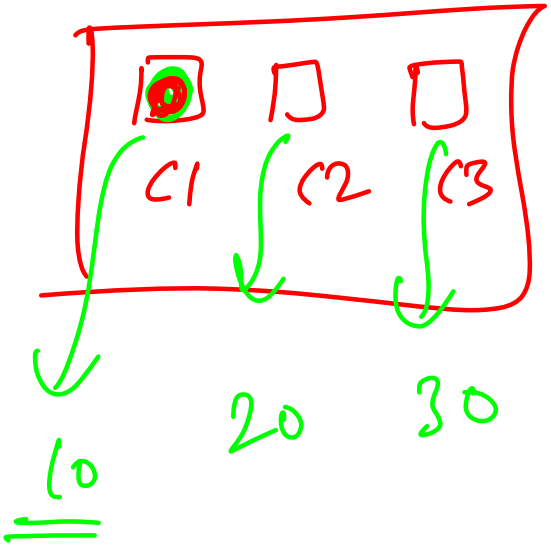
Dynamic Programming 1.1

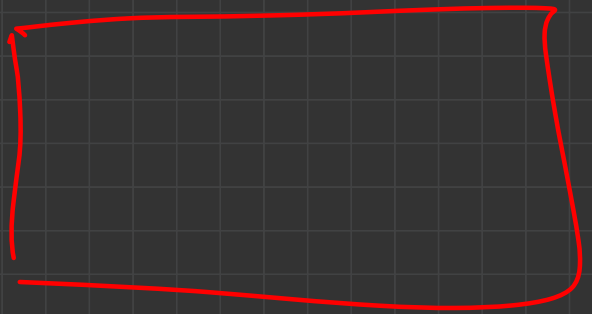
- Priyansh Agarwal

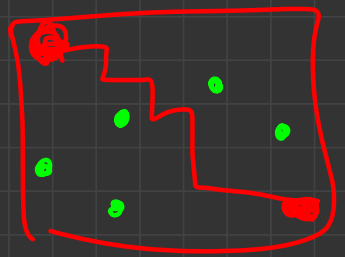
Why Dynamic Programming?

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



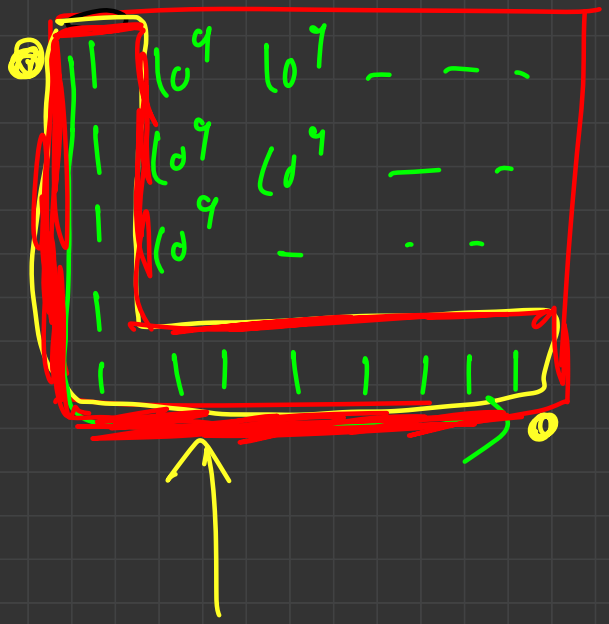
$25^2 = 625$

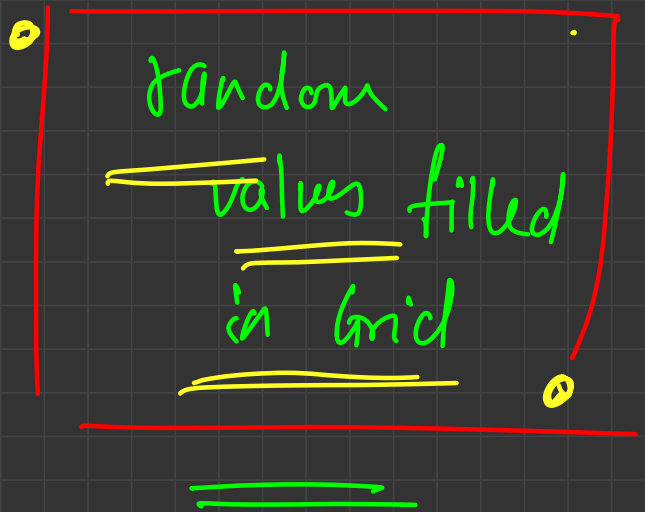




Greedy

DP



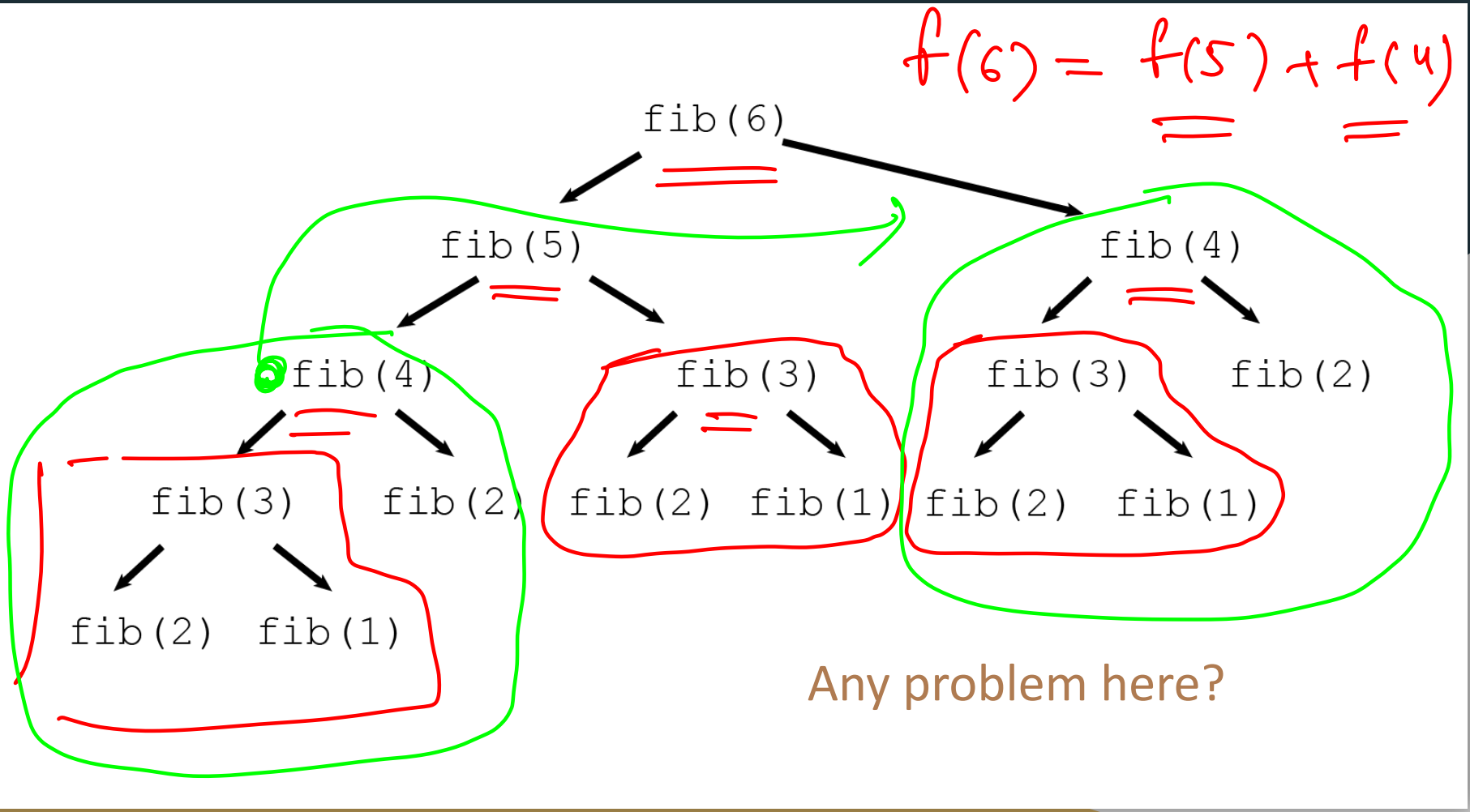


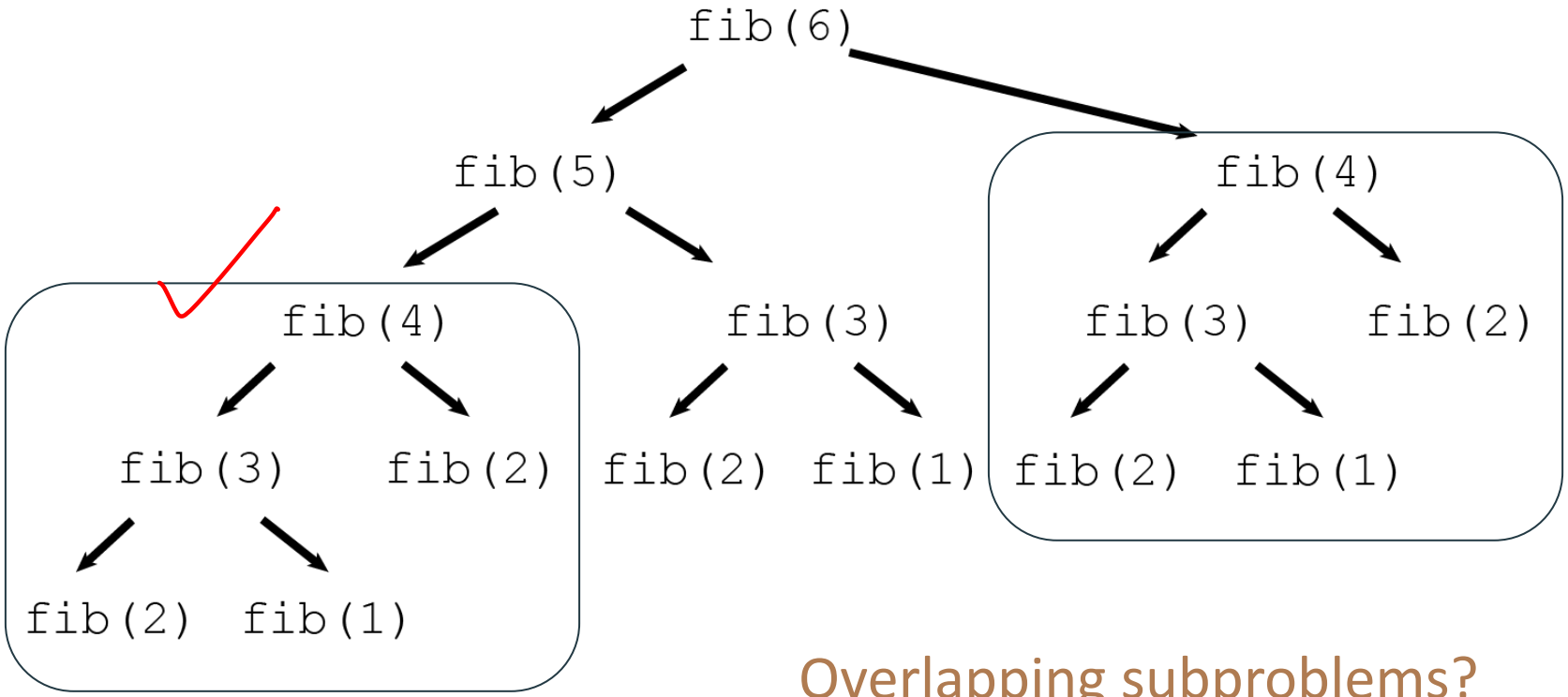
Greedy

DP

Need of DP

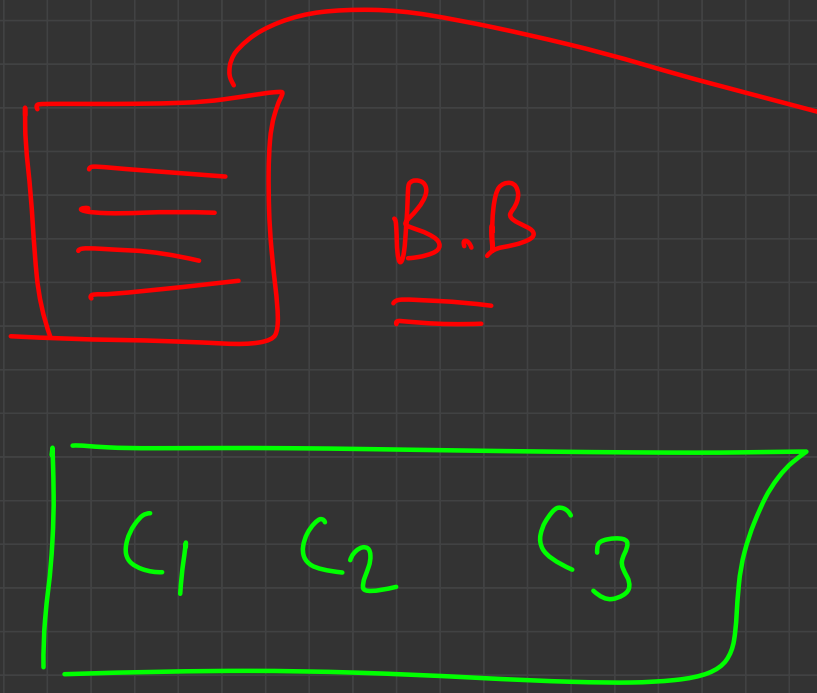
- Let’s understand this from a problem
  - Find n^{th} fibonacci number
  - $F(n) = F(n - 1) + F(n - 2)$
  - $F(1) = F(2) = 1$

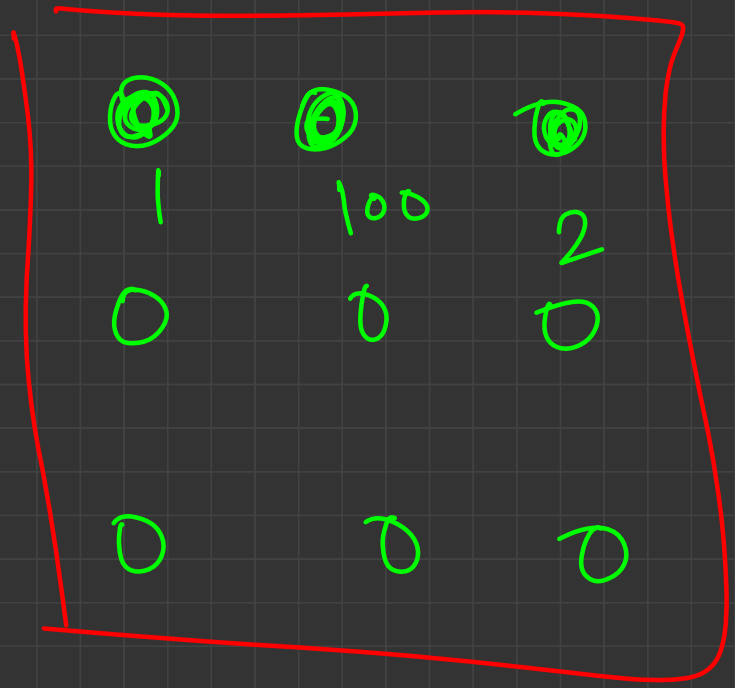

f(6) = f(5) + f(4)
fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(2)
fib(1)
Any problem here?



Overlapping subproblems?

Memoization : Memorization





$1 \rightarrow x$

$100 \rightarrow y$

S1 . x

S2 . y

S3 . 2

S4 . 9

S5 . 5

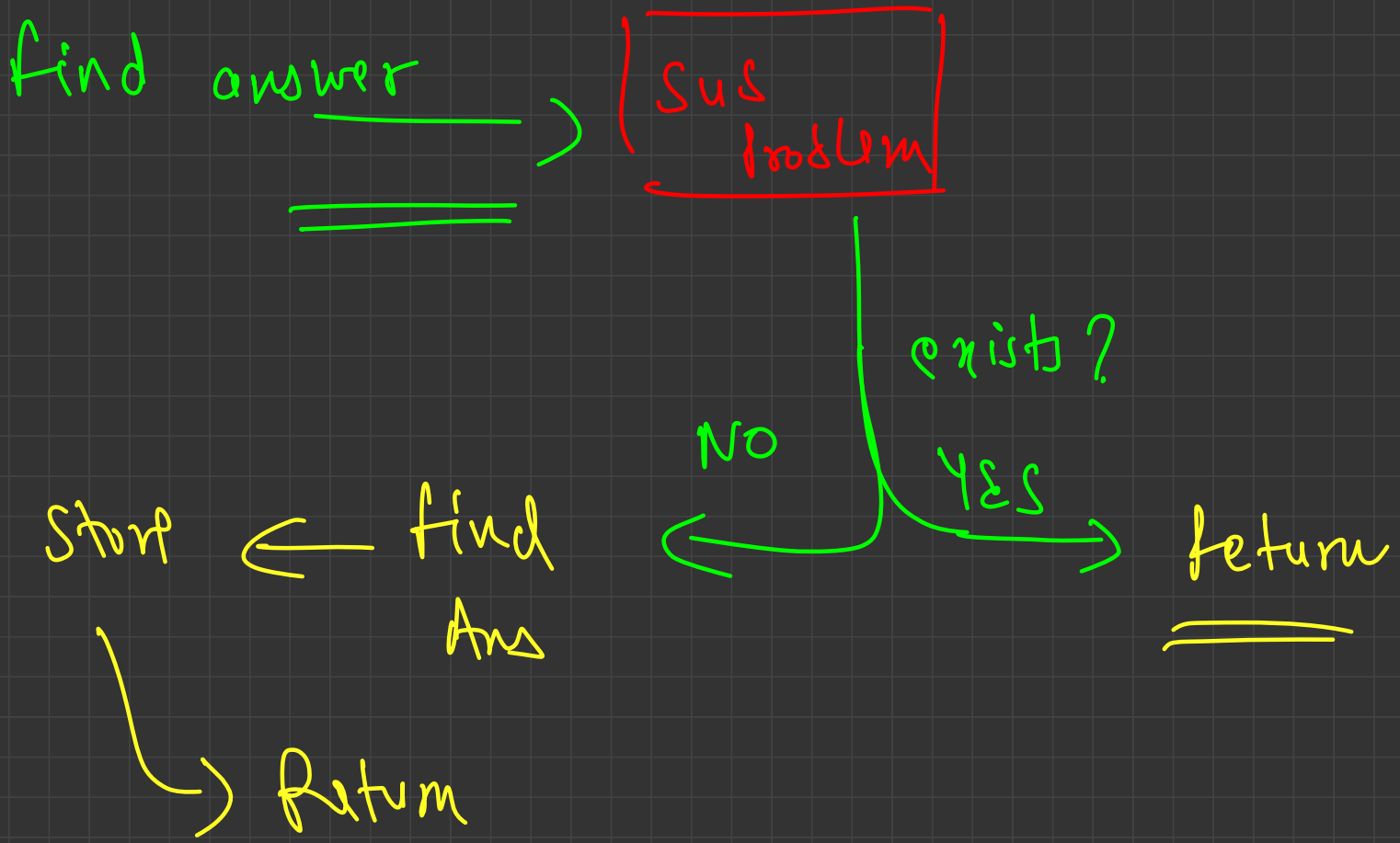
S1 → x

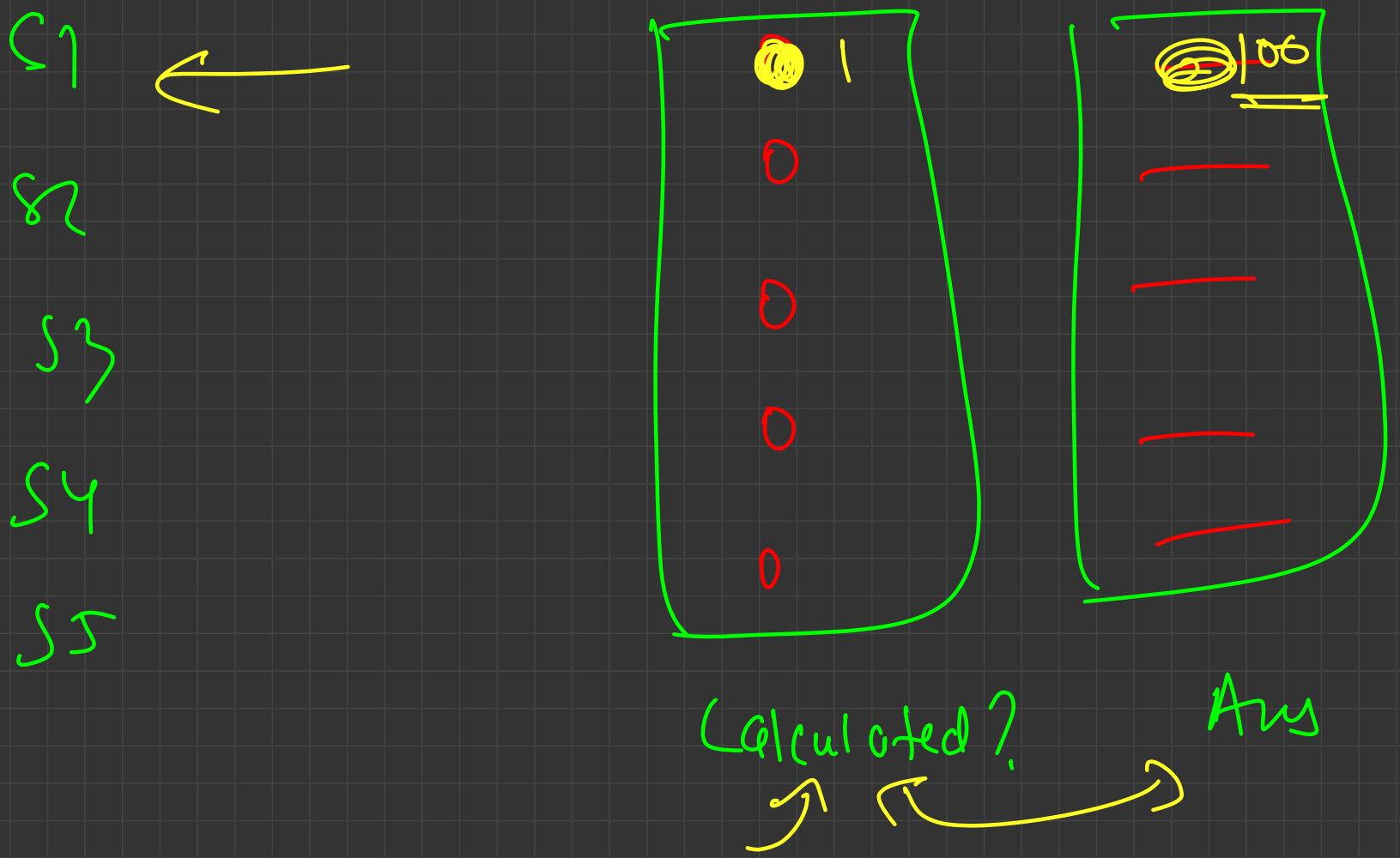
S2 → y

S3 → 2

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


find answer
Sus problem
exists ?
NO
YES
Stop
find Ans
return
Return


S1
S2
S3
S4
S5
1
0
0
0
0
106
Calculated?
Ans

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

6 5 4 3 2 1

Calculated	0	0	0	1	1	1
	6	5	4	3	2	1

Ans	_	_	_	2	1	1

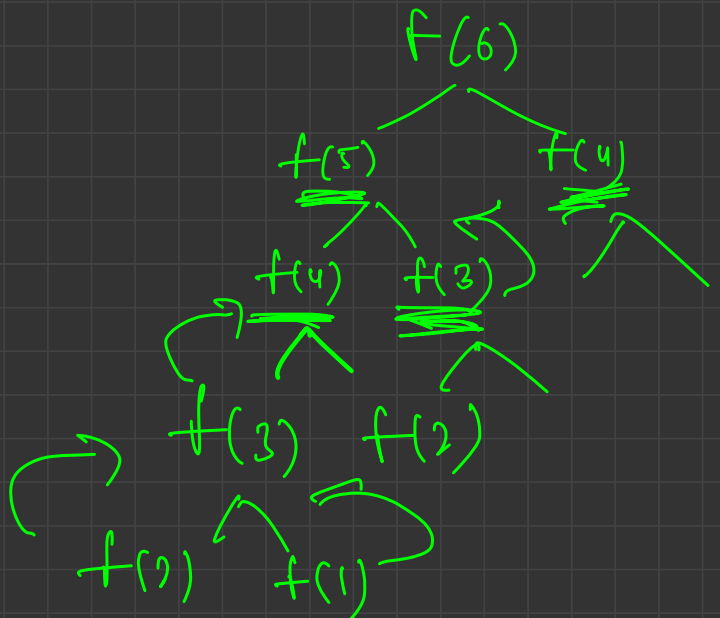
(−100, 100)

Ans

Default Value

Ans —)

1	1	2	3	5	8
1	2	3	4	5	6



Without DP

```
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;
}
```

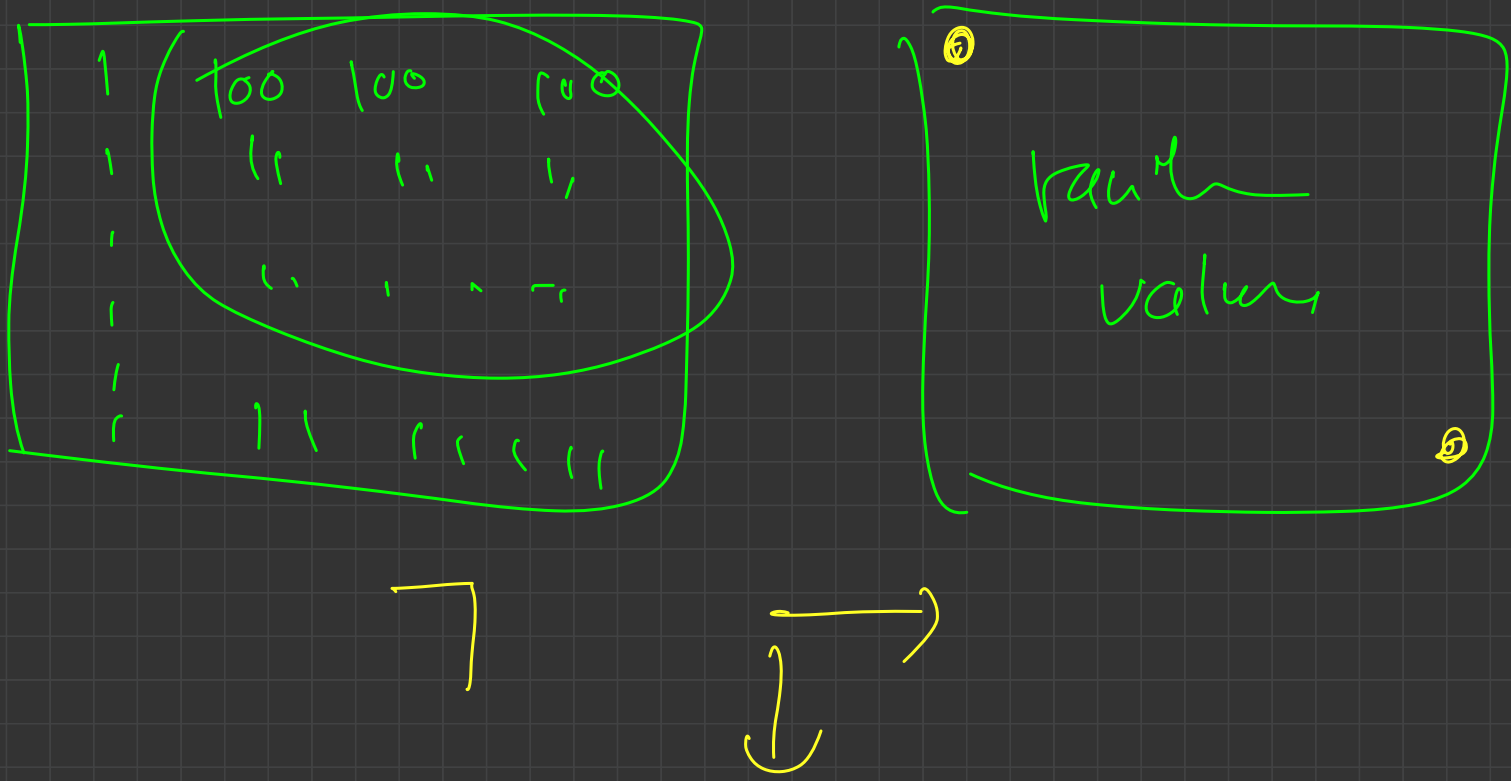
functionEntered = 1664079
with n = 30

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;
}
```

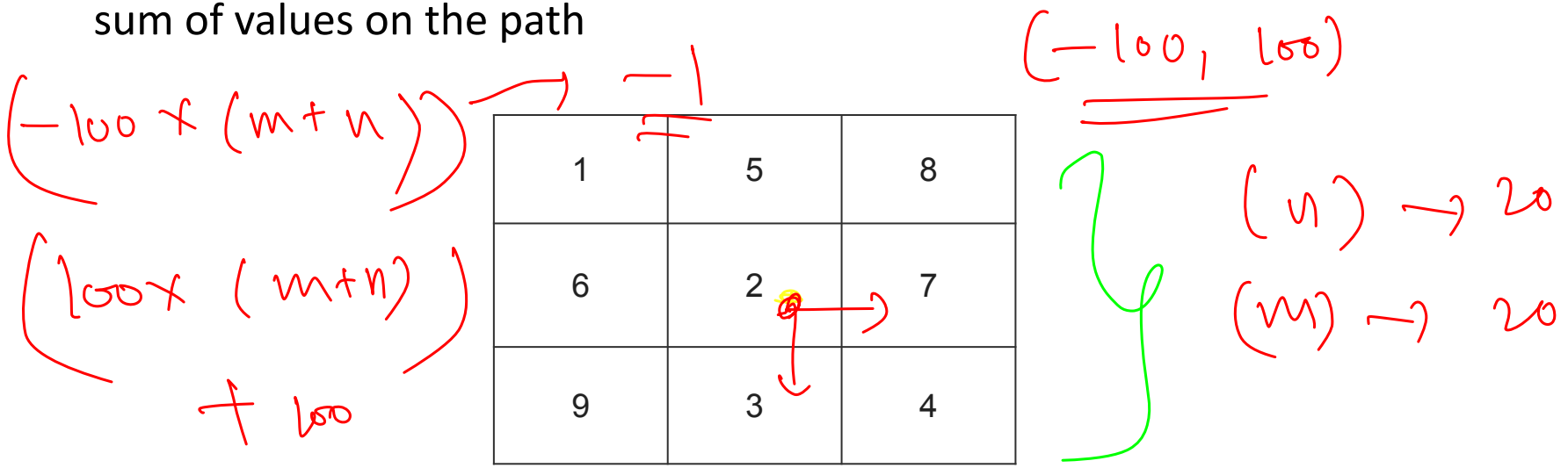
-1 -1 -1 -1 -1

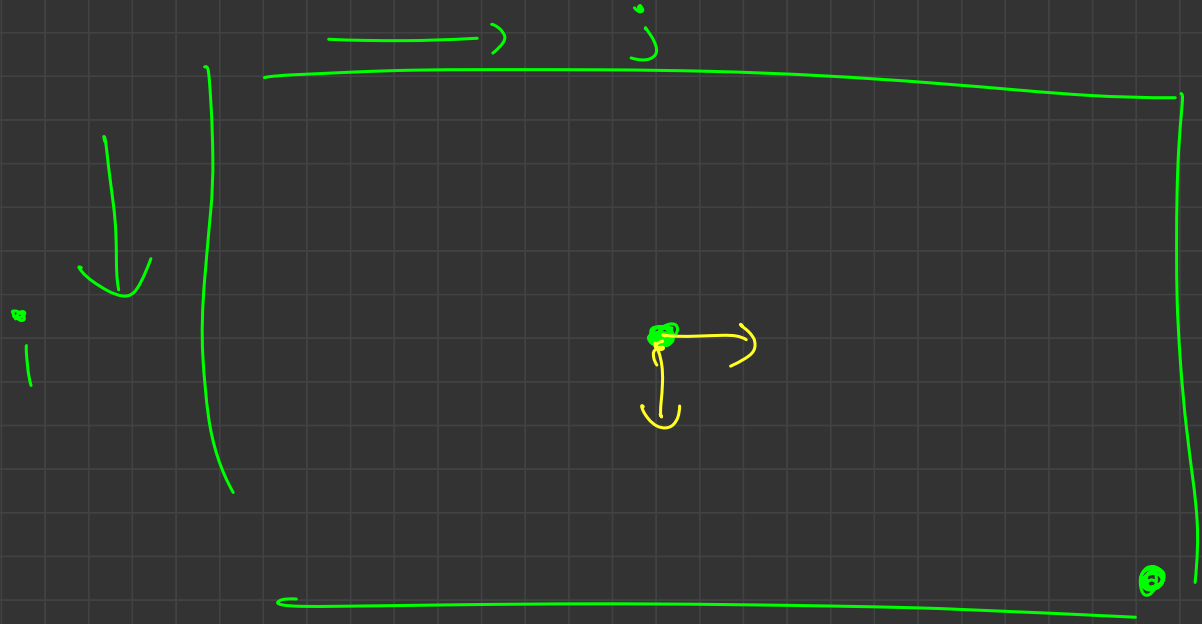
functionEntered = 57
with n = 30


100 100 100
0
0

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





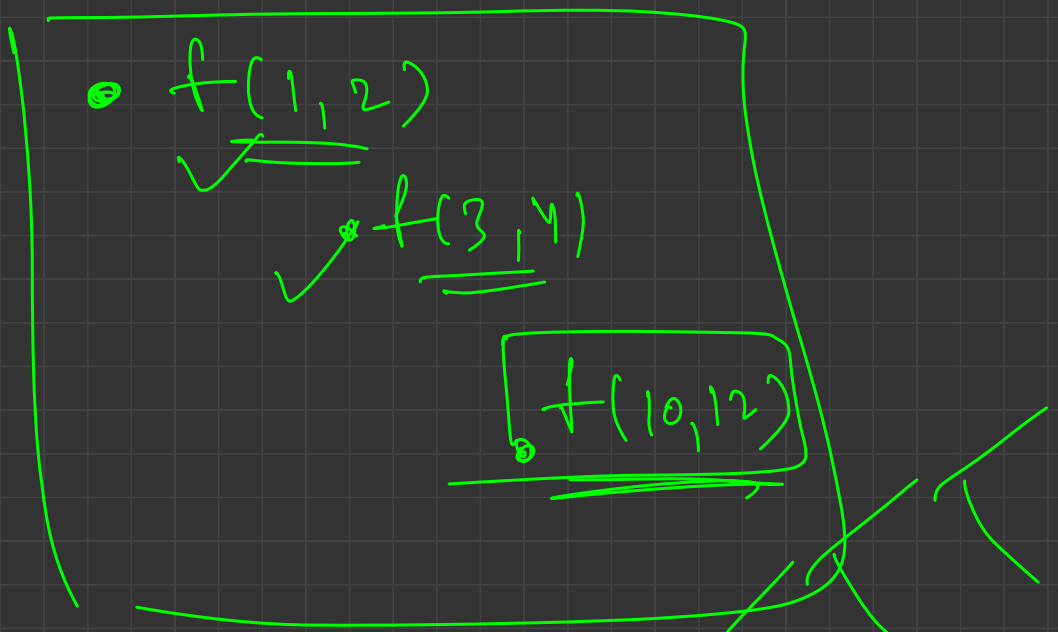
$f(i, j)$ = minimum sum path from (i, j) to bottom right

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

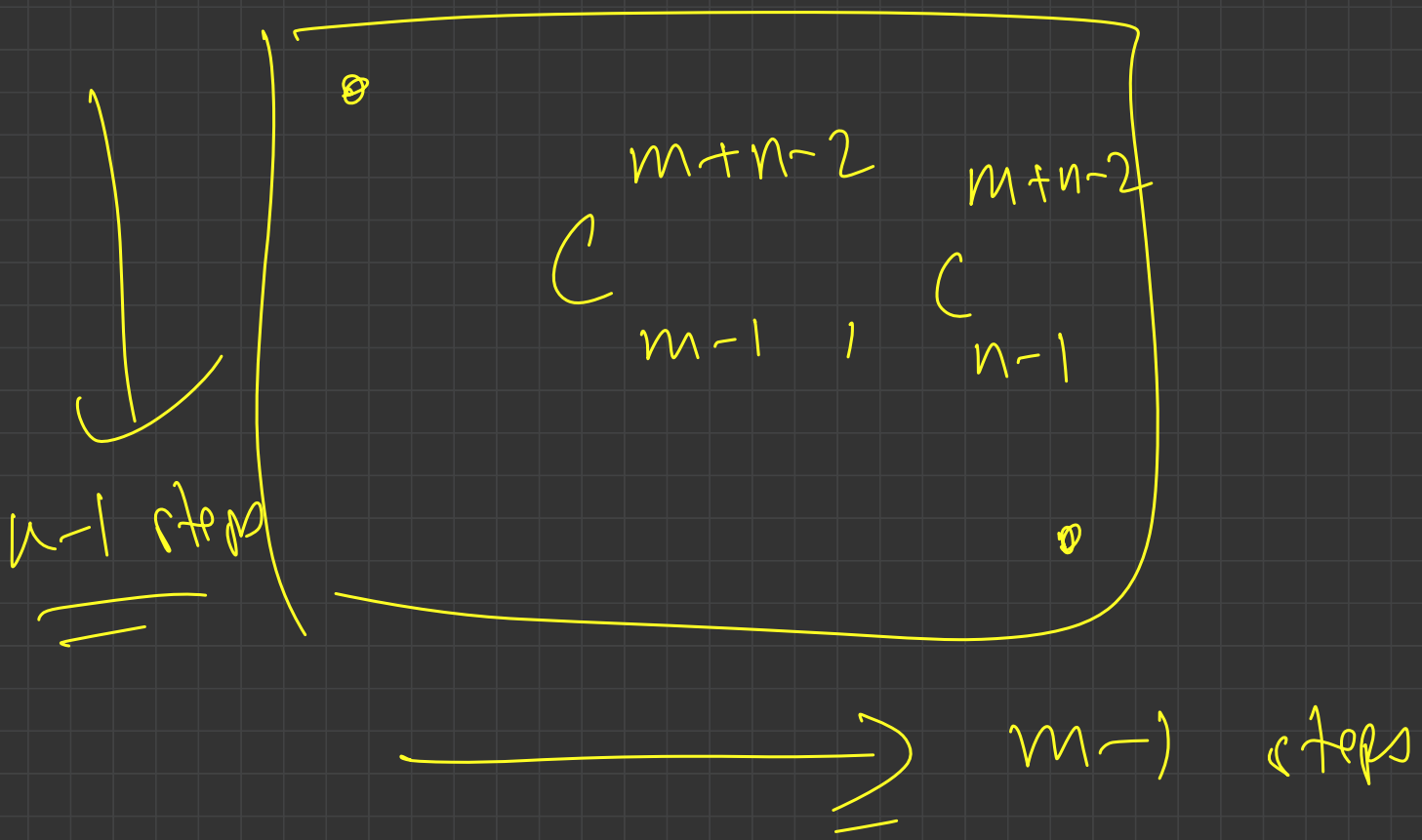
$m+n$

$2^{m \cdot n}$

2^{m+n}



$f(1,2) < f(2,2) < f(1,3)$



$\Rightarrow$ m-1 steps

D R D D R R R D D D

R → m-1

D → n-1

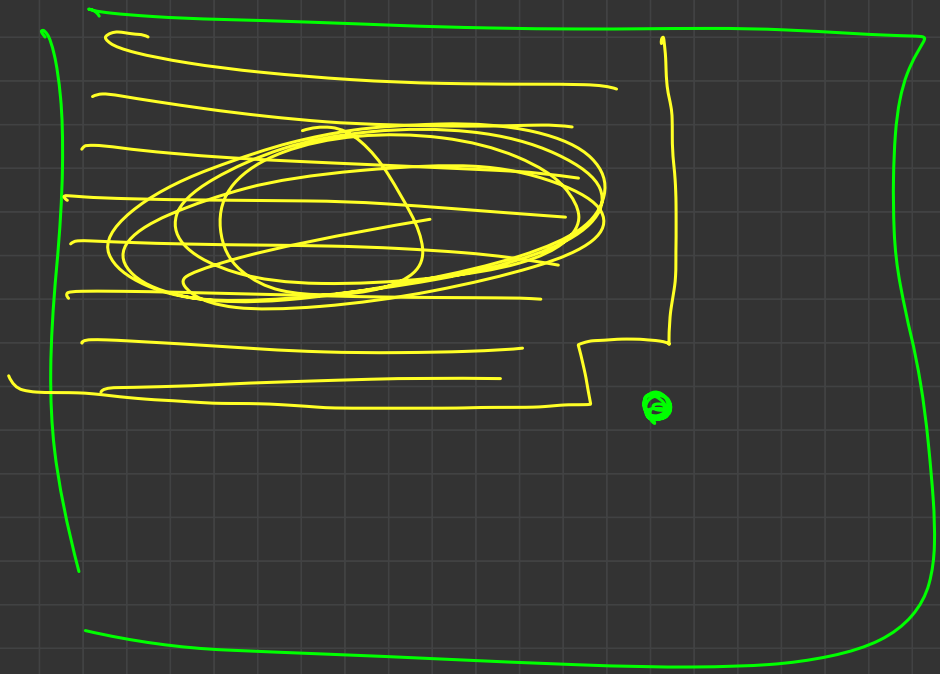
a a a a a a 6

b b b 3

$$\frac{(6+3)!}{6!\,3!}$$

$$6\,\frac{(m+n-2)!}{(n-1)!\,(m-1)!}$$

$$= {}^{m+n-2}C_{n-1}$$



$f(100, 200)$

100×200

$20,000$

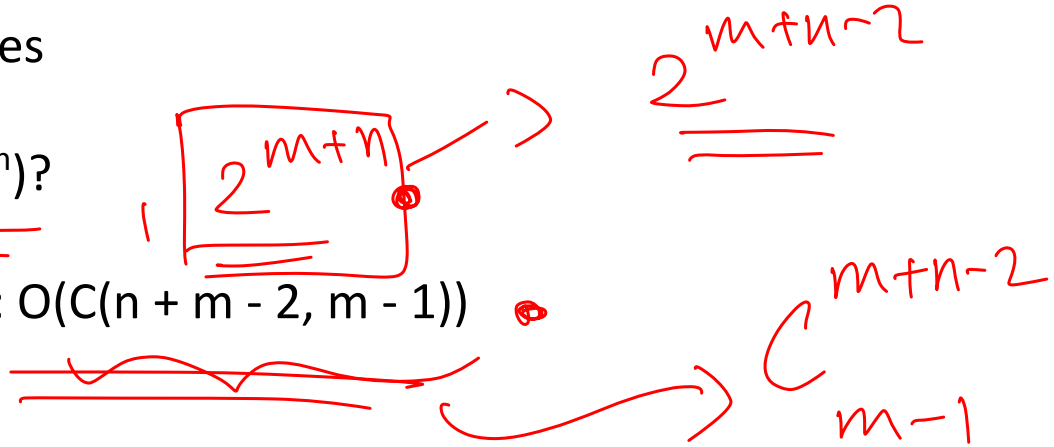
Naive Way

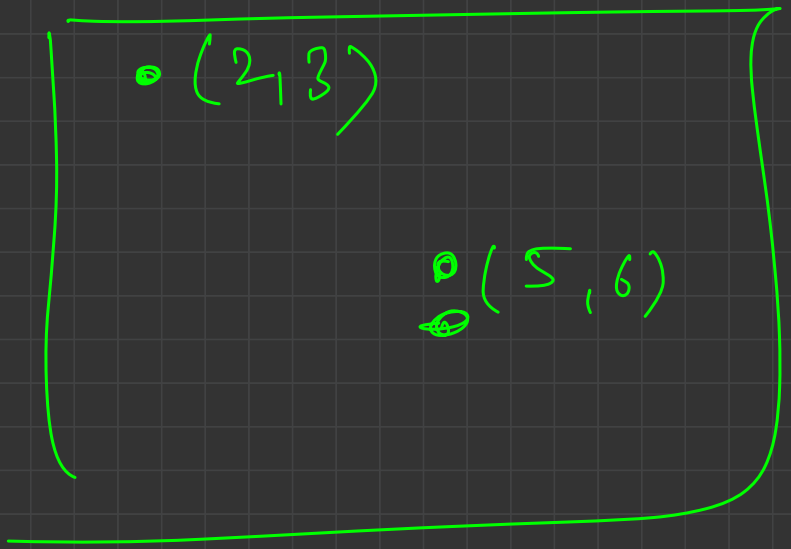
Explore all paths. Standing at (i, j) try both possibilities (i + 1, j), (i, j + 1)

Every cell has two choices

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

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





f(6) -> 6

(i, j)

dp[n][m]

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;
}
```

0 based indexing

f(i j)

```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;
}
```

counter = 0

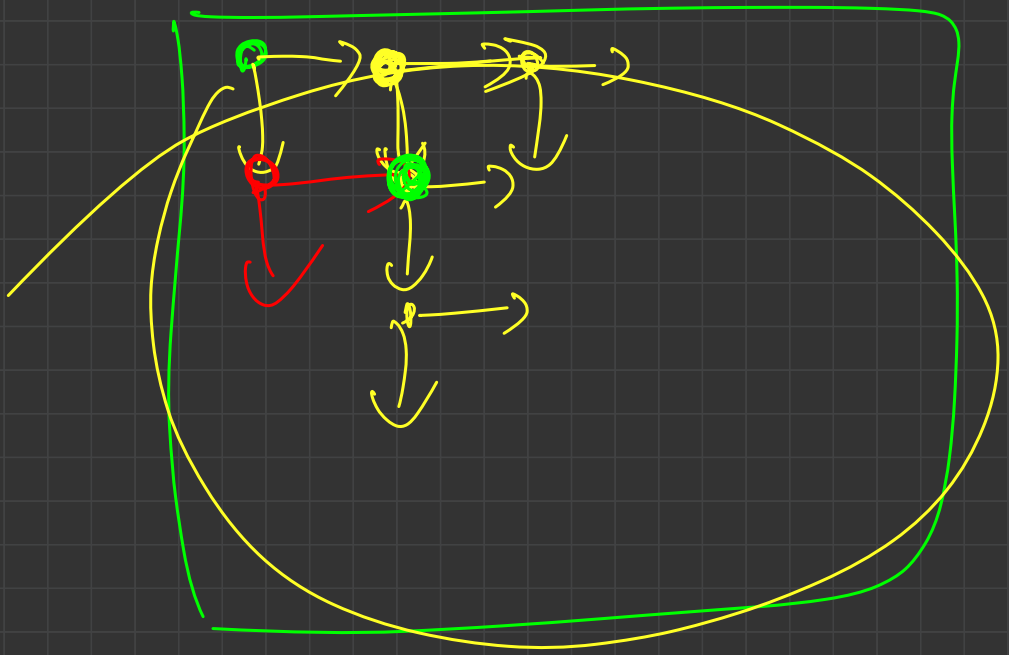
counter ++

K. (counter incr). work done in 1 time

$O\left(k \cdot (n \cdot m) \cdot O(1)\right)$

constant

$O(n \cdot m)$



$m \cdot n$

Important Terminology

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.

nth fibonacci

f(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.

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

Transition Equation

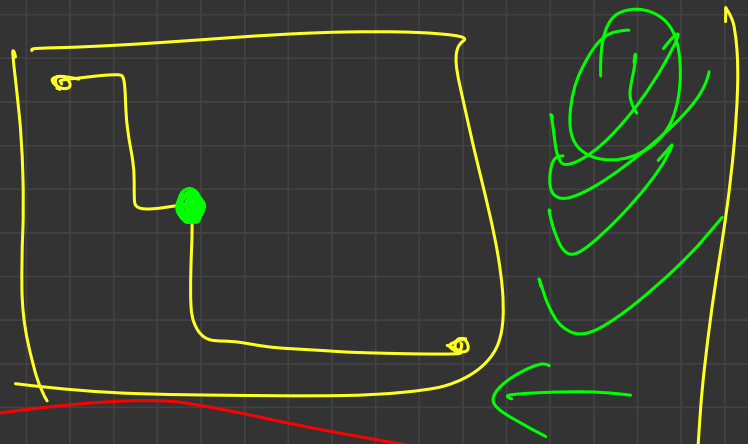
L.H.S = R.H.S

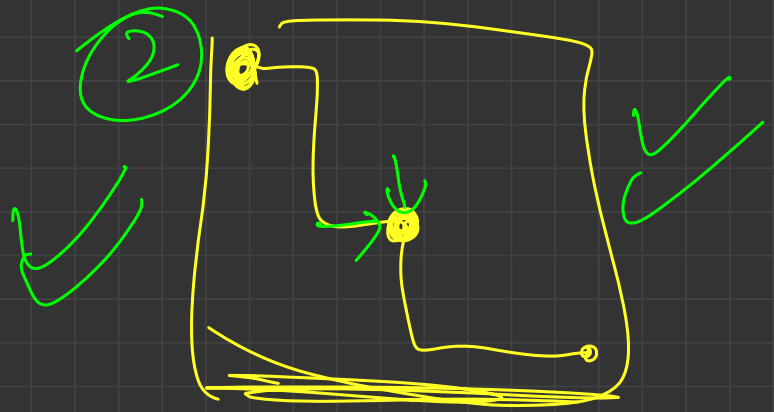
$$f(n) = f(n-1) + 2f(n-2) + 5f(n-5)$$

Exercise

Fibonacci Problem:

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





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

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

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

dp(0)(0)

dp(i)(j) = grid(i)(j) +
min { dp(i-1)(j)
dp(i)(j-1)

dp(n-1)(m-1)

top down
bottom up

State : dp(i)(j) ∝

f(i,j) ∝ dp(row)(col) ∝

i, j ∝ dp(i)(j) = ——— ∝

dp(i)(j) = min cost of path from (i, j) to bottom right

f(i, j)

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])
- final suppostion
  dfs

Let’s solve another problem

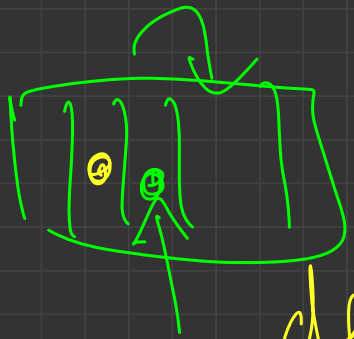
Given an array of integers (both positive and negative). Pick a subsequence of elements from it such that no 2 adjacent elements are picked and the sum of picked elements is maximized.

Sum = 14

Sum = 13

State :

$dp[i][0]$ = max sum from (i to n-1) such that (i-1) was not picked



$dp[i][1]$ = max sum from (i to n-1) such that (i-1) was picked

$\hookrightarrow dp[i+1][0]$

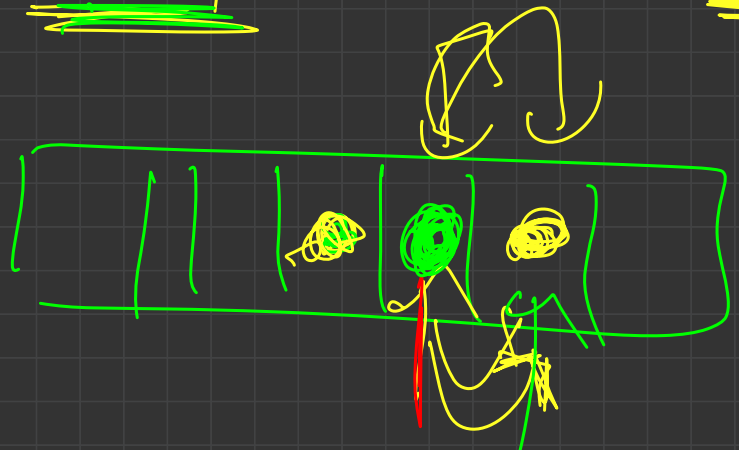
dp[i][0]

pick up

not pick up

dp[i+1][1] + a[i]

dp[i+1][0]



$$dp[i][0] = \max(dp[i+1][1] + a[i],\ dp[i+1][0])$$

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

final → $dp[0][0]$, $\max(dp[0][0], dp[0][1])$

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

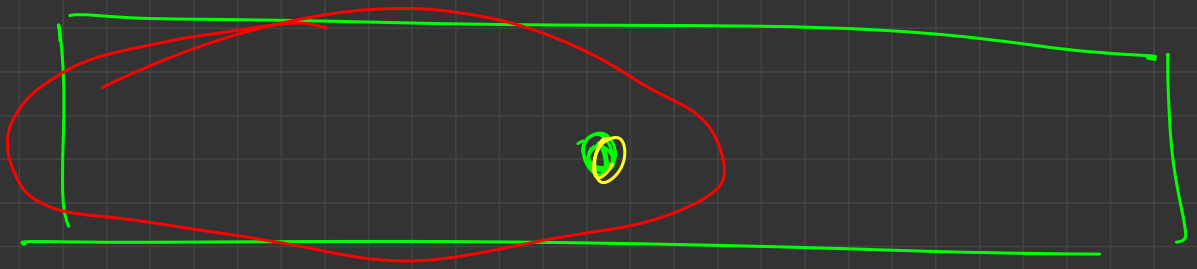
   Transition:

   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])



$dp[n-1][0]$

$dp[n-1][1]$

$dp[i][0] = \max\{dp[i-1][0], dp[i-1][1]\}$

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

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;
}
```

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

dp(i) → { n/i } steps

dp(10) = { n/10 }

dp(1) = { n }

dp(n) = { a/n = 0 } = transition steps

$dp[1] = n$

$dp[2] = n/2$

$dp[3] = n/3$

$\vdots$

$dp[n] = n/n$ steps

$n + \frac{n}{2} + \frac{n}{3} \cdots \frac{n}{n}$

$n\left(1 + \frac{1}{2} + \frac{1}{3} \cdots \frac{1}{n}\right)$

$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} \cdots \frac{1}{n} < \log n$

$n \log n$

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

states $= O(n)$

T.T $= O(1)$

$\Rightarrow O(n)$

$dp(i)(j) = grid(i)(i) + \min \begin{cases} dp(i-1)(j) \\ dp(i)(j-1) \end{cases}$

$O(n \cdot m) \cdot O(1) = O(n \cdot m)$

dp(i)(0) = ——————

O(1)

dp(i)(1) = O(1)

O(n.2) . O(1) = O(n)