NAME: ADITYA JETHANI
DIV-3 **G-**5
ROLL NO: 21BCP166
DAA: Extra lab assignment

AIM: Making a program to print fibonacci series using an iterative and recursive approach.

Theory: Fibonacci series is a sequence of whole numbers which are generated by the addition of previous two numbers. The seed elements are 0 and 1.
The series is as follows:
0 1 1 2 3 and so on.

Algorithm and Code for iterative approach:

```
Procedure Iterative_Fibonacci(n):
    int f0, f1, fib
    f0 = 0
    f1 = 1
    display f0, f1
    for int i := 1 to n:
        fib := f0 + f1
        f0 := f1
        f1 := fib
        display fib
    END for loop
END Iterative_Fibonacci
```

ANALYSIS:

The algorithm uses a **for-loop** to calculate the Fibonacci numbers, and two variables **f0** and **f1** to keep track of the previous two terms in the sequence.

At the start of the procedure, **f0** and **f1** are both initialized to 0 and 1 respectively. The first two terms of the sequence are then displayed.

The **for-loop** then calculates and displays each subsequent term in the sequence. The value of **fib** is calculated as the sum of **f0** and **f1**, and the values of **f0** and **f1** are then updated to **f1** and **fib** respectively. This allows the algorithm to keep track of the previous two terms as it calculates each new term.

In mathematical terms, the time complexity of the algorithm can be expressed as **O(n)**. This means that the time taken by the algorithm grows linearly with the input size **n**. As **n** increases, the time taken by the algorithm will increase in proportion to **n**.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
int fib(int n)
{
    int a = 0, b = 1, c, i;
    if (n == 0)
        return a;
    for (i = 2; i <= n; i++)
    {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
int main()
{
    int n;
    cout << "Enter the number of terms: ";
    cin >> n;
    cout << "Fibonacci Series: ";
    for (int i = 0; i < n; i++)
        cout << fib(i) << " ";
    return 0;
}
```

Algorithm and Code for recursive approach:

```
Procedure Recursive_Fibonacci(n)
   int f0, f1
   f0 := 0
   f1 := 1
   if(n <= 1):
      return n
   return Recursive_Fibonacci(n-1) + Recursive_Fibonacci(n-2)
END Recursive_Fibonacci
```

ANALYSIS:

The time complexity of the recursive implementation of the Fibonacci sequence can be determined by counting the number of recursive calls needed to compute the nth Fibonacci number.

Let **T(n)** represent the number of operations required to compute the **nth** Fibonacci number. Then, the following recurrence relation can be derived:
$T(n) = T(n-1) + T(n-2) + 1$
where the **+1** term represents the operation to add the two previous Fibonacci numbers. The base case **$T(0) = T(1) = 1$**, as each of these numbers only requires one operation to compute.

This recurrence relation represents an exponential growth in the number of operations required to compute the nth Fibonacci number. In other words, **T(n)** grows as **2^n**, so the time complexity is **O(2^n)**.

CODE:

```cpp
#include <bits/stdc++.h>
using namespace std;
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
int main()
{
    int n;
    cout << "Enter the number of terms: ";
    cin >> n;
    cout << "Fibonacci Series: ";
    for (int i = 0; i < n; i++)
        cout << fib(i) << " ";
    return 0;
}
```

FINDING AND SOLVING THE RECURRENCE EQUATION:

The time complexity of the recursive implementation of the Fibonacci sequence can be determined using the substitution method. To use this method, we need to determine a closed-form expression for T(n) by substituting smaller values of n into the recurrence relation until we reach the base cases T(0) and T(1).

Let's consider the following recurrence relation:

$T(n) = T(n-1) + T(n-2) + 1$, for $n > 1$
$T(0) = 1$ $T(1) = 1$

We can start by substituting $T(0)$ and $T(1)$ into the recurrence relation to get $T(2)$:

$T(2) = T(1) + T(0) + 1 = 2$

Next, we can substitute $T(1)$ and $T(2)$ into the recurrence relation to get $T(3)$:

$T(3) = T(2) + T(1) + 1 = 4$

Continuing this process, we can see that $T(n)$ grows exponentially with n, so the time complexity of this implementation is O(2^n).