Dynamic Programming 1.2

- Priyansh Agarwal

Recursive vs Iterative DP

Memoization

```
int f(int n){
    if(n==1 || n==2)
        return 1;
    if(dp[n]!=-1 default value)
        return dp[n]
    return dp[n] = f(n-1) + f(n-2)
}
```

dp[n]

-1 == actual value

$dp[1] = 1$, $dp[2] = 1$

```
for ( int i = 3 ; i <= n ; i++)
```

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

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

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

Recursive

Iterative ←

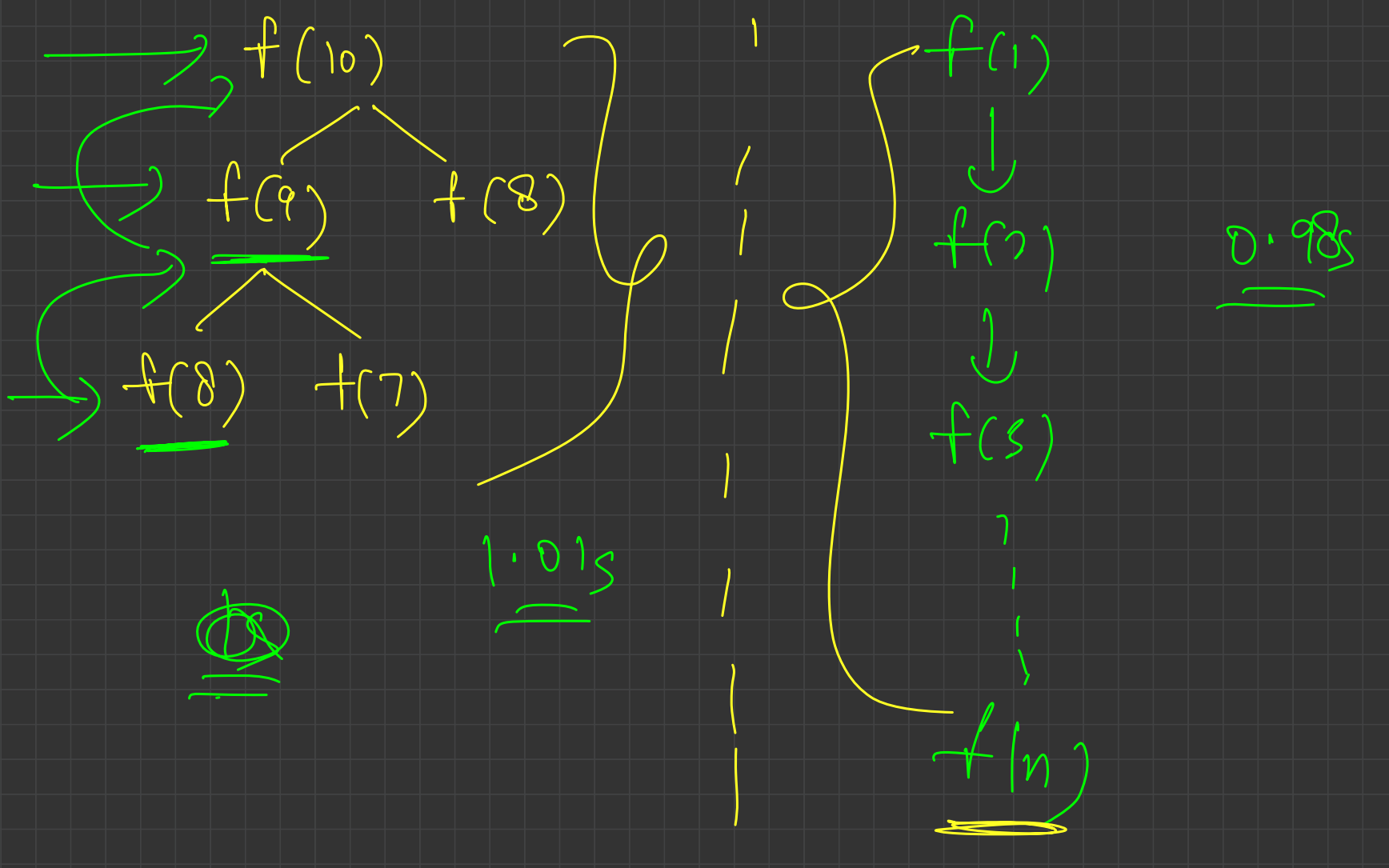
dp(i)(j)(k)

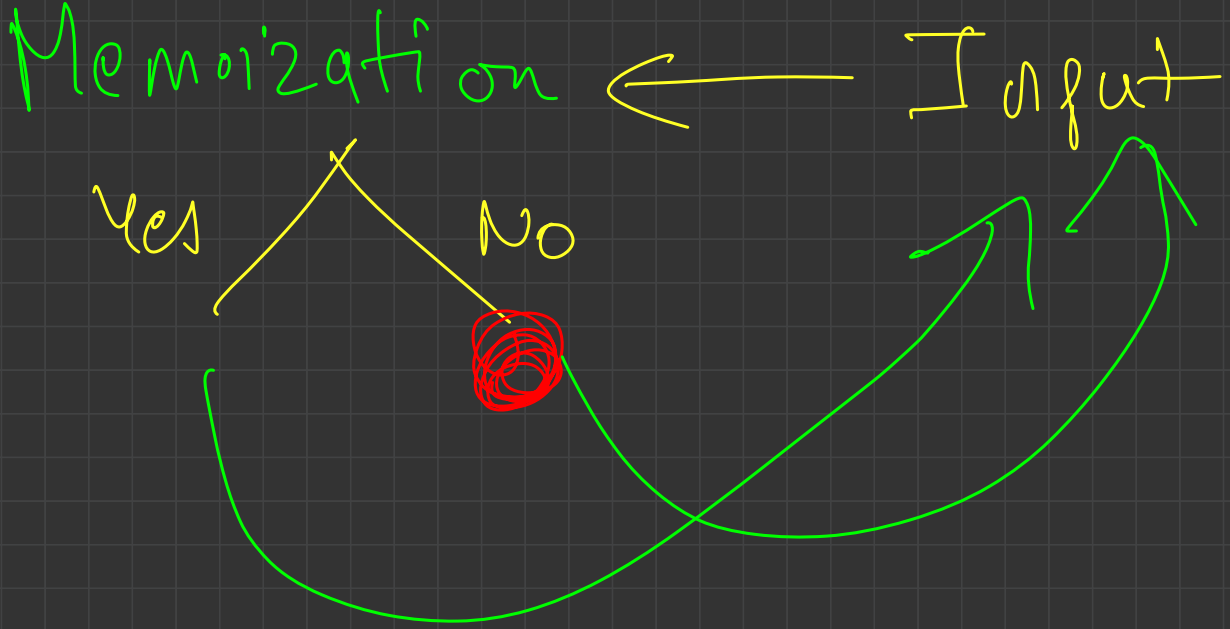
= f(i-1, j, k ...)

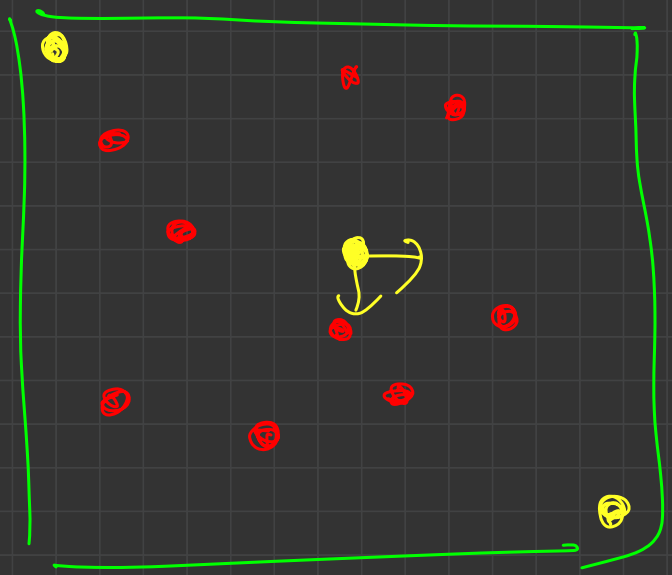
dp(i)(j)(k)

= dp(i-1)(j)(k

flow of states


f(10)
f(9)
f(8)
f(8)
f(7)
f(1)
f(2)
f(5)
f(n)
1.01s
0.98s


Memoization
Input
Yes
No



$10^5 \times 10^5$

$n \times n$

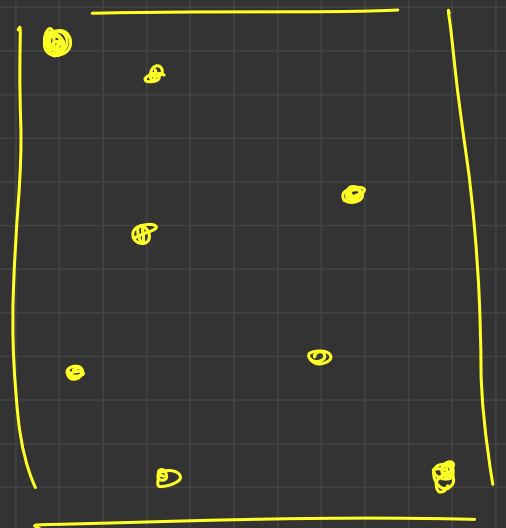
$= n^2$

$= n^2$

find # of ways to go from Top left to bottom right

$dp[i][j]$ = # of ways to go from (i, j) to $(n-1, m-1)$

$$\hookrightarrow dp(i)(j) = dp(i+1)(j) + dp(i)(j+1)$$



$10^5 \times 10^5$

imaginary grid

with 99.99% cells

filled with obstacles

Given a list of valid points

$\frac{10^{10}}{100 \times 100} = 10^6$

```
map < pair<int, int>, int> dp
set < pair <int, int> validpoints

int f(int i, int j)
    if (i==n || j==n)
        return 0;
    if (validpoints.find({i, j}) != vali.end())
        return 0;
    if (dp.find(i, j) != dp.end()) ==
```

return dp[(i,j)]

$dp[(i,j)] = f(i+1,j) + f(i,j+1)$

return dp[(i,j)]

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

Flow of States

State , Transition

L.H.S R.H.S

$dp(i) = dp[i-1] + dp[i-2]$

$dp[2] = 1$

$dp[1] = 1$

```
for( i=3 ; i<=n ; i++)
```

①

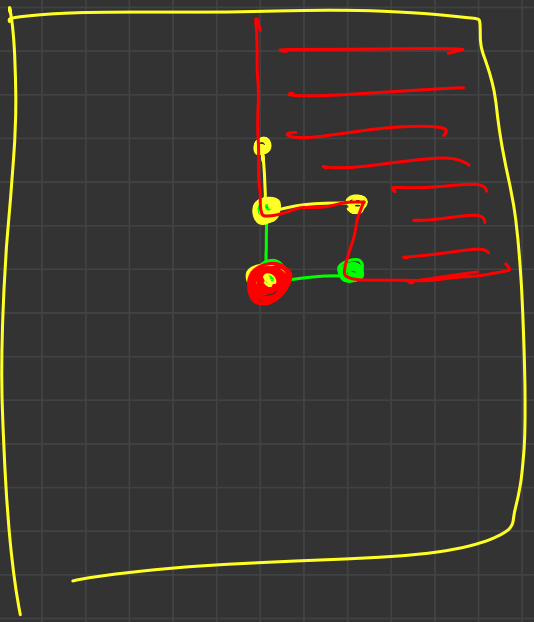
$dp(i) = dp(i-1) + dp(i-2)$

```
for( i=n ; i>=3 ; i--)
```

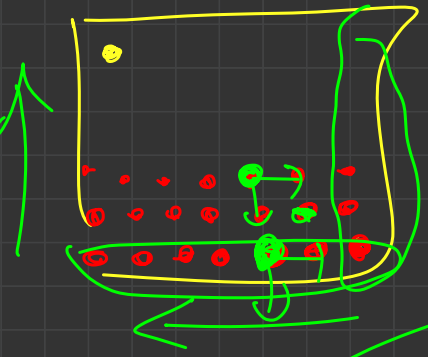
$dp(i) = dp(i-1) + dp(i-2)$

②

$$dp[i][j] = dp[i-1][j] + dp[i][j+1]$$



```
for(int i = 0; i < n ; i++)
    for(int j = m-1; j >= 0; j--)
        dp[i][j] = dp[i-1][j]
                 + dp[i][j+1]
```

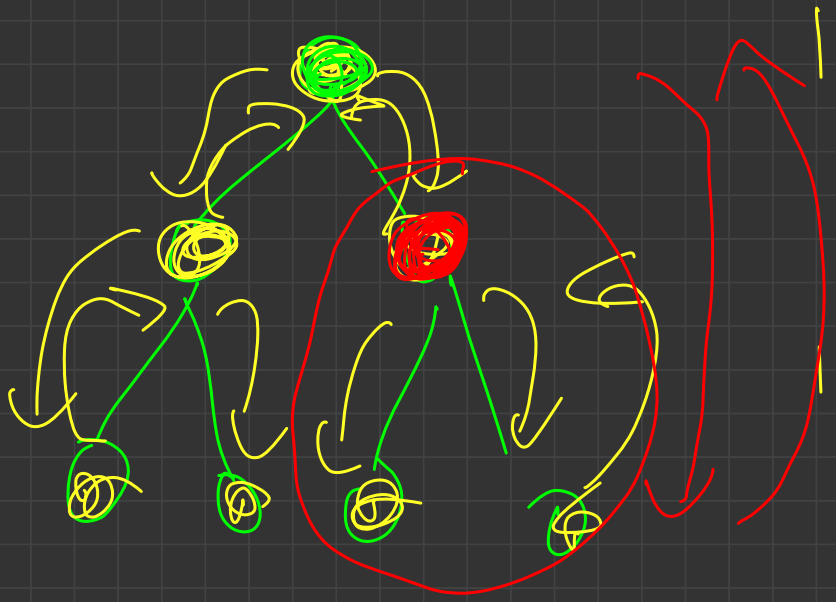


dp[i][j] = min. sum path from (i, j)
to (n-1, m-1)

dp[i][j] = grid[i][j] + min { dp[i+1][j], dp[i][j+1] }

dp[n-1][m-1] = grid[n-1][m-1]

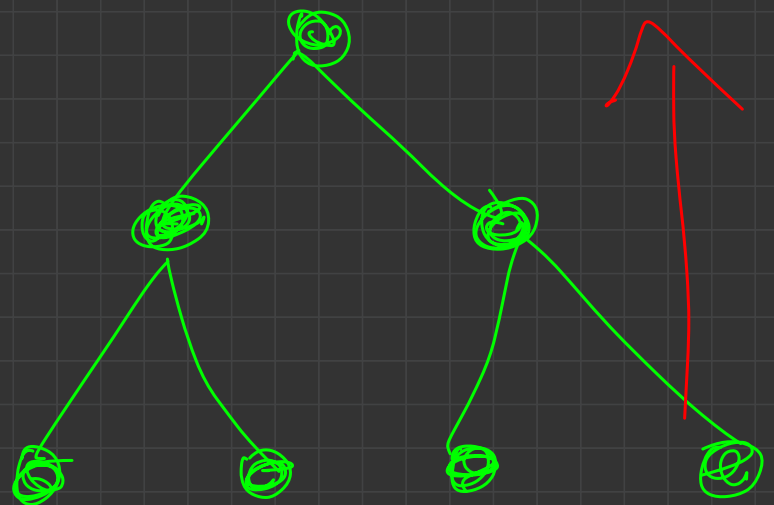
for (int i = n-1 ; i ≥ 0 ; i--)
  for (int j = m-1 ; j ≥ 0 ; j--)



slower

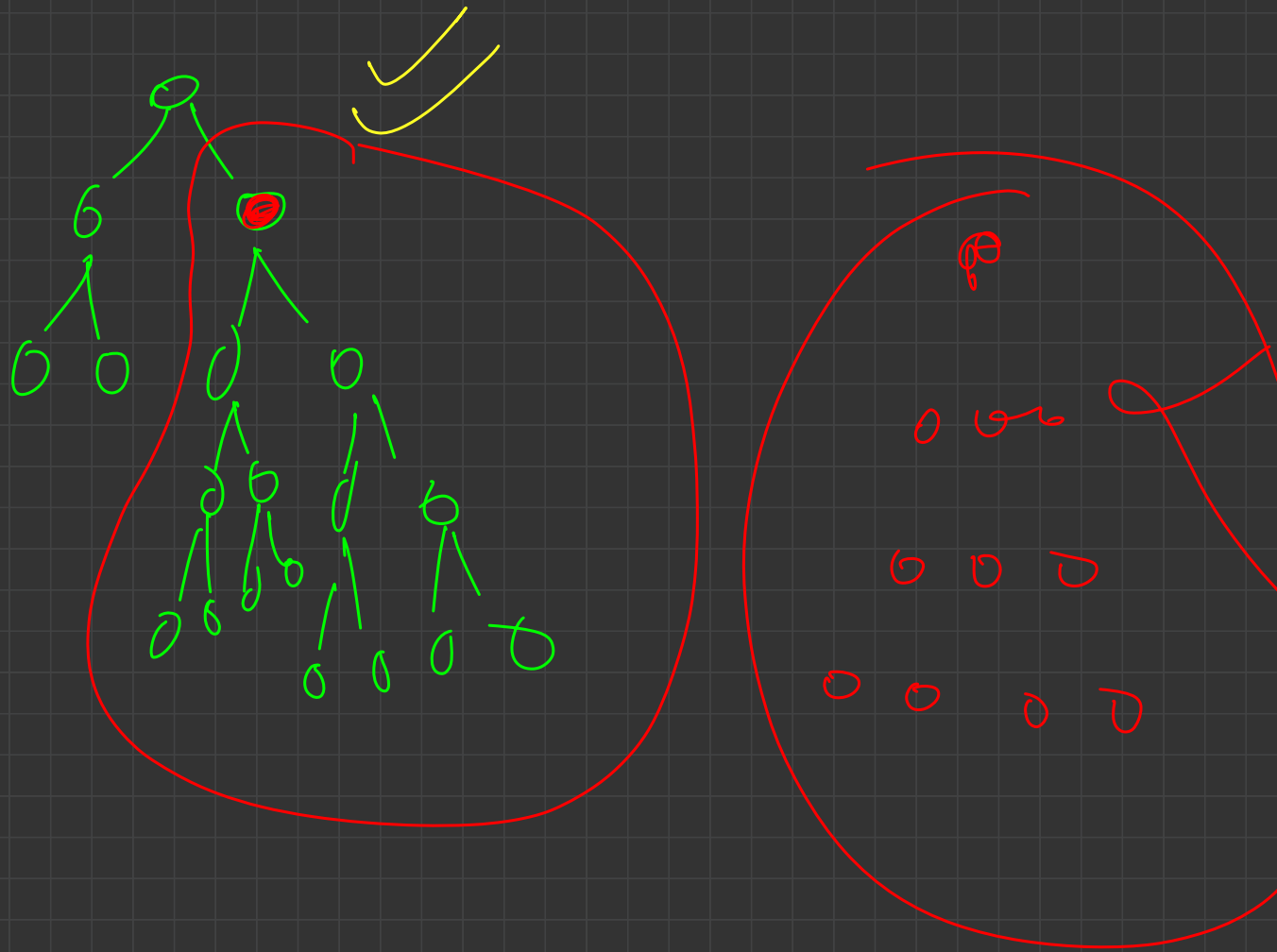
Easier to think | diff to think

diff to code |



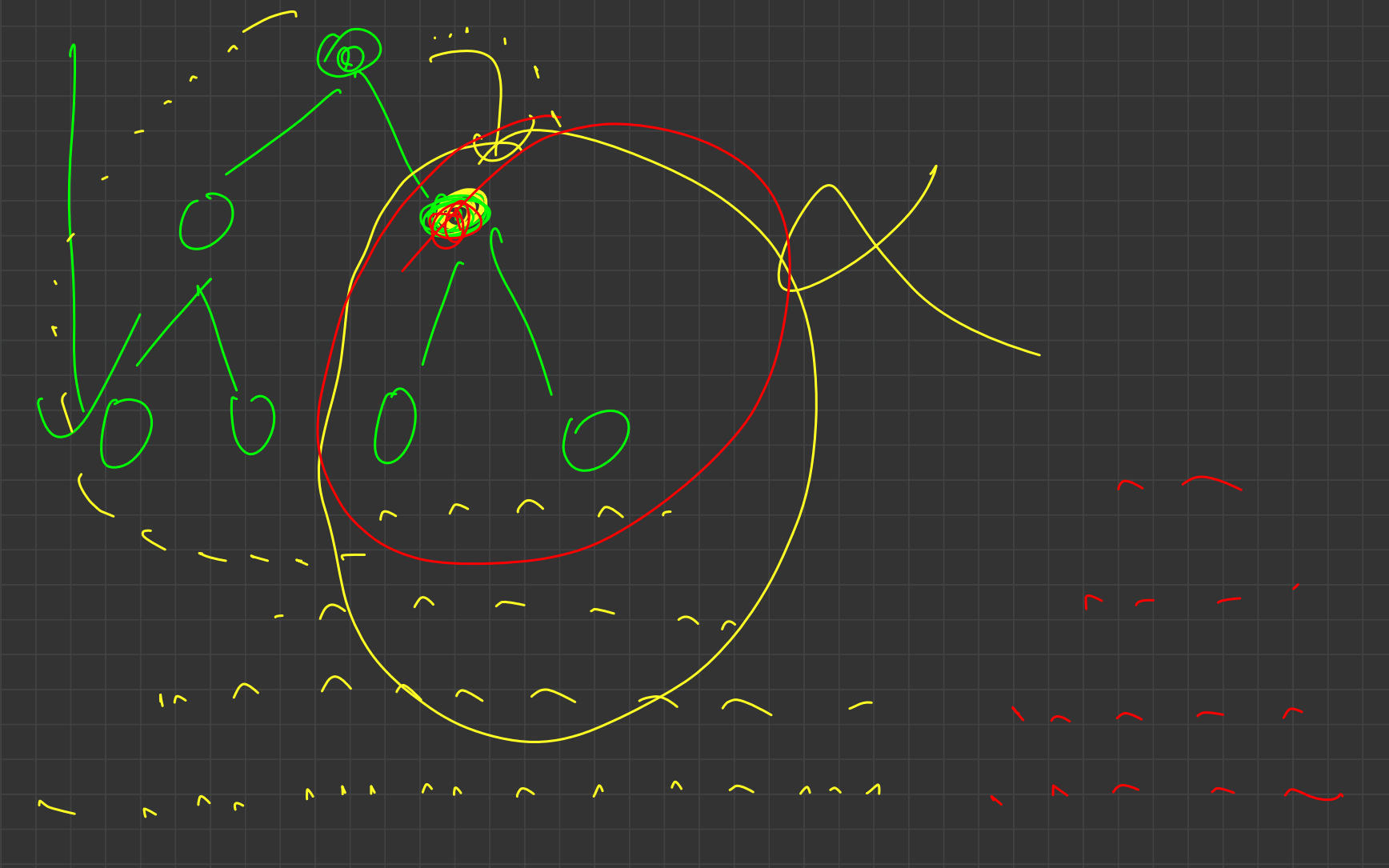
faster

easier to code



Recursive

Iterative



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.

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.

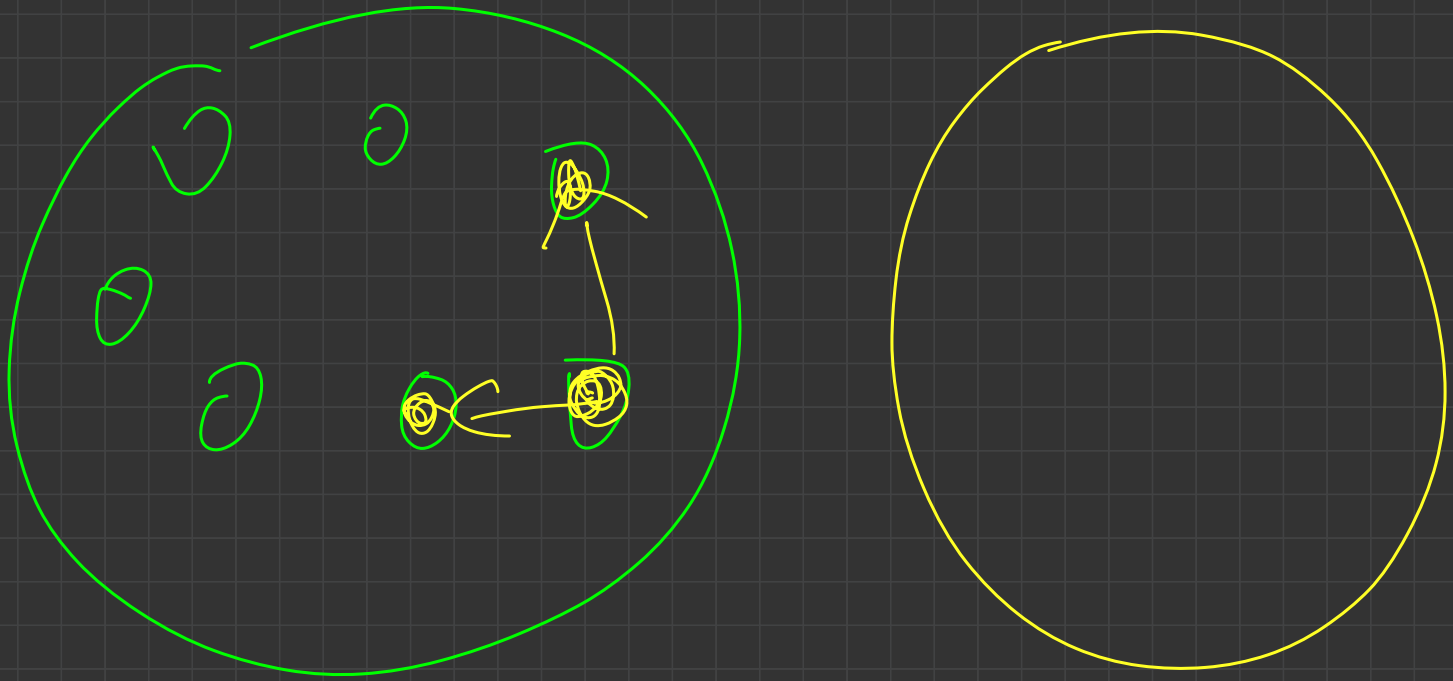
4. Final Subproblem

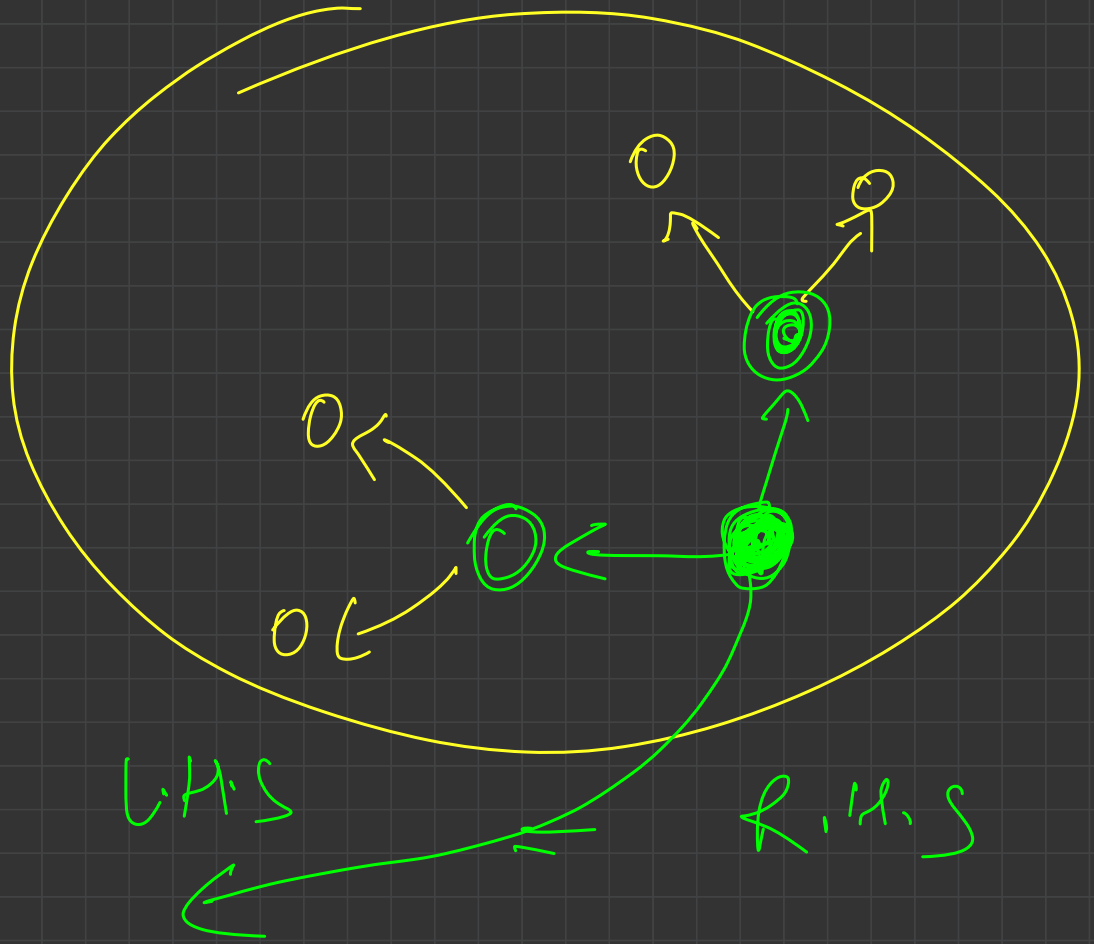
   What is the problem demanding you to find?

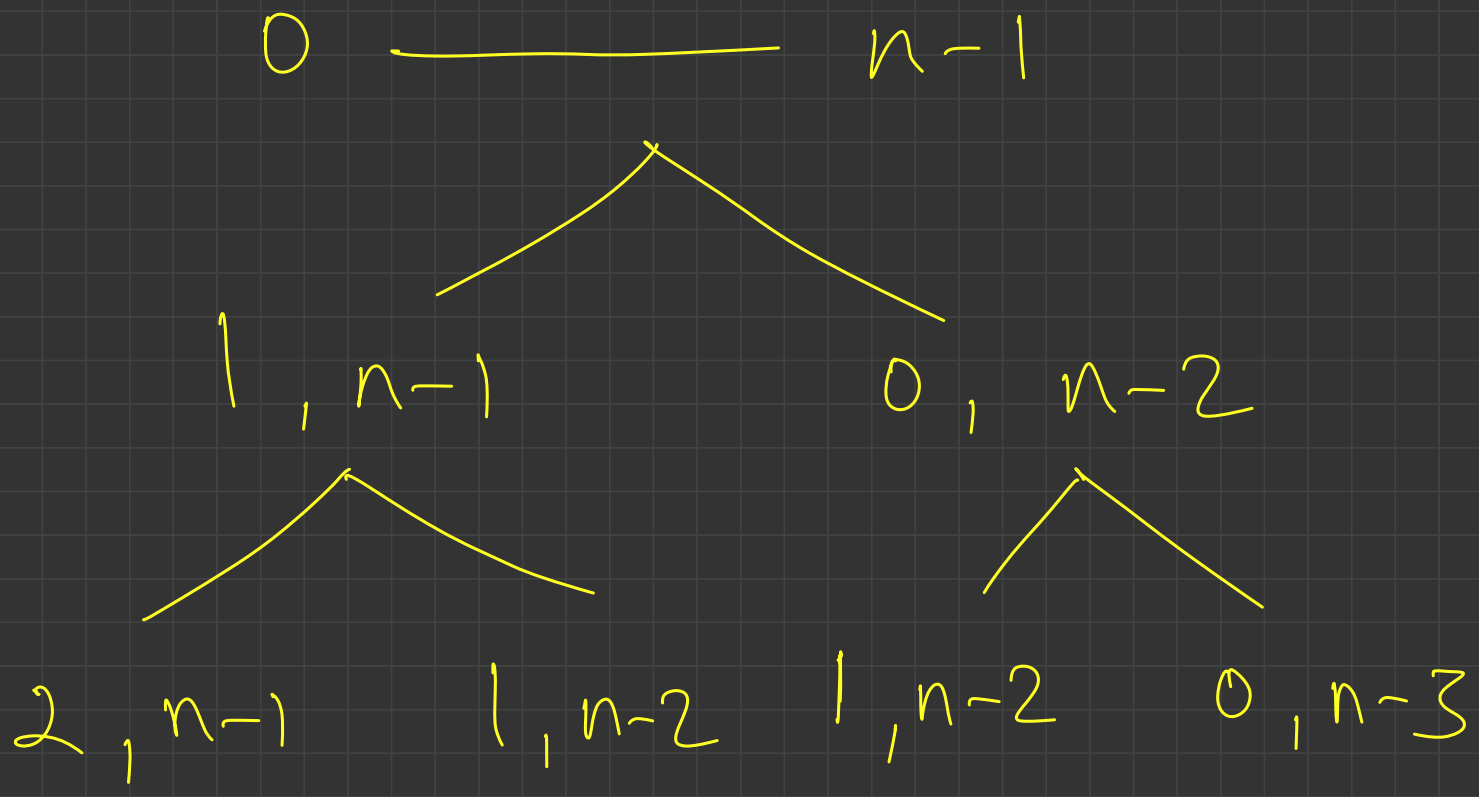
$$dp(i)(j) = dp(i+1)(j) + dp(i)(j+1)$$

$i = n-1$ $\quad 0 \rightarrow n-1$

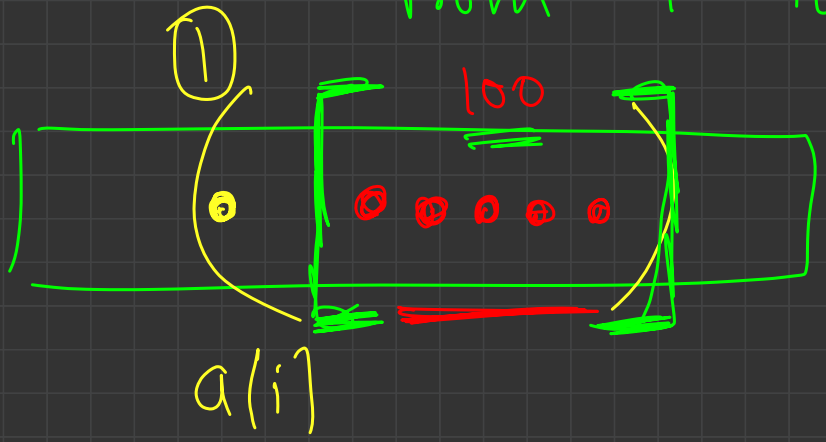
$j = m-1$ $\quad 0 \rightarrow m-1$



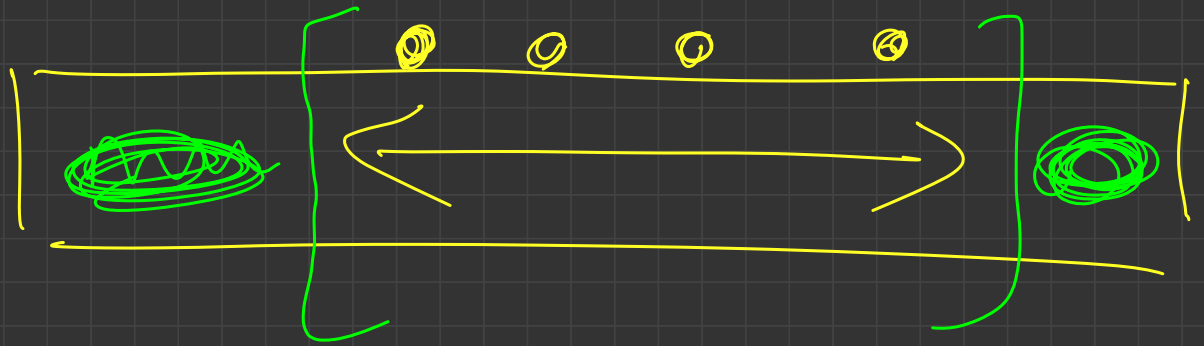

L.H.S
R.H.S


0 n-1
1, n-1
0, n-2
2, n-1
1, n-2
1, n-2
0, n-3

$dp[i][j]$ = max score that current player can get when the array from i to j is remaining



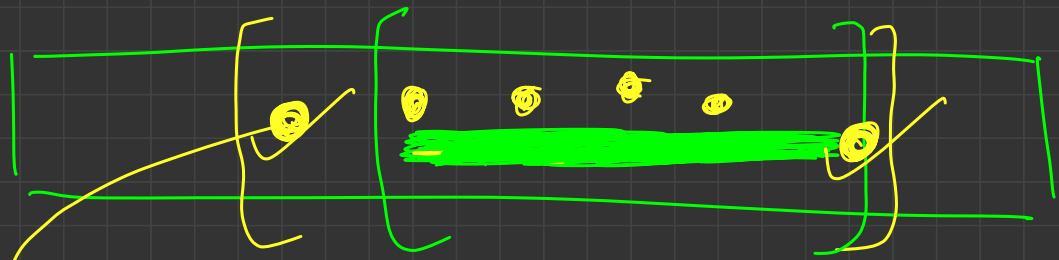
$dp[i+1][j] = 60$



$dp[i][j]$

$i \neq j$

$i = j$



i

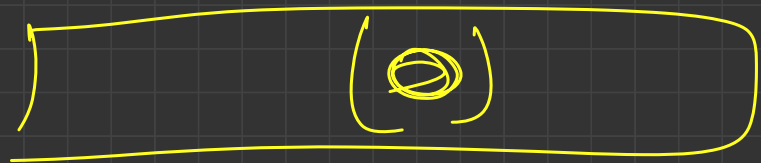
j

max

a[i] + (sum[i+1][j] − dp[i+1][j])

a[j] + sum[i][j−1] − dp[i][j−1]

$dp[i][j]$ $\quad i == j$



$dp[i][i] = a[i]$

$\rightarrow dp[0][n-1] \leftarrow$

$(sum[0][n-1] - dp[0][n-1])$

Problem 1: Link

Problem 2: Link

Problem 3: Link