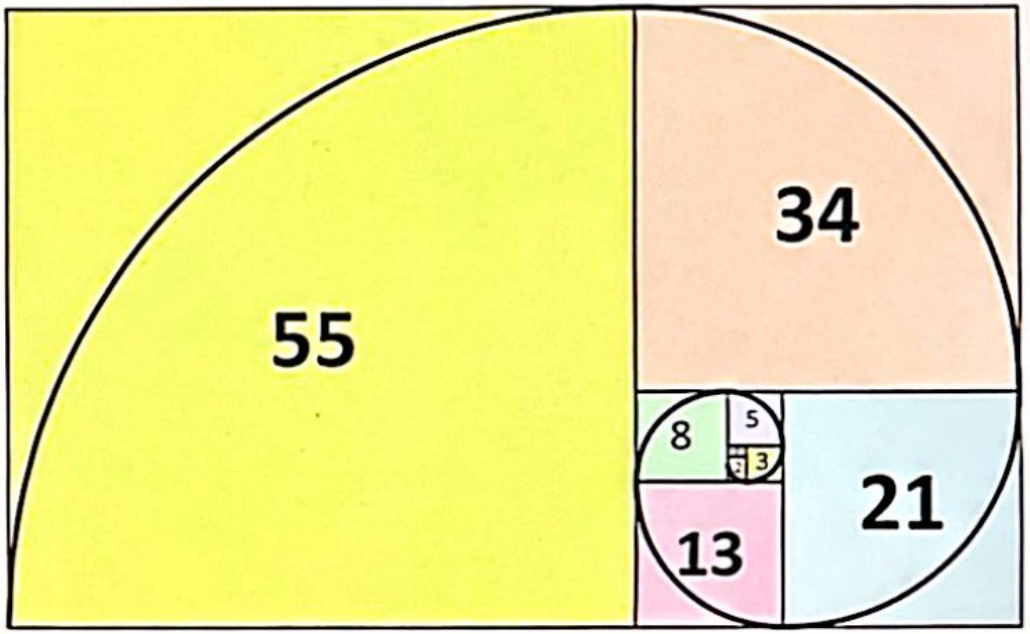
Mathematical Implications:

1. Fibonacci Sequence (Iterative Method).
2. Fibonacci Sequence (Recursive Method).
3. Graph theory.



Fibonacci Sequence (Iterative Method)

- **Time Complexity :**

4n+1 is the total number of operations performed by our Fibonacci function. However, in Big O notation, constants and lower-order terms are ignored because they don't significantly affect the growth rate as n increases. Therefore, the time complexity is O(n).

- **Space Complexity :**

The function uses a constant amount of memory regardless the size of n. The only space it uses is for variables 'x' , 'y' and 'z'. So, the space complexity is O(1).

ALGORITHM 8 An Iterative Algorithm for Computing Fibonacci Numbers.

```
procedure iterative fibonacci(n: nonnegative integer)
if n = 0 then return 0
else
    x := 0
    y := 1
    for i := 1 to n − 1
        z := x + y
        x := y
        y := z
    return y
{output is the nth Fibonacci number}
```

Fibonacci Sequence (Recursive Method)

- **Formula:**

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

- **Number of Operations :**

The number of operations(recursive calls) is therefore proportional to the number of recursive calls, which grows exponentially as O(2n).

- **Time Complexity :**

The number of operations grows exponentially with respect to n because each function call spawns two more recursive calls. Thus, the total number of recursive calls grows as O(2n). So, time Complexity O(2n). This is because the function recursively calls itself for both n−1n-1n−1 and n−2n-2n−2, which leads to an exponential number of calls.

- **Space Complexity :**

The space complexity of the recursive Fibonacci Sequence is determined by the maximum depth of the recursion stack, as each recursive call adds a new frame to the stack. In the worst case , the recursion depth is n, since each recursive call reduces *'n'* only by 1 until base case is reached. So, Space Complexity is O(n).

ALGORITHM 7 A Recursive Algorithm for Fibonacci Numbers.

procedure *fibonacci*(n: nonnegative integer)
if $n = 0$ **then return** 0
else if $n = 1$ **then return** 1
else return $fibonacci(n - 1) + fibonacci(n - 2)$
{output is $fibonacci(n)$}

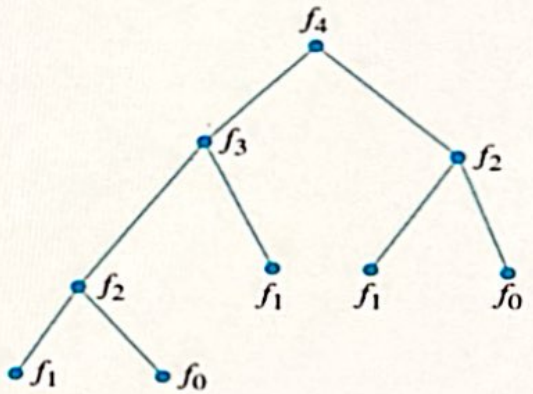
Recursive Method and Recursive stack

- In the recursive method, the number of function calls doubles with each increase in 'n'. This leads to an exponential growth in the number of operations, which we represent as O(2n). This means that as 'n' gets larger, the total number of operations increases at an incredibly fast rate. For example, for n=10, we have approximately 1024 operations, and for n=20, the number of operations exceeds a million!
- Recursive stack is the memory structure used to keep track of all the active recursive calls and their execution state. The number of active calls at any given time determines the depth of the stack

Graph Theory

- **What are Fibonacci Trees?**

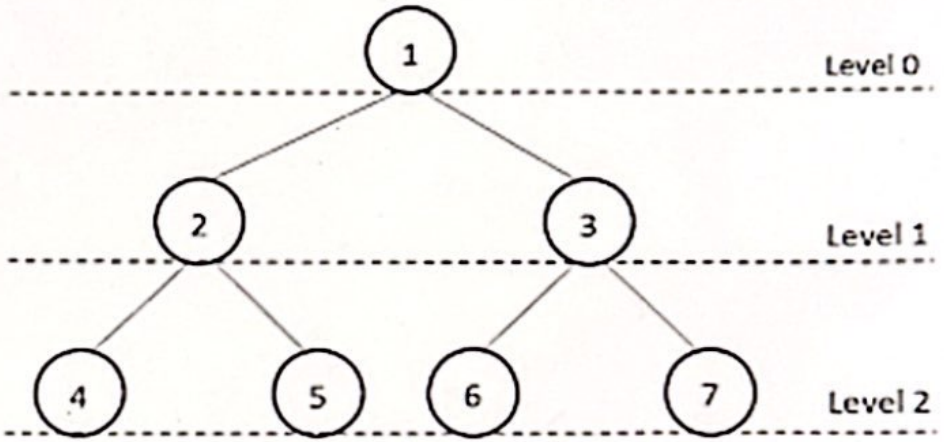
A Fibonacci *Tree is a* Binary tree where the number of nodes at *each level* follows Fibonacci Sequence.



Number of Levels

- **How to calculate the number of levels?**

It's easy we can already see that the number of levels in a Fibonacci tree *for F(n)* is 'n'. The tree starts at level 0 with F(n) and *recursively* branches down until it reaches the *base cases* at *level* (n-1) and n. As we move to higher Fibonacci *numbers*, the number of levels *in* the Fibonacci tree *also* increases. For example, the Fibonacci tree *for* F(5) has 5 levels, while the tree for F(8) has *8 levels*. This reflects the growing complexity of the *recursive* structure as the Fibonacci number increases.



Number of Nodes

- **How to calculate the number of Nodes?**

Now that we've *understood the* structure of Fibonacci tree *along* with the calculation of *the number* of levels, we are *now* going to see how to calculate the *number of* nodes in a tree. The way of calculating the number of nodes is pretty straight forward too.

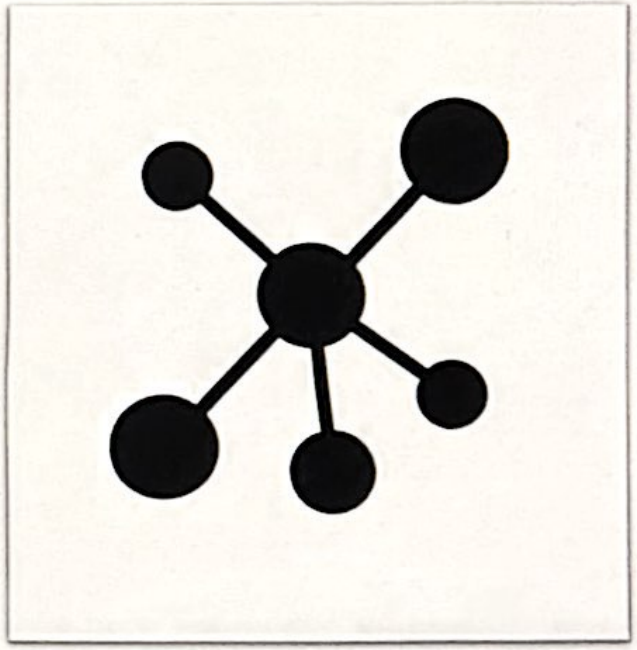
N(n)=N(n-1) + N(n-2) + 1

N(n): The number of nodes at level n in a Fibonacci tree.

N(n-1): The number of nodes at level n-1.

N(n-2): The number of nodes at level n-2.

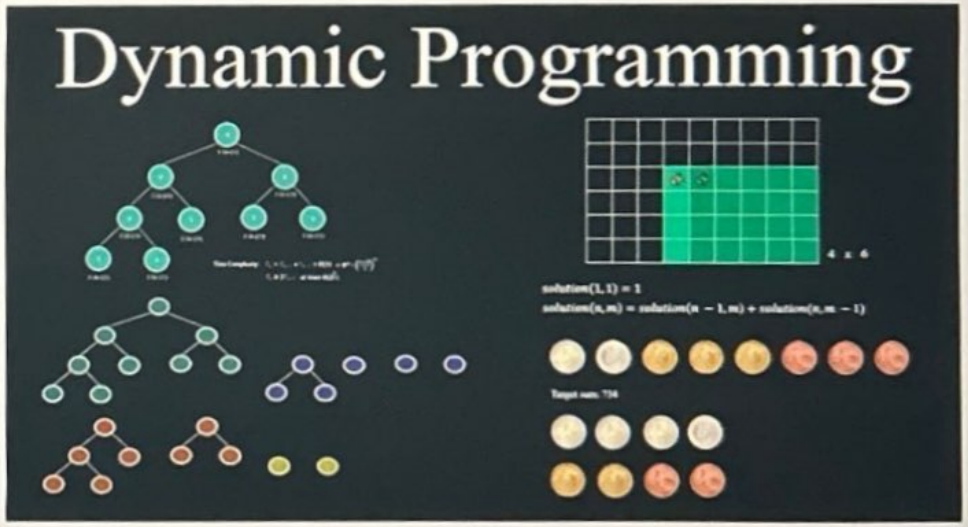
+1 : This accounts for the root level node.



Dynamic Programming

- **What is Dynamic Programming?**

Dynamic Programming is a problem-solving technique used to solve problems that can be broken down into smaller, overlapping subproblems. The key idea behind DP is to avoid redundant work by storing the results of already solved subproblems and reusing these results in subsequent computations. This significantly improves efficiency.



Memoization

- We can also observe in the Fibonacci tree that the same branches appear multiple times. Instead of recalculating these branches each time, we can store these values and reuse them when needed. This technique is called Memoization. In Memoization, we save the results of function calls that are repeated and use them directly when we encounter the same subproblem again, thus eliminating unnecessary recalculations.