CodeIT

Lesson 6 Objectives:

To gain an understanding of:

1. What **dynamic programming** (DP) is and how it works
2. How to **identify** DP problems
3. How to use of **memoization** and **tabulation** to solve DP problems
4. How to analyse **time complexities** of recursive and DP algorithms

Dynamic Programming

Dynamic Programming

- **Dynamic Programming (DP)** is an optimisation technique where we take the answers of sub problems, and use that to help solve the larger problem.

- This may sound reminiscent of **RECURSION** and this is because DP is essentially a **optimisation over plain recursion**

Dynamic Programming

- Plainly speaking, DP is useful in allowing us to make algorithms, **where the same computations are done over and over again**, more efficient through the use of **caching**

Dynamic Programming

- Plainly speaking, DP is useful in allowing us to make algorithms, **where the same computations are done over and over again**, more efficient through the use of **caching**

- This can be done through two methods:
  1. **Memoization:** Top down approach
  2. **Tabulation:** Bottom up approach

Introduction to DP & Memoization

Problem #1: Fibonacci

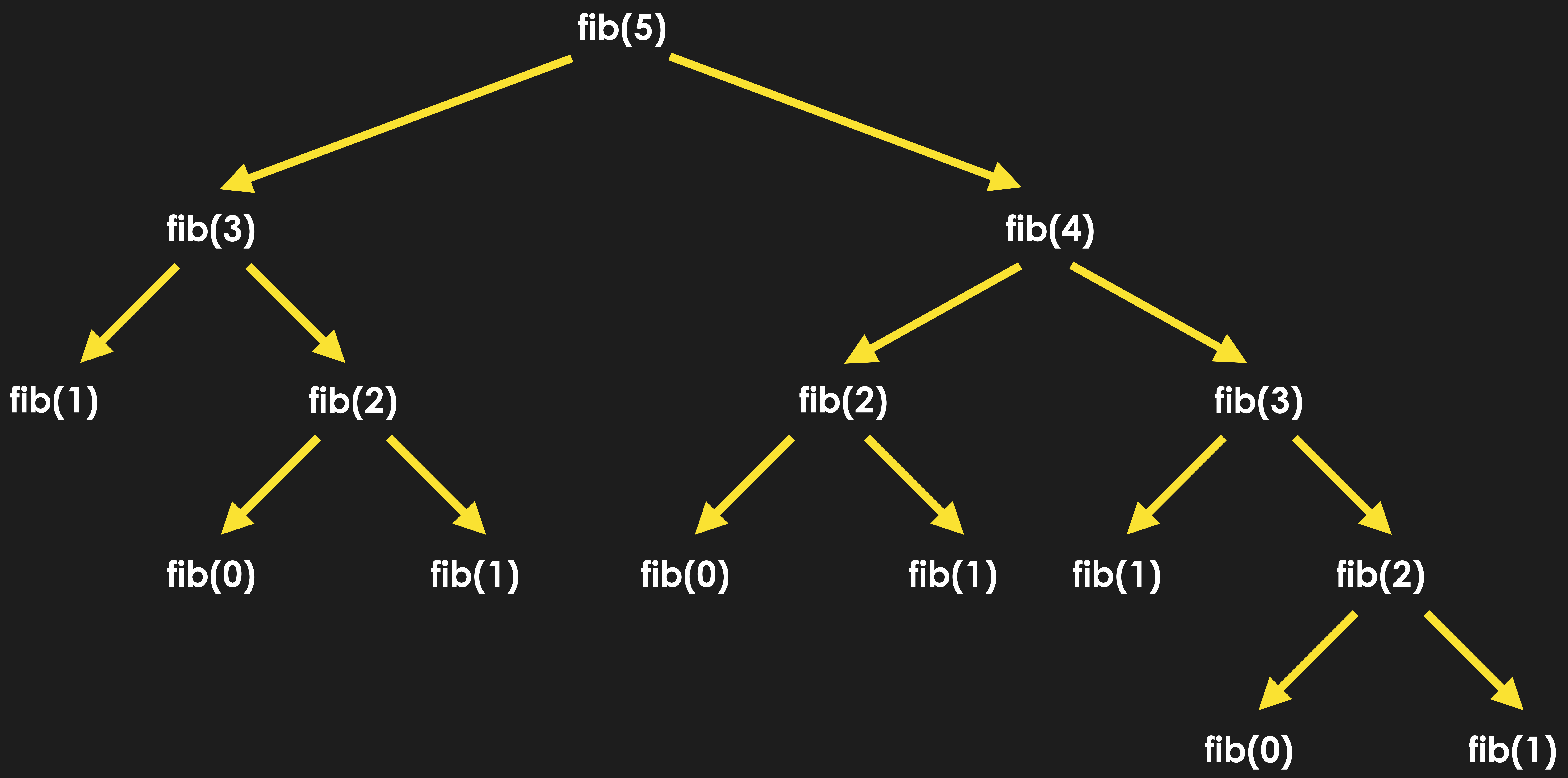
Fib: Calculate the **fibonacci number** given an input **N**

n	0	1	2	3	4	5	6
fib(n)	1	1	2	3	5	8	13

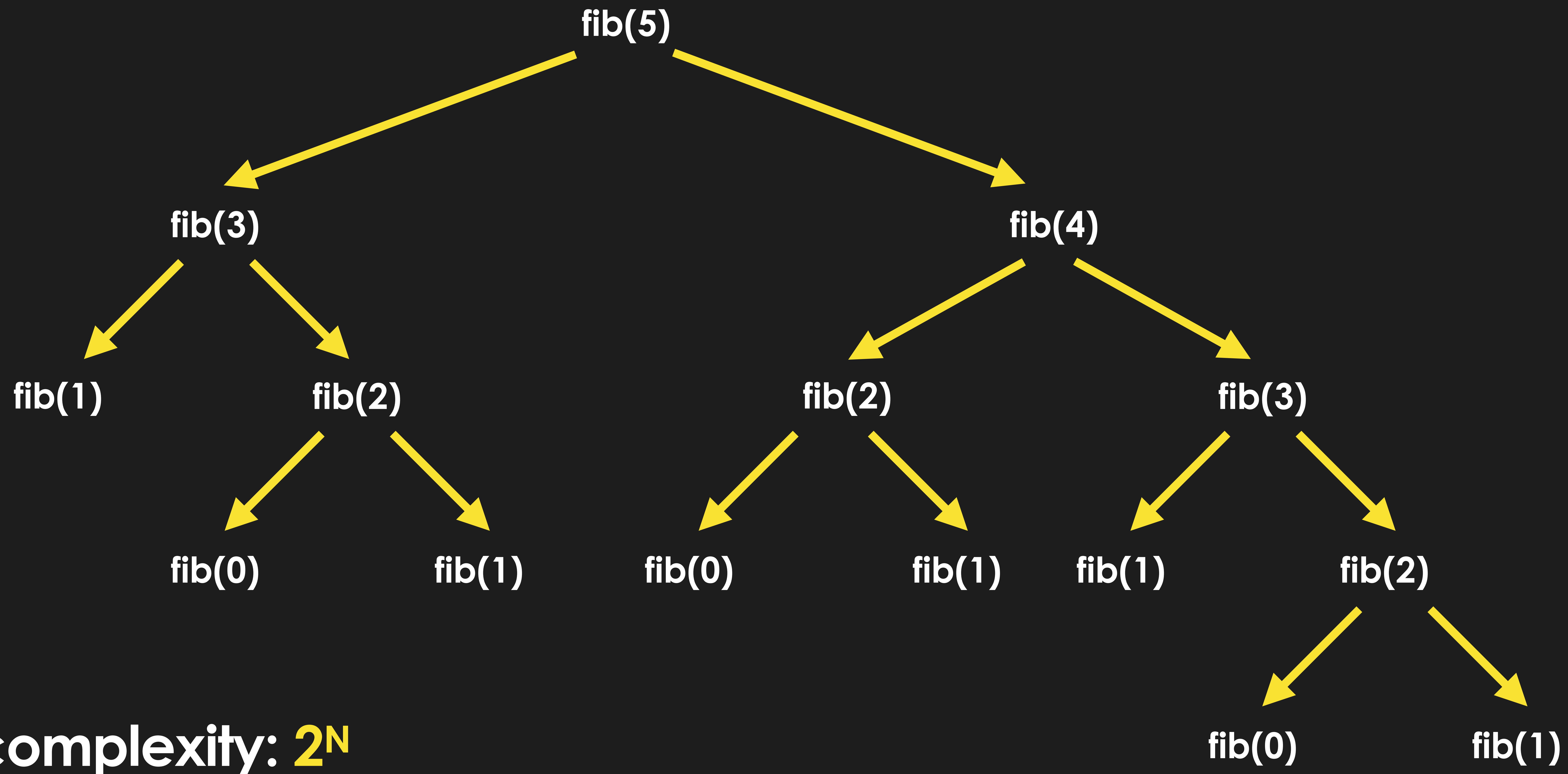
fib

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

fib


fib(5)
fib(3)
fib(4)
fib(1)
fib(2)
fib(2)
fib(3)
fib(0)
fib(1)
fib(0)
fib(1)
fib(1)
fib(2)
fib(0)
fib(1)

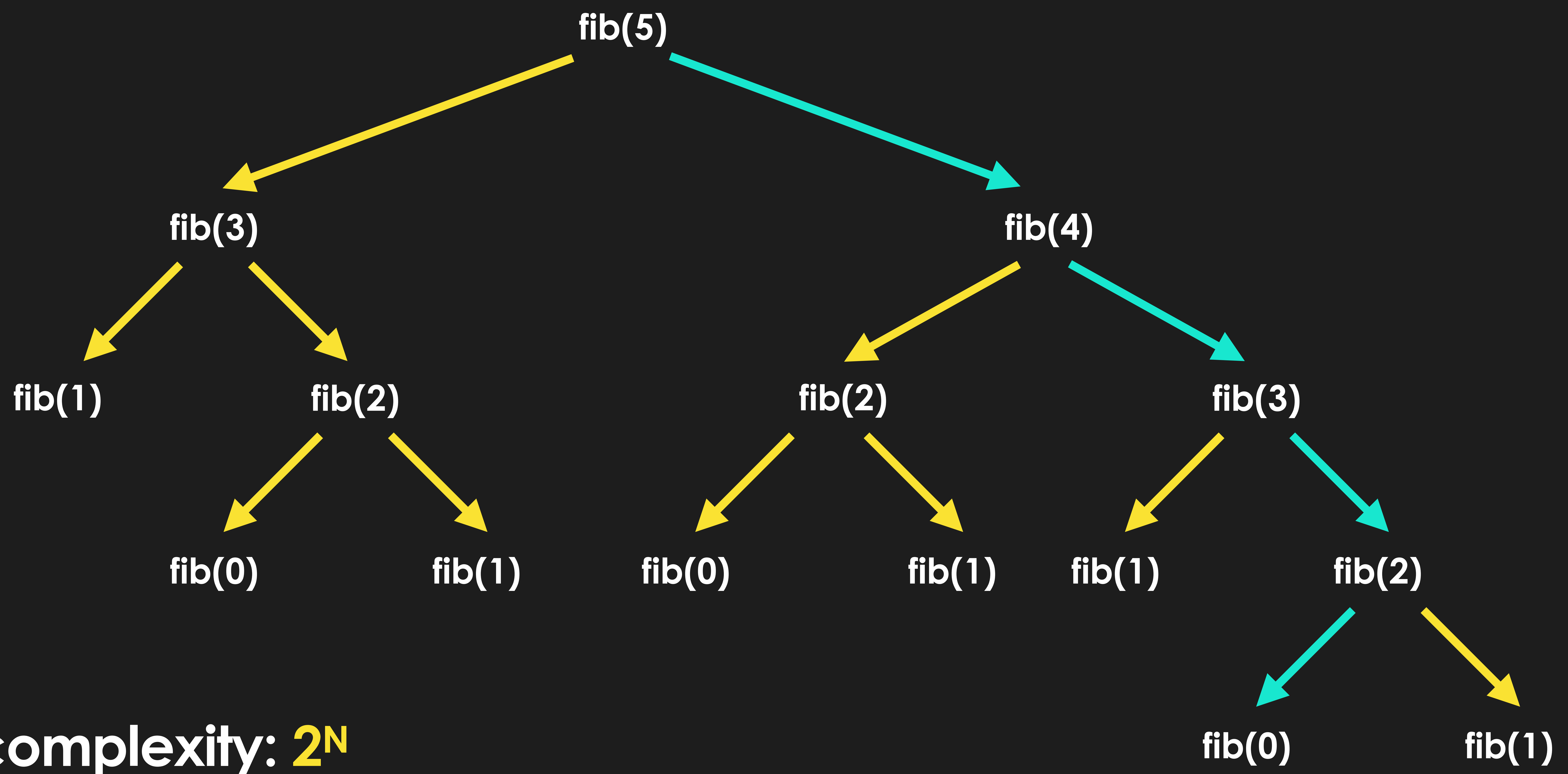
fib



time complexity: 2^N

CodeIT

fib

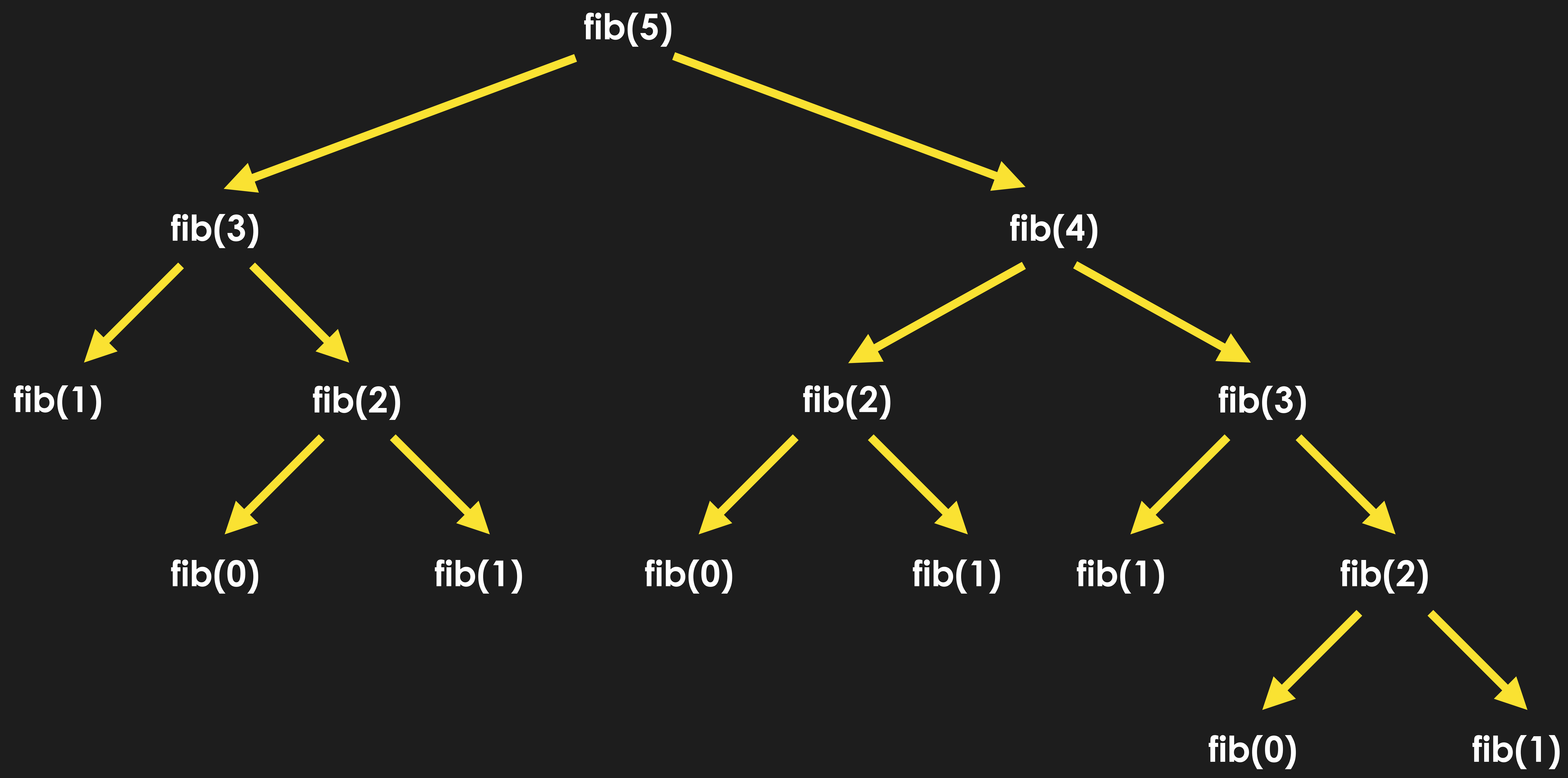


time complexity: 2^N

space complexity: N

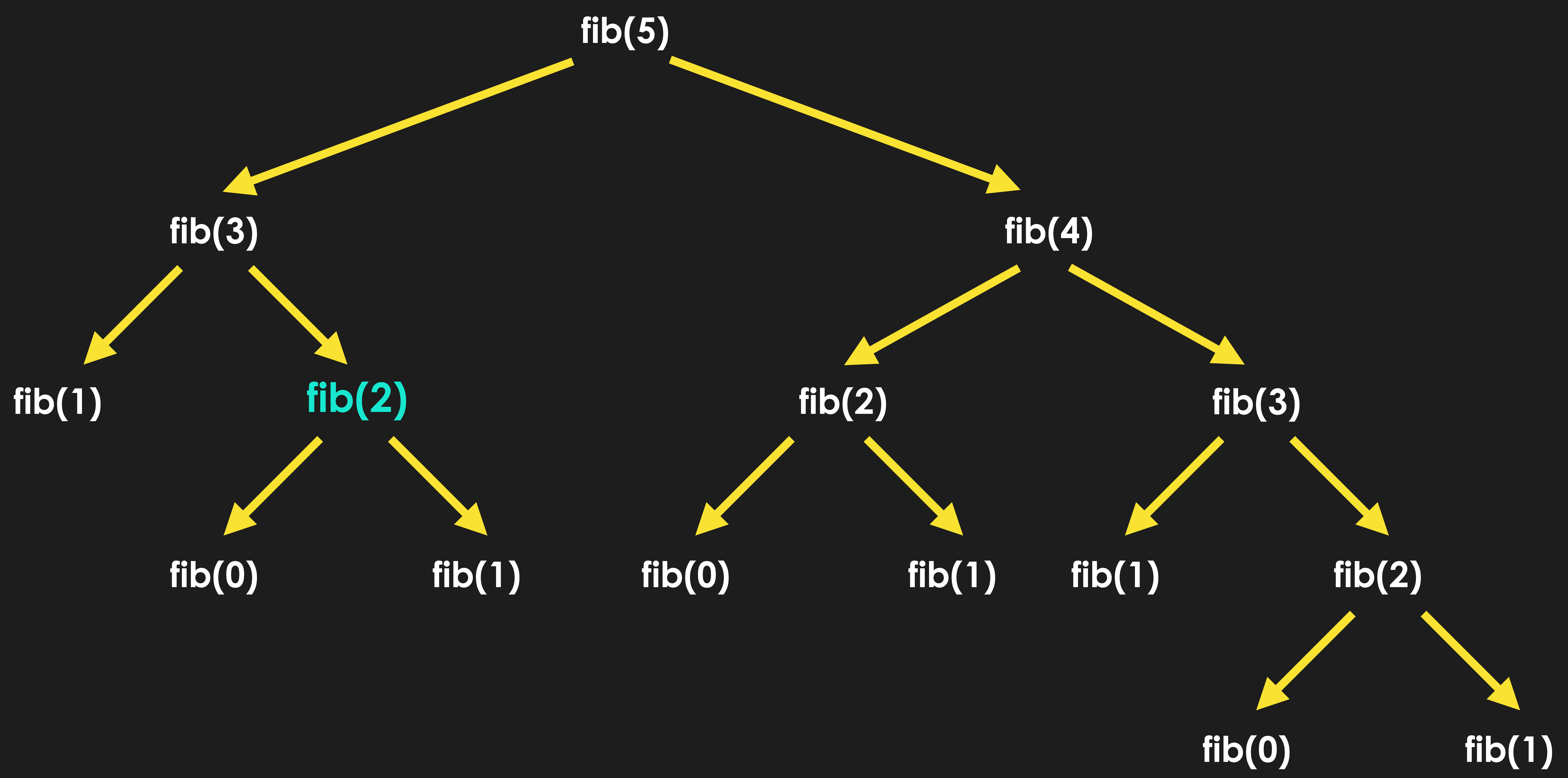
(at any time, the number of recursive calls on the stack is N)

Notice how we are repeating certain computations

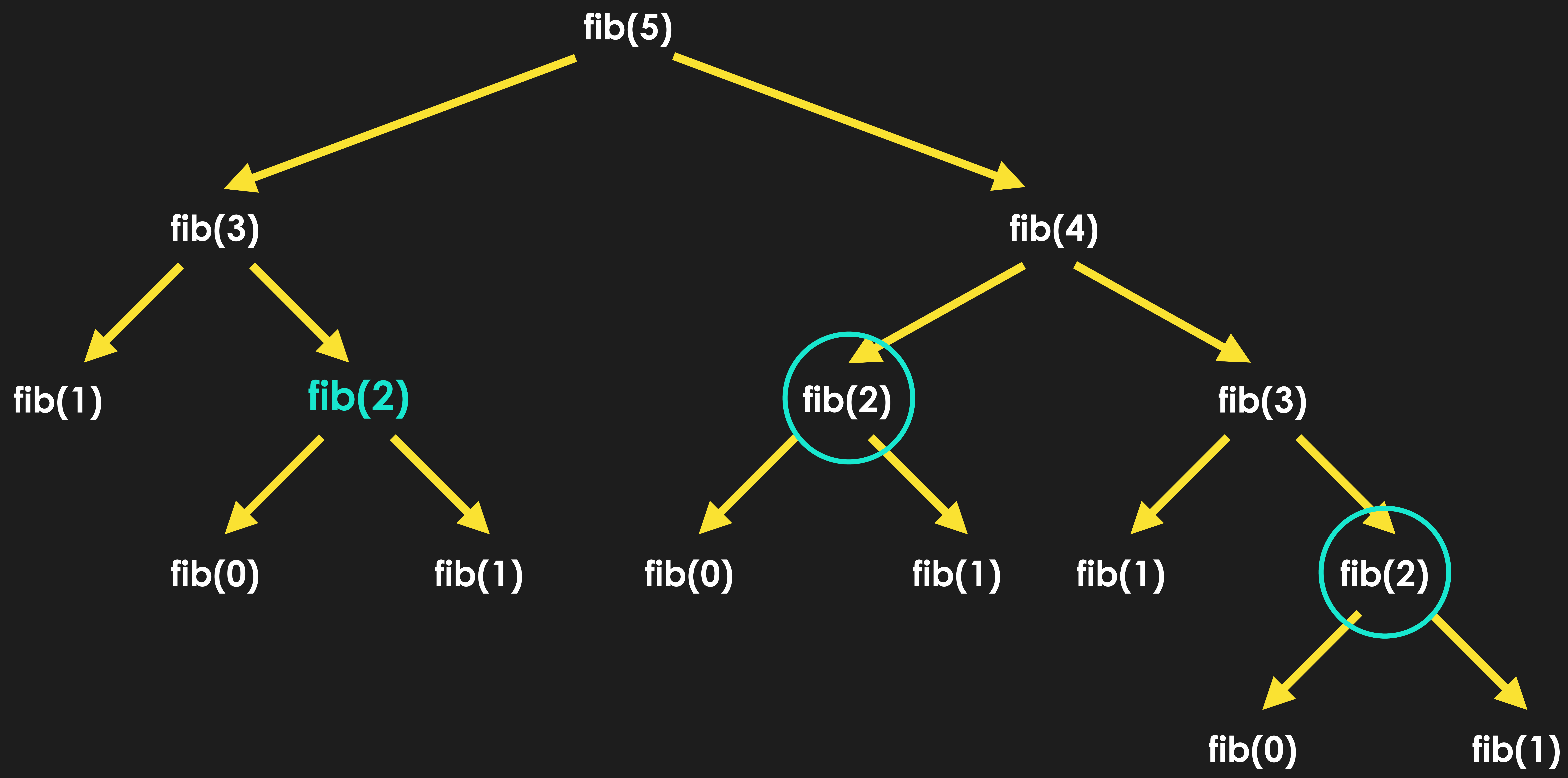


CodeIT

Notice how we are repeating certain computations

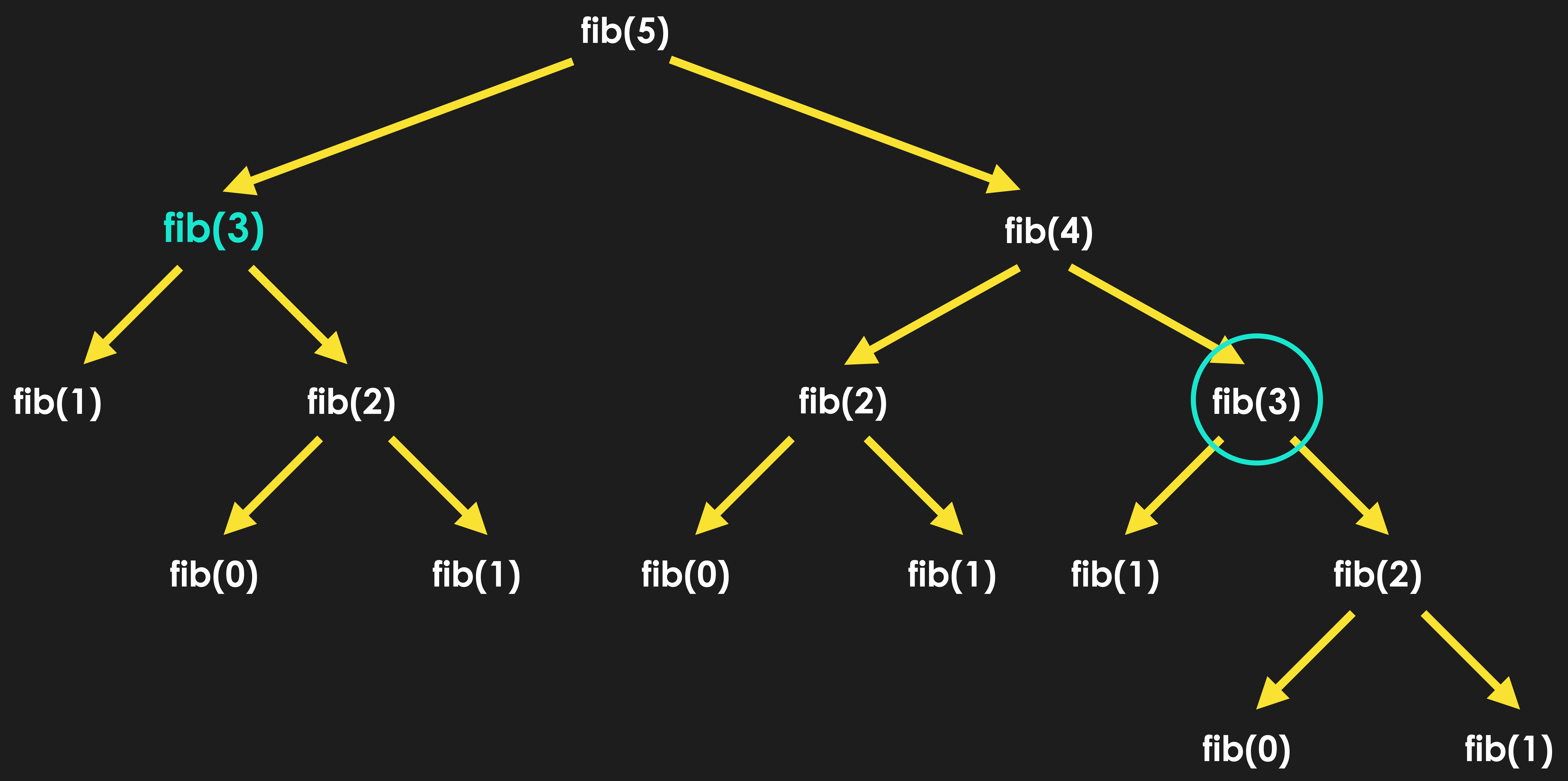


Notice how we are repeating certain computations



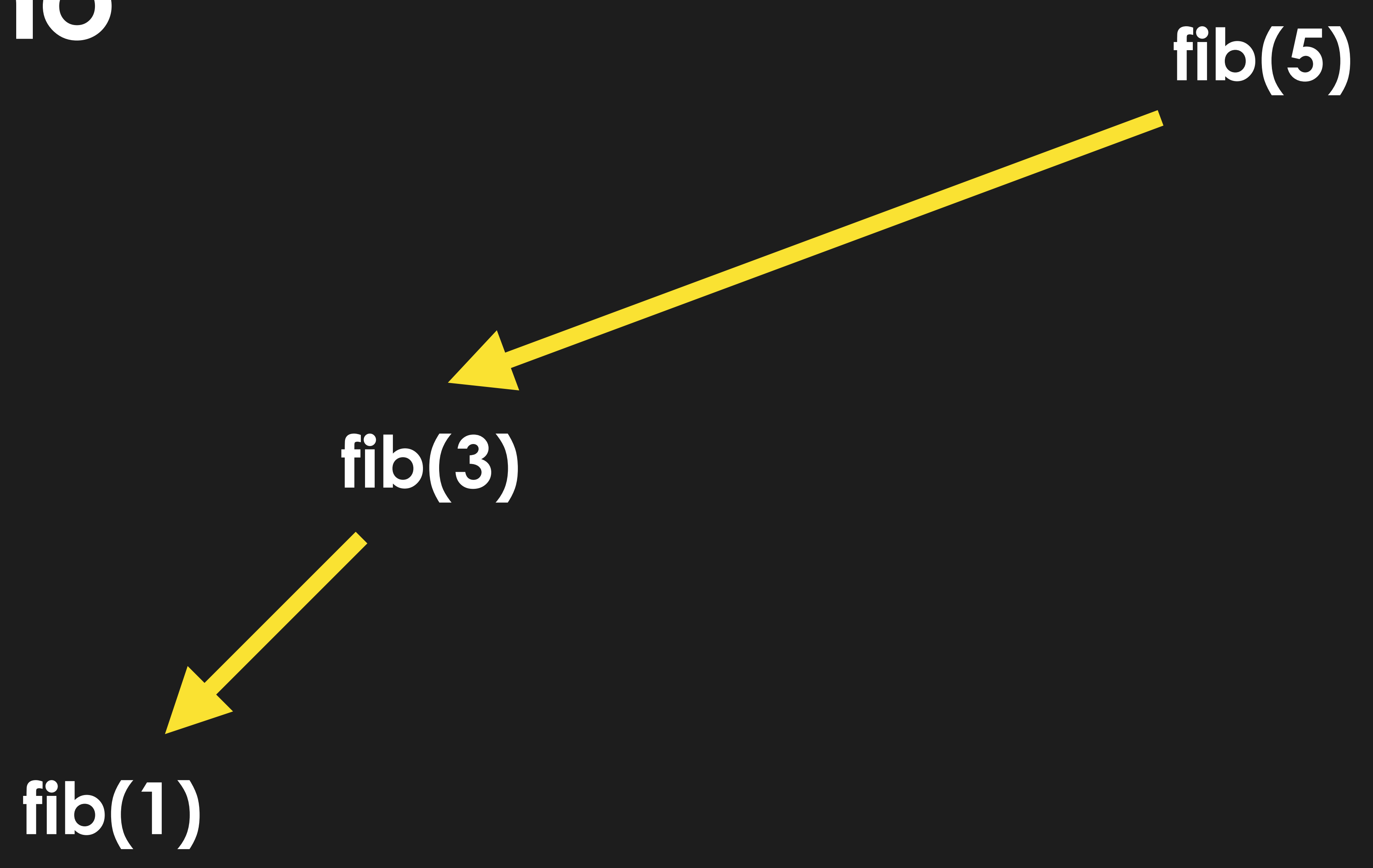
CodeIT

Notice how we are repeating certain computations



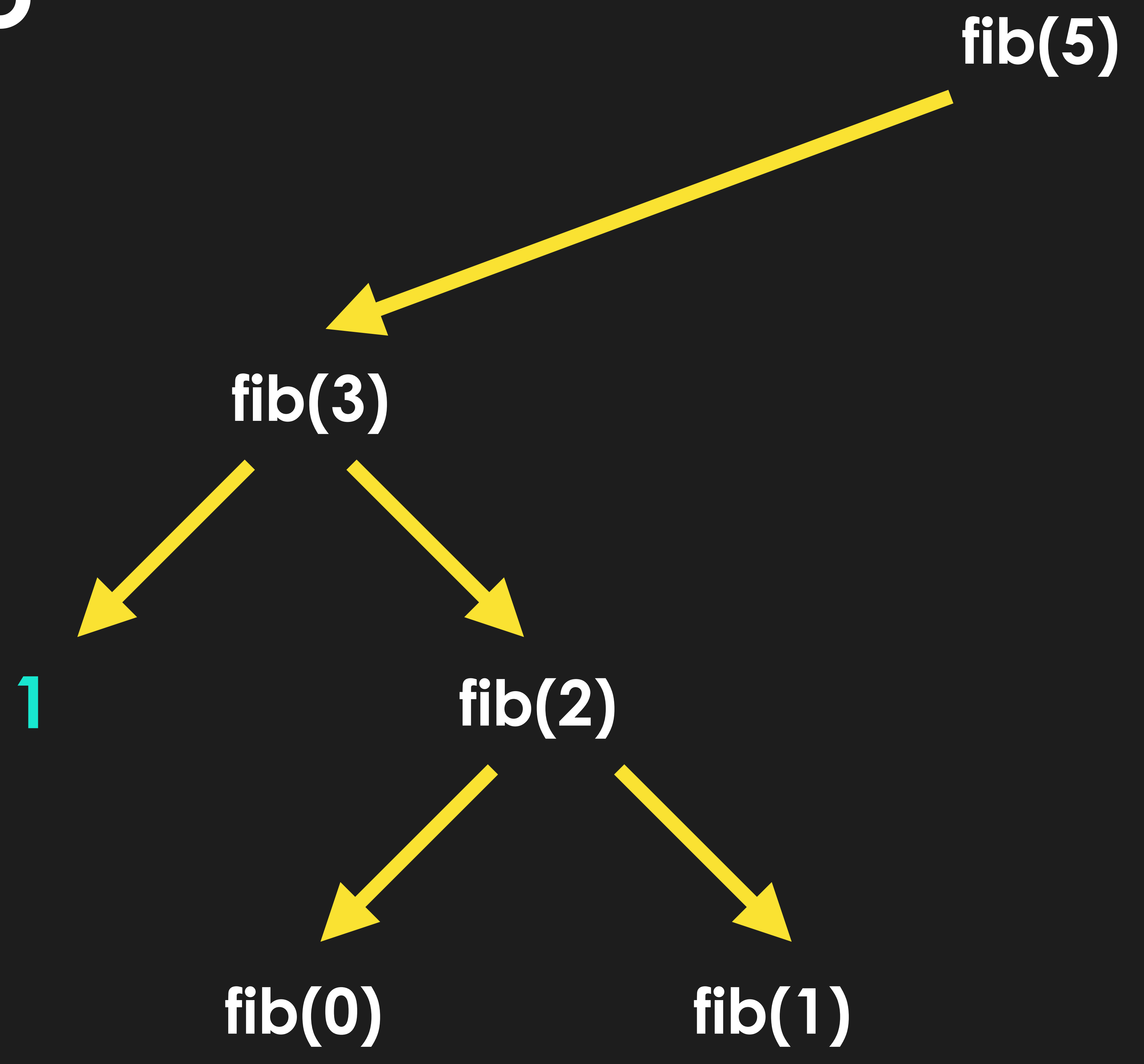
How to avoid recomputation? Memoization!

Fib Memo



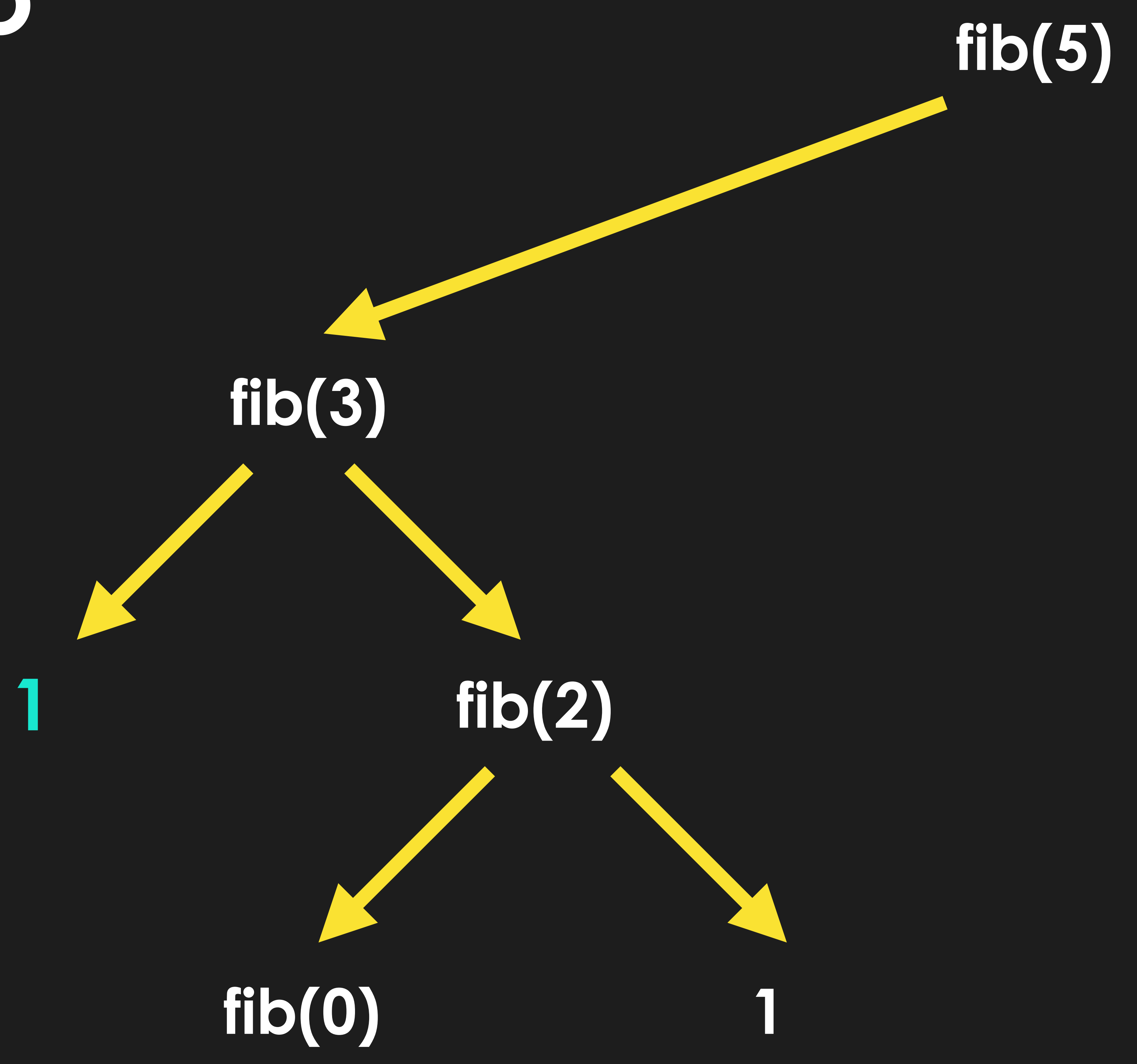
n						
memo[n]						

Fib Memo



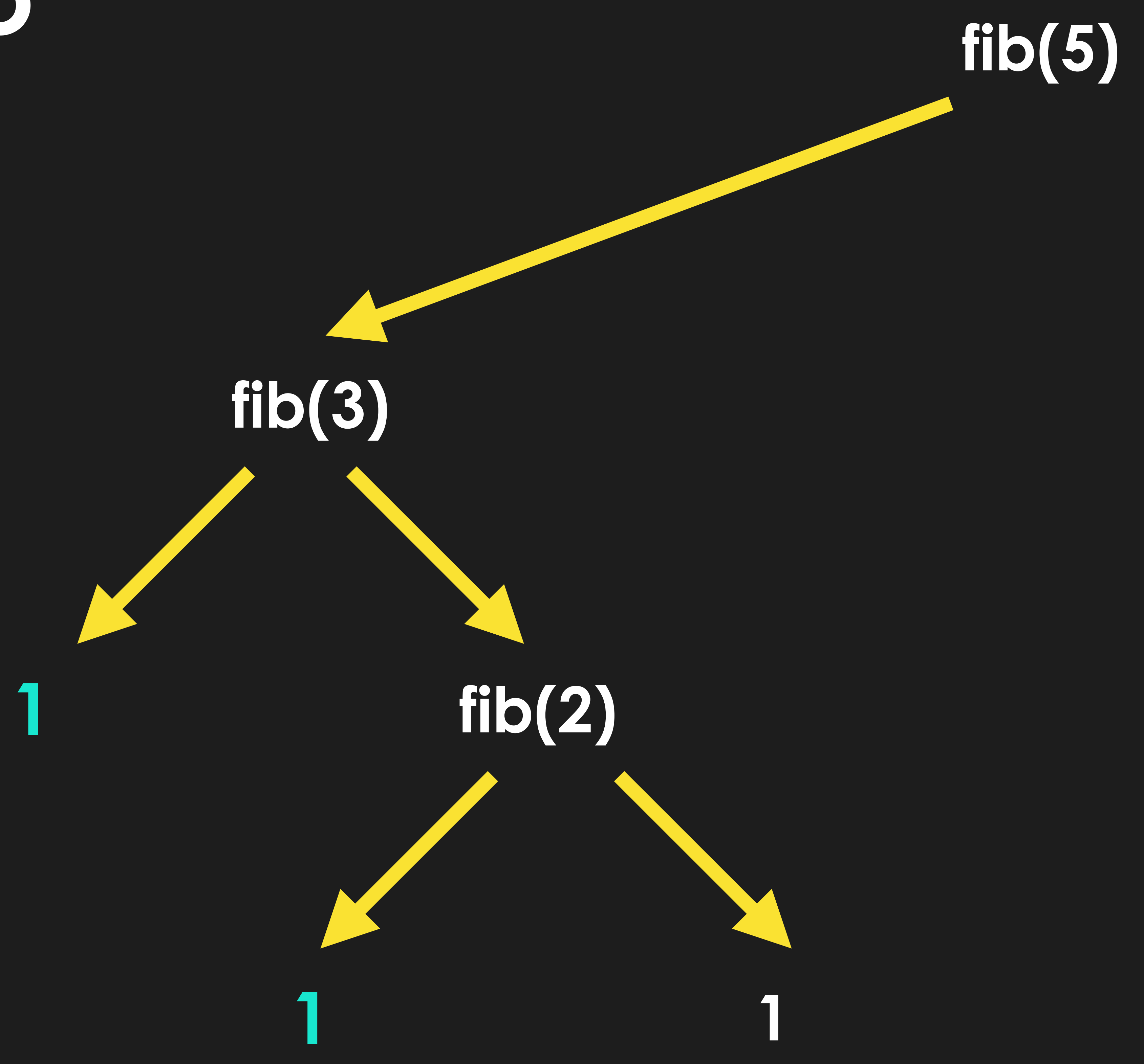
n		1				
memo[n]		1				

Fib Memo



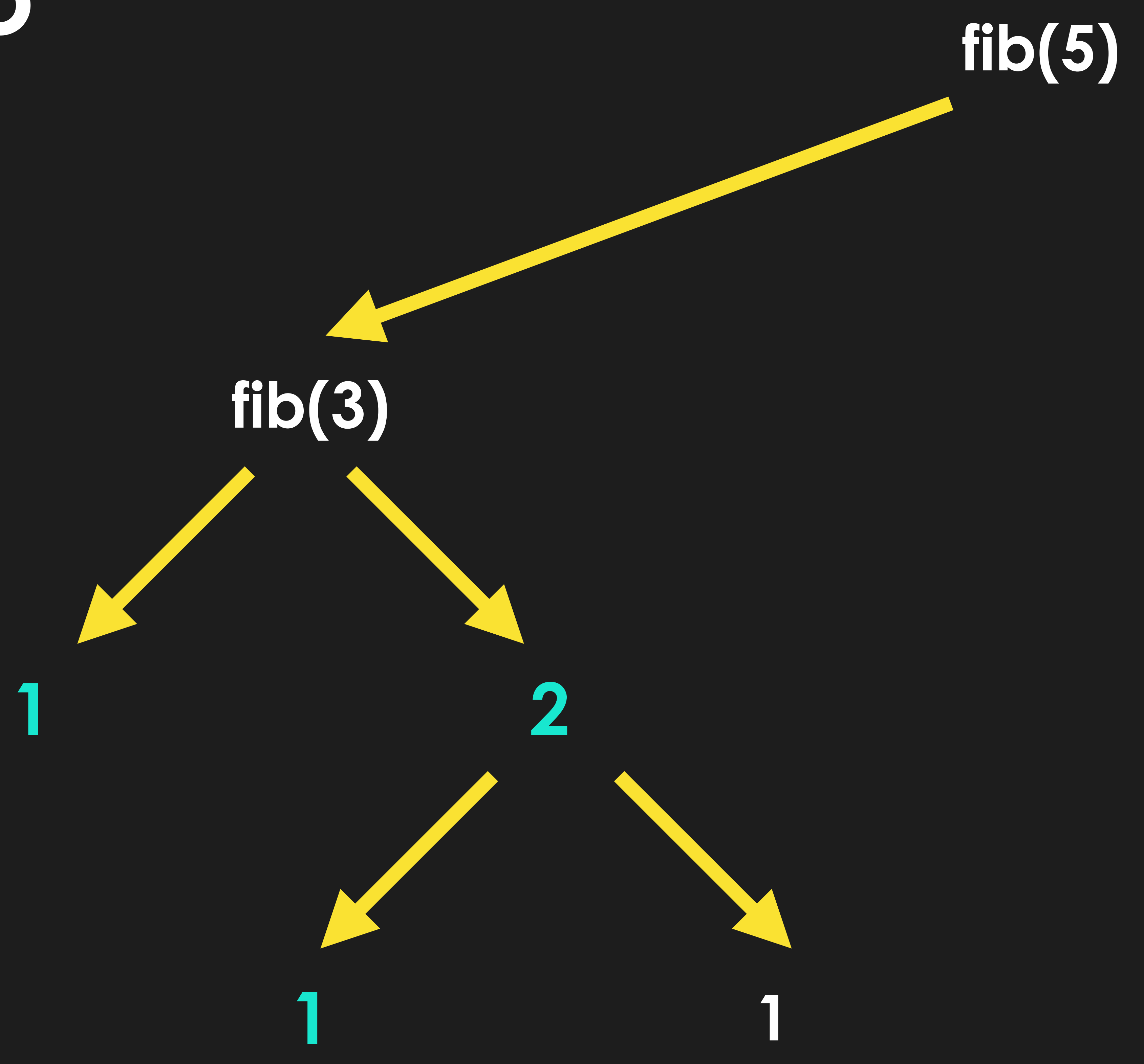
n		1				
memo[n]		1				

Fib Memo



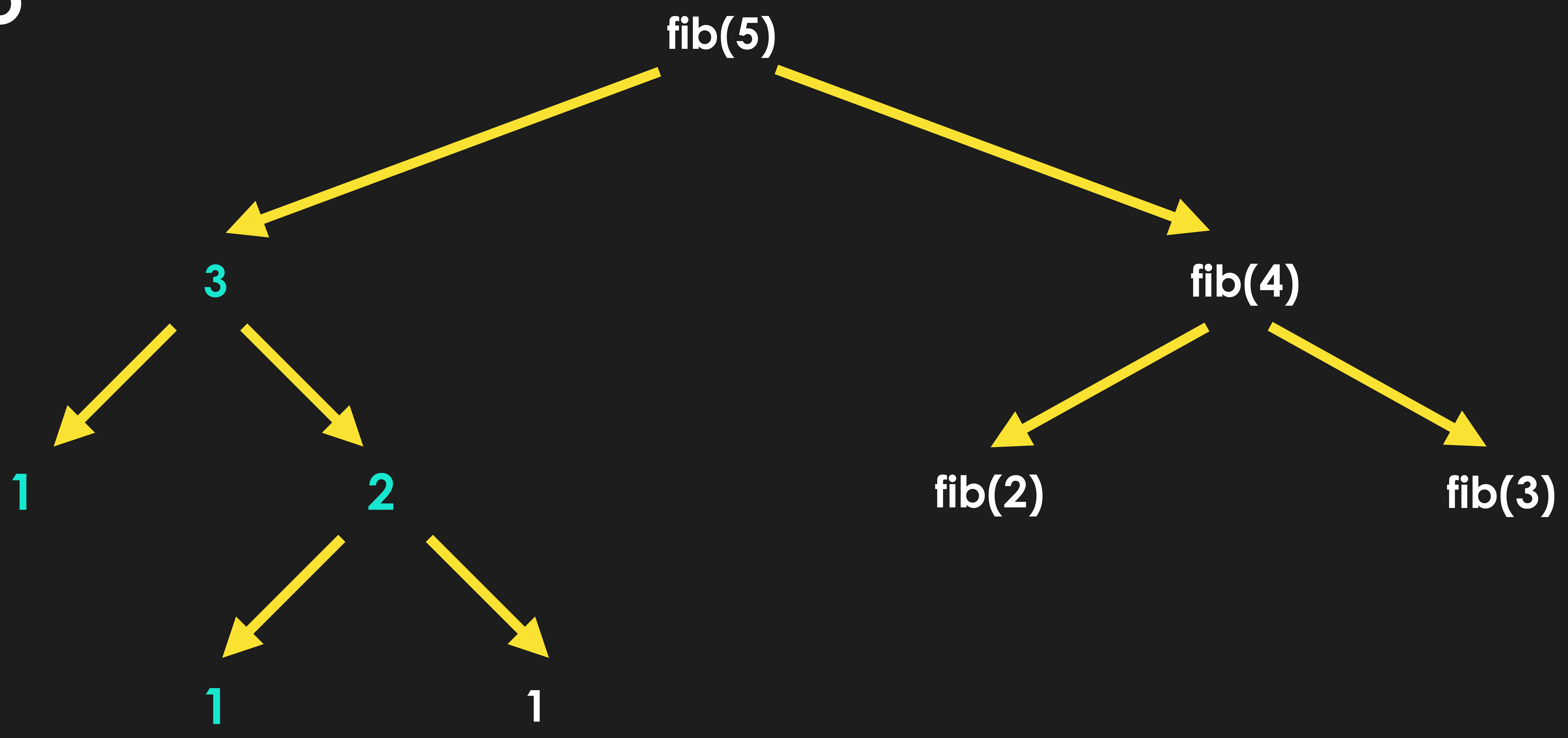
n	0	1				
memo[n]	1	1				

Fib Memo



n	0	1	2			
memo[n]	1	1	2			

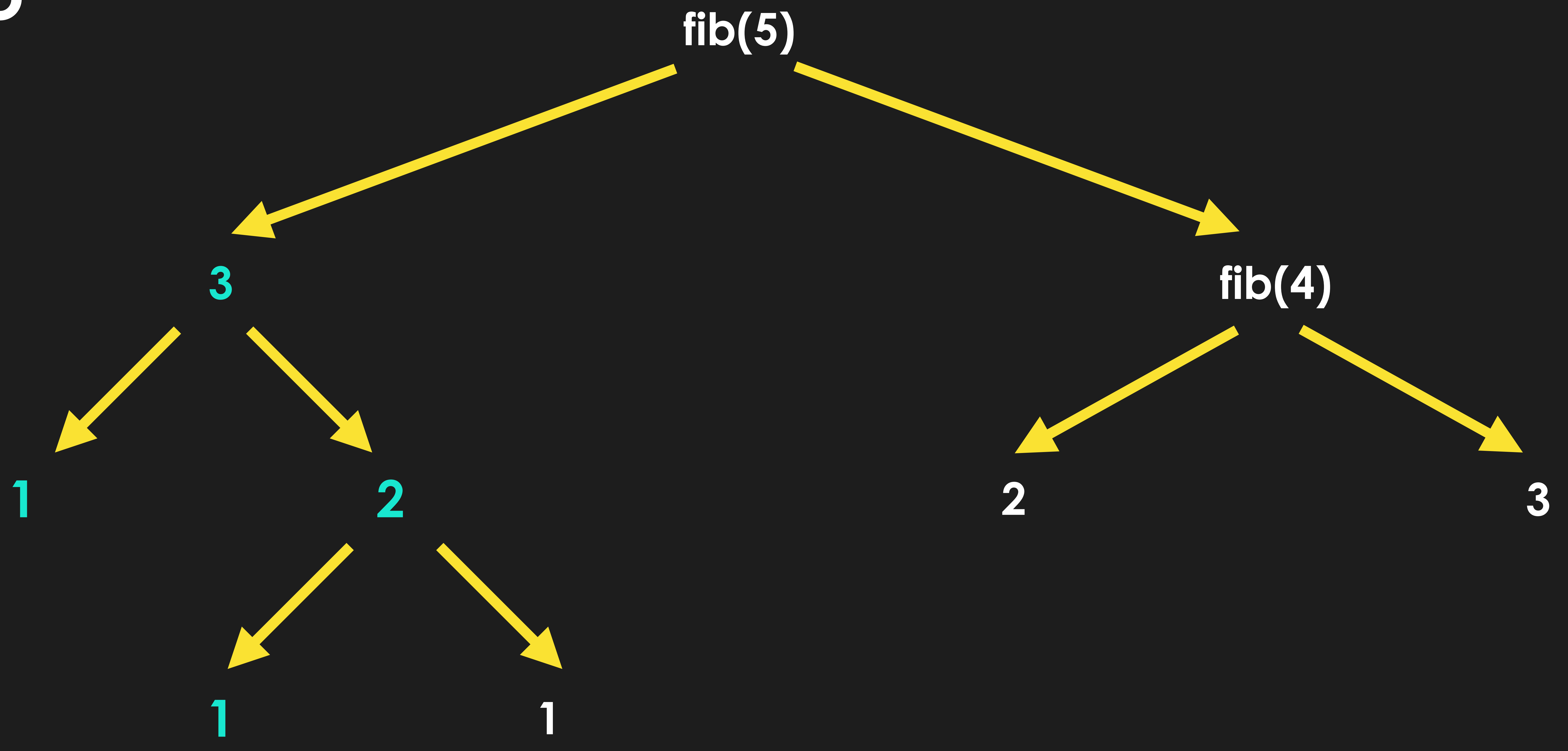
Fib Memo



n	0	1	2	3		
memo[n]	1	1	2	3		

CodeIT

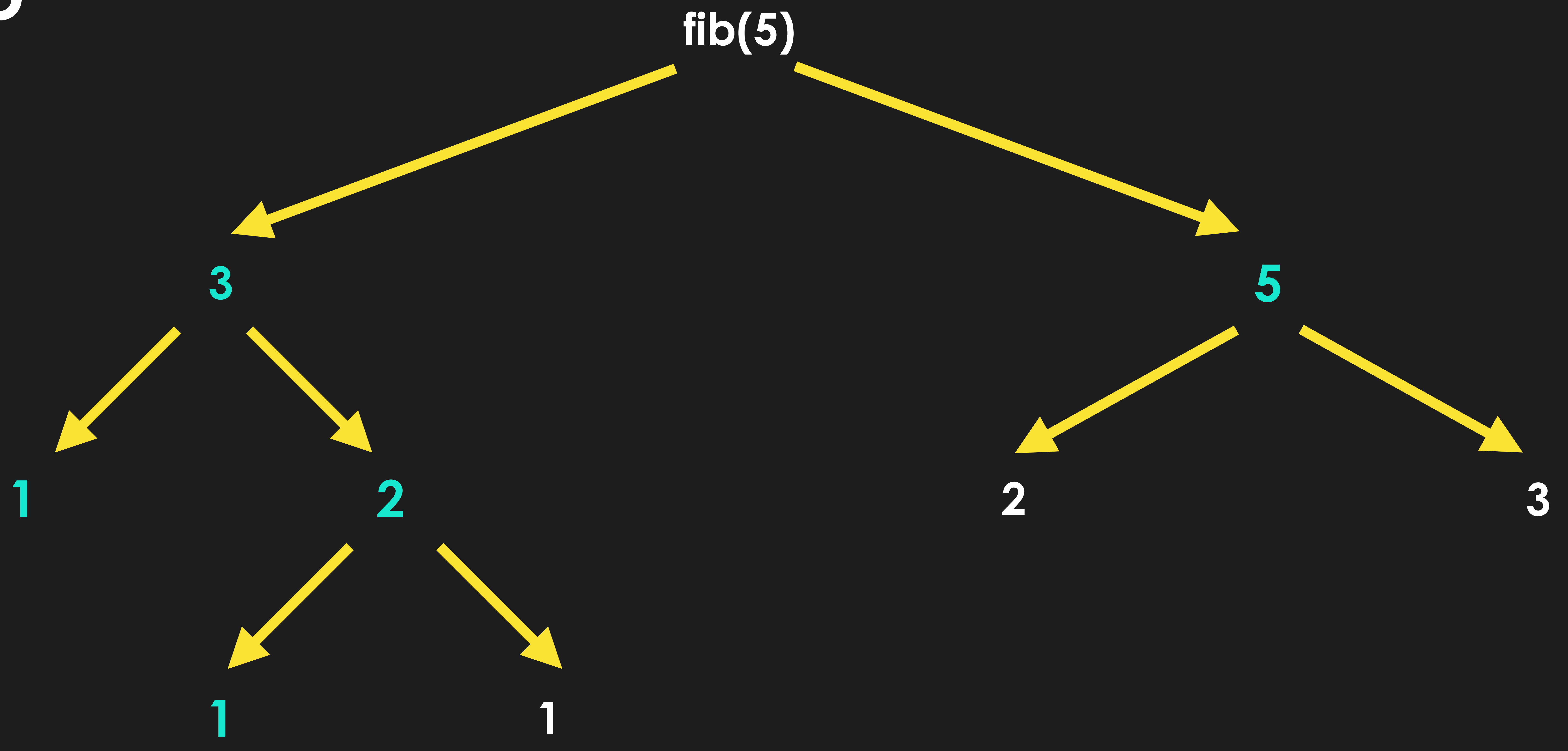
Fib Memo



n	0	1	2	3		
memo[n]	1	1	2	3		

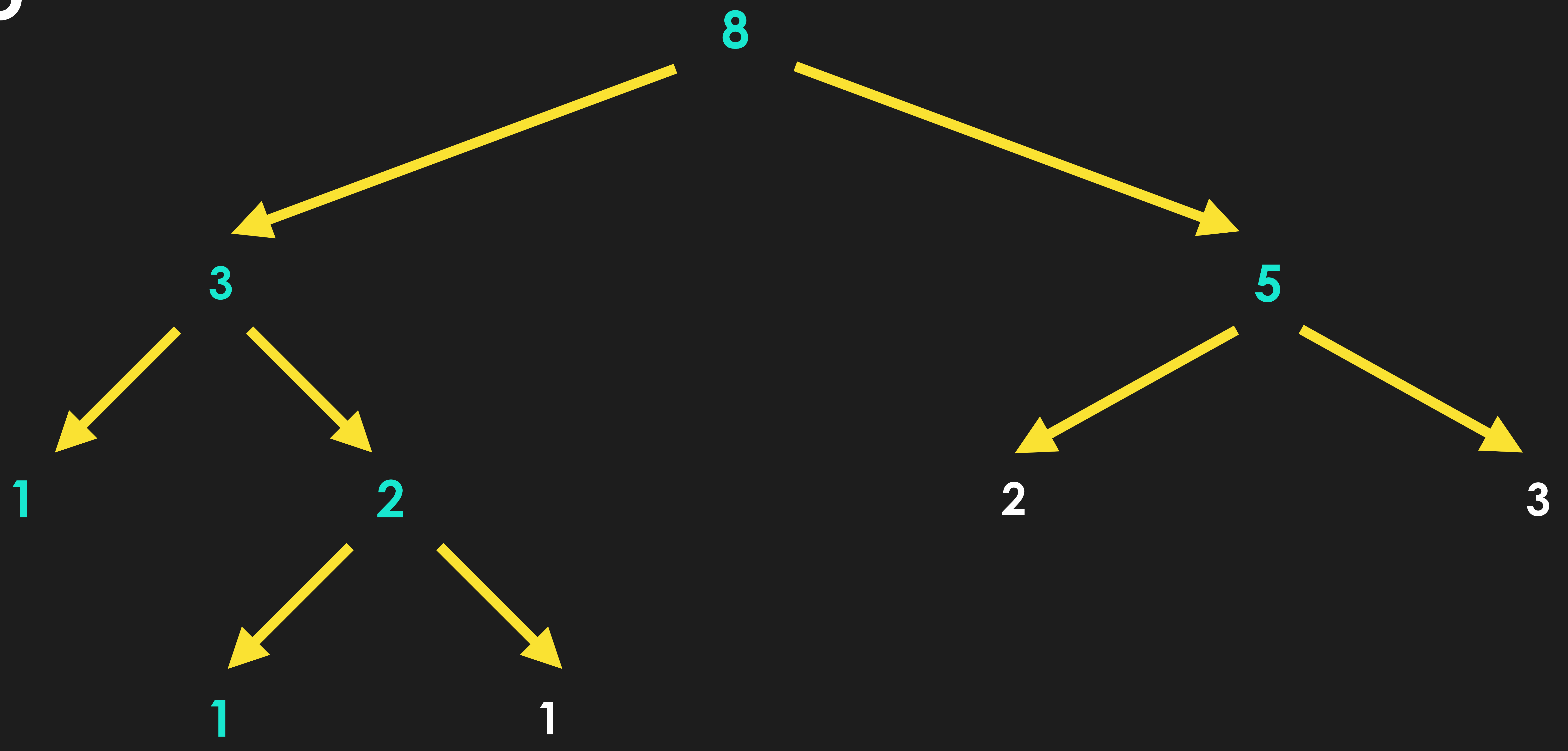
Already in table!

Fib Memo



n	0	1	2	3	4	
memo[n]	1	1	2	3	5	

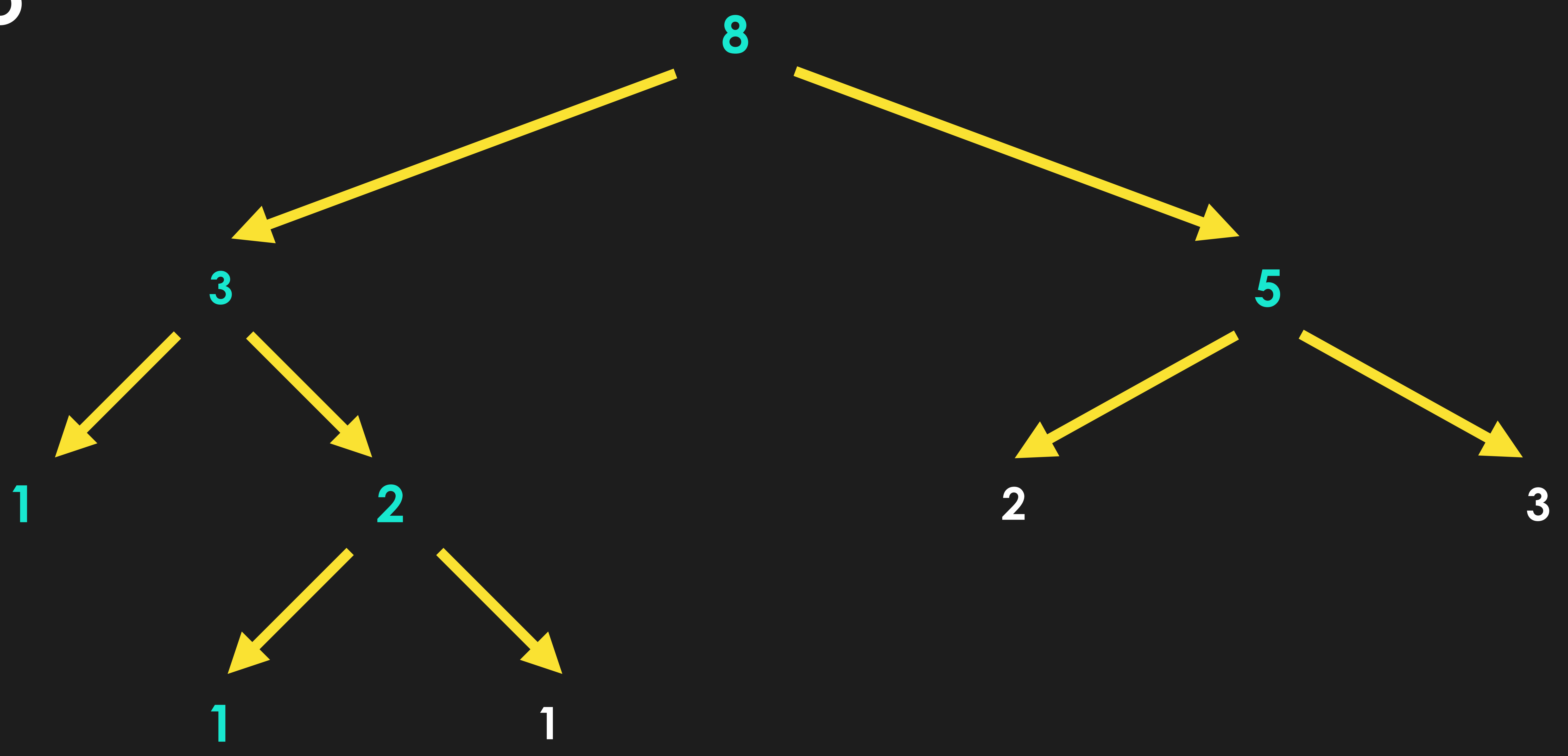
Fib Memo



n	0	1	2	3	4	5
memo[n]	1	1	2	3	5	8

CodeIT

Fib Memo



n	0	1	2	3	4	5
memo[n]	1	1	2	3	5	8

time complexity: N

space complexity: N

Fib Memo

```
memo = {}

def fibMemo(n):
    if n in memo:
        return memo[n]

    elif n == 0 or n == 1:
        return 1

    else:
        memo[n] = fibMemo(n - 1) + fibMemo(n - 2)
        return memo[n]
```

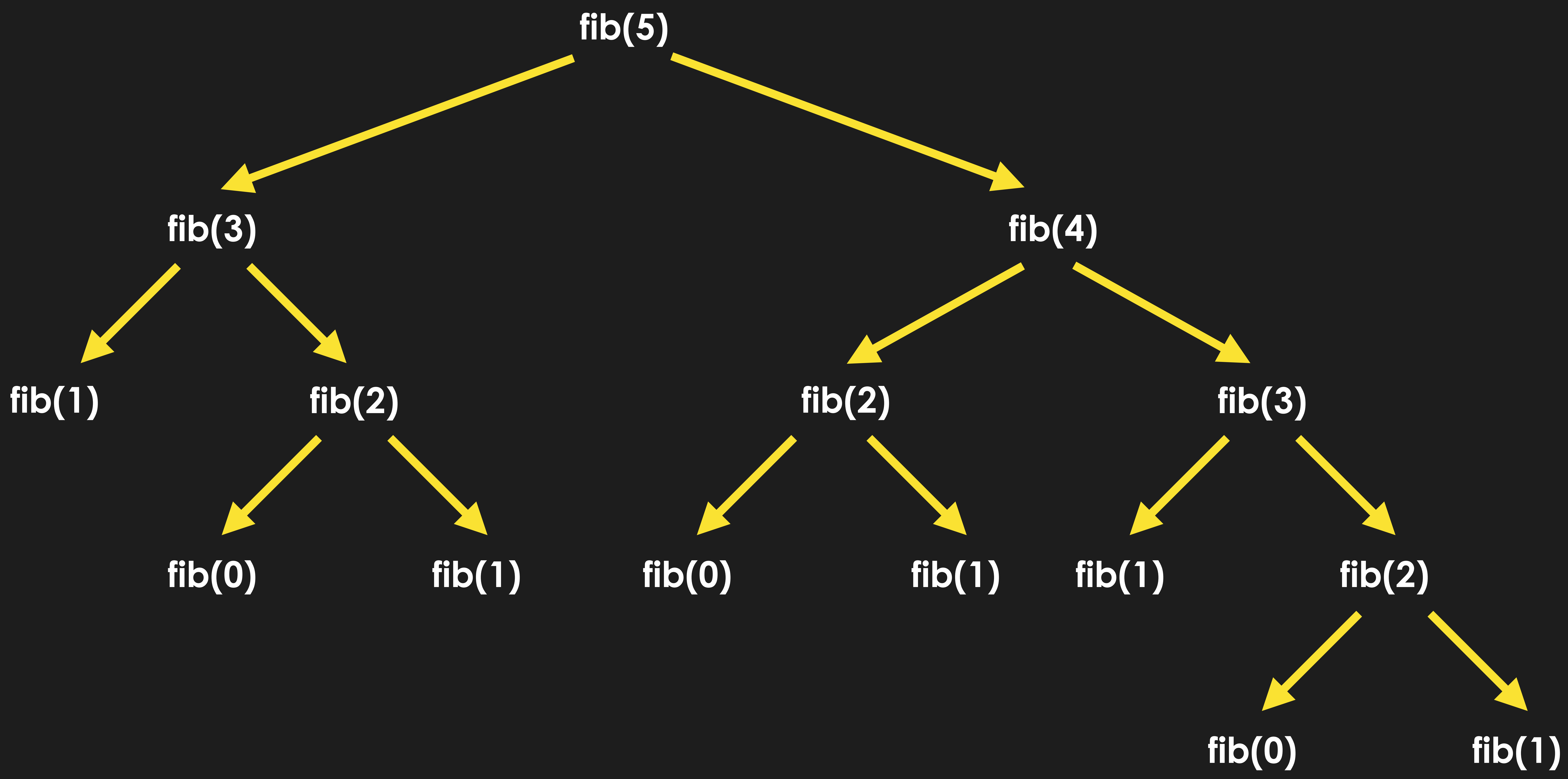
CodeIT

Identifying a DP problem

Properties of a dynamic problem

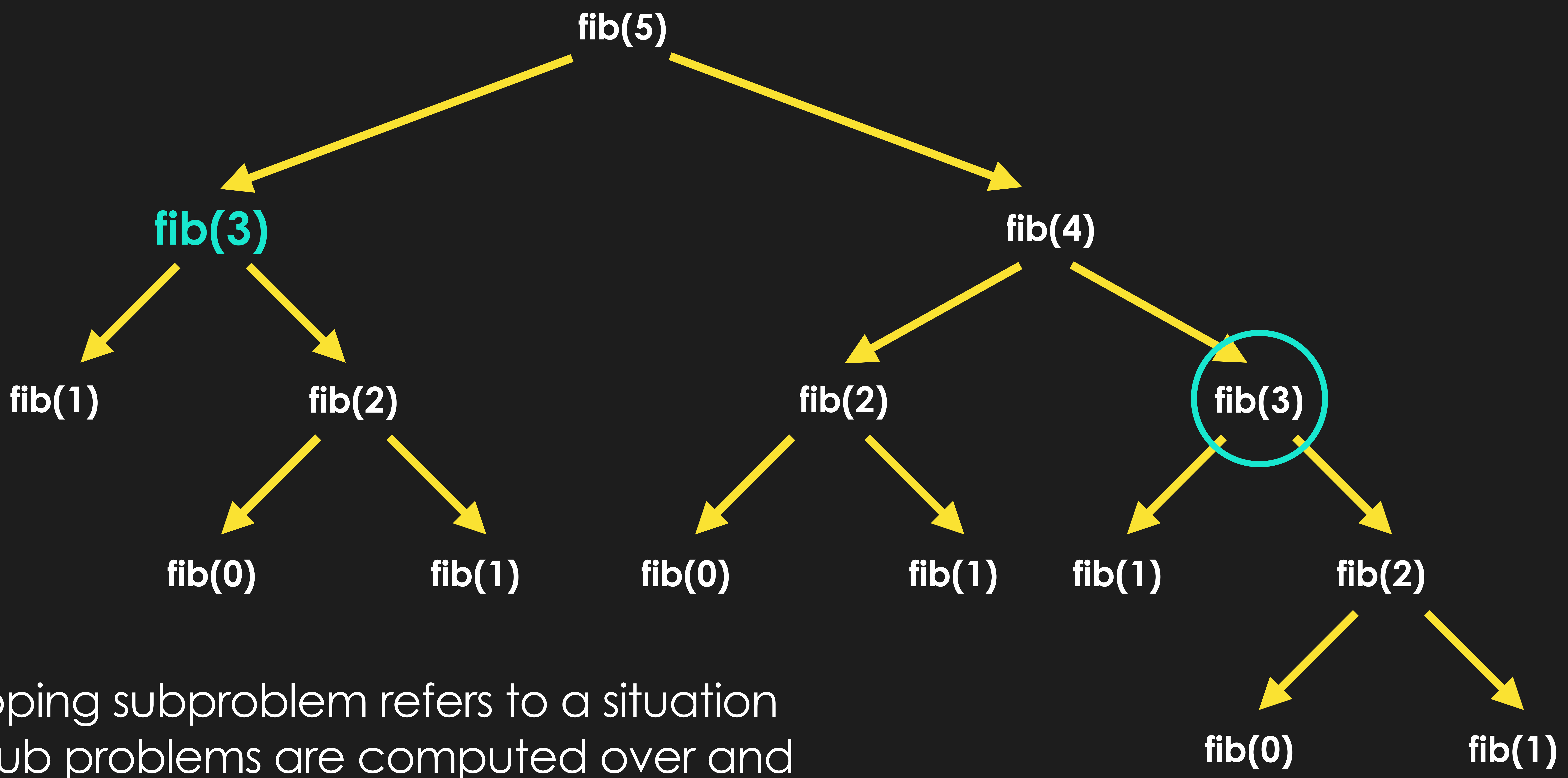
- Overlapping subproblems
- Optimal substructure

Overlapping subproblems



CodeIT

Overlapping subproblems



Overlapping subproblem refers to a situation where sub problems are computed over and over again

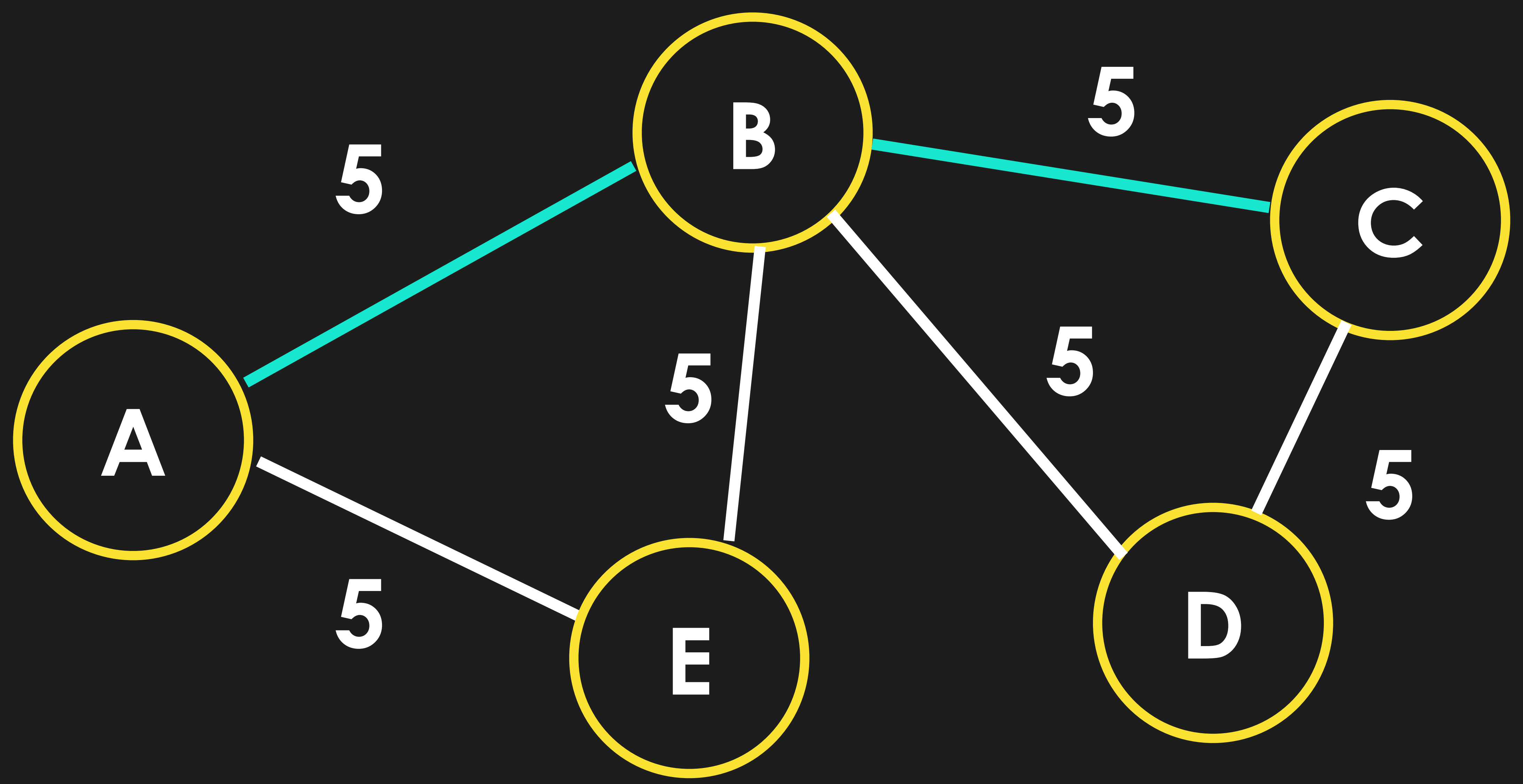
CodeIT

Optimal substructure is when for a **problem of size N**, the optimal solution can be derived or calculated from the **optimal solutions to some number of sub problems**

Optimal substructure in fib

- fib(5) = fib(4) + fib(3)
- The **optimal solution for fib(5)**, can be calculated from the **optimal solution of sub problem fib(4)** and **optimal solution of sub problem fib(3)**

Another example of optimal substructure: shortest paths



Given solutions to subproblems [**SP(A, D)**, **SP (B, C)**, **SP (A, D)**, **SP (D, C)**], the larger problem **SP(A, C)** can be found

Brain Teaser: How do you think greedy algorithms and optimal substructures relate?

CodeIT

Back to properties of a dynamic problem

- **Overlapping subproblems**

If a problem has overlapping subproblems, then recursive solutions to subproblems can be stored in a cache (memo object) and thus recomputation can be avoided

- **Optimal substructure:**

If a problem has an optimal substructure, then it can be recursively broken down into sub problems and built back up using the subproblem solutions

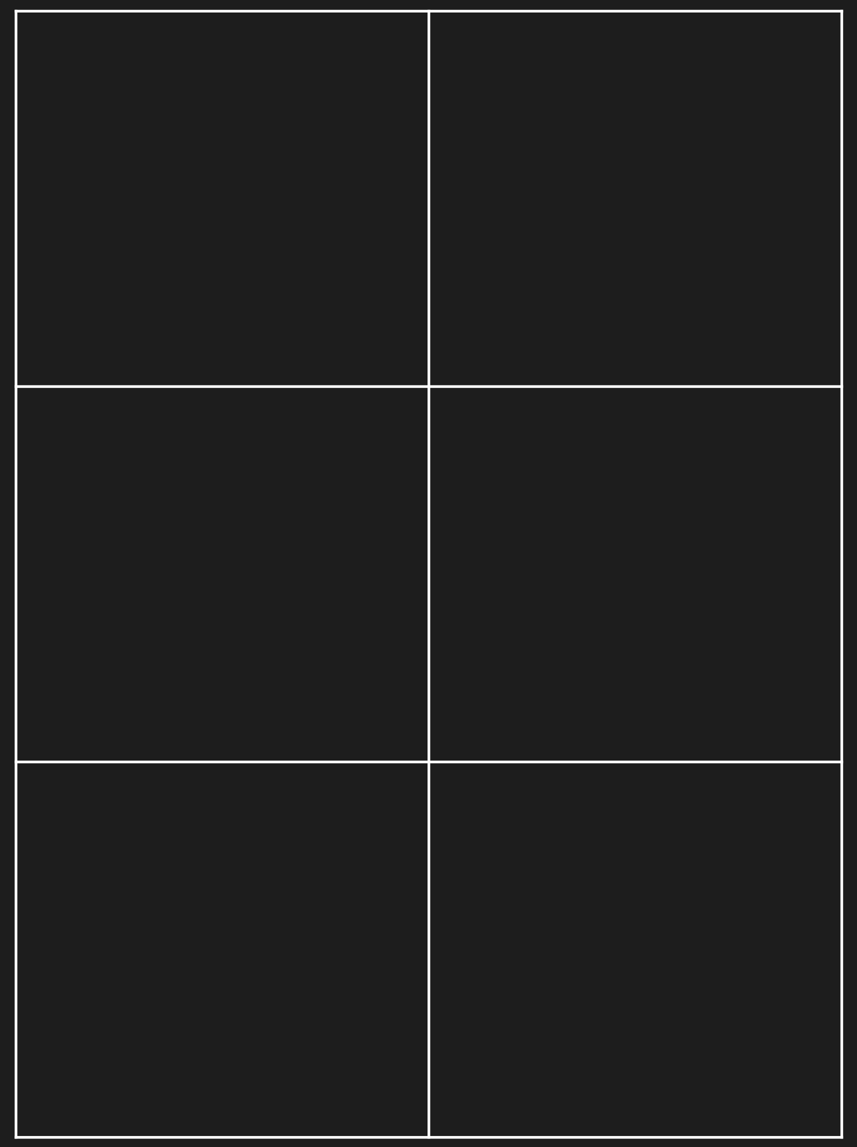
This is the rationale behind memoization and recursion to solve DP problems!

DP Problem #2: Unique Paths

DP Problem #2: Unique Paths

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

Example: 3 x 2 grid

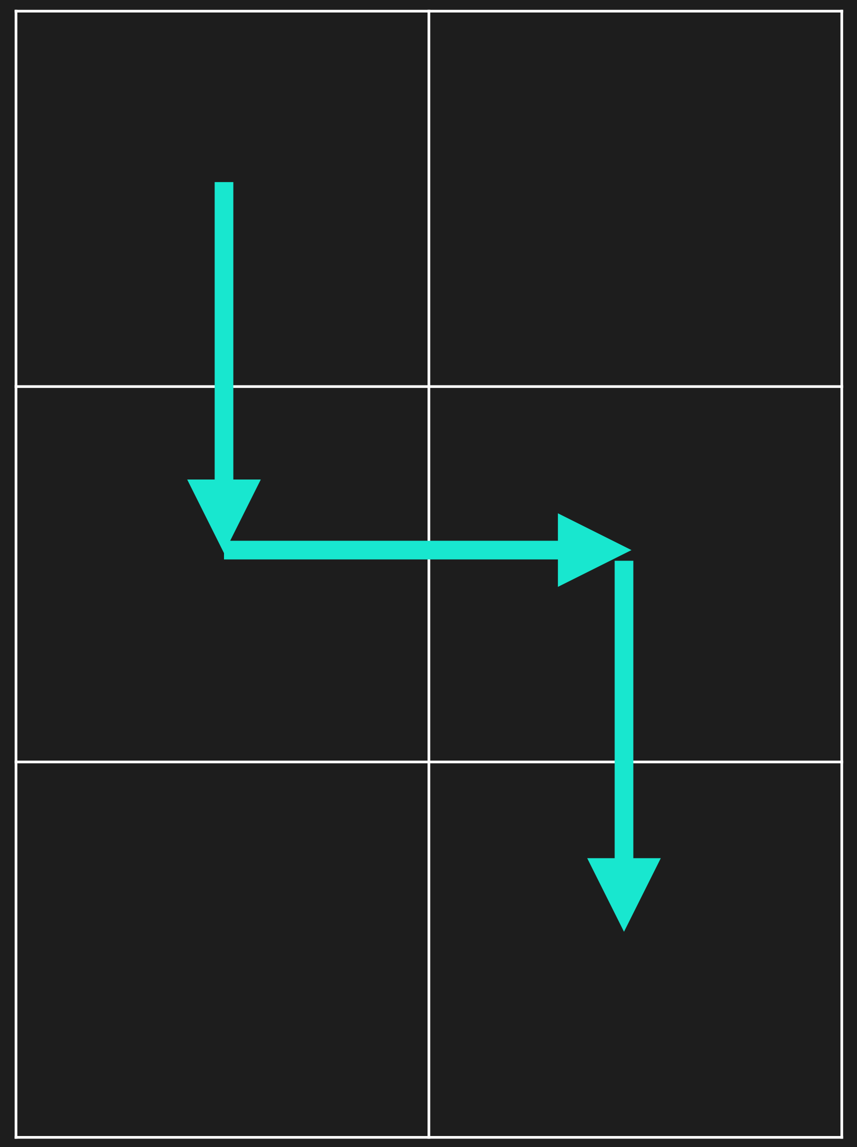


UniquePaths(3, 2) =>

DP Problem #2: Unique Paths

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

Example: 3 x 2 grid

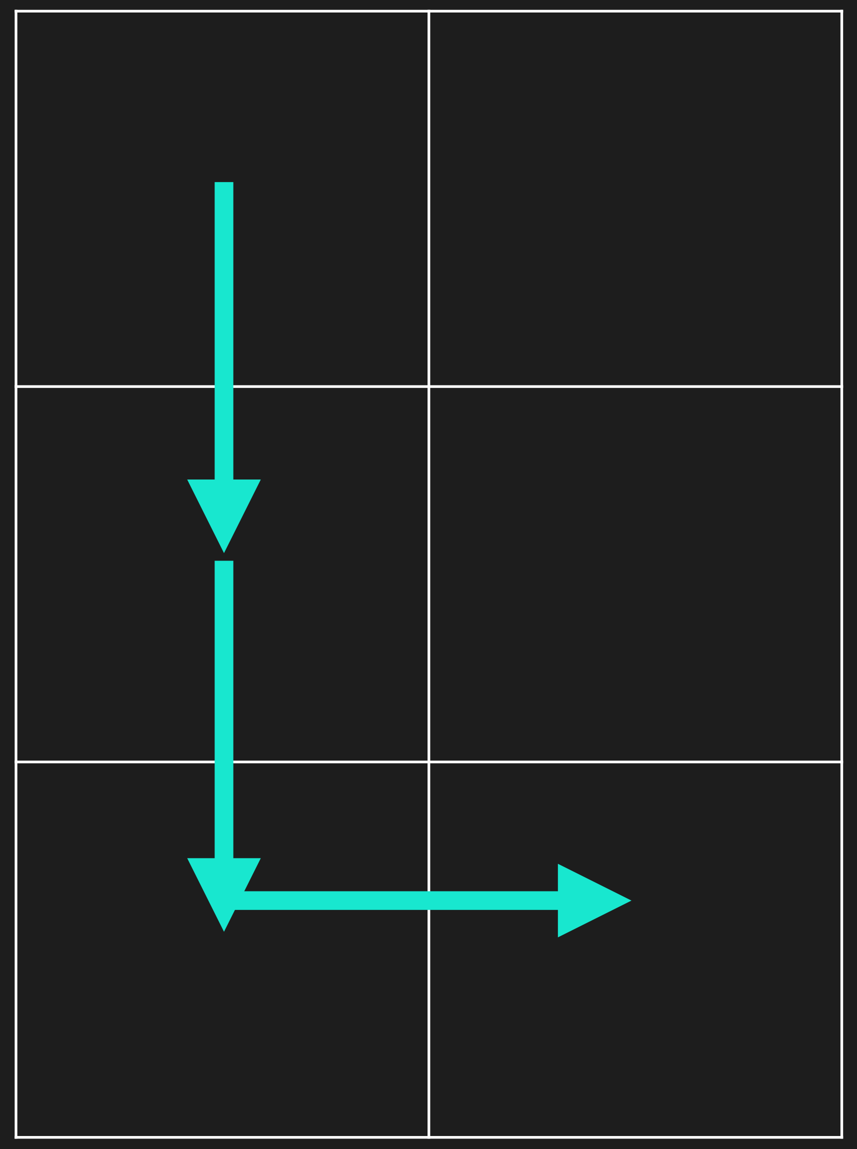


UniquePaths(3, 2) => 1

DP Problem #2: Unique Paths

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

Example: 3 x 2 grid

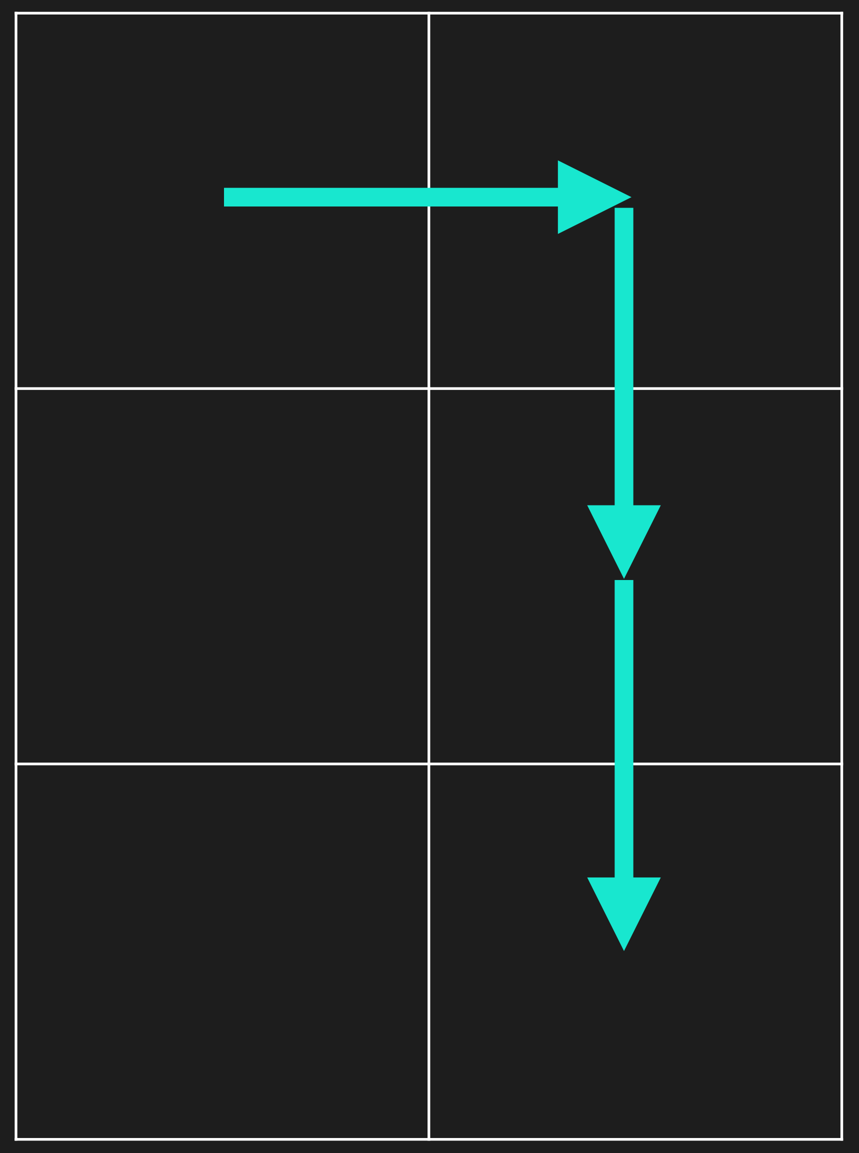


UniquePaths(3, 2) => 2

DP Problem #2: Unique Paths

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

Example: 3 x 2 grid



UniquePaths(3, 2) => 3

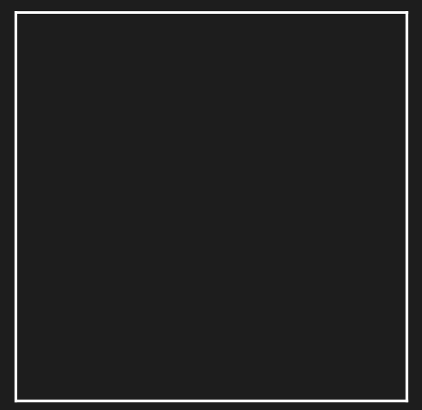
DP Problem #2: Unique Paths

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

Defining the problem: smallest problem size

If ether m or n is 0:
uniquePaths is 0

For a 1 x 1, 1 x 2, 2 x 1 board:
uniquePaths is 1



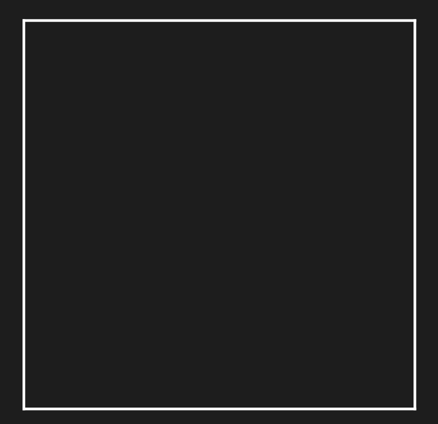
DP Problem #2: Unique Paths

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

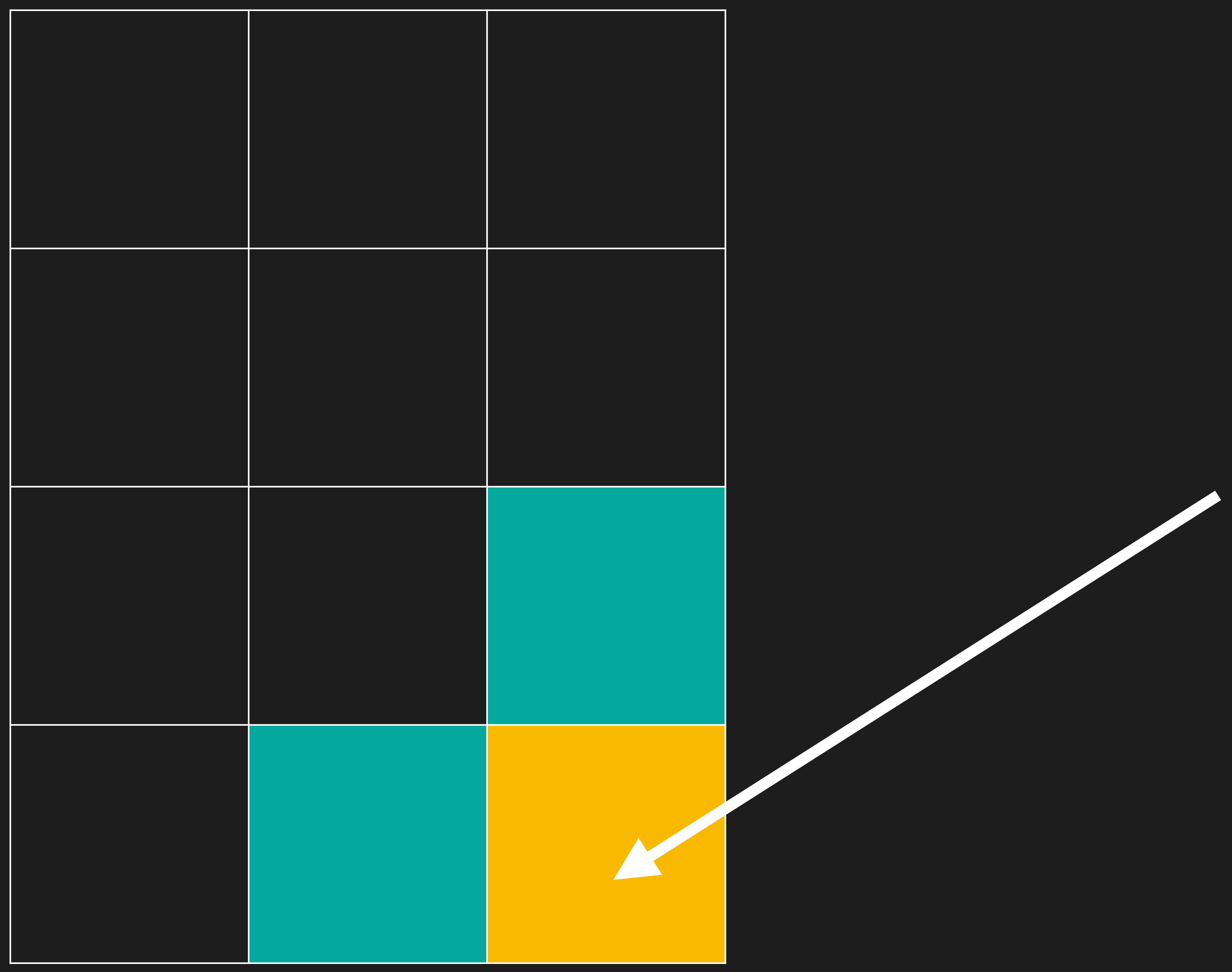
Defining the problem: smallest problem size

If ether m or n is 0:
uniquePaths is 0

For a 1 x 1, 1 x 2, 2 x 1 board:
uniquePaths is 1



Unique Paths Optimal Substructure



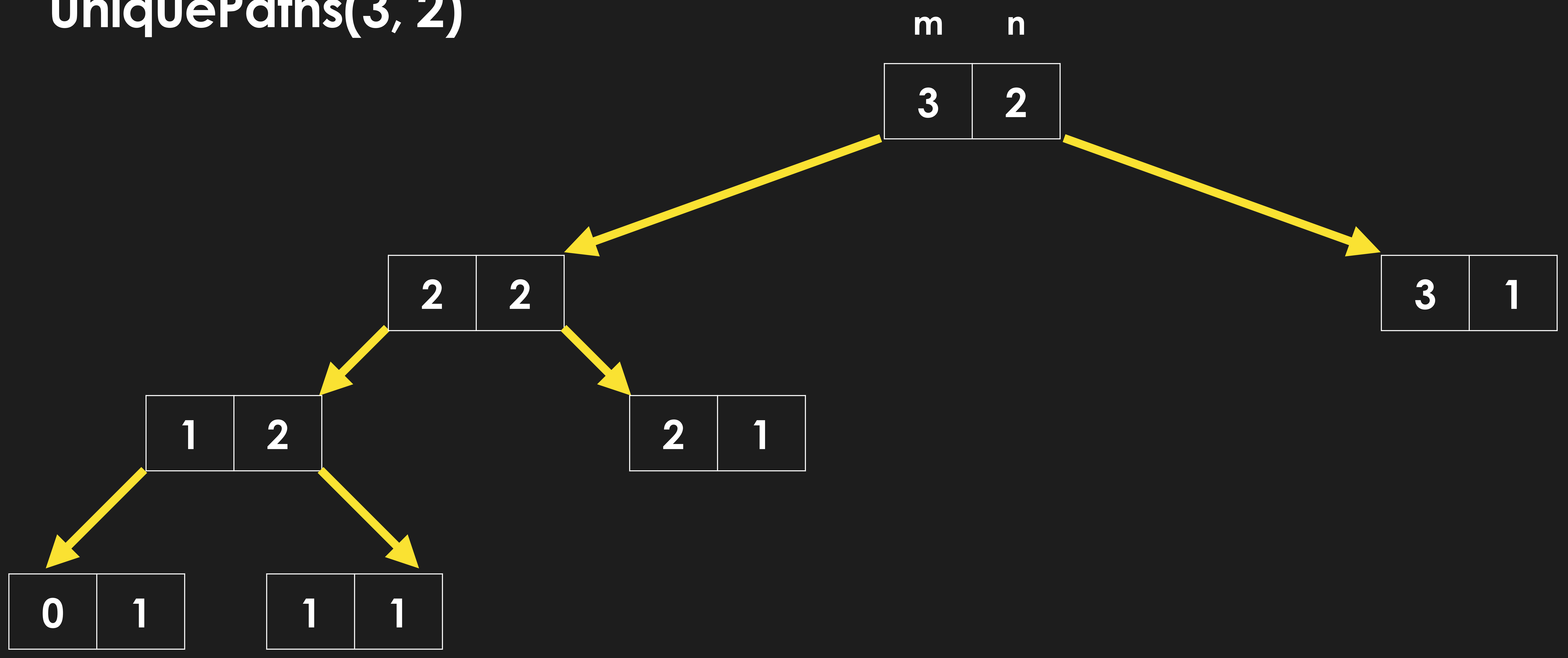
To get to this cell, you can only come from top or left

Hence, the total number of unique paths to the last cell will be:

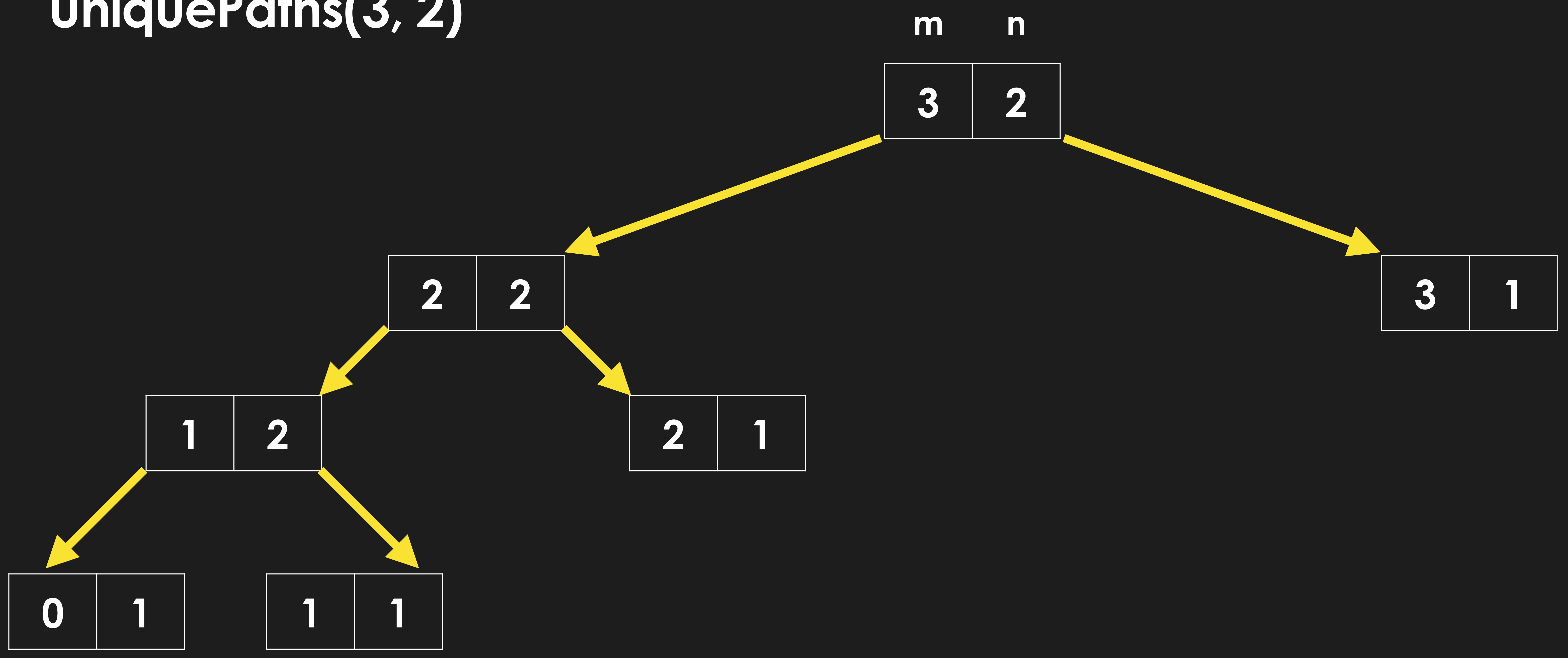
Total number of unique paths to top cell + total number of unique paths to left cell

This is essentially the same as:
uniquePaths(m, n) =
uniquePaths(m - 1, n) + **uniquePaths(m, n - 1)**

UniquePaths(3, 2)

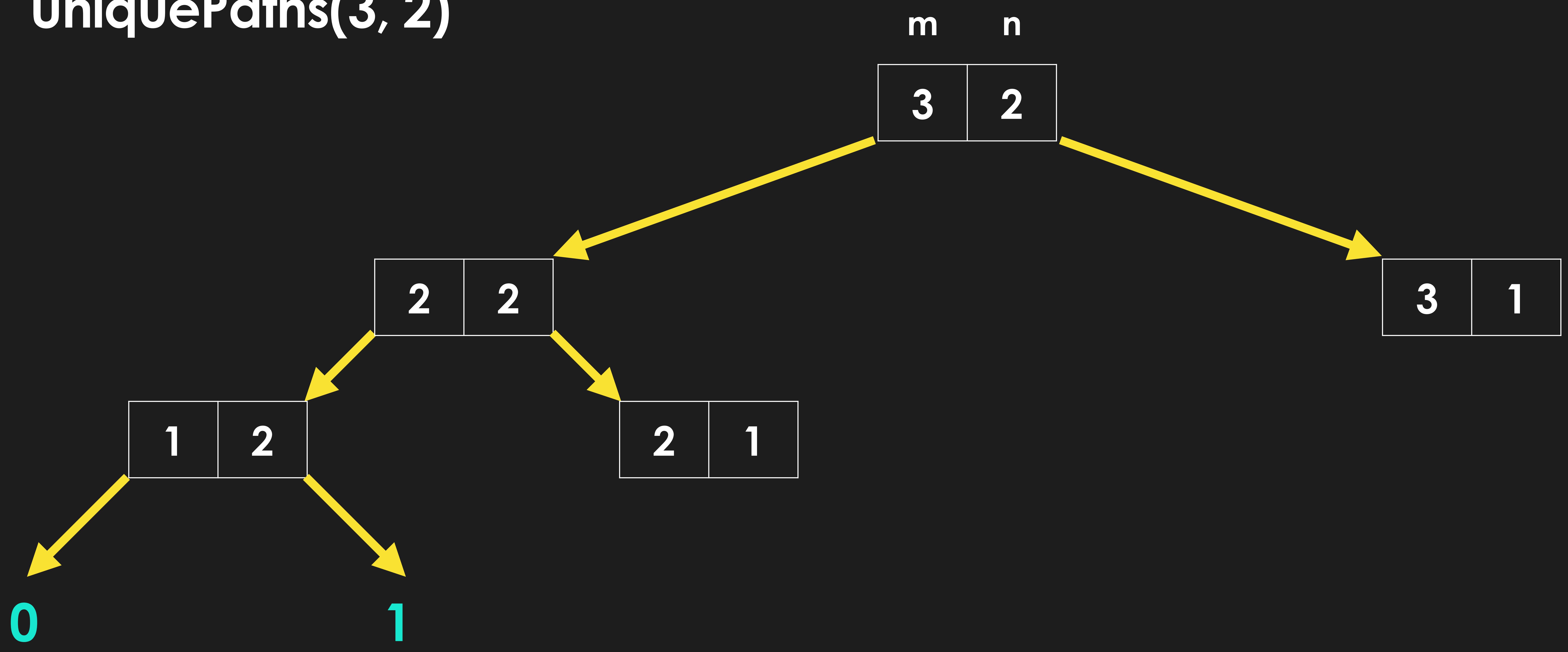


UniquePaths(3, 2)



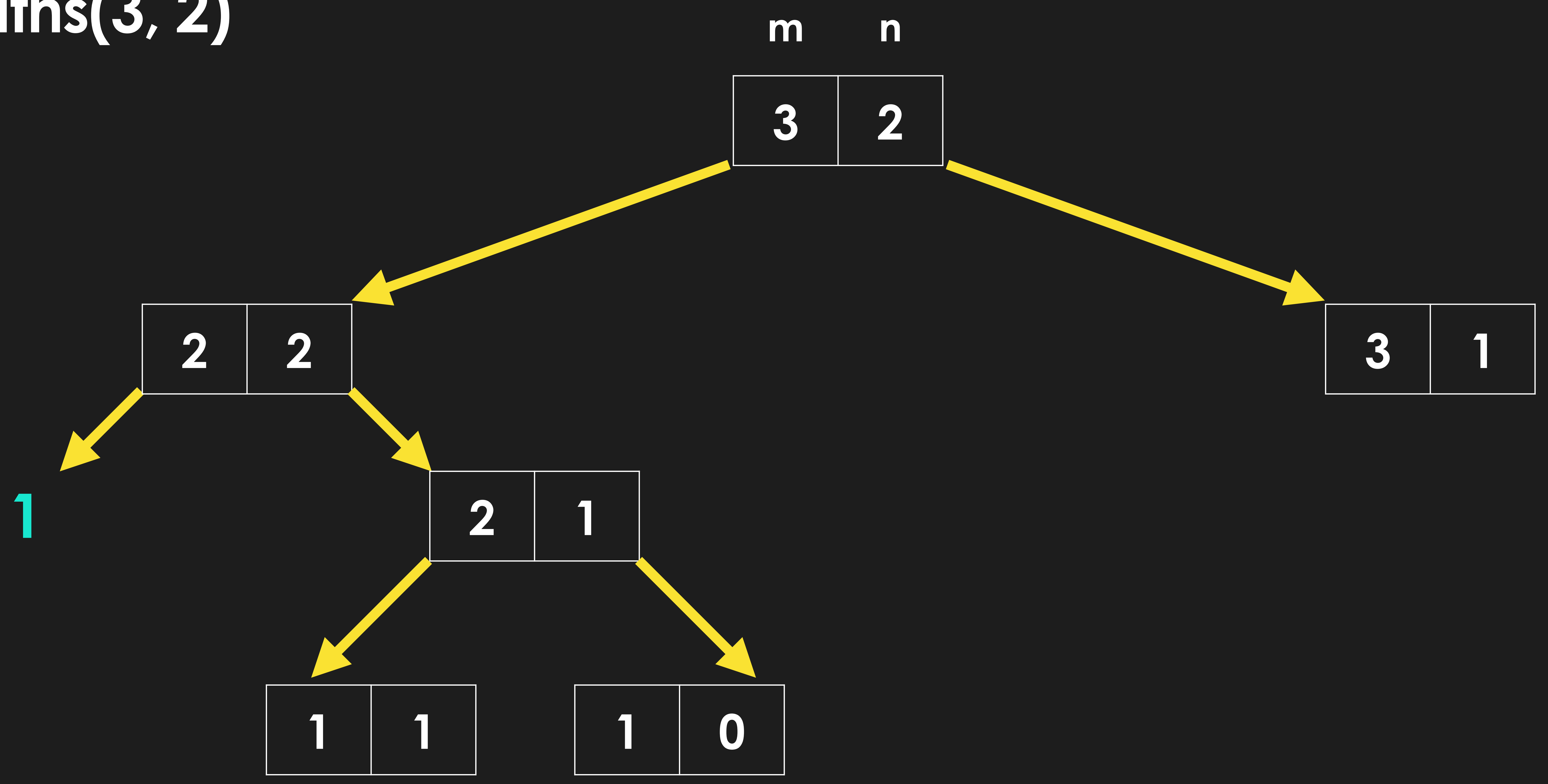
Base cases

UniquePaths(3, 2)



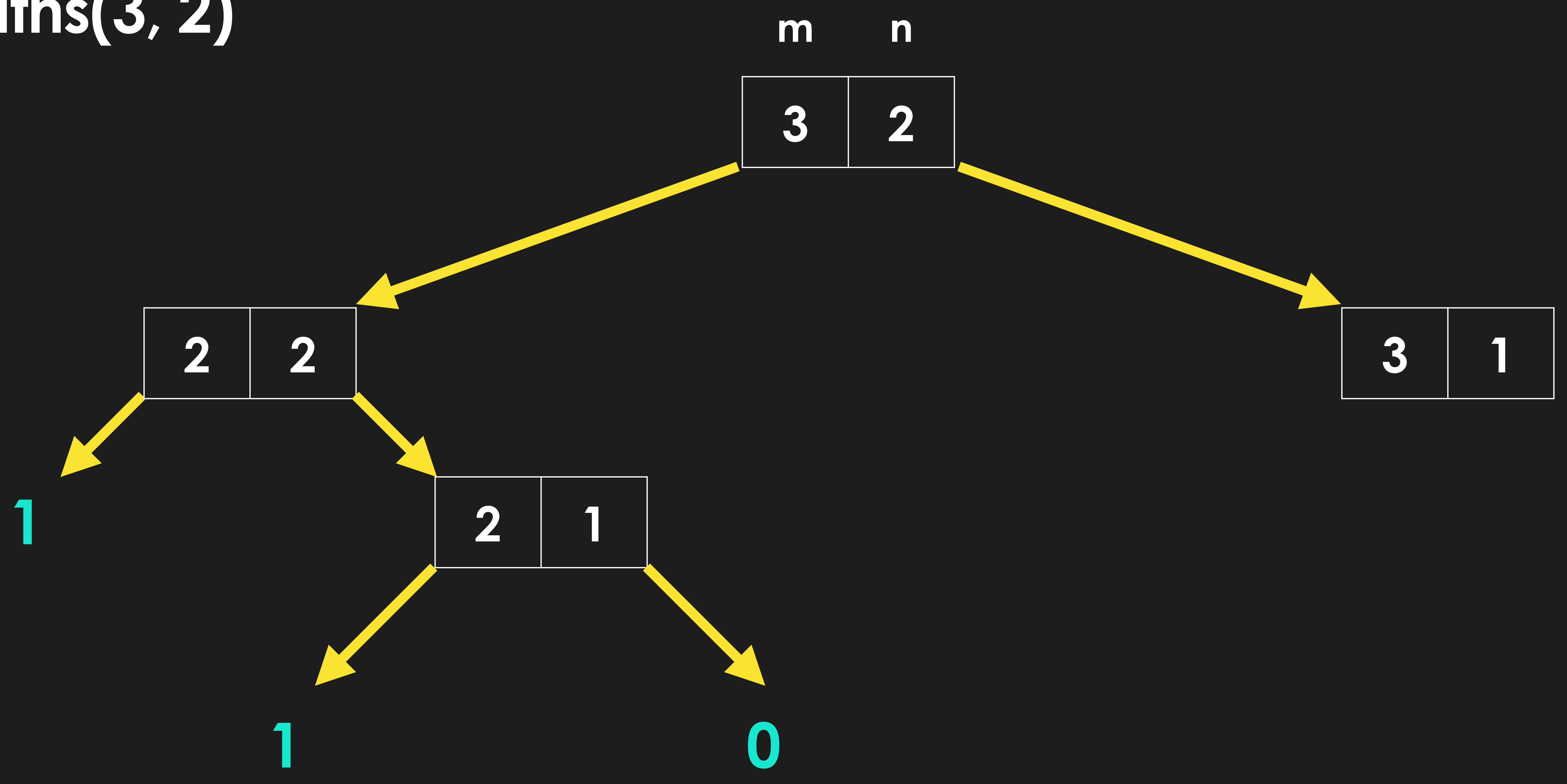
Base cases

UniquePaths(3, 2)



Base cases

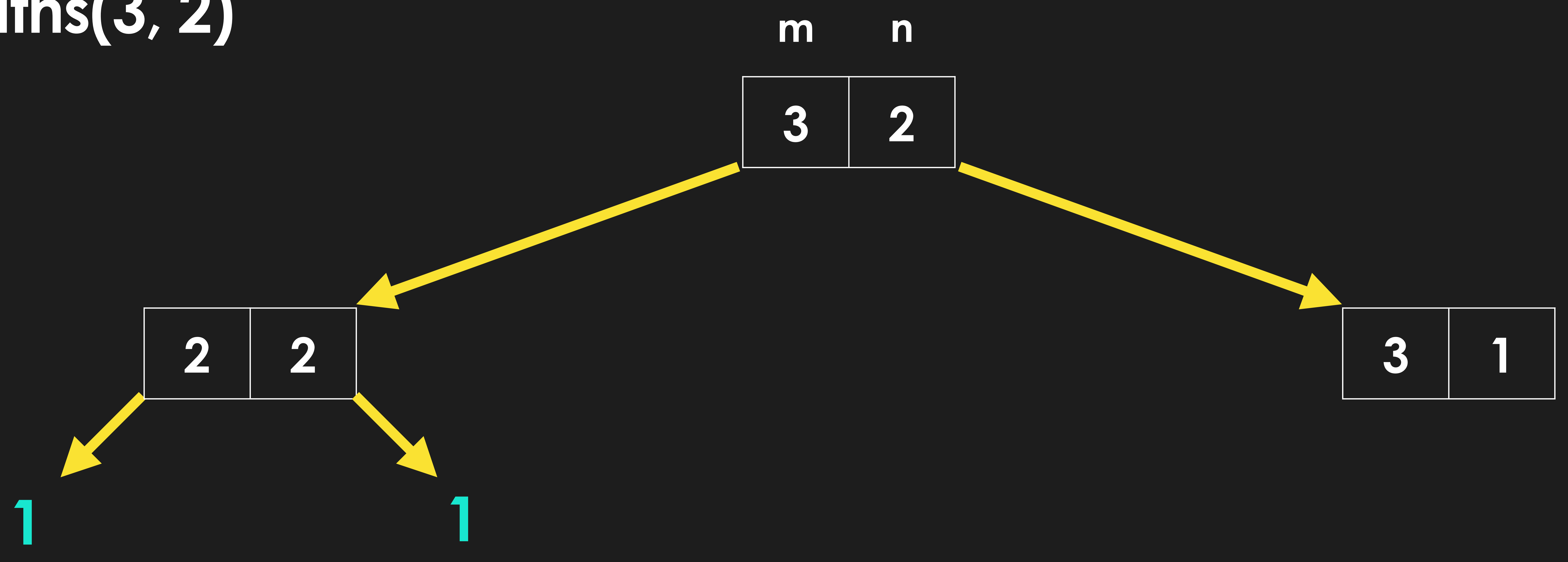
UniquePaths(3, 2)



Base cases

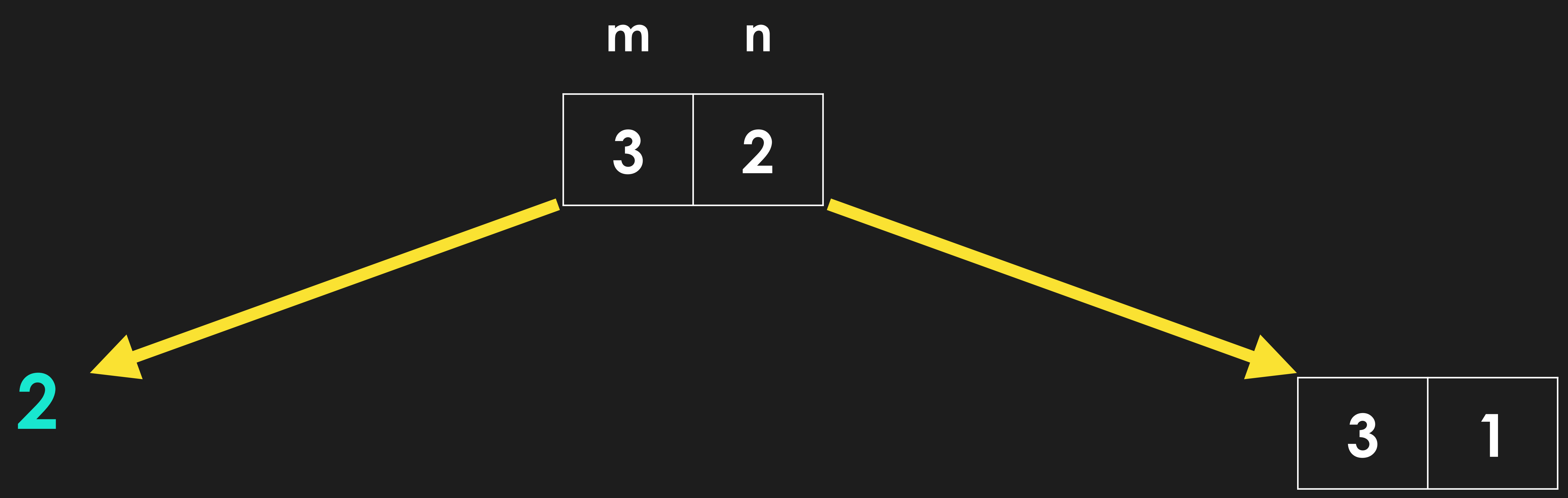
CodeIT

UniquePaths(3, 2)



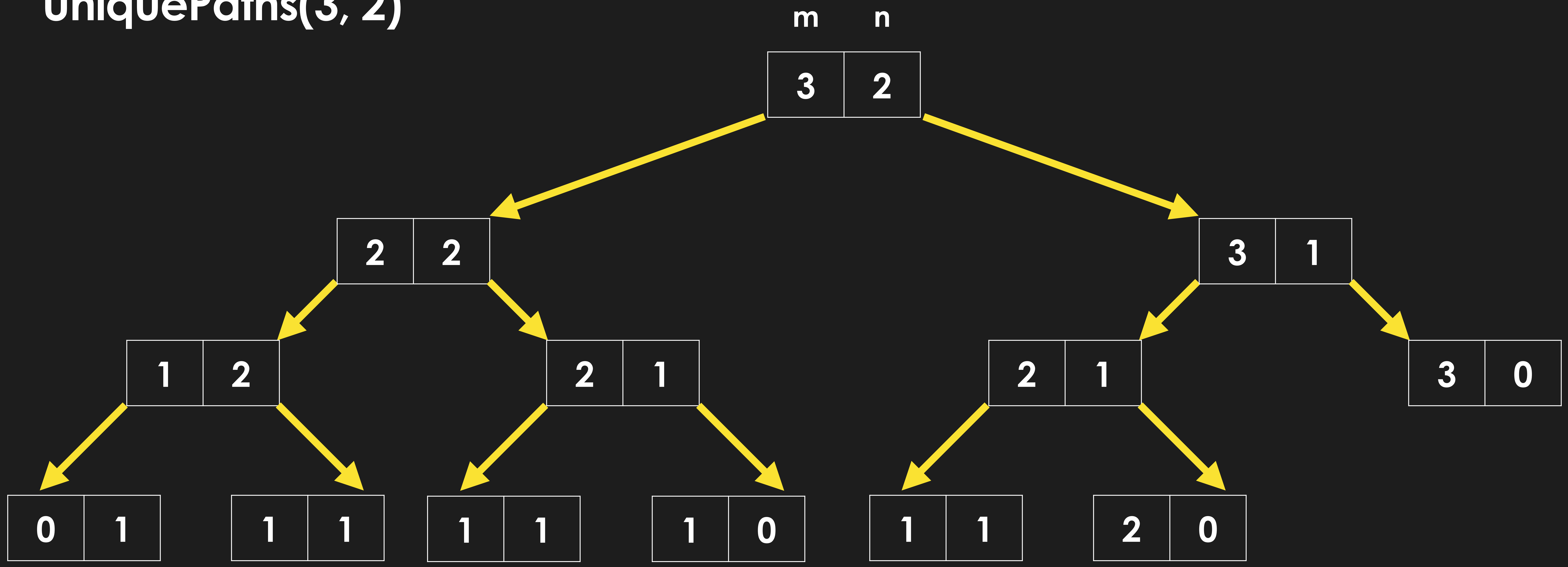
Base cases

UniquePaths(3, 2)



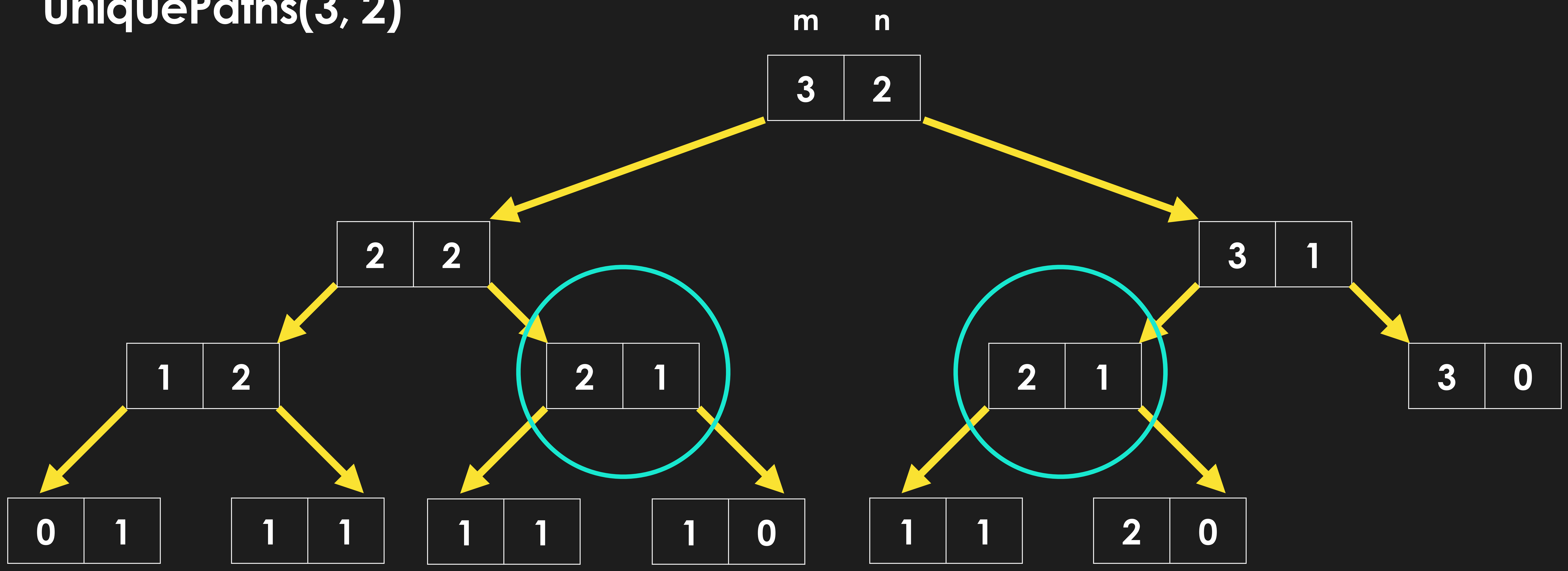
Base cases

UniquePaths(3, 2)



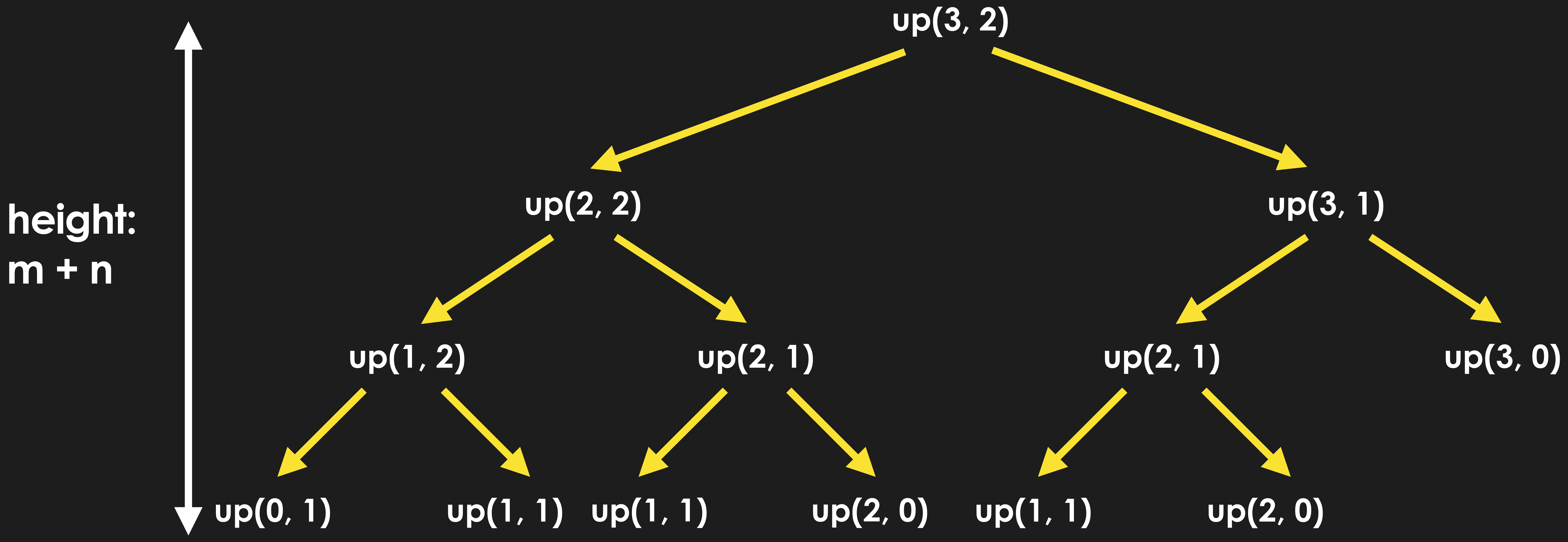
CodeIT

UniquePaths(3, 2)

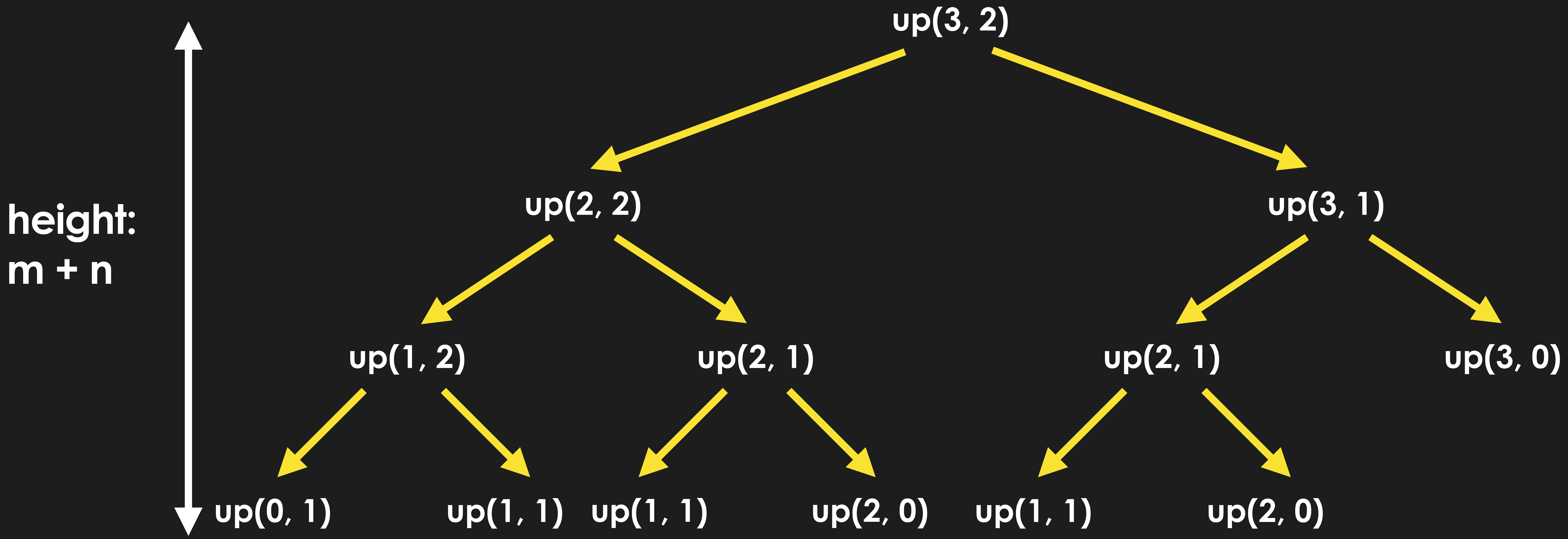


Overlapping subproblems

UniquePaths(3, 2)



UniquePaths(3, 2)



Time Complexity: $2^{m + n}$

Space Complexity: $m + n$

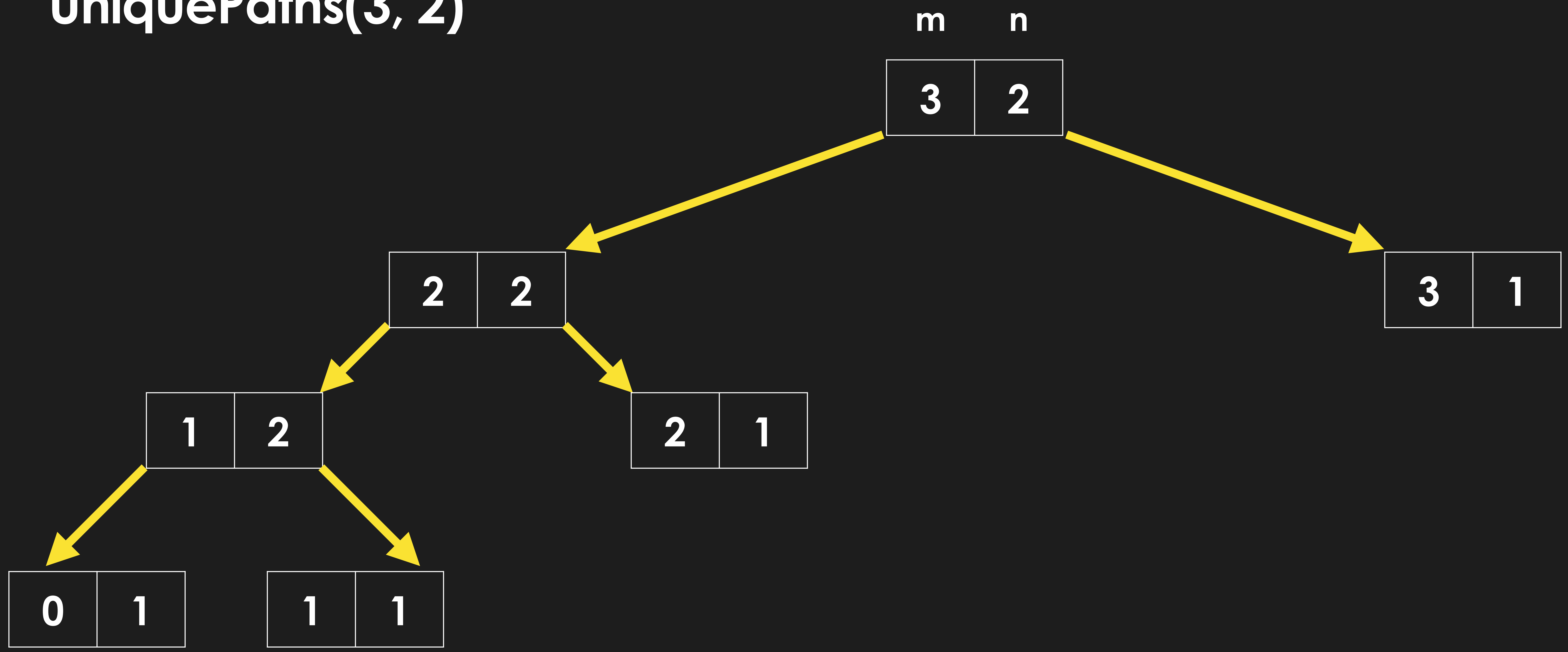
Unique Paths

```
def uniquePaths(m, n):
    if m == 0 or n == 0:
        return 0
    elif m == 1 and n == 1:
        return 1
    else:
        return uniquePaths(m - 1, n) + uniquePaths(m, n - 1)
```

Memoization

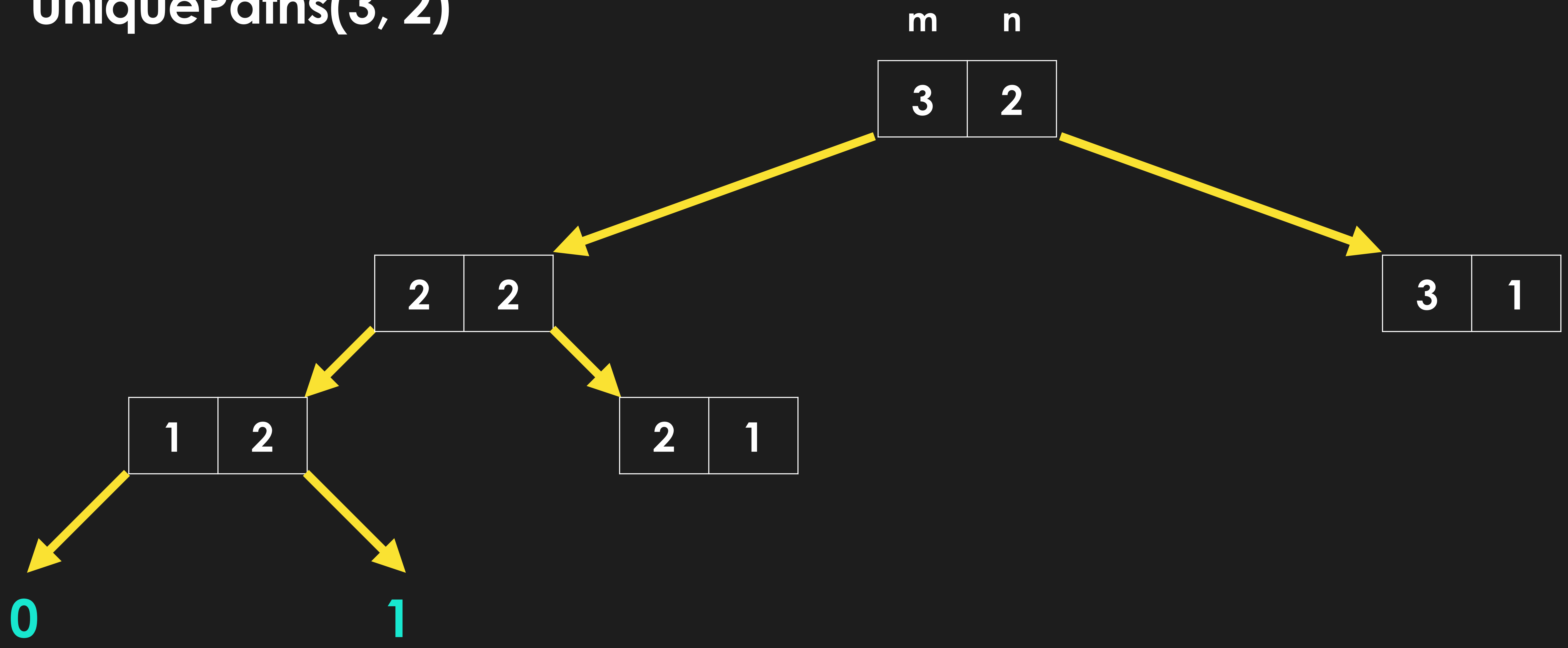
CodeIT

UniquePaths(3, 2)



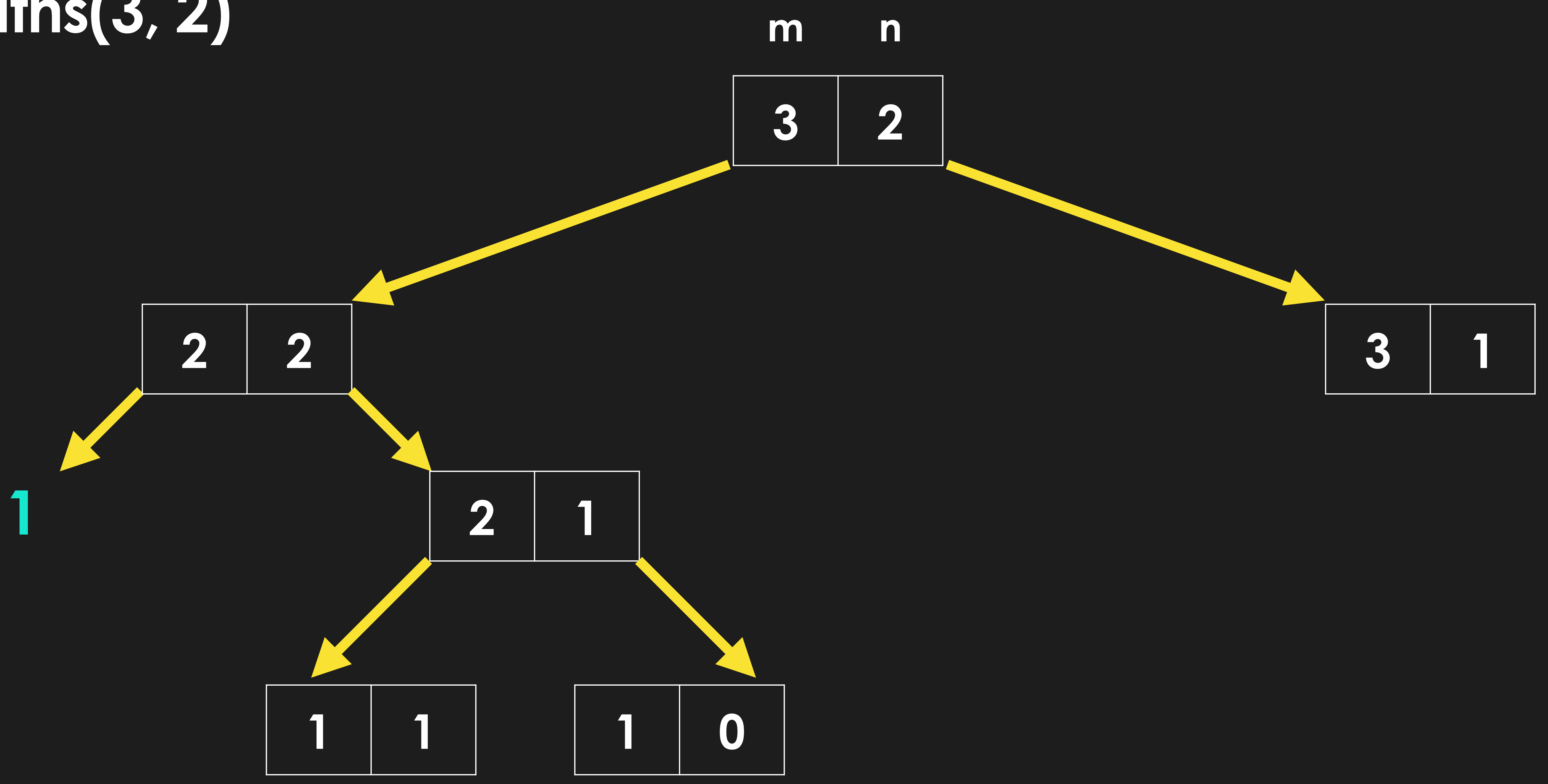
(m, n)					
memo					

UniquePaths(3, 2)



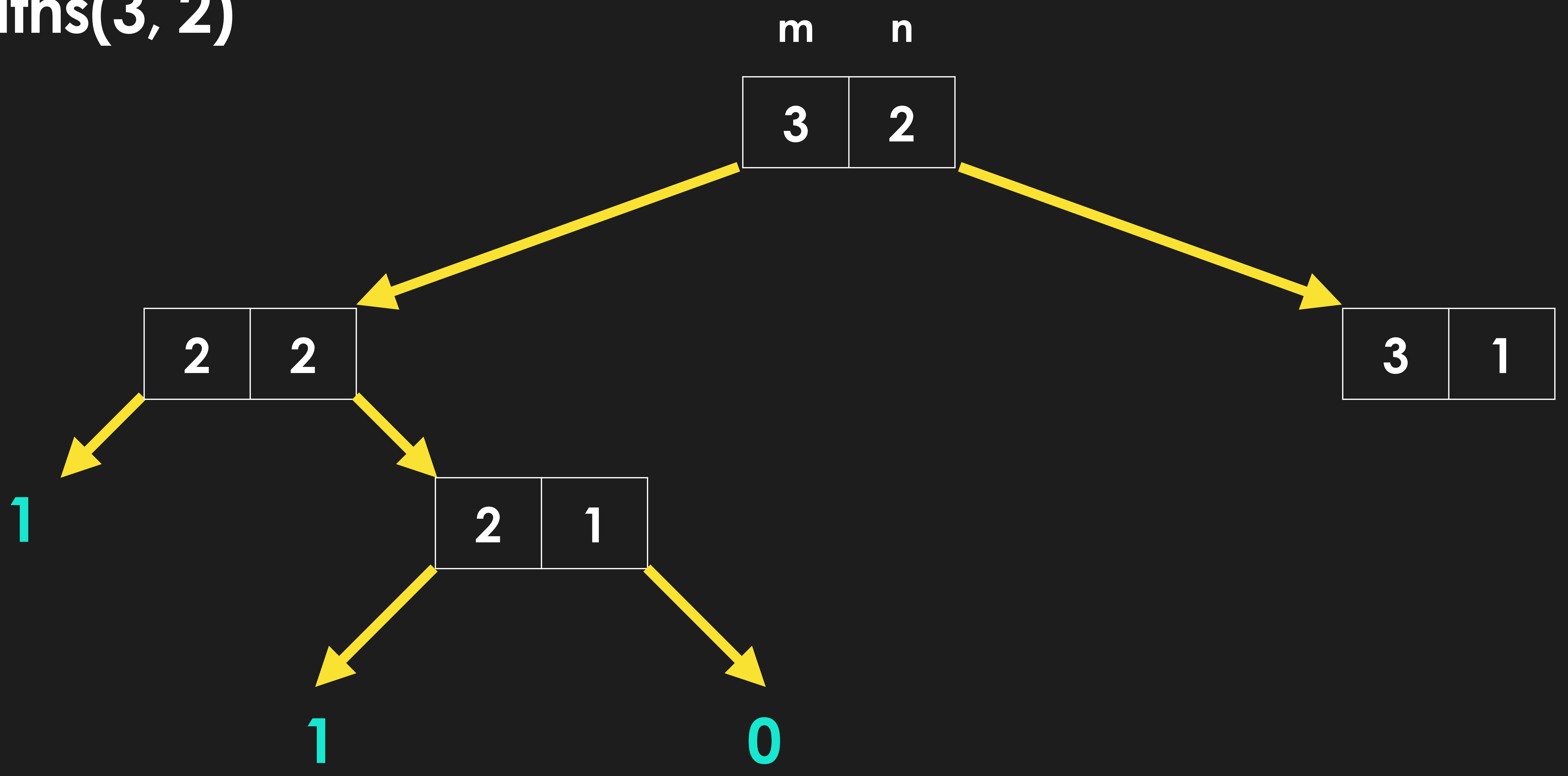
(m, n)					
memo					

UniquePaths(3, 2)



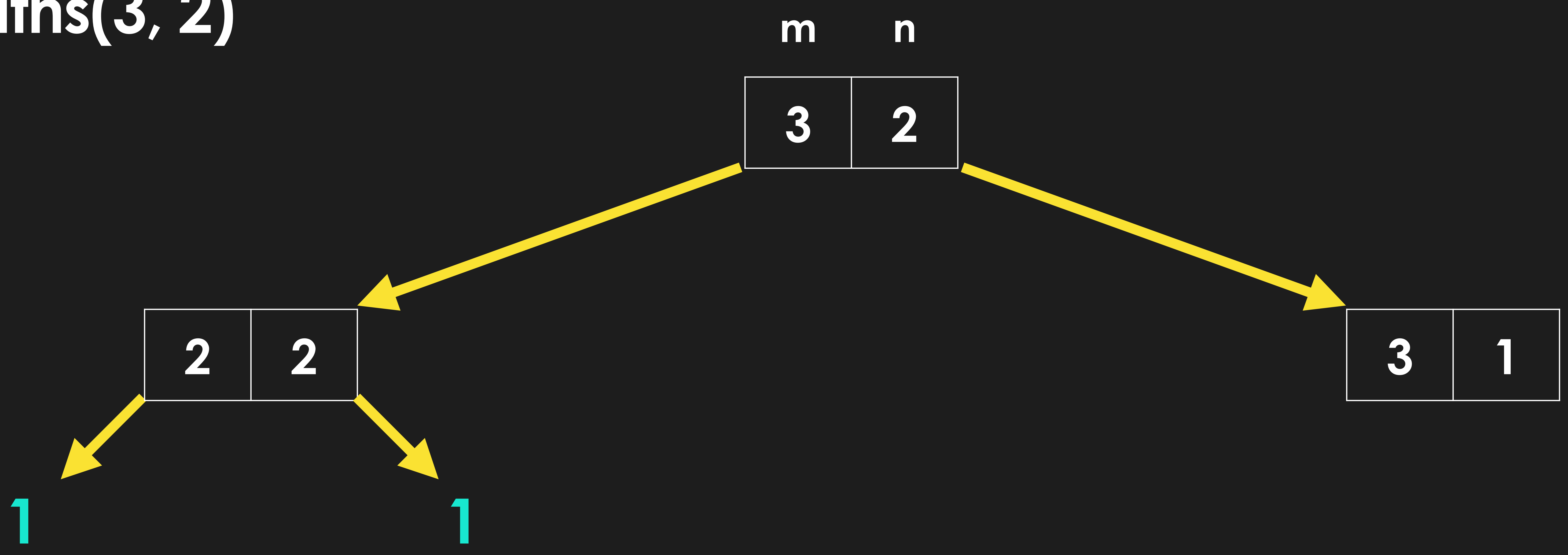
(m, n)	(1, 2)				
memo	1				

UniquePaths(3, 2)



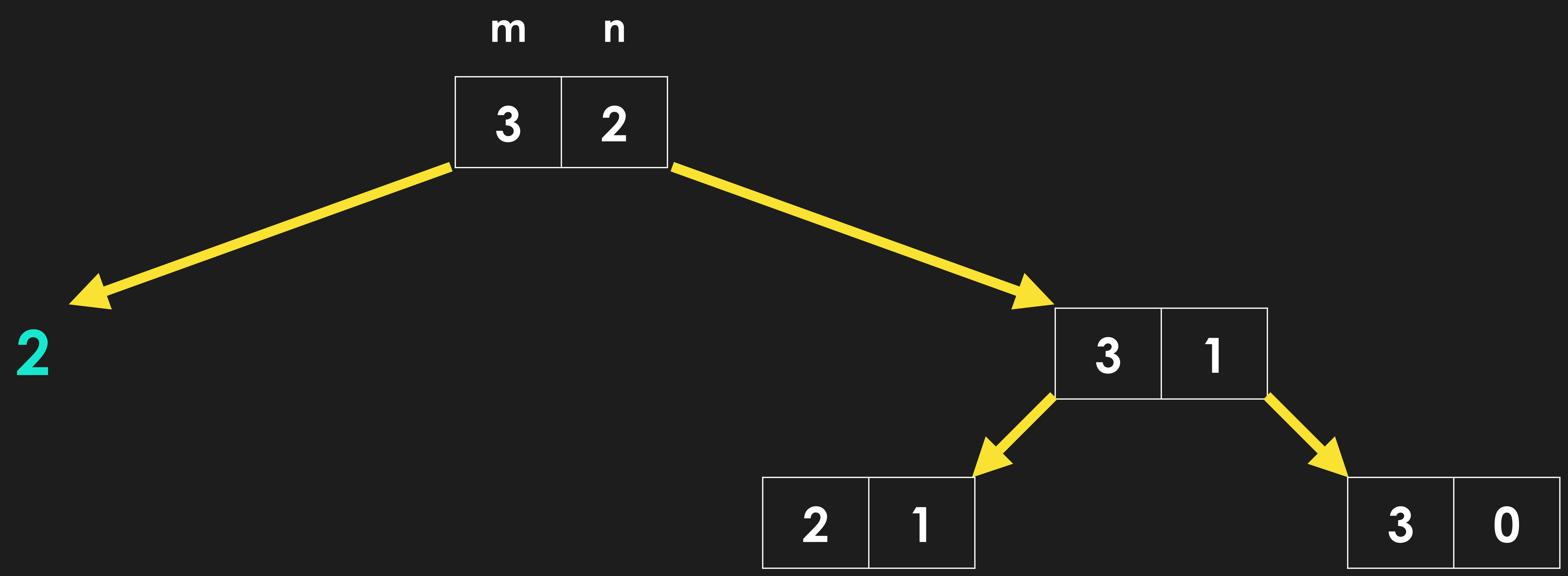
(m, n)	(1, 2)				
memo	1				

UniquePaths(3, 2)



(m, n)	(1, 2)	(2, 1)			
memo	1	1			

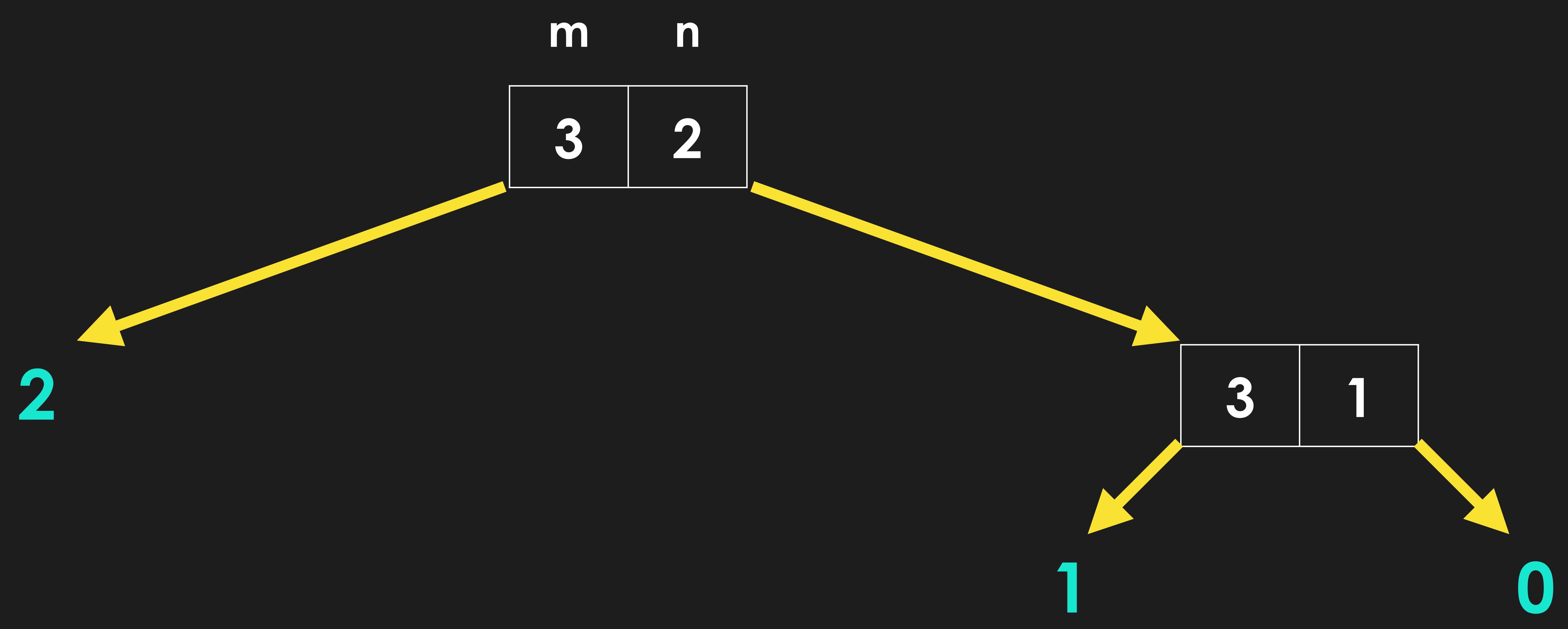
UniquePaths(3, 2)



(m, n)	(1, 2)	(2, 1)	(2, 2)		
memo	1	1	2		

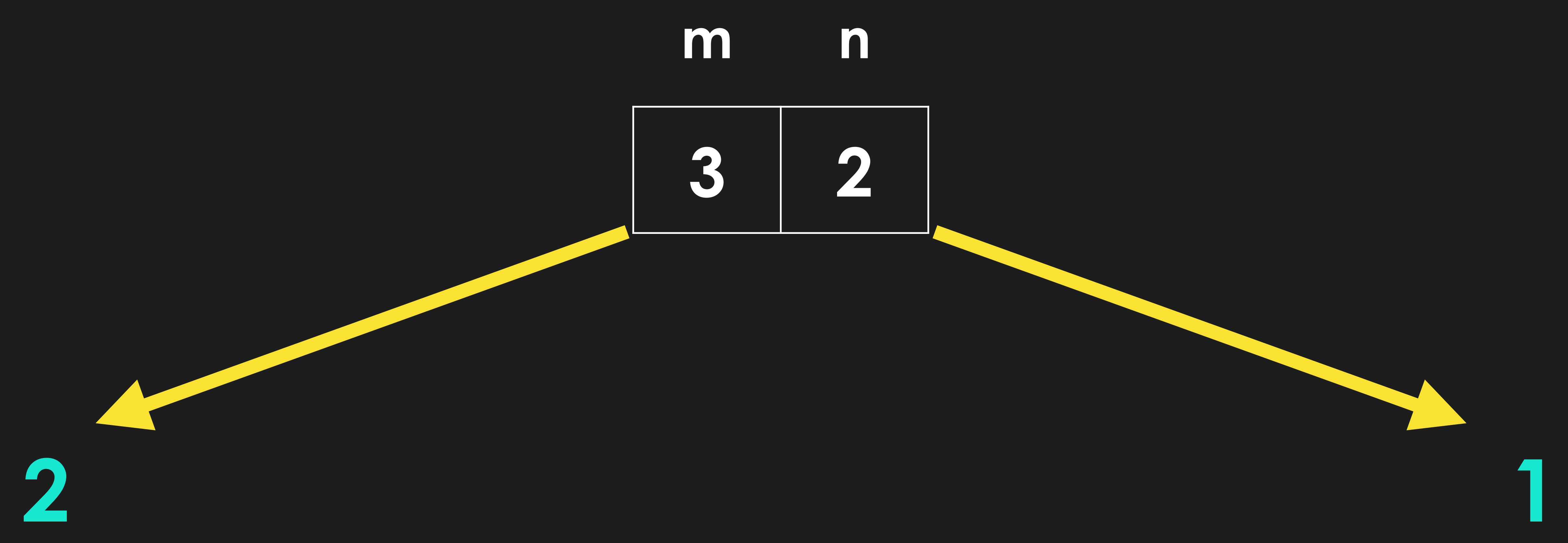
CodeIT

UniquePaths(3, 2)



(m, n)	(1, 2)	(2, 1)	(2, 2)		
memo	1	1	2		

UniquePaths(3, 2)



(m, n)	(1, 2)	(2, 1)	(2, 2)	(3, 1)	
memo	1	1	2	1	

UniquePaths(3, 2)

m n

3

(m, n)	(1, 2)	(2, 1)	(2, 2)	(3, 1)	(3, 2)
memo	1	1	2	1	3

CodeIT

Unique Paths Memo

```
memo = {}

def uniquePathsMemo(m, n):
    if m == 0 or n == 0:
        return 0
    elif m == 1 and n == 1:
        return 1
    elif (m, n) in memo:
        return memo[(m, n)]

    else:
        cost = uniquePathsMemo(m - 1, n) + uniquePathsMemo(m, n - 1)
        memo[(m, n)] = cost
        memo[(n, m)] = cost
        return cost
```

DP Problem #3: Can Sum

Can Sum

Given an array of integers, and a target sum, print out whether there exists a combination of the array of numbers, with repetition, that adds to the target sum

Can Sum

Given an array of integers, and a target sum, print out whether there exists a combination of the array of numbers, with repetition, that adds to the target sum

Example:

array: [3, 6, 7]
target: 10

canSum: True
combinations: [5, 5]

Can Sum

Given an array of integers, and a target sum, print out whether there exists a combination of the array of numbers, with repetition, that adds to the target sum

Example:

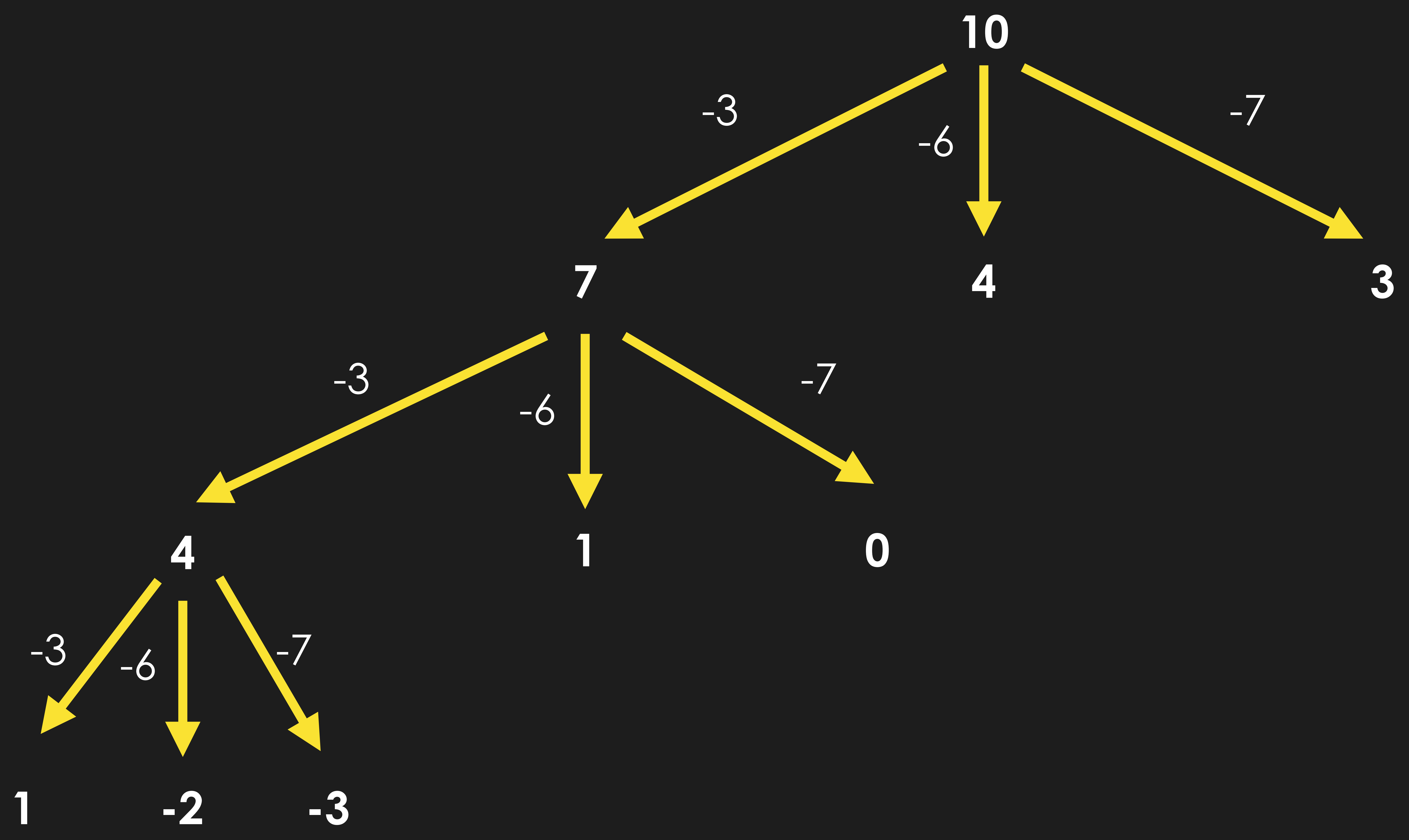
array: [3, 6, 7]
target: 10

canSum: True
combinations: [5, 5]

Optimal Substructure

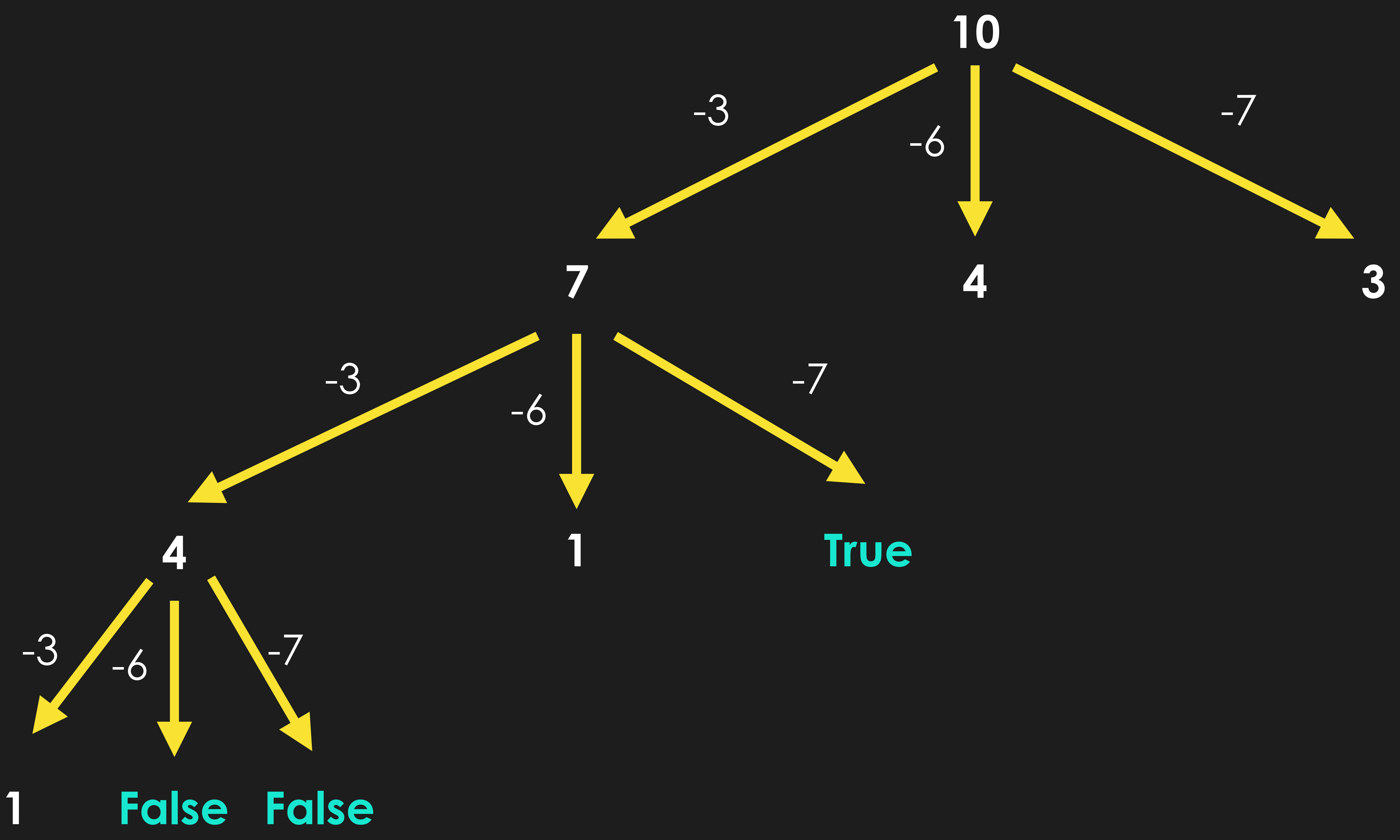
If **canSum(array, target - array[i])** is **True** where i is a valid index in the array, then **canSum(array, target)** is **True**

CanSum



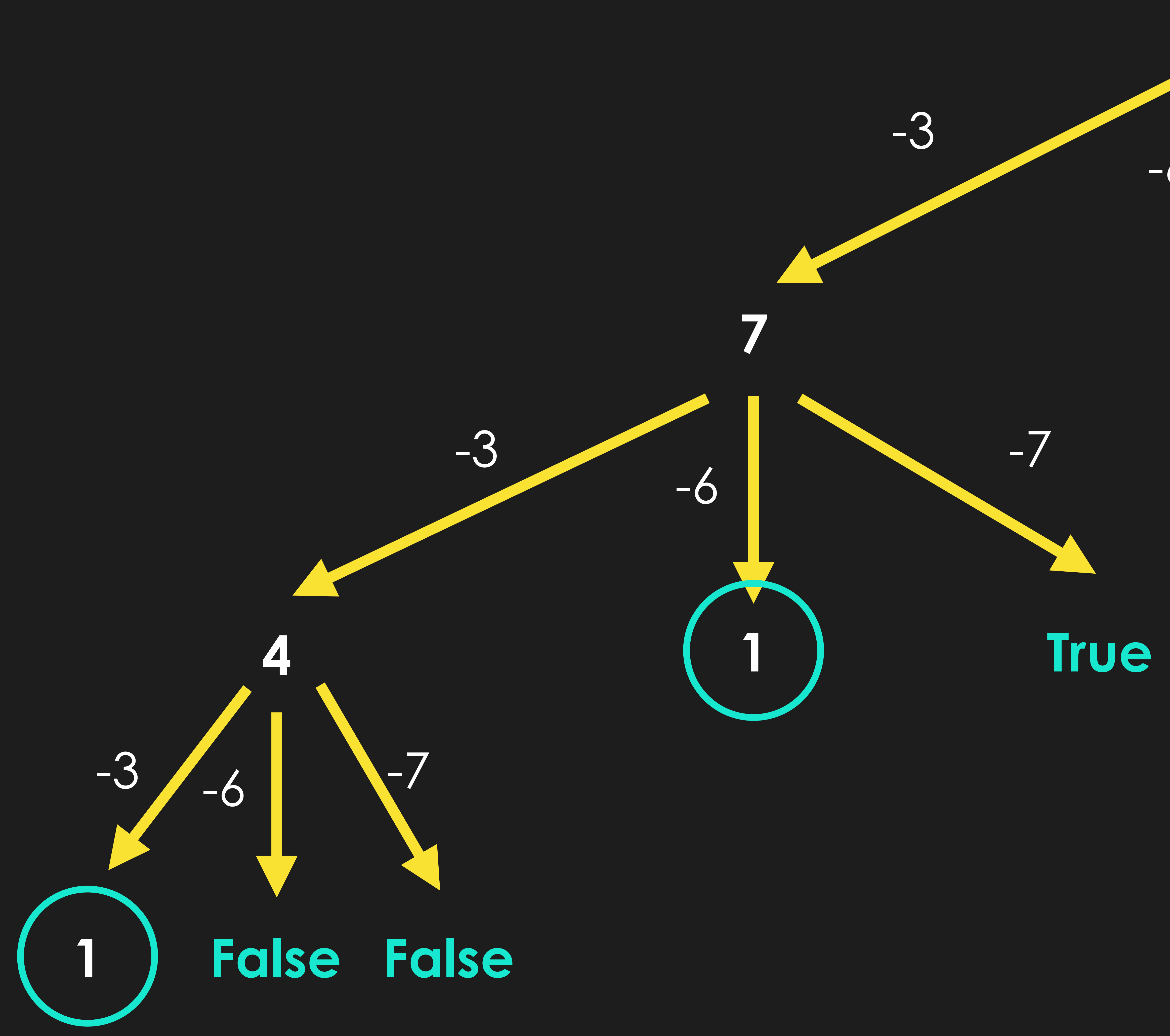
CodeIT

CanSum



Base case

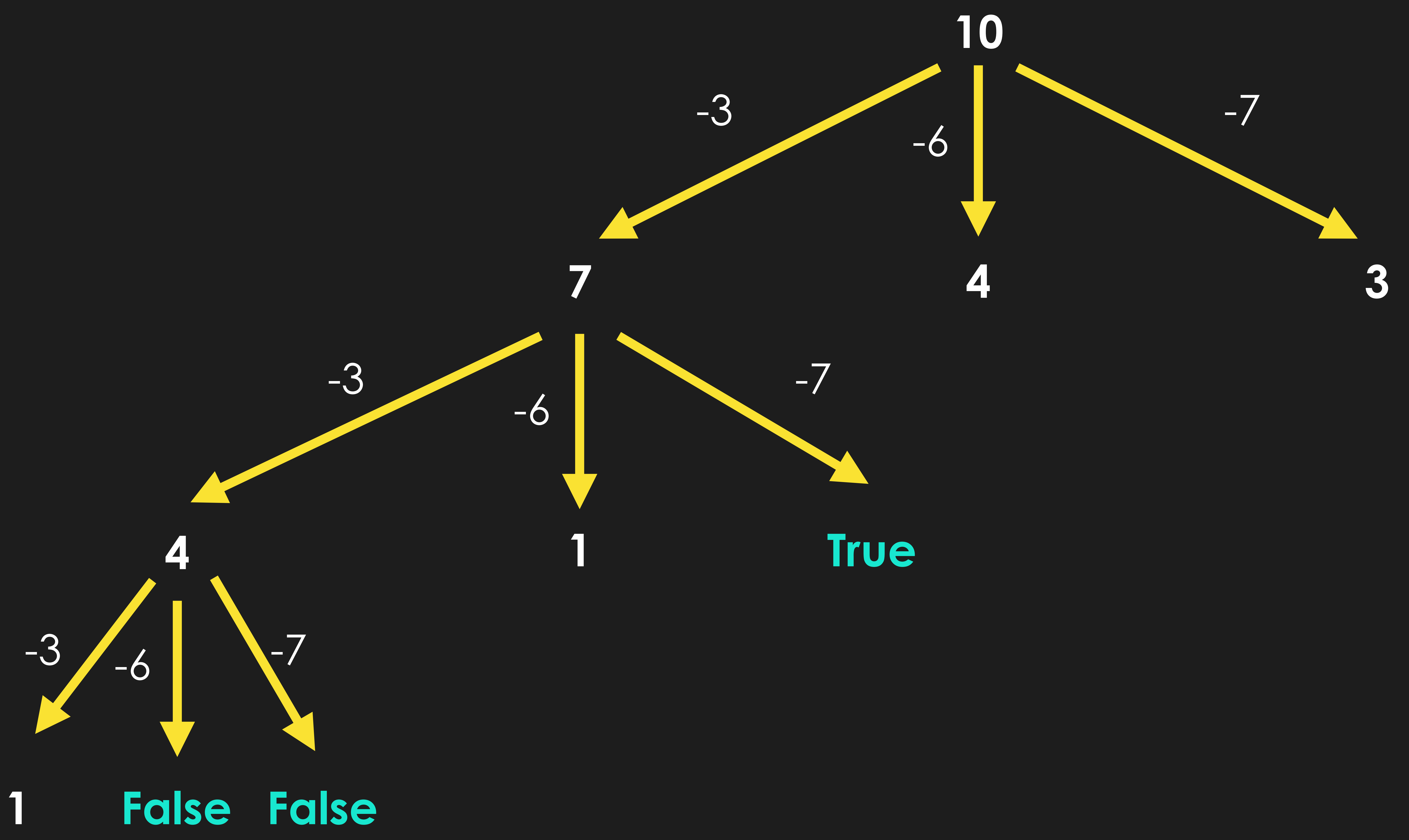
CanSum



Overlapping subproblem!

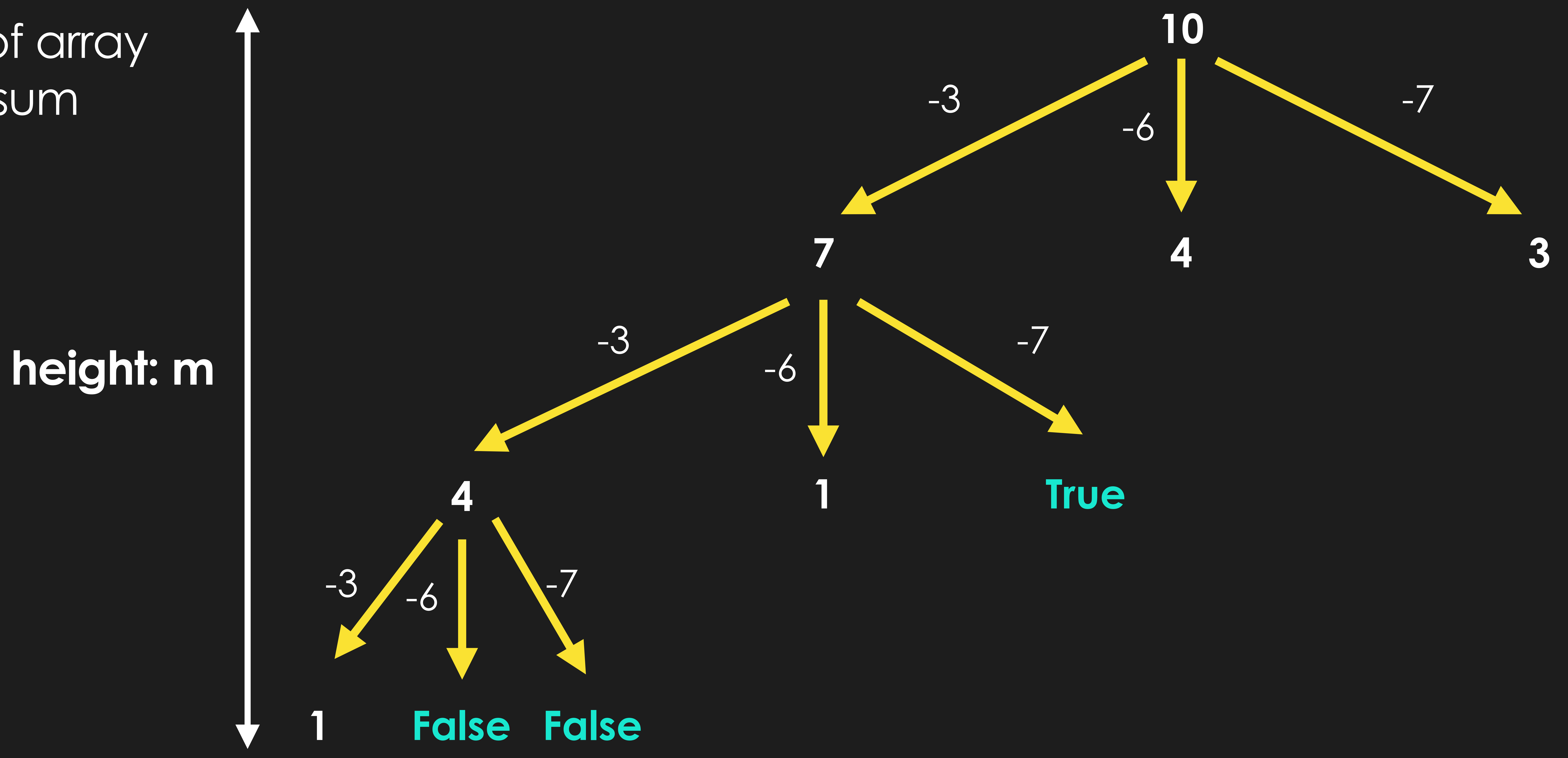
CanSum

n: length of array
m: target sum



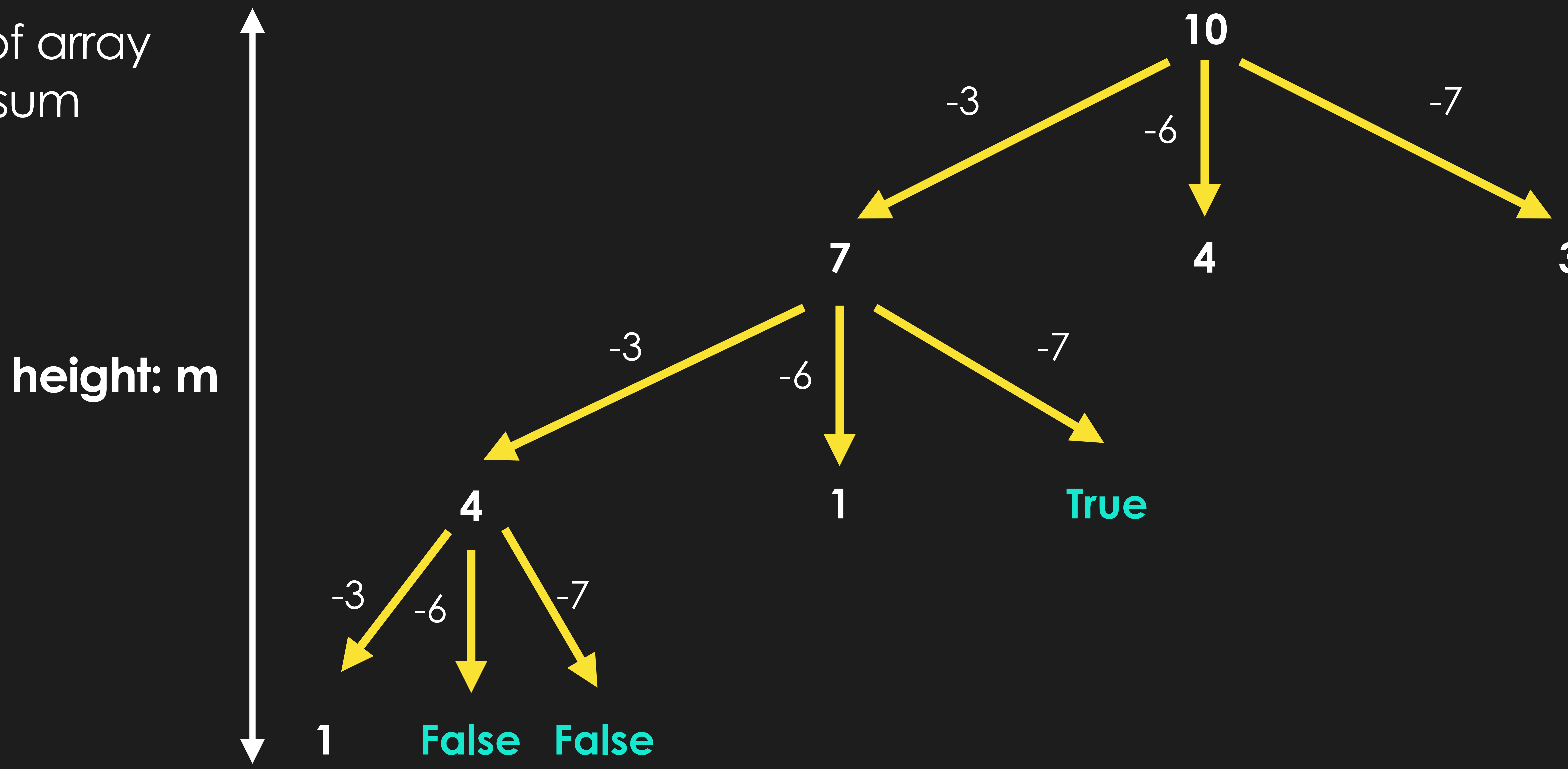
CanSum

n: length of array
m: target sum



CanSum

n: length of array
m: target sum



Time Complexity: n^m

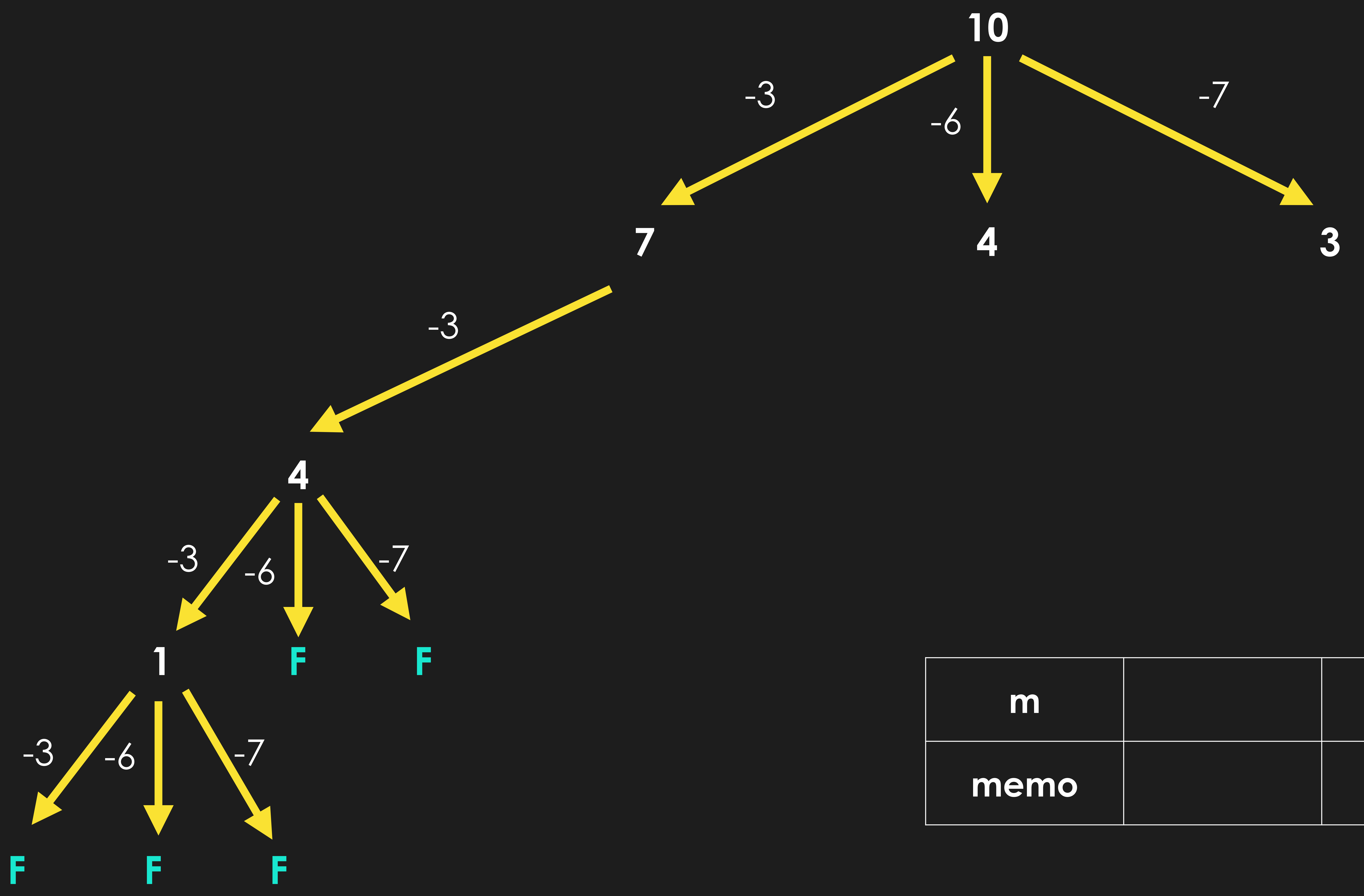
Space Complexity: m

CanSum

```python
def CanSum(array, target):
    if target == 0:
        return 1
    if target < 0:
        return 0

    for i in array:
        newTarget = target - i
        if (CanSum(array, newTarget)):
            return True

    return False
```

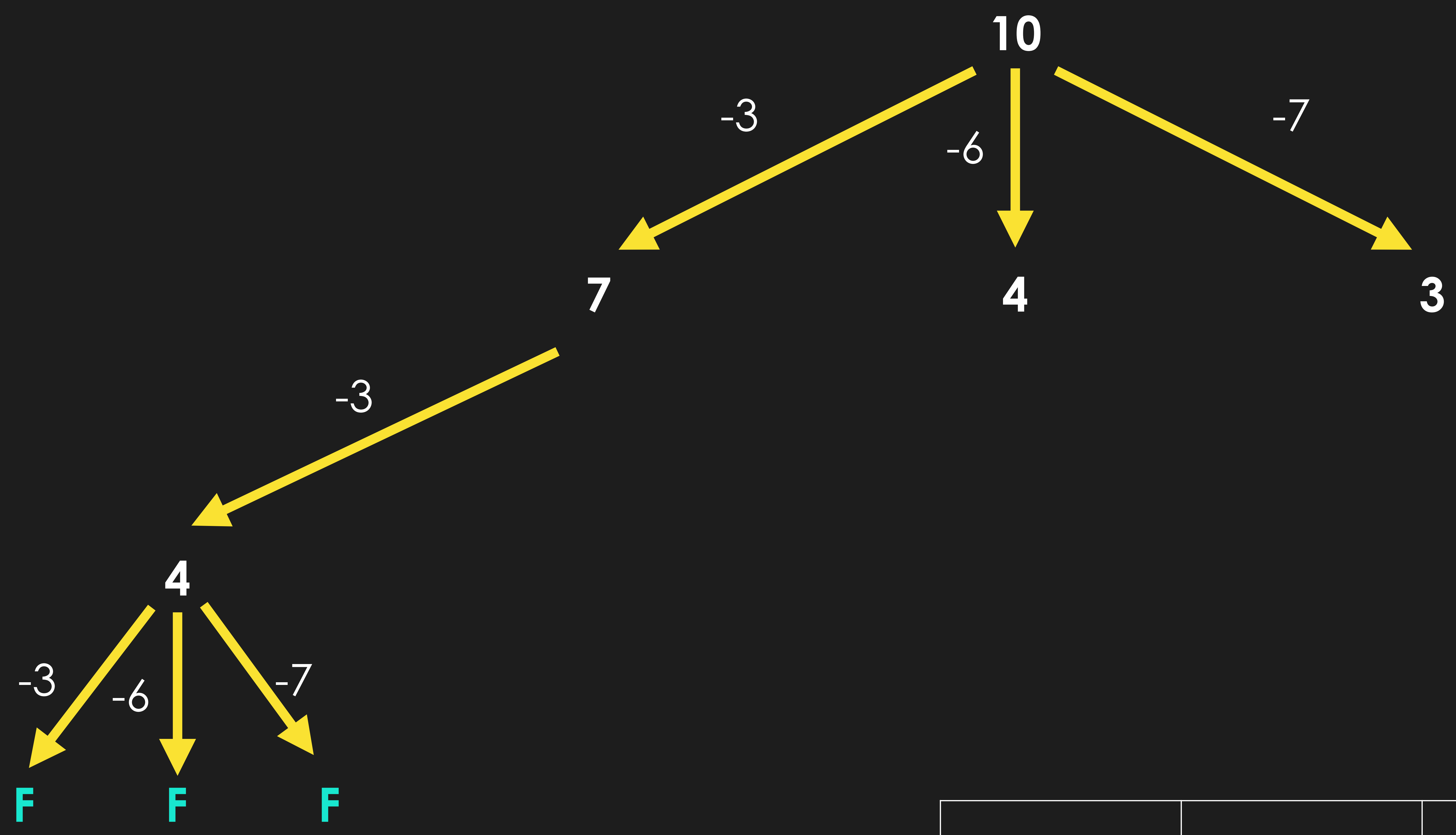
Memoization!

CanSumMemo



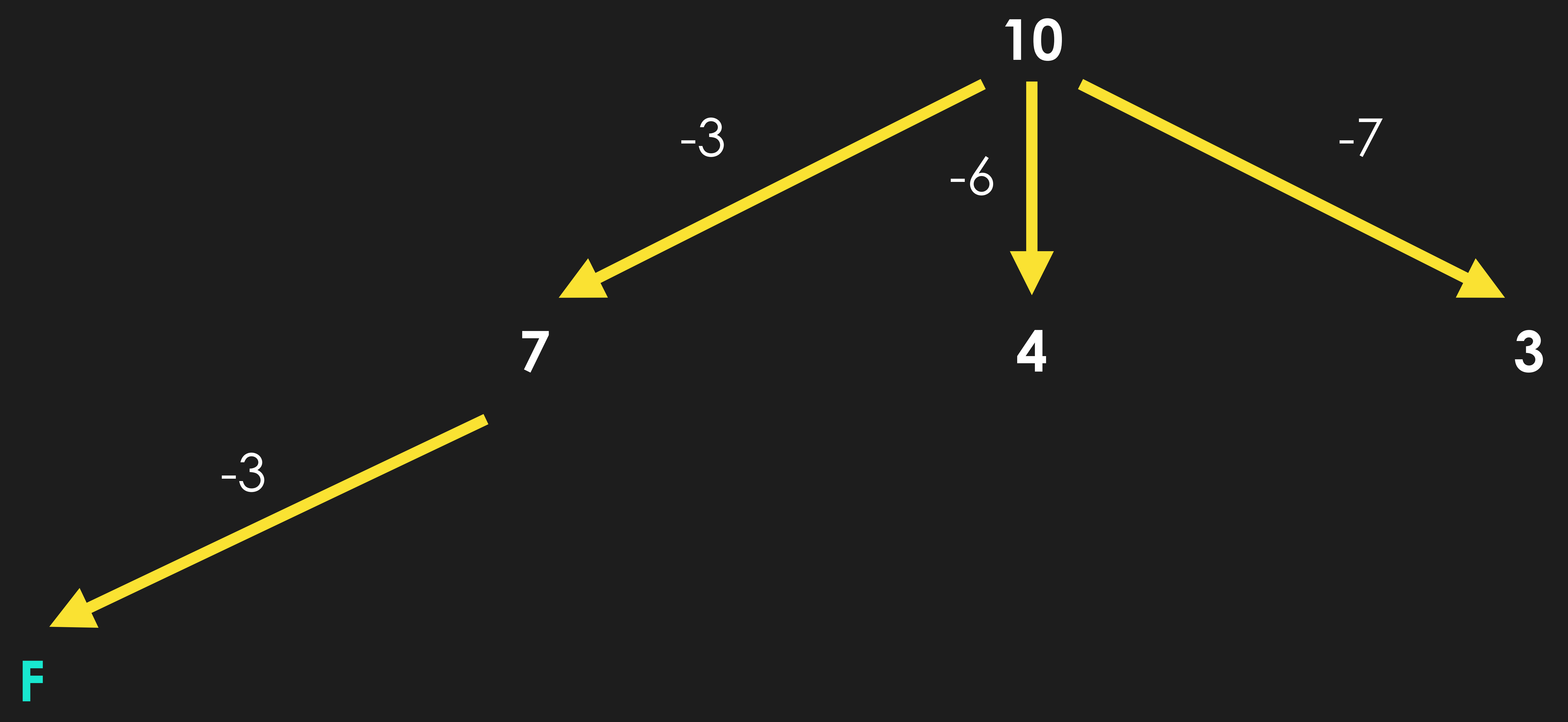
m			
memo			

CanSumMemo



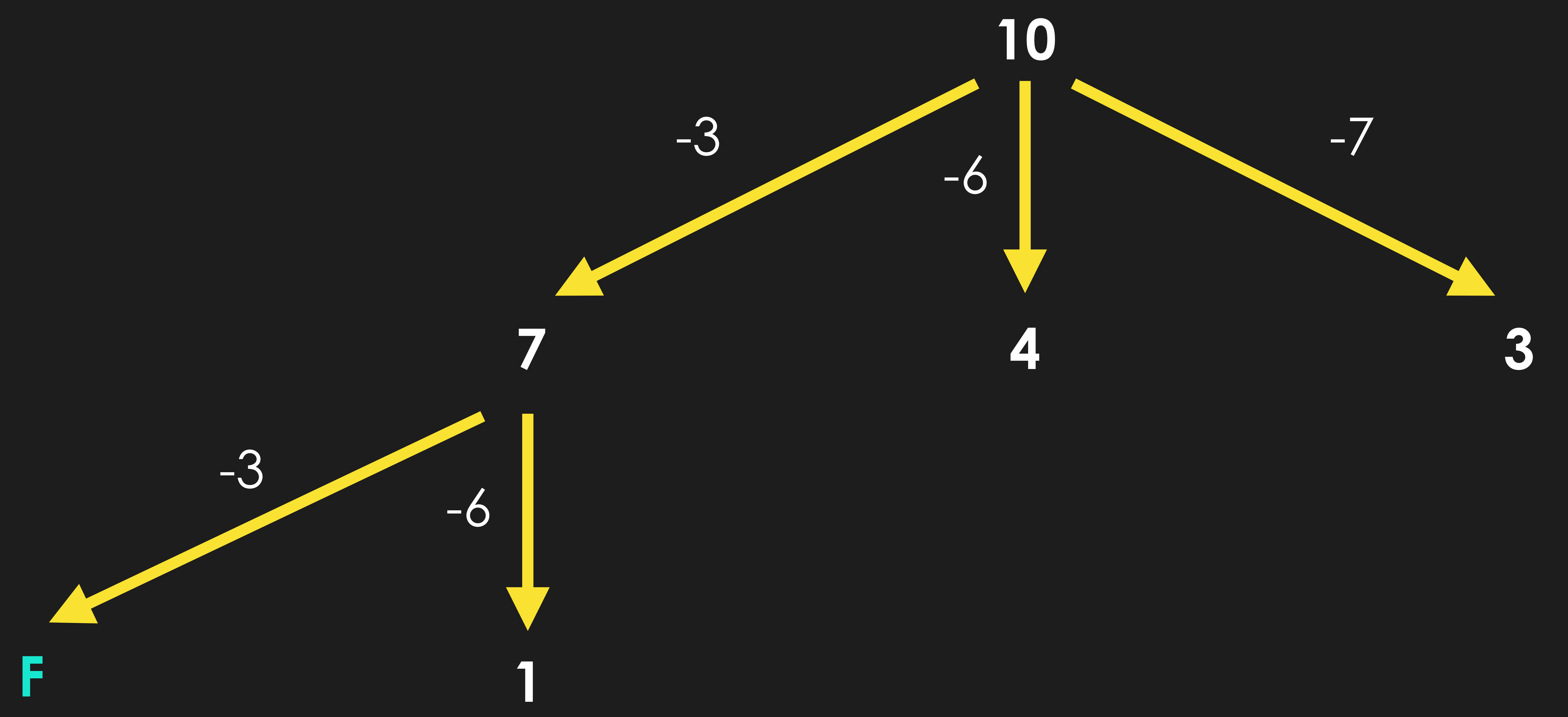
m	1		
memo	FALSE		

CanSumMemo



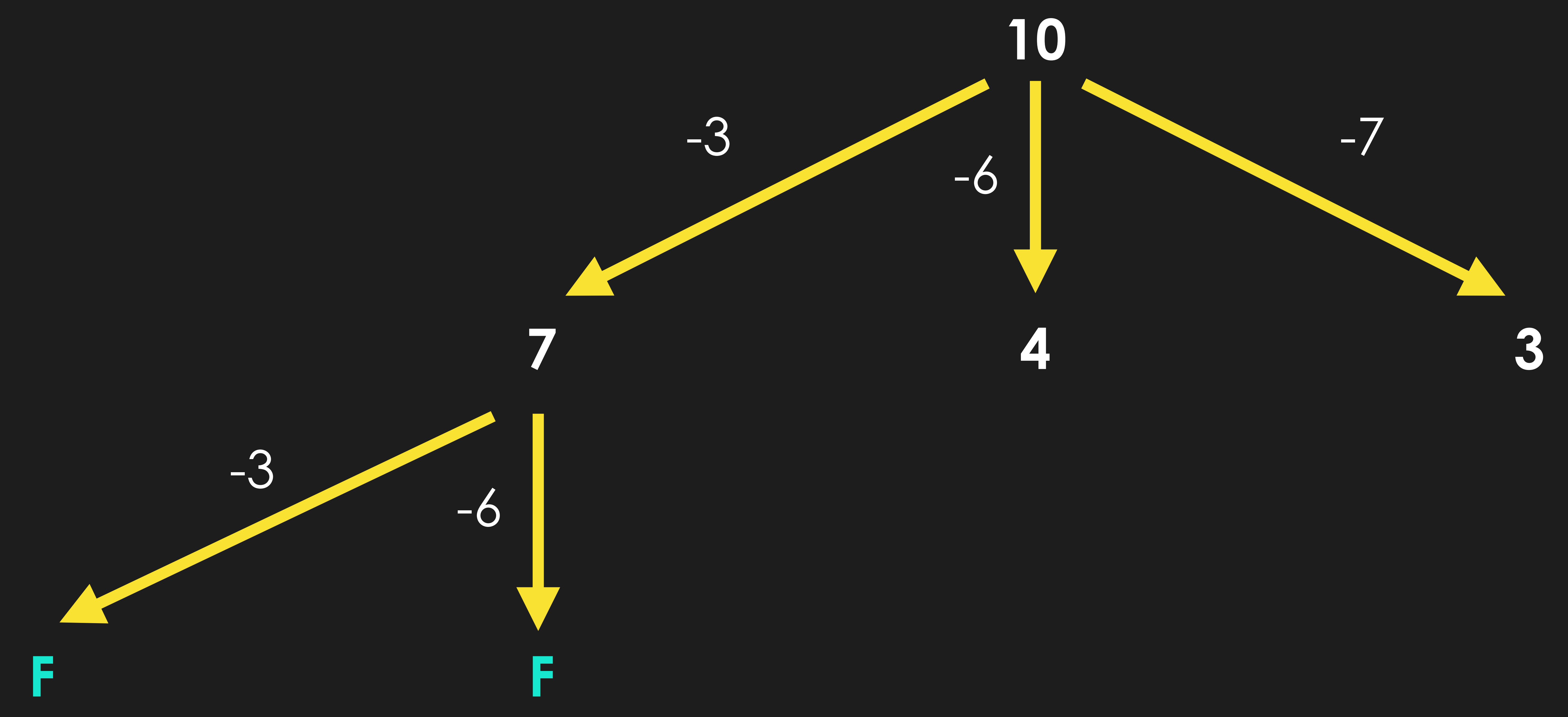
m	1	4	
memo	FALSE	FALSE	

CanSumMemo



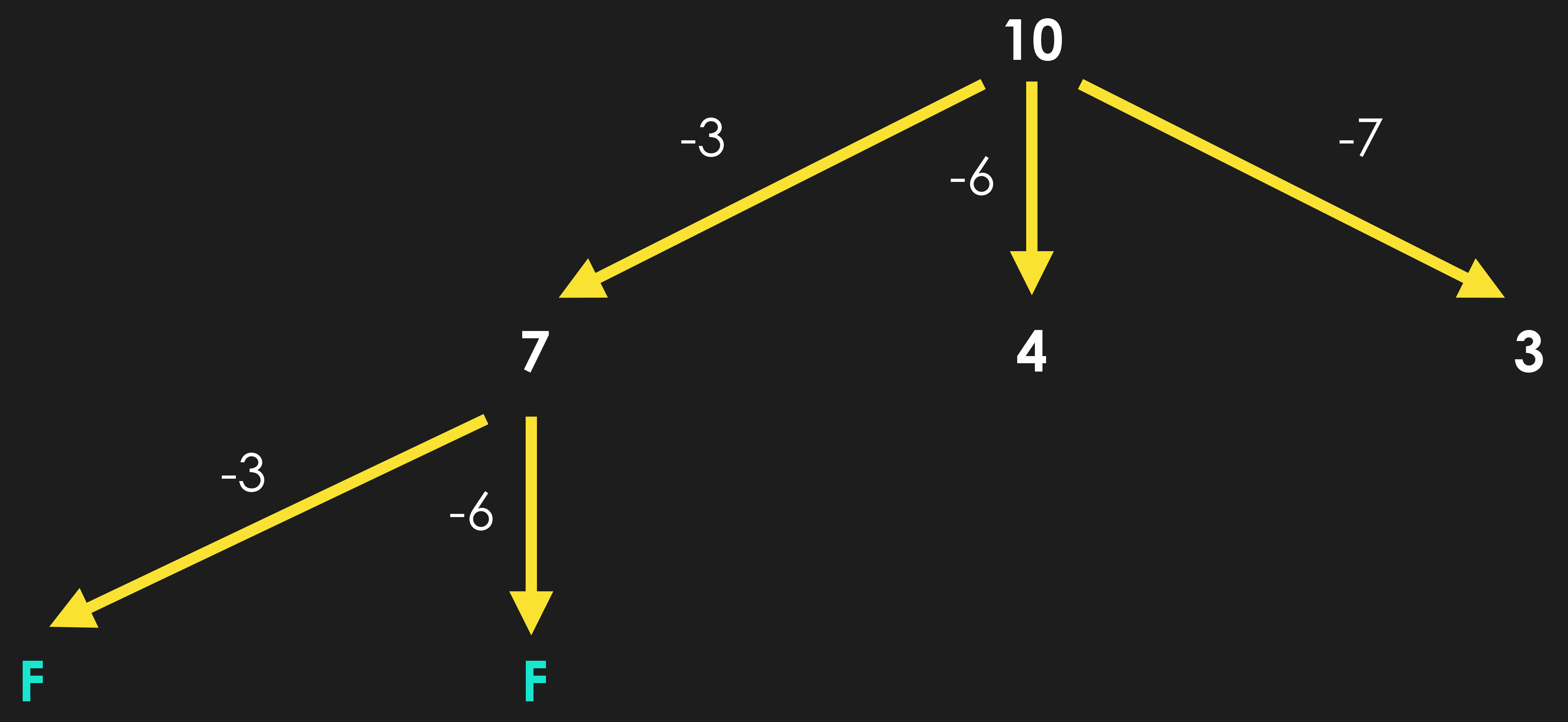
m	1	4	
memo	FALSE	FALSE	

CanSumMemo



m	1	4	
memo	FALSE	FALSE	

CanSumMemo



m	1	4	
memo	FALSE	FALSE	

Time Complexity: m * n

Space Complexity: m

CanSumMemo

```python
def CanSum(array, target):
    if target == 0:
        return 1
    if target < 0:
        return 0

    for i in array:
        newTarget = target - i
        if (CanSum(array, newTarget)):
            return True

    return False
```

DP Problem #4: Best String Construct

DP Problem #4: Best String Construct

Given a target string, and an array of strings, determine whether the array of strings **can be used to construct the target string**, returning the combination with the **least number of strings**

Example:

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

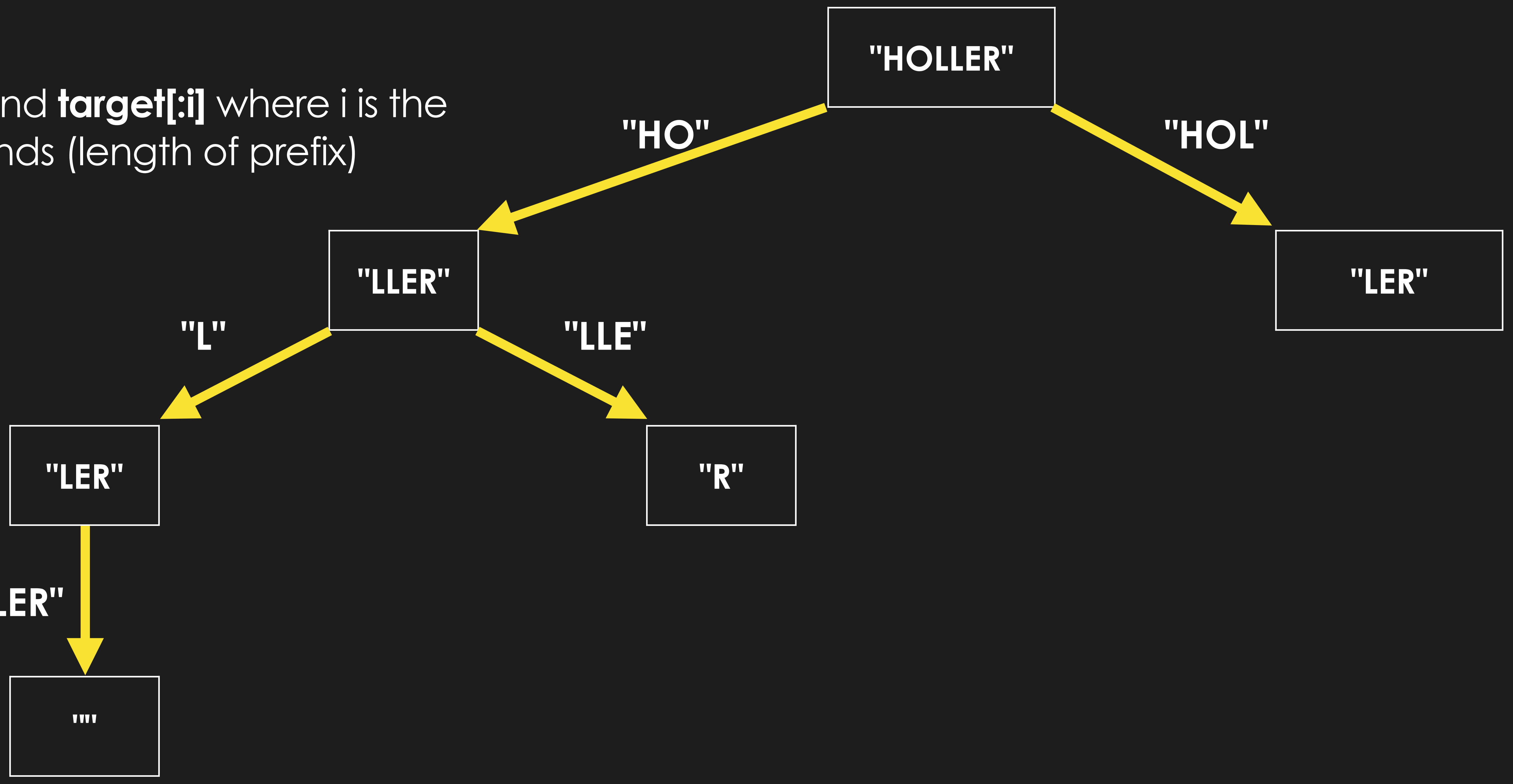
stringConstruct(array, target) => ["HOL", "LER"]

Note: ["HO", "LL", "ER"] is also an answer, but it is not the array with the least elements

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

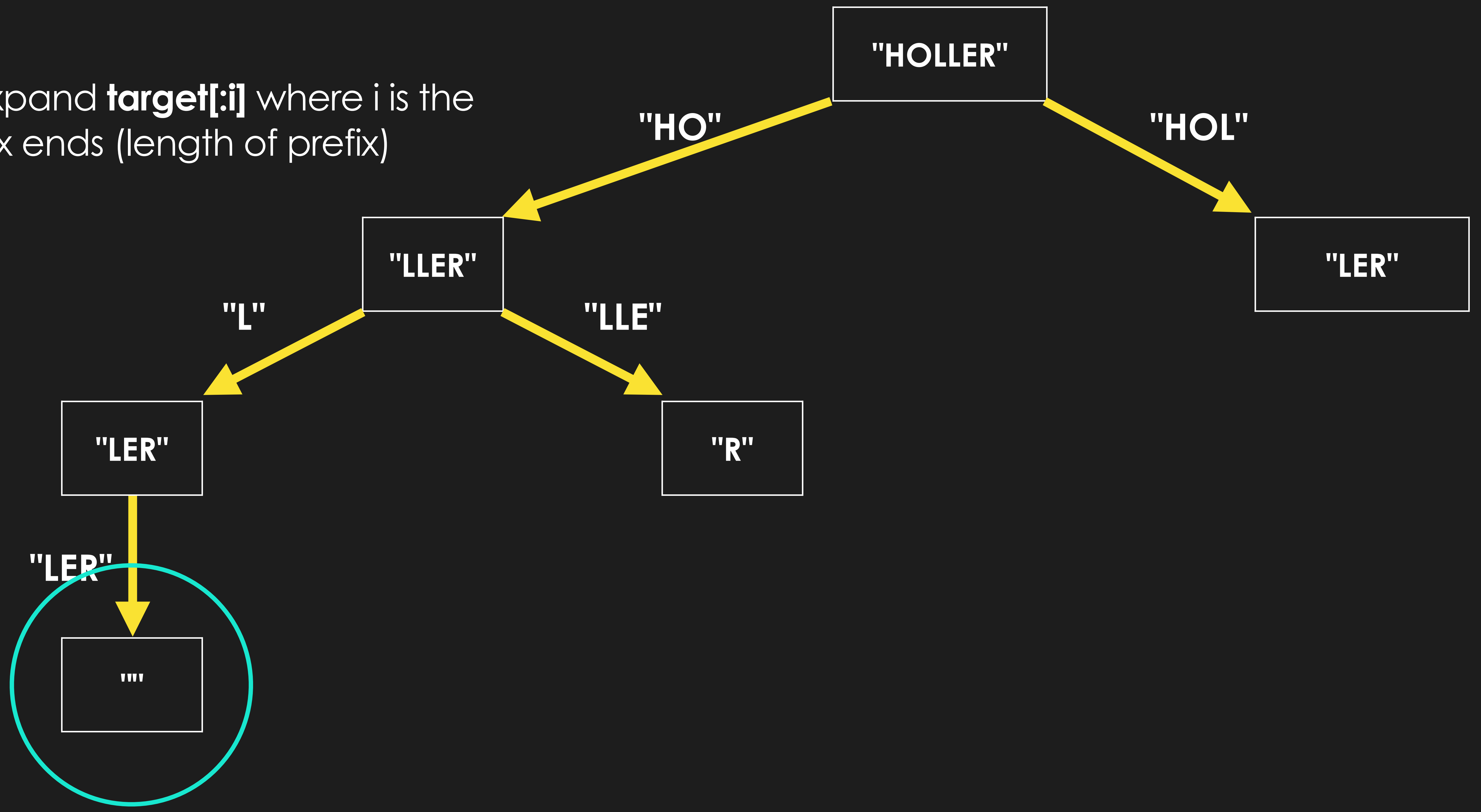


CodeIT

String Construct
Optimal Substructure

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

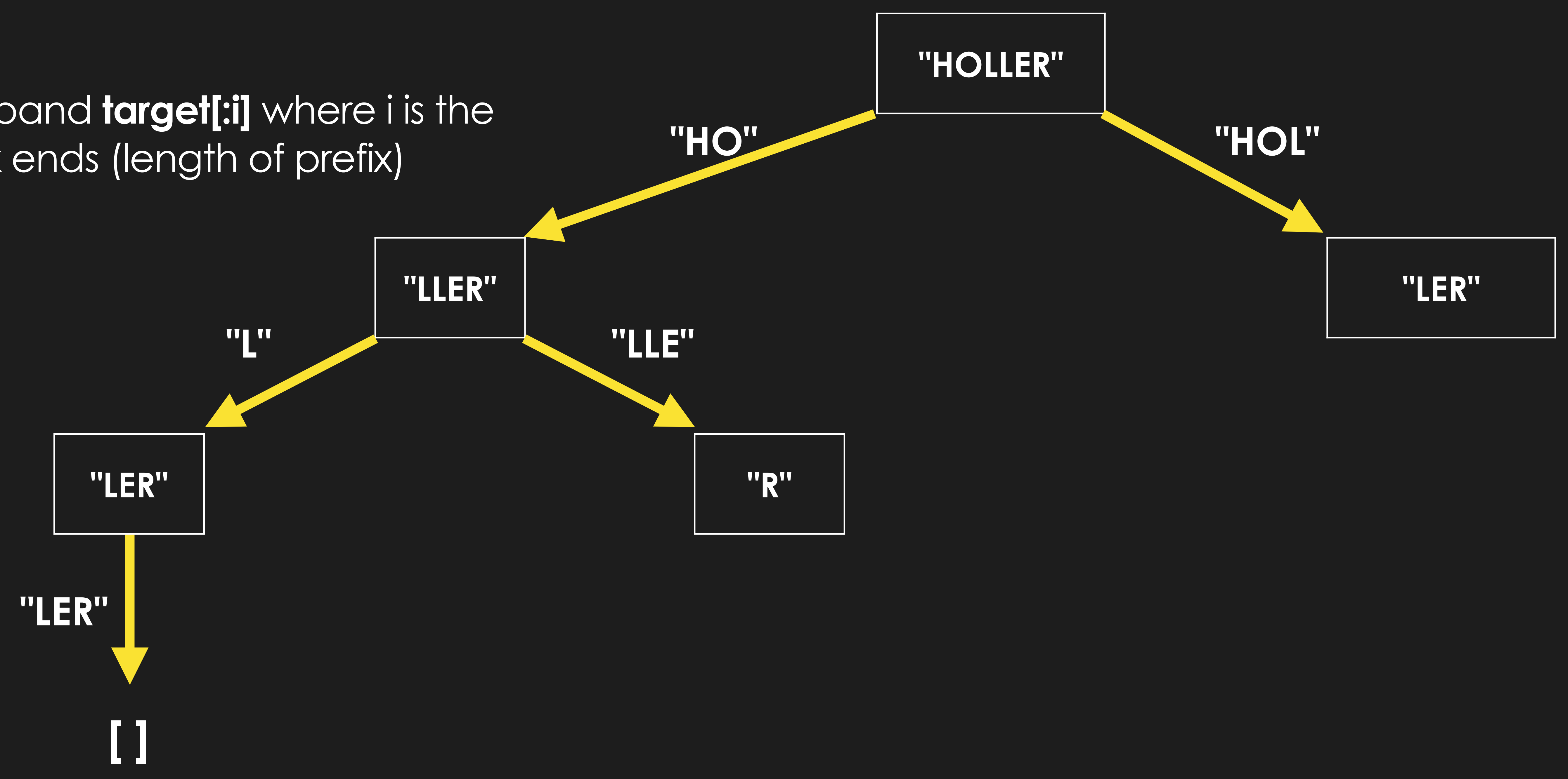


Base case 1: empty string (string can be constructed, return **empty array**)

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

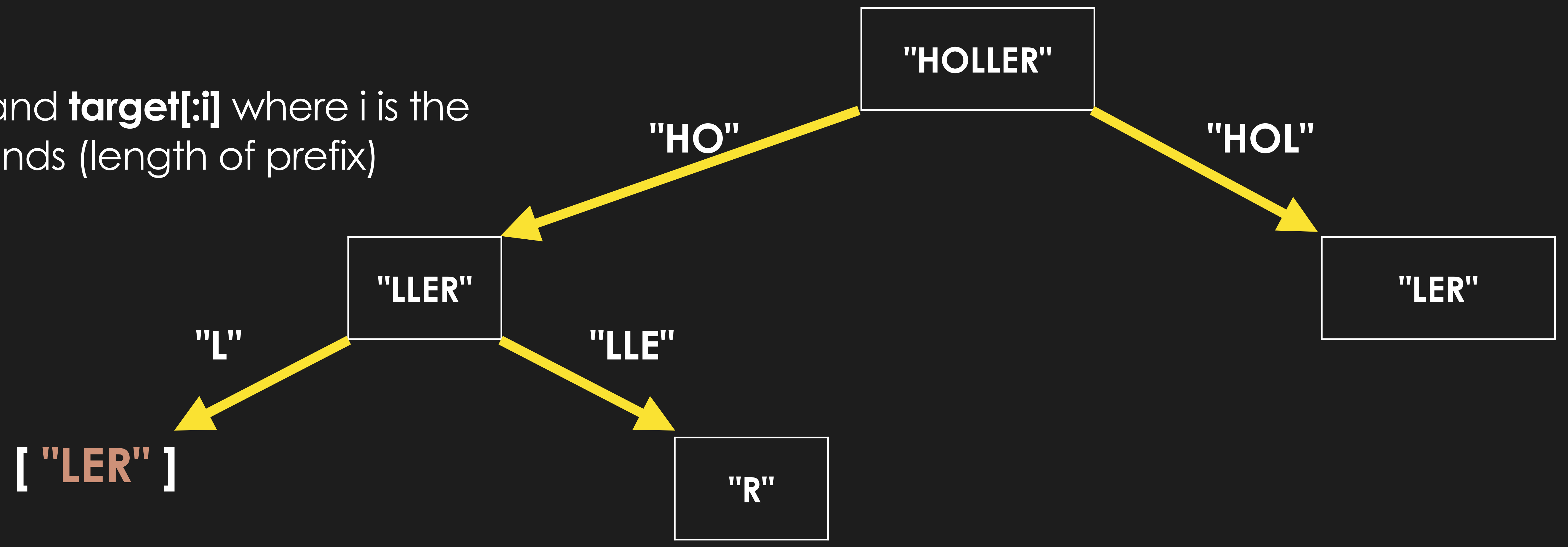


CodeIT

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

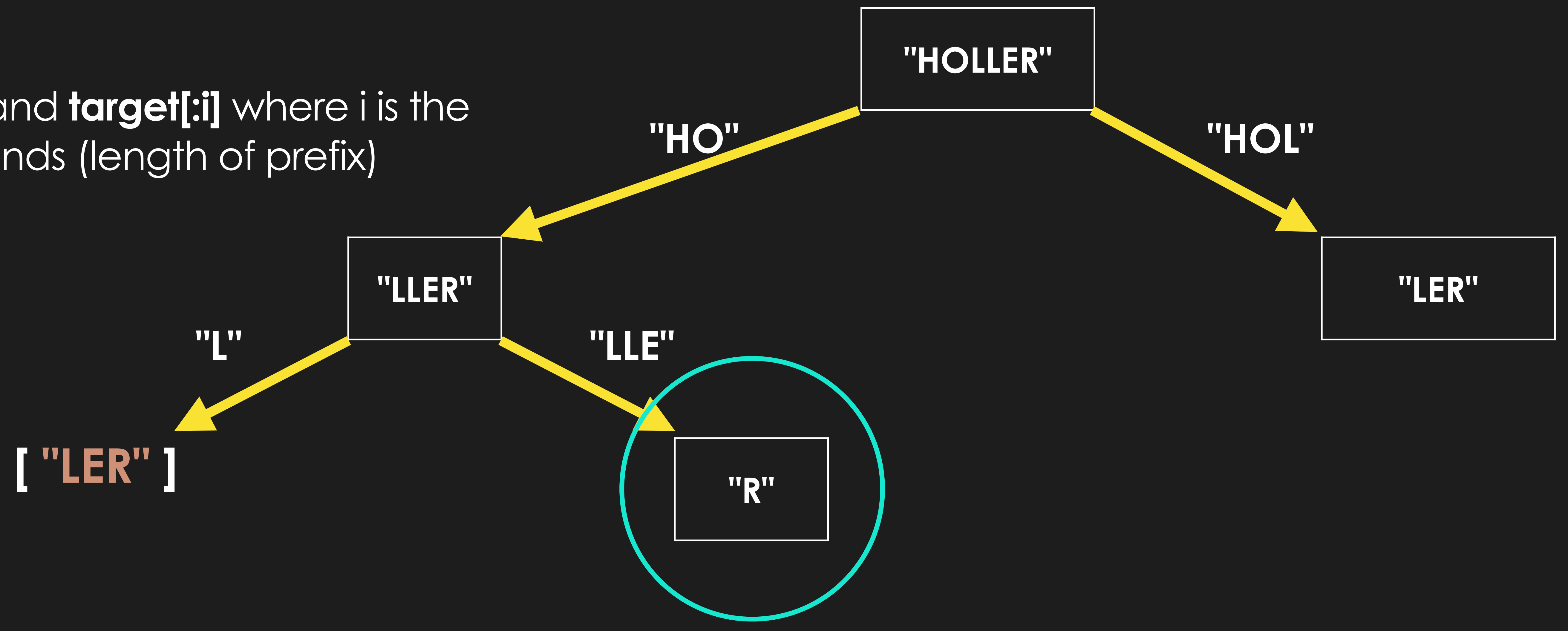
Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)



array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

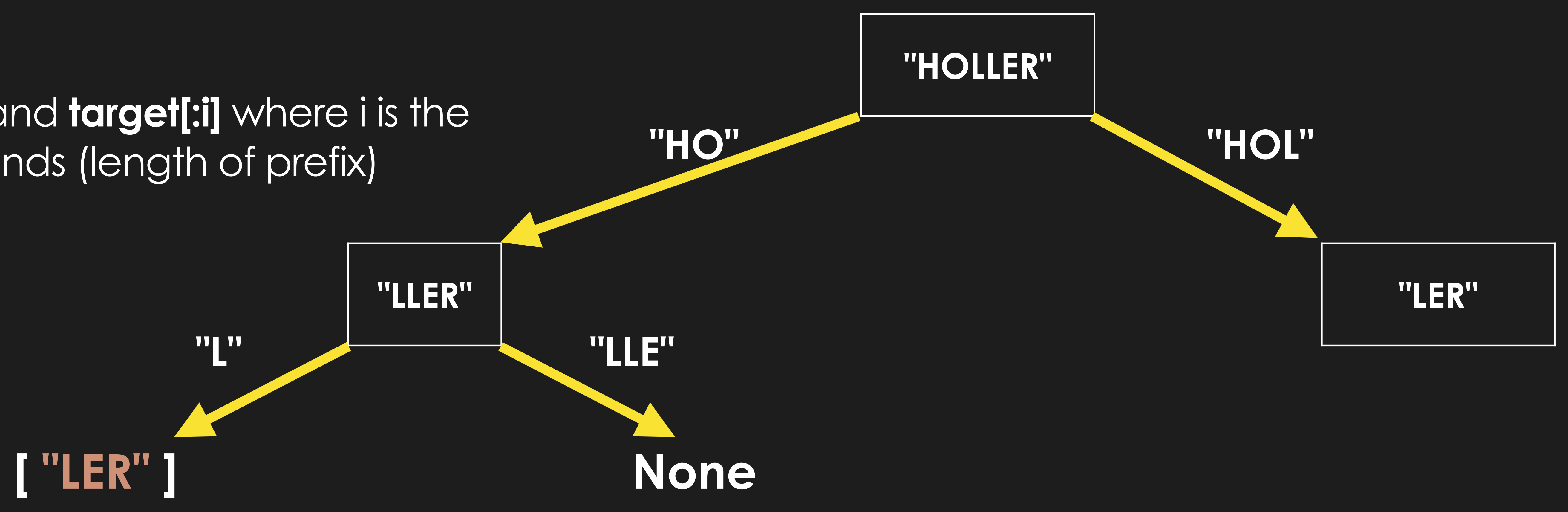


Base case 2: non-empty string (string cannot be constructed, return **None**)

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

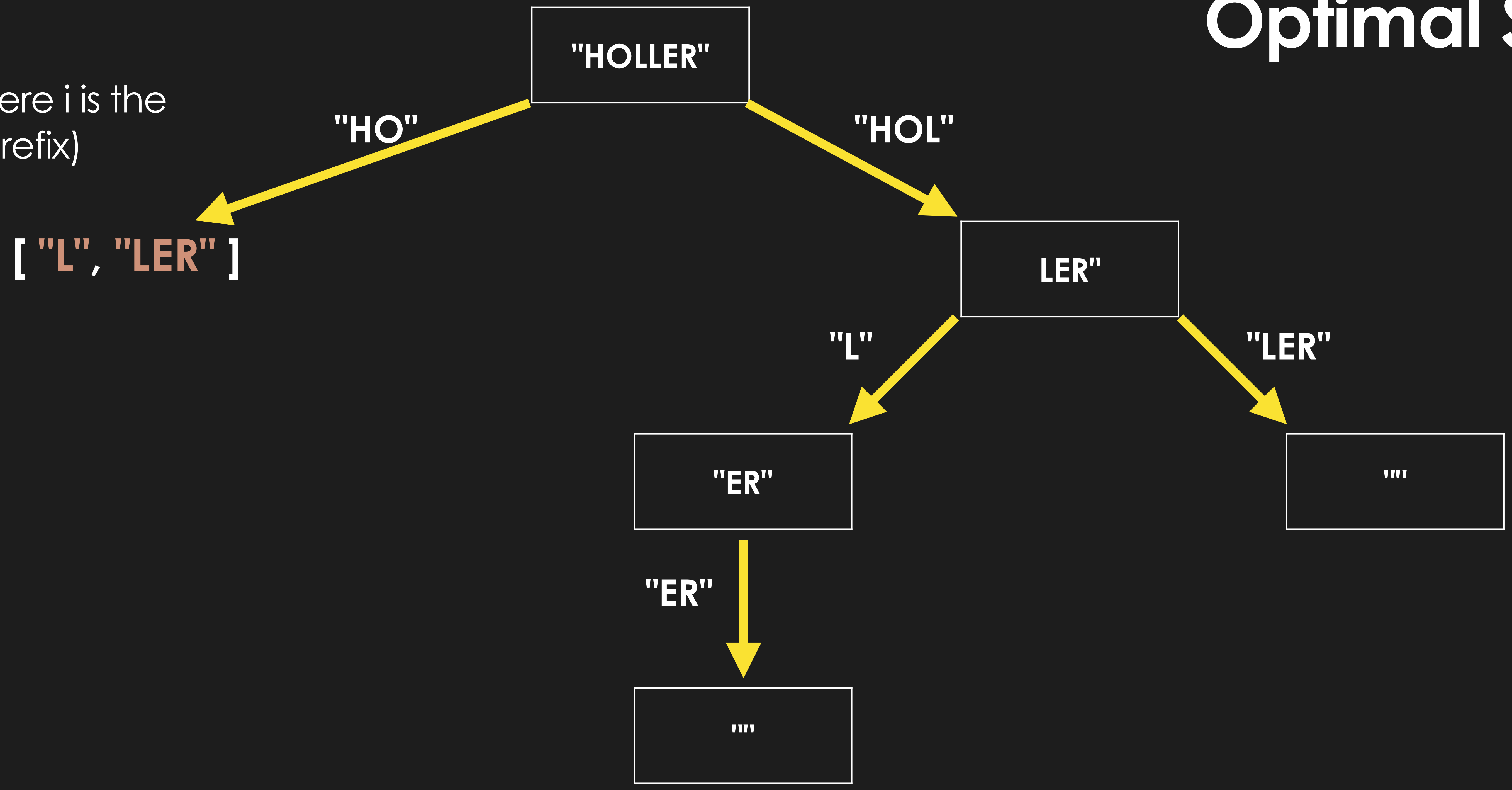
Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)



array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

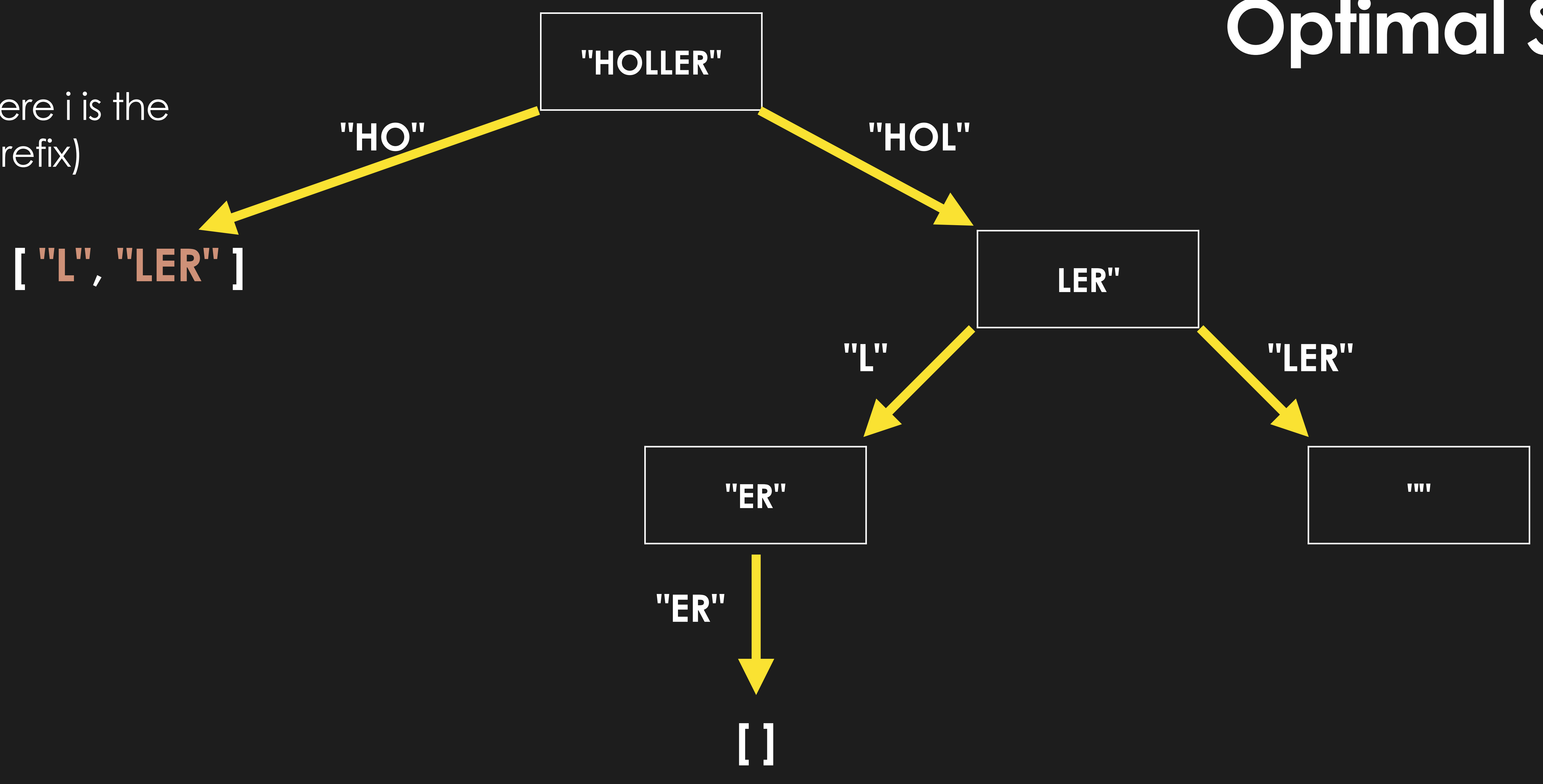


CodeIT

String Construct
Optimal Substructure

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

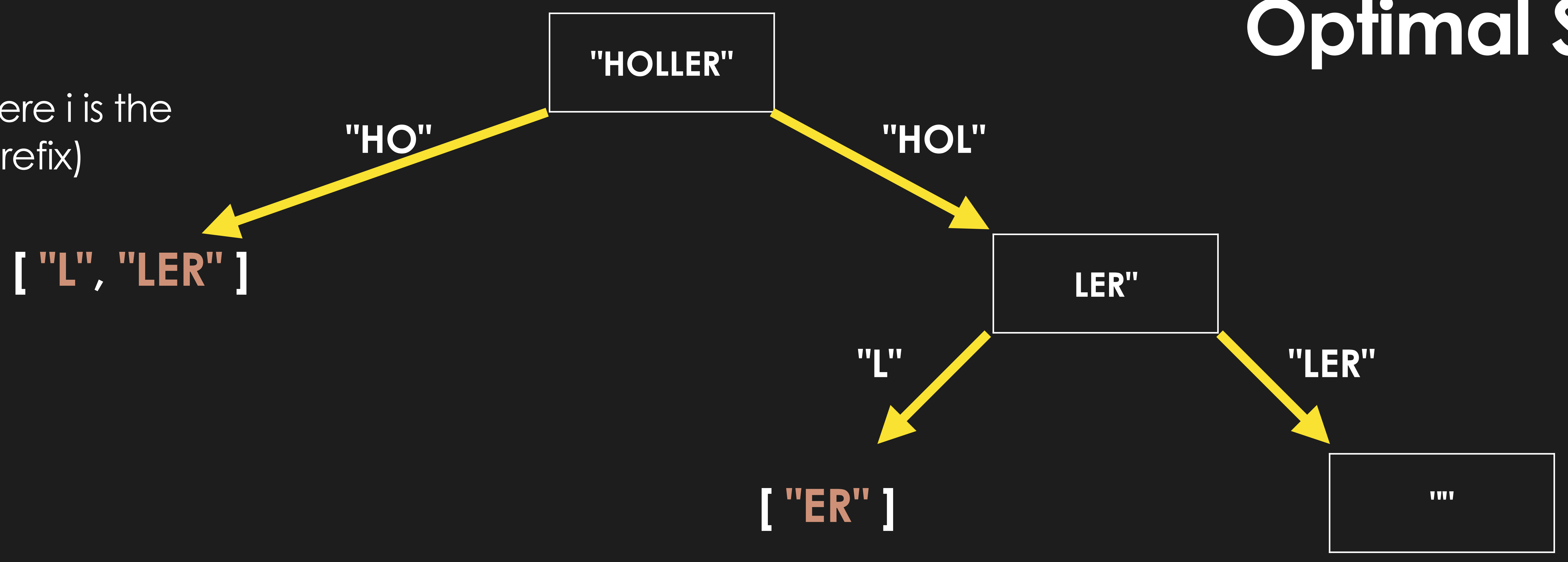
Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)



String Construct
Optimal Substructure

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

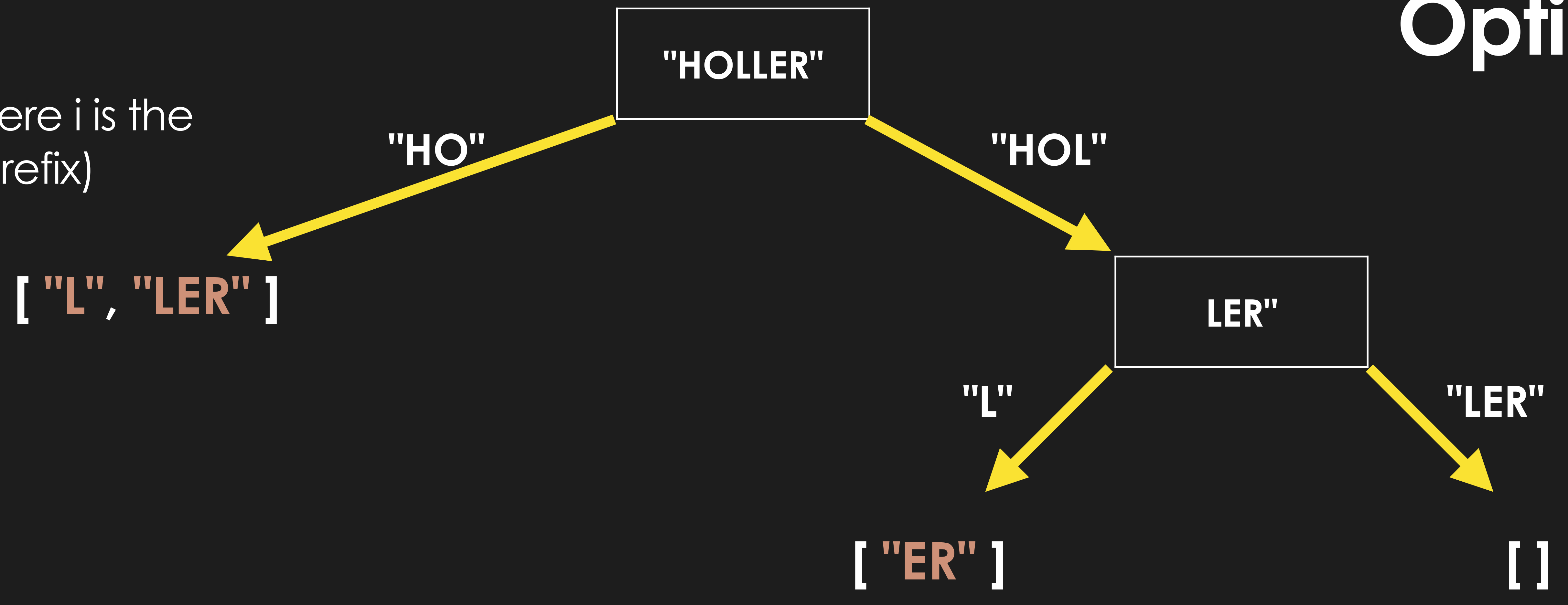


CodeIT

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

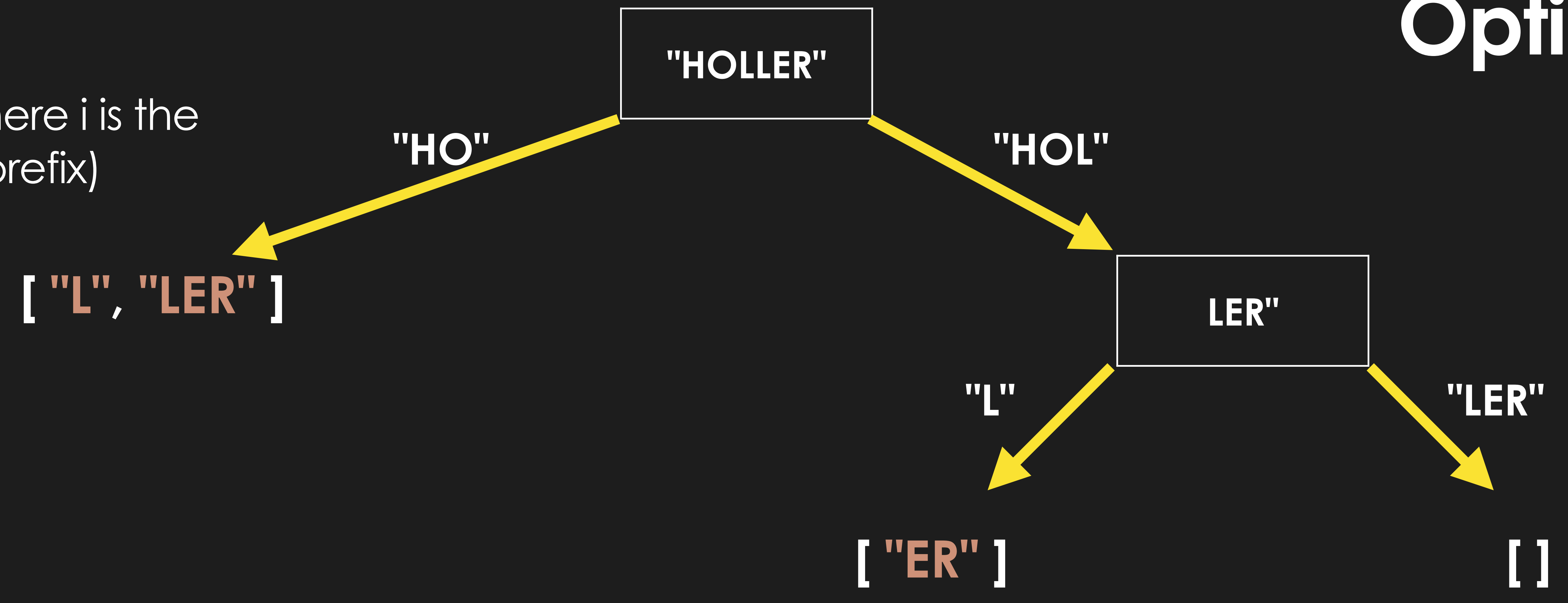


CodeIT

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)



Since right path has lesser elements in returning array, use that

CodeIT

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

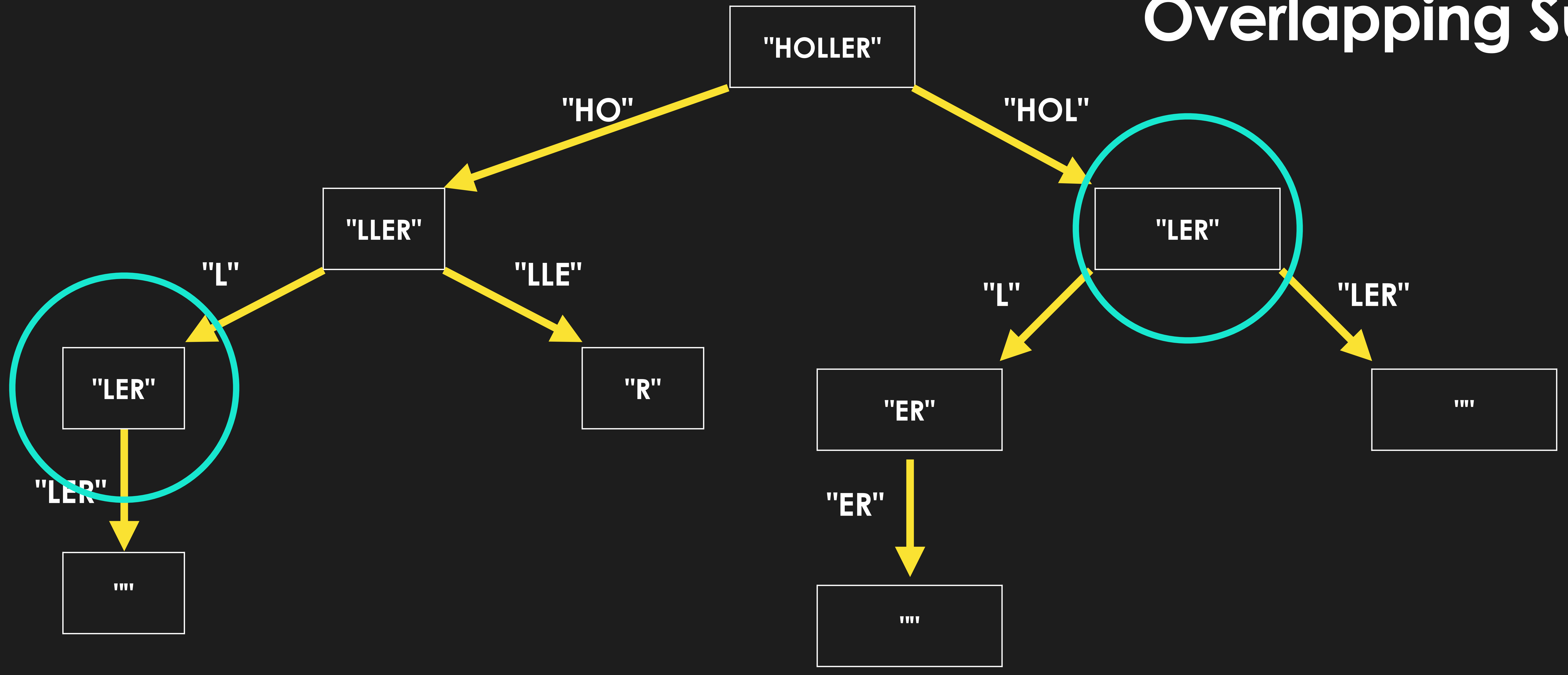
String Construct
Optimal Substructure

Recursive pattern:
If prefix exists in array, expand **target[:i]** where i is the index at which the prefix ends (length of prefix)

["HOLLER"]

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Overlapping Subproblems



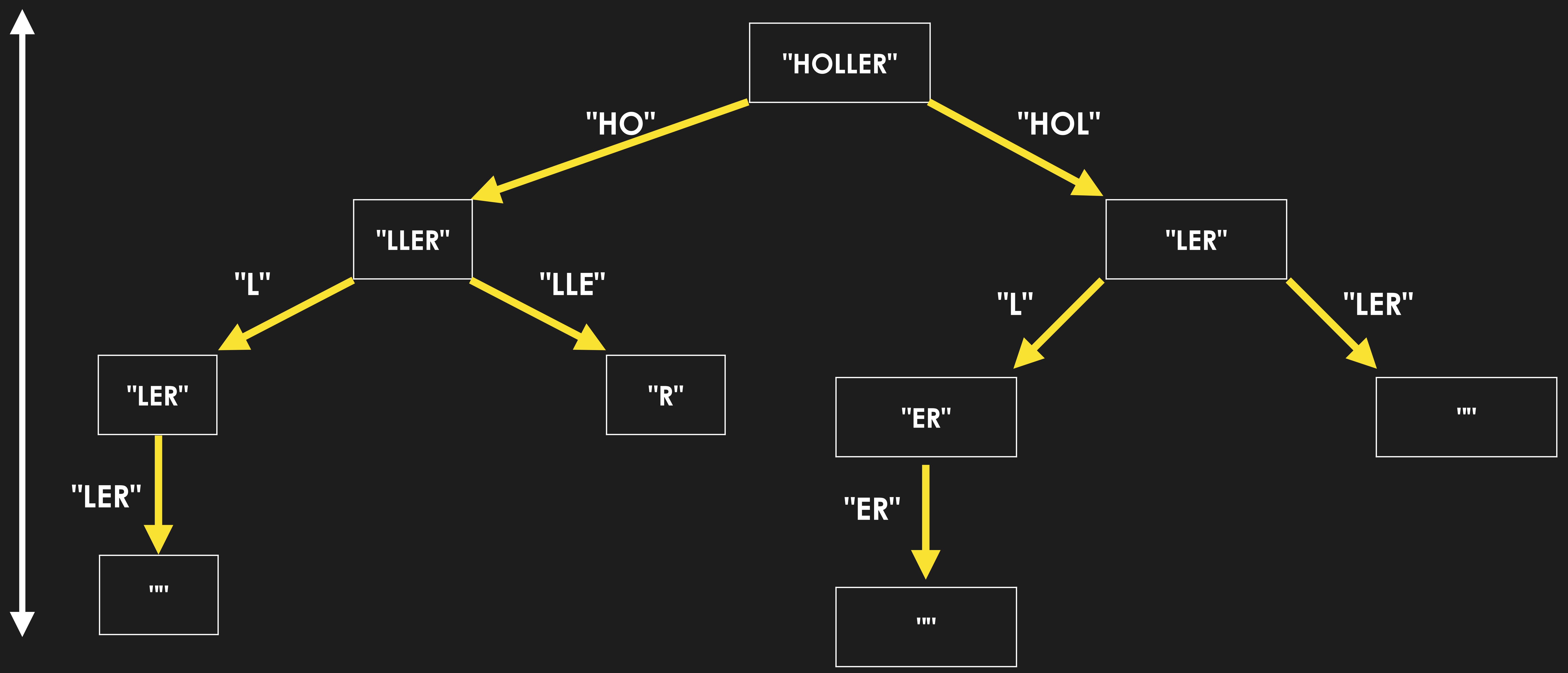
CodeIT

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Overlapping Subproblems

length of target: **m**
length of array: **n**

height: m



CodeIT

Implementation of string construct

```python
def stringConstruct(array, string):
    if string == "":
        return []

    bestRes = None
    for str in array:
        i = string.find(str)

        if i != 0:
            continue

        res = stringConstruct(array, string[len(str):])
        if res != None:
            if bestRes == None or len(bestRes) > len(res):
                bestRes = res
                bestRes.insert(0, str)

    return bestRes
```

CodeIT

```python
def stringConstruct(array, string):
    if string == "":
        return []

    bestRes = None
    for str in array:
        i = string.find(str)

        if i != 0:
            continue

        res = stringConstruct(array, string[len(str):])
        if res != None:
            if bestRes == None or len(bestRes) > len(res):
                bestRes = res.copy()
                bestRes.insert(0, str)

    return bestRes
```

n

m

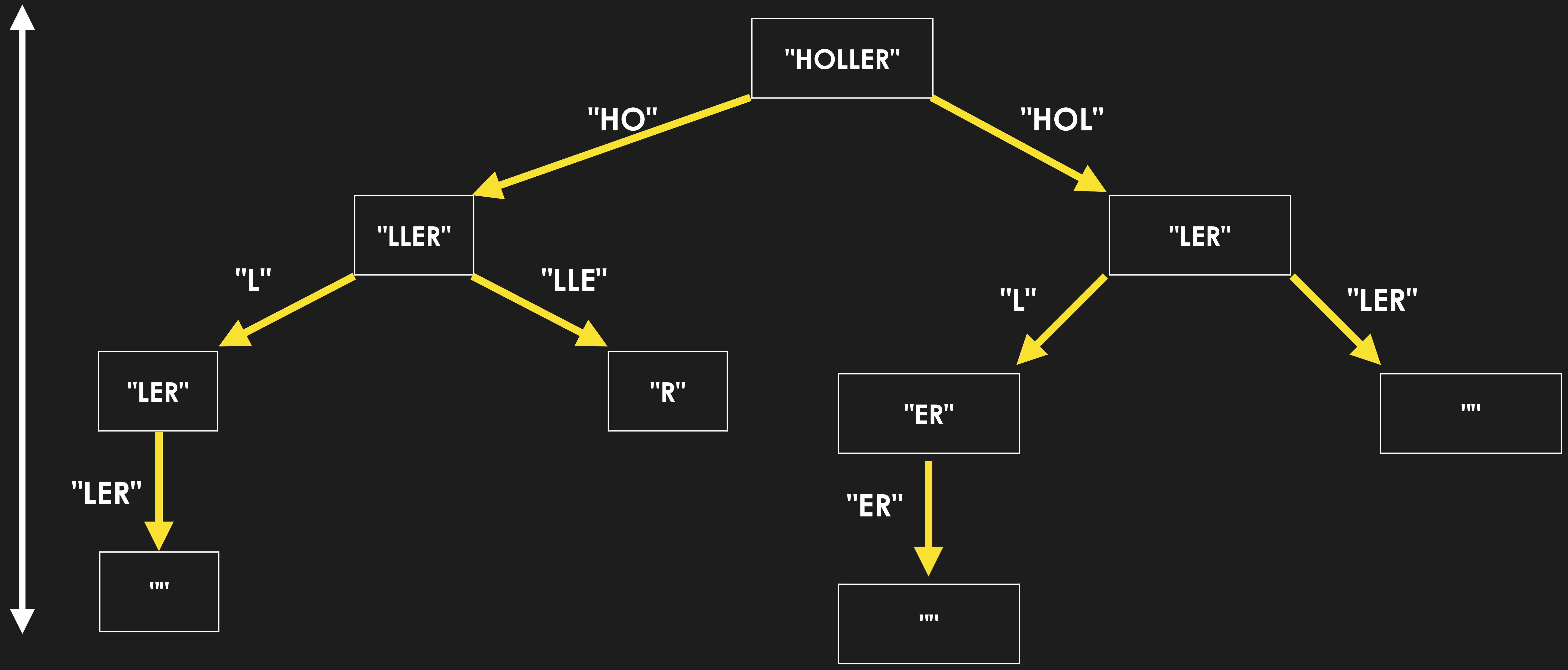
n^m

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Overlapping Substructure

length of target: **m**
length of array: **n**

height: m



time complexity: $m * n^{2m}$

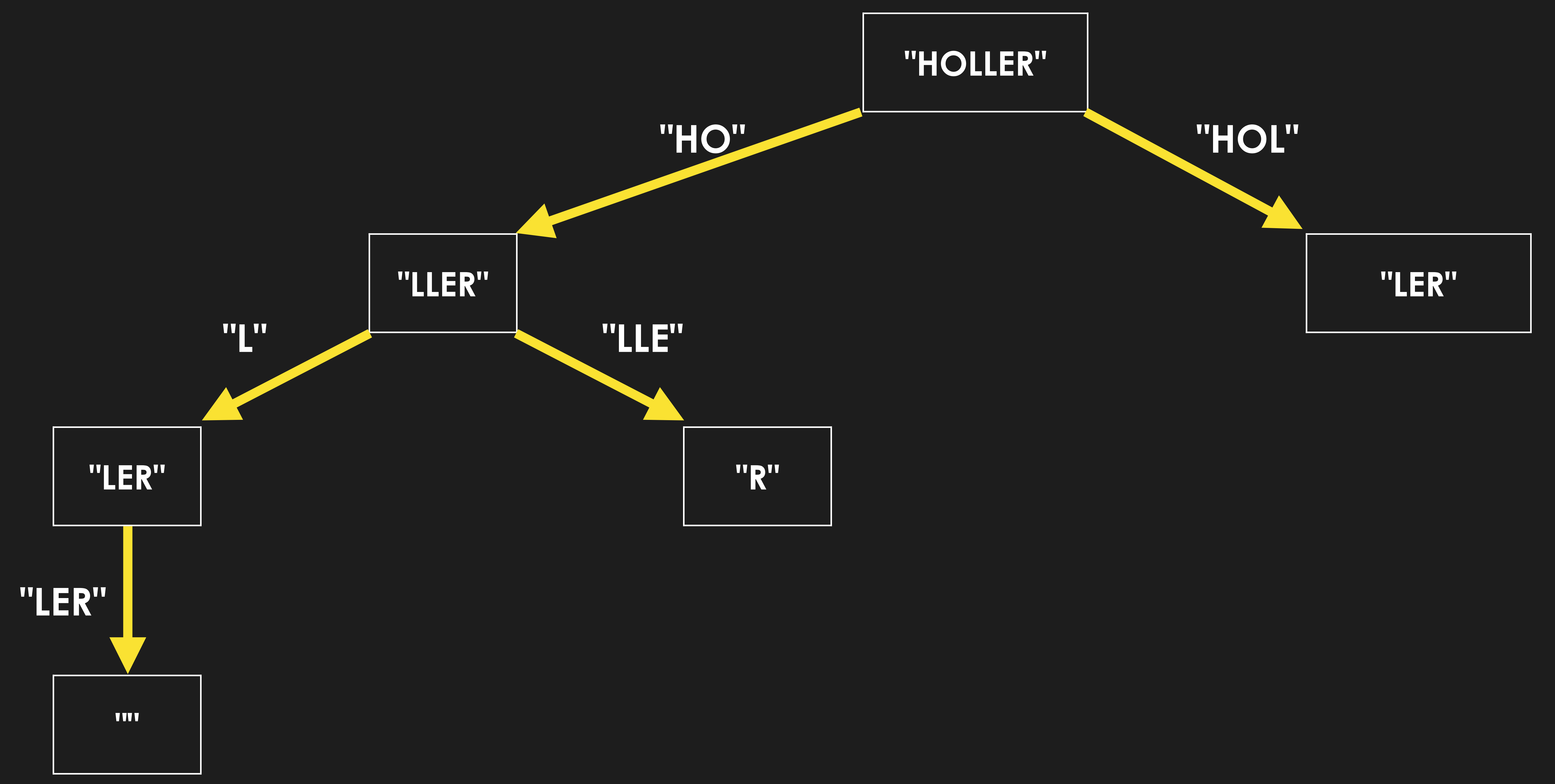
space complexity: m

Memoization

CodeIT

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

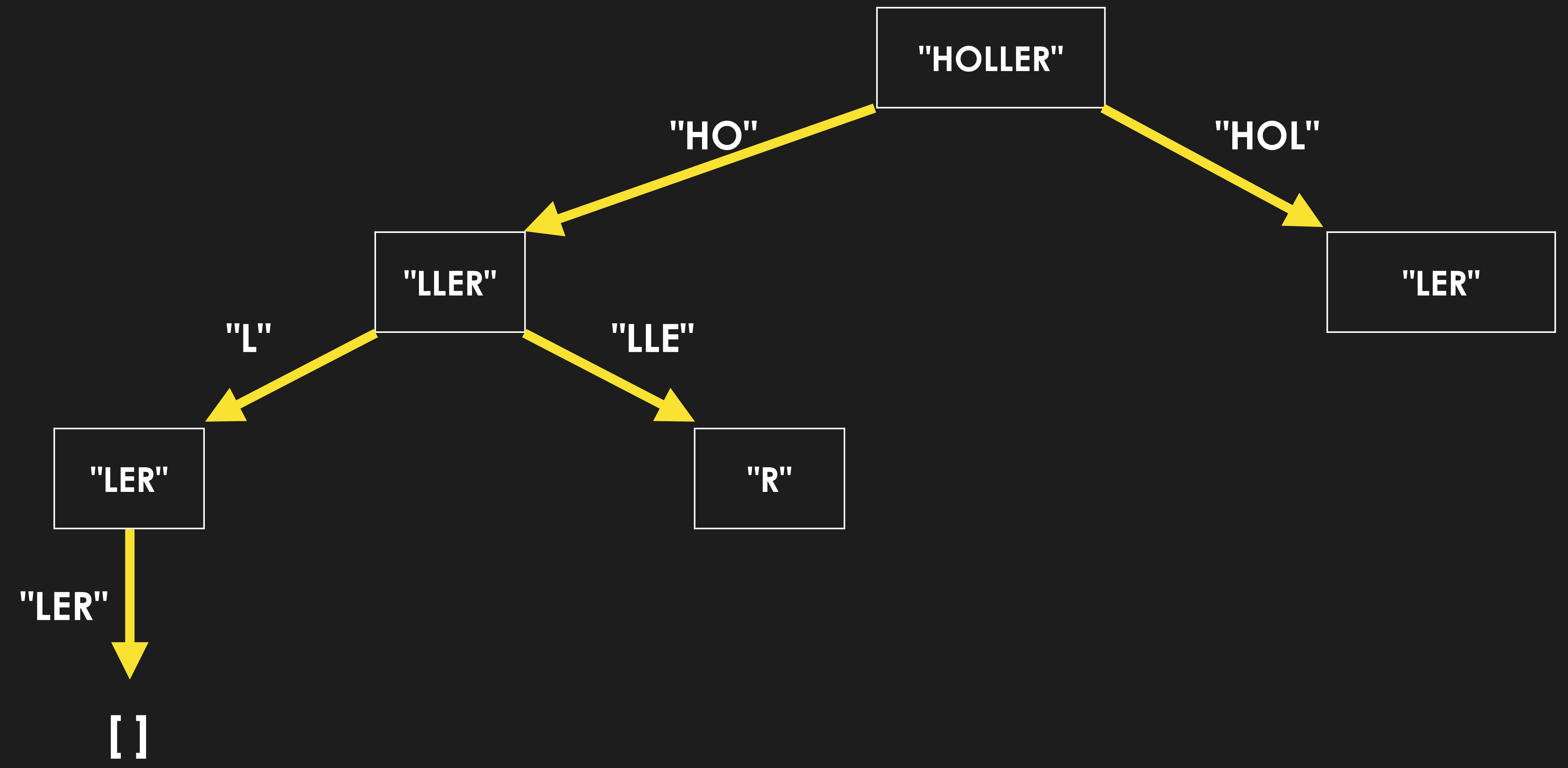
String Construct
Optimal Substructure



str						
mem						

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

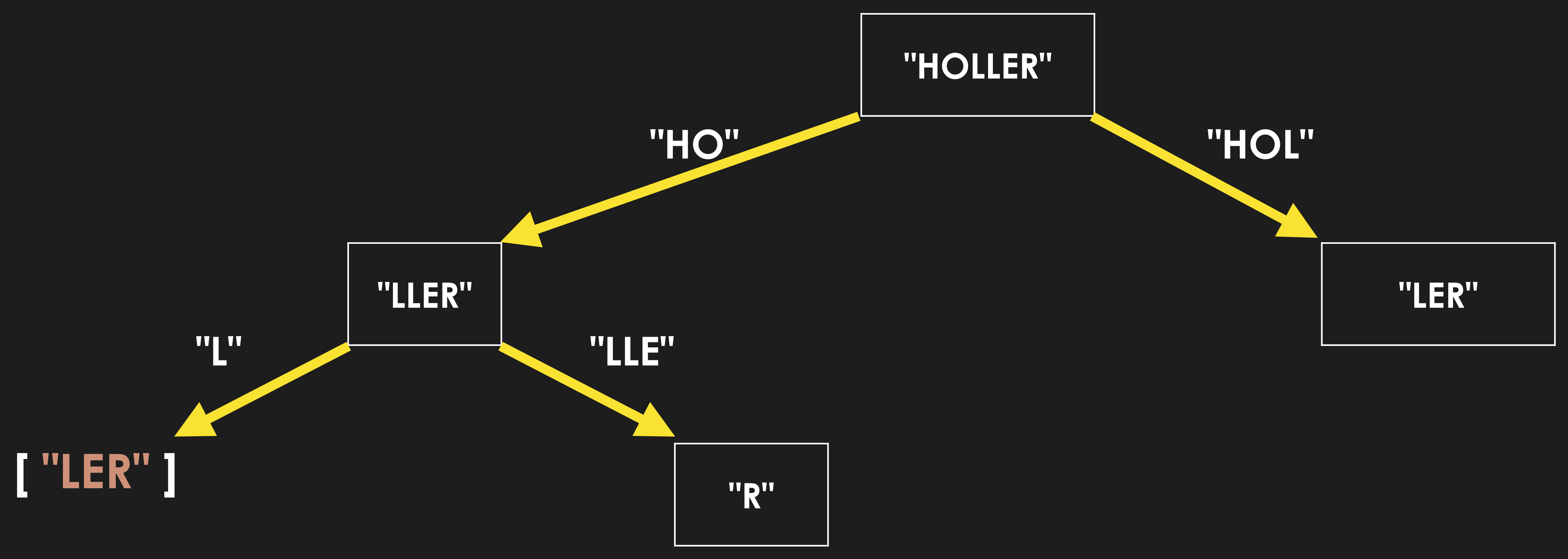
String Construct Optimal Substructure



str						
mem						

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

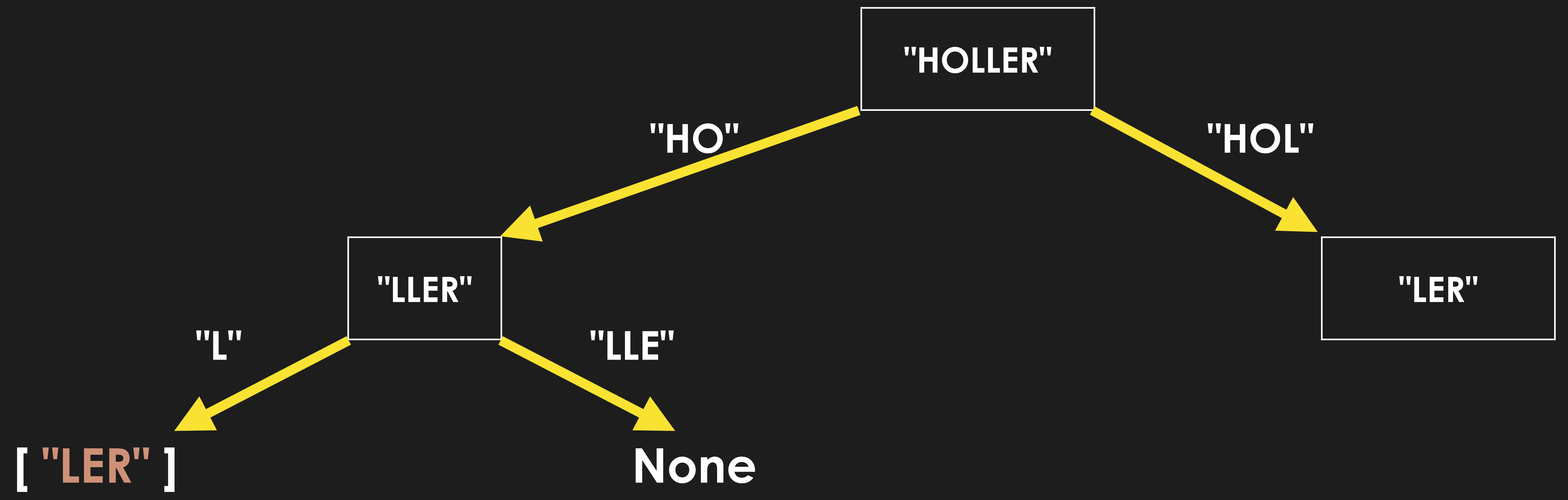
String Construct Optimal Substructure



str	"LER"					
mem	["LER"]					

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure

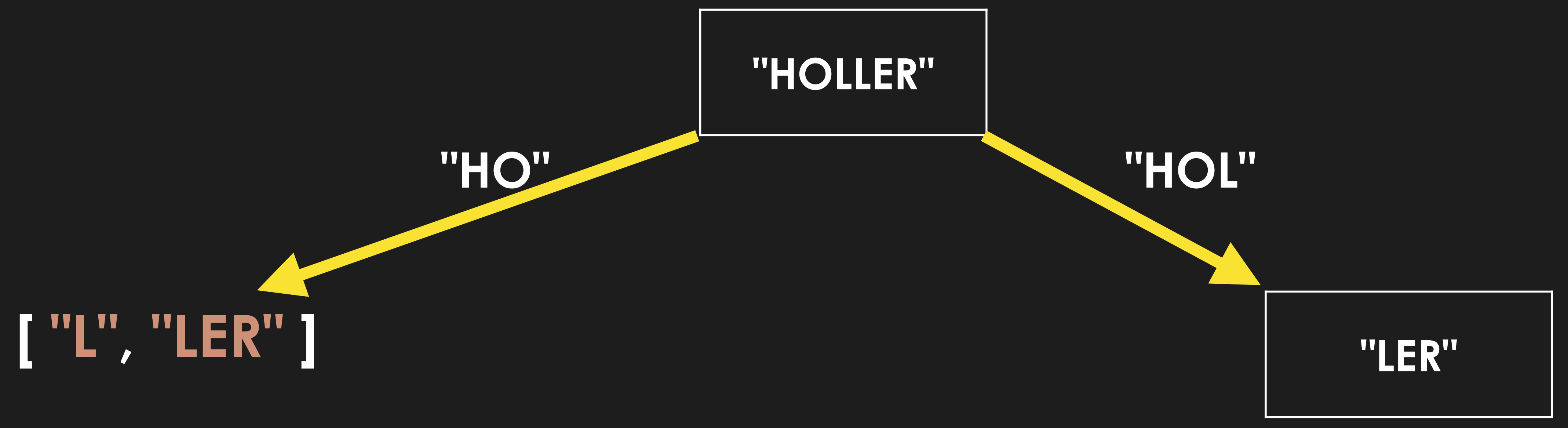


str	"LER"					
mem	["LER"]					

CodeIT

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

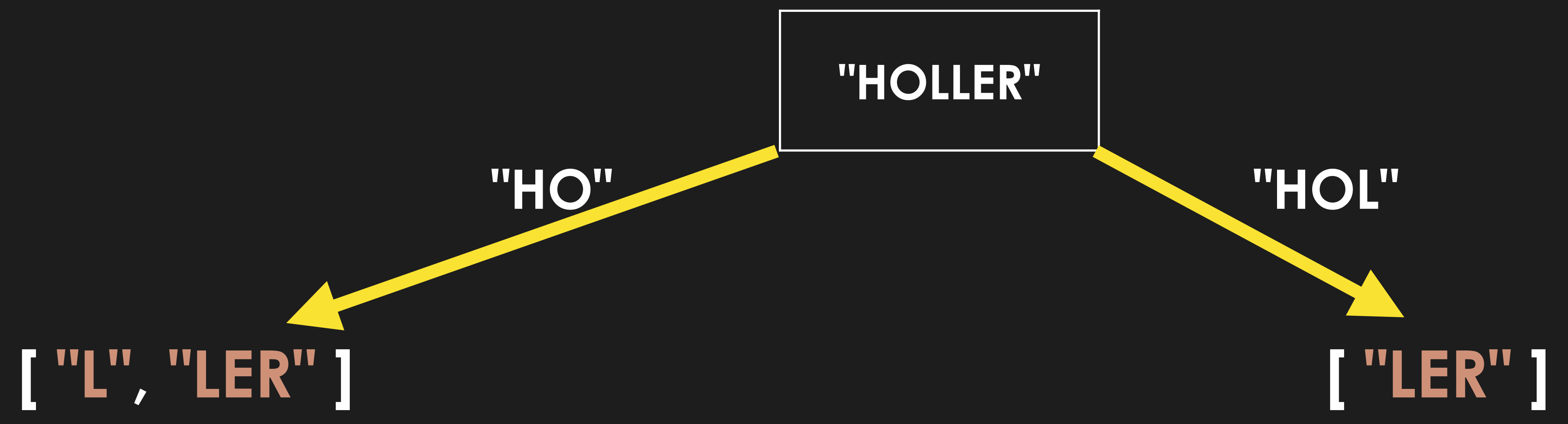
String Construct
Optimal Substructure



str	"LER"	"LLER"				
mem	["LER"]	["L", LER"]				

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure



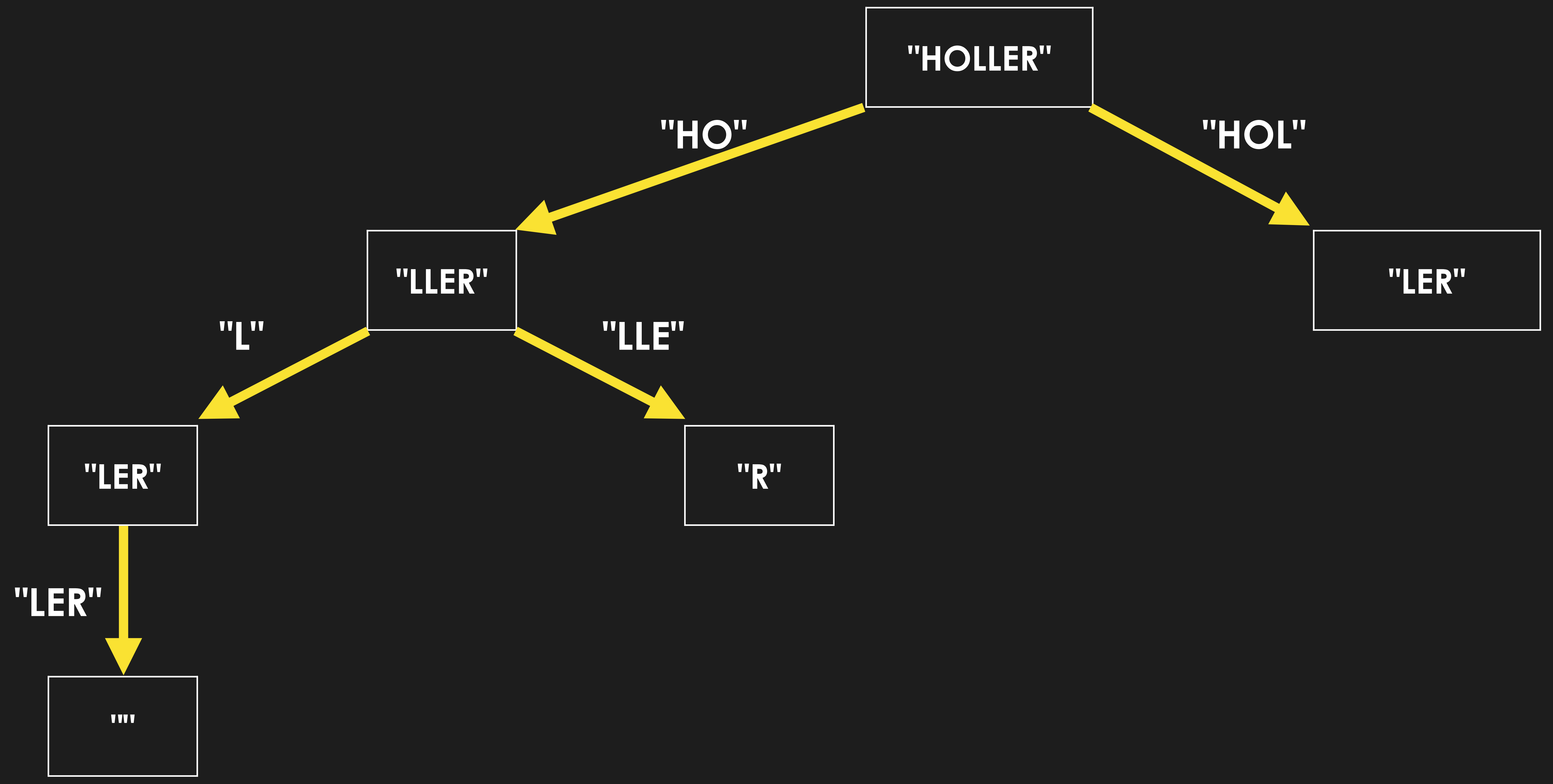
str	"LER"	"LLER"				
mem	["LER"]	["L", LER"]				

Implementation of string construct memo

```python
memo = {}

def stringConstruct(array, string):
    if string in memo:
        return memo[string]

    if string == "":
        return []

    bestRes = None

    for str in array:                 n
        i = string.find(str)               m

        if i != 0:
            continue

        res = stringConstruct(array, string[len(str):])          m
        if res != None:
            if bestRes == None or len(bestRes) > len(res):
                bestRes = res.copy()
                bestRes.insert(0, str)

    memo[string] = bestRes

    return bestRes
```

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

String Construct
Optimal Substructure



time complexity: m² * n

space complexity: m

DP Problem #5: Longest Common Subsequence

DP Problem #5: Common Subsequences

Given two strings, find the longest subsequence present in both strings

DP Problem #5: Longest Common Subsequence

Given two strings, find the longest subsequence present in both strings

Example:

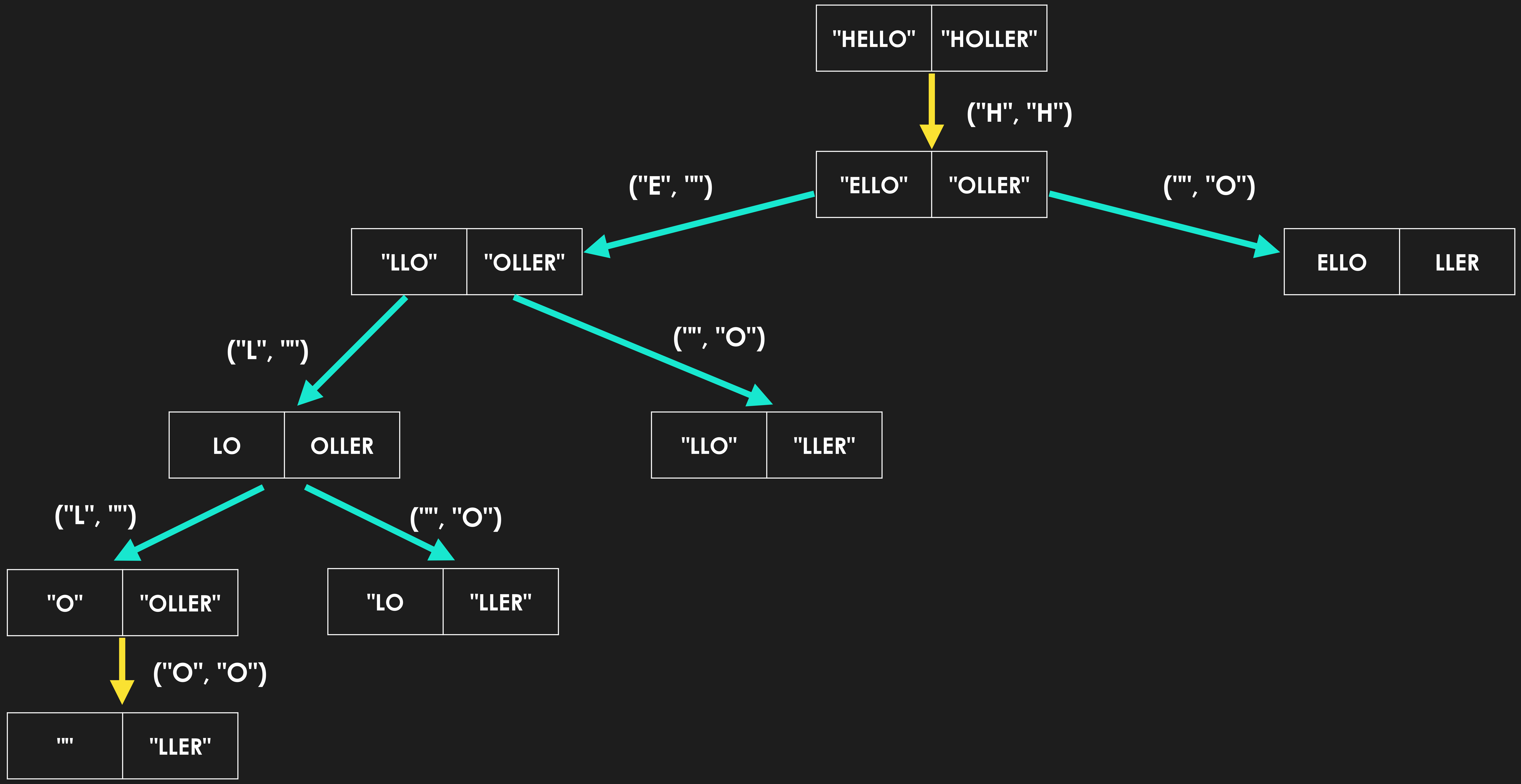
string1 = "HELLO"
string2 = "HOLLER"

lcs(string1, string2) => "HLL"

What do you think the optimal substructure is?

LCS Optimal Substructure

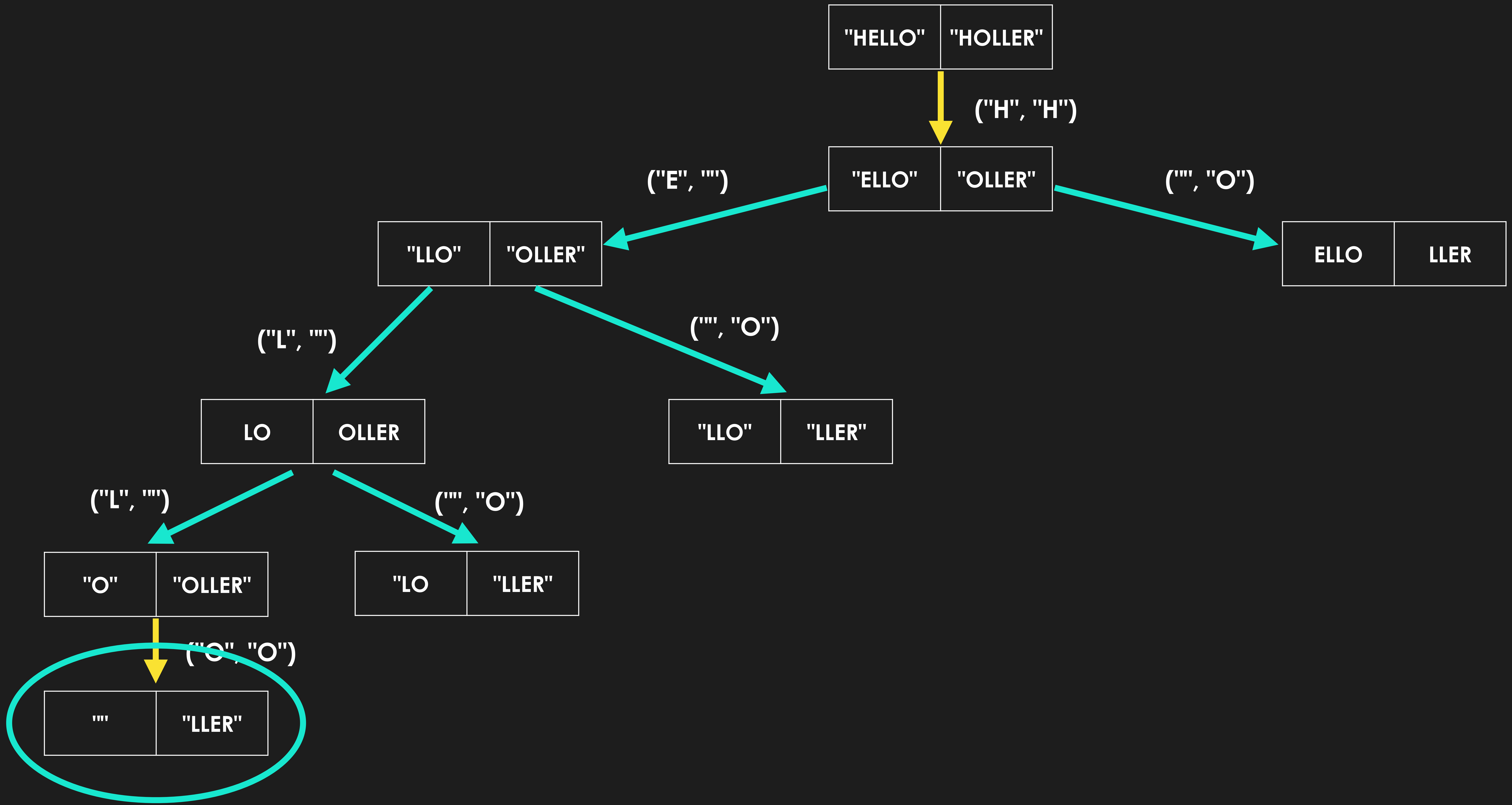
- **case 1**: first char matches
- **case 2**: first char doesn't match



CodeIT

- case 1: first char matches
- case 2: first char doesn't match

LCS Optimal Substructure

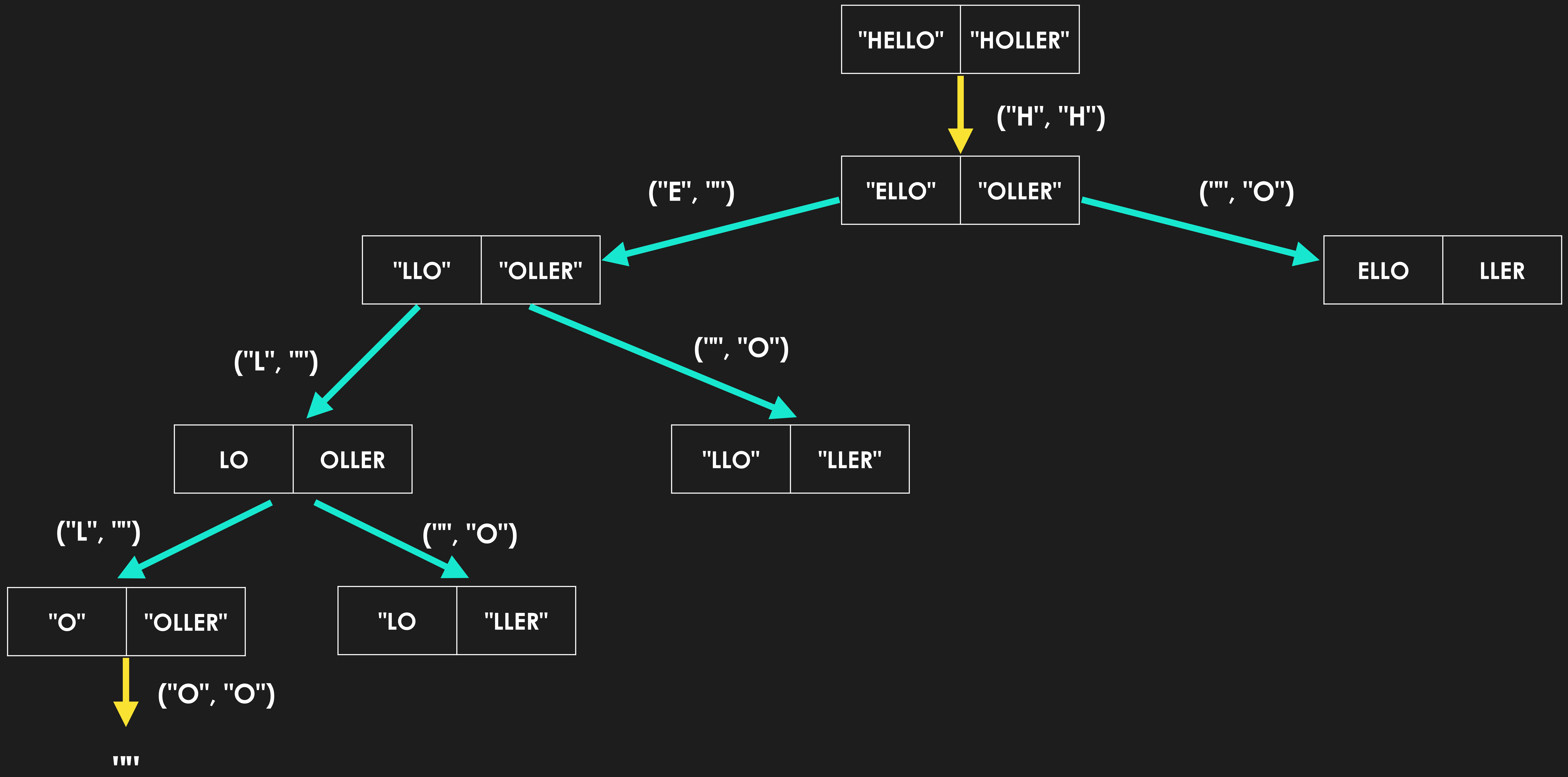


Base case!

CodeIT

LCS Optimal Substructure

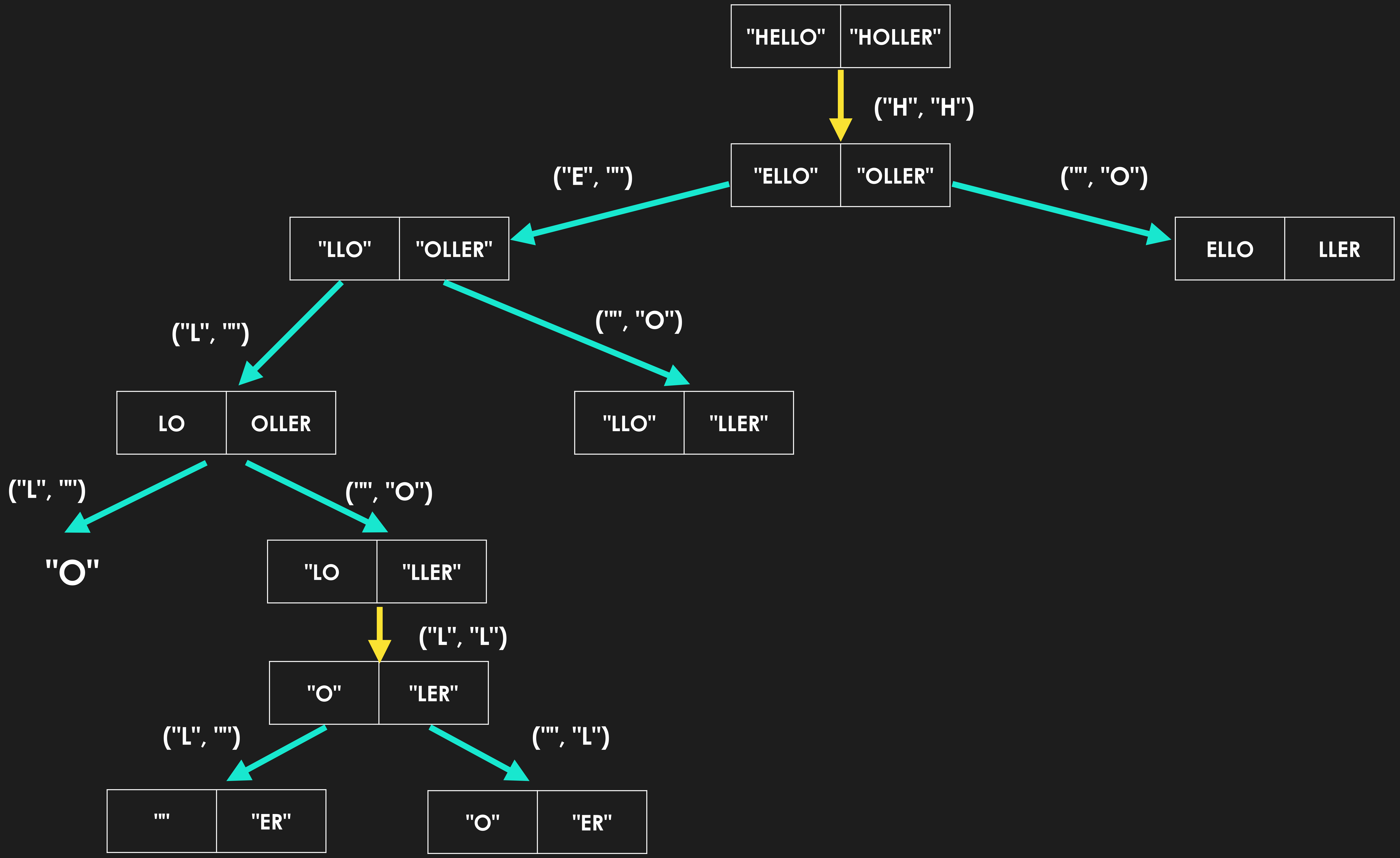
- **case 1**: first char matches
- **case 2**: first char doesn't match



Base case!

LCS Optimal Substructure

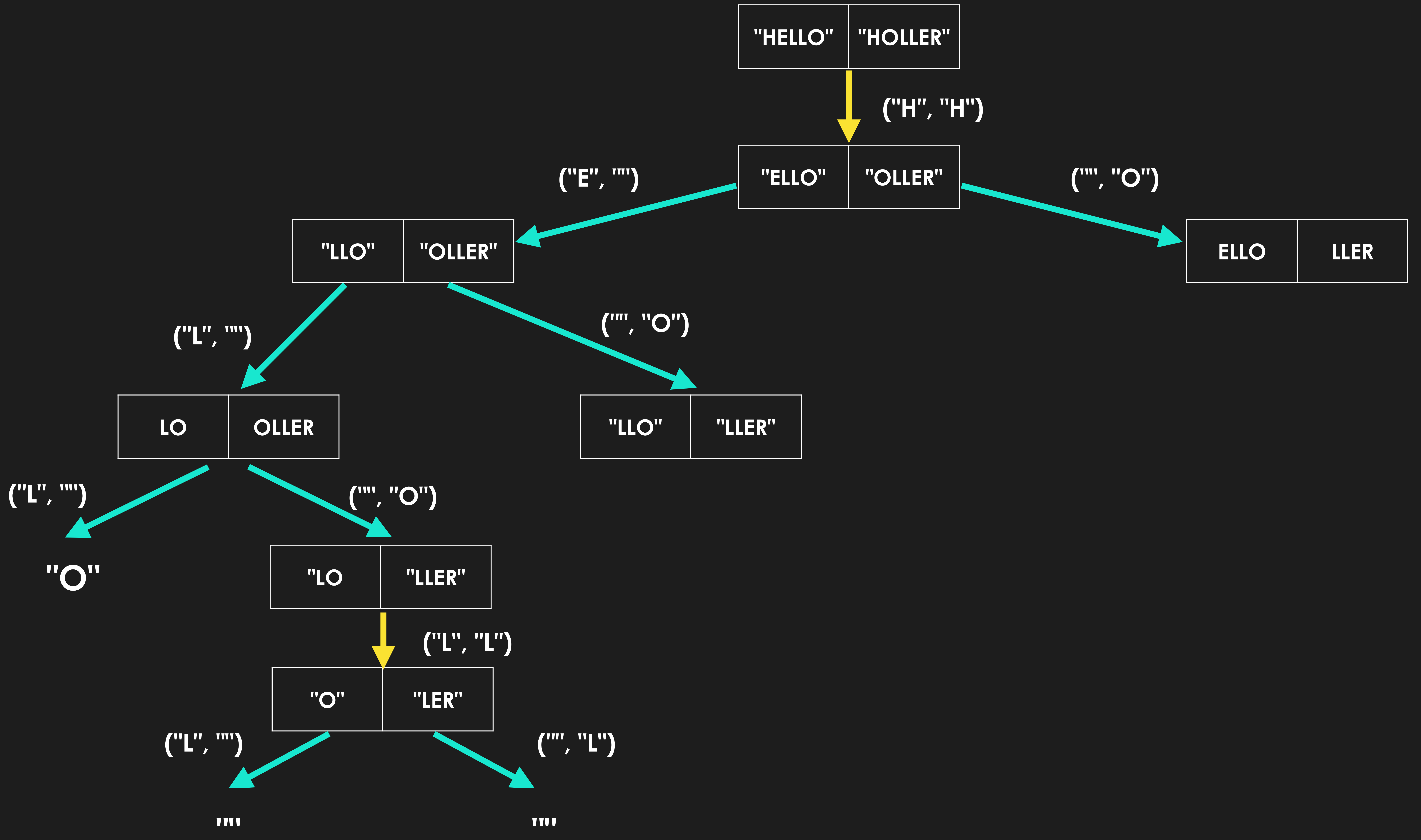
- **case 1**: first char matches
- **case 2**: first char doesn't match



CodeIT

- **case 1**: first char matches
- **case 2**: first char doesn't match

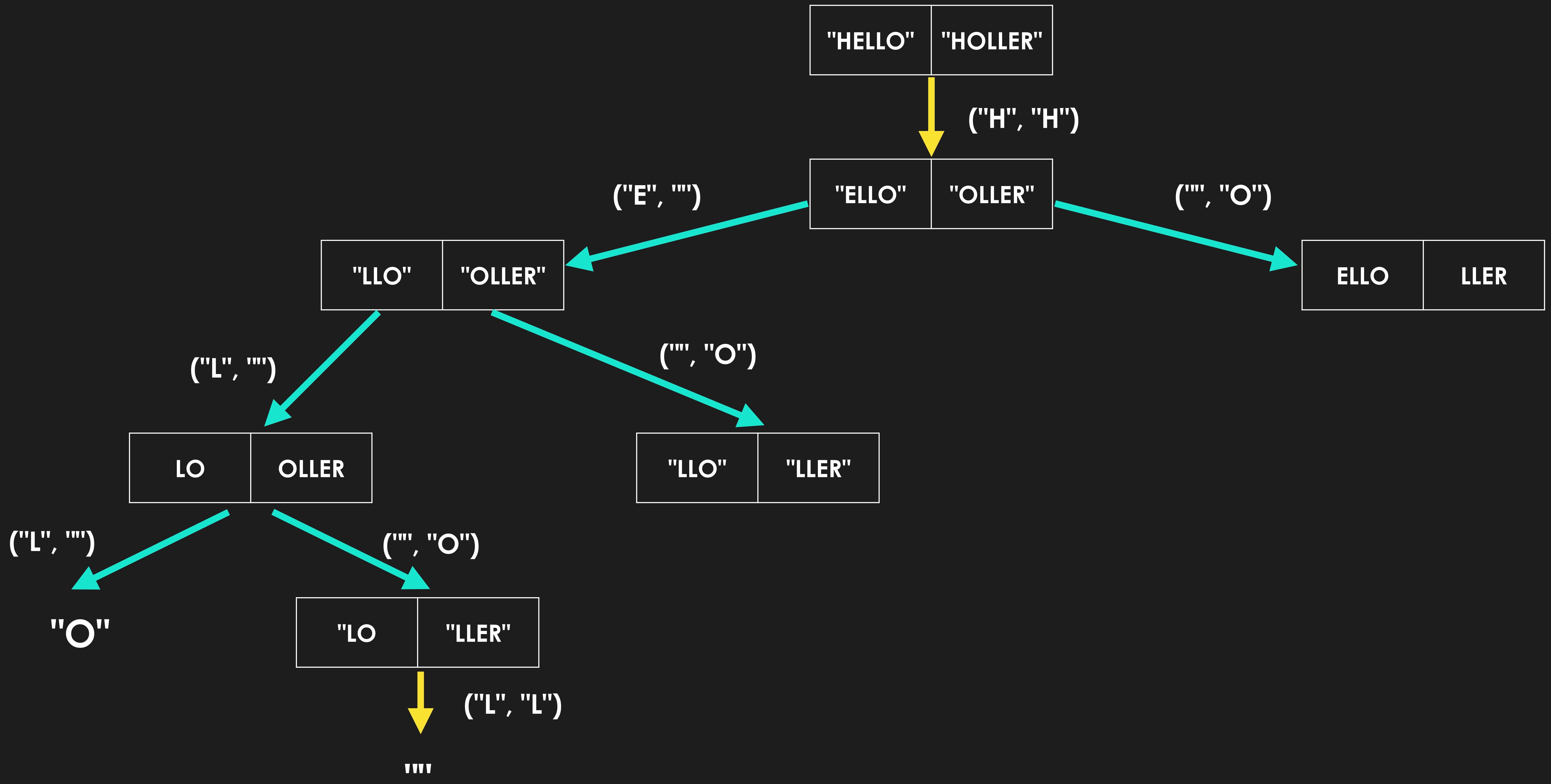
LCS Optimal Substructure



CodeIT

LCS Optimal Substructure

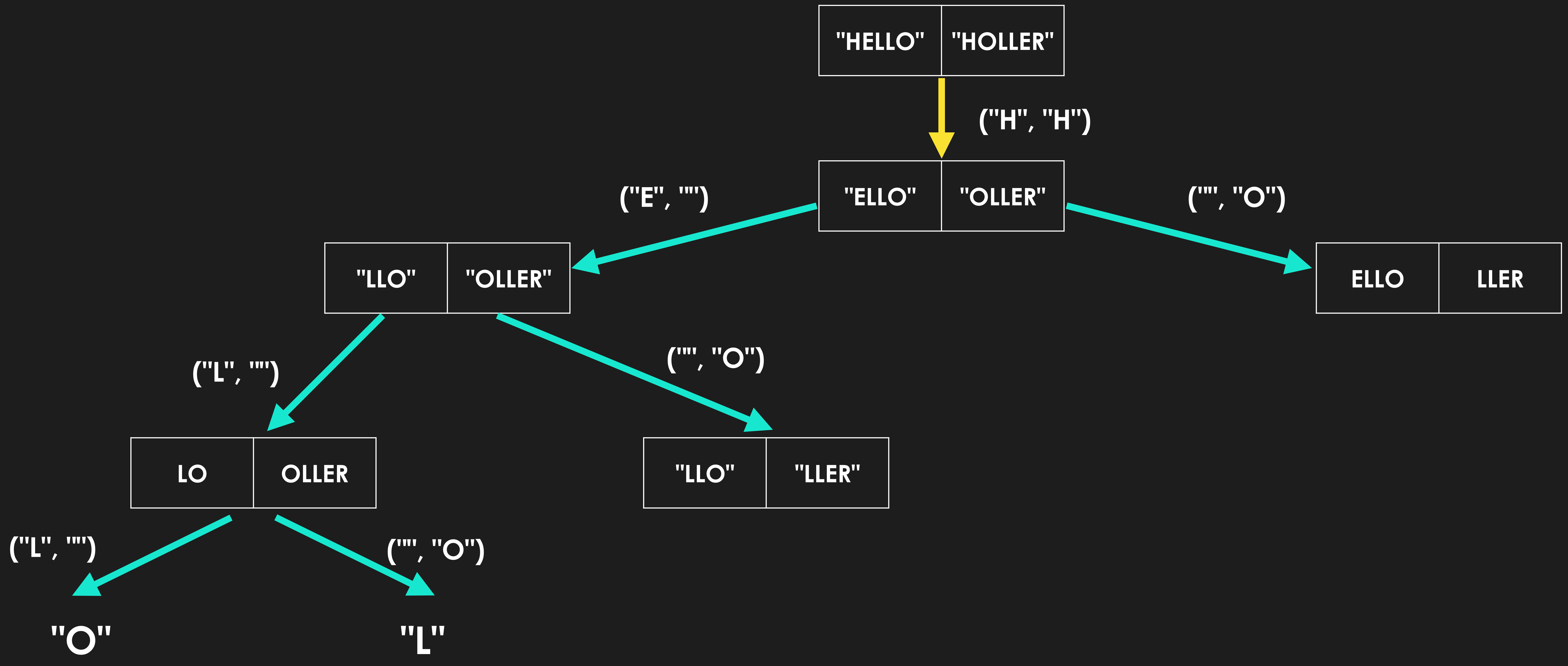
- **case 1**: first char matches
- **case 2**: first char doesn't match



CodeIT

- **case 1**: first char matches
- **case 2**: first char doesn't match

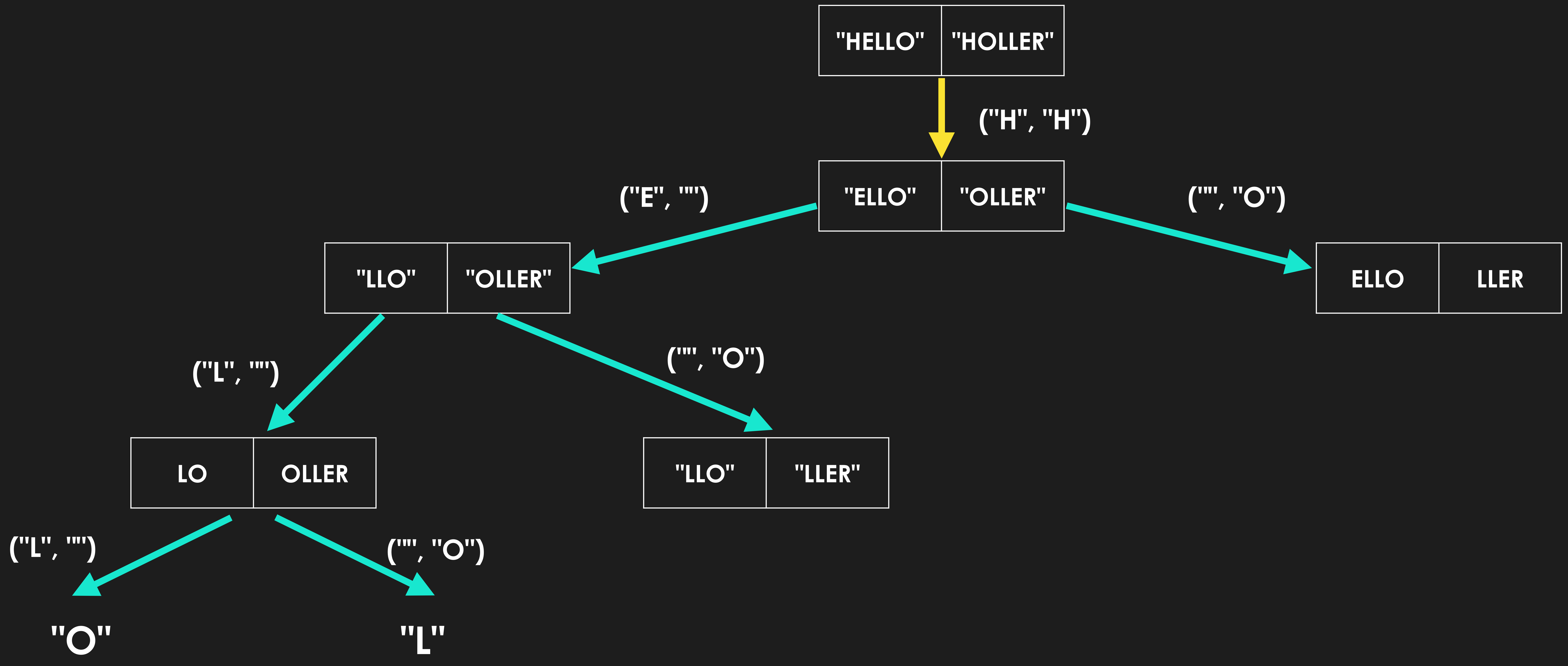
LCS Optimal Substructure



CodeIT

LCS Optimal Substructure

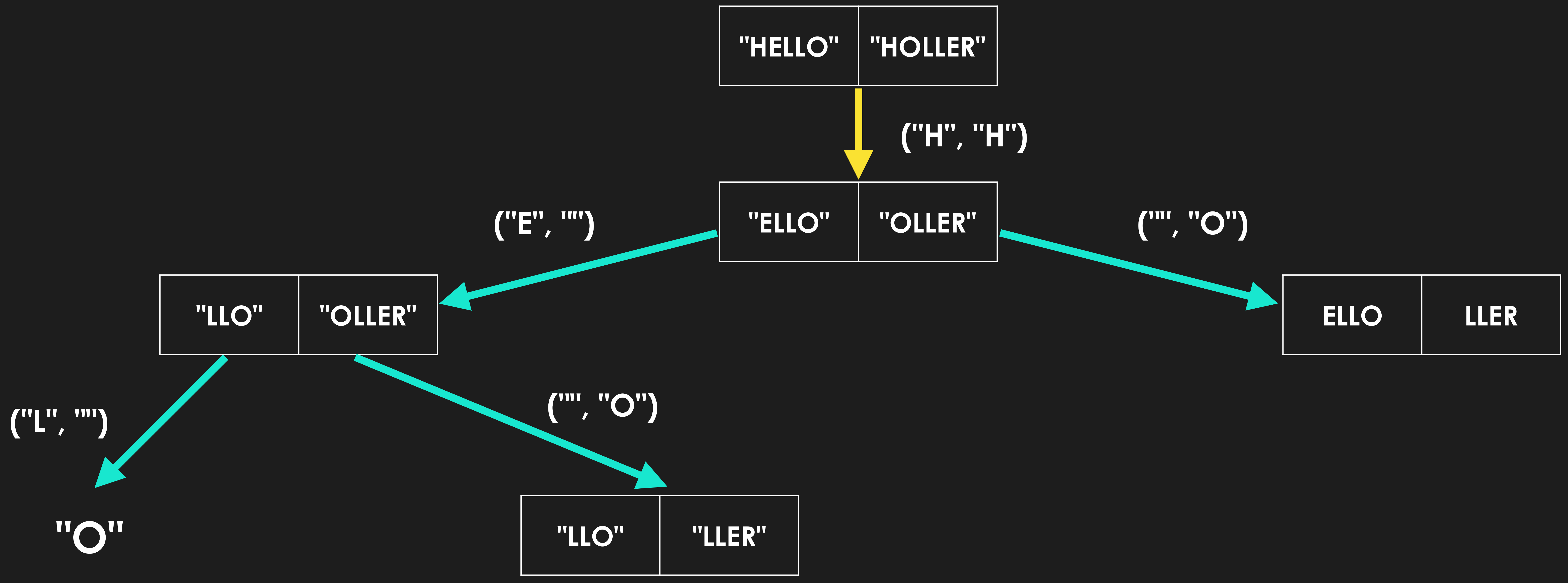
- case 1: first char matches
- case 2: first char doesn't match



Since "O" and "L" are the same length, return either

LCS Optimal Substructure

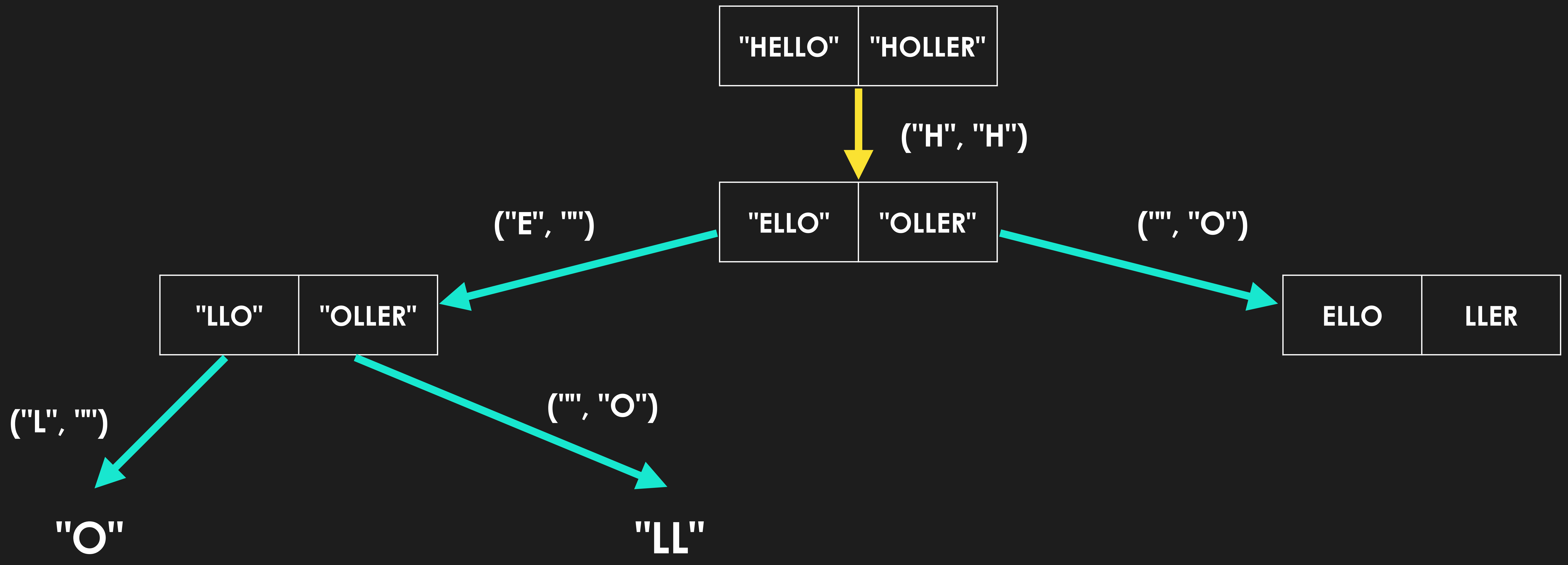
- case 1: first char matches
- case 2: first char doesn't match



CodeIT

- **case 1**: first char matches
- **case 2**: first char doesn't match

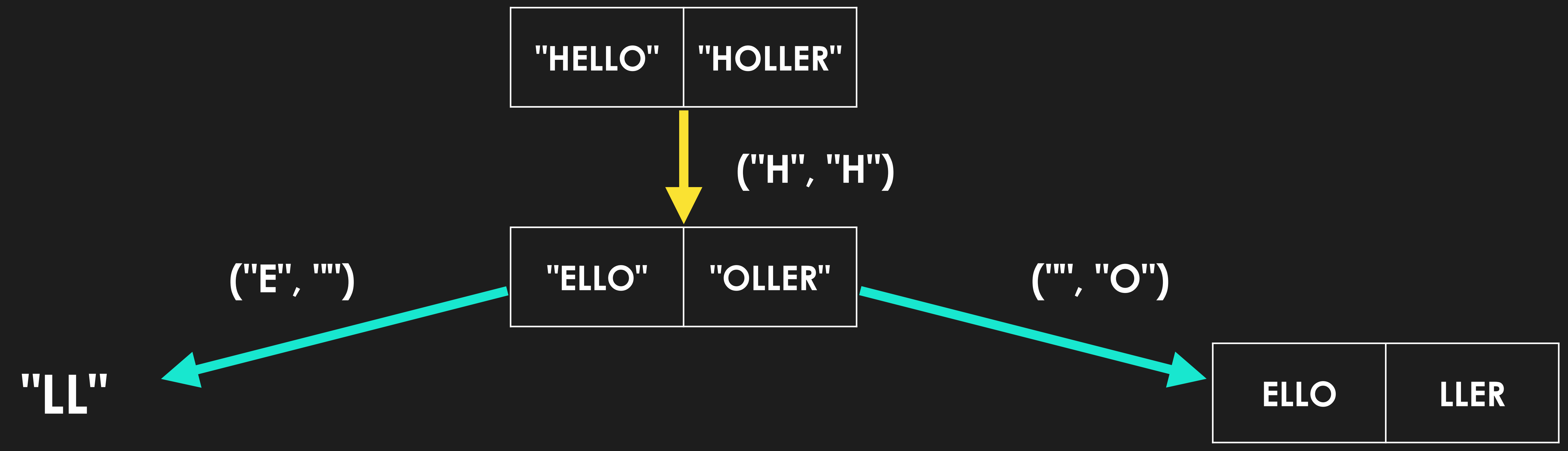
LCS Optimal Substructure



Fast Forward....

LCS Optimal Substructure

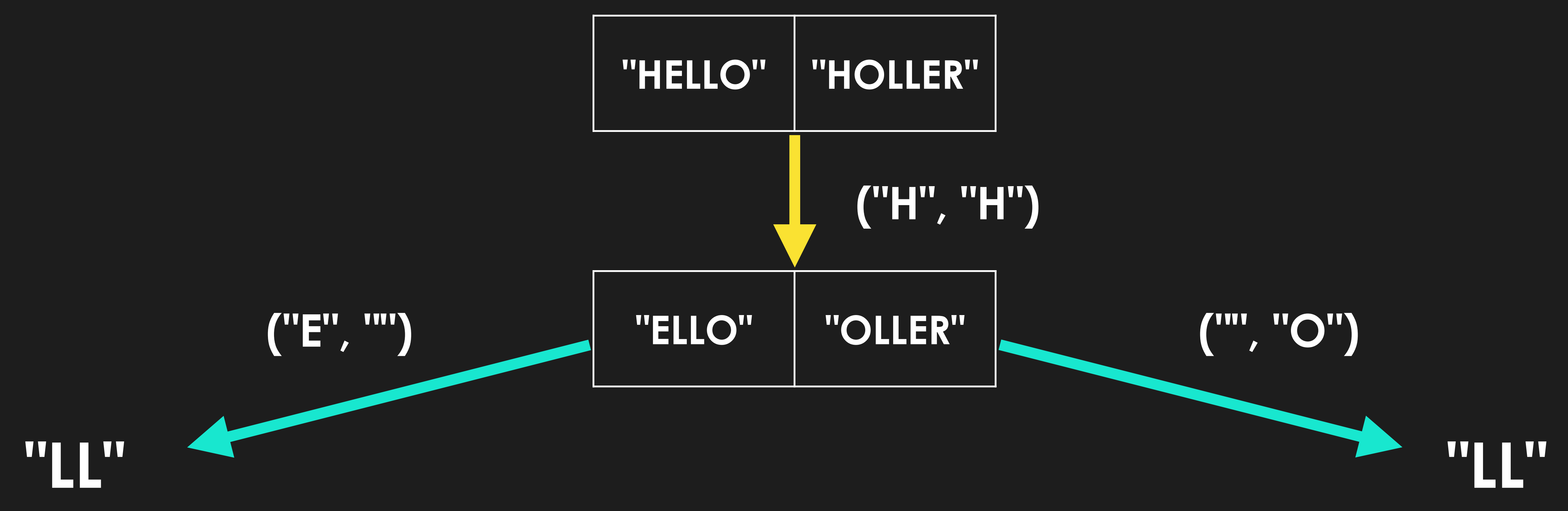
- **case 1**: first char matches
- **case 2**: first char doesn't match



CodeIT

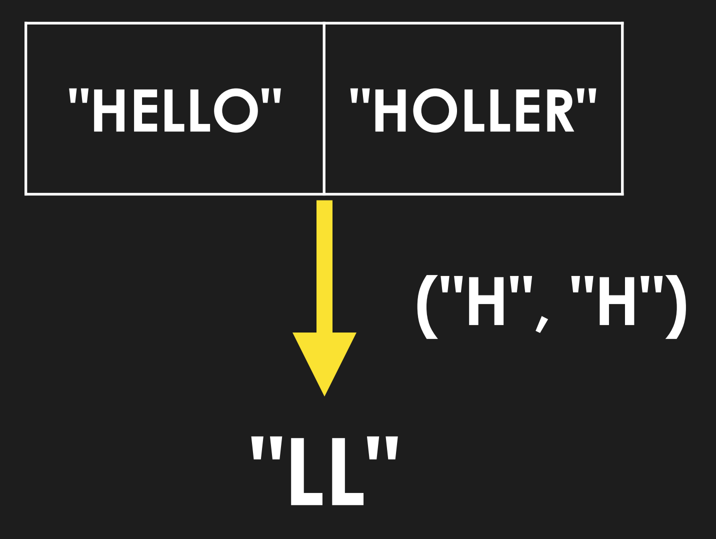
LCS Optimal Substructure

- **case 1**: first char matches
- **case 2**: first char doesn't match



LCS Optimal Substructure

- **case 1**: first char matches
- **case 2**: first char doesn't match



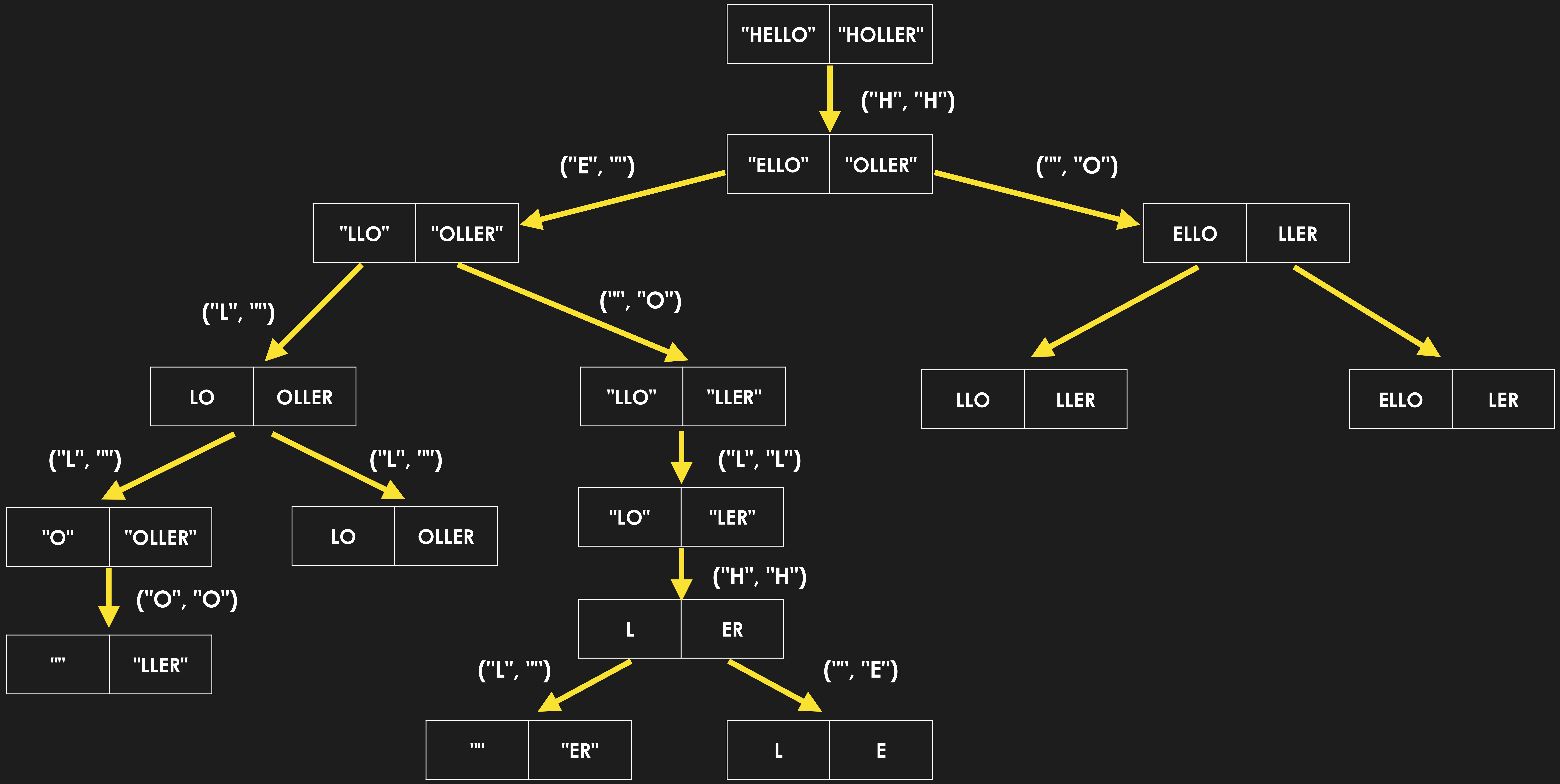
LCS Optimal Substructure

- **case 1**: first char matches
- **case 2**: first char doesn't match

"HLL"

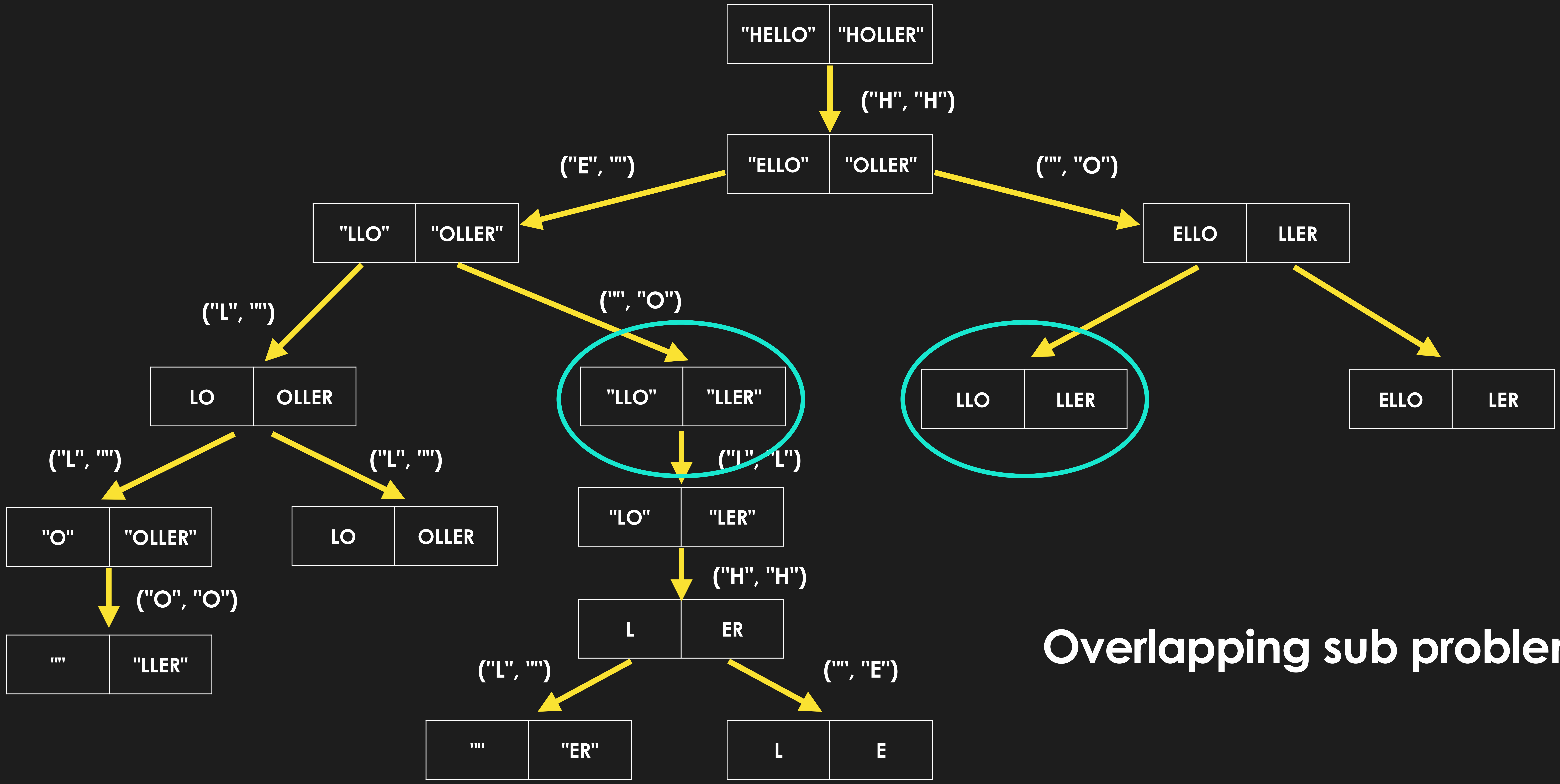
LCS Optimal Substructure

- **case 1:** first char matches
- **case 2:** first char doesn't match



LCS Optimal Substructure

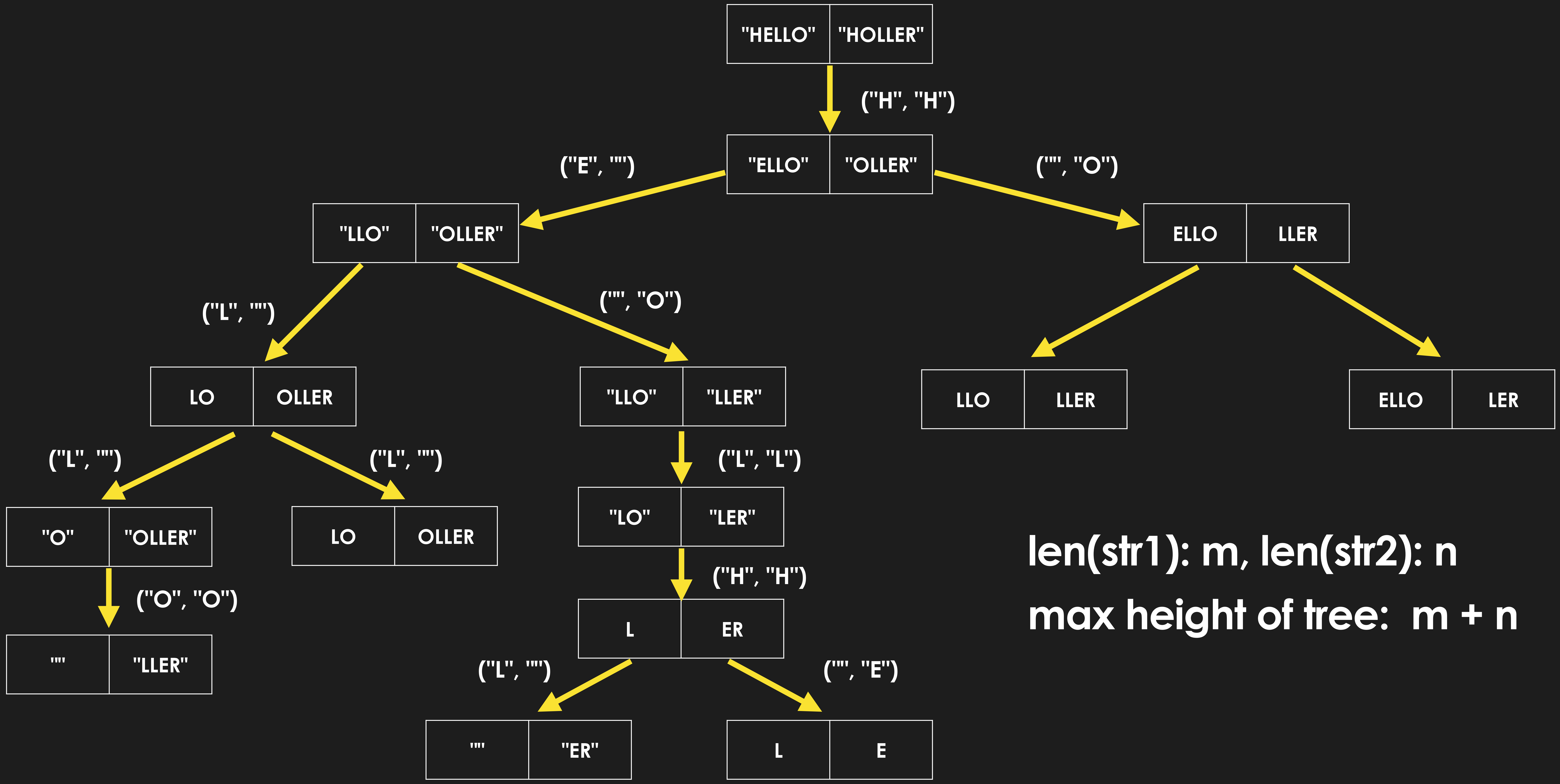
- **case 1:** first char matches
- **case 2:** first char doesn't match



Overlapping sub problems!

LCS Optimal Substructure

- **case 1:** first char matches
- **case 2:** first char doesn't match

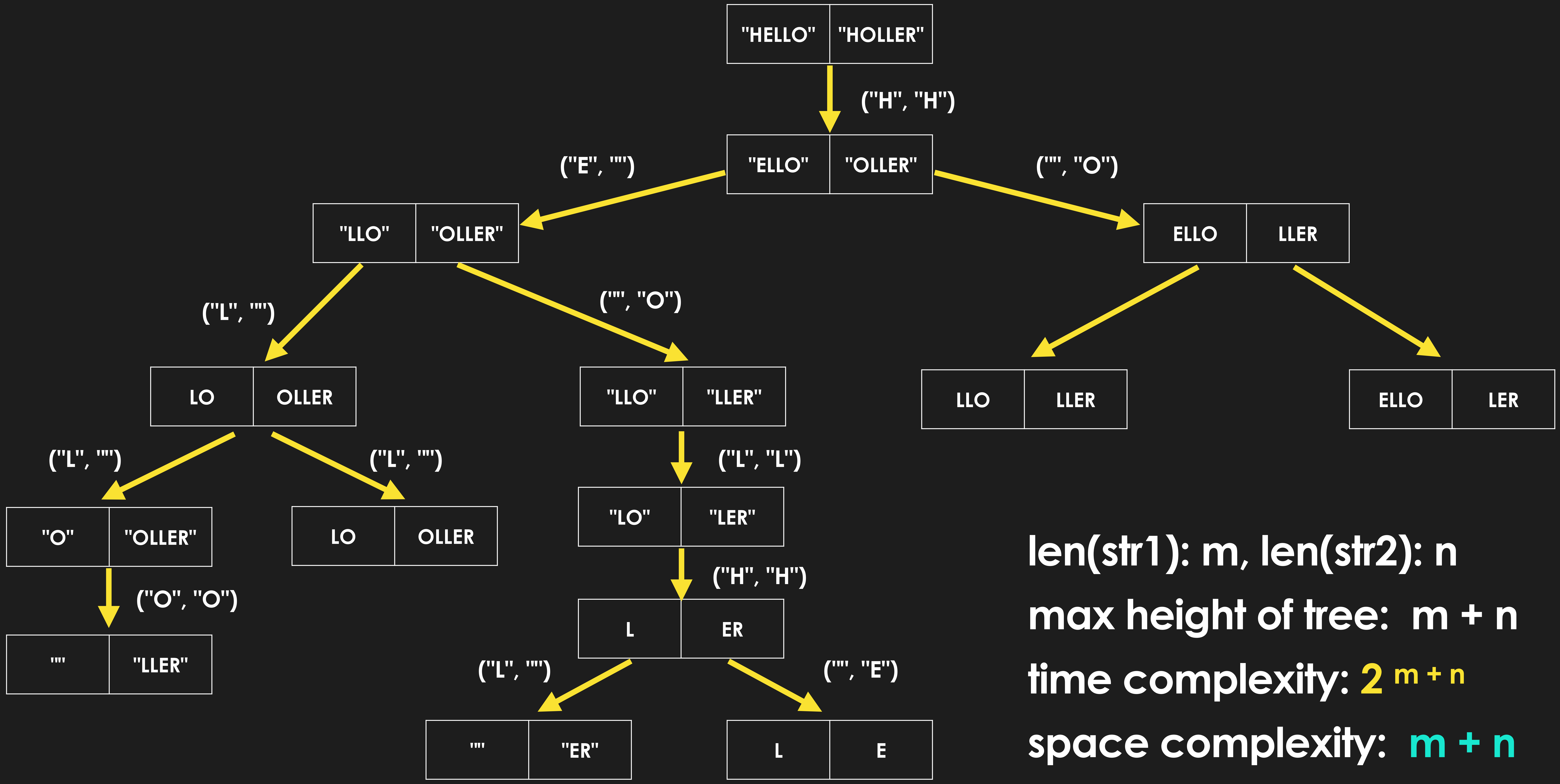


len(str1): m, len(str2): n

max height of tree: m + n

- **case 1:** first char matches
- **case 2:** first char doesn't match

LCS Optimal Substructure



len(str1): m, len(str2): n

max height of tree: $m + n$

time complexity: 2^{m+n}

space complexity: $m + n$

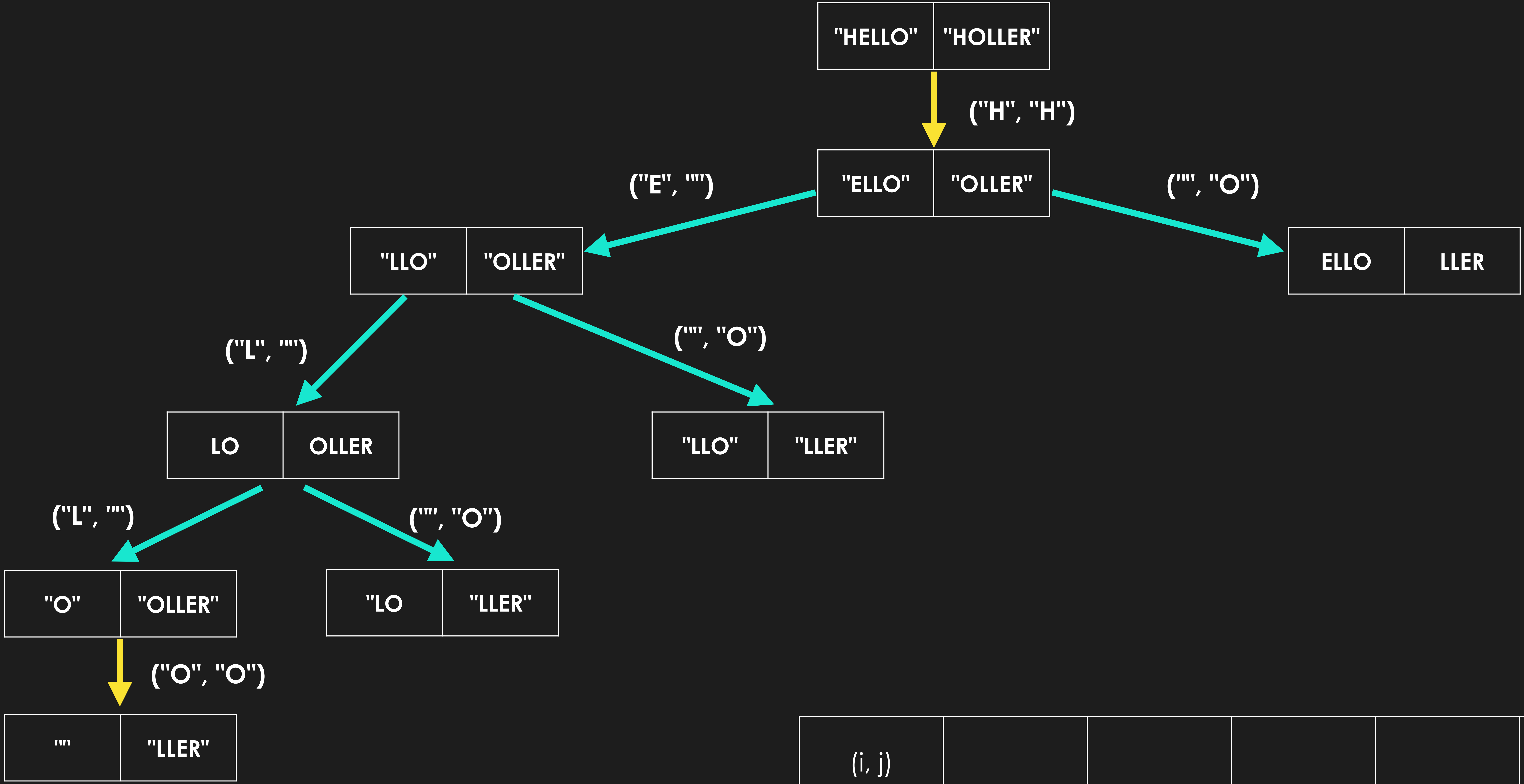
CodeIT

LCS

```python
def lcsRecurse(string1, string2, i, j):
    if i >= len(string1) or j >= len(string2):
        return ""

    if string1[i] == string2[j]:
        res = lcsRecurse(string1, string2, i + 1, j + 1)
        return string1[i] + res

    else:
        res1 = lcsRecurse(string1, string2, i + 1, j)
        res2 = lcsRecurse(string1, string2, i, j + 1)
        if len(res1) > len(res2):
            return res1
        else:
            return res2

def longestCommonSubsequence(string1, string2):
    print(lcsRecurse(string1, string2, 0, 0))
```

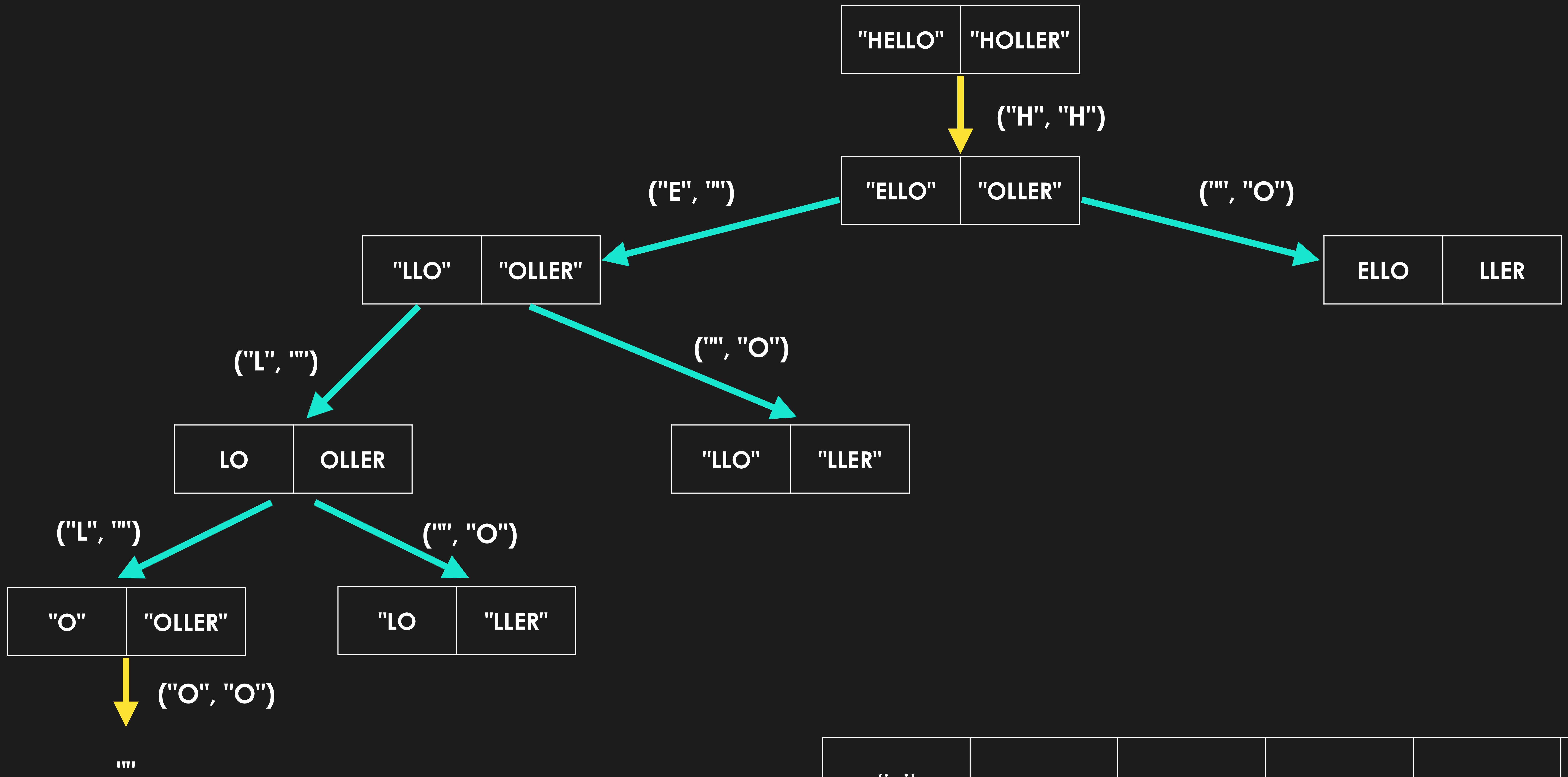
Memoization!

LCS Memo



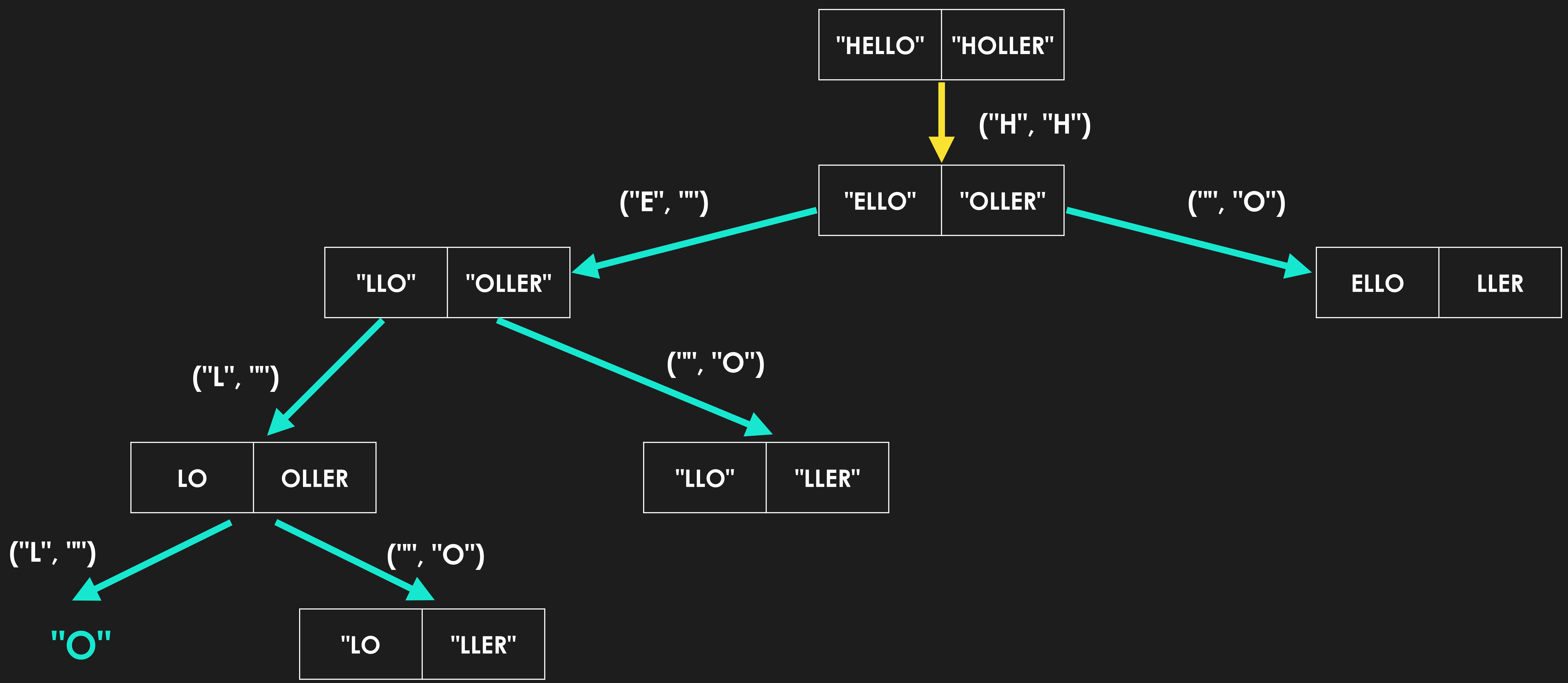
(i, j)						
mem						

LCS Memo



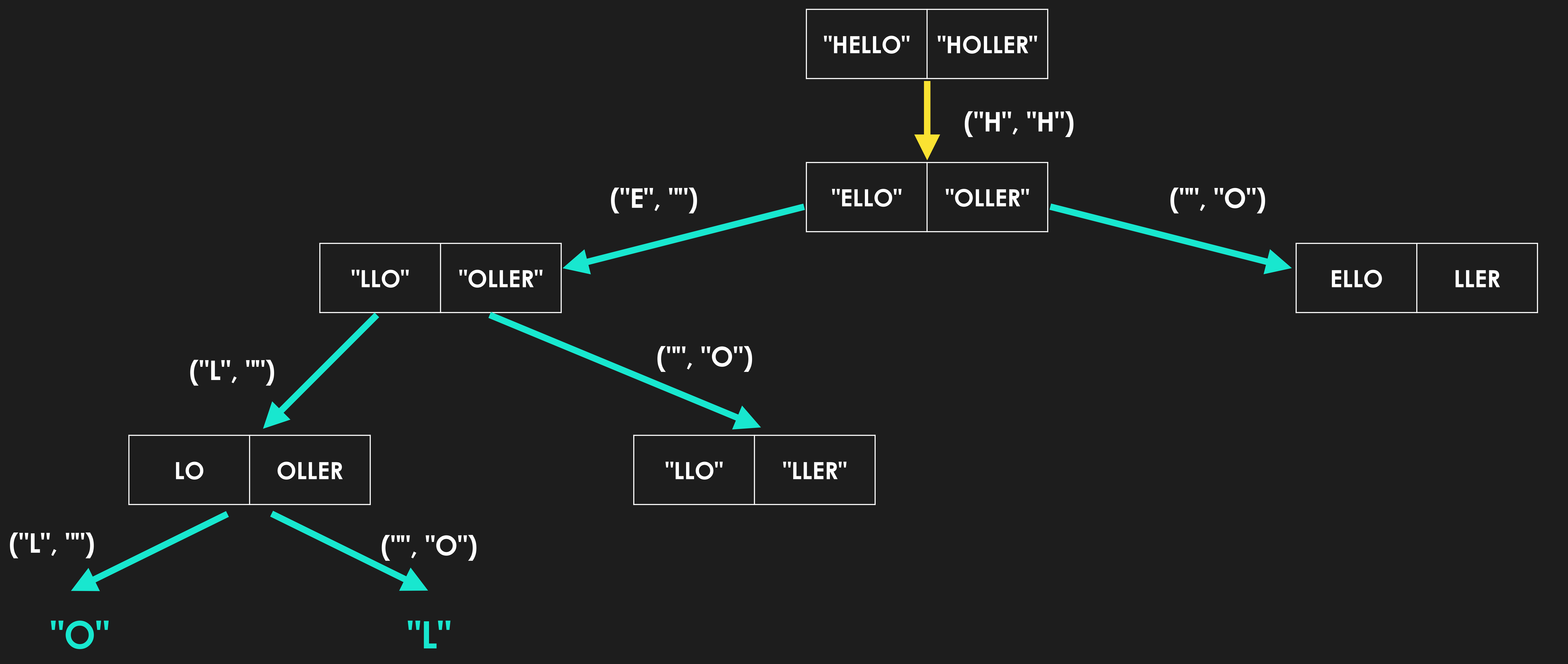
(i, j)						
mem						

LCS Memo



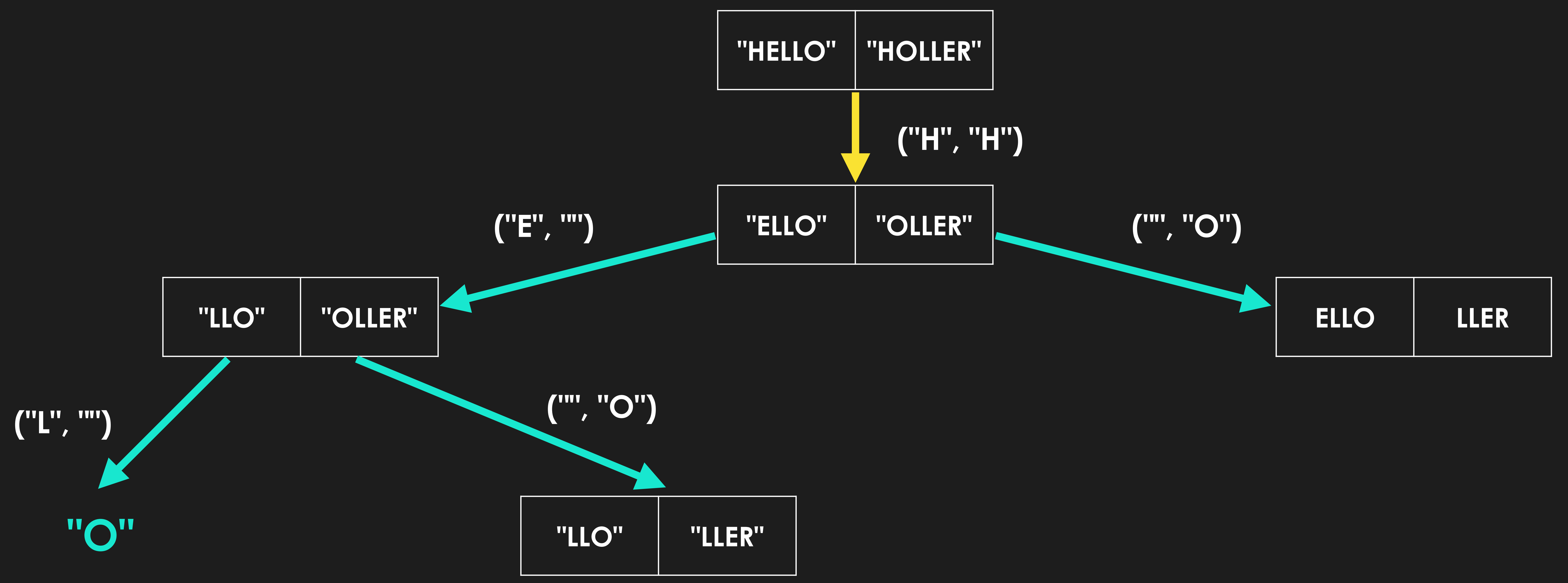
(i, j)	("O", "OLLER")					
mem	"O"					

LCS Memo



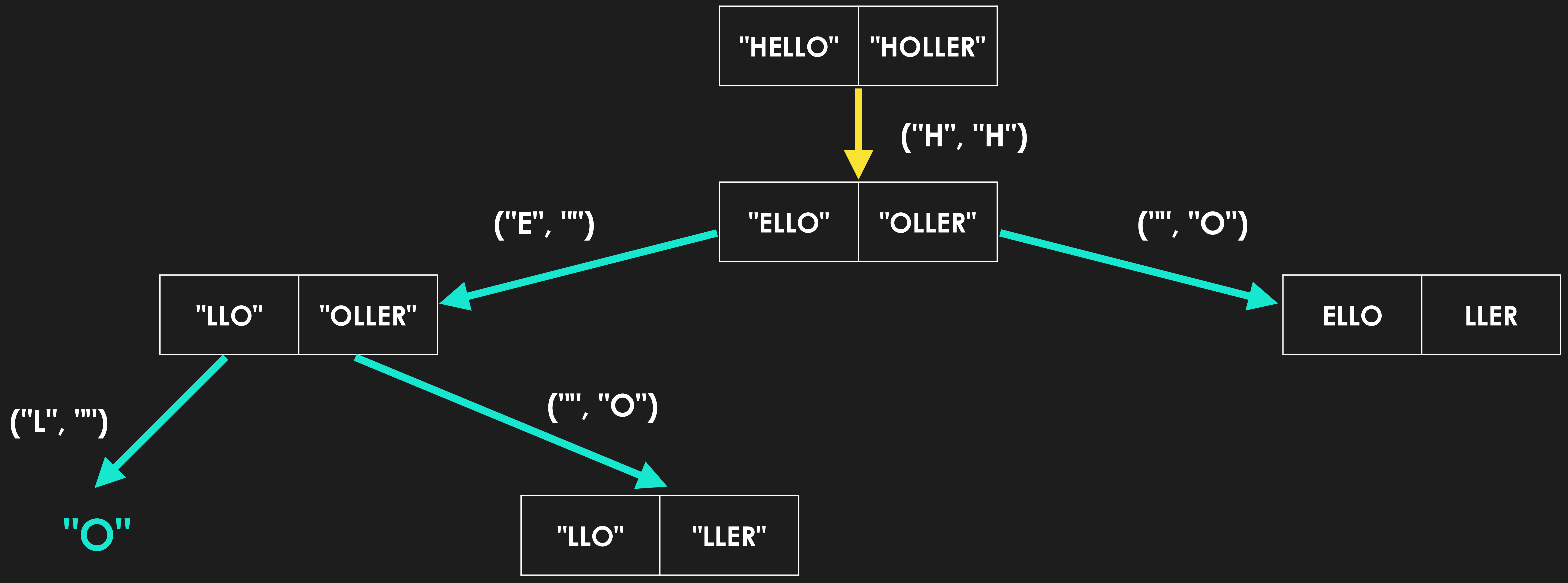
(i, j)	("O", "OLLER")	("LO", "LLER")				
mem	"O"	"L"				

LCS Memo



(i, j)	("O", "OLLER")	("LO", "LLER")	("LO", "OLLER")			
mem	"O"	"L"	"O"			

LCS Memo



time complexity: m * n

space complexity: m + n

(i, j)	("O", "OLLER")	("LO", "LLER")	("LO", "OLLER")			
mem	"O"	"L"	"O"			

LCSMemo

```python
def longestCommonSubsequenceMemo(string1, string2):
    memo = {}
    return lcsMemoRecurse(string1, string2, 0, 0, memo)
```

LCSMemo

```python
def lcsMemoRecurse(string1, string2, i, j, memo):
    if (i, j) in memo:
        return memo[(i, j)]

    if i >= len(string1) or j >= len(string2):
        return ""

    if string1[i] == string2[j]:
        sub = lcsMemoRecurse(string1, string2, i + 1, j + 1, memo)
        memo[(i, j)] = string1[i] + sub
        return string1[i] + sub

    else:
        sub1 = lcsMemoRecurse(string1, string2, i + 1, j, memo)
        sub2 = lcsMemoRecurse(string1, string2, i, j + 1, memo)
        result = sub1 if len(sub1) > len(sub2) else sub2
        memo[(i, j)] = result
        return result
```

Takeaways from memoization

1. Memoization reduces time complexity from **exponential to polynomial**

2. The memoized time complexity of a recursive implementation becomes the **height of the tree**

Lab Session 1

Lab Session 1

- In this lab session, you will be implementing **memoization.py**
- Your task is to implement **the following functions:**

1. **fib:**
   - Takes in an int n and returns the nth number in the fibonacci sequence
   - This function should run in **O(N) time**

2. **bestSum:**
   - Takes in an array of integers and a target sum
   - Should return an array of any combination of integers included in the array, with repeats allowed, which sum to the target
   - This array should be the of the smallest size possible, or empty if there is no solution
   - Should run in **$O(NM^2)$ time**, where N is the length of the array and M is the target sum

Lab Session 1

3. longestCommonSubsequence:
   - Takes in two strings string1 & string2
   - Returns the longest common subsequence between string1 & string2 as a string, or an empty string if there is none
   - Should run in **O(NM) time**

- All three problems should be done using the memoization method (recursion)
- All problems take in a memo object as an argument. You may assume the memo is empty for the first recursive call of the function
- To test the problems, run **`python utils/mem_test.py`**

Solution: fib

```
def fib(n: int, memo: Dict):

    if n in memo:
        return memo[n]

    if n == 0 or n == 1:
        return 1

    else:
        memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
        return fib(n - 1, memo) + fib(n - 2, memo)
```

Solution: bestSum

```python
def bestSum(array: List, targetSum: int, memo: Dict):

    if targetSum in memo:
        return memo[targetSum]

    if targetSum == 0:
            return []

    if targetSum < 0:
        return None

    bestArr = None

    for num in array:
        newTarget = targetSum - num
        res = bestSum(array, newTarget, memo)
        if res is not None:
            if bestArr is None or len(res) + 1 < len(bestArr):
                bestArr = res.copy()
                bestArr.append(num)

    memo[targetSum] = bestArr
    return bestArr
```

CodeIT

Solution: longestCommonSubsequence

```python
def lcsMemoRecurse(string1, string2, i, j, memo):
    if (i, j) in memo:
        return memo[(i, j)]

    if i >= len(string1) or j >= len(string2):
        return ""

    if string1[i] == string2[j]:
        sub = lcsMemoRecurse(string1, string2, i + 1, j + 1, memo)
        memo[(i, j)] = string1[i] + sub
        return string1[i] + sub

    else:
        sub1 = lcsMemoRecurse(string1, string2, i + 1, j, memo)
        sub2 = lcsMemoRecurse(string1, string2, i, j + 1, memo)
        result = sub1 if len(sub1) > len(sub2) else sub2
        memo[(i, j)] = result
        return result

def longestCommonSubsequence(string1: str, string2: str, memo: Dict):
    return lcsMemoRecurse(string1, string2, 0, 0, memo)
```

Tabulation: Bottom Up approach

CodeIT