Recursive | Iterative DP

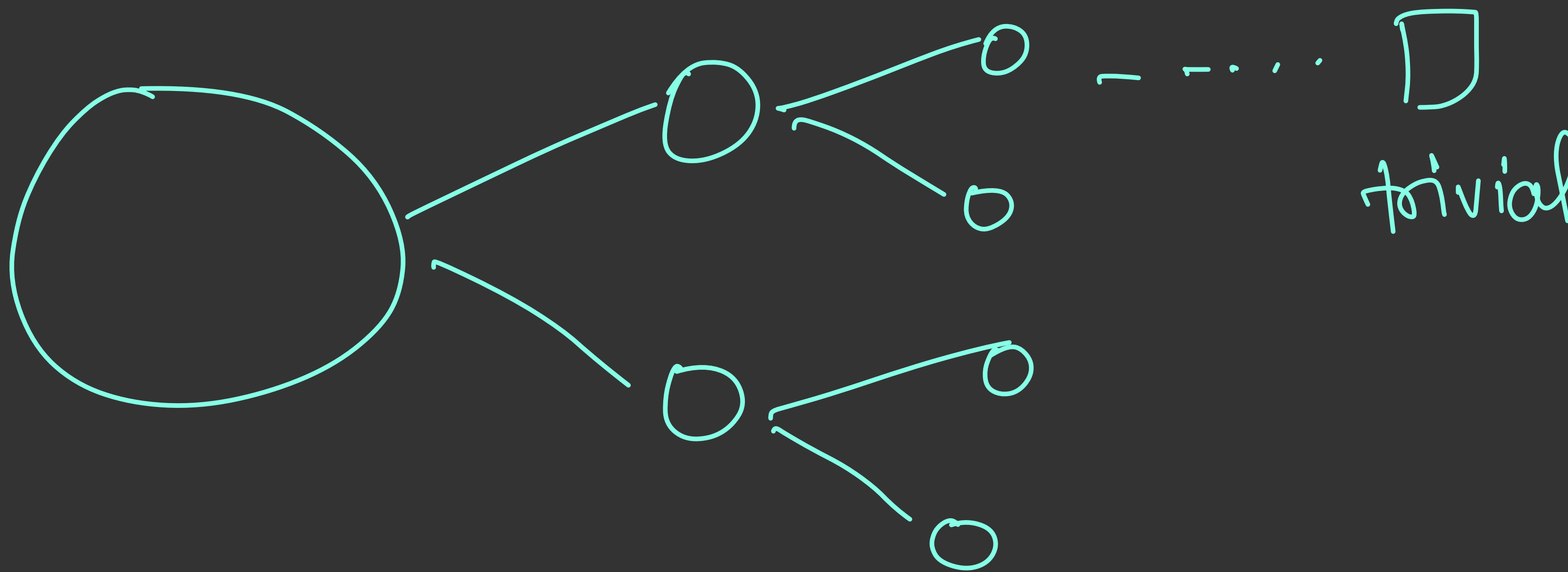
General Technique to solve all DP problems

Dynamic Programming 2

2 problems

- Priyansh Agarwal

Divide & Conquer mindset



relation b/w smaller subproblems to find out the answer for bigger problems

Dynamic Programming

- making sure that answer to a subproblem is not calculated more than once

① fibonacci problem

② Grid problem

state

transition

meaning of the state

Time & Space Complexity

T.C = # of states × transition time per ~~st~~ state

total transition time of all states

S.C = # of states × space per state

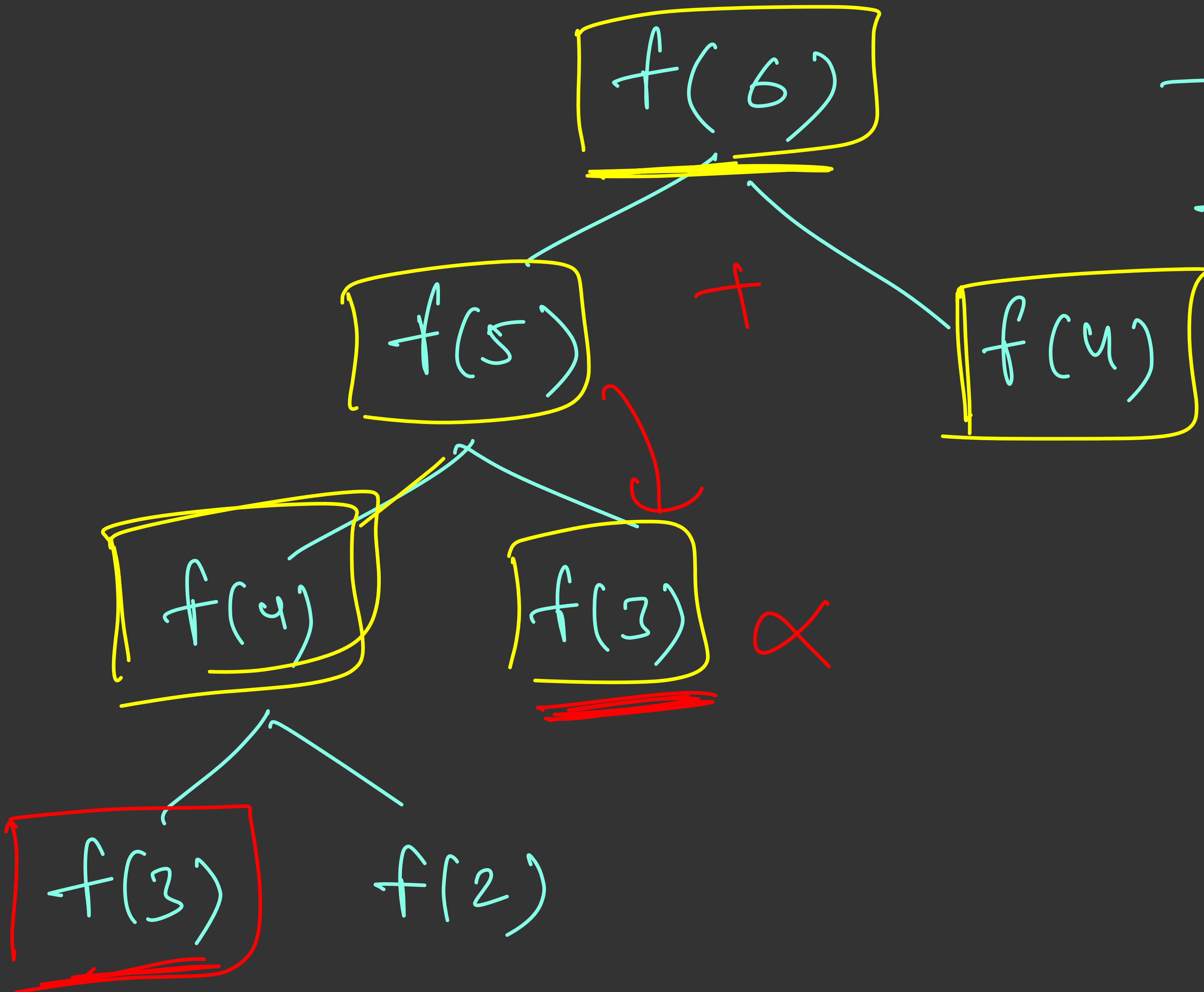
Grid problem

$\#$ states $\times$ space per state

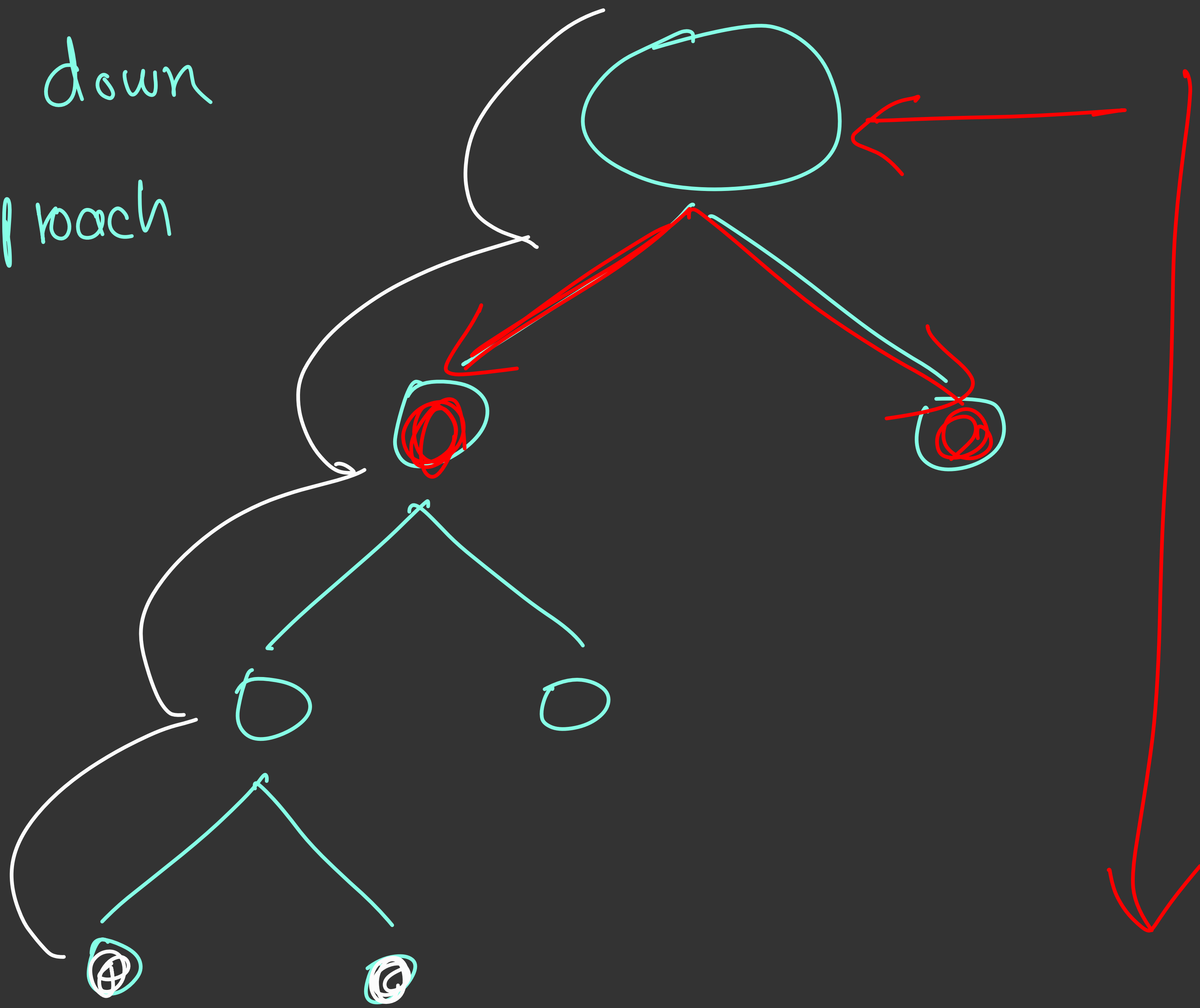
$O(n \cdot m) \times \underline{\underline{O(1)}}$

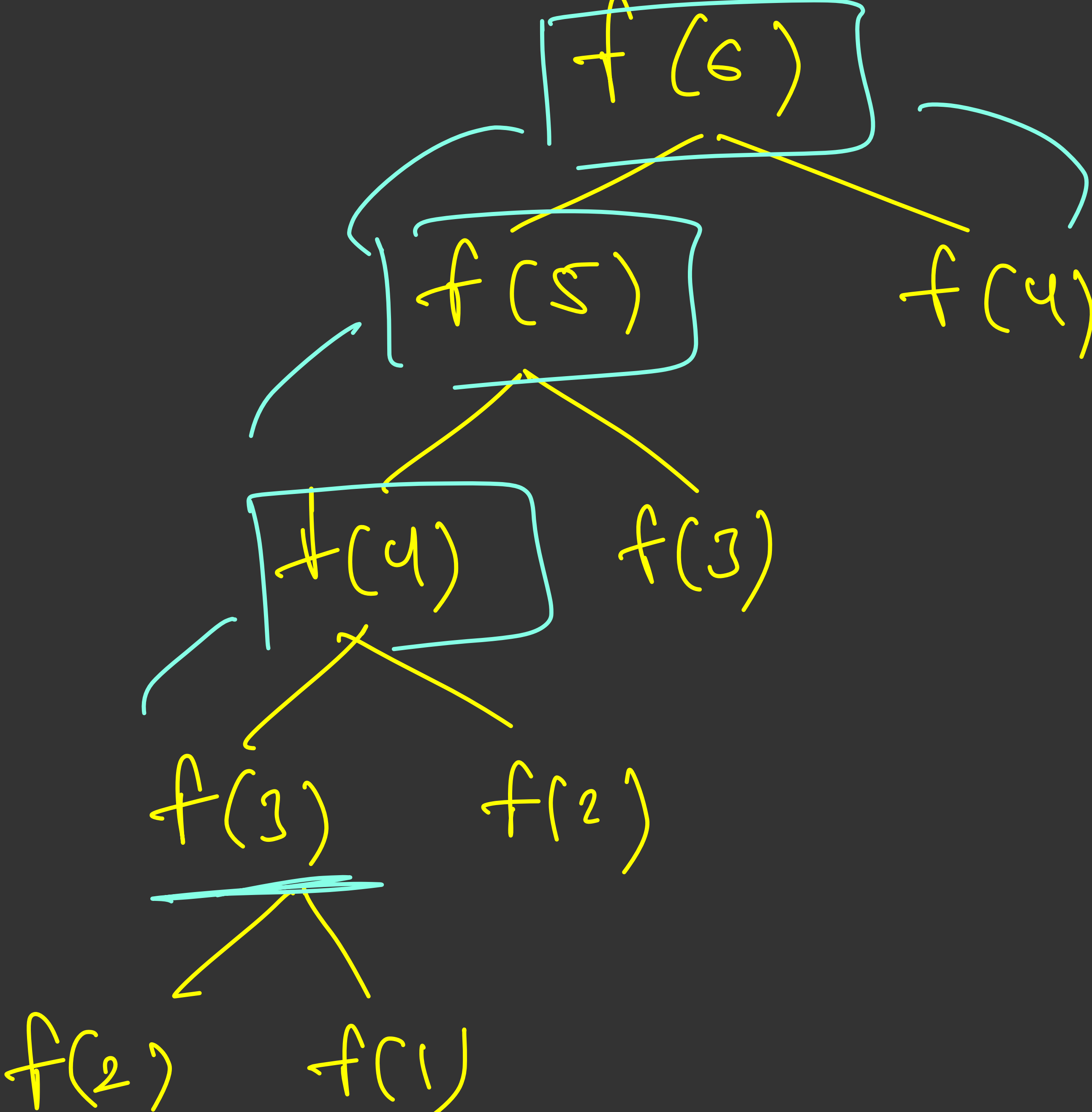
$\underline{\underline{O(n \cdot m)}}$

Top-down



Top down Approach

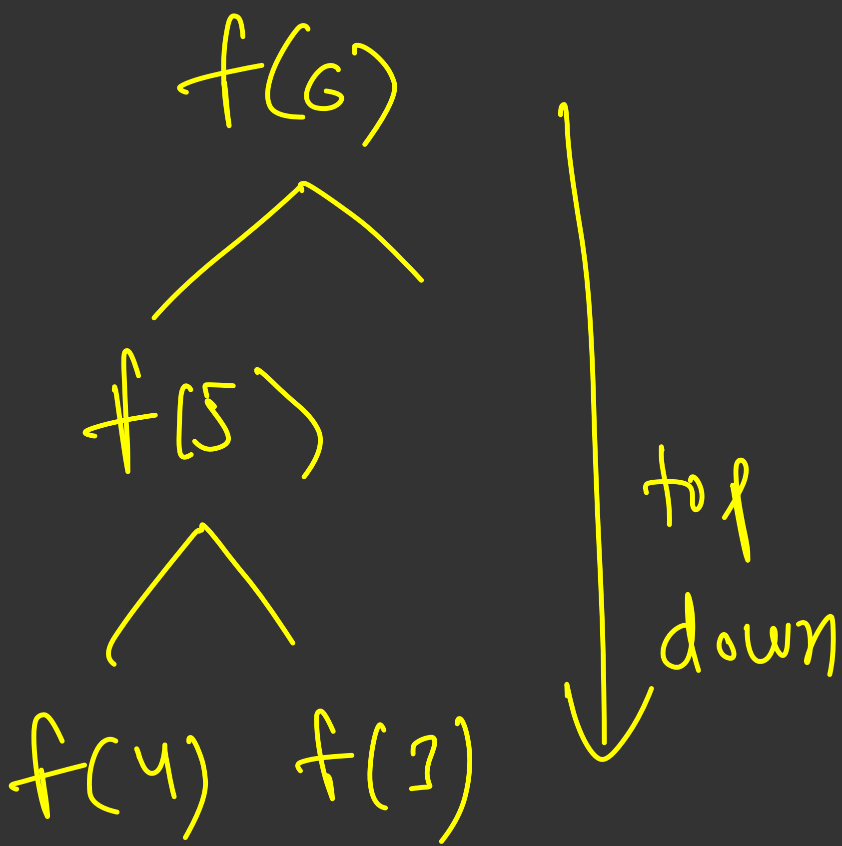



f(6)
f(5)
f(4)
f(4)
f(3)
f(3)
f(2)
f(2)
f(1)

$f(2) = 1$ $f(1) = 1$

$f(3) = 2$ $f(4) = 3$

```
int f[n+1];
f[1] = 1 , f[2] = 1
for (int i = 3; i <= n; i++){
        f[i] = f[i-1] + f[i-2]
}
```



recursive

$$f(1) = 1 \quad f(2) = 1$$

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

$$\vdots$$

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

iterative

Recursive vs Iterative DP

Recursive	Iterative
Slower (runtime)	Faster (runtime)
No need to care about the flow	Important to calculate states in a way that current state can be derived from previously calculated states
Does not evaluate unnecessary states	All states are evaluated
Cannot apply many optimizations	Can apply optimizations

f(6)

f(5)

f(4)

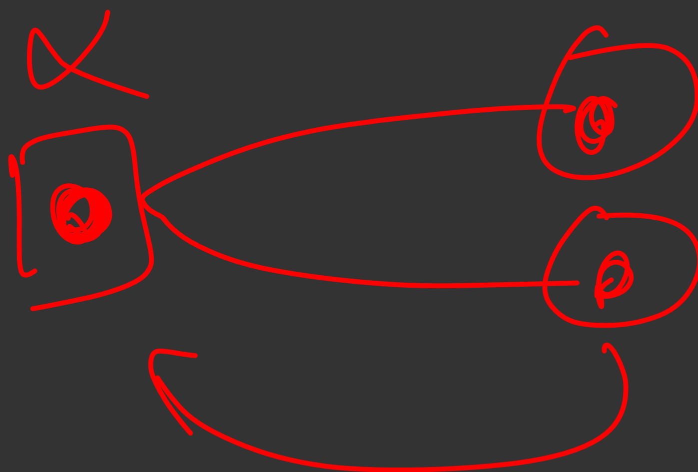
f(2)

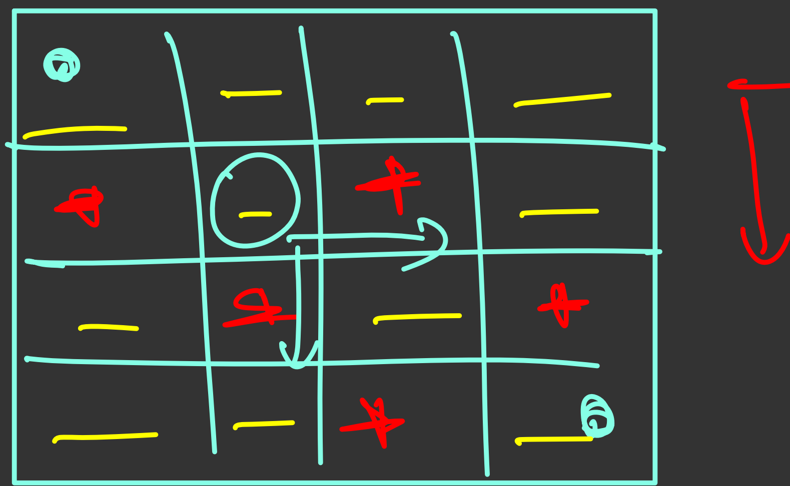
f(3)

f(4)

f(5)

f(6)



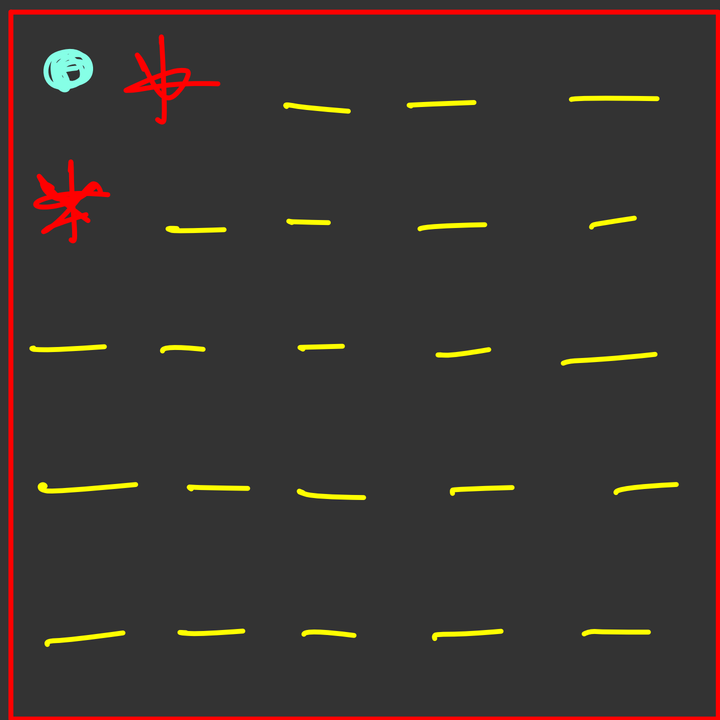


$n \times m$

find out whether you can go from top-left to bottom-right

$f(i,j)$ = true if you can go from (i,j) to $(n-1, m-1)$

$f(i,j)$ = (i,j) must not be an obstacle & $(f(i+1,j)$ or $f(i,j+1) ==$ True$)$



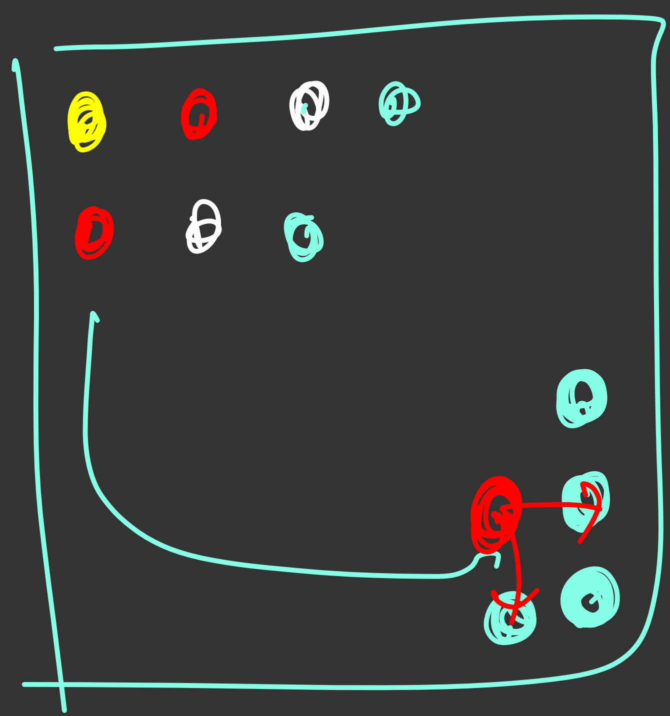
$$f(0,0)$$

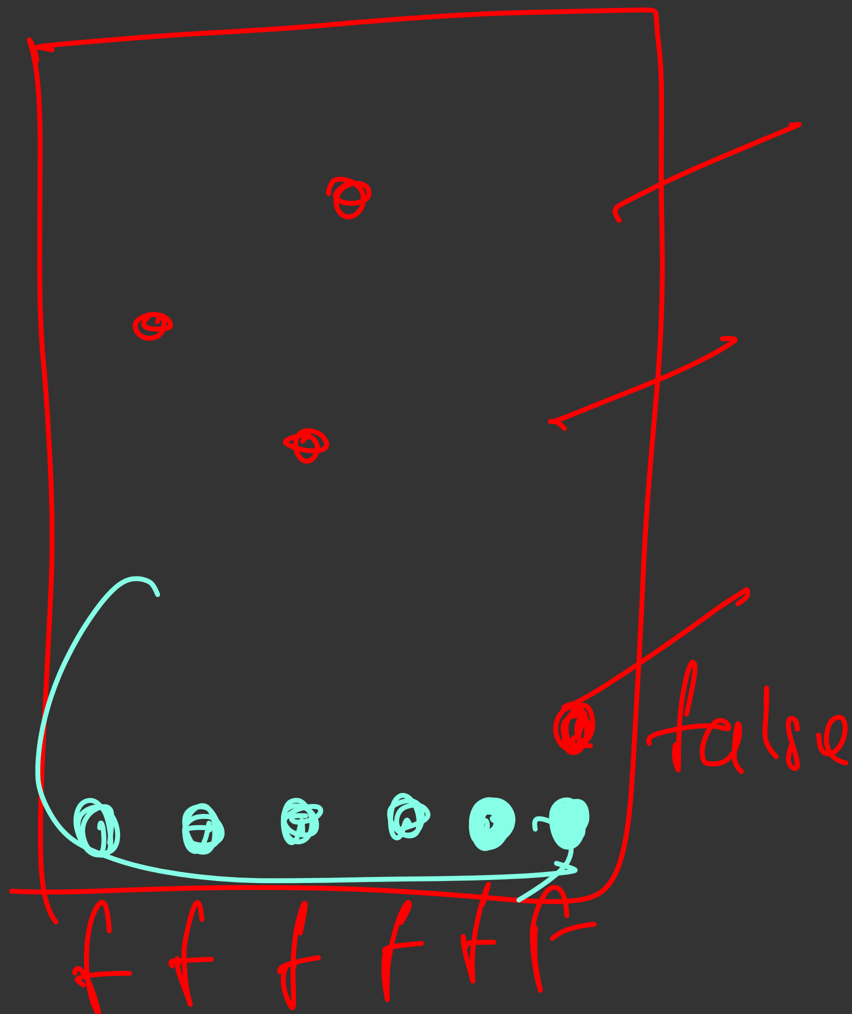
$$f(1,0) \qquad f(0,1)$$

$$f(i,j) \rightarrow n \times m \text{ subproblems}$$

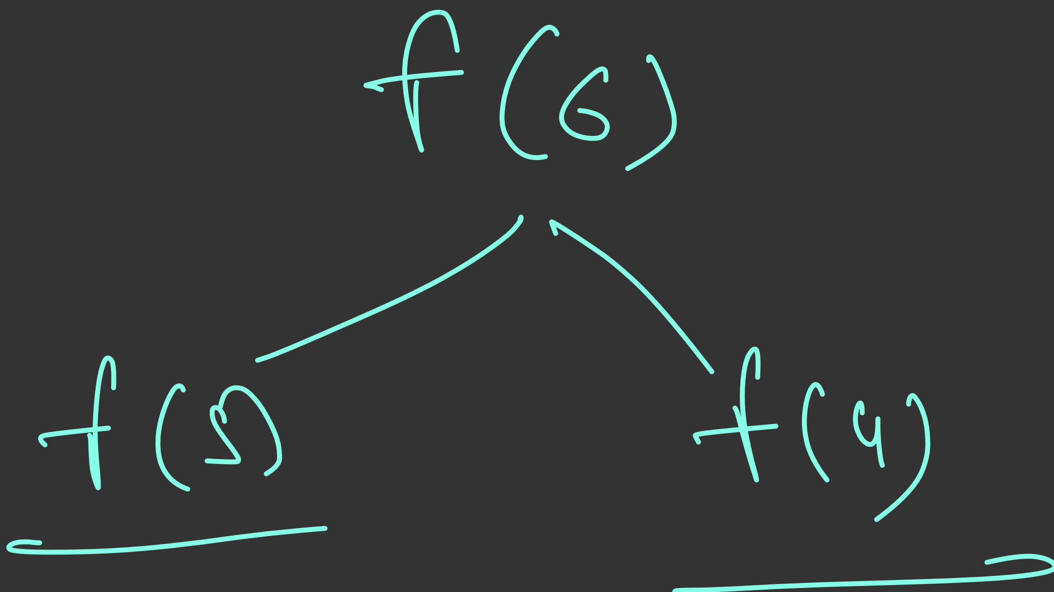
$$i \downarrow n \qquad j \downarrow m$$

$f(i,j) < \begin{matrix} f(i+1, j) \\ f(i, j+1) \end{matrix}$

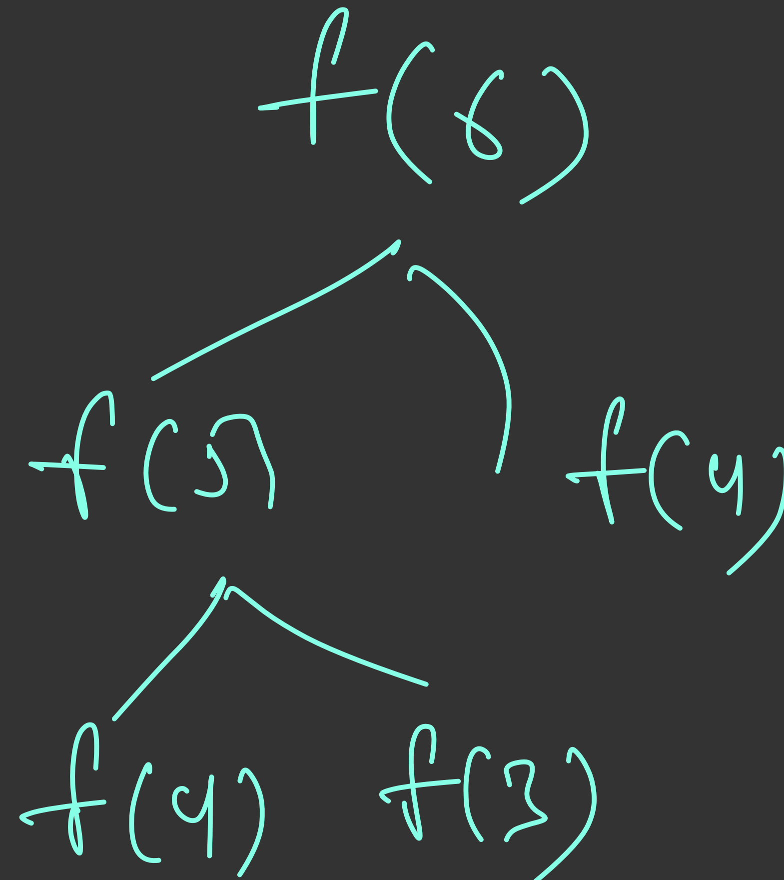



false
f f f f f f

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



$f(1)$ $f(2)$



$f(5) \longrightarrow$ do i know the answer already

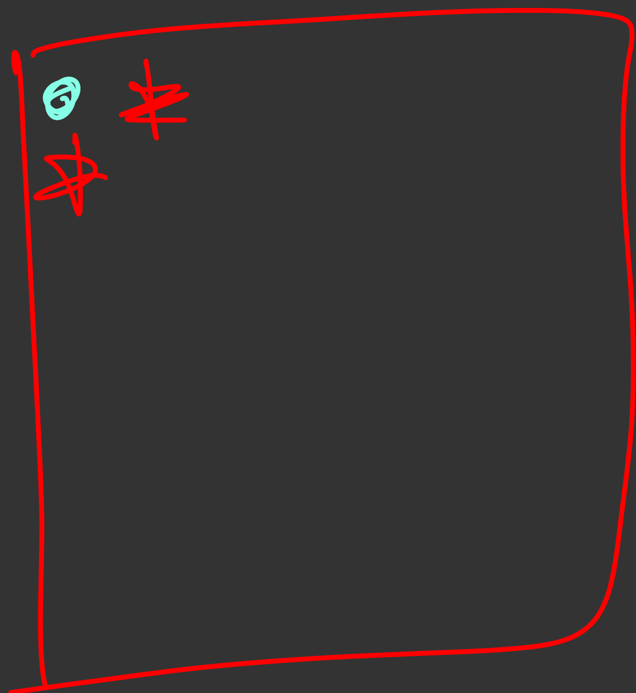
Yes

return

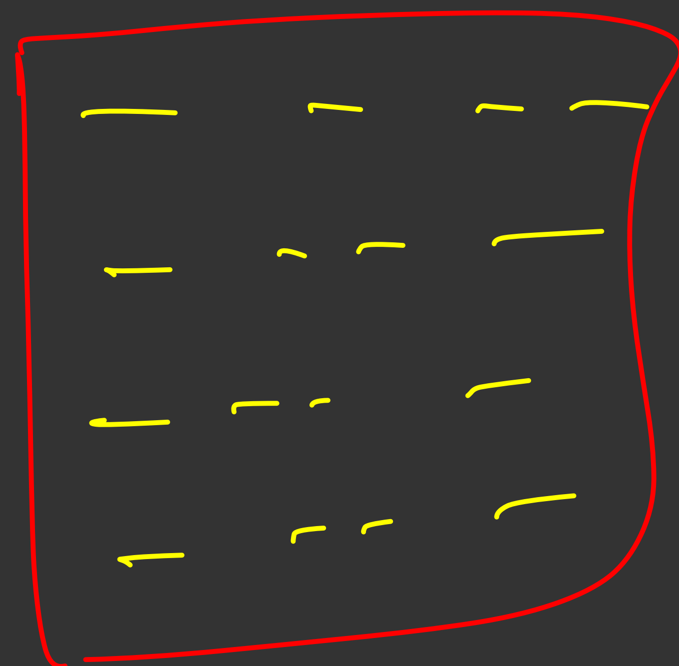
NO

Calculate

$$f(5) = f(4) + f(3)$$



good test case for recursive code



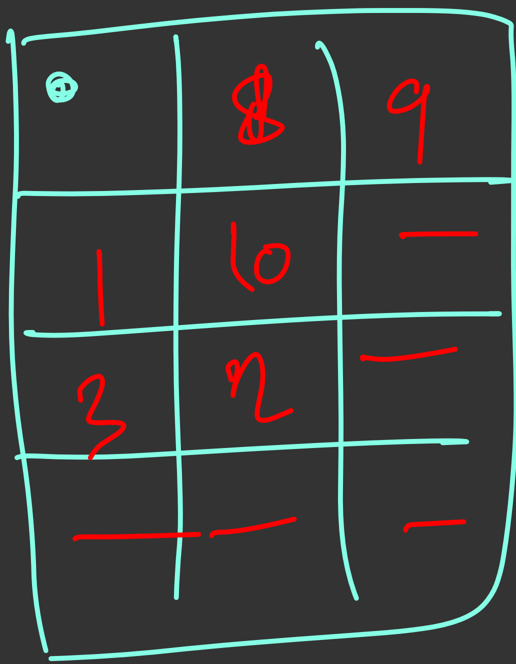
Converting Recursive to Iterative

Rule 1:

All the states that a particular state depends on must be evaluated before that state

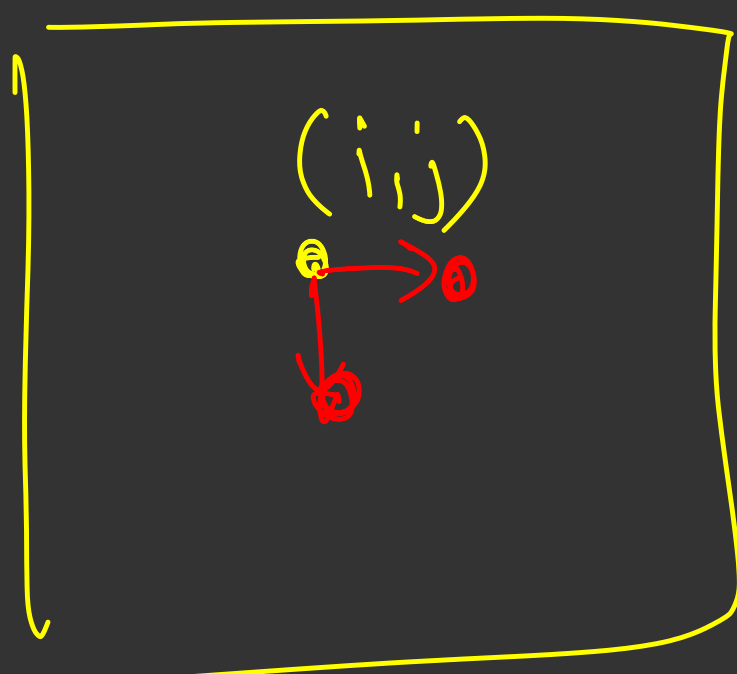
Note:

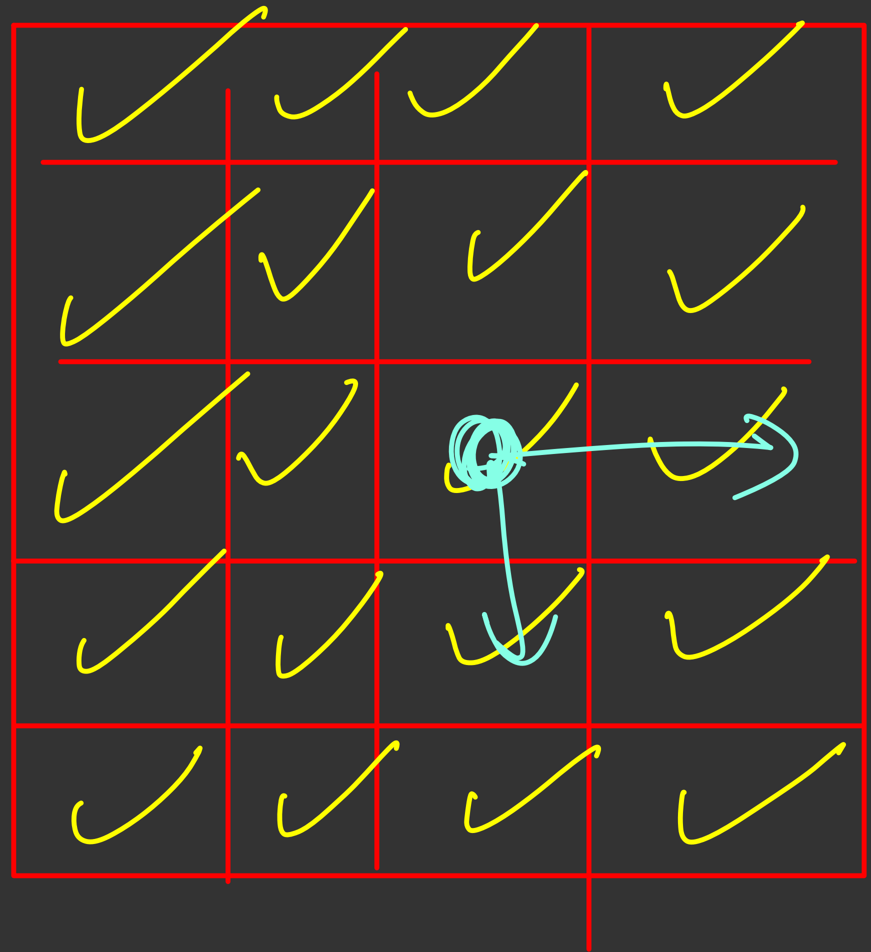
You don't have to convert Recursive to Iterative if it is not intuitive at this point.



$f(i,j)$ = min sum path from (i,j) to $(n-1, m-1)$

$$f(i,j) = \min\begin{Bmatrix} f(i+1,j) \\ f(i,j+1) \end{Bmatrix} + grid[i][j]$$





for (i = n-1 ; i >= 0 ; i--){
    for (j = m-1 ; j >= 0 ; j--){

dp[i][j]

$$\min \left\{ \begin{array}{c} dp[i+1][j] \\ dp[i][j+1] \end{array} \right\} + grid[i][j]$$

General Technique to solve any DP problem

1. State

    Clearly define the subproblem. Clearly understand when you are saying dp[i][j][k], what does it represent exactly

2. Transition:

    Define a relation b/w states. Assume that states on the right side of the equation have been calculated. Don't worry about them.

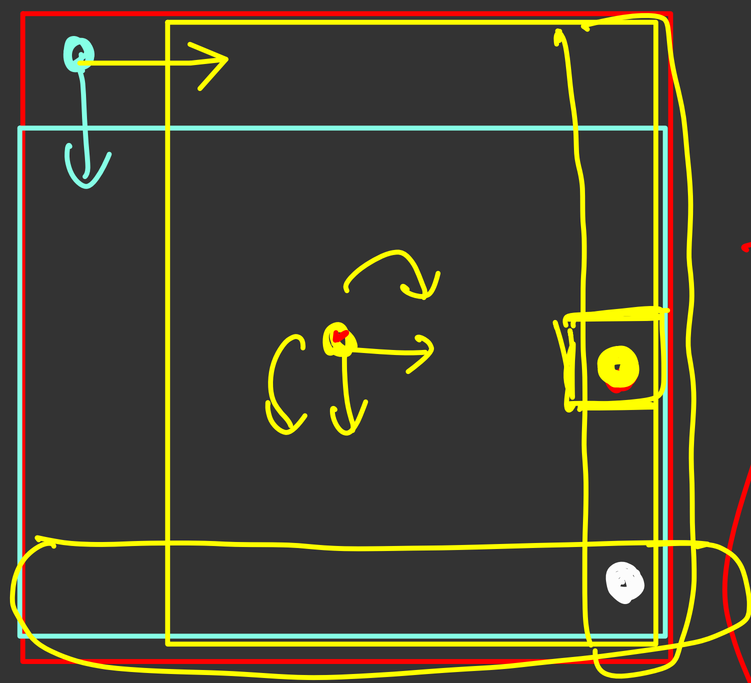
3. Base Case

    When does your transition fail? Call them base cases answer before hand. Basically handle them separately.

4. Final Subproblem

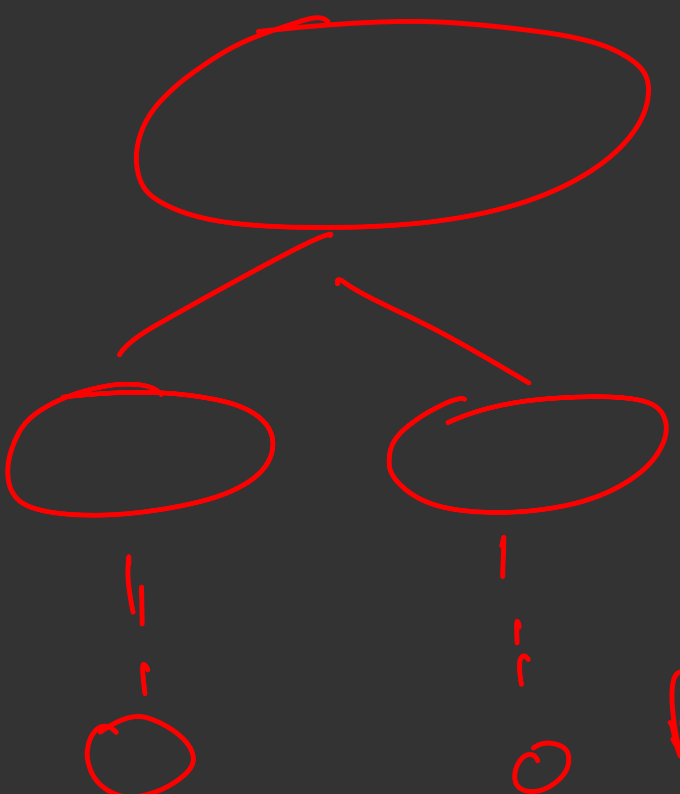
    What is the problem demanding you to find?

define a subproblem

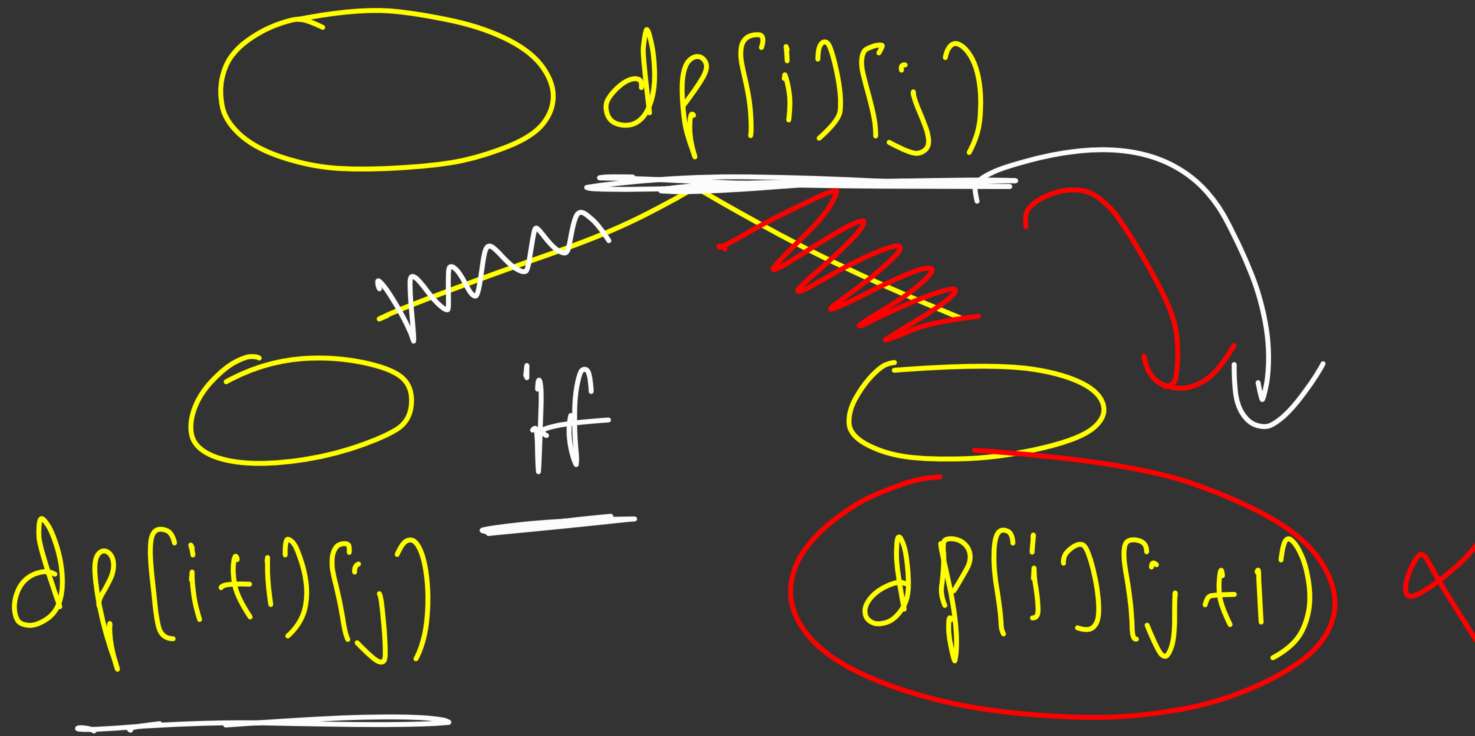


$f(i,j)$ = min sum path from (i,j) to $(n-1, m-1)$

$f(i+1, j)$, $f(i, j+1)$



Base case or Trivial case


dp[i][j]
if
dp[i+1][j]
dp[i][j+1]
x

B.C ① $dp[n-1][m-1] = grid[n-1][m-1]$

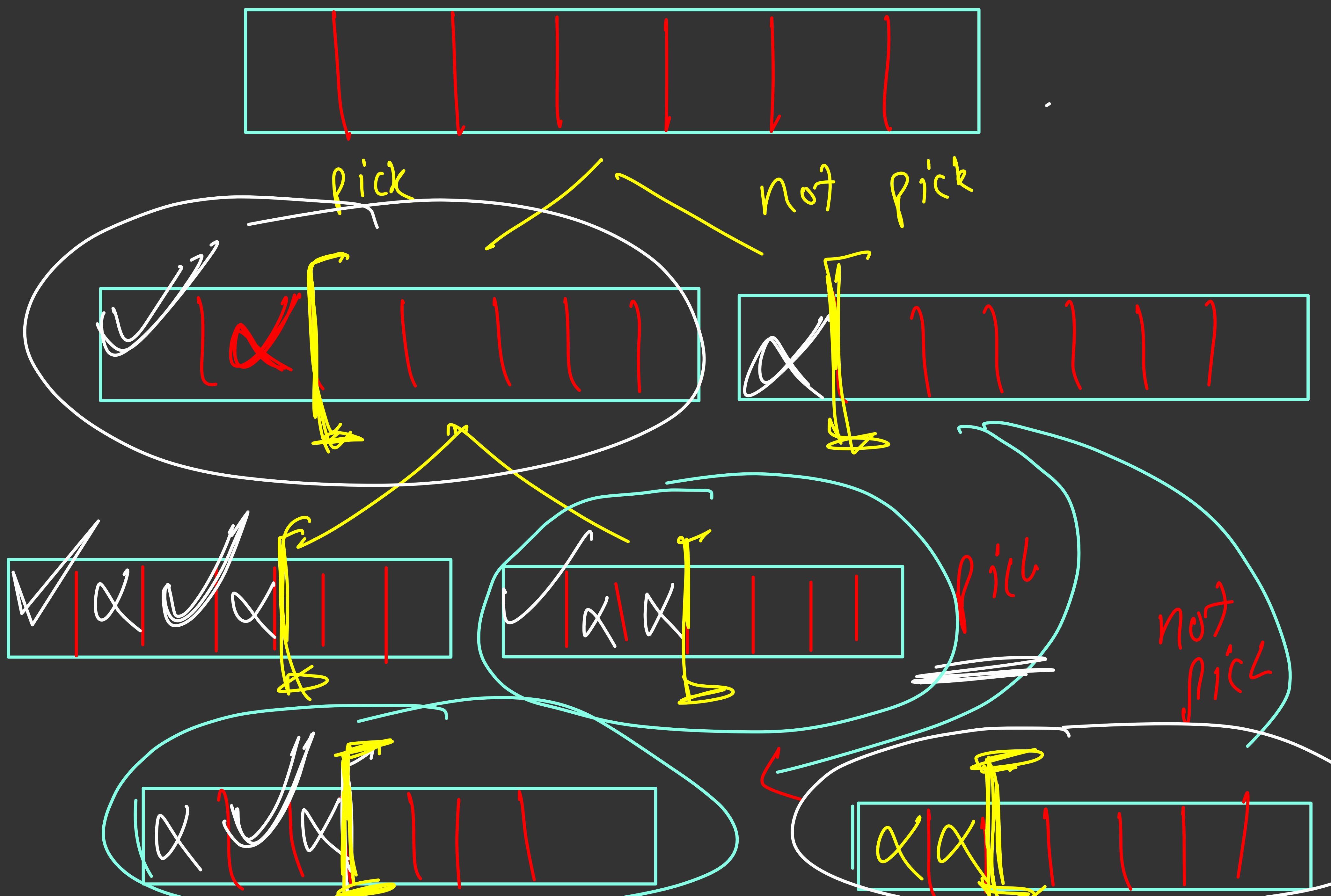
B.C ② last row

```
for(int i = m-2 ; i >= 0 ; i--){
        dp[n-1][i] = grid[n-1][i]
                     + dp[n-1][i+1]
} B.C ②
for(int i = n-2; i >= 0 ; i--){
        dp[i][m-1] = grid[i][m-1]
                     + dp[i+1][m-1]
} last column
```

```
for (int i = n-2; i >= 0; i--)
    for (int j = m-2; j >= 0; j--)
```

$$dp[i][j] = \min\left\{\begin{array}{l} dp[i+1][j] \\ dp[i][j+1] \end{array}\right\} + grid[i][j]$$

Problem 1: Link


pick
not pick
pick
not pick

$\boxed{f(i)}$ = max sum we can get from ith element to n-1 th element

pick

$$\boxed{f(i)} \;\; \max \begin{cases} arr(i) + f(i+2) & \text{pick} \\ f(i+1) & \text{not pick} \end{cases}$$

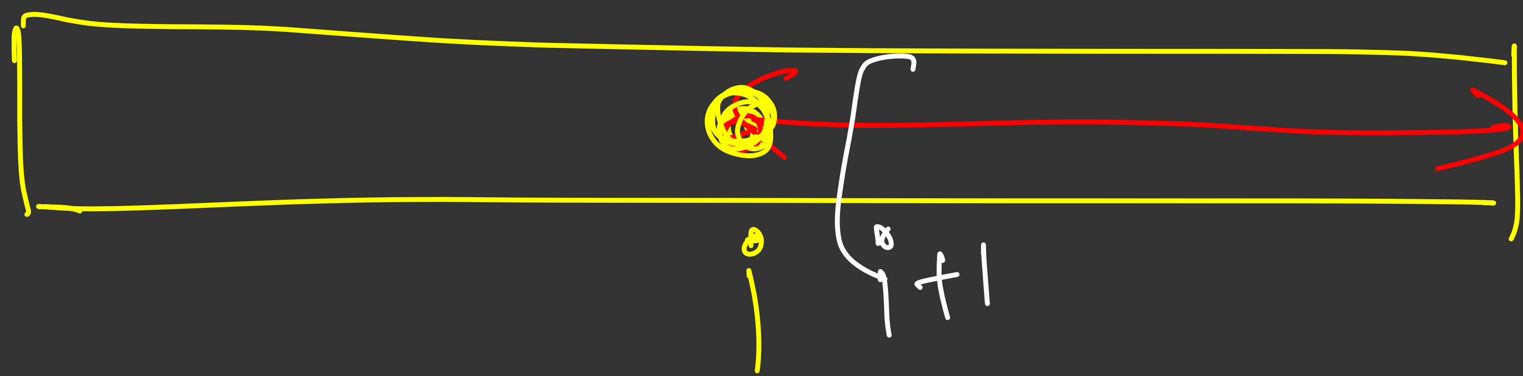
$$\boxed{f(n-1) = arr(n-1)}$$

$$\boxed{f(n-2) \rightarrow \max \begin{cases} arr(n-1) \\ arr(n-2) \end{cases}}$$

$\boxed{f(0) = \text{final subproblems}}$

dp[i][1] = max sum we can get from ith house to n-1 th house such that we pick up the ith house

dp[i][0] = max (such that) ith house is not picked



$$dp[i][1] = arr[i] + dp[i+1][0]$$

$$dp[i][0] = \max\begin{cases} dp[i+1][0] \\ dp[i+1][1] \end{cases}$$

$$\left.\begin{aligned} dp[n-1][1] &= arr[n-1] \\ dp[n-1][0] &= 0 \end{aligned}\right\}$$

$$f.s = \max \begin{cases} dp[0][1] \\ dp[0][0] \end{cases}$$

Some ways to solve the problem

1. Having 2 parameters to represent the state

State:

dp[i][0] = maximum sum in (0 to i) if we don't pick i^{th} element

dp[i][1] = maximum sum in (0 to i) if we pick i^{th} element

Transition:

dp[i][0] = max(dp[i - 1][1], dp[i - 1][0])

dp[i][1] = arr[i] + dp[i - 1][0]

Final Answer:

max(dp[n - 1][0], dp[n - 1][1])

Some ways to solve the problem

2. Having only 1 parameter to represent the state

State:

dp[i] = max sum in (0 to i) not caring if we picked i^{th} element or not

Transition: 2 cases

- pick i^{th} element: cannot pick the last element : arr[i] + dp[i - 2]
- leave i^{th} element: can pick the last element : dp[i - 1]

dp[i] = max(arr[i] + dp[i - 2], dp[i - 1])

Final Answer:

dp[n - 1]

```cpp
int a[n]; // input array

int dp[n]; // filled with -INF to represent uncalculated state

// f(i) = max sum till index i
int f(int index){
    if(index < 0) // reached outside the array
        return 0;
    if(dp[index] != -INF) // state already calculated
        return dp[index];

    // try both cases and store the answer
    dp[index] = max(a[index] + f(index - 2), f(index - 1));
    return dp[index];
}

void solve(){
    cout << f(n - 1) << nline;
}
```