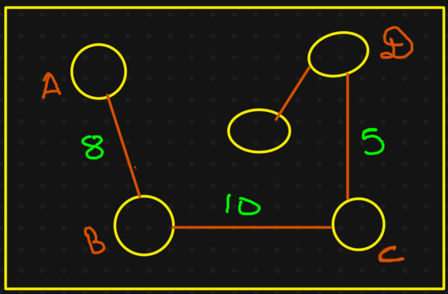
Dynamic Programming

'Those who don't remember the past are condemned to repeat it'

Dynamic Programming is an optimization technique used to solve problem by

→ breaking them into smaller subproblem
→ solving each sub problem once
→ storing their solution.

Eg: Trip Itinerary



D - C - B
↳ DC + CB = 15

A - C
↳ AB + BC = 18

A - D
↳ AB + BC + CD = 23

Pre-requisites:-

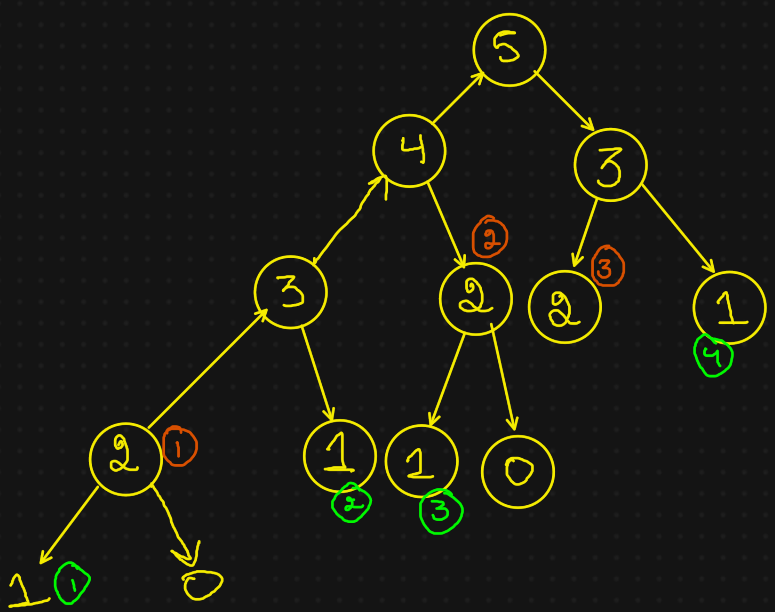
1. Recursion
2. List
3. 2-D List
4. Indexing
5. Recursion

Fibonicci Series

1 1 2 3 5 8 13 . . . .

```
def fib_recursive(n):
    if (n<=1)
        return 1

    return fib_recursive(n-1) + fib_recursive(n-2)
```



We are making a lots of duplicate calls

Time complexity = 2^n

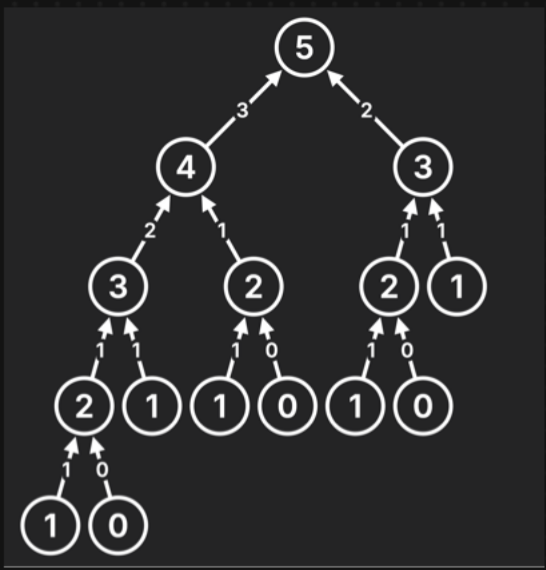
We are just making $(n+1)$ unique calls.

0 . . . . . n

We should somehow store the results for numbers we already have calculated fibonicci / answer for.

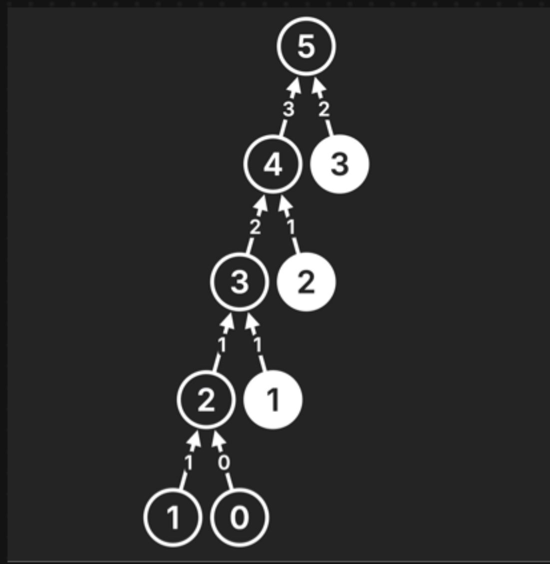
List which stores the answer already calculated.

Time Complexity : Memoization code



Recursion

$O(2^n)$



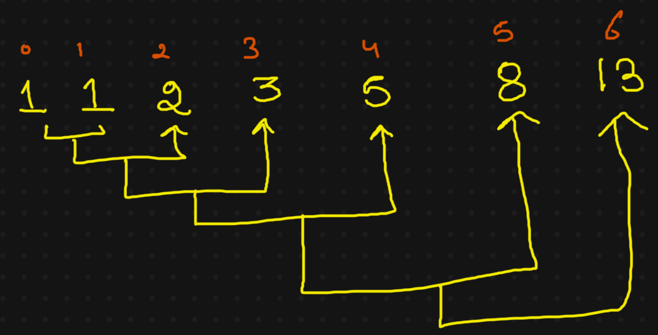
Memoization

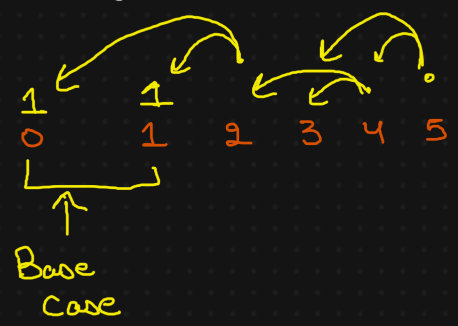
$O(n)$

memo

each number, fibonacci
just calculated 1 time.

Fibonicci Number : A Dynamic Programming Approach





Bottom up

Tabulation

We want to calculate the answer from bottom.

Top - down

Memoization

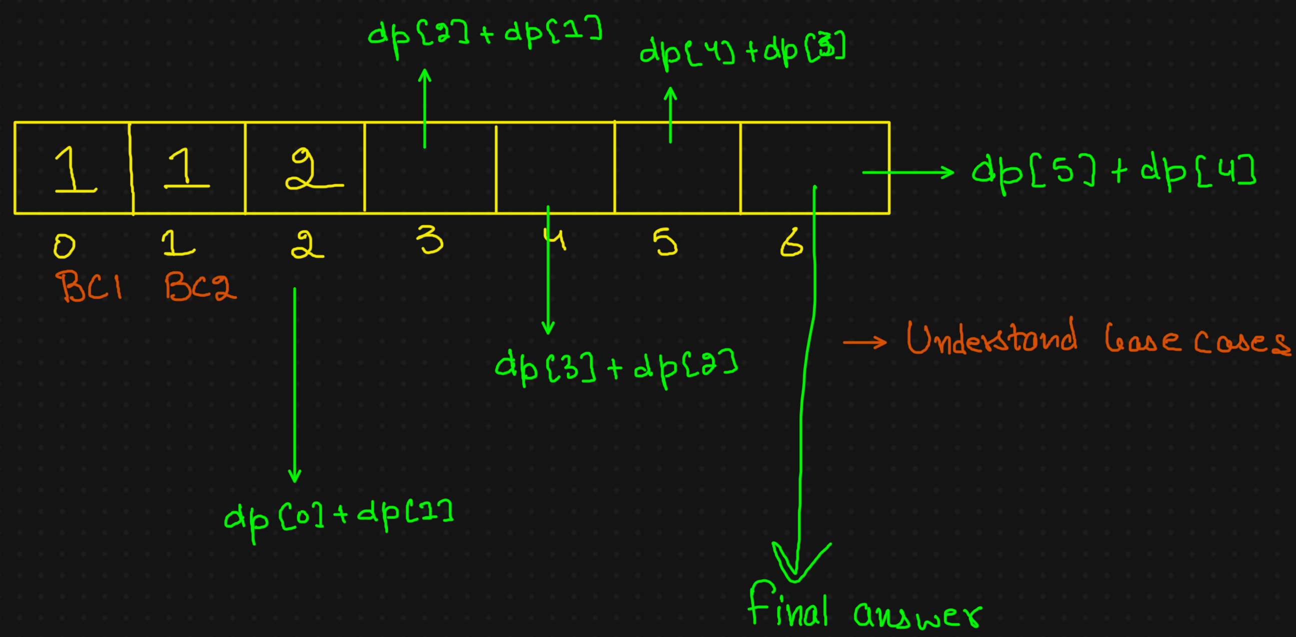
here as we use recursion, extra call stack memory is required

→ Recursion

→ Recursion + Memoization → Storing sub problem solution

→ bottom up / tabulation → no recursion, directly storing the solution.

Tabulation Approach



Time Complexity : Tabulation Approach

dp → used to store answer to sub-problems in a bottom-up manner

→ recursive

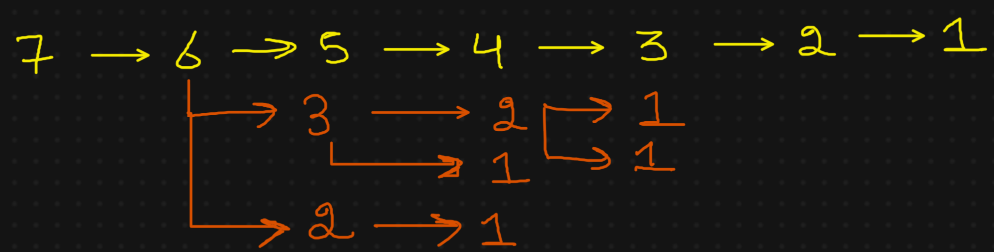
→ memoization

→ tabulation / Pure-DP

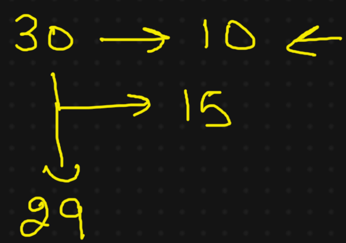
Minimum steps to reduce n to 1

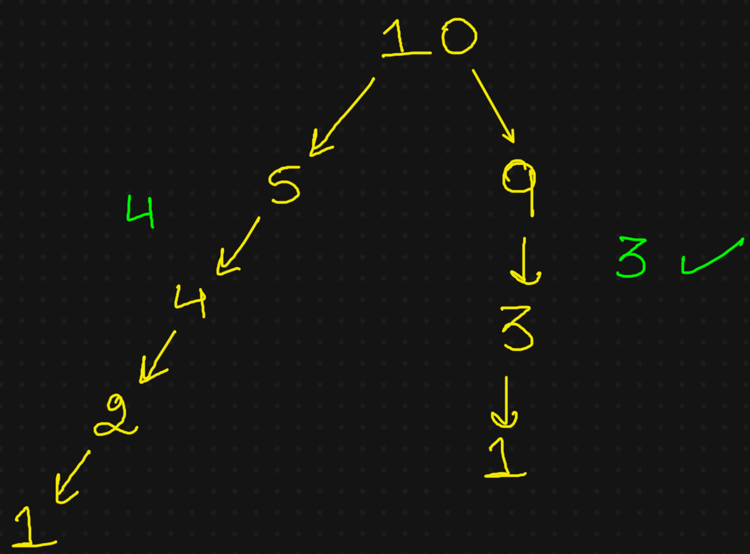
→ Given a number n, we have to tell minimum number of steps required to reduce the number to 1. Below operations are allowed

1. Subtract 1
2. divide by 2
3. divide by 3



$9 \rightarrow 3 \rightarrow 1$





always reducing by greatest magnitude is not the solution.

1. recursion
2. Memoization
3. Tabulation

Minimum steps to 1 → Solution

1] reduce by 1
2] divide by 2
3] divide by 3

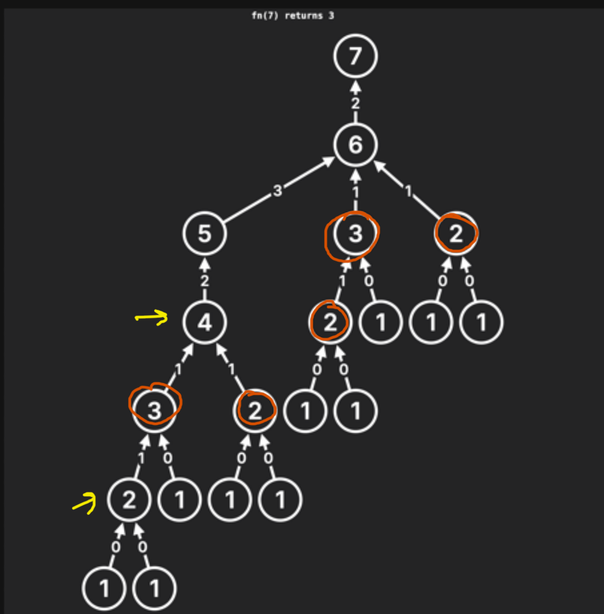
Recursion

1. Base case   if n == 1
2. Recursive call

6 → 5, 3, 2

n → n-1, n/2, n/3
s1, s2, s3

3. Our work   min(s1, s2, s3) + 1



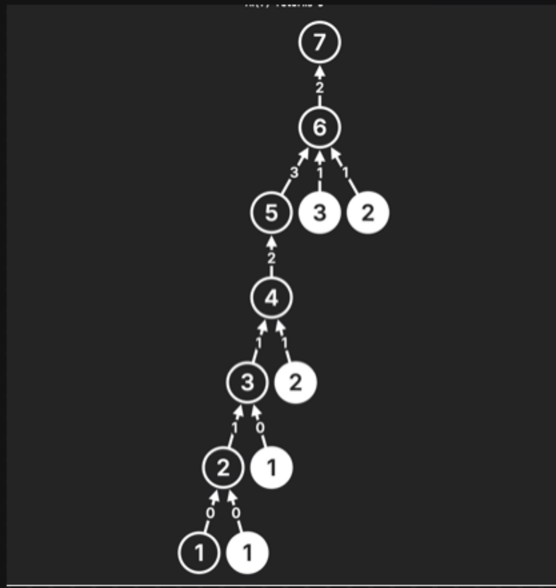
We see that in recursion approach we have many subproblems which we need to solve again and again

→ we can use DP here to solve & save the solution of sub-problems

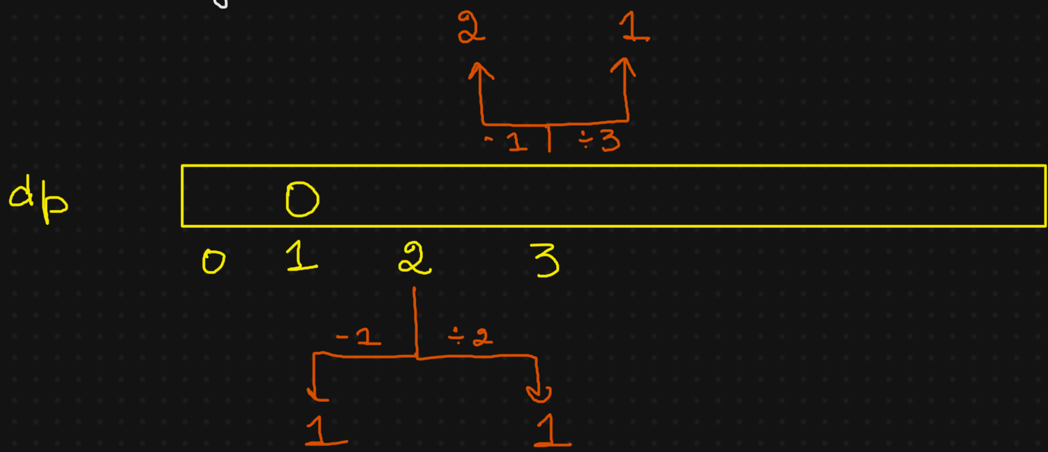
memo

	0	1	1	2	3	2	3
0	1	2	3	4	5	6	7

Recursion tree
with memoization.



Solution using tabulation



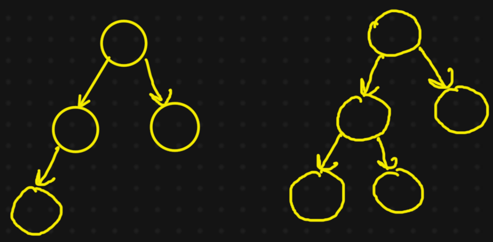
Number of possible balanced binary tree with given height

h = 1

h = 2

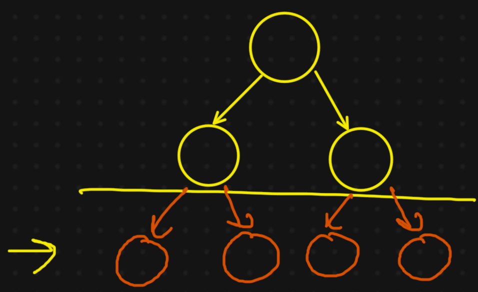
≈ 3

h = 3



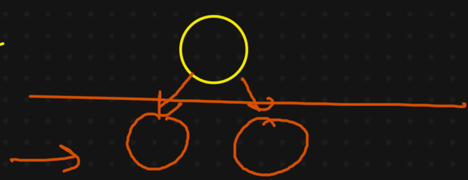
. . . . . .

≈ 15



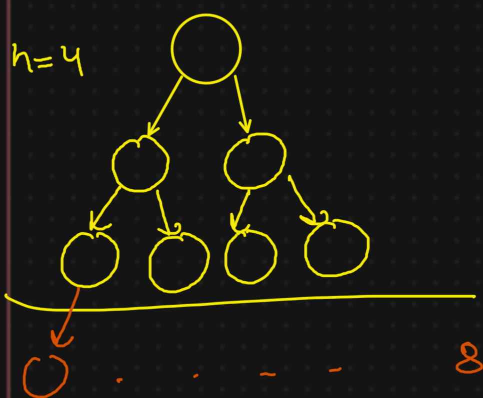
$2^4 - 1 = 15$

h = 2



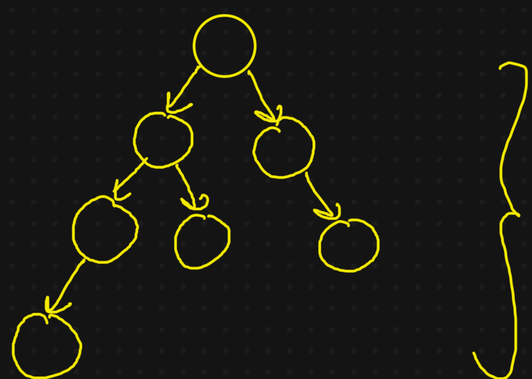
$2^2 - 1 = 4 - 1 = 3$

h = 4



$2^8 - 1 = 256 - 1 = 255$

We are assuming that tree with (h-1) height will be complete.



h = 4

h and balanced.

→ (h-1) (h-1)

(h-1) (h-2) ⇒ as still balanced because diff. ≤ 1

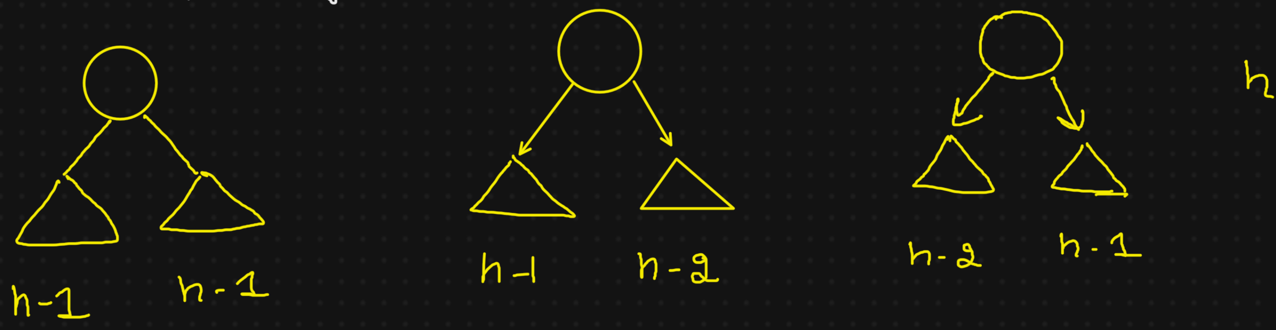
(h-2) (h-1)

h_minus_1

h_minus_2

ans = (h1 * h1) + (2 * h1 * h2)

Balanced Binary Tree of given height h — Solution

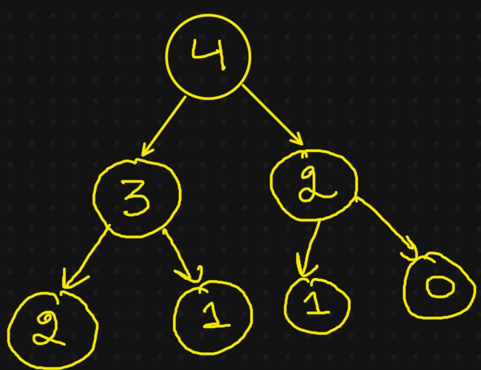


6 * 6 → 36

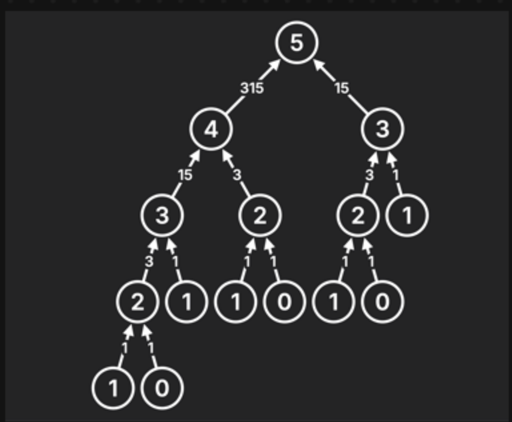
1. Base Case ✓
2. Recursive call ✓
3. Our work

$$ans = h1 * h1 + 2(h1 * h2)$$

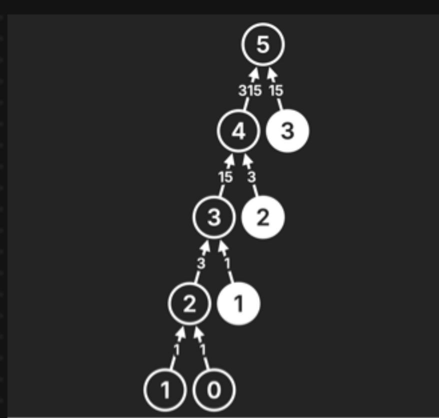
✱ In some other languages, to prevent overflow the answer is bounded by taking mod wrt a number.



Complexity → 2^n



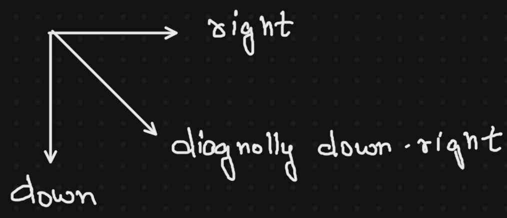
Recursion



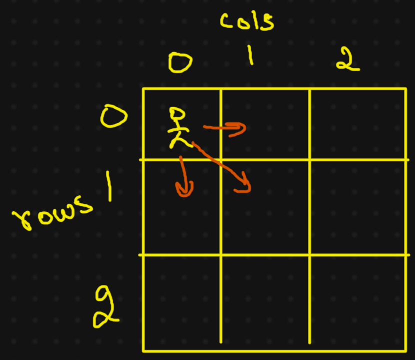
Memoization

Min Cost path

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 8 & 2 \\ 1 & 5 & 3 \end{bmatrix}$$

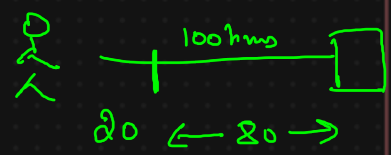


1. recursive
2. Memoization
3. Tabulation



0, 0

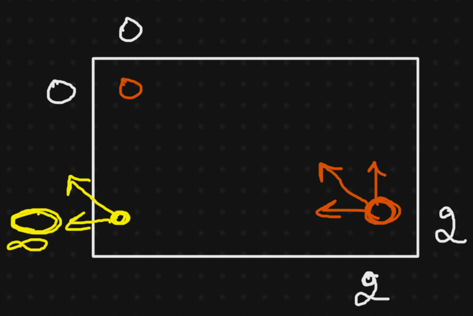
2, 2



Directions in which we can move –

1. right (0,0) ⟶ (0,1)
2. down (0,0) ⟶ (1,0)
3. Diagnolly (0,0) ⟶ (1,1)

Min cost path - Recursive Solution



m, n

(0,0)

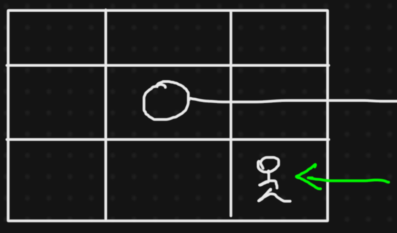
1. left 2. Top 3. diag

```
cost_matrix = [
    [1, 2, 3],
    [4, 8, 2],
    [1, 5, 3]
]
```

1 2 3

4 8 2

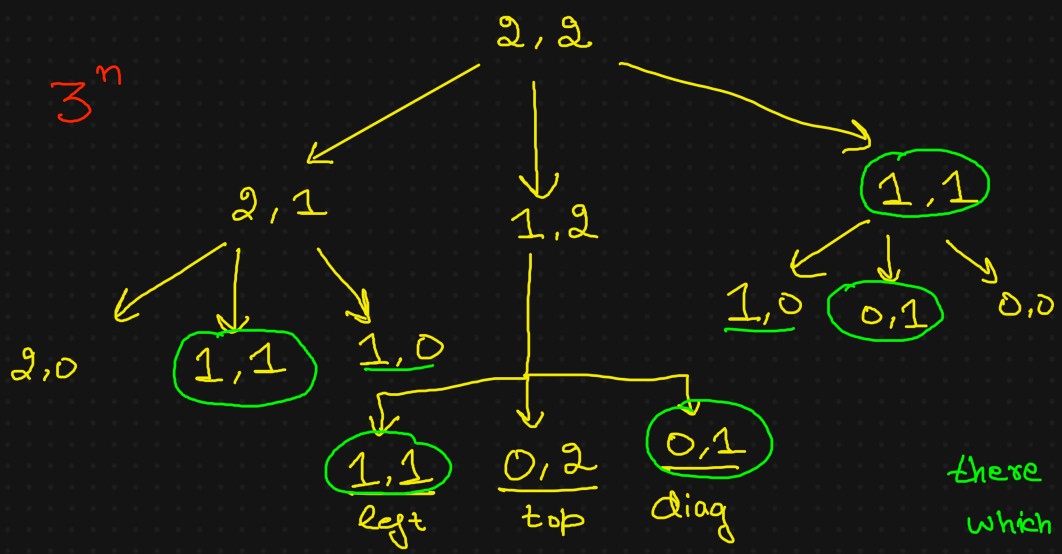
1 ← 5 ← 3

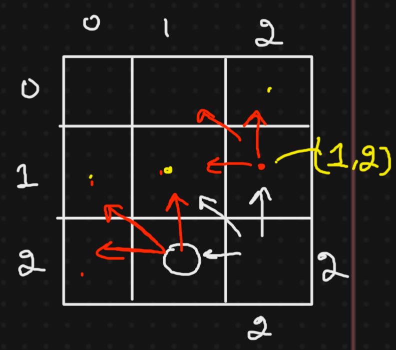


minimum cost to reach this point from (m, n)

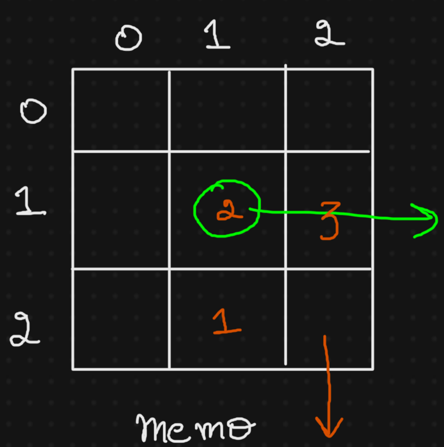
start, end

3^n





there are subproblems which we are solving again and again



1. Space or list size
2. what each box signifies

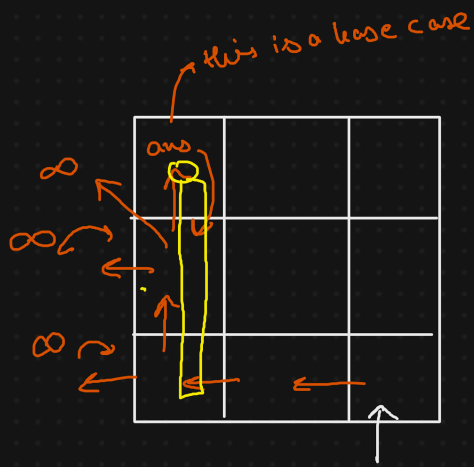
give the ans to our subproblem i.e. min^m cost path from 0,0 to this box (or vice versa)

$\min(c_1, c_2, c_3) + cost_{m,n}$

m x n will be size of our memo list

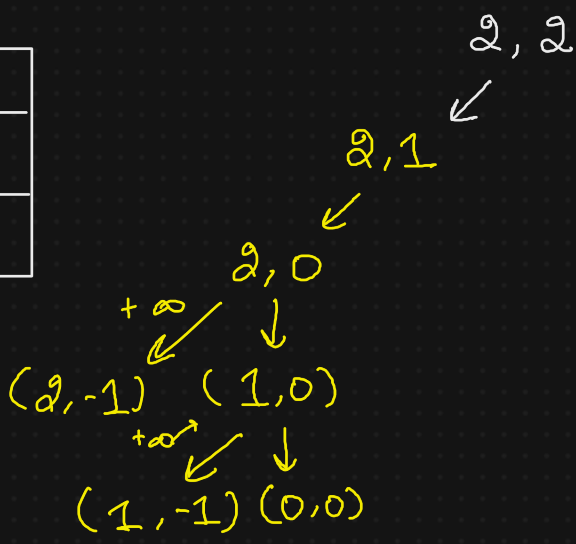
Steps for solving 2-D DP using memoization.

1. Solve problem using recursion
2. Create recursion tree and see if sub-problems are overlapping
3. Find out unique subproblems.
4. Create the memo space
5. Define what each box or individual memory in space signify.
6. Write down code and save the solution.



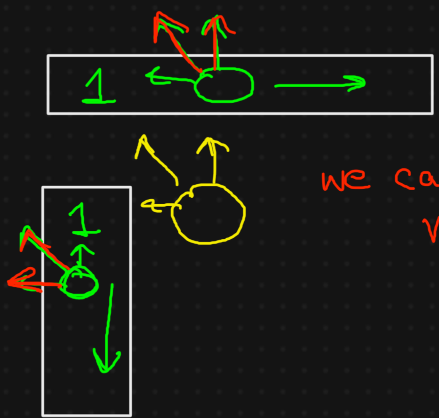
Minimum cost path → tabulation approach

	0	1	2
0	1	3	6
1	5	9	5
2	6	10	8



1	2	3
4	8	2
1	5	3

	0	1	2
0	1	3	6
1	5	9	5
2	6	10	8



we can fill the boundary values easily.