Problem Statement:

Write a non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

Fibonacci Sequence Definition:

The Fibonacci sequence is a series of numbers in which each number (Fibonacci number) is the sum of the two preceding ones, typically starting with 0 and 1.

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2), \quad \text{for } n \geq 2$$

1. Non-Recursive Program Logic:

In a non-recursive (iterative) solution, we calculate the Fibonacci numbers in a bottom-up manner, starting from the smallest subproblems, i.e., `F(0)` and `F(1)`.

Procedure (Non-Recursive):

- Initialize two variables, `a = 0` and `b = 1`, to store `F(0)` and `F(1)`.
- For each subsequent Fibonacci number, calculate it as the sum of the two previous numbers (`a` and `b`).
- Update `a` and `b` for each step.
- Continue this process until you reach the desired Fibonacci number.

Python Code

```
def fibonacci_non_recursive(n):
 if n == 0:
 return 0
 elif n == 1:
 return 1

 a, b = 0, 1
 for i in range(2, n + 1):
 fib = a + b
 a = b
 b = fib
 return b
```

Time Complexity (Non-Recursive):

- Time complexity: **O(n)** (since we loop `n` times to compute `F(n)`).

Space Complexity (Non-Recursive):

- Space complexity: **O(1)** (we only use a constant amount of space for the two variables `a` and `b`).

2. Recursive Program Logic:

In a recursive solution, the Fibonacci number `F(n)` is computed by recursively calling the function for `F(n-1)` and `F(n-2)` until the base cases (`F(0)` and `F(1)`) are reached.

Procedure (Recursive):

- If `n == 0`, return 0.
- If `n == 1`, return 1.
- Otherwise, recursively compute the value of `F(n)` by adding the results of `F(n-1)` and `F(n-2)`.

Time Complexity (Recursive):

- Time complexity: **O(2^n)** (since the recursive algorithm calculates many overlapping subproblems repeatedly, leading to an exponential number of function calls).

Space Complexity (Recursive):

- Space complexity: **O(n)** (due to the depth of the recursion tree, which is proportional to `n`).

Comparison of Recursive vs Non-Recursive:

Approach	Time Complexity	Space Complexity	Remarks
Recursive	O(2^n)	O(n)	Inefficient due to recomputation of subproblems.
Non Recursive	O(n)	O(1)	Efficient and scalable for large Fibonacci numbers.

Conclusion:

For large values of `n`, the non-recursive approach is significantly more efficient in terms of both time and space. The recursive approach is intuitive but impractical for large input sizes due to its exponential time complexity.