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.  
  
**Using recursion**    
  
Algorithm

* Check if *n* ≤ 1. If so, return *n*.
* Check if *n* > 1. If so, call our function *F* with inputs *n*-1 and *n*-2, and return the sum of the two results.

Pseudocode   
 algorithm **F**(n):

// INPUT

// n = Some non-negative integer

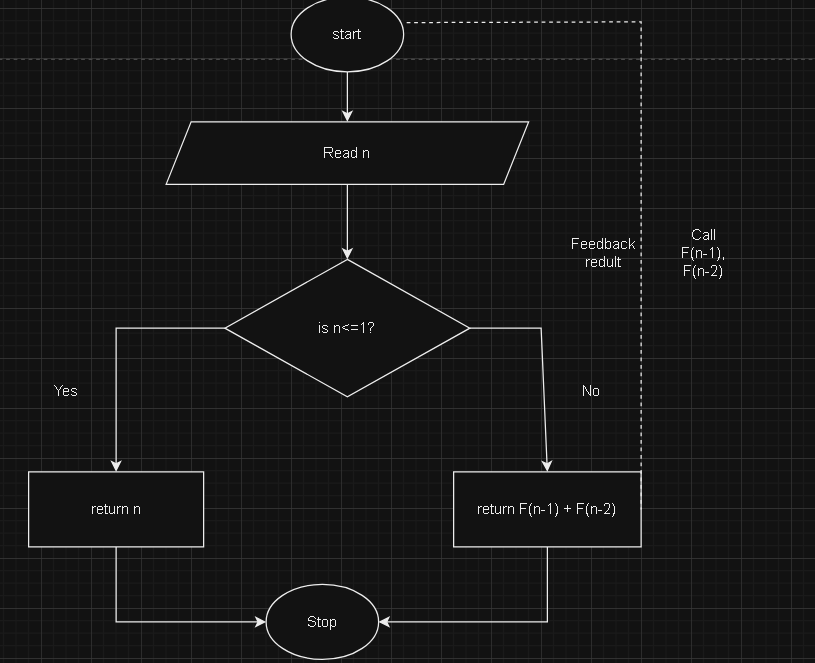
// OUTPUT

// The nth number in the Fibonacci Sequence

if n <= 1:

return n

else:

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


**Using Iterative**

Algorithm

First, we’ll store 0 and 1 in *F*[0] and *F*[1], respectively.

Next, we’ll iterate through array positions 2 to *n-1*. At each position *i*, we store the sum of the two preceding array values in *F*[*i*].

Finally, we return the value of *F*[*n*-1], giving us the number at position *n* in the sequence.

Pseudocode  
 algorithm F(n):

// INPUT

// n = Some non-negative integer

// OUTPUT

// The nth number in the Fibonacci Sequence

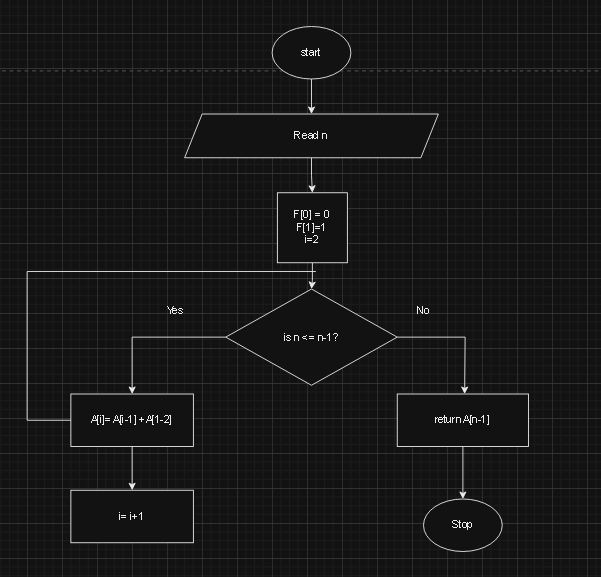
A[0] <- 0

A[1] <- 1

for i <- 2 to n - 1:

A[i] <- A[i - 1] + A[i - 2]

return A[n - 1]

Flowchart  


**Time complexity and space**  
  
Analyzing the time complexity for our iterative algorithm is a lot more straightforward than its recursive counterpart.

In this case, our most costly operation is assignment. Firstly, our assignments of *F*[0] and *F*[1] cost O(1) each. Secondly, our loop performs one assignment per iteration and executes (*n*-1)-2 times, costing a total of O(*n*-3) = O(*n*).

Therefore, our iterative algorithm has a time complexity of O(*n*) + O(1) + O(1) = O(*n*).

This is a marked improvement from our recursive algorithm!