SHOW-N-TELL

Fibonacci Sequence

Mathematical Implications:

1. **Fibonacci Sequence (Iterative Method)**
2. **Fibonacci Sequence (Recursive Method)**
3. **Recurrence Relation**
4. **Graph theory**
    - Fibonacci Trees
        - How to calculate Nodes
        - How to calculate Levels
    - Dynamic Programming
        - Memoization
    - Fibonacci Graphs

Introduction:

So, now after the brief introduction by Zain we are now going to dive into the mathematical implications of Fibonacci Sequence and it's going to be interesting. It's fantastic, isn't it? That the simple yet profound sequence starting from 0 and 1 , with each number being the sum of two previous terms have so many properties in so many phenomena? We are going to dive deep into Fibonacci Sequence and connect it to various topics in Discrete Mathematics. We'll begin by examining the computational techniques -both recursive and iterative- That allow us to calculate Fibonacci Numbers. Then we'll move into the theoretical realm, delving into recurrence relations and their foundational role in defining sequences like Fibonacci.

From there, we'll explore how the Fibonacci sequence naturally emerges in **Graph Theory**, where we'll encounter fascinating structures called **Fibonacci Trees**. Using these trees, I'll introduce you to an incredible optimization technique from **Dynamic Programming** known as **Memoization**, which makes calculations much more efficient.

We'll see how Fibonacci Sequence serves as a bridge between computation, structure, and pure Mathematics.

Fibonacci Sequence (Iterative Method)

Now, let's look at how we can calculate Fibonacci numbers by iterative method. So, what does iterative method means? In iterative method we generally use for loops or while loops to calculate the sequence. We start from base case and build up to the desired term. This method avoids recursion by maintaining only the recent Fibonacci Numbers in variables, which makes it efficient in both space and time.

Given Below is the Pseudocode for calculating Fibonacci Numbers using Iterative Method.

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}
```

Explanation of Pseudocode:

1- The input to the function is 'n' which is the desired term.
2- Then we define the condition that if 'n' is equal to 0 or 1, the function immediately return 'n' as this is the base case.
3- We initialize our 'x' variable by 0 representing **F(0)** and 'y' variable as 1 representing **F(1).**
4- Then the loop starts from 1 and goes up to (n-1).{At i=1, the loop is calculating the second Fibonacci number so the loop will only go up to n-1}.
5- At each iteration, 'z' is calculated as the next Fibonacci term while x and y are initialized to the next terms.
6- After the loop ends y holds the desired value.

You May Ask why does 'y' holds the desired value?

So, the reason is simple, 'y' always holds the most recent Fibonacci number.

So, when the loop ends 'y' holds **F(n).**

Number of Operations:

The number of operations in the above code are **4n+1**.

Base Case Check : (n==0 || n==1)

Total operations : 3 (two assignment, one logical)

Variable Initialization : (a=0 && b==1)

Total operations : 2 (one for x one for y)

Loop : (i=1 and The loop runs n-1 times)

Inside The Loops: (1 Addition, 2 Assignments)

Total operations : 4(n-1).

Total number of Operations : 3 + 2 + (4n-4) = **4n+1.**

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).**

This means that the number of operations grow linearly with input size **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).**

Fibonacci Sequence (Recursive Method)

Now, as we have studied Iterative method in great detail let's now move on to another way of calculating Fibonacci Numbers- That is Recursive Method. In recursive approach, we define the Fibonacci sequence in terms of itself, where each term is calculated by recursively calling the function to compute the previous two terms. This method directly uses the definition of Fibonacci, with base cases defined for F(0) and F(1). However, it can be inefficient for large value of 'n' because it involves repeated calculations of the same Fibonacci numbers. Despite its simplicity, this method can quickly lead to exponential time complexity."

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 $fibonacci(n)$}

Explanation of Pseudocode :

1- The input of the function is 'n', which is the desired Fibonacci Term.
2- If 'n==0' return 0 that is the first Fibonacci Number.
3- If 'n==1' return 1 that is the second Fibonacci Number.
4- If n > 1, The function makes two recursive calls, which gives the next Fibonacci number as per the relation :

$$\mathbf{F(n)=F(n-1) + F(n-2)}$$

5- The final result returned from the recursive calls will be **F(n).**

Example:

Fibonacci(5) = Fibonacci(4) + Fibonacci (3) = (Fibonacci(3) +Fibonacci(2)) + (Fibonacci(2) +Fibonacci(1)).

and so on until it reaches base cases of 0 and 1.

Number of Operations :

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

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 **$O(2^n)$.** 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!**

Base Case Check: (n==0 || n==1)

Total Operations: 3 (two assignment, one logical)

Recursive Calls : (Fibonacci(n-1) and Fibonacci(n-2))

Total Operations : **$O(2^n)$**.

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 **$O(2^n)$.**

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).**

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**

Recurrence Relation.

Now that we've explored the iterative and recursive methods for calculating Fibonacci Numbers, let's take a moment to dive deeper into the underlying structure that makes both of these methods work: **Recurrence Relations!**

Definition: "A recurrence relation is an equation that recursively defines a sequence of values based on previous terms. In the case of Fibonacci Sequence, each number is the sum of two preceding ones, which is essentially a **Recurrence Relation.**"

So, whether we're using the iterative or recursive method, both are just ways of solving recurrence relation that defines the Fibonacci Sequence. Let's now take a closer look at recurrence relations and how they help us formalize Fibonacci Numbers.

In Fibonacci the recurrence relation is:

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

With base cases,

$$F(0)=0 , F(1)=1$$

This type of recurrence relation is **Linear and homogeneous.** Which means:

Linear: Term is a combination of preceding terms.

Homogeneous: The Sequence depends only on previous terms

Some other types of **Recurrence Relations** are :

Non-Linear: A relation where the terms are combinations in a non-linear way i.e., through multiplications or exponents.

Example: Factorial Sequence :

$$F(n) = n. F(n-1), F(0) =1.$$

Non-Homogeneous : A Sequence that not only depends on the previous terms but also has any additional independent terms.

Example: Arithmetic Sequence with a Linear term

$$a_n = 2a_{n-1} +n , a(0)=1$$

This recurrence has an additional term 'n which makes it non-linear.

Solving Recurrence Relations:

1- One of the ways of solving these relations is the iterative method which you have already seen , where we just plug in values and solve them step by step.
2- Another way is the closed form solutions !

The closed form solution of the Recurrence Relation is an expression that directly computes the value of a sequence without the need for recursion or iteration. Instead of repeatedly applying the recurrence relations, the closed form solution gives you the formula that allows you to calculate the 'n^{th}' term directly without referencing the previous terms.

So, can we derive such a closed form solution for Fibonacci Sequence?

The answer is **Yes!** And it is called the **Binet's Formula** and here is how we can derive it:

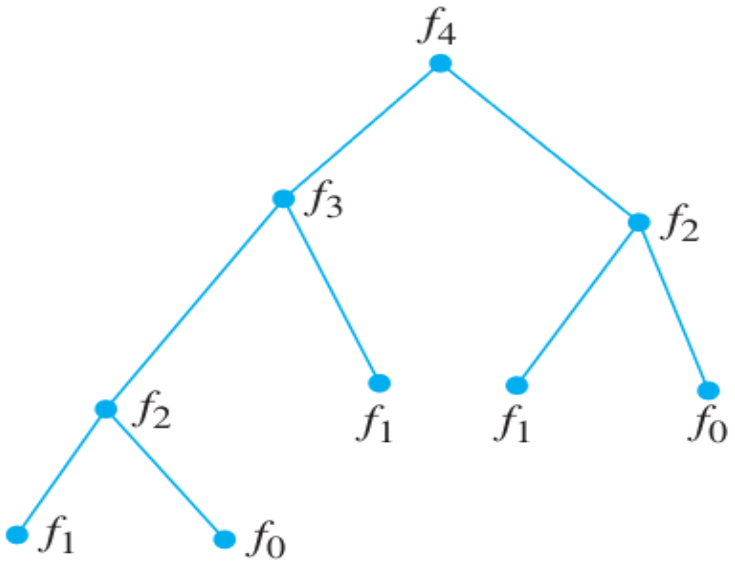
Binet's Formula:

Graph Theory.

Up until now we have only seen how we can algebraically derive Fibonacci numbers but they are not just limited to algebra , they also have important characteristics in graph theory and one of the beautiful ways we can see Fibonacci numbers graphically is **Fibonacci trees**.

What are Fibonacci Trees?

A **Fibonacci Tree** is a Binary tree where the number of nodes at each level follows Fibonacci Sequence. The structure of the tree directly reflects the Fibonacci sequence, making it the perfect graphical representation of Fibonacci Numbers.



This is an example of a Fibonacci Tree! .

- The root is **F(n),** which in this case is F(4)
- Each node represents a Fibonacci number and it branches into two child nodes.
- The left child represents **F(n-1).**
- The right child represents **F(n-2).**

The base cases **F(1)** and **F(0)** are the leaves of the tree where recursion terminates.

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**.**

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

As we can see, for larger Fibonacci numbers, the same sequence is being calculated repeatedly. This inefficiency is also visible in the Fibonacci tree, where the same branches appear again and again. To address this, we can use **Dynamic Programming (DP)** to optimize our approach.

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.

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.