05/01/2024

DYNAMIC PROGRAMMING
CLASS - 1

Dynamic Programming

📁 What is Dynamic Programming?

This is a technique to solve the problem janha recursion ko improve kar diya gya hai.

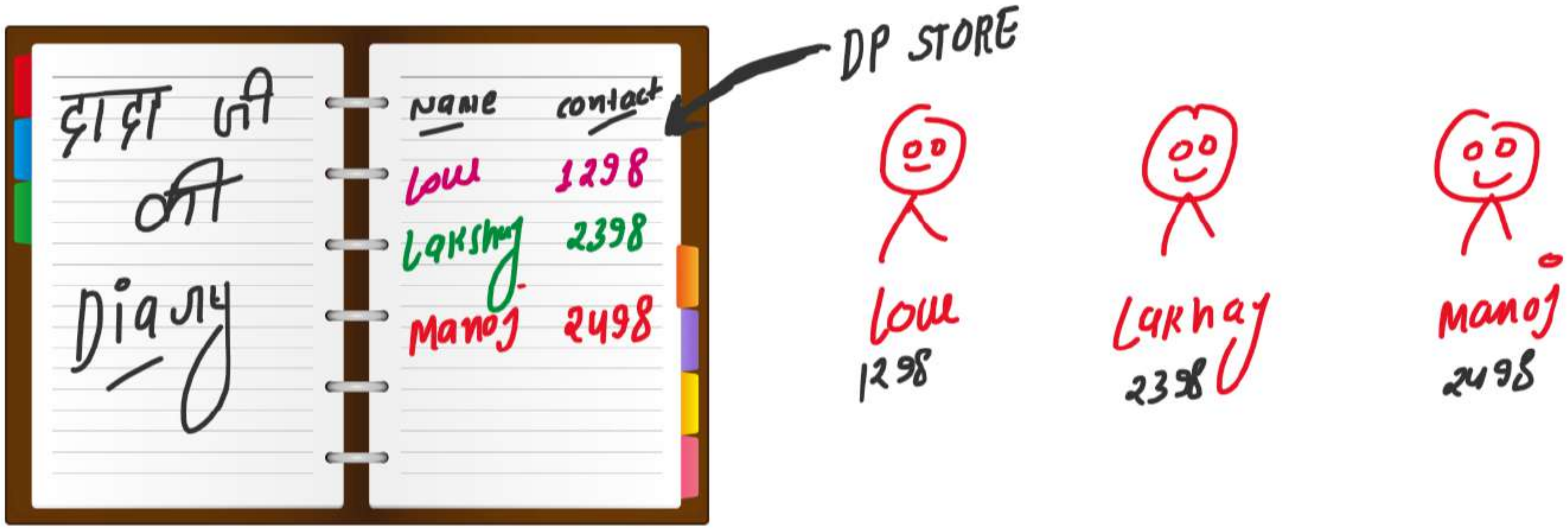
📁 Where to Use Dynamic Programming?

a. jav same subproblems overlapping (Reapeating) ho rahi ho

b. Jab ek badi problem ka optimized solution depend karta ho ek chotti problem ke optimized solution par.

📁 The Code Help Headline for DP:

Ek bar me problem ko solve karta hu fir use dovara solve nahi karta hu kyunki solved problem ka ans me store kar leta hu.



📁 Three Approaches for DP:

I. Top Down Approach (Memoization)

II. Bottom-up Approach (Tabulation)

III. Patterns Approach (Space Optimization)

I. Top Down Approach (Memoization)

Yanha hum recursion and memoization ka use krte hai with three steps as

Step 1: create DP array

Step 2: store ans and return ans using DP array

Step 3: if ans already exist then return ans

II. Bottom-up Approach (Tabulation Method)

Yanha hum iterative approach ka use krte hai naki recursion ka with three steps as

Step 1: create DP array

Step 2: fill initial data in DP array according to recursion base case

Step 3: fill the remaining DP array according to recursion formula/logic

III. Patterns Approach (Space Optimization)

Yanha space optimization karte hai according jab koi pattern ban rha ho

Find nth Fibonacci Number

To understand the these Approaches and overlapping subproblem and optimal solution also.

Find Nth Fibonacci Number

using Recursion

0	1	1	2	3	5	8	13	. . . . .
0th	1th	2th	3th	4th	5th	6th	7th	

$$f(N) = f(N-1) + f(N-2)$$

Recursive formula

Nth Fib NO = (N-1)th Fib NO + (N-2)th NO

Ex

Nth = 4

output = 2

Explanation

$4^{th} = 3^{th} + 2^{th}$

$= 2 + 1$

$4th = 3$

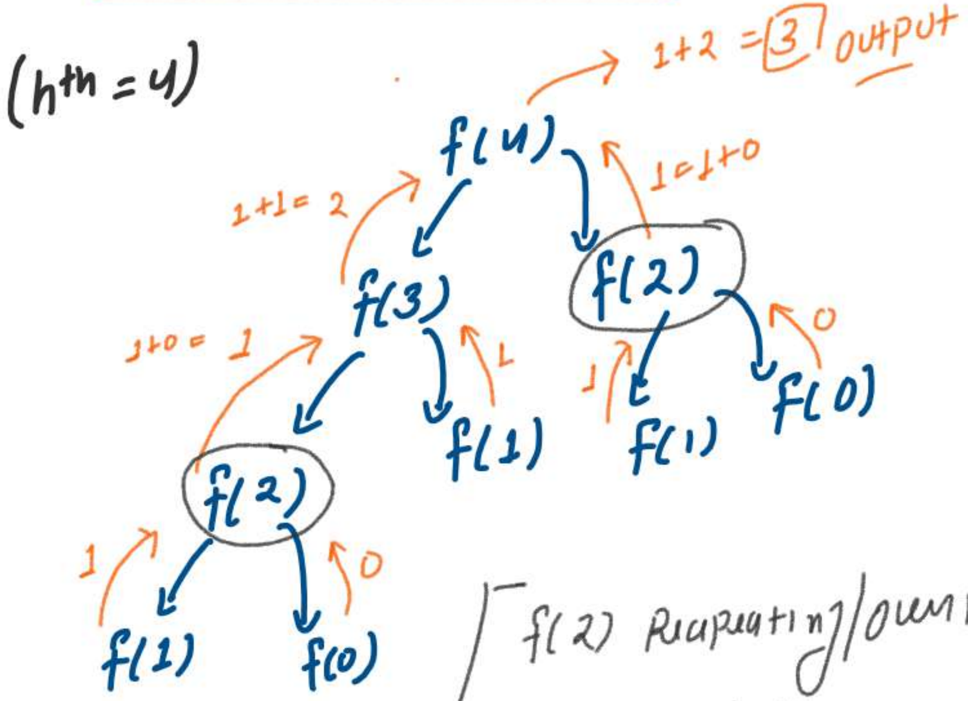
Base Cases

$f(0) = 0$

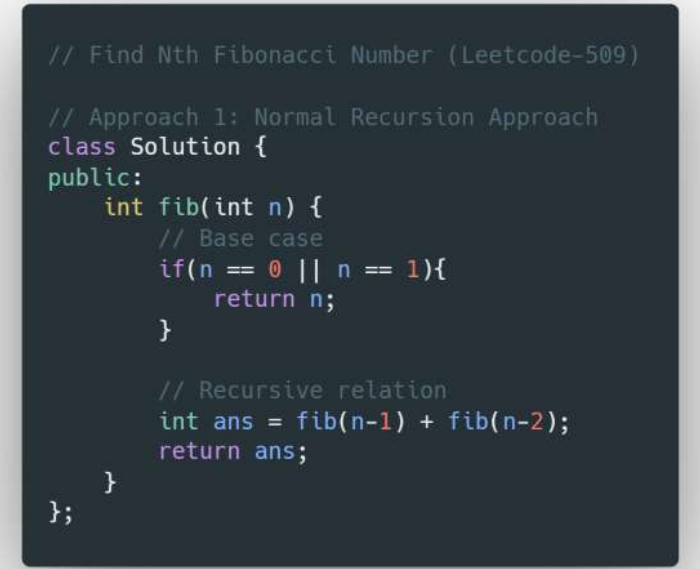
$f(1) = 1$

Normal Recursive call

(nth = 4)



f(2) Repeating/overlapping Two Times

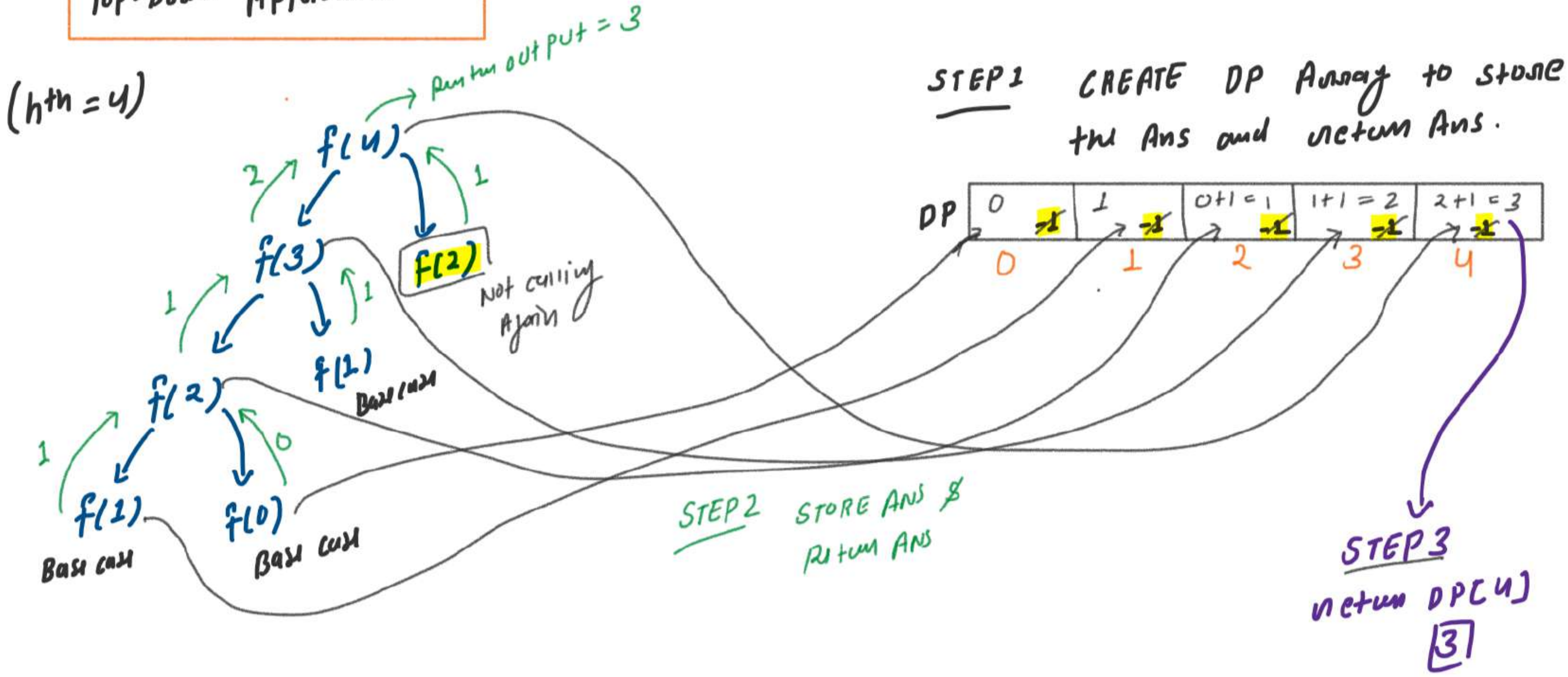


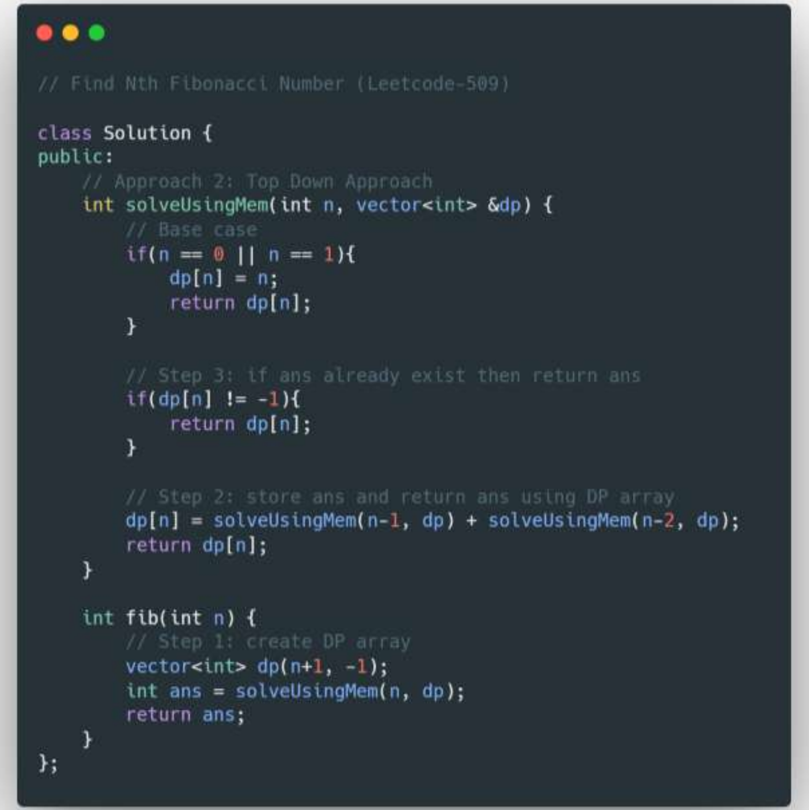
```cpp
// Find Nth Fibonacci Number (Leetcode-509)

// Approach 1: Normal Recursion Approach
class Solution {
public:
    int fib(int n) {
        // Base case
        if(n == 0 || n == 1){
            return n;
        }

        // Recursive relation
        int ans = fib(n-1) + fib(n-2);
        return ans;
    }
};
```

Top-Down Approach





```cpp
// Find Nth Fibonacci Number (Leetcode-509)

class Solution {
public:
    // Approach 2: Top Down Approach
    int solveUsingMem(int n, vector<int> &dp) {
        // Base case
        if(n == 0 || n == 1){
            dp[n] = n;
            return dp[n];
        }

        // Step 3: if ans already exist then return ans
        if(dp[n] != -1){
            return dp[n];
        }

        // Step 2: store ans and return ans using DP array
        dp[n] = solveUsingMem(n-1, dp) + solveUsingMem(n-2, dp);
        return dp[n];
    }

    int fib(int n) {
        // Step 1: create DP array
        vector<int> dp(n+1, -1);
        int ans = solveUsingMem(n, dp);
        return ans;
    }
};
```

TOP DOWN APPROACH

→ (MEMOIZATION + OPTIMAL REC call)

DisAdvantage

→ DP Array takes Extra memory space

→ Function call overhead Also takes Extra memory space.

Tabulation method

nth = 5

Fib. NO. ⇒ 0, 1, 1, 2, 3, 5, 8, ...

Base cases

nth = 0 } o/p = 0
nth = 1 } o/p = 1

STEP1 Create DP Array **size of DP = N+1**

DP	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

STEP2 Fill initial data in DP According to **B.C.**

DP	0	1	-1	-1	-1	-1
	0	1	2	3	4	5

Remaining ARRAY

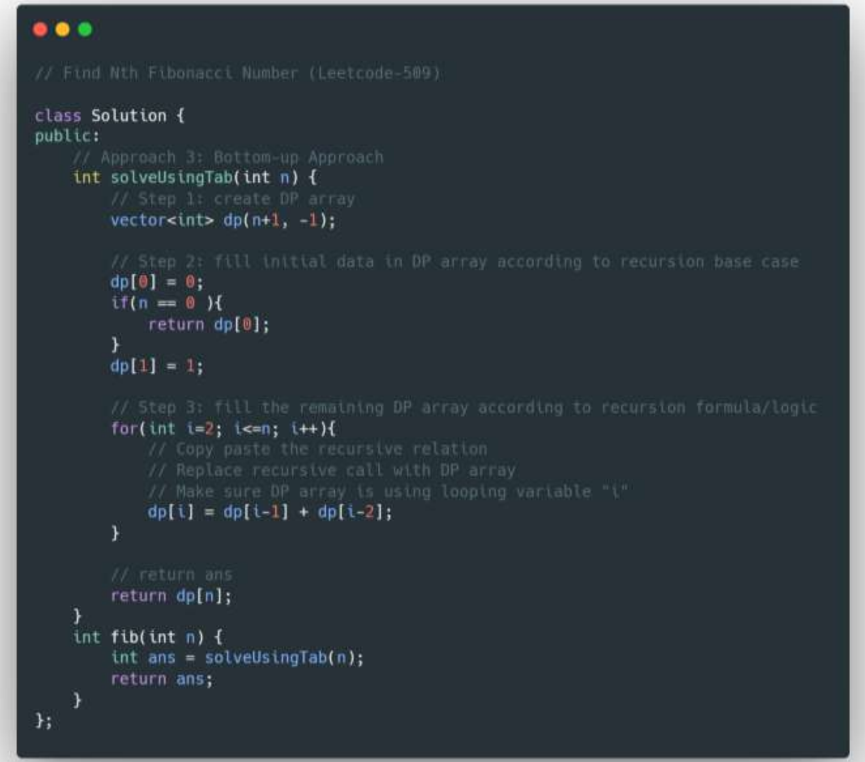
STEP3 fill remaining DP according to **recursion relation.**

Formula

$$DP[n^{th}] = DP[n-1]^{th} + DP[n-2]^{th}$$

DP	0	1	1	2	3	5
	0	1	2	3	4	5

OUTPUT



```cpp
// Find Nth Fibonacci Number (Leetcode-509)

class Solution {
public:
    // Approach 3: Bottom-up Approach
    int solveUsingTab(int n) {
        // Step 1: create DP array
        vector<int> dp(n+1, -1);

        // Step 2: fill initial data in DP array according to recursion base case
        dp[0] = 0;
        if(n == 0 ){
            return dp[0];
        }
        dp[1] = 1;

        // Step 3: fill the remaining DP array according to recursion formula/logic
        for(int i=2; i<=n; i++){
            // Copy paste the recursive relation
            // Replace recursive call with DP array
            // Make sure DP array is using looping variable "i"
            dp[i] = dp[i-1] + dp[i-2];
        }

        // return ans
        return dp[n];
    }
    int fib(int n) {
        int ans = solveUsingTab(n);
        return ans;
    }
};
```

Bottom-UP Approach

(Tabulation + iterative method)

Dis-Advantages $\rightarrow$ Take Extra $\overset{\text{DP Array}}{\text{space}}$ only

Advantages $\rightarrow$ No Function overhead

PATTERNS (SPACE OPTIMIZATION)

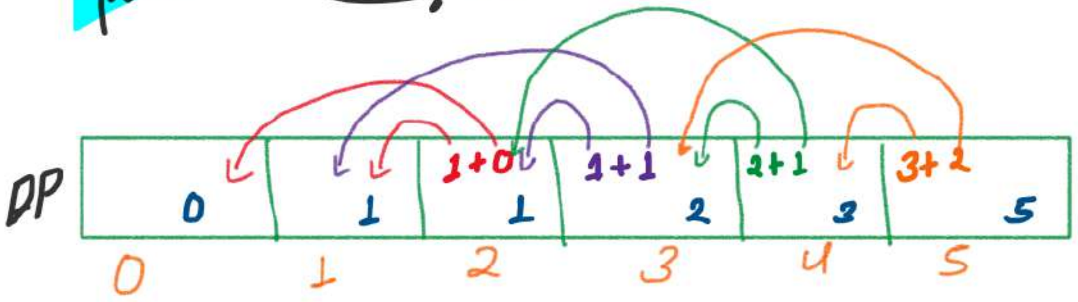
$\boxed{n = 5}$

REC FORMULA REPLACE with Tabulation

$DP[i] = DP[i-1] + DP[i-2]$

Pattern → Ans = prev + next

→ Check karo kya koi pattern ban rha hai?

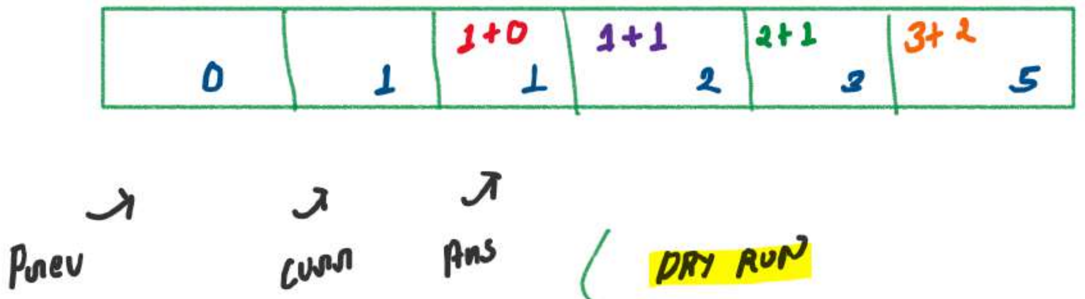


YAHA EK pattern create ho rha hai

int prev = 0
int curr = 1
int Ans = prev + curr

$\boxed{n+n=5}$

DRY RUN



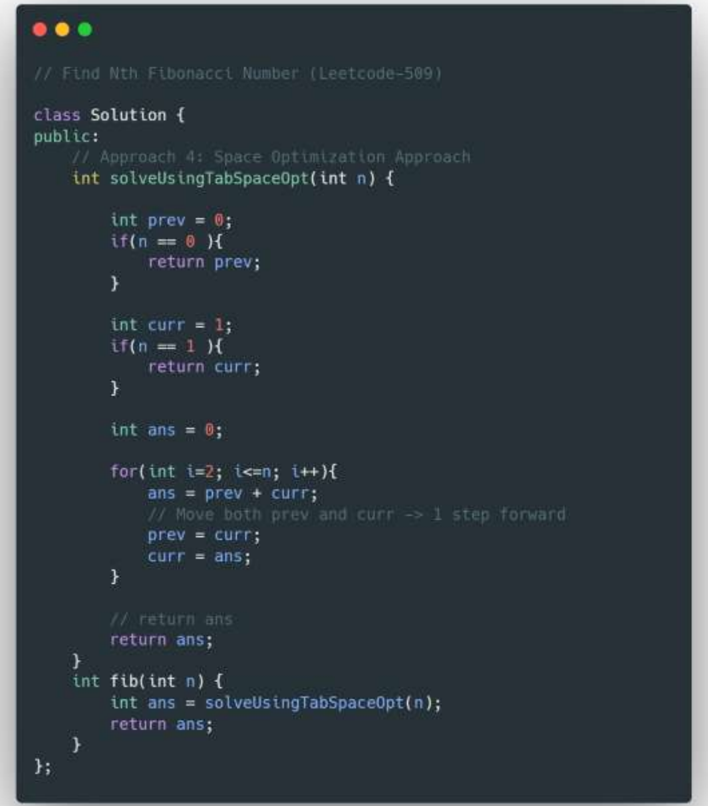
DRY RUN

Prev	Curr	ANS	Prev = Curr	Curr = Ans
0	1	0+1 = 1	1	1
1	1	1+1 = 2	1	2
1	2	1+2 = 3	2	3
2	3	2+3 = 5		

Yauha Galti Hoti Hai

** Updating Prev and count

→ STOP $\boxed{\text{indx} <= 5}$



```cpp
// Find Nth Fibonacci Number (Leetcode-509)

class Solution {
public:
    // Approach 4: Space Optimization Approach
    int solveUsingTabSpaceOpt(int n) {

        int prev = 0;
        if(n == 0 ){
            return prev;
        }

        int curr = 1;
        if(n == 1 ){
            return curr;
        }

        int ans = 0;

        for(int i=2; i<=n; i++){
            ans = prev + curr;
            // Move both prev and curr -> 1 step forward
            prev = curr;
            curr = ans;
        }

        // return ans
        return ans;
    }
    int fib(int n) {
        int ans = solveUsingTabSpaceOpt(n);
        return ans;
    }
};
```

T.C. $\Rightarrow O(N)$

S.C. $\Rightarrow O(1)$

Where N is input size.