If you don't remember the past,
You are condemned to repeat it.

Fibonacci Series.

index :	1	2	3	4	5	6	7	8
value :	0	1	1	2	3	5	8	13

Q Find n^{th} Fibonacci Number !

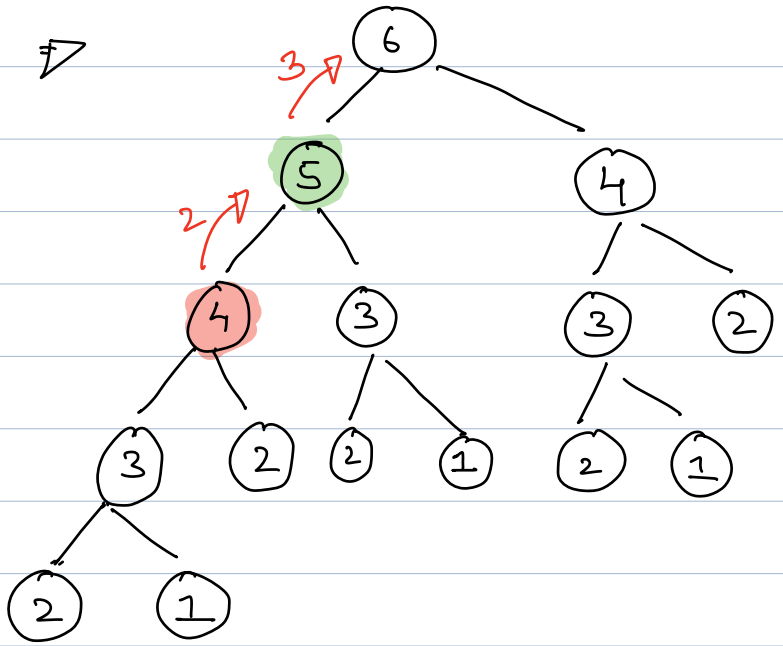
Recurrence Relation : $F(n) = F(n-1) + F(n-2)$

Pseudo Code

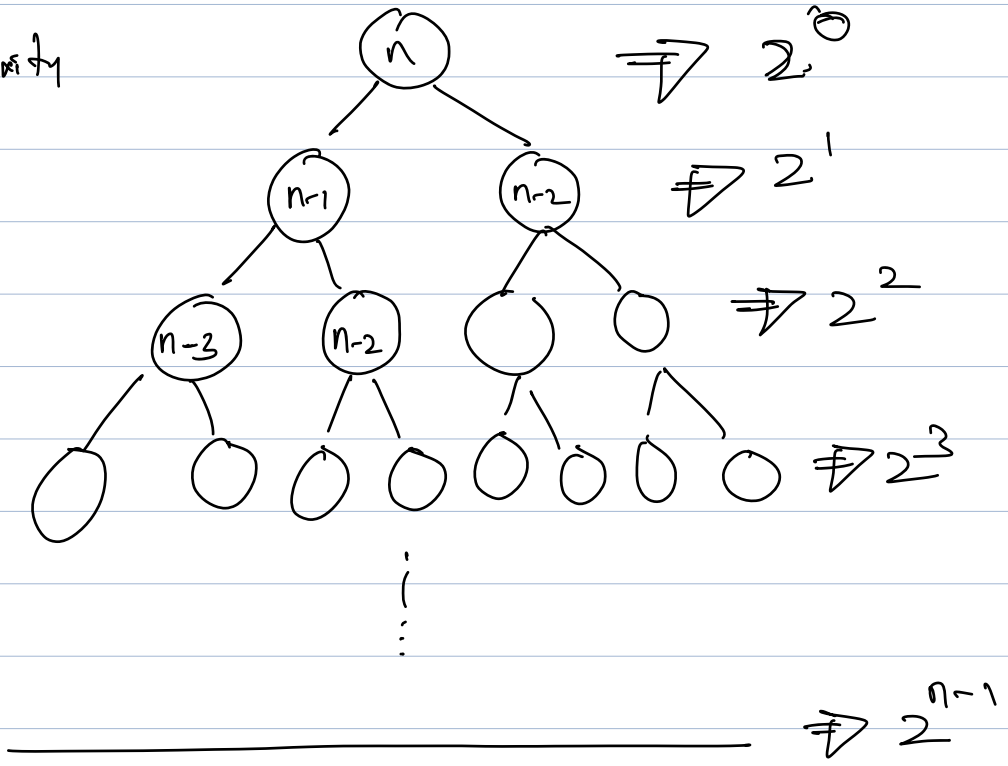
```
int Fib (int n) {
    if (n <= 2)
        return (n-1) ;

    return Fib(n-1) + Fib(n-2);
}
```

Fib(6) ⇒



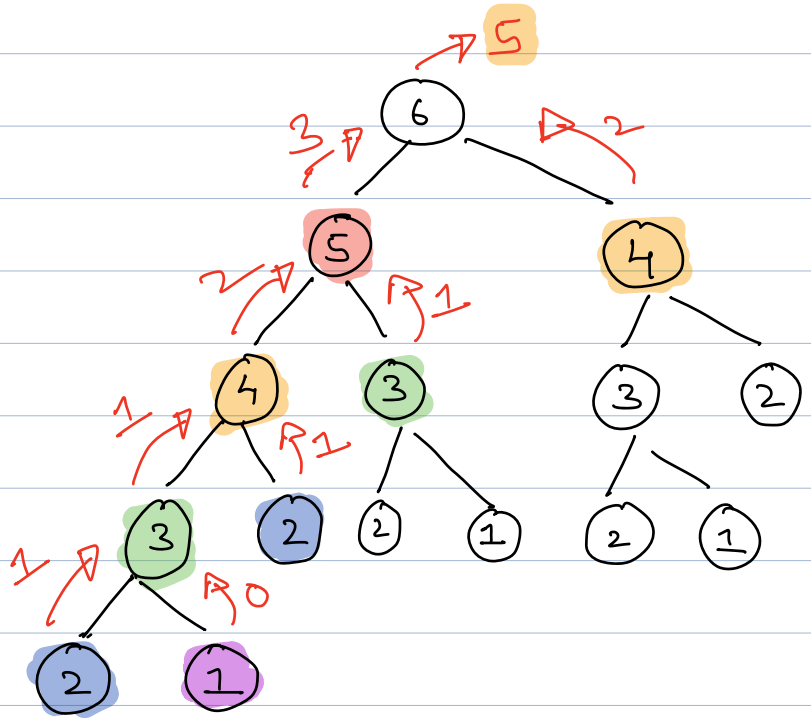
Time Complexity



Total func. calls ⇒ 2^n

Time taken by each call ⇒ $O(1)$

TC: $O(2^n)$ SC: $O(n)$



0	1	2	3	4	5	6	7
-1	0	1	1	2	3	5	-1

Total function calls $\simeq 2n$

$TC = O(n)$

$SC = O(n)$

When can DP be used ?

1) Optimal Substructure

If you can find optimal solution to a problem using optimal solutions of smaller problems (sub-problems)

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

Sum(n)
↓
Sum(n-1)
↓
Sum(n-2)

2) Overlapping sub-problems.

Repitition of sub-problems

Total Func calls > Total Unique fun calls

Memoization

↳ Storing results of states (sub-problems) to use them in future.

Pseudo Code (Fibonacci using DP)

```
int dp[n+1] = {-1} , dp[1] = 0 , dp[2] = 1;
```

initialize array — fill base values.

```
int fib (int n) {

    if (dp[n] != -1)
        return dp[n];

    dp[n] = fib(n-1) + fib(n-2);
    return dp[n];

}
```

Check if previously computed.

find the answer & store it.

return answer

Two Ways of solving a DP Problem

Top Down	Bottom up.
→ recursive Code	→ iterative code
→ More space would be needed	→ More control on Space.
→ Easier to write	→ Not as Easy as Top down to code
→ memoization is used.	→ Tabulation.

Bottom up_ fibonacci

```
int dp[n+1] = {-1} , dp[1] = 0 , dp[2] = 1

for (int i=3; i<=n; i++) {

    dp[i] = dp[i-1] + dp[i-2];

}

return dp[n];
```

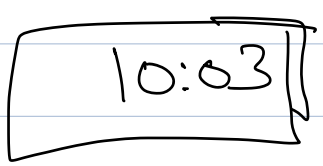
Tc : O(n)

Sc : O(n)

↓

O(1) using

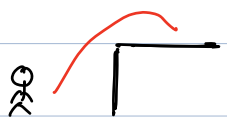
3 variables.



2 variables.

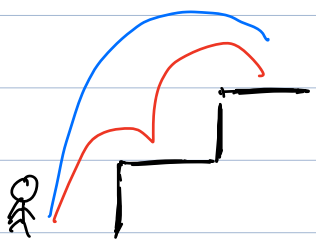
Q2 Given N steps.
Find the no of ways of reaching the N^{th} stair. At a time, you can climb 1 stair or 2 stairs

Ex1 n = 1



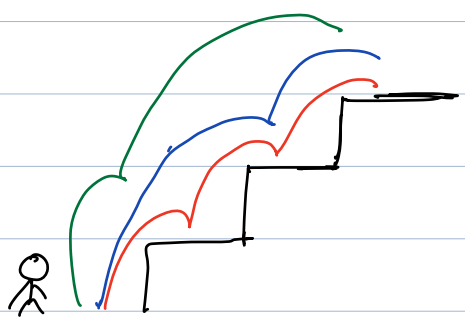
1 way

Ex2 n = 2

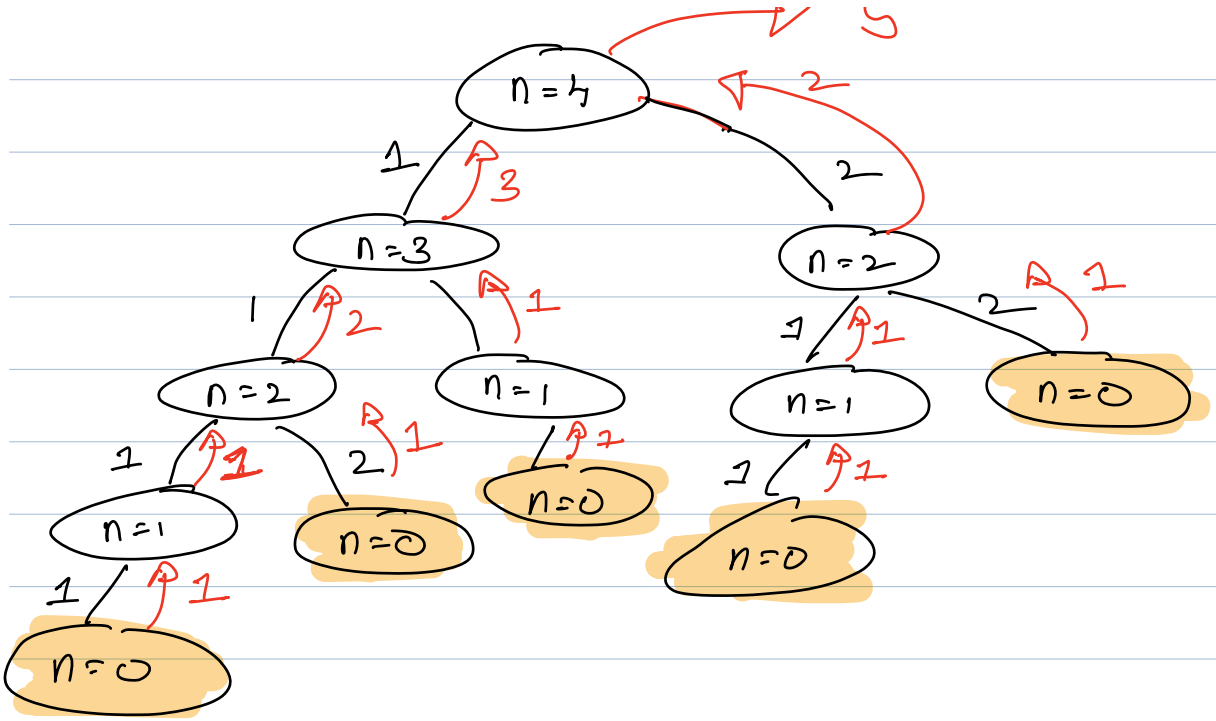


2 ways

Ex3 n = 3



3 ways



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

Q3 Find minimum number of perfect squares to get sum = N.

Ex1 N=6 = $1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2$

$2^2 + 1^2 + 1^2$

Ex2 N=10 = $3^2 + 1^2$

Greedy N - { nearest perfect Square }.

Ex3 N=12

12 - 9 = 3

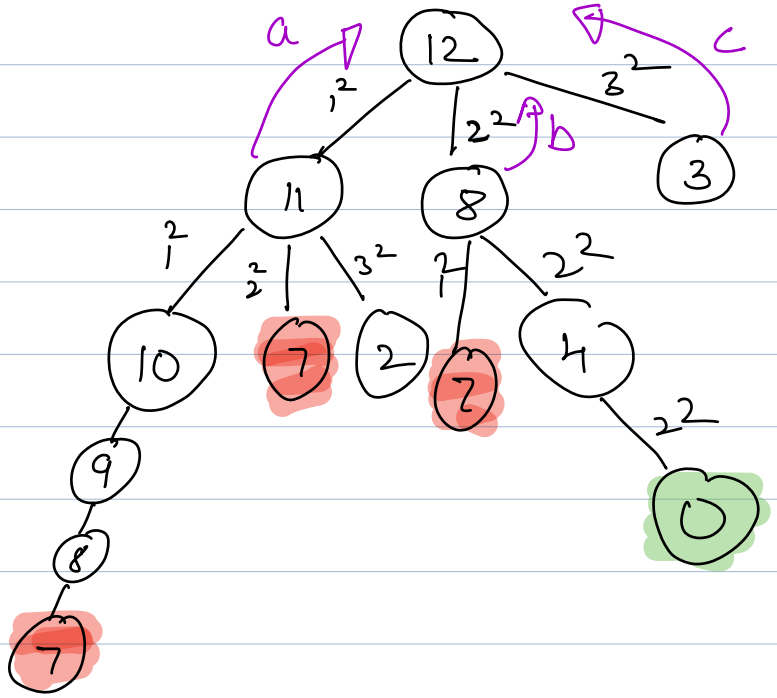
3 - 1 = 2

2 - 1 = 1

1 - 1 = 0

ans = 4

12 = $2^2 + 2^2 + 2^2$ ans = 3



$\Rightarrow \min(a, b, c) + 1$

$$\text{min\_sq}[12] = \min \left\{ \begin{array}{l} \text{min\_sq}[12 - 1^2], \\ \text{min\_sq}[12 - 2^2], \\ \text{min\_sq}[12 - 3^2] \end{array} \right\} + 1$$

$$\text{min\_sq}[i] = \min \left\{ \text{min\_sq}[i - x^2] \right\} + 1$$
$$\forall\, x^2 \leq i$$

Top Down.

```
min_sq [n+1] = {-1} , min_sq [0] = 0;

int p_sq (int n) {
    if ( min_sq [n] != -1)
        return min_sq [n];

    int ans = INT.MAX;

    for (int i = 1 ; i*i <= n ; i++ ) {
        ans = min ( p_sq (n - i*i), ans )
    }

    min_sq [n] = ans + 1;

    return min_sq [n];
}
```

TC : $O(N\sqrt{N})$

SC : $O(N)$

Bottom up!

```
min_sq[n+1] = {-1}, min_sq[0] = 0;

for(int i = 1; i <= n; i++) {

    ans => INT.MAX

    for(int k = 1; k*k <= i; k++) {

        ans => min(min_sq[i - k^2], ans);

    }

    min_sq[i] = ans + 1;

}
```

TC: $O(N\sqrt{N})$

SC: $O(N)$