Dynamic Programming

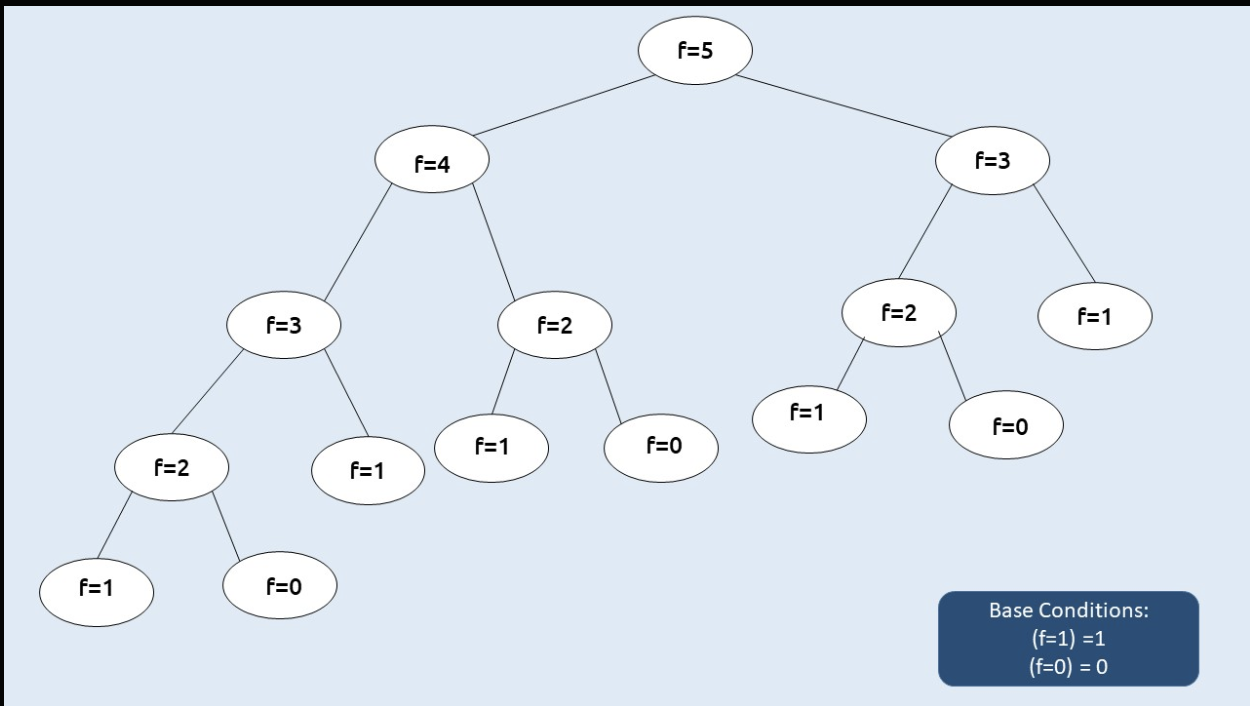
Why and when to use dynamic programming?

Feels like every dp can be solved using recursion then why do we need dynamic programming

- Yes, every sum of dp can be solved using dp but lets look at an example

- Take a fibonacci example, so the f(n) = f(n-1) + f(n-2)

- so if we follow recursive way we keep on checking the previous sums like this



- We are calculating the f2, f1,f0 ,f3 several times, but if we have to do it can we find an optimal way ?

- Yes, that’s where the dynamic programming coming into the play

- so instead of calculating them again and again we can calculate it once save it and when needed we can just call that instead of performing that calculation again.

- **Use dynamic programming : when we have to perform on overlapping sub problems.**

- So when we have to perform on overlapping subproblems instead of recursion which will lead to time exceeding the limit. so we have to use dynamic programming.

## Methods of Dynamic Programming

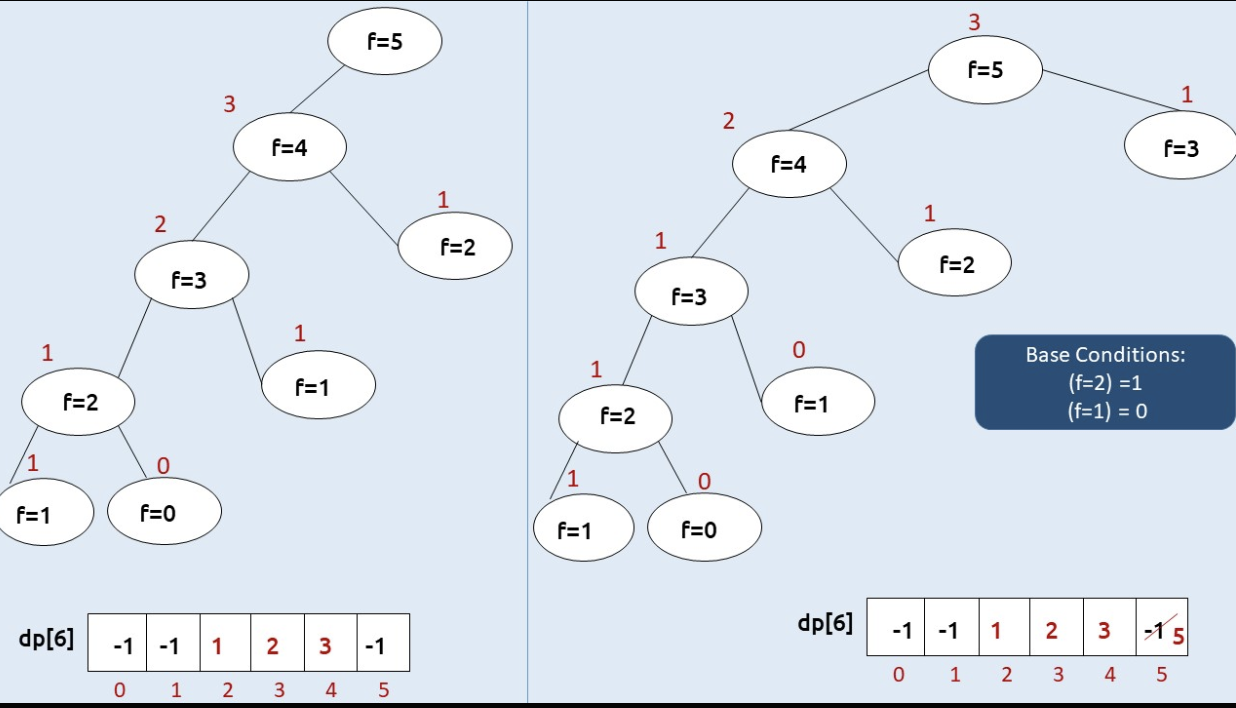
* Memorization
* Tabulation

## Memorization

Memorization is a top-down approach, so like it starts from f(5) goes to f(4), f(3)... so on, while gng saves in the array

Steps of memorization

* Create a dp[n+1] array initialized to -1.
* Whenever we want to find the answer of a particular value (say n), we first check whether the answer is already calculated using the dp array(i.e dp[n]!= -1 ). If yes, simply return the value from the dp array.
* If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[n] to the solution we get.



* Time complexity : O(n) each subproblem takes O(1) to find the value or calculating it for n problems so the complexity is O(n)
* Space complexity : each subproblem recursively stack pace, goes down we do it as a stack array which is O(n), and the dp array of O(n) total 2O(n) -> O(n)
* def f(n, dp): # dp = [-1] \* (n+1)
* if n <= 1:
* return n
* if dp[n] != -1:
* return dp[n]
* dp[n] = f(n-1, dp) + f(n-2, dp)
* return dp[n]

## Tabulation

Tabulation is a bottom-top approach, so instead of the stack space of saving we can start from the bottom like starting with f(1), f(2), then we can calculate f(3),, and so on .

Steps to convert Recursive Solution to Tabulation one.

* Declare a dp[] array of size n+1.
* First initialize the base condition values, i.e i=0 and i=1 of the dp array as 0 and 1 respectively.
* Set an iterative loop that traverses the array( from index 2 to n) and for every index set its value as dp[i-1] + dp[i-2].

def main():

n = 5

dp = [-1]\*(n+1)

dp[0] = 0

dp[1] = 1

for i in range(2, n+1):

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

def main():

n = 5

prev2 = 0

prev = 1

for i in range(2, n+1):

cur\_i = prev2 + prev

prev2 = prev

prev = cur\_i

print(prev)

Time complexity : O(n)

Space complexity : O(n)

Time complexity : O(n)

Space complexity : O(1)

* To return the whole dp array then we have to do like this but if we just want a particular value we can adjust n like which f(n) we want give n that and we don’t need to save all the values in the array we only need to save the previous two array values

**Note:**- If we have only one deciding factor like here we just want the f(4) smg like that so we can get the dp[4] so in this case we can use array

* But if we have two deciding factors then we can use matrix so we can retrieve dp[col][row] like that
* Based on the code, we decide if we want an array or matrix or smg else based on that.

-Sometimes we can solve this using **greedy algorithms,** based on the task think if the greedy works or not, if not then go for recursive and dp