why

Dynamic Programming → algo paradigm

↓

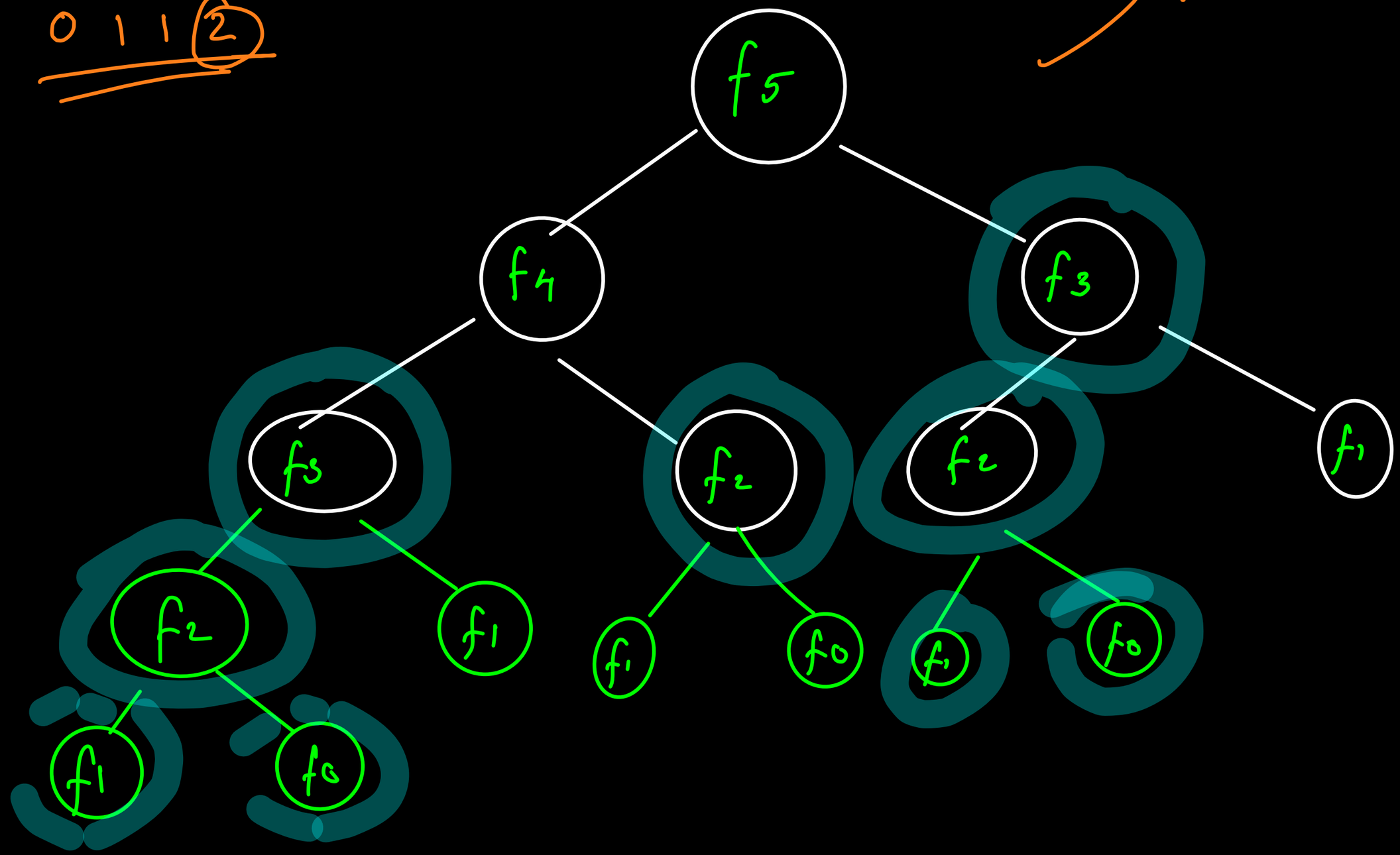
optimisation technique

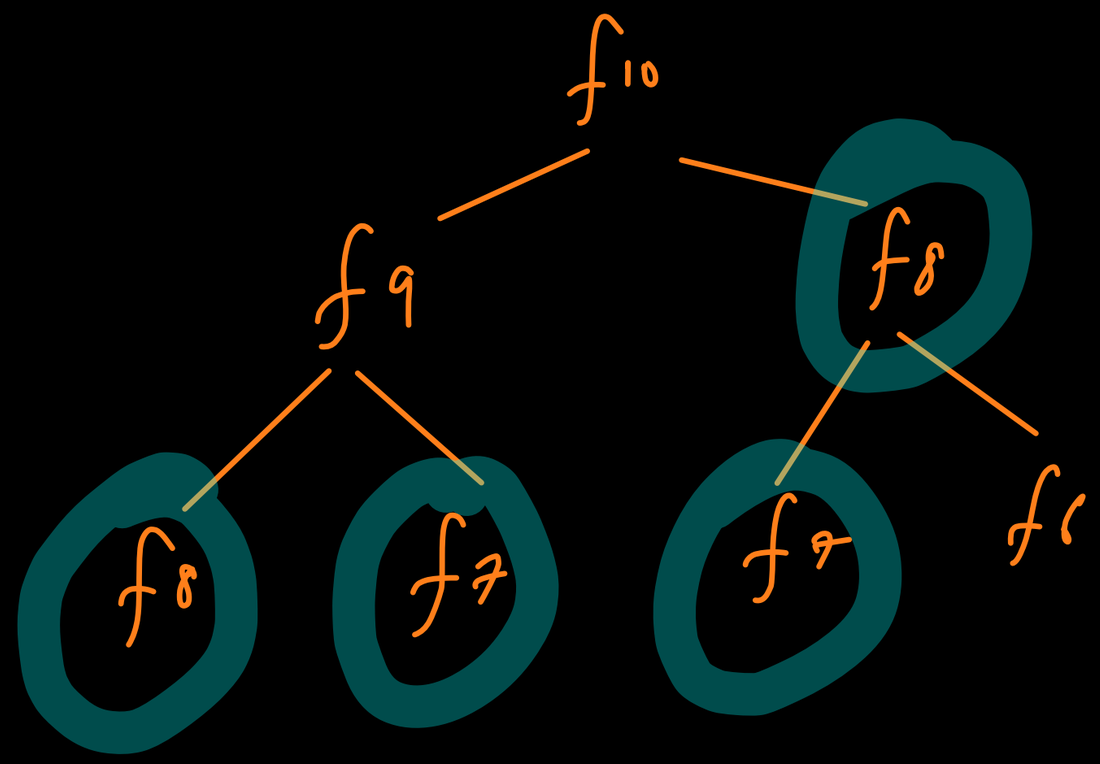
↓

generally our brute force is optimized

Fib → $f(n) = f(n-1) + f(n-2)$ ✓ Recursion tree for fib

0 1 1 2





Repeating Subproblems / overlapping subproblems

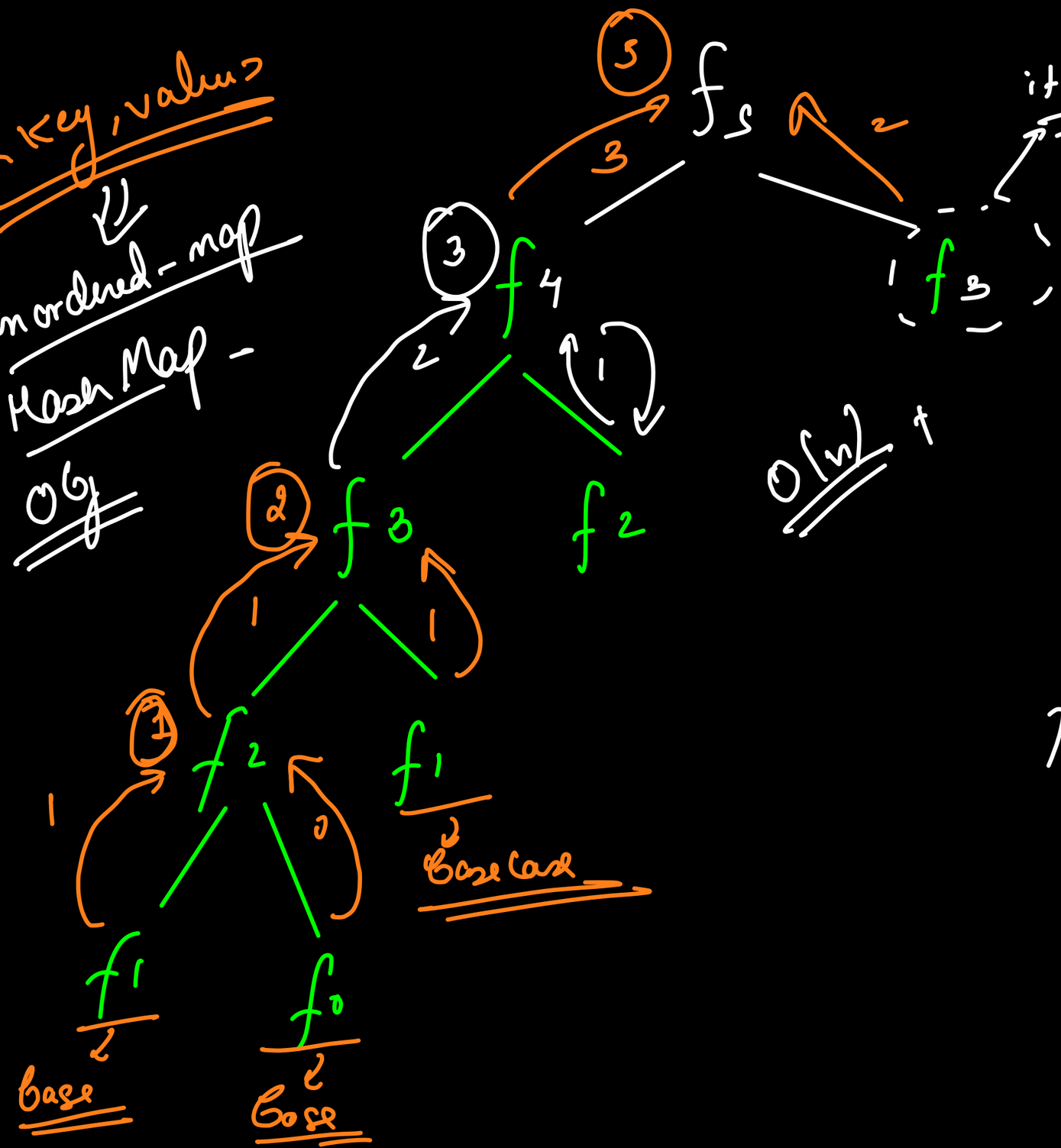
<key, value>

⇓

unordered-map

HashMap -

O(1)



O(n) +

it is precomputed

array

Store

↳ $f_2 \to 1$

$f_3 \to 2$

$f_4 \to 3$

$f_5 \to 5$

$f_5 = f_4 + f_3$

$n \times 2 \longrightarrow O(n)$

minimizing/
maximizing

counting

How to identify a problem can be solved using dp??

↳ ① Repeating subproblems

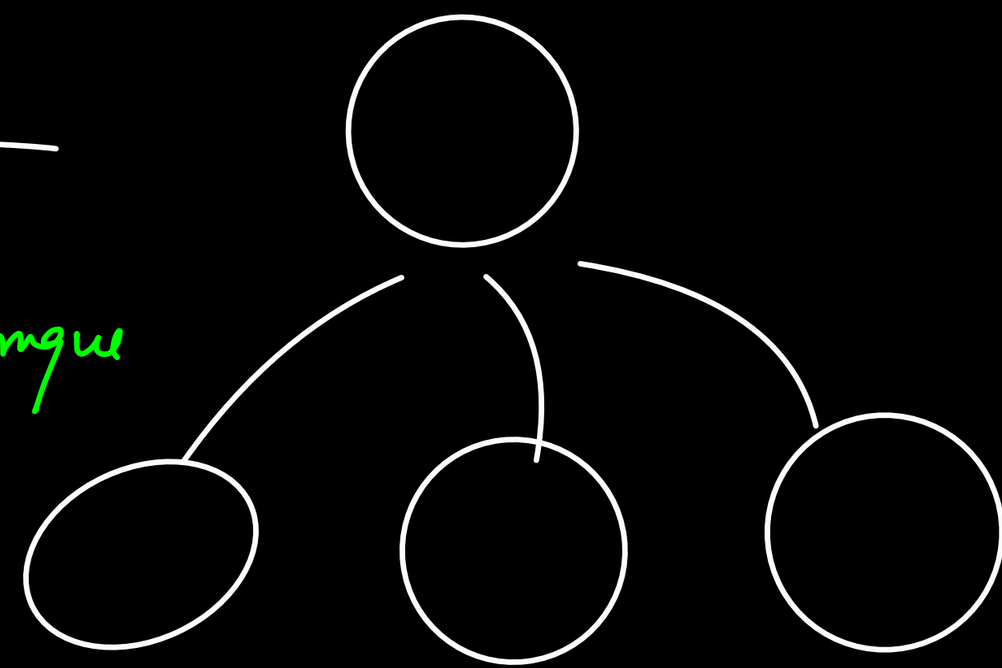
② Optimal substructure → the optimal solution to a bigger problem can be prepared using optimal sol^n of smaller subproblem

State of a subproblem

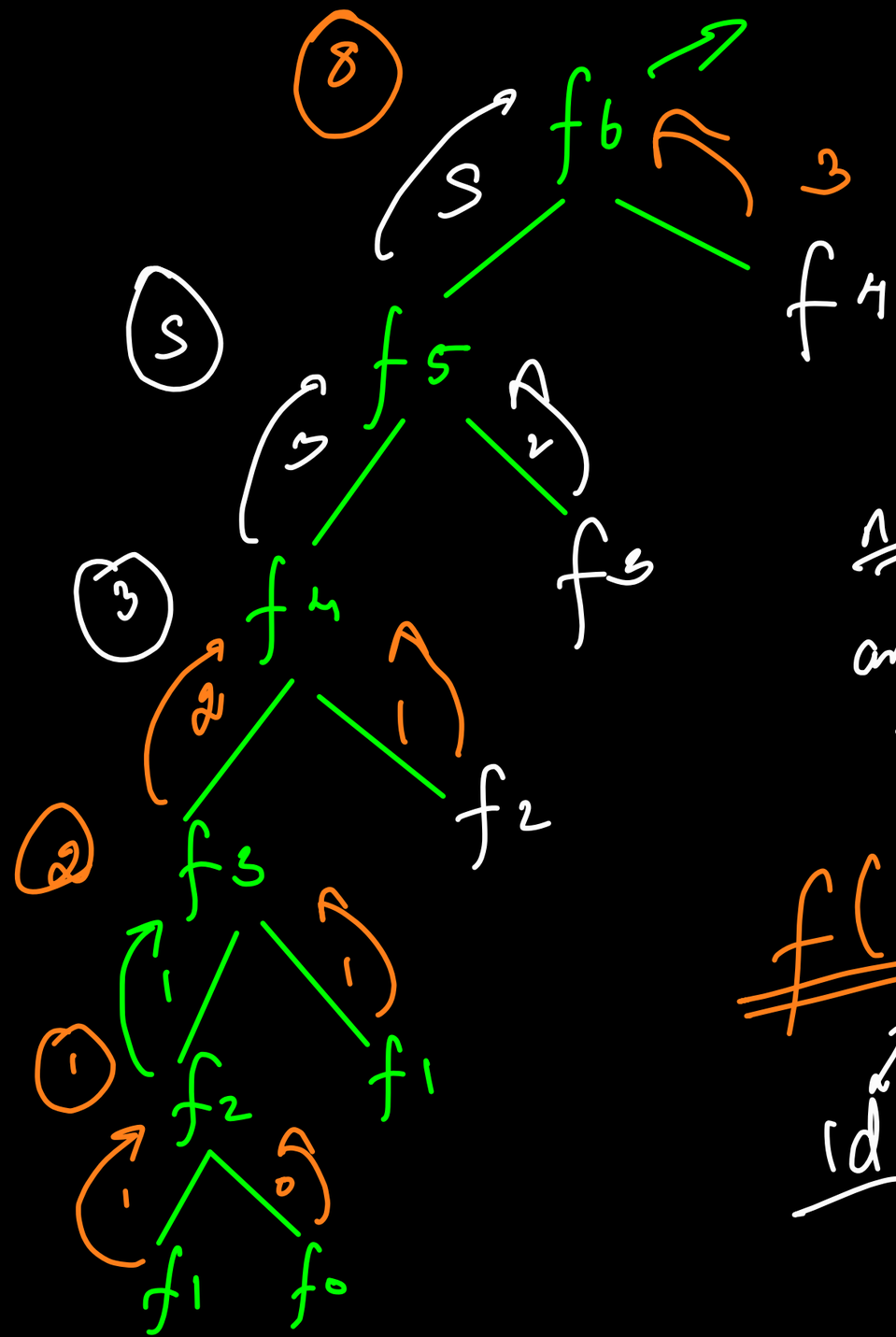
?

Has helps to identify a unique subproblem.

$f(i, j, k)$



f_s



whether the subproblem is already computed or not

input

n

ans = dp[n]

```
f(n) {
  if(n==0 or n==1)
      return n;
  if (dp[n] != -1) return dp[n]
  ans = f(n-1) + f(n-2)
  dp[n] = ans
  return ans;
}
```

f(n)

id dp

storage → we will try to store ans to all very subproblem

↳ n+1 unique subproblems

n+1 size storage

dp

0	1	2	3	4	5	6
-1	-1	~~-1~~ 1	2	3	5	8

↳ filling this array with a val that can never be the ans.

-1 → the ans to subproblem is not computed yet

The prev approach is a top down DP. (Recursion)

Bottom up → iterative

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

we can optimize space here

dp

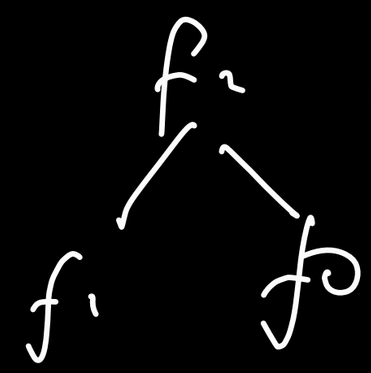
dp[2]

$dp_2 = dp_1 + dp_0$

$dp_3 = dp_2 + dp_1$

$dp_5 = dp_4 + dp_3$

$dp_6 = dp_5 + dp_4$



dp[n] ← nth fib

dp[0] = 0
dp[1] = 1 } Base cases

```
for (i = 2; i<=n; i++)
        dp[i] = dp[i-1] + dp[i-2]

return dp[n];
```

3 variables

$a \leftarrow i-2$

$b \leftarrow i-1$

$c \leftarrow i$

$c = a + b$

$i = i-1 + i-2$

```
count = 1
while (count ≤ n)
    c = a + b
    a = b
    b = c
    count++
}
```

Space optimization

$O(n)$

$O(1)$