# Unit 4: Dynamic Programming

Dynamic programming is a commonly used algorithmic technique used to optimize recursive solutions when the same subproblems are called again.

- The core idea behind DP is to store solutions to subproblems so that each is solved only once.
- To solve DP problems, we first write a recursive solution in a way that there are overlapping subproblems in the recursion tree (the recursive function is called with the same parameters multiple times)
- To make sure that a recursive value is computed only once (to improve the time taken by the algorithm), we store the results of the recursive calls.
- There are two ways to store the results; one is top-down (or memoization), and the other is bottom-up (or tabulation).

## When to Use Dynamic Programming (DP)?

Dynamic programming is used for solving problems that consist of the following characteristics:

### 1. Optimal Substructure:

The property Optimal substructure means that we use the optimal results of subproblems to achieve the optimal result of the bigger problem.
Example:
Consider the problem of finding the minimum cost path in a weighted graph from a source node to a destination node. We can break this problem down into smaller subproblems:

- Find the minimum cost path from the source node to each intermediate node.
- Find the minimum cost path from each intermediate node to the destination node.

The solution to the larger problem (finding the minimum cost path from the source node to the destination node) can be constructed from the solutions to these smaller subproblems.

### 2. Overlapping Subproblems:

The same subproblems are solved repeatedly in different parts of the problem refer to Overlapping Subproblems Property in Dynamic Programming.
Example:
Consider the problem of computing the Fibonacci series. To compute the Fibonacci number at index n, we need to compute the Fibonacci numbers at indices n-1 and n-2. This means that the subproblem of computing the Fibonacci number at index n-2 is used twice (note that the call for n – 1 will make two calls, one for n-2 and other for n-3) in the solution to the larger problem of computing the Fibonacci number at index n.

## Approaches of Dynamic Programming (DP)

Dynamic programming can be achieved using two approaches:

### 1. Top-Down Approach (Memoization):

In the top-down approach, also known as memoization, we keep the solution recursive and add a memoization table to avoid repeated calls of same subproblems.

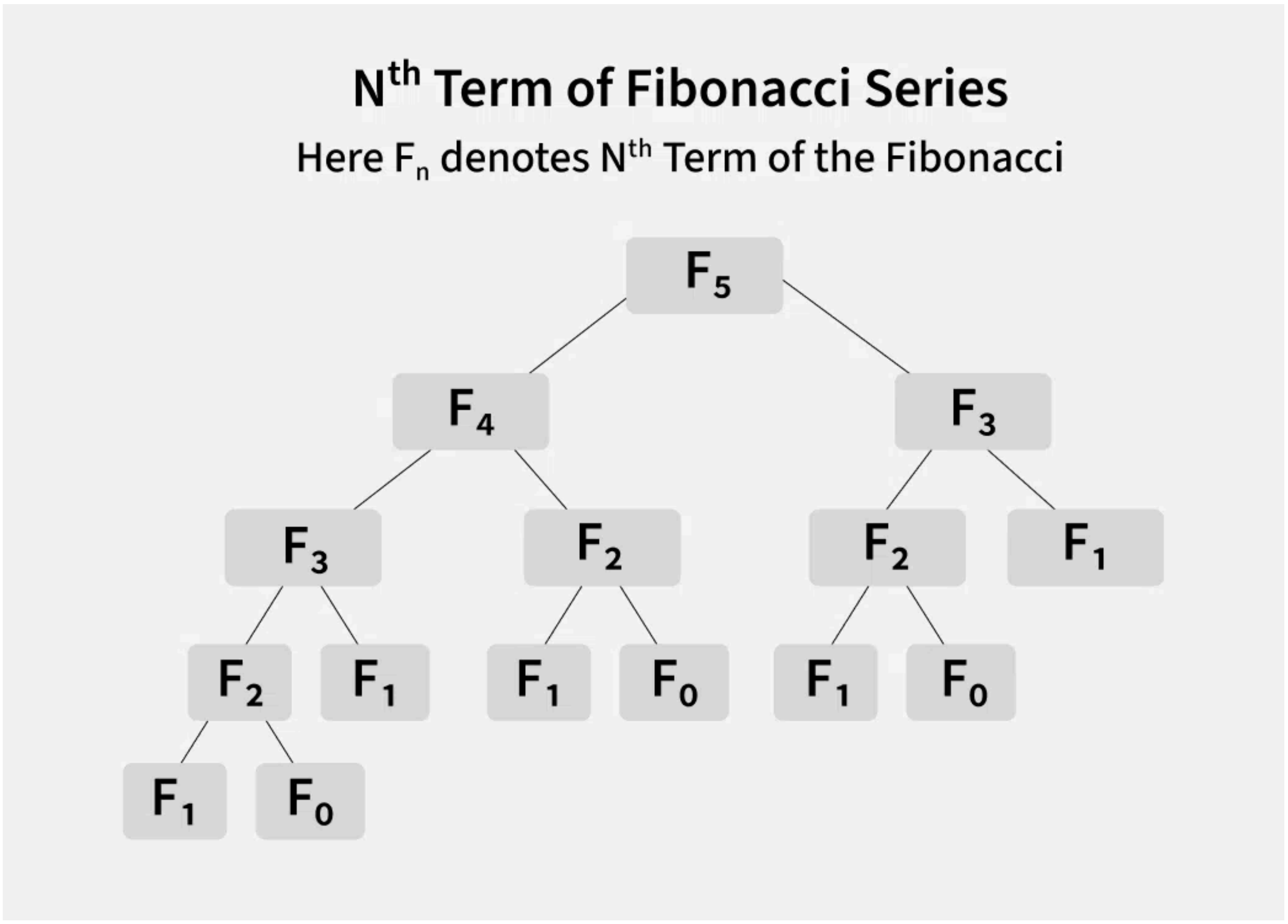
- Before making any recursive call, we first check if the memoization table already has solution for it.
- After the recursive call is over, we store the solution in the memoization table.

### 2. Bottom-Up Approach (Tabulation):

In the bottom-up approach, also known as tabulation, we start with the smallest subproblems and gradually build up to the final solution.

- We write an iterative solution (avoid recursion overhead) and build the solution in bottom-up manner.
- We use a dp table where we first fill the solution for base cases and then fill the remaining entries of the table using a recursive formula.
- We only use recursive formulas on table entries and do not make recursive calls.

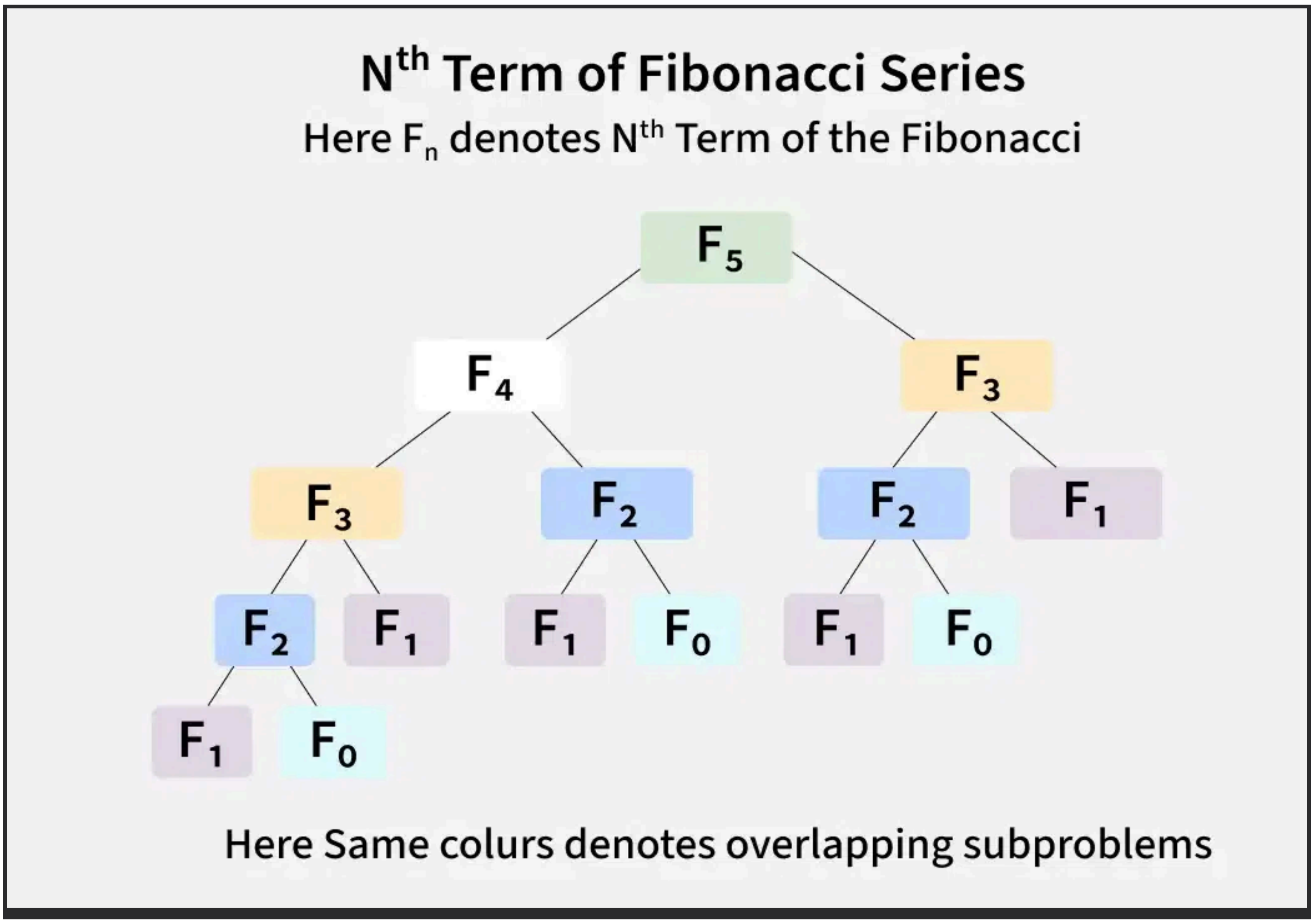
Below is the recursion tree of the above recursive solution.


$N^{th}$ Term of Fibonacci Series
Here $F_n$ denotes $N^{th}$ Term of the Fibonacci
$F_5$
$F_4$
$F_3$
$F_3$
$F_2$
$F_2$
$F_1$
$F_2$
$F_1$
$F_1$
$F_0$
$F_1$
$F_0$
$F_1$
$F_0$

The time complexity of the above approach is exponential and upper bounded by O(2n) as we make two recursive calls in every function.

## How will dynamic programming (DP) work?

Let us now see the above recursion tree with overlapping subproblems highlighted with the same color. We can clearly see that that recursive solution is doing a lot of work again and again, which is causing the time complexity to be exponential. Imagine the time taken for computing a large Fibonacci number.


N th Term of Fibonacci Series
Here $F_n$ denotes N th Term of the Fibonacci
$F_5$
$F_4$
$F_3$
$F_3$
$F_2$
$F_2$
$F_1$
$F_2$
$F_1$
$F_1$
$F_0$
$F_1$
$F_0$
$F_1$
$F_0$
Here Same colurs denotes overlapping subproblems

- Identify Subproblems: Divide the main problem into smaller, independent subproblems, i.e., F(n-1) and F(n-2)
- Store Solutions: Solve each subproblem and store the solution in a table or array so that we do not have to recompute the same thing again.
- Build Up Solutions: Use the stored solutions to build up the solution to the main problem. For F(n), look up F(n-1) and F(n-2) in the table and add them.
- Avoid Recomputation: By storing solutions, DP ensures that each subproblem (for example, F(2)) is solved only once, reducing computation time.

## Using Memoization Approach—O(n) Time and O(n) Space

To achieve this in our example, we simply take a memo array initialized to -1. As we make a recursive call, we first check if the value stored in the memo array corresponding to that position is -1. The value -1 indicates that we haven't calculated it yet and have to recursively compute it. The output must be stored in the memo array so that, next time, if the same value is encountered, it can be directly used from the memo array.

### Using Tabulation Approach—O(n) Time and O(n) Space

In this approach, we use an array of size (n + 1), often called dp[], to store Fibonacci numbers. The array is initialized with base values at the appropriate indices, such as dp[0] = 0 and dp[1] = 1. Then, we iteratively calculate Fibonacci values from dp[2] to dp[n] by using the relation dp[i] = dp[i-1] + dp[i-2]. This allows us to efficiently compute Fibonacci numbers in a loop. Finally, the value at dp[n] gives the Fibonacci number for the input n, as each index holds the answer for its corresponding Fibonacci number.

# Recursive Formula of Binomial Coefficient

The recursive formula for the binomial coefficient is based on Pascal's triangle, where each entry is the sum of the two entries directly above it.
Let n and k be integers such that $0 \le k \le n$. If k = 0 or k = n, then set (n, k) = 1. If $0 < k < n$, then set
B(n, k ) = B(n - 1, k - 1 ) + B(n - 1, k )
This formula is particularly useful for calculating binomial coefficients without needing factorials, as it reduces the computation to a series of additions.

## Proof of Recursive Formula for the Binomial Coefficient

We want to prove the recursive formula for the binomial coefficient:
To Prove: B(n, k ) = B(n - 1,  k - 1 ) + B(n - 1, k )
For integers n and k with $0 \le k \le n$.
Base case 1:
We begin by establishing the initial conditions.
If k = 0 or k = n, we have B(n, 0) = B(n, n) = 1. This indicates that there is exactly one way to choose zero elements or all n elements from a set of n elements.
Inductive Step:
Next, we consider the case where $1 \le k \le n - 1$.
Using the explicit formula for the binomial coefficient:

Binomial coefficients are positive integers represented as nCk where n >= k >= 0. The following are important properties of binomial coefficients.

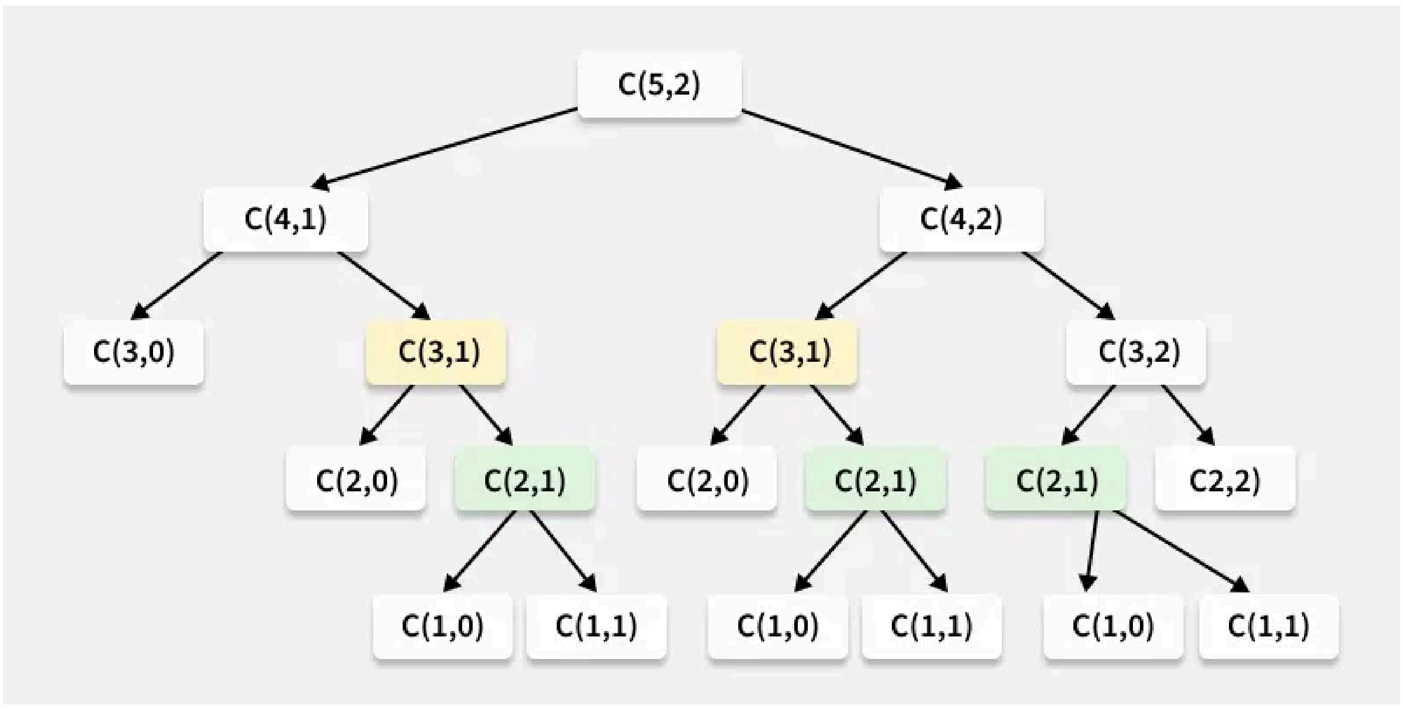
The value of nCk represents the number of possibilities to pick "k" items from n elements.

The formula for nCk is n! / {k! (n - k)!}, where ! represents factorial.

These binomial coefficients of the Binomial Theorem , which gives a formula for expanding statements such as (a + b)n = nC0 anb0 + nC1 an-1 b1 + nC2 an-2 b2 + … + nCr an-r br + … + nCn a0bn . For instance, the binomial coefficients in $(x + y)^3$ are 1, 3, 3, and 1.

Given integer values n and k, the task is to find the value of the binomial coefficient C(n, k).

- A binomial coefficient C(n, k) can be defined as the coefficient of x^k in the expansion of (1 + x)^n.
- A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.


C(5,2)
C(4,1)
C(4,2)
C(3,0)
C(3,1)
C(3,1)
C(3,2)
C(2,0)
C(2,1)
C(2,0)
C(2,1)
C(2,1)
C2,2)
C(1,0)
C(1,1)
C(1,0)
C(1,1)
C(1,0)
C(1,1)