2. Recursive Function and Efficiency Analysis - Write a recursive function pseudocode And calculate the nth Fibonacci number and use Big O notation to analyze its efficiency. Compare this with an iterative approach and discuss the pros and cons in terms of space and time complexity.

|  |  |
| --- | --- |
| **Recursive Fibonacci Function** | |
| pseudocode for a recursive function to calculate the nth Fibonacci number. | |
| Fibo(n): |  |

if n <= 1:

return n

else:

return Fibo(n - 1) + Fibo(n - 2)

In this pseudocode:

|  |  |  |
| --- | --- | --- |
| ● | The base case checks if n is less than or equal to 1. If so, it returns n. | |
| ● | Otherwise, it recursively calculates Fibo(n - 1) and Fibo(n - 2) and adds | |
|  | them together. | |
| **Time Complexity Analysis** | | | |
| Now let’s analyze the time complexity using Big O notation. For the recursive Fibonacci | | | |
| function: | | | |
| 1. Each recursive call results in two additional recursive calls (one for Fibo(n - | | |
| 1) and one for Fibo(n - 2)). | | |
| 2. The recursion tree has a depth of n. | | |
| Therefore, the total number of function calls is roughly exponential, specifically O(2^n). | | | |
| As n increases, the function becomes significantly slower due to the repeated | | | |
| calculations. | | | |
| **Iterative Approach** | | | |
| An alternative approach is an iterative solution that avoids the exponential time | | | |
| complexity. Here’s an example of an iterative Fibonacci function: | | | |
| Iterative Fibo(n): | | |  |

if n <= 1:

return n

a, b = 0, 1

for \_ in range(n - 1):

a, b = b, a + b

return b

In this iterative approach:

|  |  |
| --- | --- |
| ● | We maintain two variables (a and b) to keep track of the current and next |
|  | Fibonacci numbers. |
| ● | We iterate n - 1 times, updating a and b in each iteration. |

**Pros and Cons**

|  |  |
| --- | --- |
| ● | **Time Complexity**: |

○ Recursive: O(2^n)

○ Iterative: O(n)

○ The iterative approach is significantly faster.

|  |  |
| --- | --- |
| ● | **Space Complexity**: |

○ Recursive: O(n) due to the recursion stack.

○ Iterative: O(1) as it doesn’t use additional memory for the stack.

|  |  |
| --- | --- |
| ● | **Pros of Recursive Approach**: |

○ Simplicity and readability.

○ Easy to understand.

○ Exponential time complexity.

○ Recursion overhead.

|  |  |
| --- | --- |
| ● | **Pros of Iterative Approach**: |

|  |  |
| --- | --- |
| ● | **Cons of Recursive Approach**: |

○ Linear time complexity.

○ Efficient.

○ No recursion overhead.

|  |  |
| --- | --- |
| ● | **Cons of Iterative Approach**: |

○ Less intuitive for some readers.