Dynamic Programming

↳ problem solving technique to design algorithm that perform [recursion without repetition]

↳ how? stores the result of [intermediate sub-problems in a tuple/ array.]

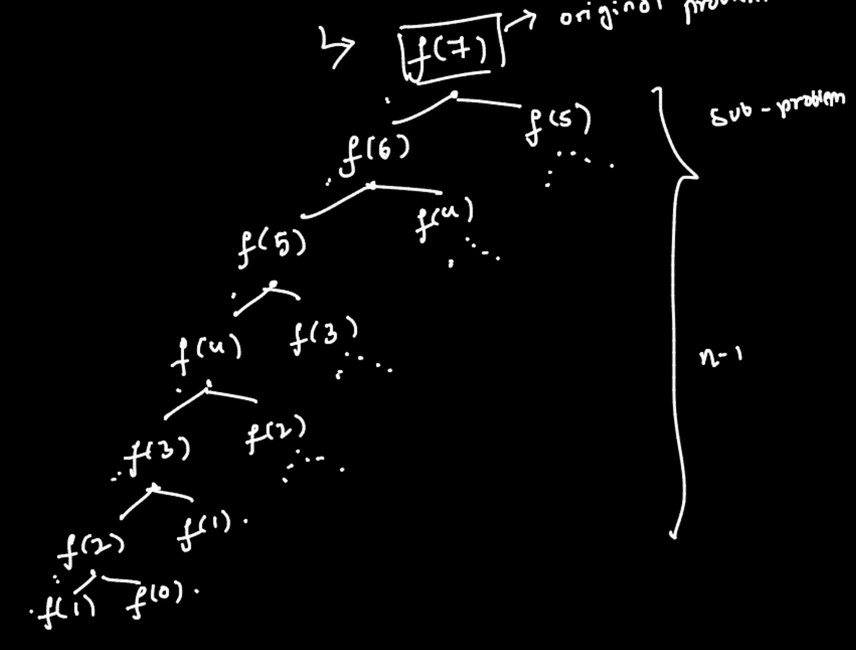
$f(n) = f(n-1) + f(n-2)$ ⇒ recursive calc

$f(0) = 0$ } Base call

$f(1) = 1$ }

[code]

↳ $f(7)$ → original problem



↳ Back Substition

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

↳ # rec. call

$\leq 2T(n-1)$

$\leq 2(2T(n-2))$

$= 2^2 T(n-2)$

$\leq 2^2(2T(n-3))$

$= 2^3 T(n-3)$

$\vdots$

$2^k T(n-k)$

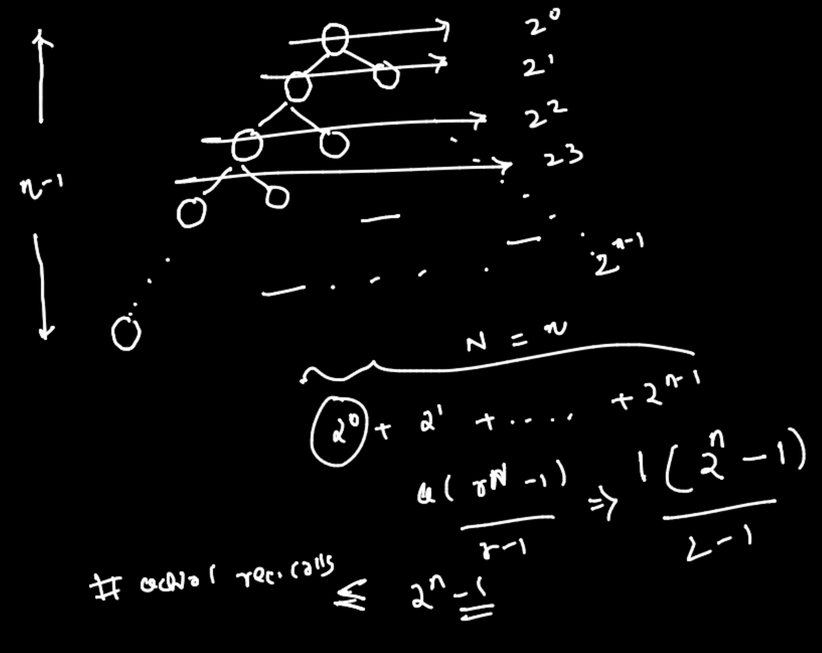
$n-k=0$

$k=n$

$T(n) = 2^n T(0)$

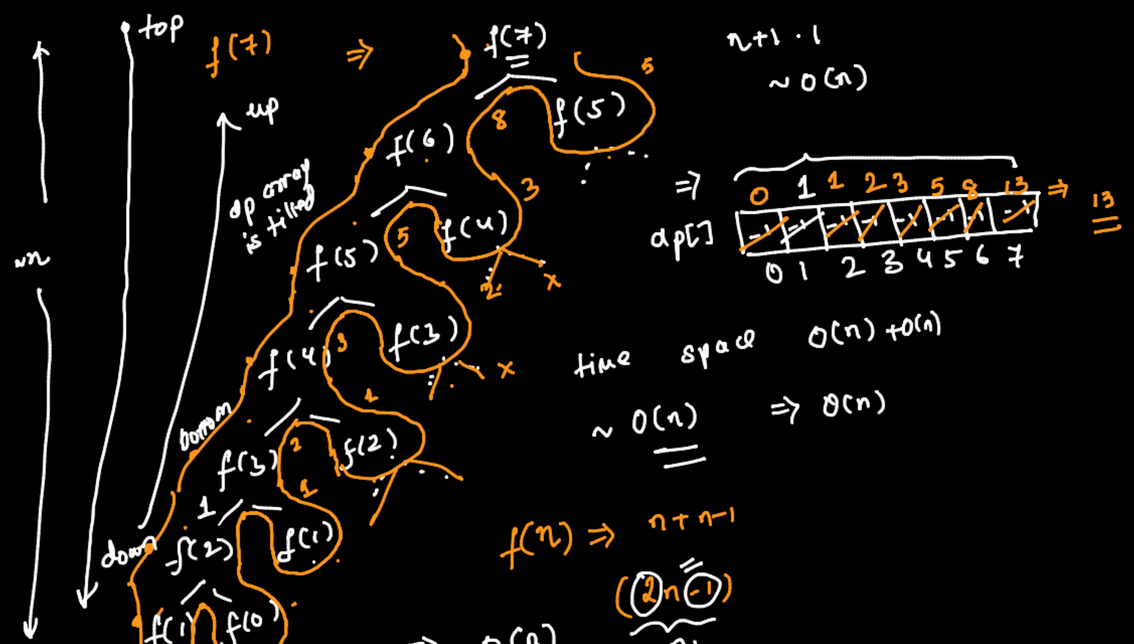
$= 2^n$

$= 2^n \cdot O(1) \sim O(2^n)$



$\leq \sim 2^n$

$O(2^n)$



why ?

time
$\Rightarrow O(2^n) \longrightarrow O(n)$ ↑
space $O(n) \longrightarrow O(n+n) \sim O(n)$ same

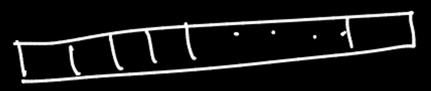
Top Down $\Rightarrow$ [recursion + memoization]
↳ table

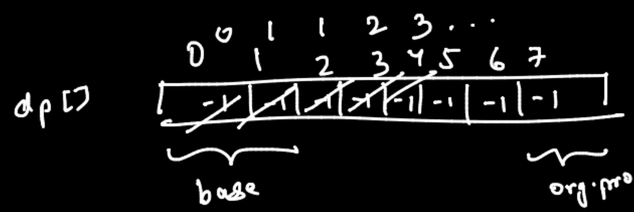
↳ why not directly fill the array starting from the base-case and going towards the org. problem in an iterative fashion

⇓

avoid the recursion completely.

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





dp(2) = dp(1) + dp(0)

⇓

f(2) = f(1) + f(0)

dp(3) = dp(2) + dp(1)

⇓

f(3) = f(2) + f(1)

dp(4) = dp(3) + dp(2)

⇓

f(4) = f(3) + f(2)

dp(i) = dp(i-1) + dp(i-2)

⇓

f(i) = f(i-1) + f(i-2)

f(n) → denotes min steps it takes to reduce n to 1

f(n) = 1 + min(x, y, z)

rec. code

f(n) = 1 + min(f(n-1), f(n/2), f(n/3))

f(1) = 0 ⇒ Base case