Dynamic Programming

↳ always correct answer
↳ More time to generate the optimized results
↳ Optimizations on top of Recursion (Really good)
  ∘ Overlapping subproblems

fib(7)

n
↳ 0 to n-1

fibonnaci series

0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	13

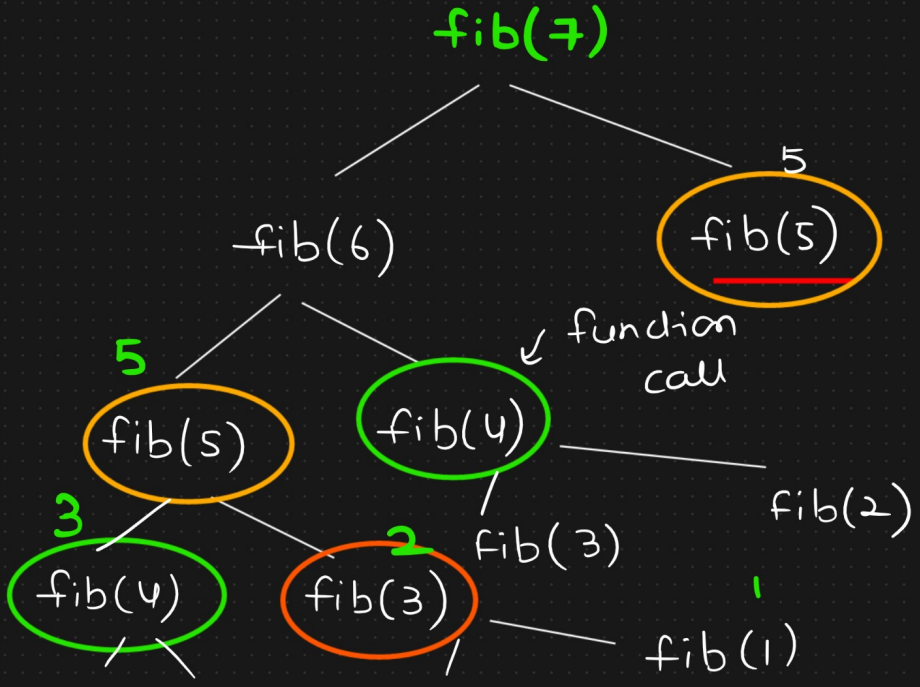
Recursion

n+1
↳ 0 to n

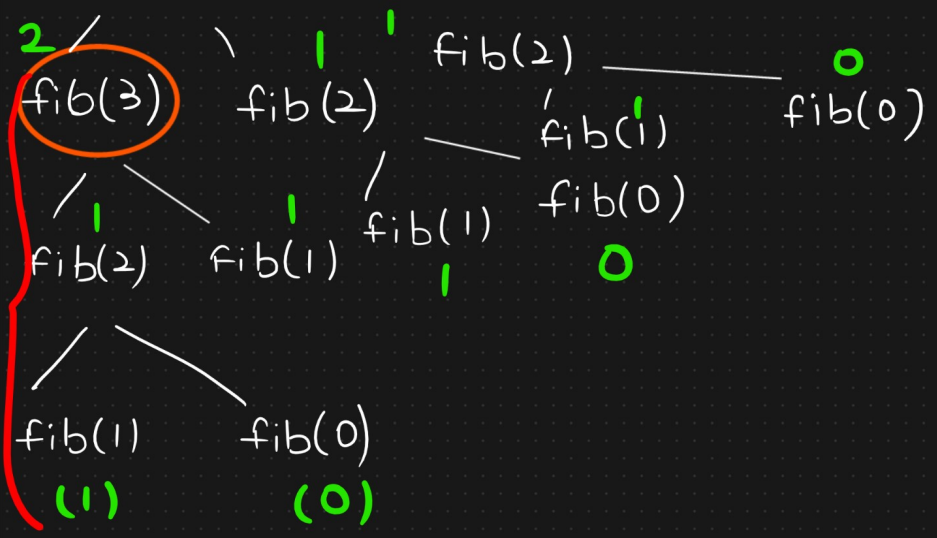
fib(n):
  if n==0 or n==1:
    return n
  else:
    return fib(n-1) + fib(n-2)

↑ Recursion

Recursive Tree



⇓
Recomputation of same subproblem again & again
↳ Dynamic Programming



Dynamic Programming

↳ Store the results of every function call

⇓

same function call come

⇓

return the answer instead of doing recomputation

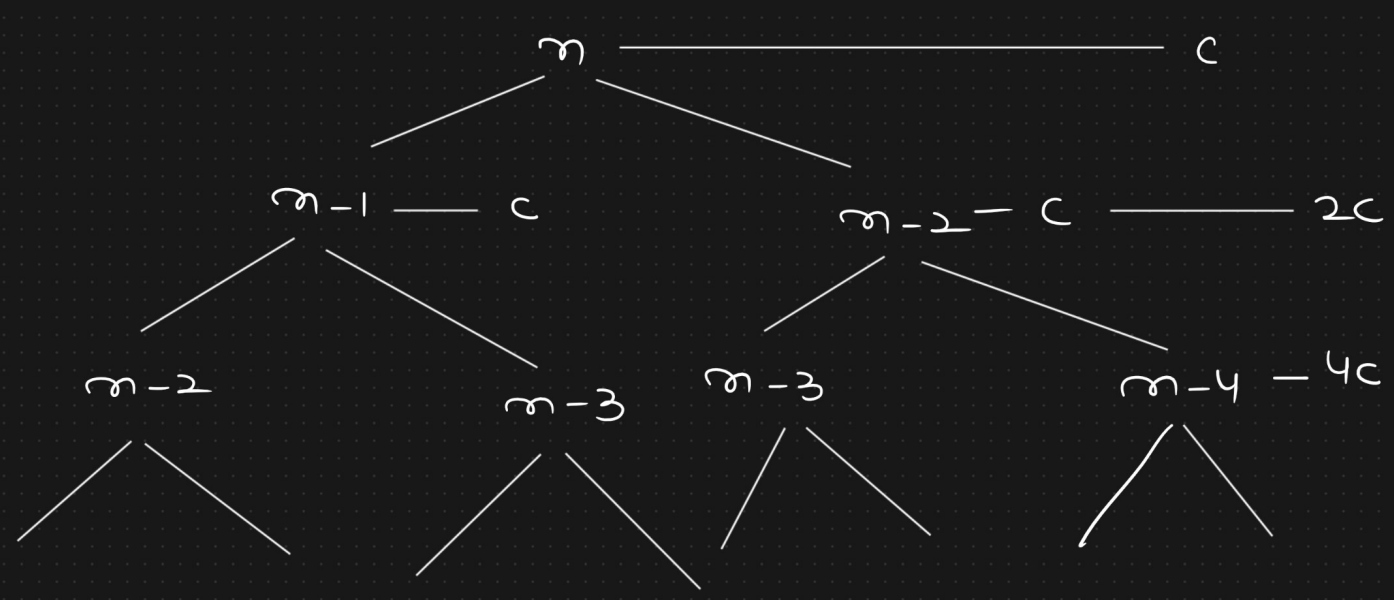
Optimization

Time complexity

Recursion

$$T(n) = \begin{cases} c & n = 0 \text{ or } n = 1 \\ T(n-1) + T(n-2) + c & n > 1 \end{cases}$$

Using Recursive Tree Approach



Left side

$n - k = 1$

$n - 1 = k$

$k = n$

Right side

$n - 2k = 1$

$k = n/2$

$$c(2^0 + 2^1 + 2^2 + \dots + 2^k)$$

$$c(2^0 + 2^1 + 2^2 + \dots + 2^n)$$

$r = 2$

$a = 2^0 = 1$

$r > 2$

Sum of GP series $= \frac{a(r^n - 1)}{r - 1}$

$= \Theta(2^n)$

CBT —

n levels

$2^n - 1$ nodes

2^n function calls

$\hookrightarrow \Theta(2^n)$

n is less

Exponential Time complexity

Dynamic Programming ↓

Polynomial Time complexity

$\Theta(n)$

n

- fib(7)
  - fib(6)
    - fib(5)
      - fib(4)
        - fib(3)
          - fib(2)
            - fib(1)

Dynamic Programming

↳ Memoization
(Top Down Approach)
↳ Recursion only
↳ Storing the result of every recursive call

↳ Tabulation
(Bottom up Approach)
↳ Not using Recursion
(Table & using some logic we are Storing the results)

Longest common subsequence

0 1 2 3 4

String 1 : Aman \0

String 2 : Ankan \0

0 1 2 3 4 5

An

String 1 : a b c d e f g h i j

String 2 : c d g i

⟹ 1) cdgi (4) → Longest common subsequence

2) dgi

3) gi

String 2' : c e d g i (4)

c e g i → Longest common subsequence

i

↓

0 1 2

S1 = b d \0

S2 = a b c d \0

↑ 0 1 2 3 4

J

Expected output = bd

↳ 2

Recursive solution

```
lcs(i, j):
    if s1(i) == '\0' or s2(j) == '\0':
        return 0
    elif s1(i) == s2(j):
        return 1 + lcs(i+1, j+1)
                 ↳ Recursion
    else:
        return max(lcs(i+1, j),
                   lcs(i, j+1))
```

Dynamic Programming

→ 1) Memoization
2) Tabulation

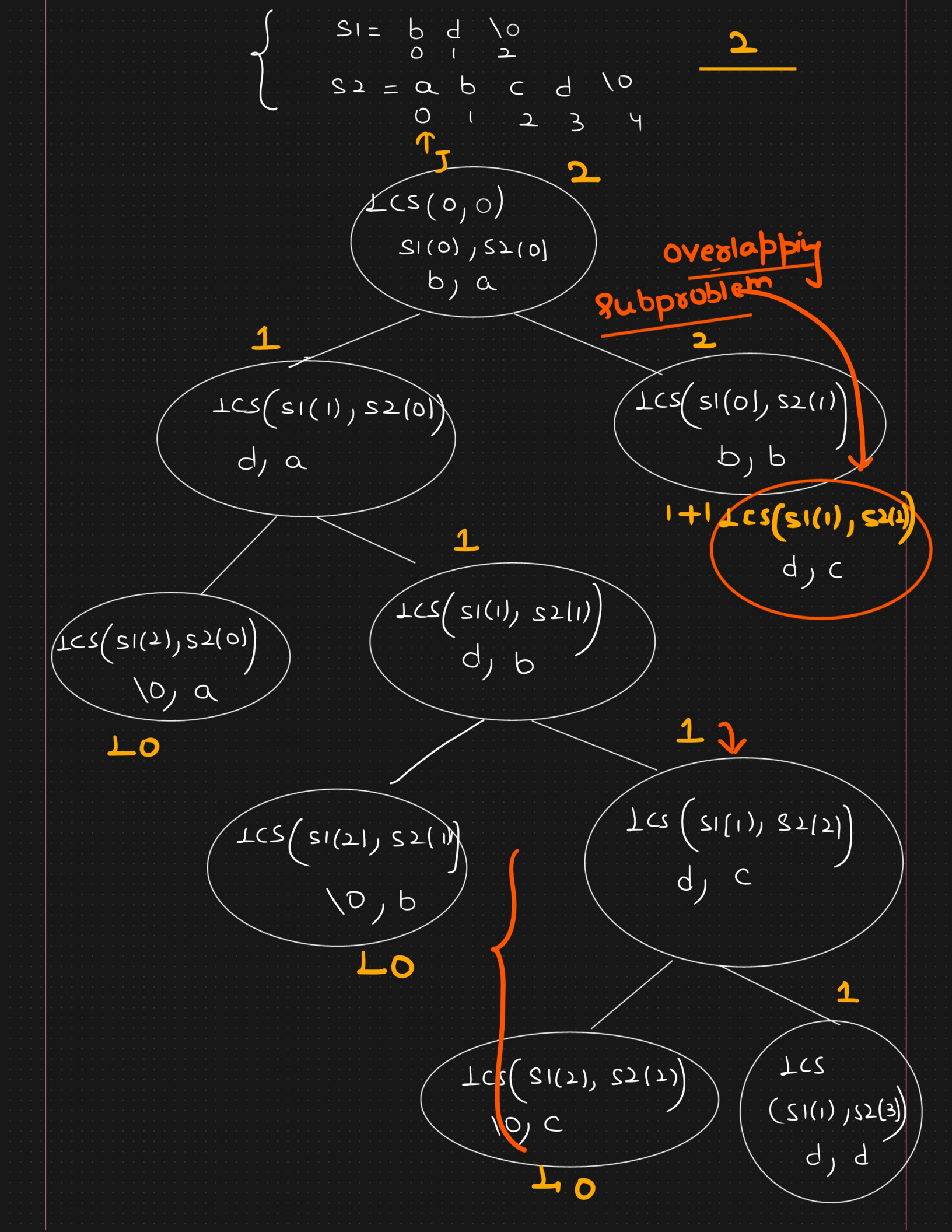
Task → Try to do the implementation of lcs using memoization

Matrix → m * n

↓ ↓

s1 s2

↓ i


s1 = b d \0
0 1 2
s2 = a b c d \0
0 1 2 3 4
2
LCS(0,0)
s1(0), s2(0)
b, a
2
overlapping subproblem
1
LCS(s1(1), s2(0))
d, a
LCS(s1(0), s2(1))
b, b
2
1+1 LCS(s1(1), s2(1))
d, c
1
LCS(s1(2), s2(0))
\0, a
LCS(s1(1), s2(1))
d, b
1
0
1
LCS(s1(2), s2(1))
\0, b
LCS(s1(1), s2(2))
d, c
0
1
LCS(s1(2), s2(2))
\0, c
LCS
(s1(1), s2(3))
d, d
0

1+0

Lcs(s1(2),

s2(4))