Optimized Brute force

Dynamic Programming 1.1

- Priyansh Agarwal

Why Dynamic Programming?

- Overlapping subproblems
- Maximize/Minimize some value
- Finding number of ways
- Covering all cases (DP vs Greedy)
- Check for possibility

Optimized

Brute force

DP

Greedy

Calculate no. of days b/w

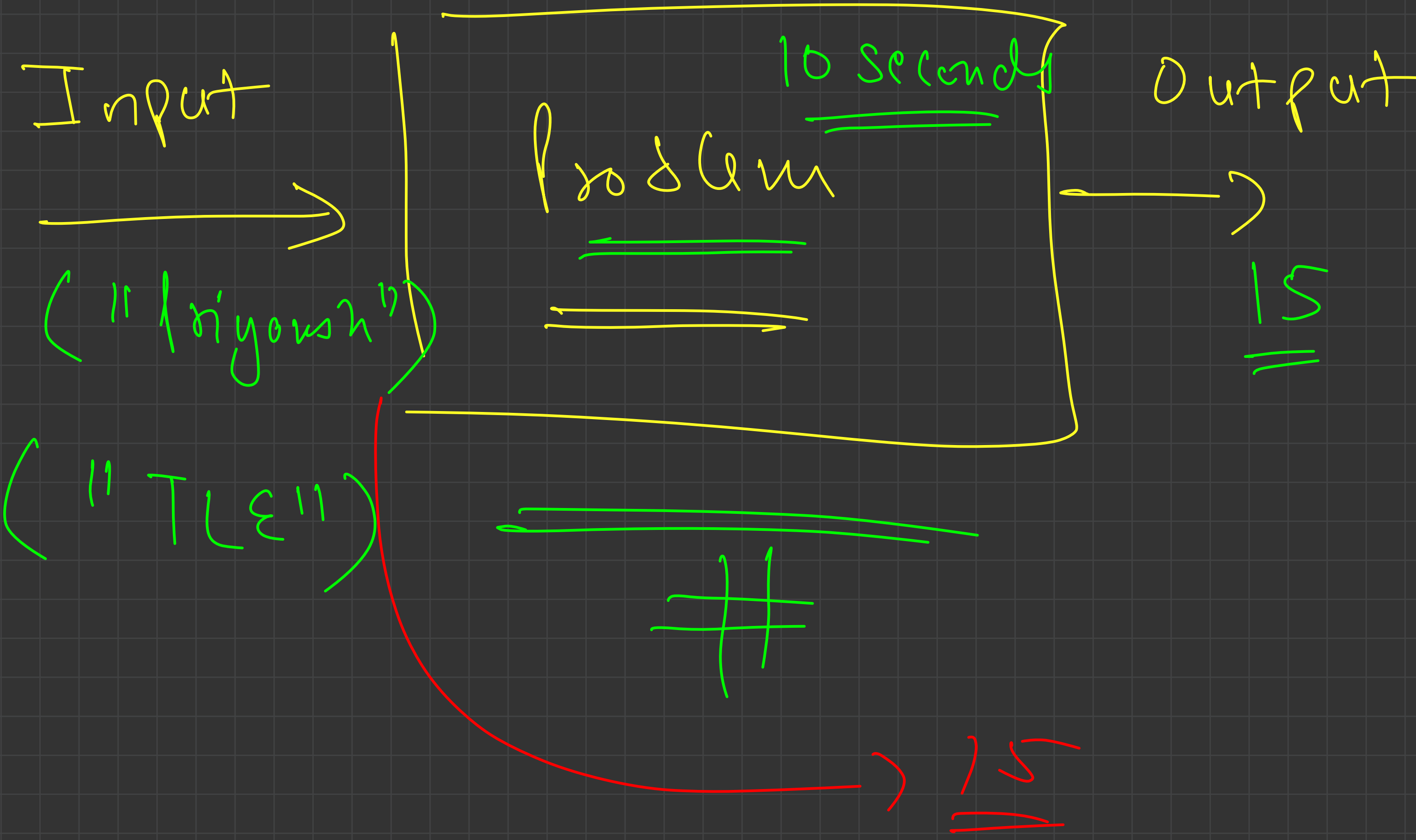
Date 1 & Date 2

<< 19/06/22 22/07/23

$= x$

19/06/22 —— 23/07/23

$= x + 1$


Input
("Priyansh")
("TLE")
Program
10 seconds
Output
15
15

Mapping Table

Inputs → Outputs

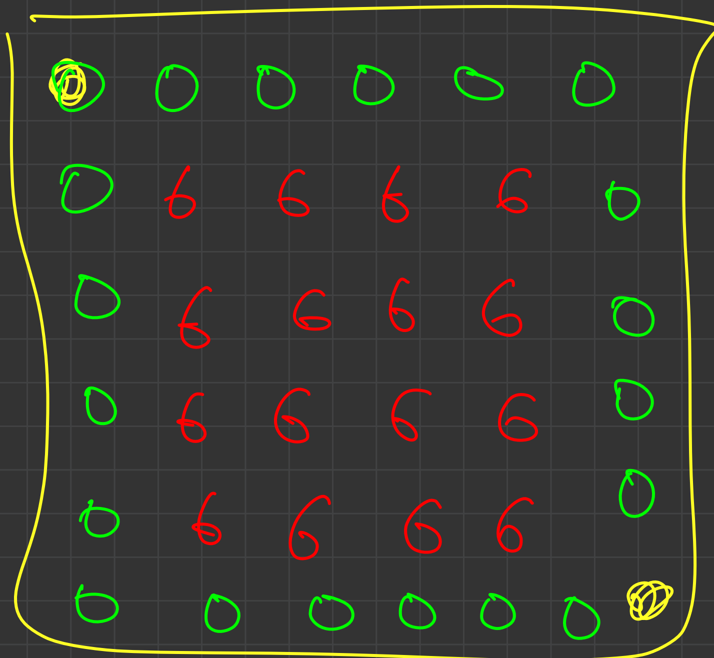
"Priyansh" → run the program → store

1 2 3 7 9

6 2 1 3

Optimized

Brute force

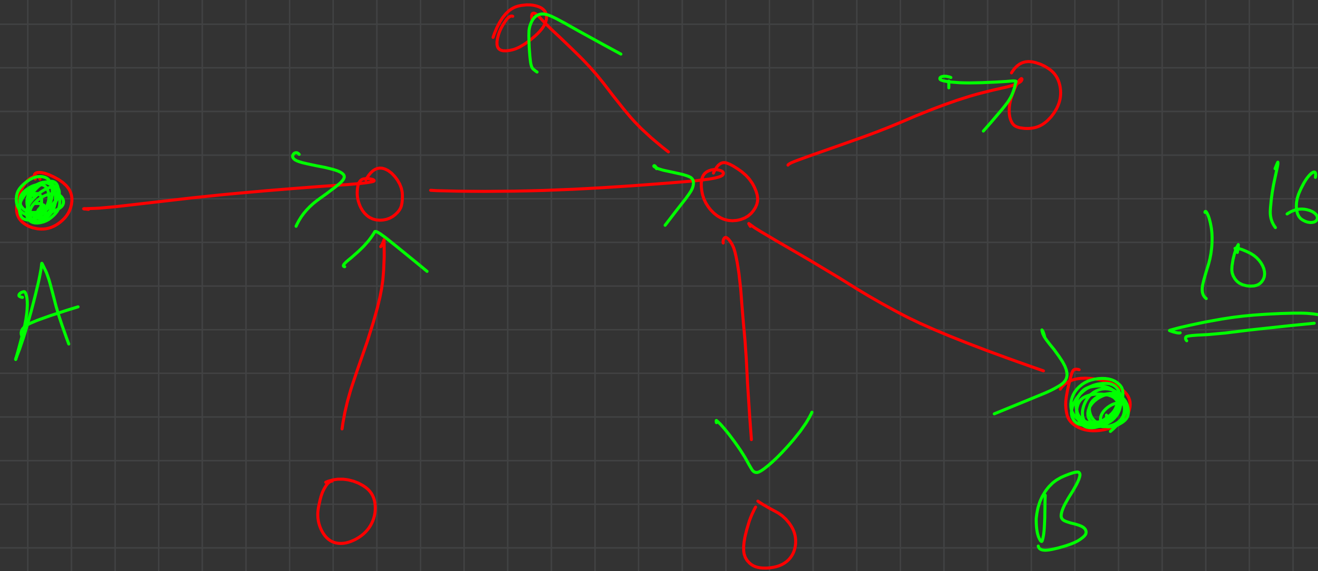


Greedy

All values are range

Brute force

optimise with DP



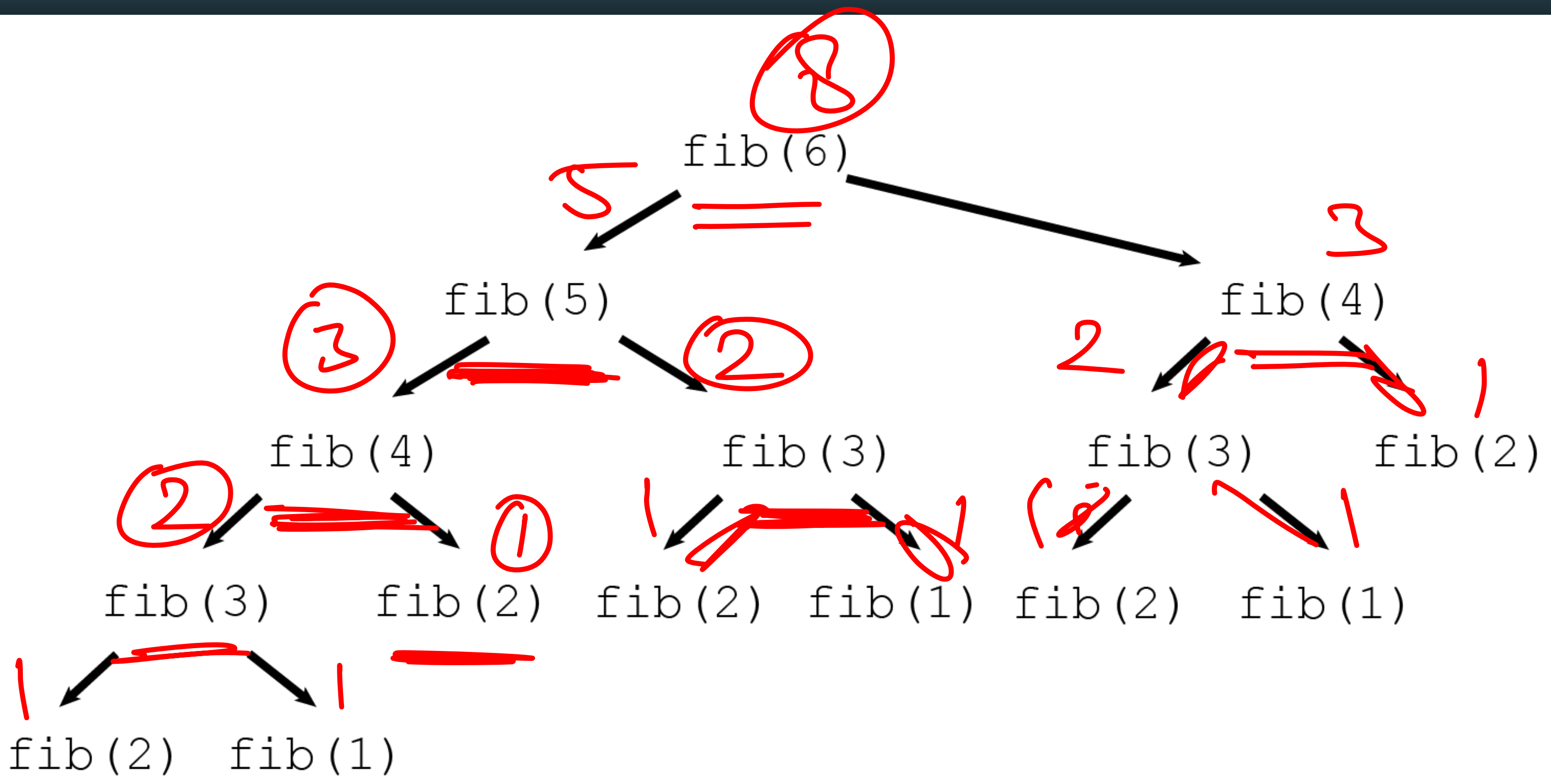
Recursive code which tries out all paths

[# of ways % 10^9+7]

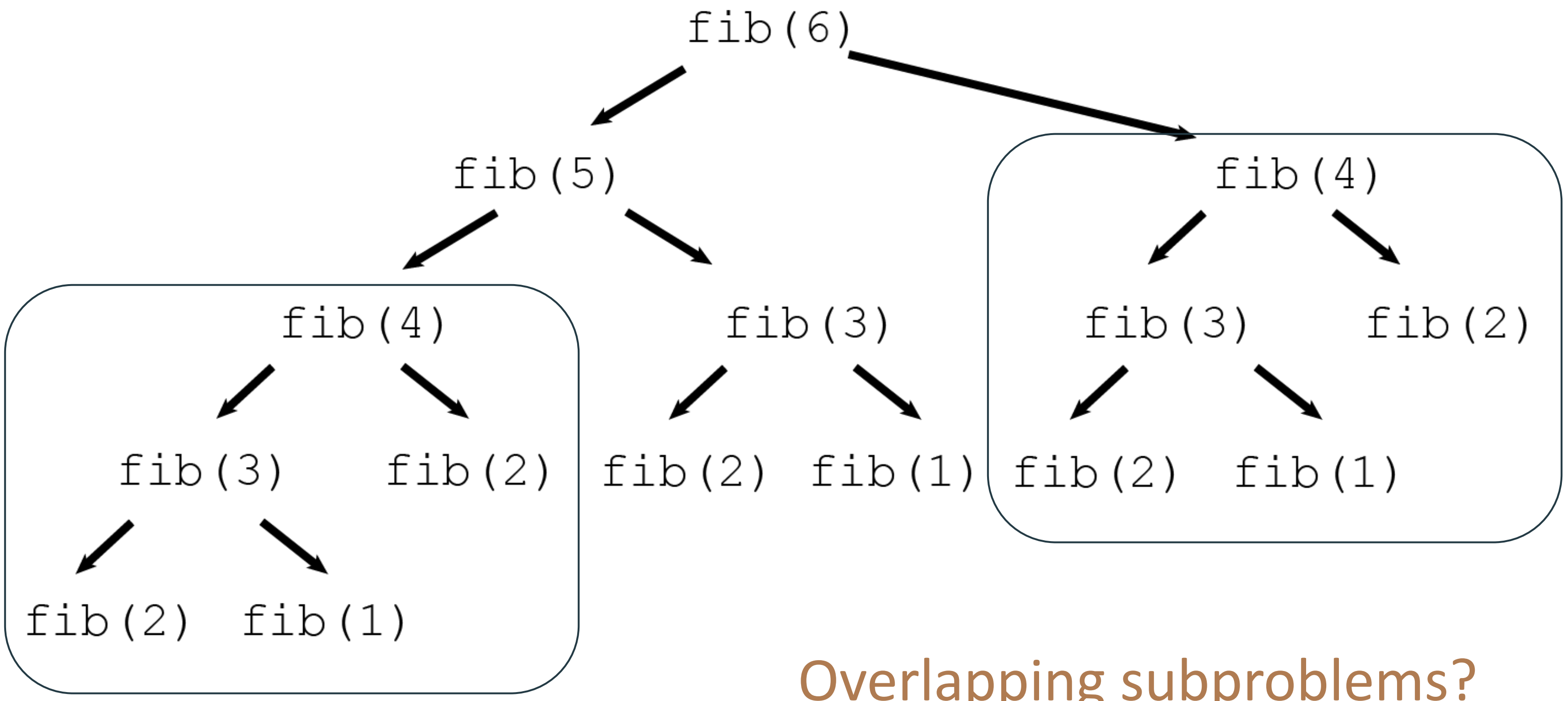
Need of DP

- Let's understand this from a problem
  - Find n^{th} fibonacci number
  - $F(n) = F(n - 1) + F(n - 2)$
  - $F(1) = F(2) = 1$

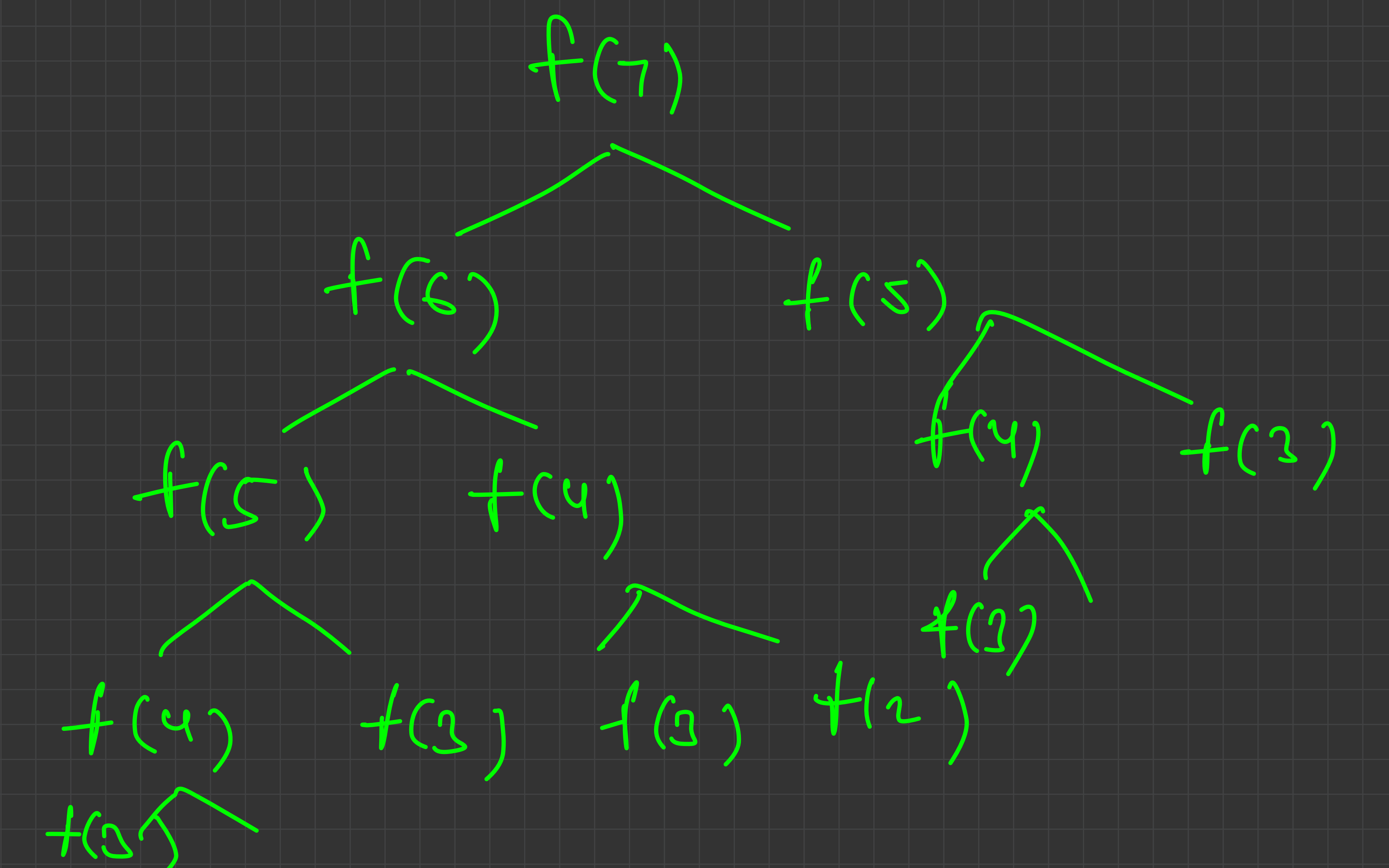
$$f(n) = f(n-1) + f(n-2)$$



Any problem here?



Overlapping subproblems?


f(7)
f(6)
f(5)
f(5)
f(4)
f(4)
f(3)
f(3)
f(4)
f(3)
f(3)
f(2)
f(3)

Memoization

- Why calculate F(x) again and again when we can calculate it once and use it every time it is required?
    - Check if F(x) has been calculated
        - If No, calculate it and store it somewhere
        - If Yes, return the value without calculating again

Input $\longrightarrow$ $\boxed{\text{Program}}$ $\longrightarrow$ Output

Input $\longrightarrow$ If this problem was solved earlier

No

Yes

Solve the problem & store the answer somewhere

Retrieve the ans and return

1. Check if solved earlier

2. Store the answer

3. Retrieve the answer

Input	Output
1	5
6	7
9	2
5	8
7	100

x

Memory

9

Hash map

7

Calculate

O(1) access

Input → [1 —— 1000]

Output → [−10 —— 10] $O(1)$

Boolean Array

f	t	t	t	t		t
1	2	3	4	5	---	100

Integer Array 2

				10	

Default Value

[0 —— 10]

A1 (boolean) | F T |

↳ calculate & populate in A2

A2 (integer

5

A3

0	0	0	0	7	0	0	0

Without DP

int dp[n]

30th fib

```
int functionEntered = 0;
int helper(int n){
    functionEntered++;
    if(n == 1 || n == 2){
        return 1;
    }
    return helper(n - 1) + helper(n - 2);
}
void solve(){
    int n;
    cin >> n;
    cout << helper(n) << nline;
    cout << functionEntered << nline;
}
```

functionEntered = 1664079
with n = 30

With DP

```
int functionEntered = 0;
int dp[40];
int helper(int n){
    functionEntered++;
    if(n == 1 || n == 2){
        return 1;
    }
    if(dp[n] != -1)
        return dp[n];
    return dp[n] = helper(n - 1) + helper(n - 2);
}
void solve(){
    int n;
    cin >> n;
    for(int i = 0; i <= n; i++)
        dp[i] = -1;
    cout << helper(n) << nline;
    cout << functionEntered << nline;
}
```

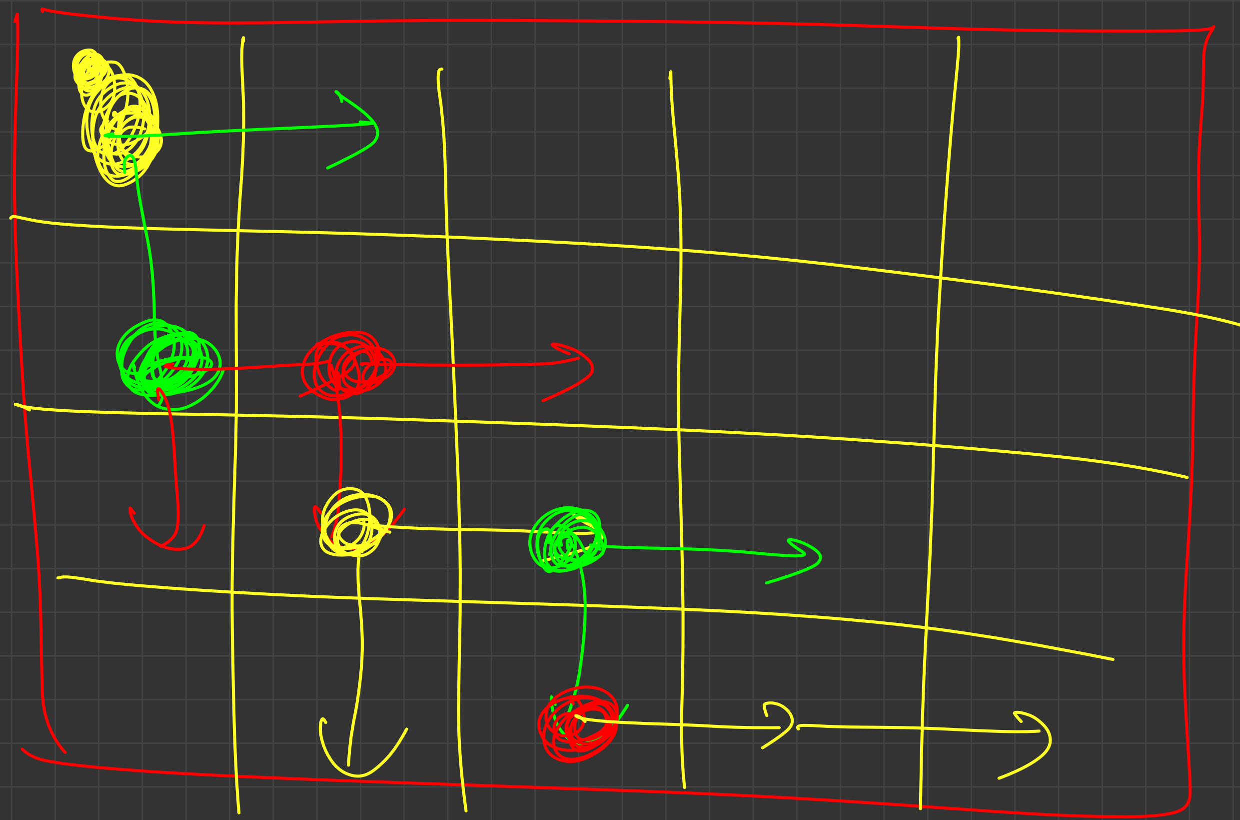
functionEntered = 57
with n = 30

Let's solve another problem!

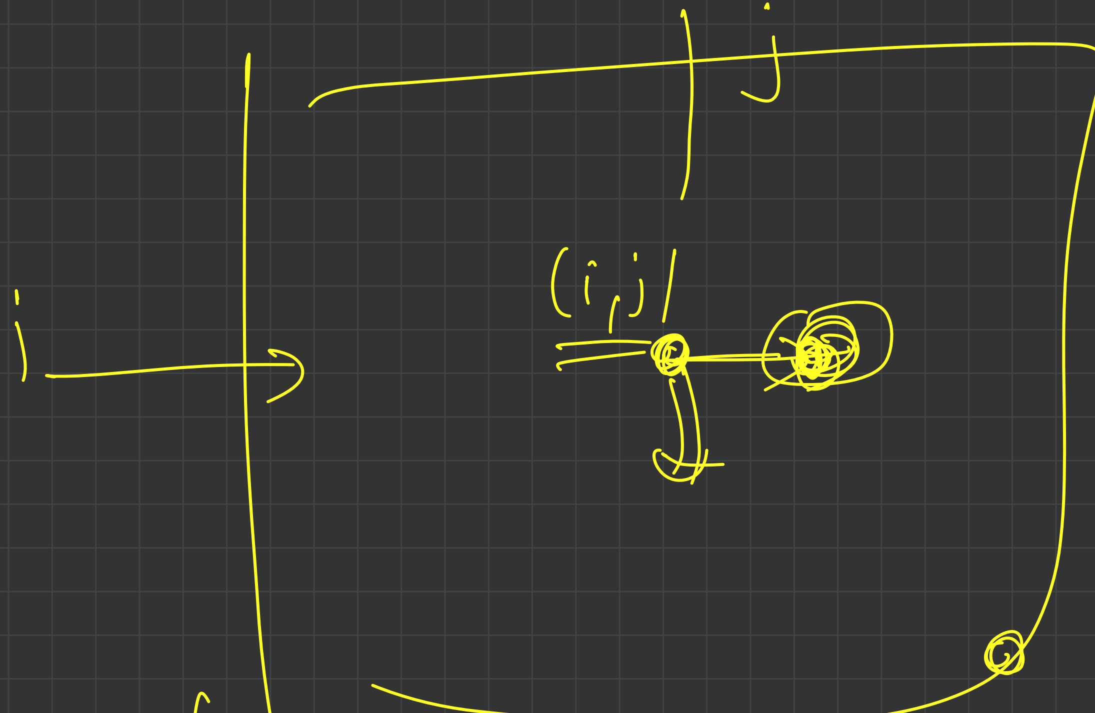
Given a 2D grid (N X M) with numbers written in each cell, find the path from top left (0, 0) to bottom right (n - 1, m - 1) with minimum sum of values on the path

All values from 0 to 10^6

1	5	8
6	2	7
9	3	4



$f(i,j)$

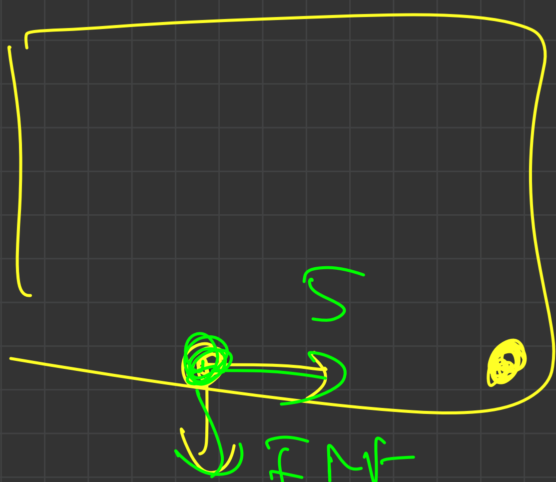


$f(i,j)$

$f(i,j+1)$

$f(i+1,j)$

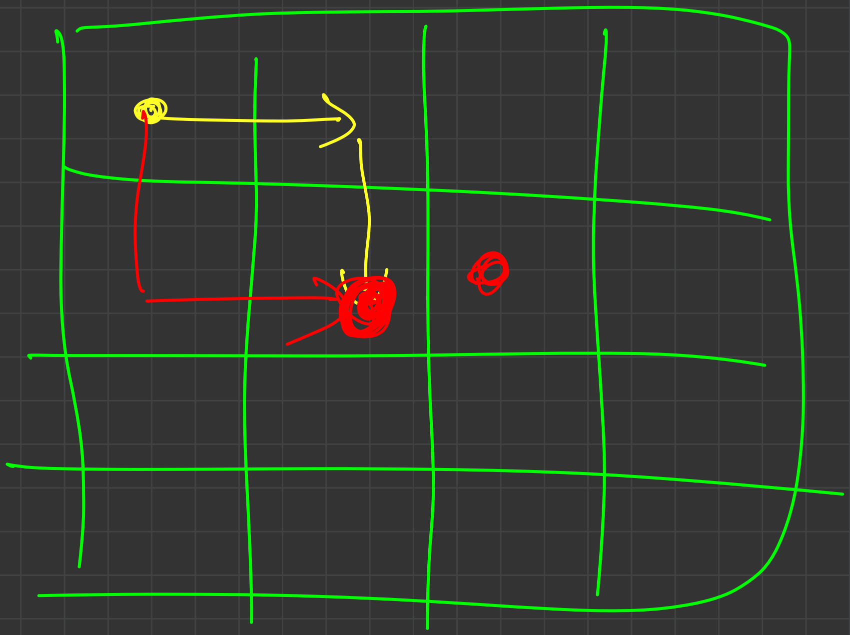
$$f(i,j) = grid[i][j] + \min \begin{cases} f(i+1, j) \\ f(i, j+1) \end{cases}$$



$$\underline{\underline{INF = 10^{18}}}$$

```
int f (int i, int j)
    if (i==n-1 and j==m-1)
            return grid[n-1][m-1]
    if (i==n or j==m)
            return INF

return grid[i][j] + min { f(i+1, j)
                          f(i, j+1) }
```



$f(2,3)$

Naive Way

Explore all paths. Standing at (i, j) try both possibilities (i + 1, j), (i, j + 1)

Every cell has two choices

Time complexity: $O(2^{m*n})$?

Actual Time complexity: $O(C(n + m - 2, m - 1))$

Efficient Way

Overlapping subproblems

Memoization

Time complexity: O(n * m)

Space complexity: O(n * m)

```cpp
int grid[n][m]; // input matrix

int dp[n][m]; // every value here is -1

// subproblem: f(i, j) represents minimum sum path from (i, j) to (n - 1, m - 1)
int f(int i, int j){ //
    if(i >= n || j >= m){ // moving outside the grid // not allowed
        return INF;
    }
    if(i == n - 1 && j == m - 1) // reached the destination
        return grid[n - 1][m - 1];

    if(dp[i][j] != -1) // this state has been calculated before
        return dp[i][j];

    // state never calculated before
    dp[i][j] = grid[i][j] + min(f(i, j + 1), f(i + 1, j));
    return dp[i][j];
}

void solve(){
    cout << f(0, 0) << nline;
}
```

```
int grid[n][m]; // input matrix

int [redacted]; // every value here is -1

// subproblem: f(i, j) represents [redacted]imum sum path from (i, j) to (n - 1, m - 1)
[redacted] f(int i, int j){ //
    if(i >= n || j >= m){ // moving outside the grid // not allowed
        [redacted]
    }
    if(i == n - 1 && j == m - 1) // reached the destination
        [redacted];

    if(dp[i][j] != -1) // this state has been calculated before
        return [redacted];

    // state never calculated before
    dp[i][j] = grid[i][j] + min([redacted], [redacted]);
    [redacted]
}

void solve(){
    cout << f(0, 0) << endline;
}
```

void

dp[n+1][m+1]

dp[i][j] = INF, return

dp[i][j] = grid[i][j], return.

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

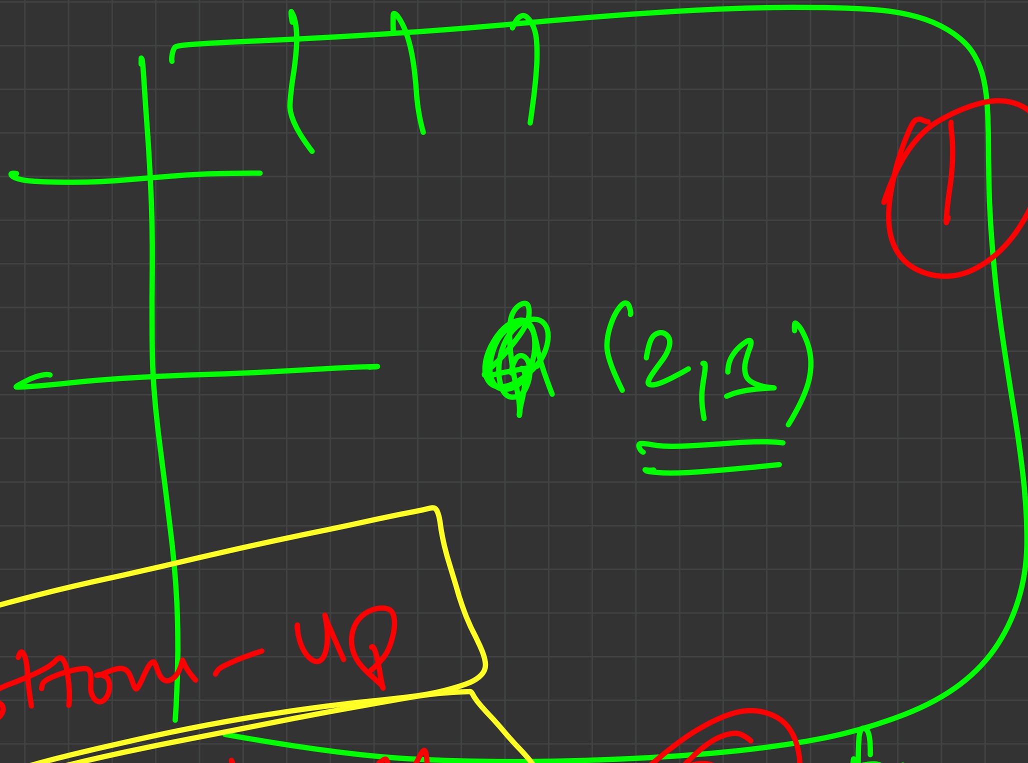
dp[i][j+1]   dp[i+1][j]

cout << dp[0][0] << endline

$$dp[i][j] = g_2 - + \min \begin{cases} dp[i+1][j] \\ dp[i][j+1] \end{cases}$$

$dp[5][4] \to dp[5][5]$

$dp[6][5]$



1 min path from (2,3) to (n-1, m-1)

2 from (0,0) to (2,3)

bottom-up

top-down

Important Terminology

dp[i][j]

State: A subproblem that we want to solve. The subproblem may be complex or easy to solve but the final aim is to solve the final problem which may be defined by a relation between the smaller subproblems. Represented with some parameters.

dp(n)

dp(n)

Transition: Calculating the answer for a state (subproblem) by using the answers of other smaller states (subproblems). Represented as a relation b/w states.

dp[n][k]

$$\boxed{dp[i][j]} = \min(\underline{dp[i+1][j]},\ \underline{dp[i][j+1]}) + grid[i][j]$$

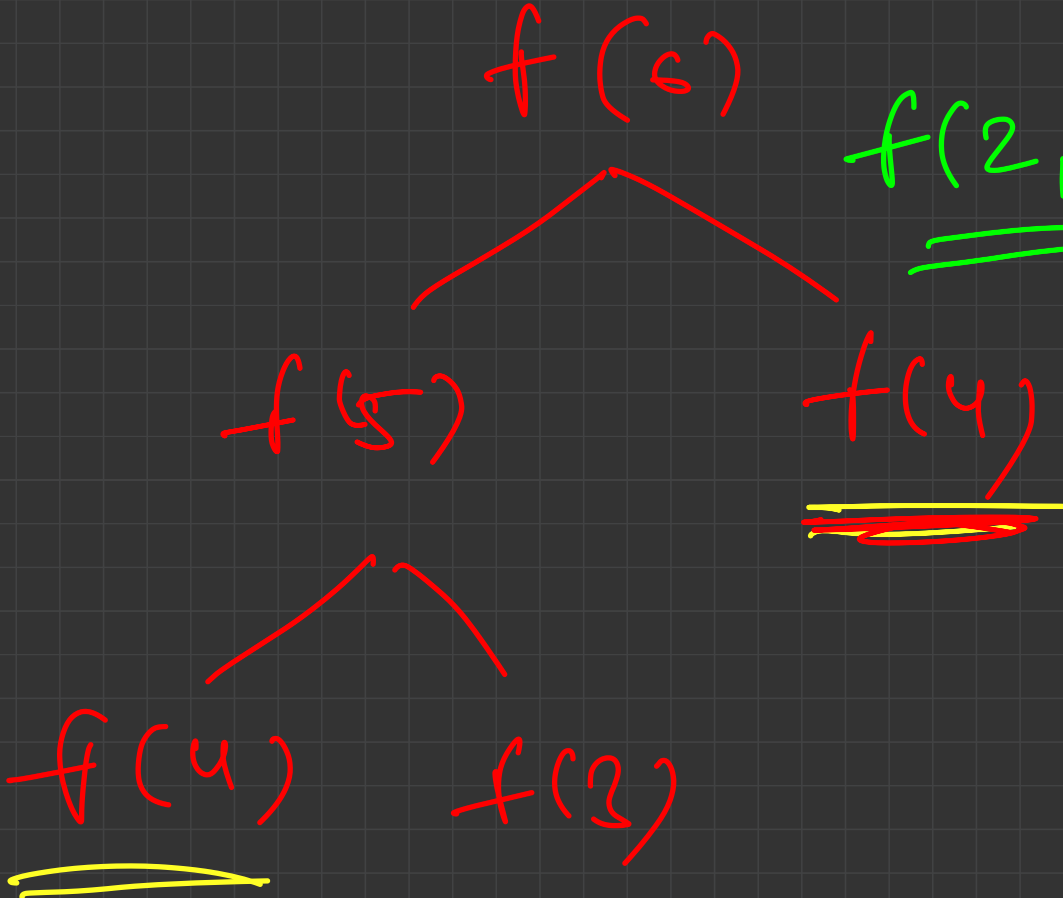
L.H.S

R.H.S

State

Or

Tree



f(2, 3)

= 4th fibn

numb

Exercise

Fibonacci Problem:

- State
  - dp[i] or f(i) meaning i^{th} fibonacci number
- Transition
  - dp[i] = dp[i - 1] + dp[i - 2]

Exercise

Matrix Problem:

- State
  - dp[i][j] = shortest sum path from (i, j) to (n - 1, m - 1)
- Transition
  - dp[i][j] = grid[i][j] + min(dp[i + 1][j], dp[i][j + 1])

dp[i][j] = min sum path from (0,0) to (i,j)

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

$dp[i][j] = grid[i][j] + \min(dp[i+1][j], dp[i][j+1])$

final → $dp[0][0]$

$dp[i][j]$ = min sum path from $(0, 0)$ to (i, j)

$dp[i][j] = grid[i][j] + \min(dp[i-1][j], dp[i][j-1])$

final → $dp[n-1][m-1]$

Let's solve another problem

Given an array of integers (both positive and negative). Pick a subsequence of elements from it such that no 2 adjacent elements are picked and the sum of picked elements is maximized.

1	4	2	-10	10	5

Sum = 14

1	4	2	-10	10	5

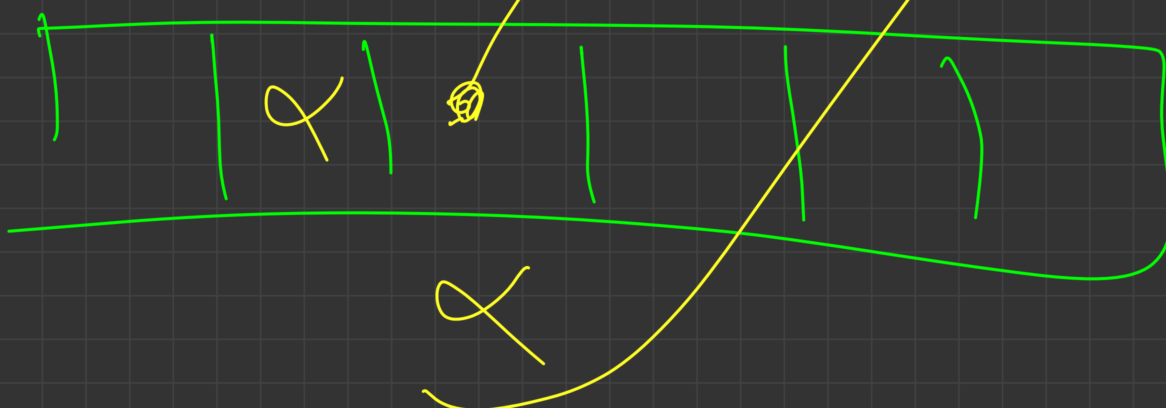
Sum = 13

$dp[i][0]$ = you are picking elements from left to right, you skipped the last element, what is the maximum score that you can get from

$(i — n-1)$

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

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



Some ways to solve the problem

1. Having 2 parameters to represent the state

State:

(i to n-1)

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]

Some ways to solve the problem

2. Having only 1 parameter to represent the state

State: i to n-1

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

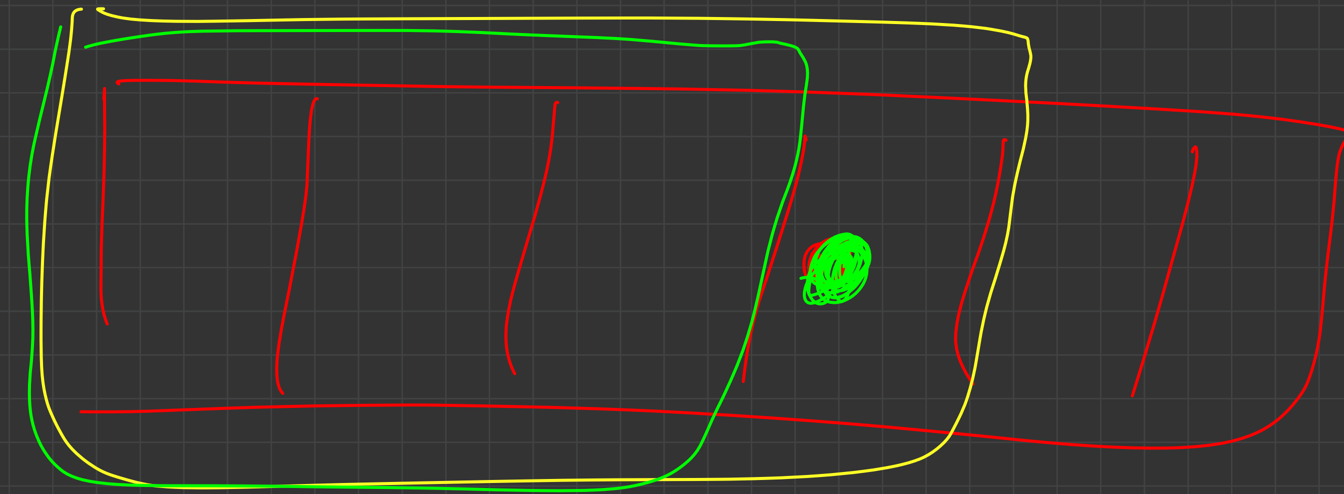
Transition: 2 cases

- 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~~]
0



dp[i]

dp[i-1]

arr[i] + dp[i-2]

```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;
}
```

Time and Space Complexity in DP

Time Complexity:

Estimate: Number of States * Transition time for each state

Exact: Total transition time for all states

Space Complexity:

Number of States * Space required for each state