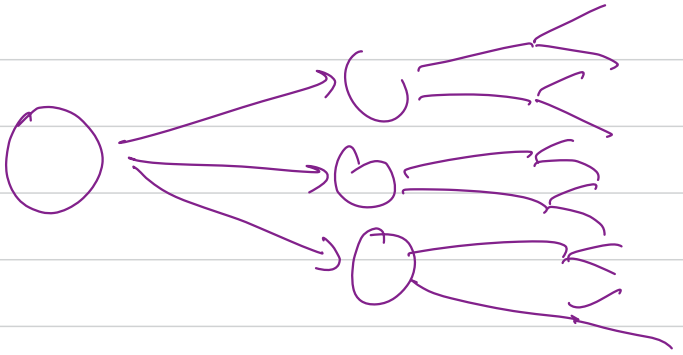
Dynamic Programming

→ Interviews

→ CP

Algorithms

- Brute force
- Divide N Conquer
- Greedy → local optimum
- Dynamic Programming → global optimum



→ Pre-requisite → Recursive logics

↓

Recurrence Relations

↓

Problems / Sub problems

↓

arrays

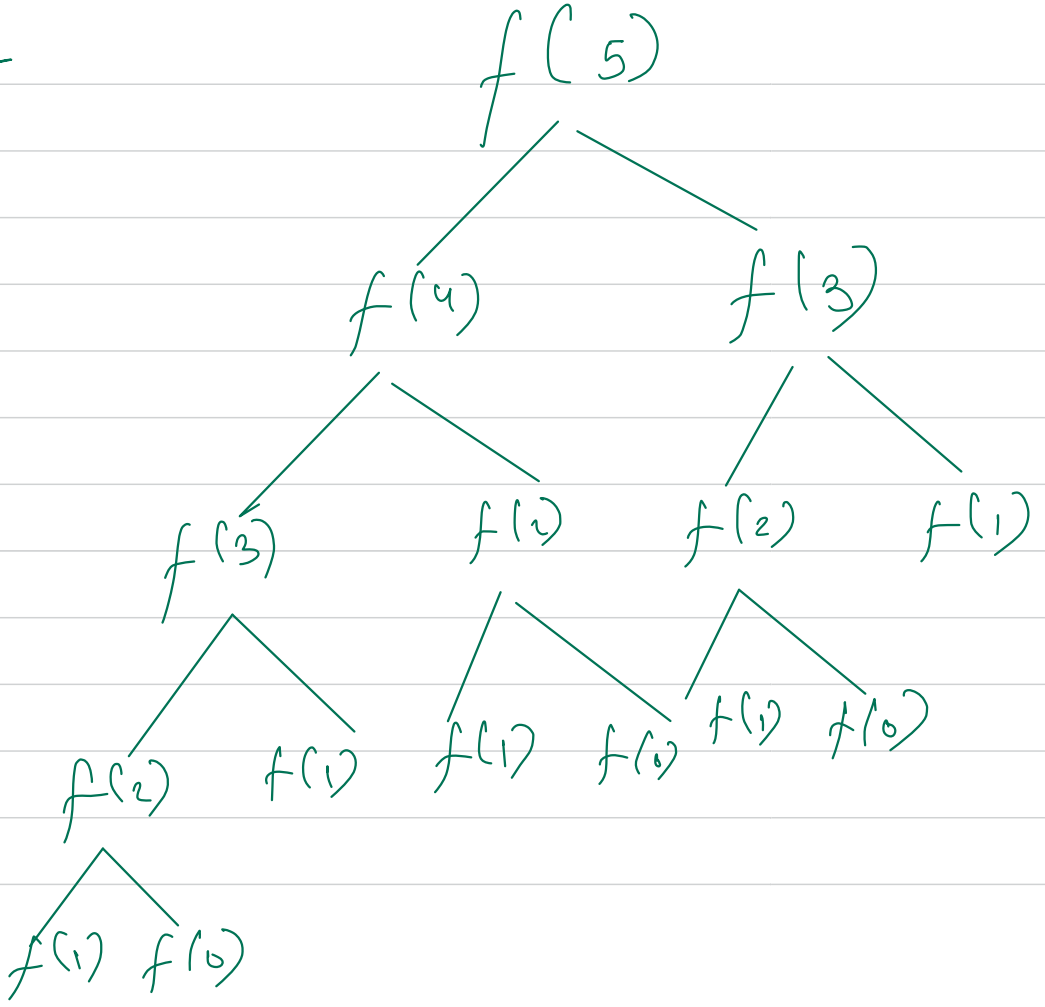
A lot of times

Problem in hand

- It's very easy to solve a sub problem first & then compute ans for original problem

↳ In our solutions, we try to repeatedly calculate answers of already computed problems & subproblems

fibonacci



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

1) Mutual Exclusiveness →

2) Mutual Exhaustiveness →

→ How to see if we need dp Solution??

(1) Repeated Subproblem / Overlapping Subproblem

(2) Optimal Substructure

↳ If we want to compute optimal answer for a bigger problem, then if we have optimal ans for the subproblem & they contribute optimally to the bigger problem s.t. that the optimal ans for big problem can be constructed from optimal ans of subproblems.

2 types

memoization

Tabulation

[Top Down]

[Bottom Up]

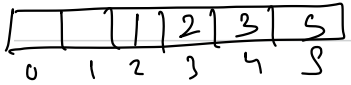
You start from a bigger subproblem and then move to a smaller one & come back

generally → Recursion

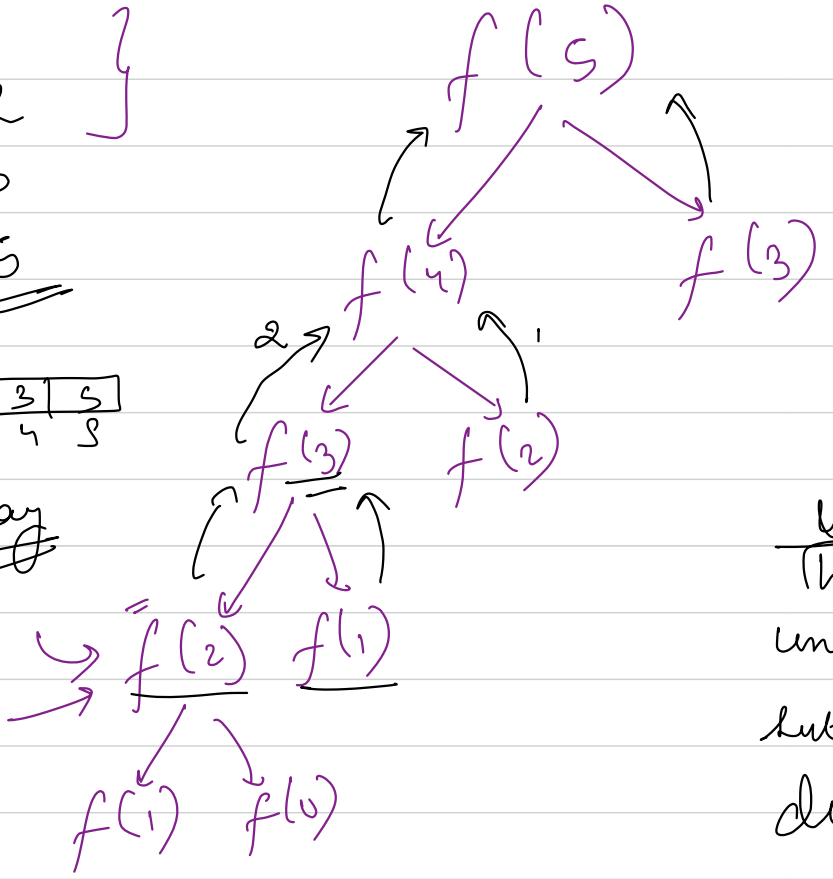
You start from the smallest subproblem & go to the bigger one

generally → iteration

$f(2) \to 1$
$f(3) \to 2$
$f(4) \to 3$
$f(5) \to 5$



1D array



State of the dp

1 unique subproblem

The no of parameters uniquely defining the subproblem gives us the dimension of the DP

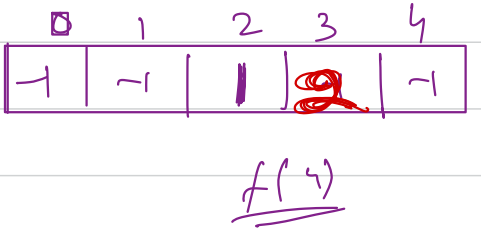
Size of dp - atleast equal to no. of uniq states

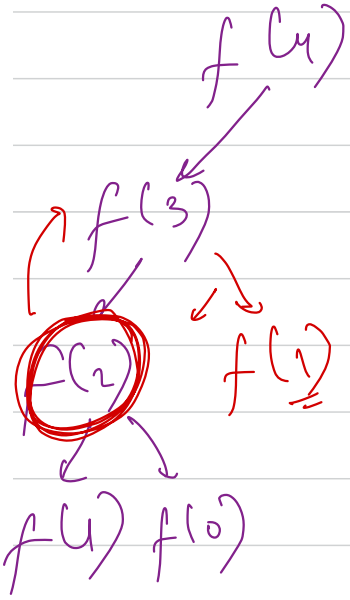
1D DP 2D DP - - .

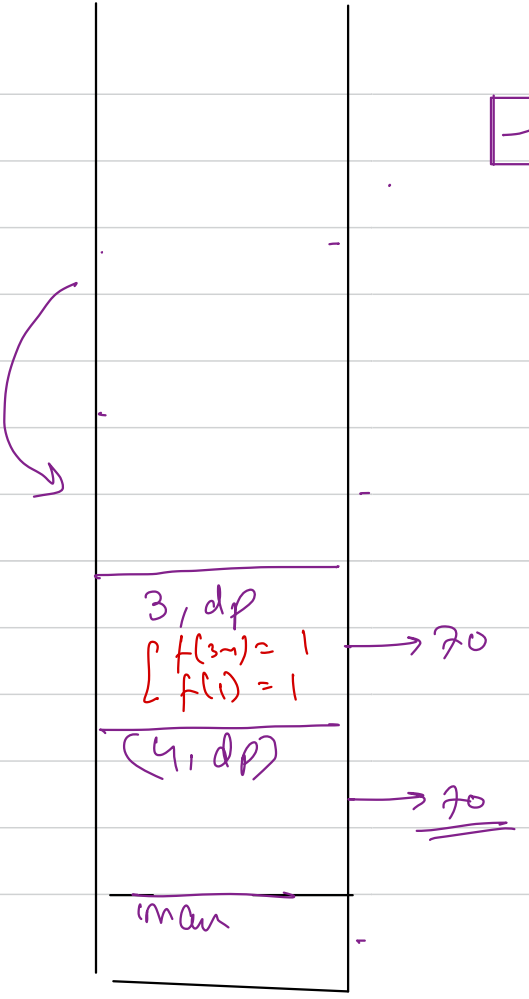
dp[n] → n^{th} fibonacci

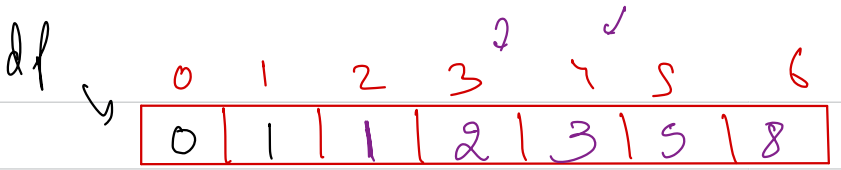
f(n)

```cpp
// top down dp
int fibTD(int n, vector<int> &dp) {
    if(n == 0 or n == 1) return n;
    if(dp[n] != -1) {
        // already computed
        return dp[n];
    }
    dp[n] = fib(n-1) + fib(n-2);
    return dp[n];
}
```









Base Case

f(6)

dp[i] → f(i)

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

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

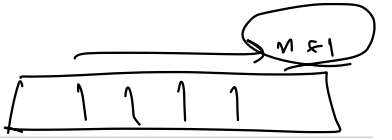
f(i) = f(i-1) + f(i-2)

```
c = a+b
a = b
b = c
```

→ fab → BUOP → SC → O(1)

↳ school

TD → TC → O(n)
SC → O(n)



→ O(n + n)

BU → TC → O(n)
SC → O(n)

Recursive Soln → TC → $O(2^n)$
SC → O(n)

You are given money present in n adjacent houses. There is a robber who wants to rob n houses but he can never rob from 2 adjacent houses. Find max loot of robber.

2, 7, 9, 3, 1

↑ ↑ ↑

2 → ans → 12

6, 2, 3, 9 → 15

pick not pick

$$f(i) = \max\left(arr[i] + f(i-2),\ f(i-1)\right)$$

↙ that denotes the max loot done till the i^{th} house