**Fibonacci Series**

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. Iterative version**:

**Pseudo Code :**

This is an efficient, space-saving method. It calculates the Fibonacci numbers in a loop from the bottom up, only needing to store the last two values at any given time.

function fibonacci\_iterative(n):

if n <= 1:

return n

set first = 0

set second = 1

for i from 2 to n:

set result = first + second

set first = second

set second = result

return result

**C Code: -**

#include<stdio.h>

#include<time.h>

void fibonacci\_iterative(int n) {

    long long int a = 0, b = 1;

    long long int next\_term;

    if (n >= 1) {

        printf("%lld  ", a);

    }

    if (n >= 2) {

        printf("%lld  ", b);

    }

    for (int i = 2; i < n; i++) {

        next\_term = a + b;

        printf("%lld  ", next\_term);

        a = b;

        b = next\_term;

    }

}

int main() {

    int n;

    printf("ENTER THE NUMBER OF TERMS IN FIBONACCI SERIES :");

    scanf("%d", &n);

    clock\_t start = clock();

    fibonacci\_iterative(n);

    clock\_t end = clock();

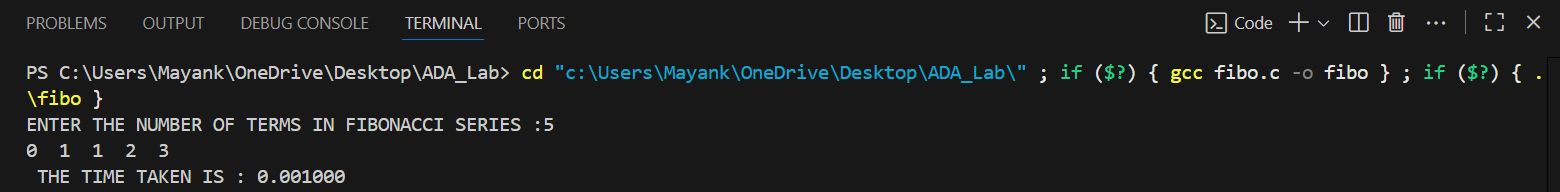
    double time\_taken = (double)(end - start) / CLOCKS\_PER\_SEC;

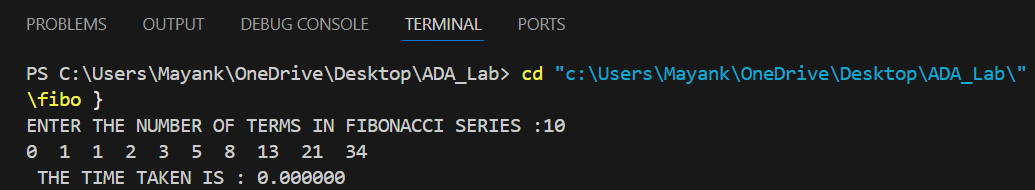
    printf("\n THE TIME TAKEN IS : %f", time\_taken);

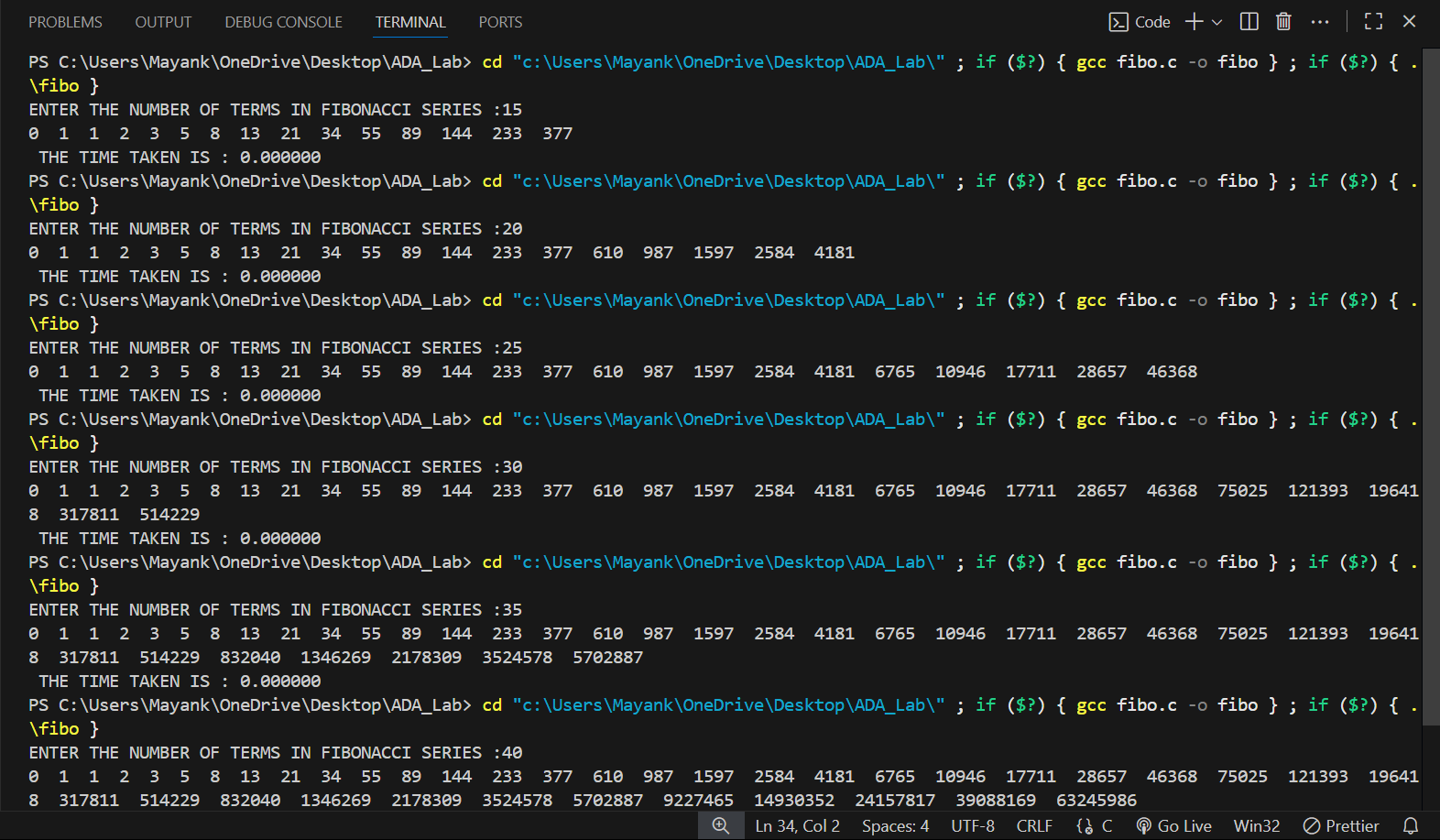
    return 0;

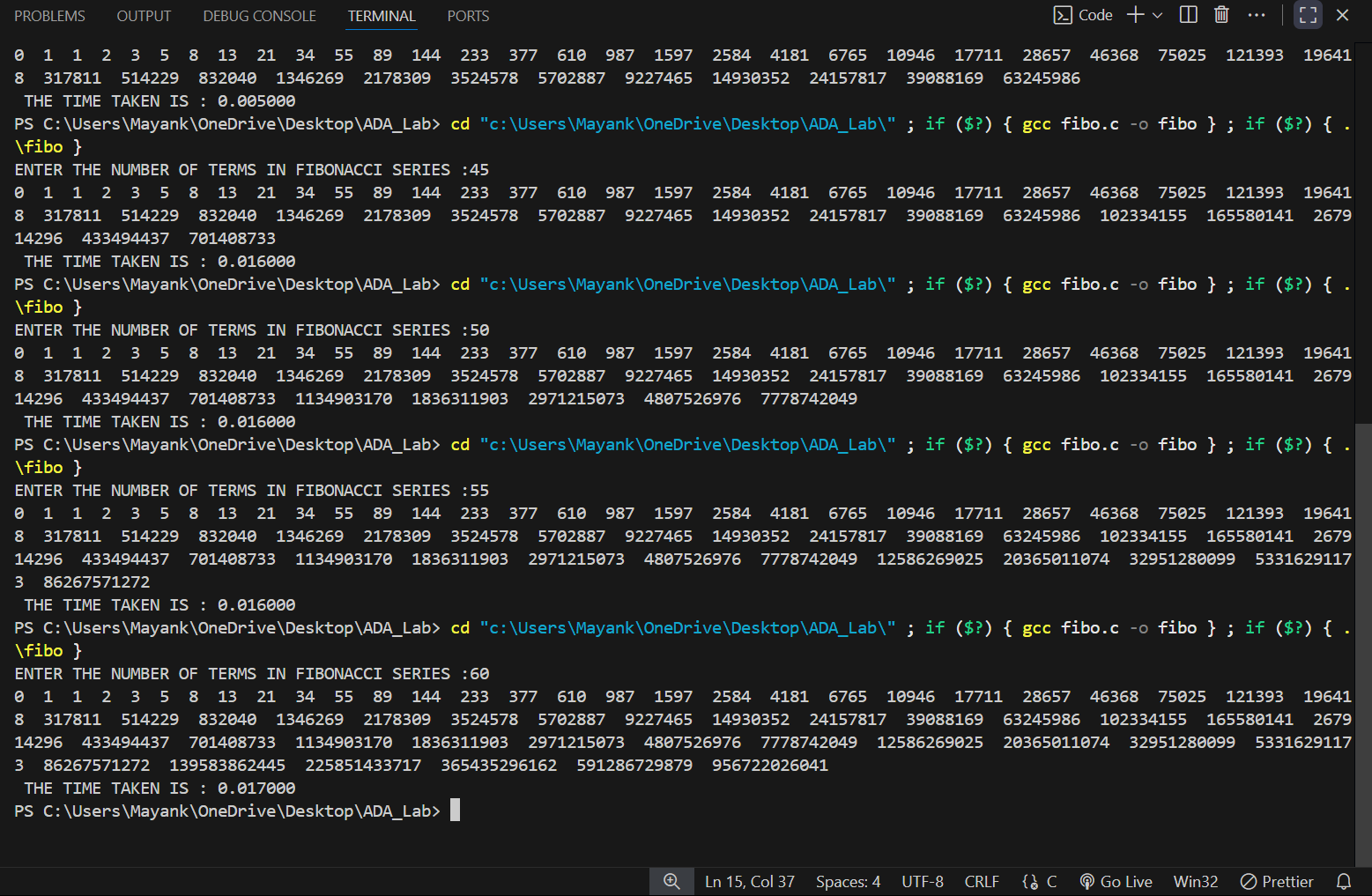
}

**Output:**









**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60]

time\_taken = [0.001, 0.0, 0.0, 0.0, 0.0, 0.0 , 0.0, 0.005, 0.016, 0.016, 0.016, 0.017 ]

# 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)")

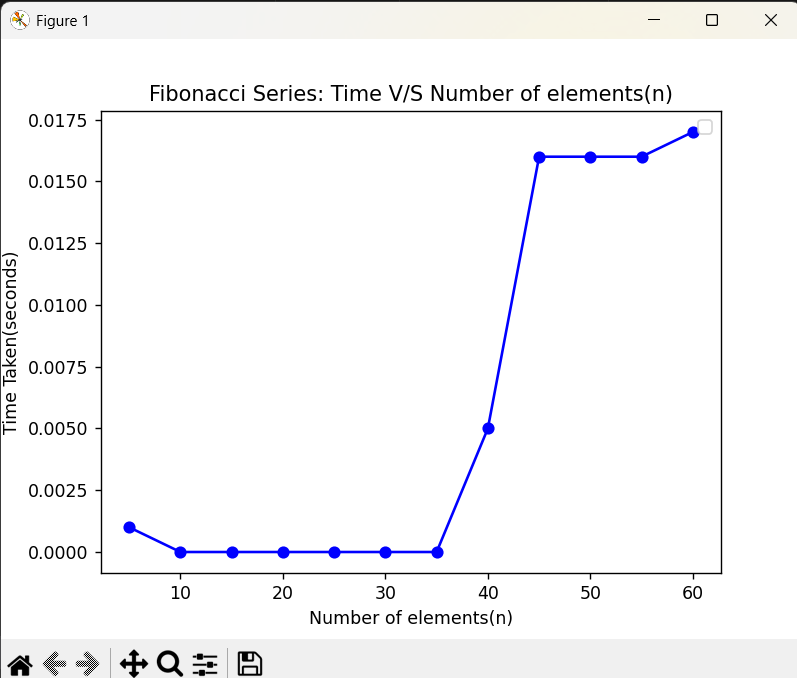
plt.title("Fibonacci Series: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**Graph:**



**4b . Recursive version**

**Pseudo Code :**

This is the most direct implementation of the Fibonacci definition, but it's very slow because it repeatedly calculates the same values.

function fibonacci\_recursive(n):

// Tracks the number of calls

increment global counter 'function\_calls'

if n <= 1:

return n

return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)

**C Code: -**

#include<stdio.h>

#include<time.h>

long long  fibonacci\_recurssive(int n){

    if(n==0){

        return 0;

    }

    if(n==1){

     return 1;

}

 return fibonacci\_recurssive(n-1)+fibonacci\_recurssive(n-2);

}

int main(){

    int n;

    printf("ENTER THE TERM FOR FIBONACCI SERIES:");

    scanf("%d",&n);

    clock\_t start =clock();

   long long  result= fibonacci\_recurssive(n);

   clock\_t end= clock();

    printf("THE NUMBER AT %d TH POSITION IS: %lld",n,result);

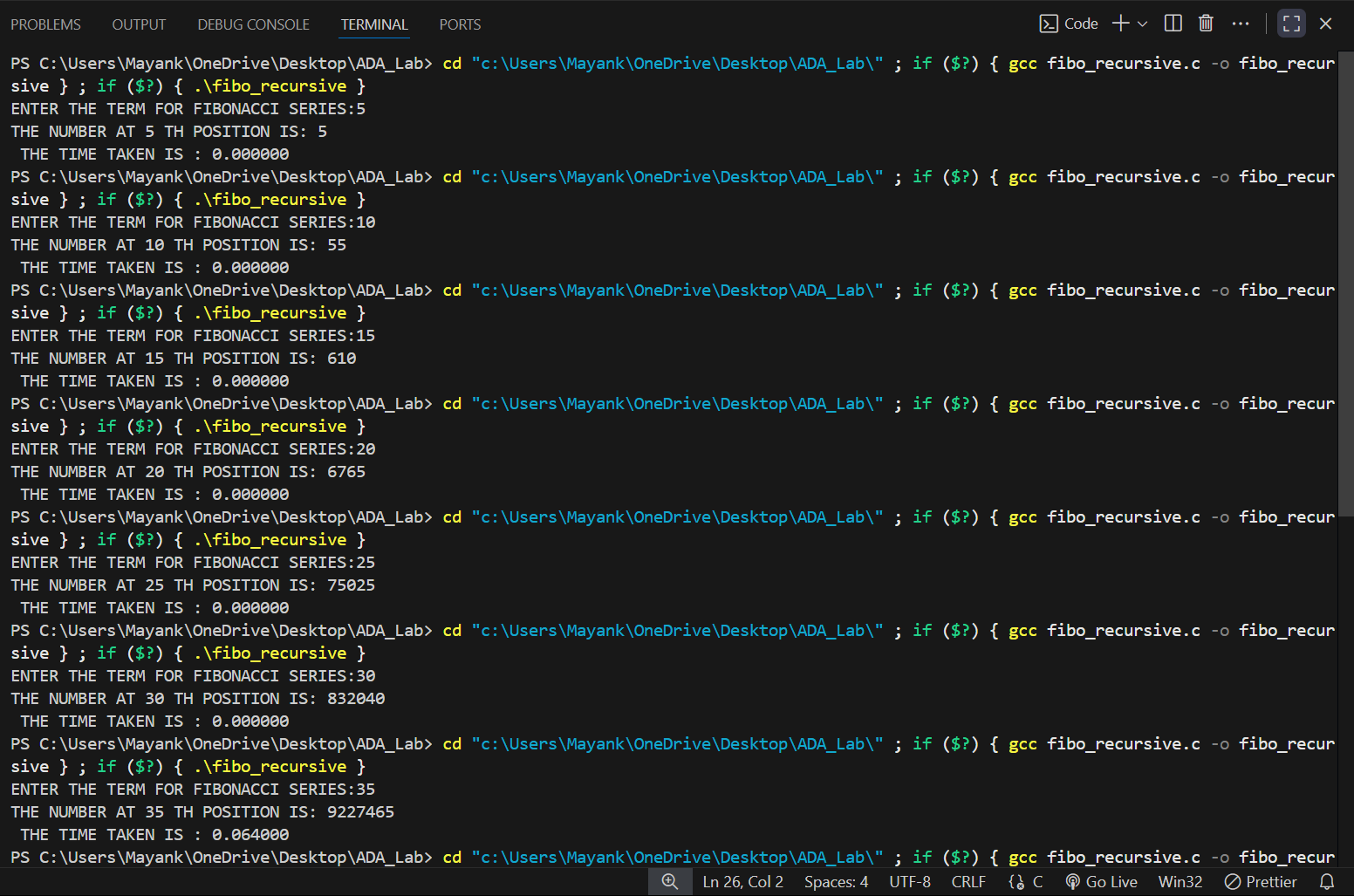
    double time\_taken =(double)(end-start)/CLOCKS\_PER\_SEC;

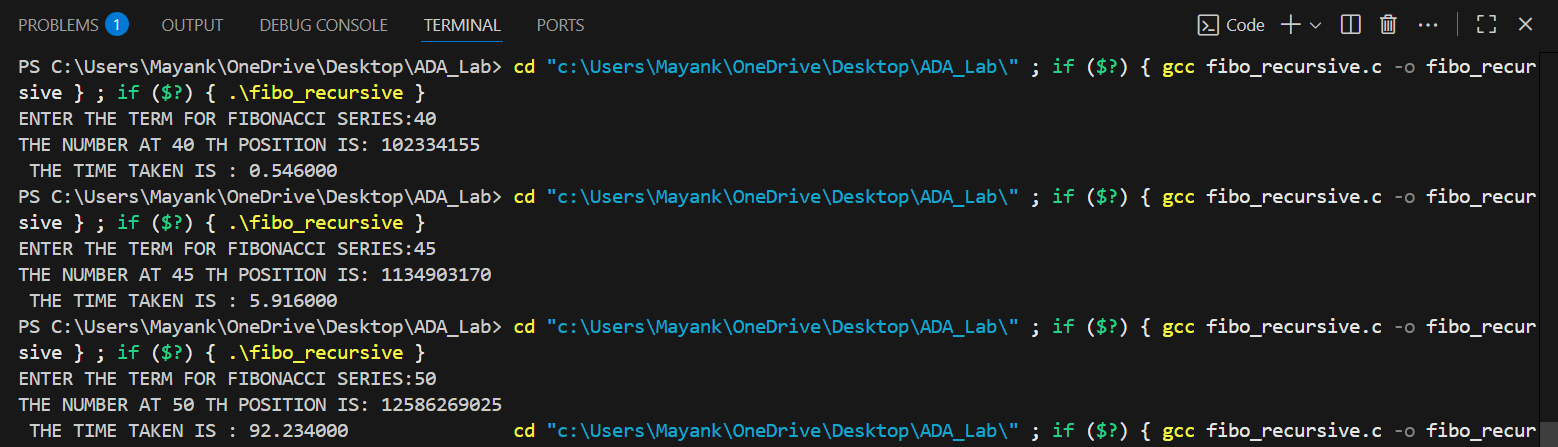
    printf("\n THE TIME TAKEN IS : %f",time\_taken);

    return 0;

}

**Output:**

****

****

**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

time\_taken = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0 , 0.064, 0.546, 5.916, 92.234,]

# 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)")

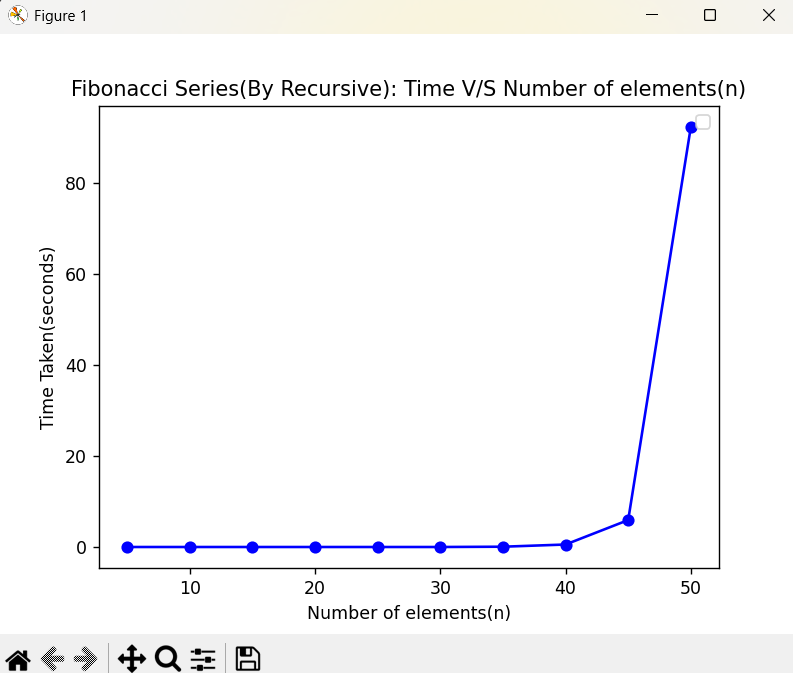
plt.title("Fibonacci Series(By Recursive): Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**Graph:**

****

**4c.  Dynamic Programming- Top Down Approach:-**

**Pseudo Code :**

This approach uses recursion but speeds it up with memoization. It stores the result of each subproblem in an array (memo) so it doesn't have to be recalculated later.

function fibonacci\_top\_down(n):

// Create an array to store results, initialized with a 'not calculated' value

set memo = new array of size (n + 1)

fill memo with -1

return fibonacci\_memo\_helper(n, memo)

// --- Helper function for the top-down approach ---

function fibonacci\_memo\_helper(n, memo):

increment global counter 'function\_calls'

if n <= 1:

return n

// If the result is already stored, return it

if memo[n] is not -1:

return memo[n]

// Otherwise, calculate, store, and then return the result

memo[n] = fibonacci\_memo\_helper(n - 1, memo) + fibonacci\_memo\_helper(n - 2, memo)

return memo[n]

**C Code: -**

#include <stdio.h>

#include<time.h>

long long  nthFibonacciUtil(int n, int memo[]) {

    if (n <= 1) {

        return n;

    }

    if (memo[n] != -1) {

        return memo[n];

    }

    memo[n] = nthFibonacciUtil(n - 1, memo)

                   + nthFibonacciUtil(n - 2, memo);

    return memo[n];

}

long long  nthFibonacci(int n) {

    int memo[n+1];

    for (int i = 0; i <= n; i++) {

        memo[i] = -1;

    }

    return nthFibonacciUtil(n, memo);

}

int main() {

    int n;

  printf("ENTER THE POSITION:");

    scanf("%d",&n);

    clock\_t start =clock();

    long long result;

    for(int i=0;i<1000;i++){

         result = nthFibonacci(n);

    }

    clock\_t end = clock();

    printf("%lld\n", result);

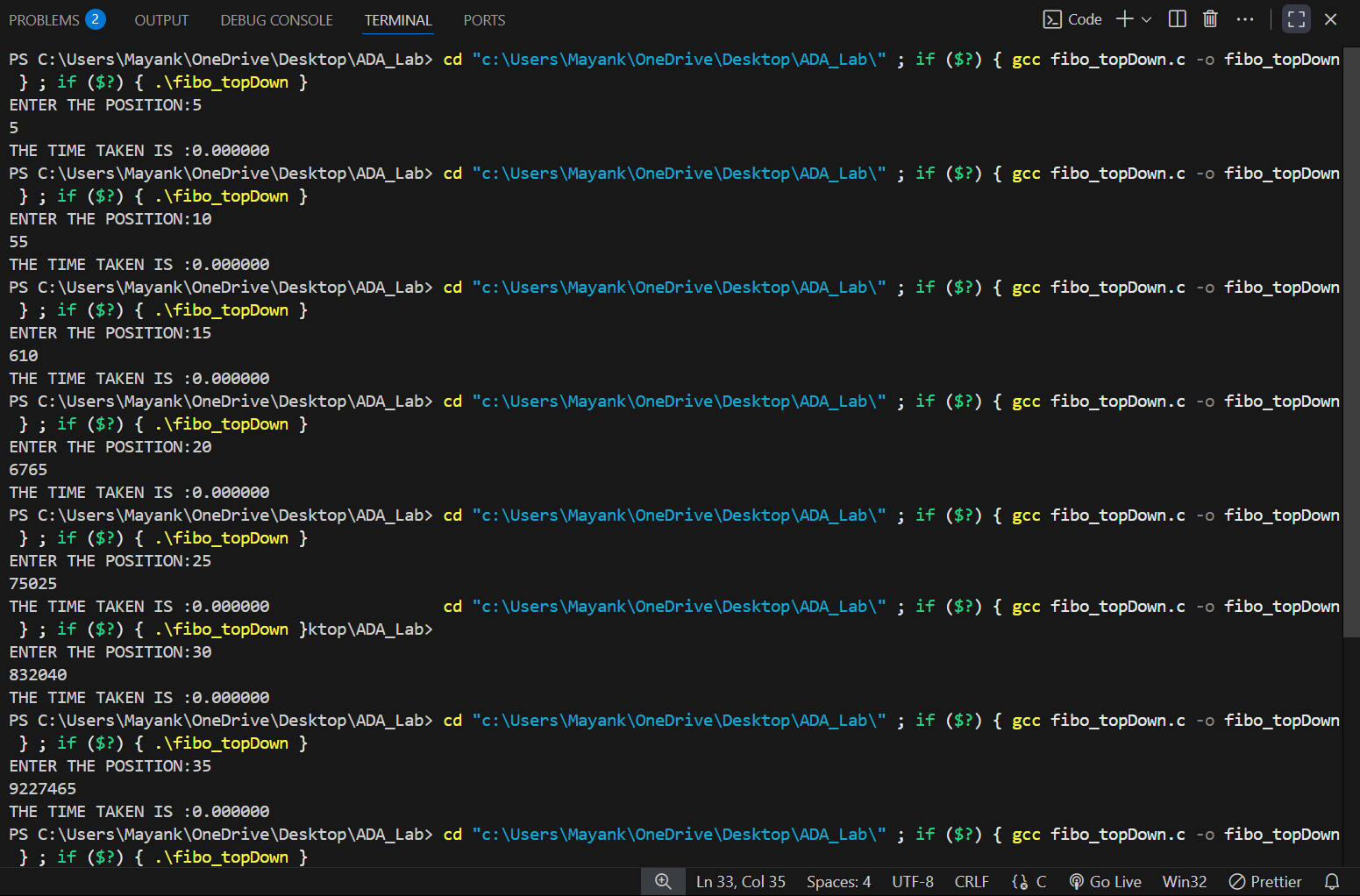
    double time\_taken = (double)(end-start)/CLOCKS\_PER\_SEC/1000;

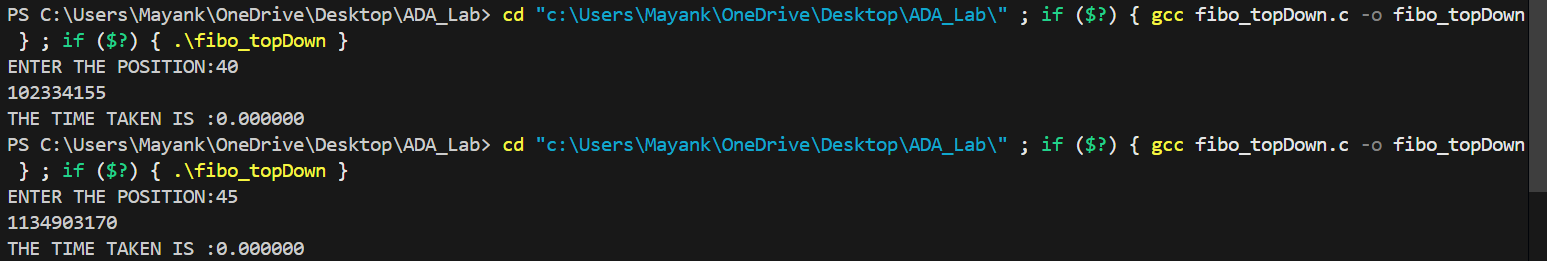
    printf("THE TIME TAKEN IS :%f",time\_taken);

    return 0;

}

**Output:**



****

**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [5, 10, 15, 20, 25, 30, 35, 40, 45]

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)")

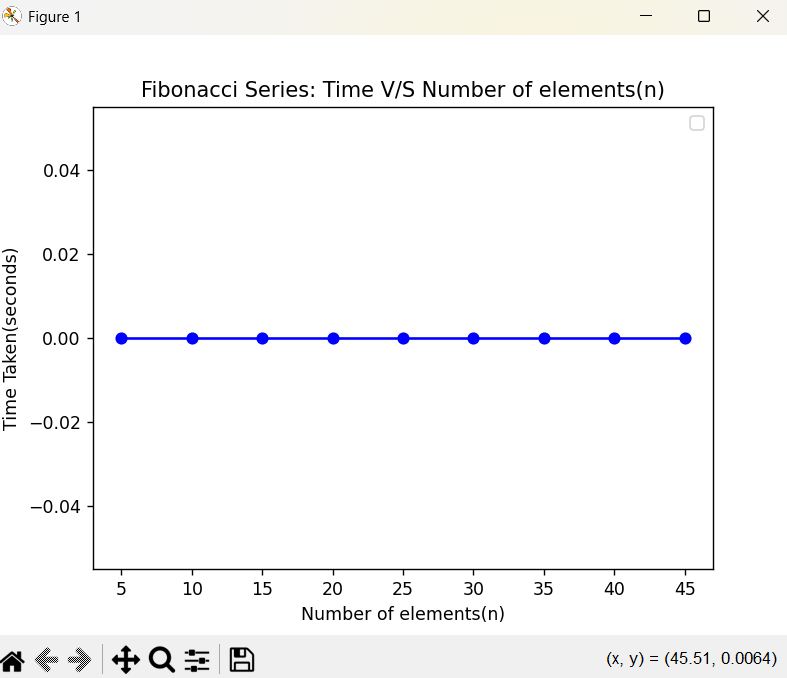
plt.title("Fibonacci Series: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**

****

**4d. Dynamic Programming- Bottom Up Approach:-**

**C Code:-**

#include <stdio.h>

#include<time.h>

long long  nthFibonacci(int n) {

    if (n <= 1) return n;

    int 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 ;

    printf("ENTER THE POSITION:");

    scanf("%d",&n);

    clock\_t start =clock();

     long long result;

    for(int i=0;i<1000;i++){

          result = nthFibonacci(n);

    }

    clock\_t end = clock();

        printf("%lld\n", result);

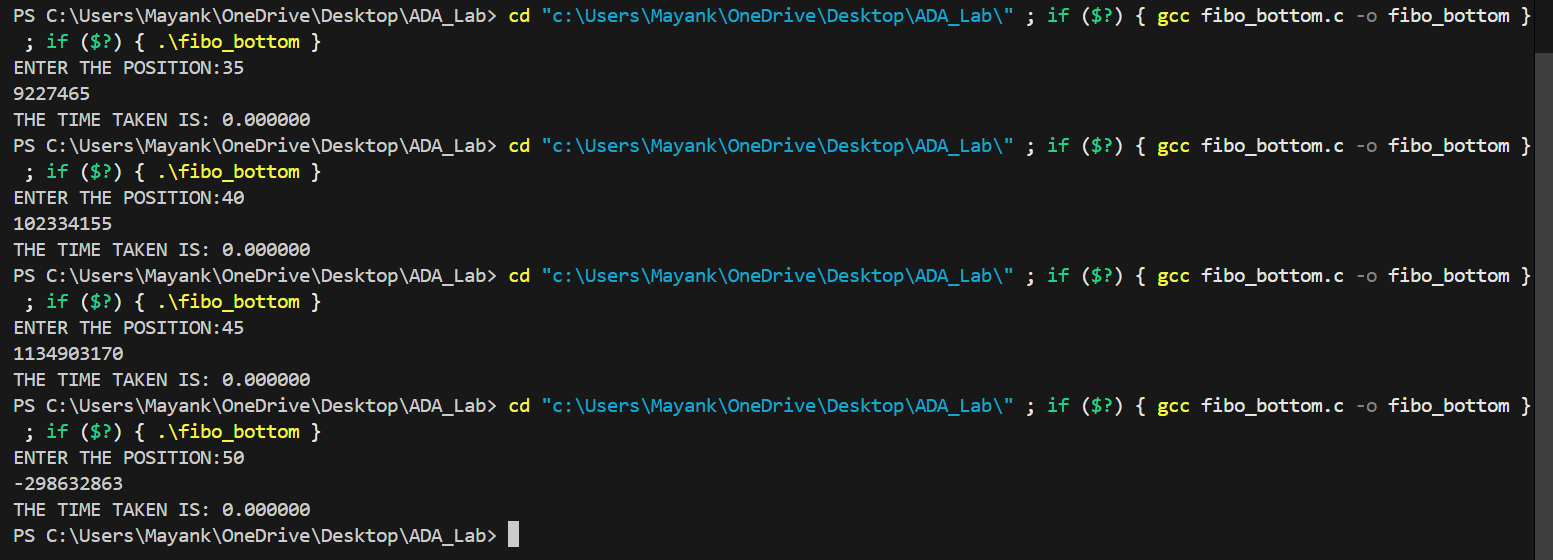
    double time\_taken = (double)(end-start)/CLOCKS\_PER\_SEC/1000;

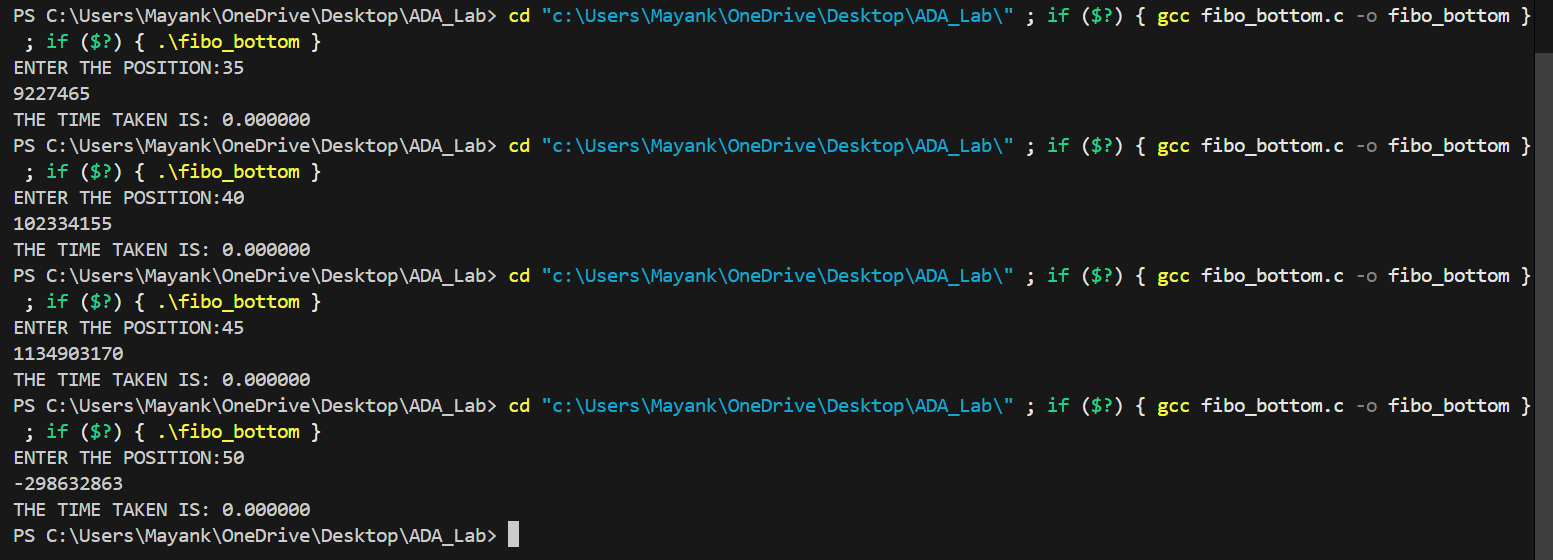
    printf("THE TIME TAKEN IS: %f",time\_taken);

    return 0;

}

**Output:-**

****

****

**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [5, 10, 15, 20, 25, 30, 35, 40, 45]

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)")

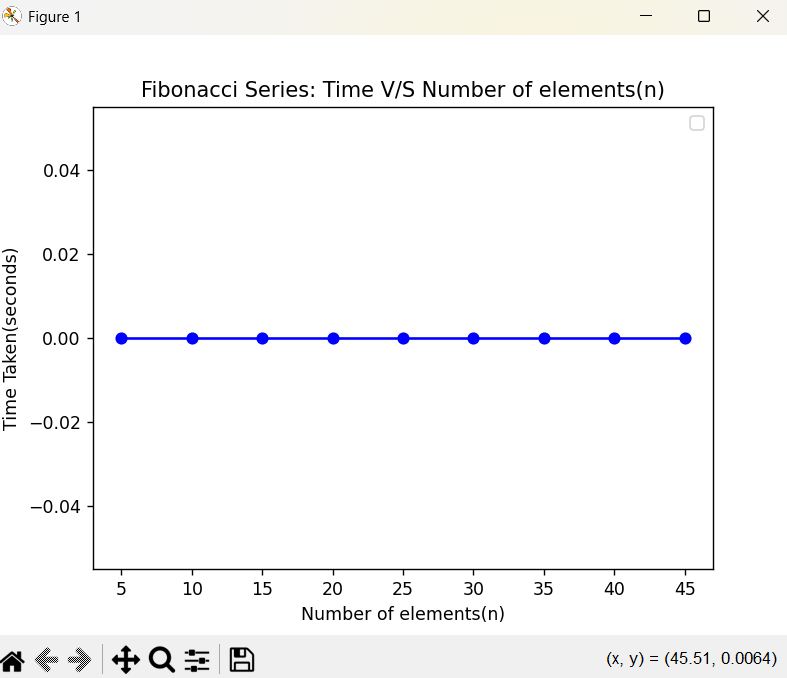
plt.title("Fibonacci Series: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**

****

**Observation –**

1. Time Complexity (Function Calls / Operations)

* Recursive Approach:
  + The recursive approach grows exponentially. For small n (1–20), you can see function calls quickly becoming large (e.g., 13th Fibonacci number already requires 465 calls).
  + This is due to overlapping subproblems being recomputed multiple times.
* Iterative Approach:
  + Linear growth: number of operations = n.
  + Very efficient for small and large n.
* Top-Down DP (Memorization):
  + Nearly linear growth, roughly 2n.
  + Extra operations come from checking the memo table before computing a value.
  + Eliminates redundant computations compared to plain recursion.
* Bottom-Up DP:
  + Linear growth similar to iterative approach.
  + Each Fibonacci number is computed exactly once, no recursion overhead.

2. Space Complexity (Memory Units)

* Recursive Approach:
  + Linear O(n) due to recursion stack.
* Iterative Approach:
  + Constant O(1) space: only a few variables are used.
* Top-Down DP:
  + Linear O(n) because of memorization array plus recursion stack.
* Bottom-Up DP:
  + Linear O(n) for the table storing computed Fibonacci numbers.
  + Slightly more memory-efficient than top-down recursion since no recursion stack is used.

**Conclusion –**

Efficiency:

* Recursive approach is highly inefficient for large n because of exponential time complexity.
* Iterative and Bottom-Up DP approaches are fastest, with linear time complexity.
* Top-Down DP is slightly slower than Bottom-Up due to recursion overhead, but still much better than naive recursion.

Memory Usage:

* Iterative approach is the most memory-efficient, using only O(1) space.
* Recursive and Top-Down DP use O(n) memory due to recursion stack.
* Bottom-Up DP uses O(n) memory for the table, but avoids recursion overhead.