READ key

CALL BubbleSort(array, n)

RECORD start time

REPEAT 10,000 times:

CALL BinarySearch(array, n, key)

STORE result in 'pos'

RECORD end time

IF pos != -1:

PRINT "Element is found at index pos"

ELSE:

PRINT "Element not found"

COMPUTE time\_taken = (end - start) / CLOCKS\_PER\_SECOND / 1000

PRINT "TIME TAKEN IS: time\_taken"

FREE allocated memory

END

**Code:-**

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

int compare(const void \*a,const void \*b){

    return ((int)a-(int)b);

}

int binary\_search(int\*arr,int n,int key){

    int low=0,high=n-1;

    while(low<=high){

        int mid =low +(high-low)/2;

        if(key==arr[mid]){

            return mid;

        }

        else if(key<arr[mid]){

            high=mid-1;

        }

        else{

            low=mid+1;

        }

    }

    return -1;

}

void random\_array(int \*arr,int n){

    for(int i=0;i<n;i++){

        arr[i]=rand()%100;

    }

}

int main()

{

    int n, key, pos;

    printf("Enter the number of elements you want in an array");

    scanf("%d", &n);

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

    if(arr==NULL){

        printf("Memory is not allocated to array");

        return -1;

    }

     random\_array(arr,n);

    printf("Enter the element you want to search");

    scanf("%d", &key);

    qsort(arr,n,sizeof(int),compare);

    clock\_t start = clock();

    for(int i=0;i<10000;i++){

          pos = binary\_search(arr, n, key);

    }

    clock\_t end = clock();

    if (pos != -1)

    {

        printf("Element is found at %d th index\n", pos);

    }

    else

    {

        printf("Element not found.\n");

    }

    double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC/1000.0;

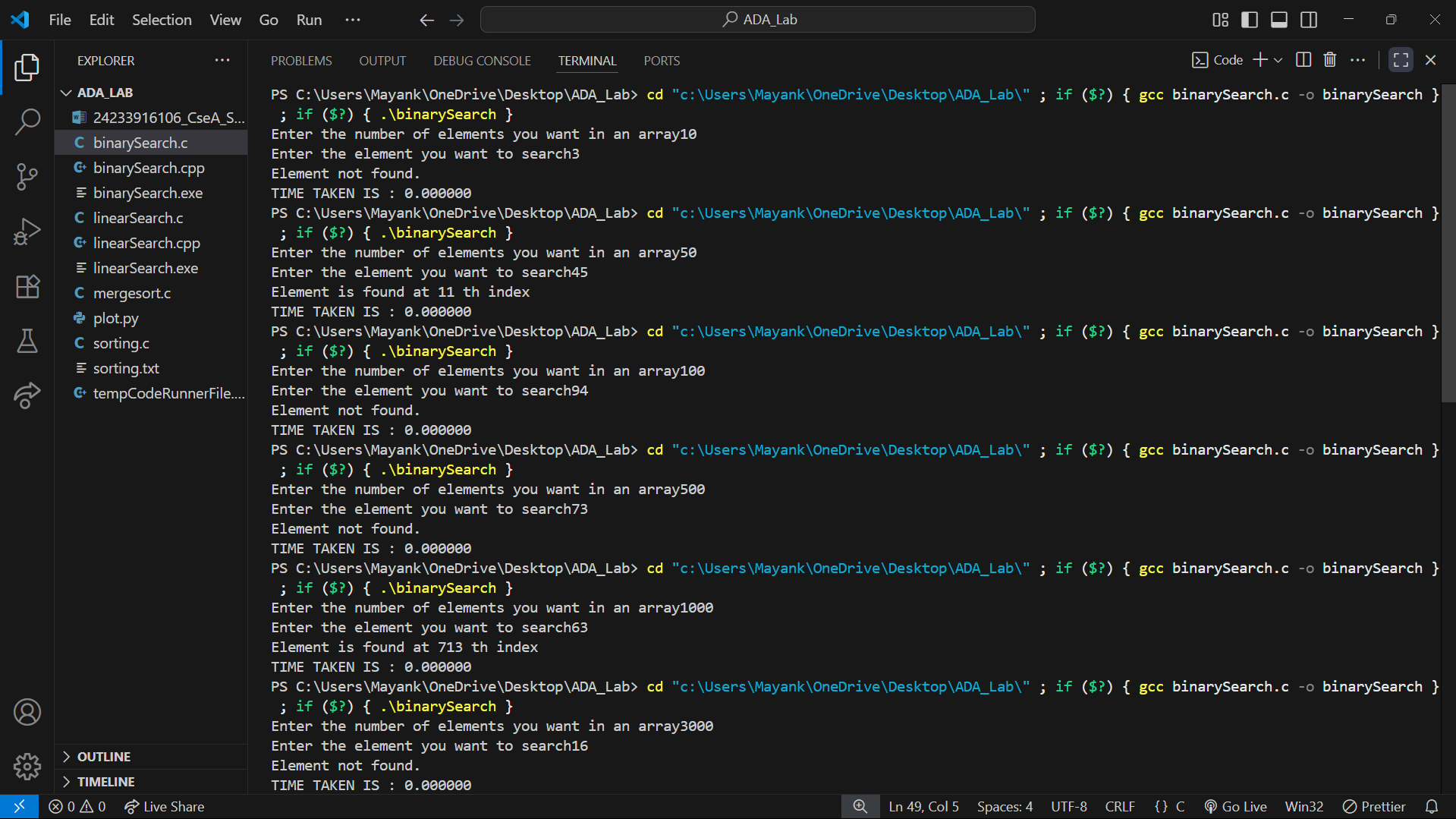
    printf("TIME TAKEN IS : %f", time\_taken);

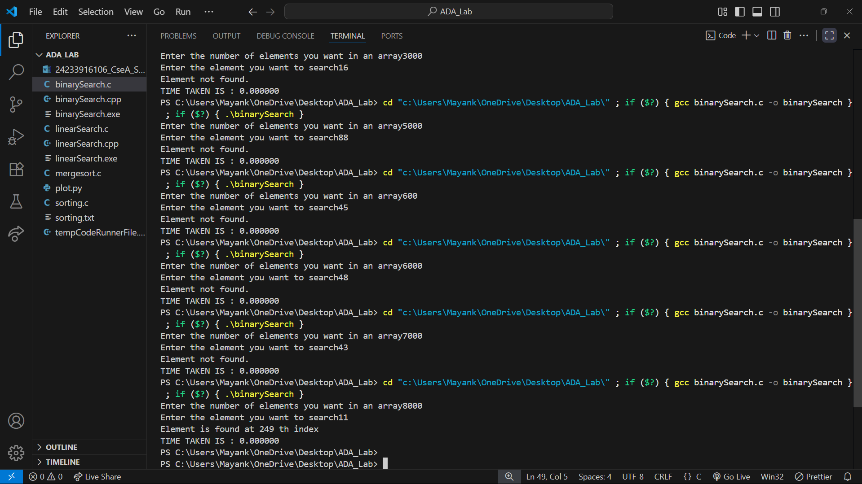
    free(arr);

    return 0;

}

**OUTPUT:-**

****



**PYTHON CODE:-**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 8000]

time\_taken = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0 , 0.0, 0.0, 0.0 ]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='--')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

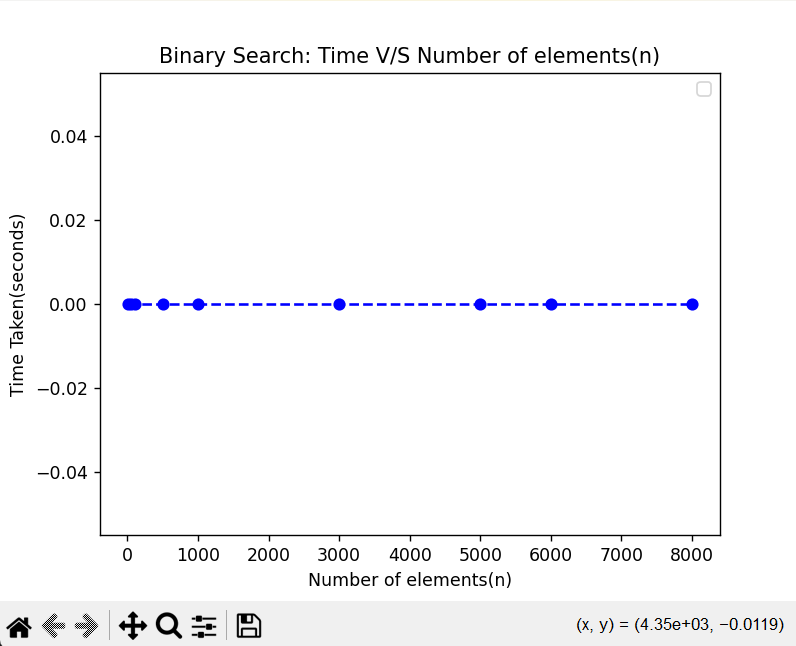
plt.title("Binary Search: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**Graph :-**



**CONCLUSION:-**Through my performed practical of binary search using a random array , I observed that the graph obtained did not match the ideal binary search growth graph when the n increases.

Generally, the time complexity of binary search grows log(n), so the graph of growth of binary search resulting in a log(n) curve.

Here, in my case ,it is zero because we take the random array so it might possible that if we enter the key sometime it occurs at early or **not present in array** so it takes less time or sometime late that takes much time.

And most important that time taken by the binary search is also depend on device i.e. operating system also.

Therefore due to this variations, there is a difference between the graph of growth of binary searche obtained and the ideal one.