Dynamic Programming Part 9

Omkar Deshpande

Agenda – Combinatorial Enumeration on Trees

Unique Binary Search Trees II (95)

All Possible Full Binary Trees (894)

Different Ways to Add Parentheses (241)

95. Unique Binary Search Trees II

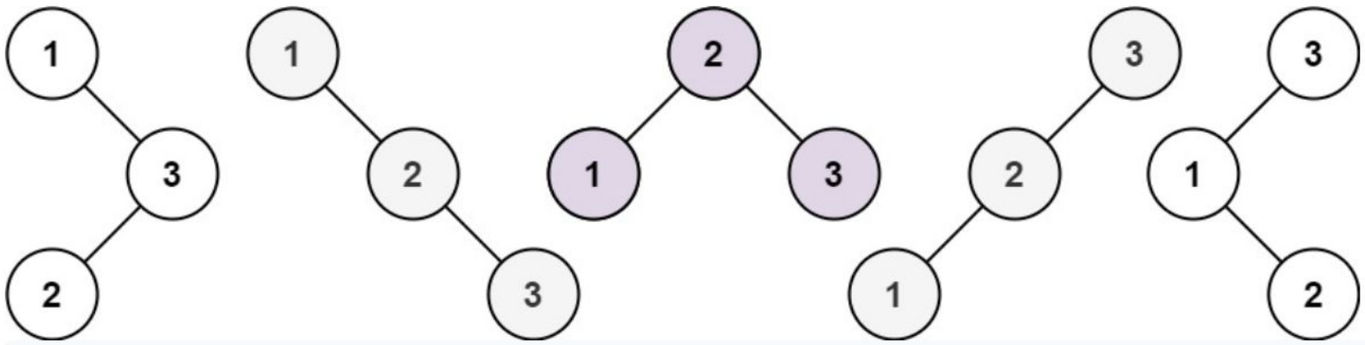
Medium 3817 247 Add to List Share

Given an integer `n`, return *all the structurally unique **BST**'s (binary search trees), which has exactly* `n` *nodes of unique values from* `1` *to* `n`. Return the answer in **any order**.

$$C_n = \frac{\binom{2n}{n}}{n+1}$$

$$C_3 = \frac{\binom{6}{3}}{4} = \frac{6\times5\times4\times1}{3\times2\times1\times4} = 5$$

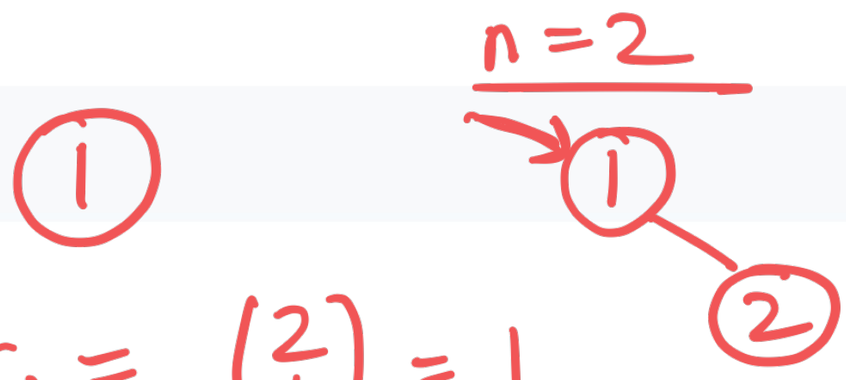
Example 1:




```
Input: n = 3
Output: [[1,null,2,null,3],[1,null,3,2],[2,1,3],[3,1,null,null,2],[3,2,null,1]]
```

Example 2:

```
Input: n = 1
Output: [[1]]
```



$$C_1 = \frac{\binom{2}{1}}{2} = 1$$

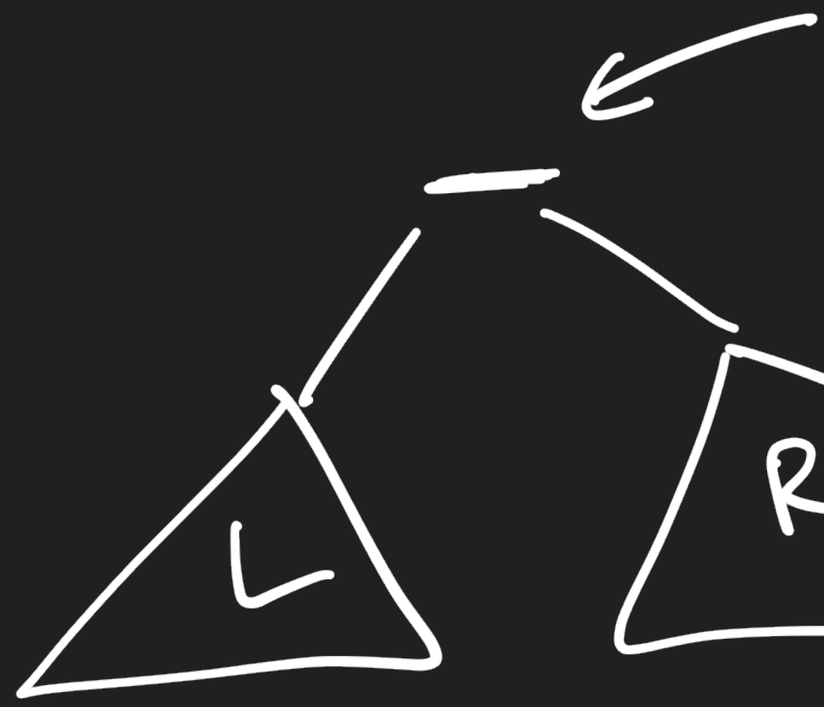
$$C_2 = \frac{\binom{4}{2}}{3} = \frac{6}{3} = 2$$

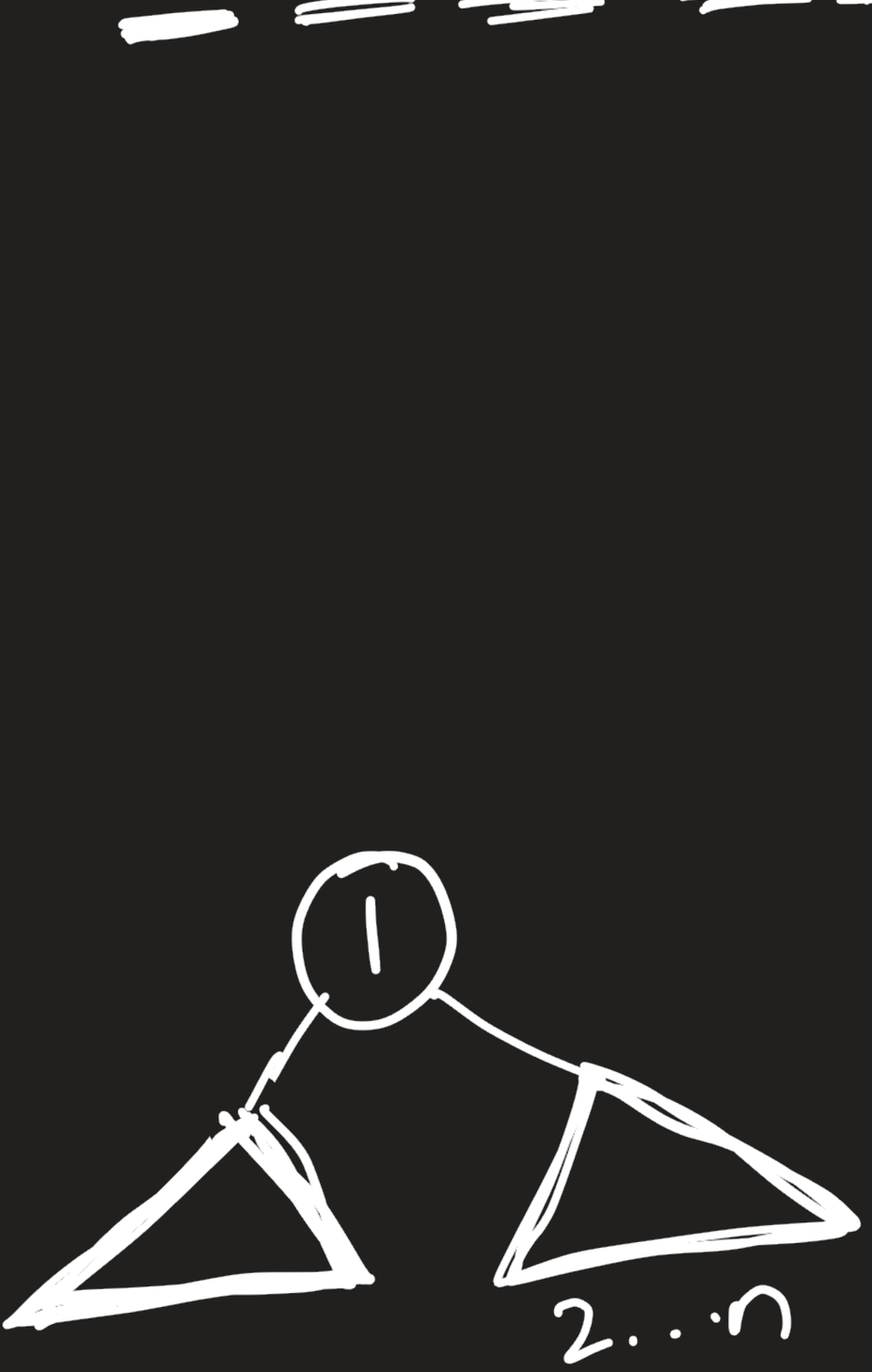
Constraints:

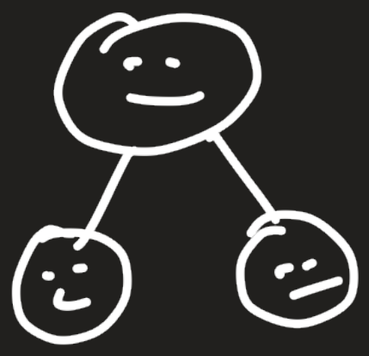
- `1 <= n <= 8`

$S = [1, \ldots, n]$

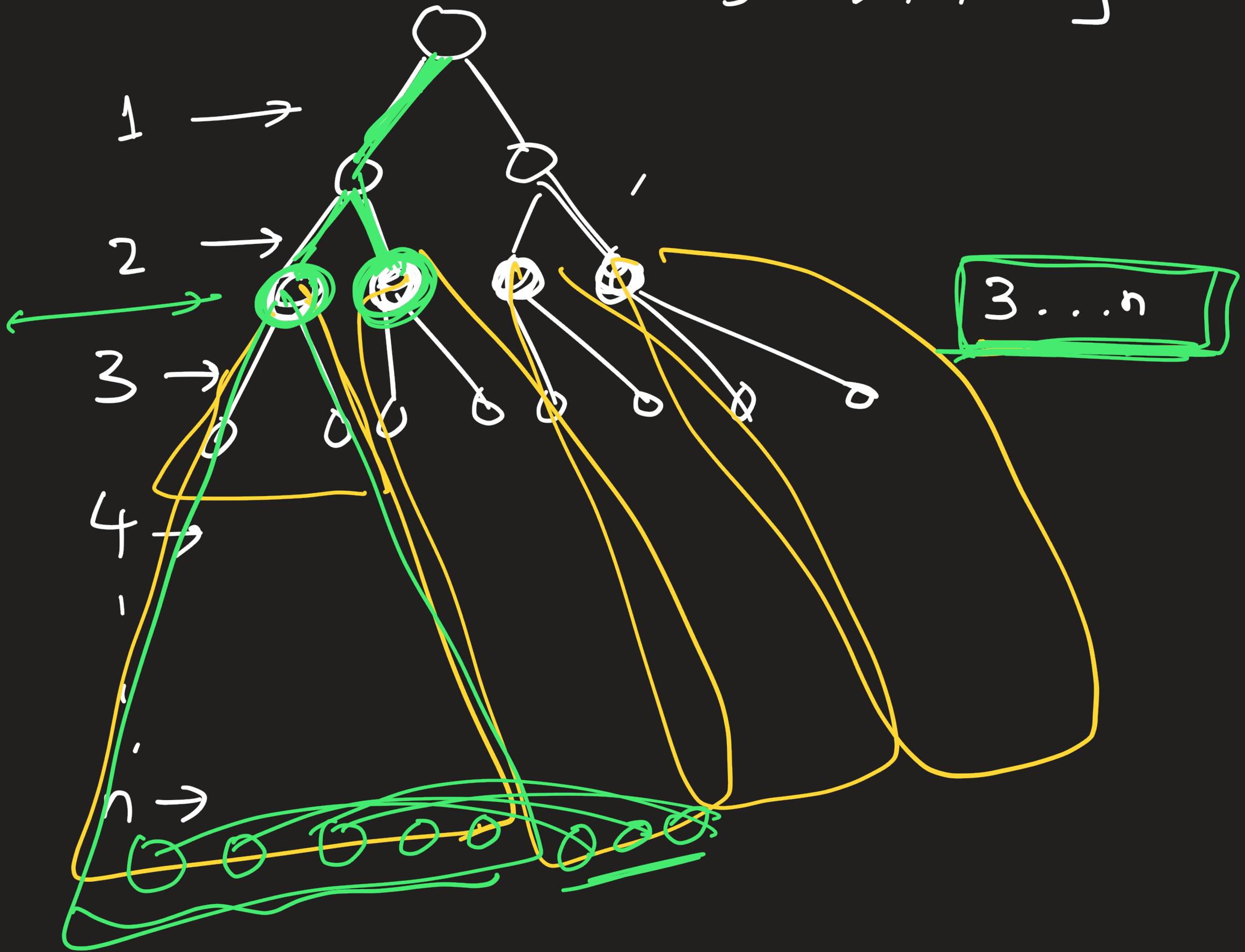
b

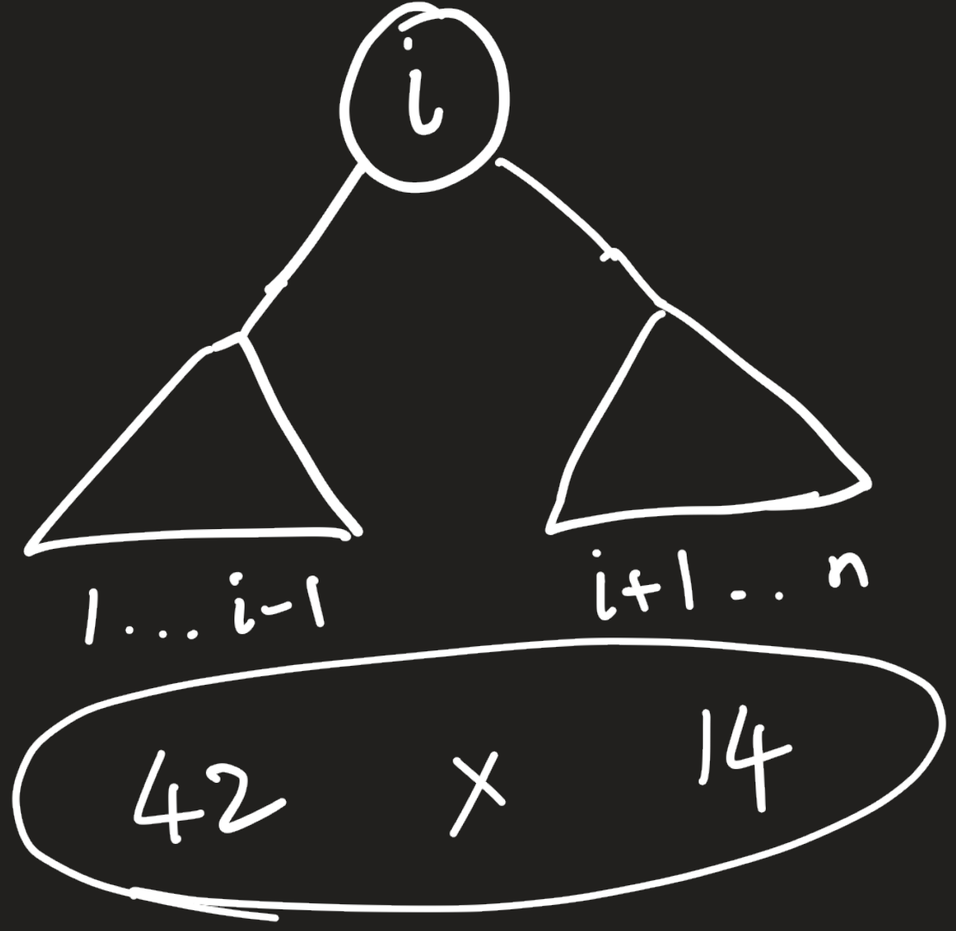


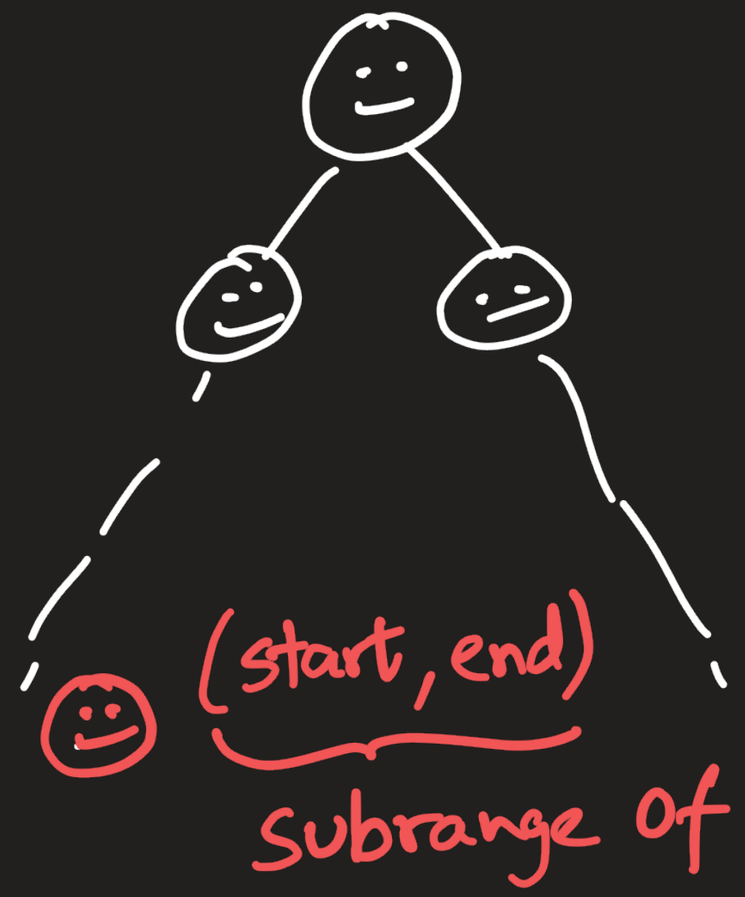




$S = [1, 2, \ldots n]$







TreeNode
- left (TreeNode)
- right (TreeNode)
- val (int)

☺ subproblem defn
1 or 0.

[null] ≠ []

size 1 size 0

Root manager:

return helper(1, n)

$\underbrace{1 \ldots i-1}_{i-1}, i, \underbrace{i+1 \ldots n}_{n-(i+1)+1 = n-i}$

$$f(n) = \sum_{i=1}^{n} f(i-1) * f(n-i)$$

↳ n^{th} catalan number

T(n) will be exponential in n

Size of result list = n^{th} Catalan number
↳ exponential in n.

memo = empty hash table

function helper (start, end) : // returns a list of trees built out of [start ... end]

Memoization Case:

if (start, end) is in memo:
return memo[(start, end)]

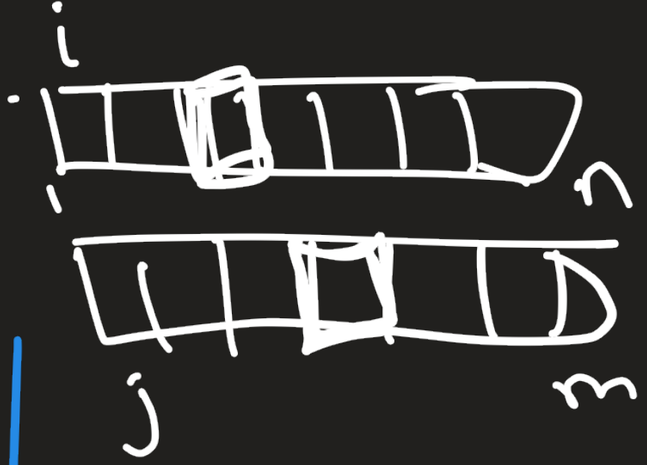
Base Case
if start > end : // subproblem size 0
return [null]
if start == end : // subproblem size 1
return [new TreeNode (start)]

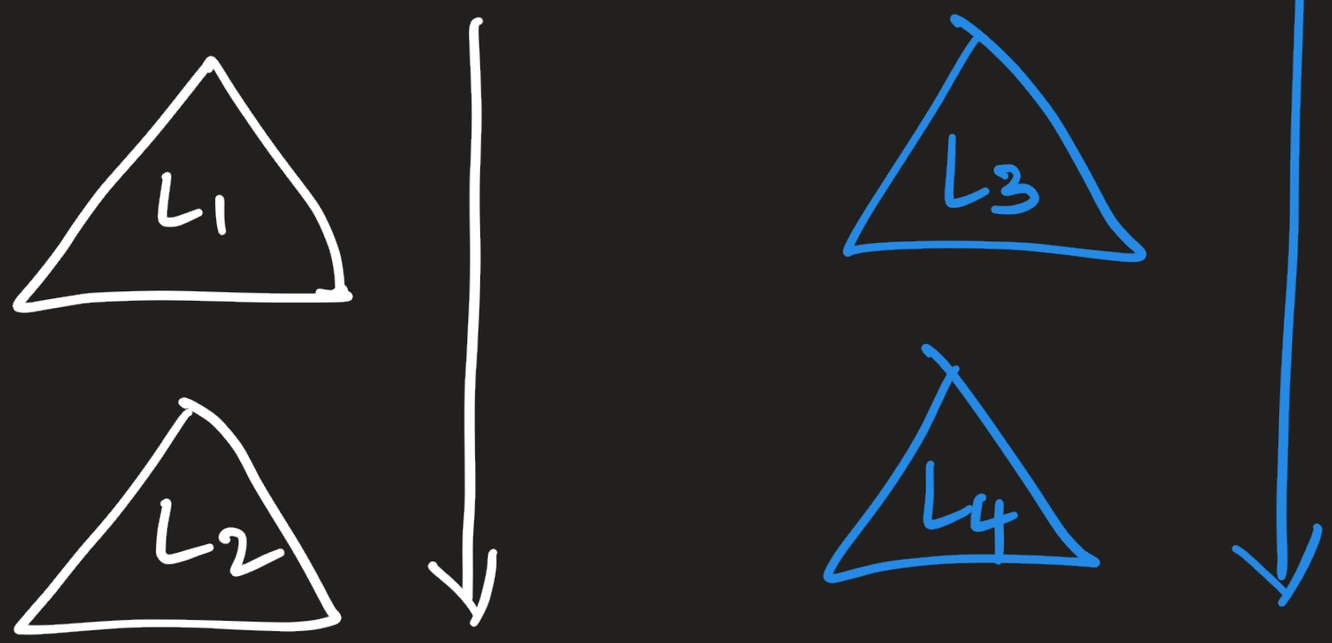
# Recursive Case // subproblem size > 1
result = []
for r in start to end :
// Pick r as the root
leftsubtrees ← helper (start, r-1)
rightsubtrees ← helper (r+1, end)
for lst in leftsubtrees :
for rst in rightsubtrees :
root = new TreeNode (r)
root.left = lst , root.right = rst
result.append (root)
return result

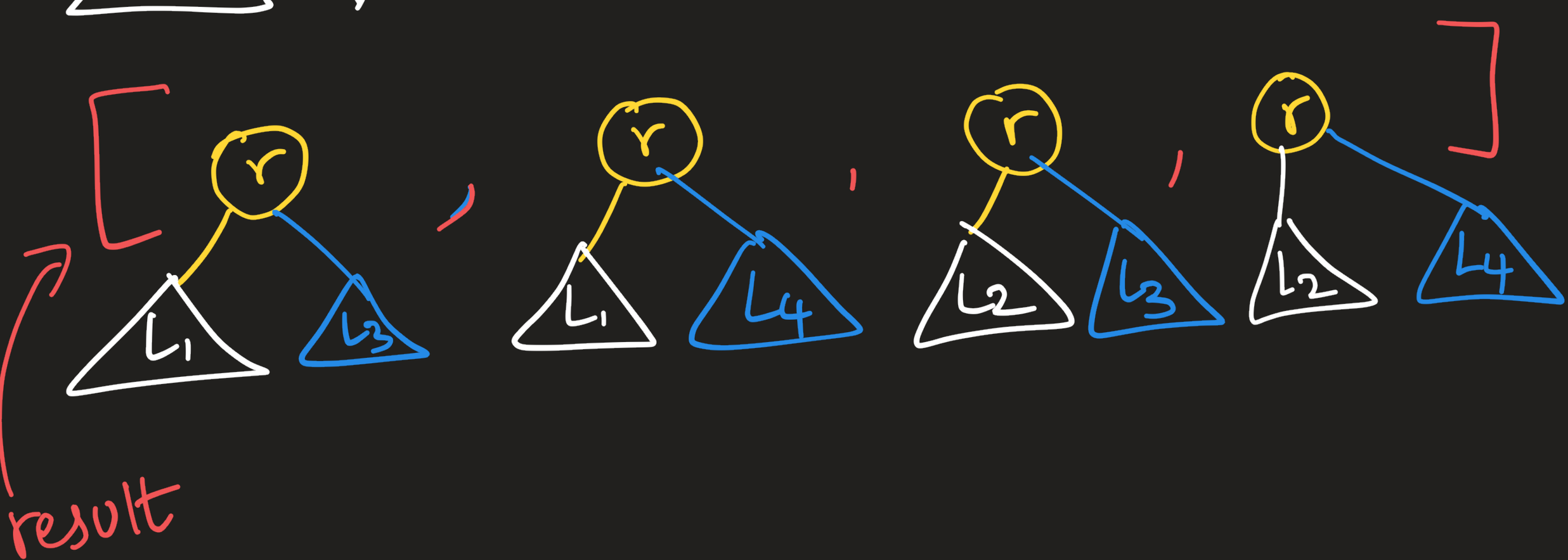
memo[(start, end)] = result

K
K==0
K==1
K>1

start+1 to end-1
2 to K-1
r-1
K-r
T(i)
T(n-i)
f(i) * f(n-i)
0


i
n
j
m


L1
L2
L3
L4

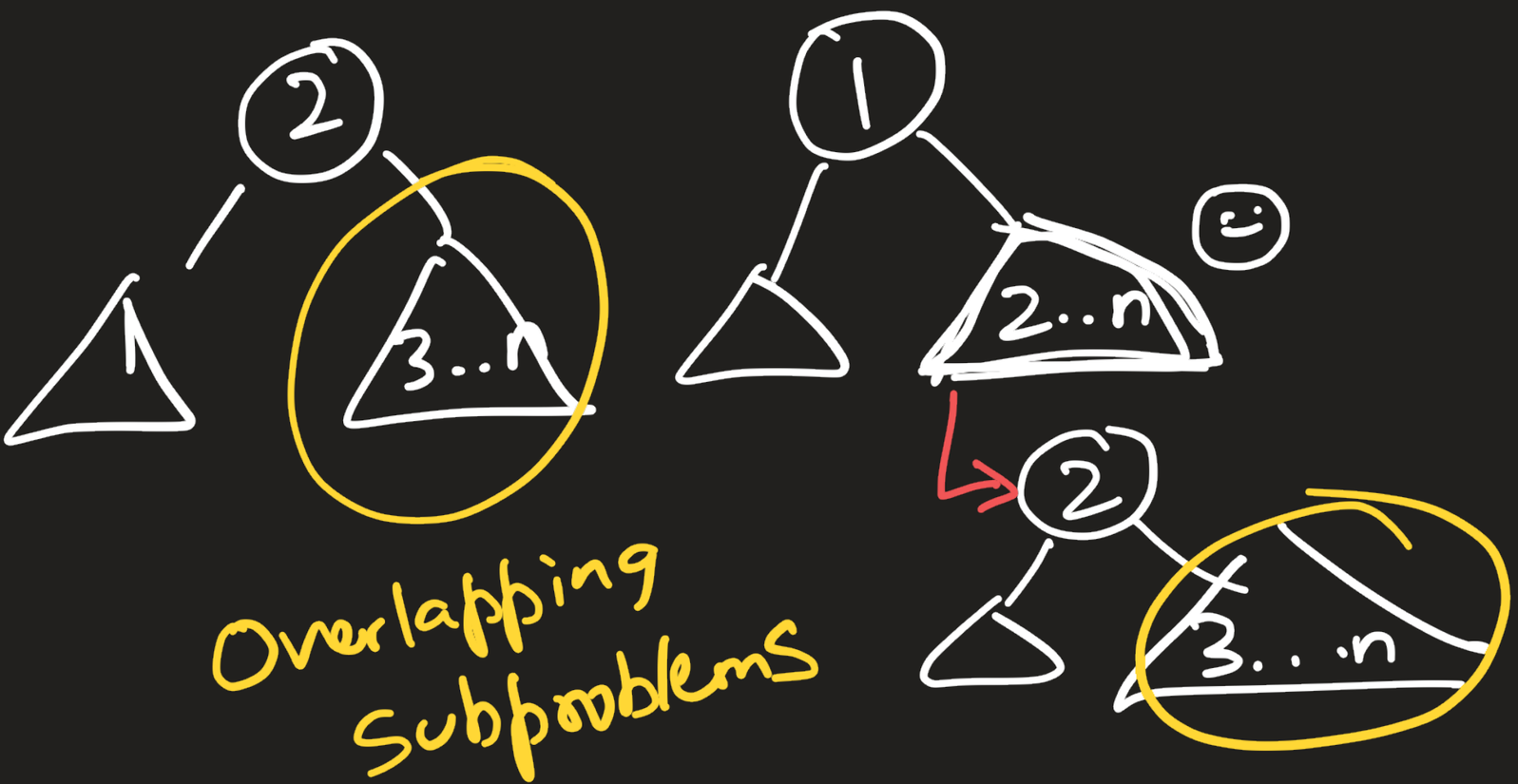

r
L1
L3
r
L1
L4
r
L2
L3
r
L2
L4
result

Start and end can take n distinct values each

$$\boxed{{}^{n}C_{2}} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$$

$O(n^2)$

$(1 \ldots n)$



$(3, n) \rightarrow$ 

Problem size ~~must~~ be odd, say $2n+1$

```python
    def generateTrees(self, n):
        """
        :type n: int
        :rtype: List[TreeNode]
        """

        def helper(start, end):

            #Base case
            if start > end:
                return [None]
            if start == end:
                return [TreeNode(start)]

            #Recursive case
            result = []
            for r in range(start, end+1):
                leftsubtrees = helper(start, r-1)
                rightsubtrees = helper(r+1, end)
                for lt in leftsubtrees:
                    for rt in rightsubtrees:
                        root = TreeNode(r)
                        root.left = lt
                        root.right = rt
                        result.append(root)

            return result

        return helper(1, n)
```

```python
    def generateTrees(self, n):
        """
        :type n: int
        :rtype: List[TreeNode]
        """

        memo = {}

        def helper(start, end):
            #Memoization case
            if (start, end) in memo:
                return memo[(start, end)]

            #Base case
            if start > end:
                return [None]
            if start == end:
                return [TreeNode(start)]

            #Recursive case
            result = []
            for r in range(start, end+1):
                leftsubtrees = helper(start, r-1)
                rightsubtrees = helper(r+1, end)
                for lt in leftsubtrees:
                    for rt in rightsubtrees:
                        root = TreeNode(r)
                        root.left = lt
                        root.right = rt
                        result.append(root)

            memo[(start, end)] = result
            return result

        return helper(1, n)
```

894. All Possible Full Binary Trees

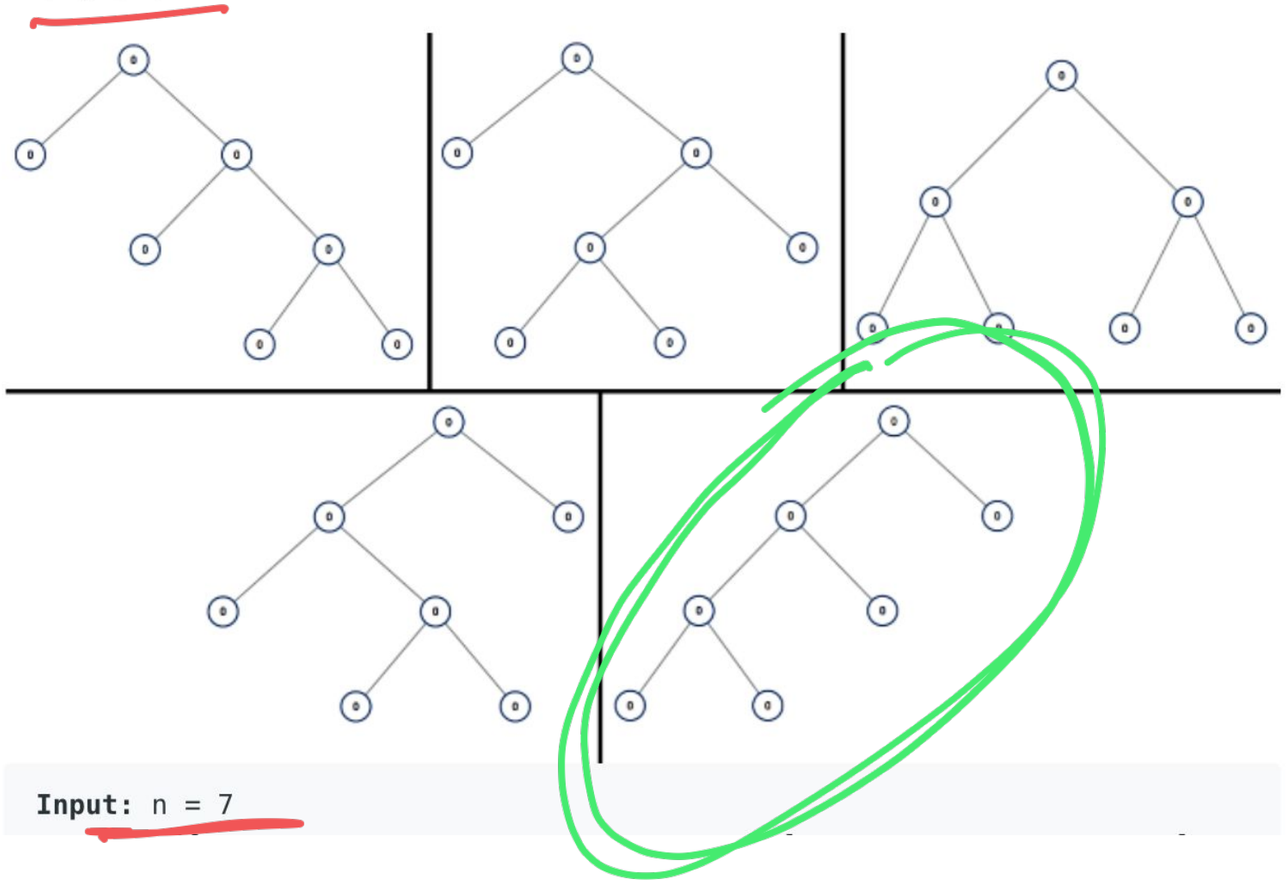
Medium 1861 153 Add to List Share

Given an integer `n`, return *a list of all possible* ***full binary trees*** *with* `n` *nodes*. Each node of each tree in the answer must have `Node.val == 0`.

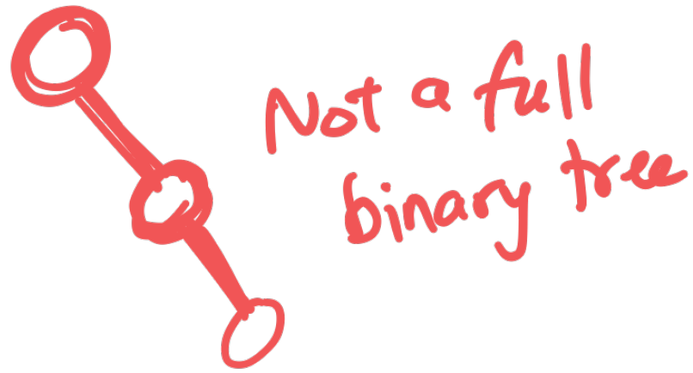
Each element of the answer is the root node of one possible tree. You may return the final list of trees in **any order**.

A **full binary tree** is a binary tree where each node has exactly `0` or `2` children.

Example 1:

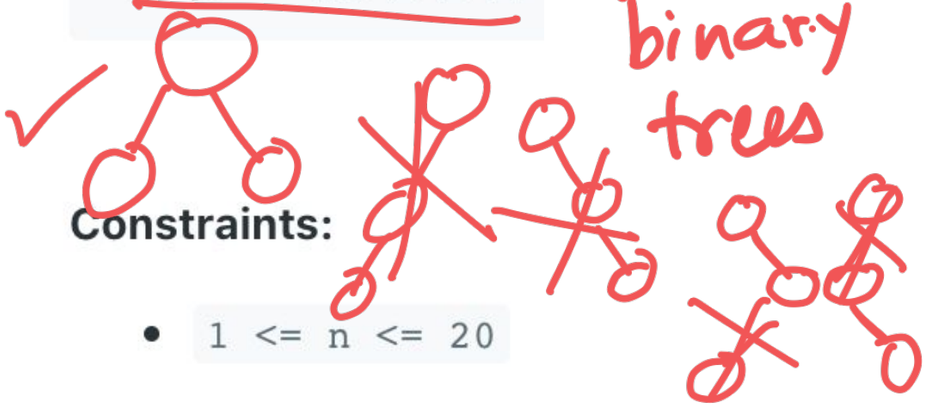


```
Input: n = 7
```



Example 2:

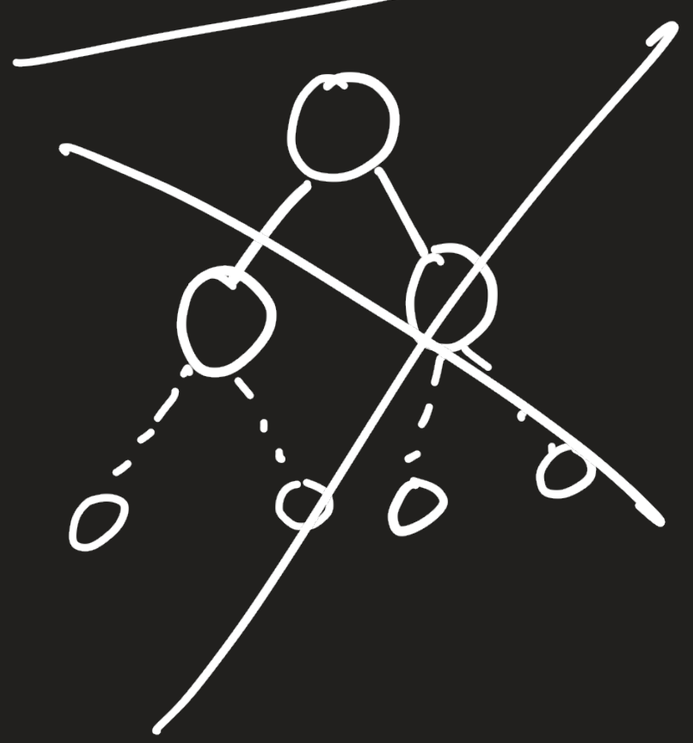
```
Input: n = 3
Output: [[0,0,0]]
```



Constraints:

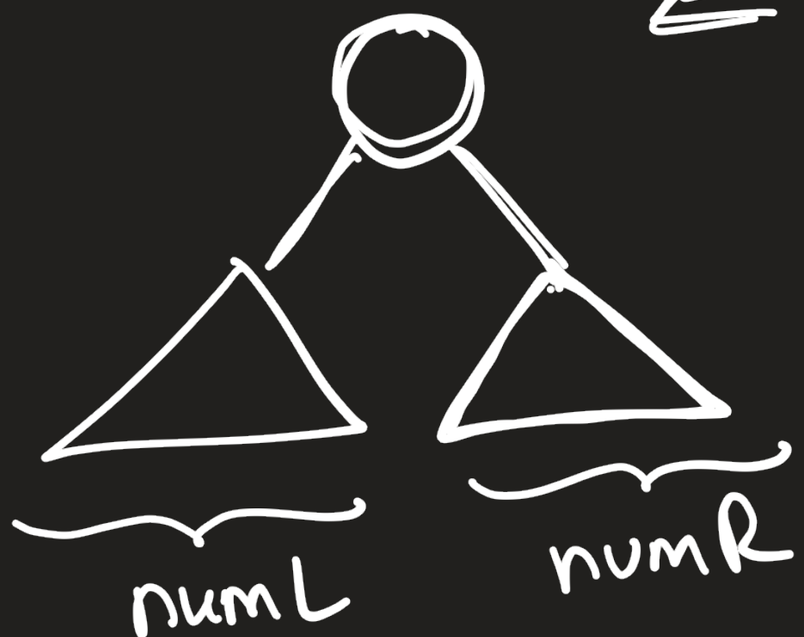
- `1 <= n <= 20`

$n = 4$ → empty list



For any even # nodes,
say $2n$, there cannot
be any full binary trees.

Why?



numL + numR = odd
= 2n-1

Exactly one of numL
or numR must be
an even number smaller
than 2n.

```
if n % 2 == 0:
    return []
```

Problem size must be odd, say $2n+1$

$$f(2n+1) = \sum_{i=1}^{2n-1} f(i) * f(2n-i) \quad \text{(in steps of 2)}$$

$f(2n+1)$: # full binary trees with $2n+1$ nodes

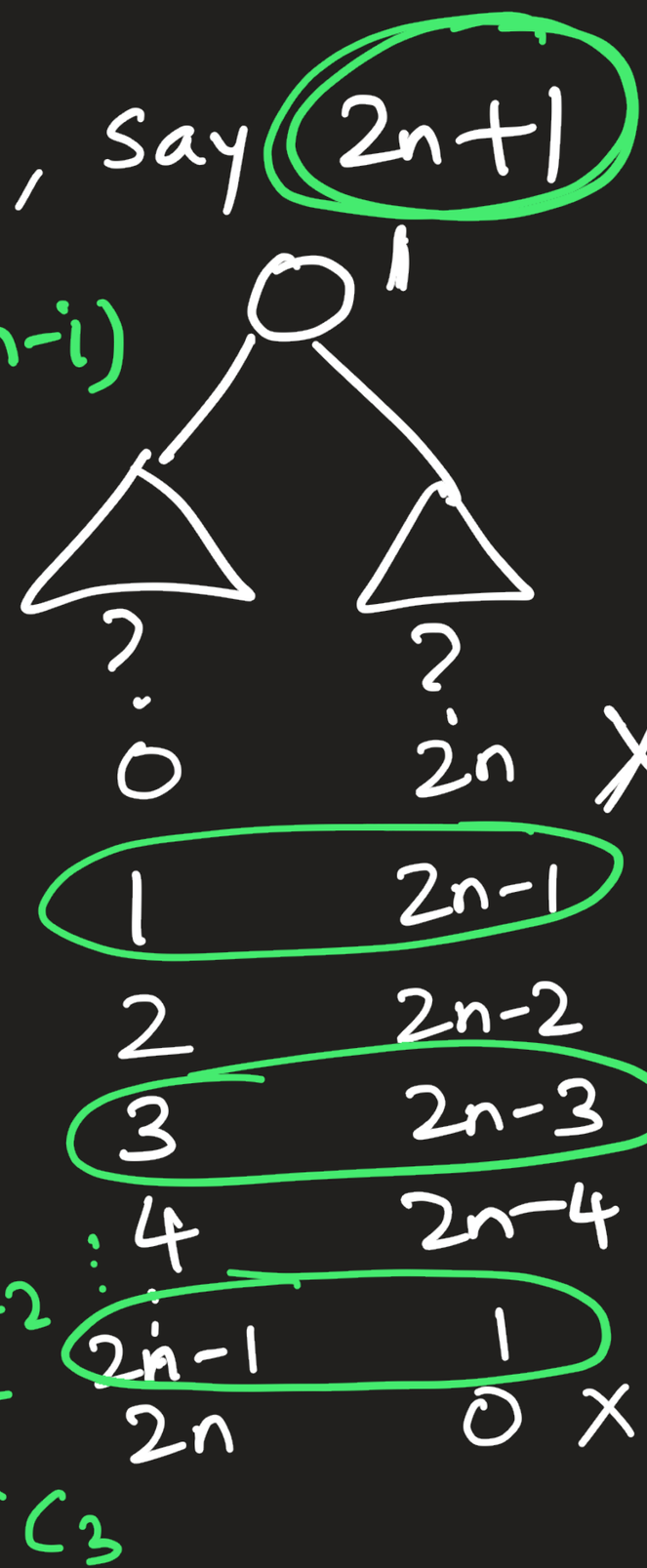
$= f_1 f_{2n-1} + f_3 f_{2n-3} + \cdots + f_{2n-1} f_1$

$f(1) = 1 \leftarrow C_0$

$f(3) = 1 \leftarrow C_1$

$f(5) = f_1 f_3 + f_3 f_1 = 1+1 = 2$ ← C_2

$f(7) = f_1 f_5 + f_3 f_3 + f_5 f_1 = 2+1+2 = 5$ ← C_3



$$f(2n+1) = C_n \quad \text{where } \frac{(2n+1)-1}{2} = n$$

$= n^{th}$ Catalan number

$$2^n < C_n < 4^n$$

$$f(i) = C_{\frac{i-1}{2}}$$

i odd

$$2^{\frac{i-1}{2}} < C_{\frac{i-1}{2}} < 4^{\frac{i-1}{2}}$$

$$(\sqrt{2})^{i-1} < C_{\frac{i-1}{2}} < 2^{i-1}$$

$$1.4^i > 1.4$$

exponential in i

$$\frac{\frac{2n!}{n!n!}}{n+1}$$

```python
    def allPossibleFBT(self, n):
        """
        :type n: int
        :rtype: List[TreeNode]
        """

        memo = {}

        def helper(start, end):
            #Memoization case
            if (start, end) in memo:
                return memo[(start, end)]

            #Base case
            if start > end:
                return [None]
            if start == end:
                return [TreeNode(0)] *

            #Recursive case
            result = []
            for r in range(start+1, end): *
                leftsubtrees = helper(start, r-1)
                rightsubtrees = helper(r+1, end)
                for lt in leftsubtrees:
                    for rt in rightsubtrees:
                        root = TreeNode(0) *
                        root.left = lt
                        root.right = rt
                        result.append(root)

            memo[(start, end)] = result
            return result

        return helper(1, n)
```

```python
        if n % 2 == 0:
            return []

        memo = {}

        def helper(start, end):
            #Memoization case
            if (start, end) in memo:
                return memo[(start, end)]
            elif (end - start + 1) % 2 == 0:
                return []

            #Base case
            if start > end:
                return [None]
            elif start == end:
                return [TreeNode(0)]


            #Recursive case
            result = []
            for r in range(start+1, end):
                Lsubtrees = helper(start, r-1)
                Rsubtrees = helper(r+1, end)
                for lsubtree in Lsubtrees:
                    for rsubtree in Rsubtrees:
                        root = TreeNode(0, lsubtree, rsubtree)
                        result.append(root)

            memo[(start, end)] = result
            return result

        return helper(1, n)
```

241. Different Ways to Add Parentheses

Medium 2633 141 Add to List Share

← an ambiguous mathematical expression

Given a string `expression` of numbers and operators, return *all possible results from computing all the different possible ways to group numbers and operators*. You may return the answer in **any order**.

(2-1)-1 → 0

2-(1-1) → 2

→ from different ways of parenthesizing it

or different parse trees for the same expression

Example 1:

```
Input: expression = "2-1-1"
Output: [0,2]
Explanation:
((2-1)-1) = 0
(2-(1-1)) = 2
```

Example 2:

```
Input: expression = "2*3-4*5"
Output: [-34,-14,-10,-10,10]
Explanation:
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

Constraints:

- `1 <= expression.length <= 20`
- `expression` consists of digits and the operator `'+'`, `'-'`, and `'*'`.

op 1 op 2 op 3 op n

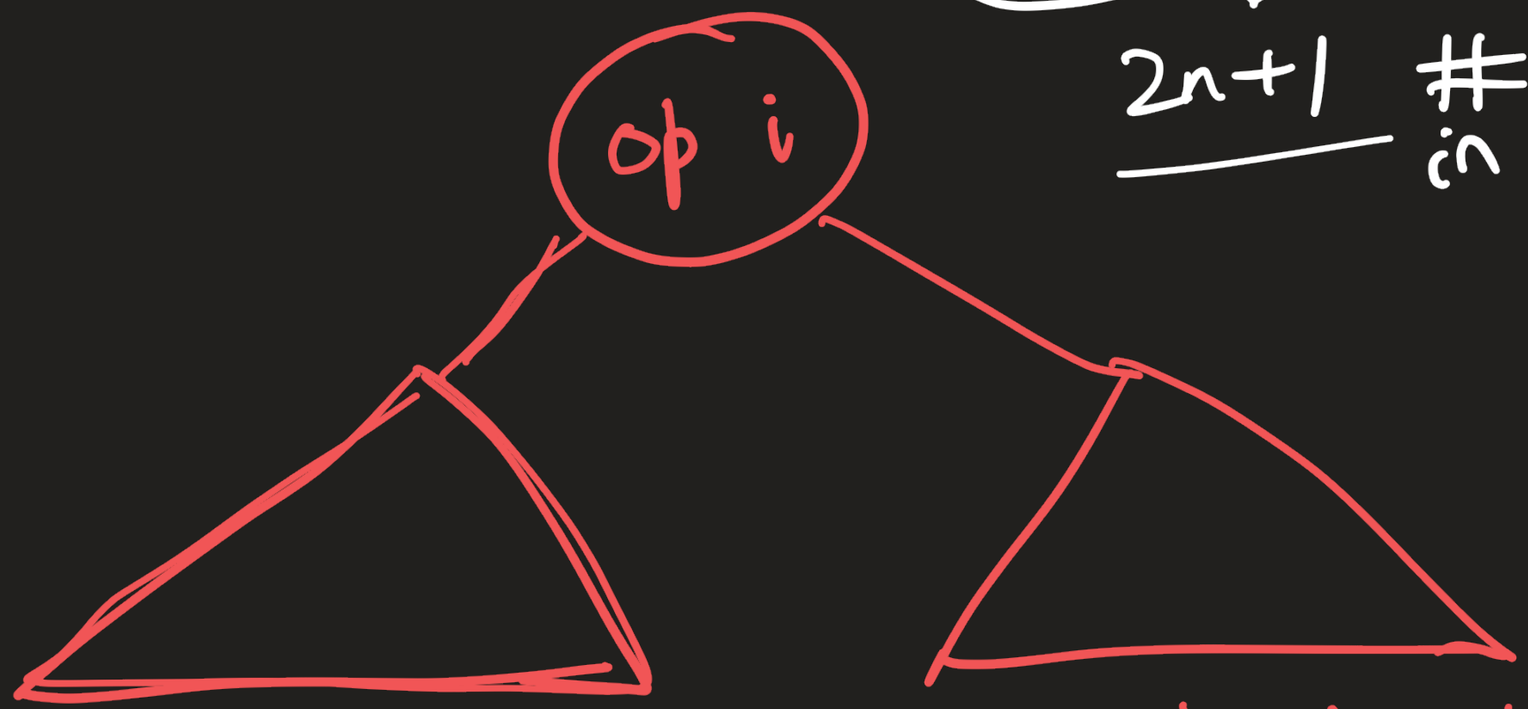
V_1 (+) V_2 (*) V_3 (−) V_4 (-) - - - (-) V_{n+1}

↑

n operators ⟶ n internal nodes

n+1 numbers ⟶ n+1 leaves

2n+1 # nodes in total



$$f(n) = \sum_{i=1}^{n} f(i-1) * f(n-i) = n^{th} \text{ Catalan number}$$

parse trees is exponential in n.

All parse trees will be full binary trees with operators as internal nodes with 2 children $+, -, *$ and numbers as leaves with 0 children.

Substring S[i...j] is the subproblem

```
function helper(S, i, j): // Build & evaluate the results of all possible parse trees & return result list
    # Base Case: No operators
    if S[i..j].isdigit(): // purely numerical
        return [int(S[i..j])]
    # Recursive case: ≥ 1 operator within the substring
    result = []
    for idx in i to j:
        if not S[idx].isdigit(): // operator
            Lresults ← helper(S, i, idx-1)
            Rresults ← helper(S, idx+1, j)
            for Lval in Lresults:
                for Rval in Rresults:
                    if S[idx] == "+":
                        result.append(Lval + Rval)
                    else if S[idx] == "-":
                        result.append(Lval - Rval)
                    else:
                        result.append(Lval * Rval)
    return result
```

return helper(expression, 0, length(expression) - 1)

$T(n) = O(n^{th}$ Catalan number$)$

#operators

exponential in n

Space: Size of results list = n^{th} Catalan number

exponential in n

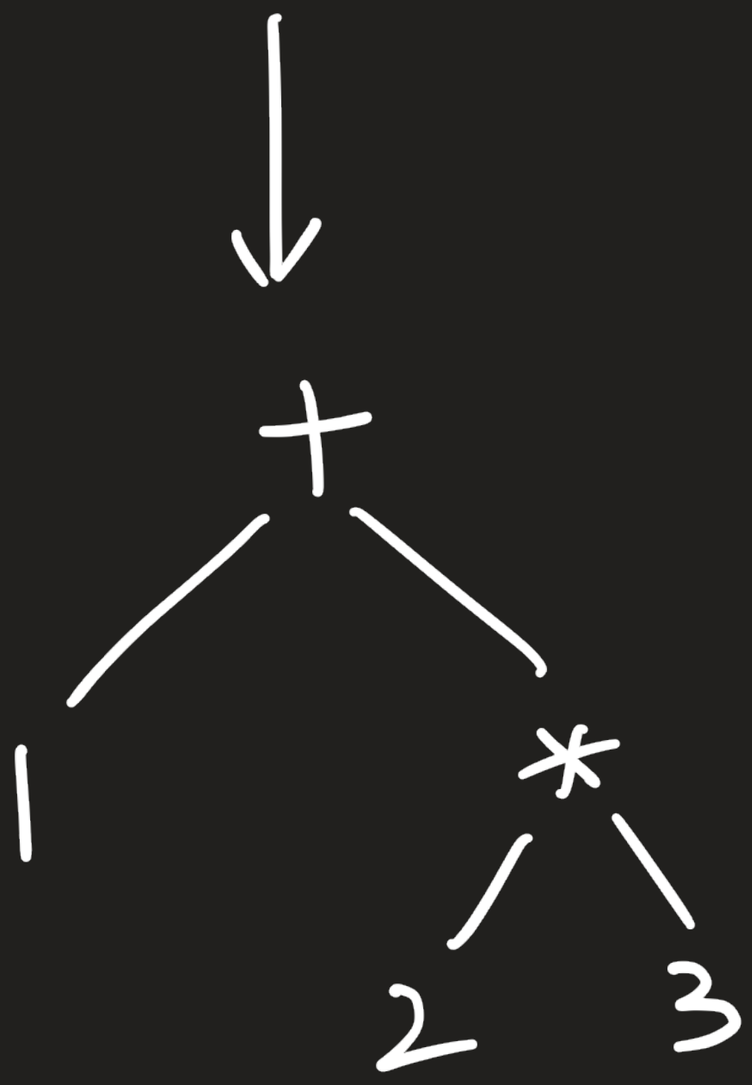
The Syntactic structure of a mathematical expression can be visualized as a parse tree / syntax tree.

A mathematical expression E is

- → A number
- → (E + E)
- → (E - E)
- → (E * E)
- → (E / E)

} Fully parenthesized notation

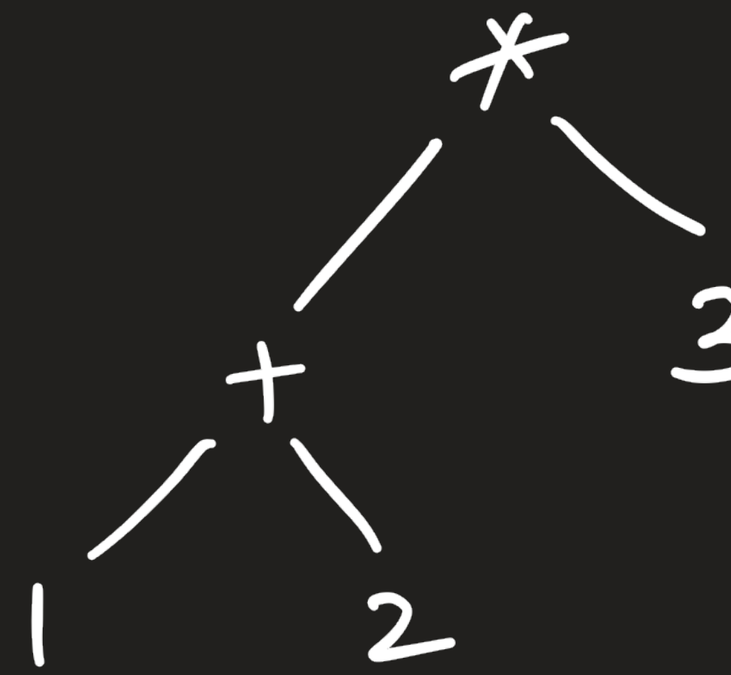
(1 + (2 * 3))



Parse Tree

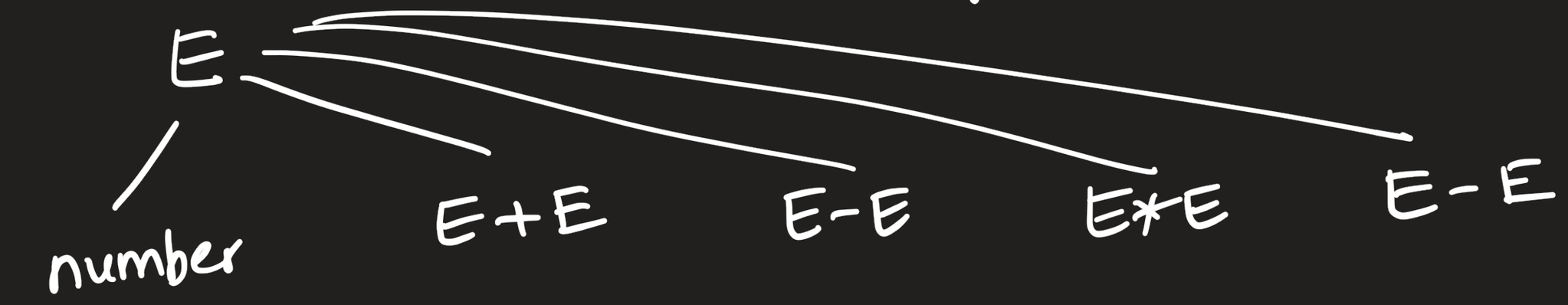
7

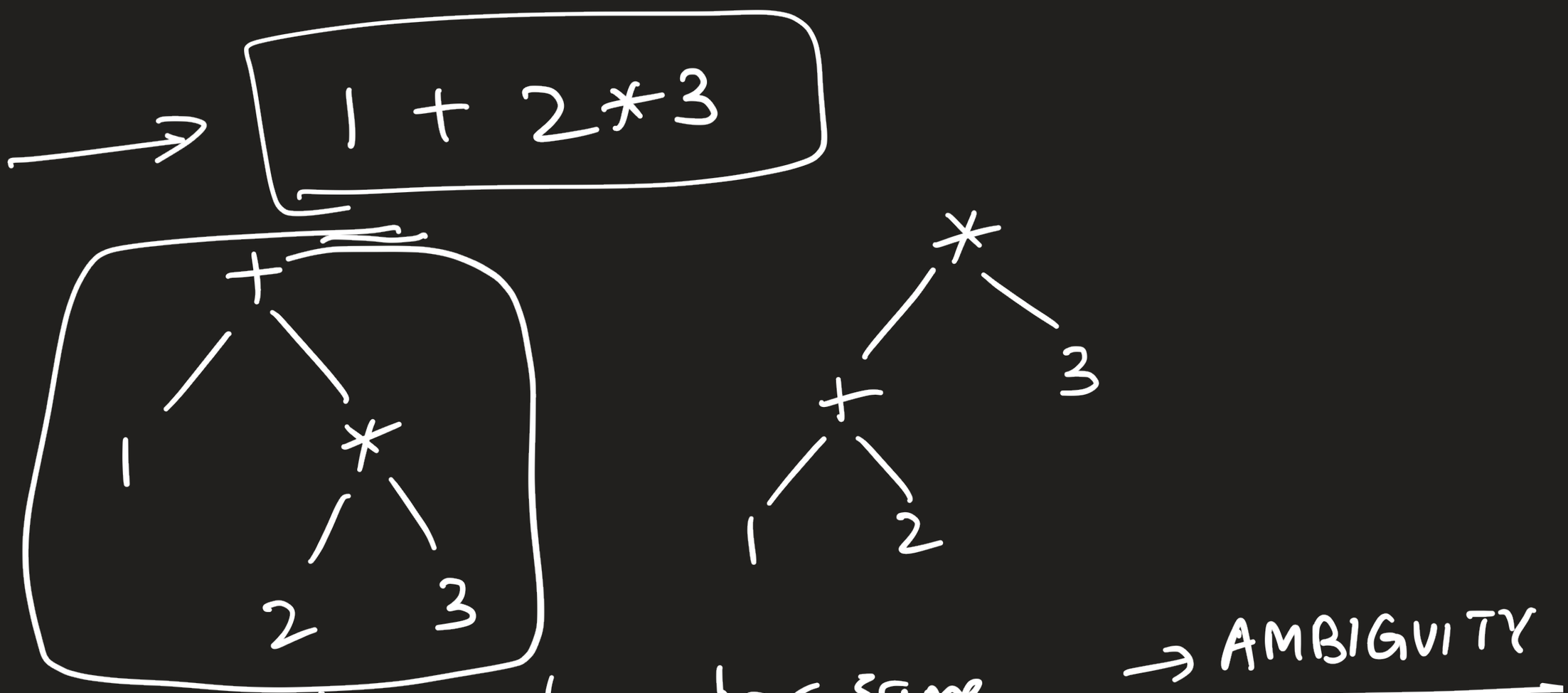
((1+2) * 3)



6

Without parentheses





Multiple parse trees for same expression → AMBIGUITY

Parentheses resolve ambiguity.

$$\left(M_1 \times M_2 \times M_3 \times \ldots \times M_{n+1} \right)$$

```python
    def diffWaysToCompute(self, expression):
        """
        :type expression: str
        :rtype: List[int]
        """

        def helper(S, i, j):

            #Base case
            if S[i:j+1].isdigit():
                return [int(S[i:j+1])]

            #Recursive case
            result = []
            for idx in range(i, j+1):
                if not S[idx].isdigit():
                    Lresults = helper(S, i, idx-1)
                    Rresults = helper(S, idx+1, j)
                    for Lval in Lresults:
                        for Rval in Rresults:
                            if S[idx] == "+":
                                result.append(Lval + Rval)
                            elif S[idx] == "-":
                                result.append(Lval - Rval)
                            else:
                                result.append(Lval * Rval)

            return result

        return helper(expression, 0, len(expression)-1)
```

```python
    def diffWaysToCompute(self, expression):
        """
        :type expression: str
        :rtype: List[int]
        """

        memo = {}

        def helper(S, i, j):
            #Memoization case
            if (i,j) in memo:
                return memo[(i,j)]

            #Recursive case
            result = []
            for index in range(i, j+1):
                if not S[index].isdigit():
                    Lresult = helper(S, i, index-1)
                    Rresult = helper(S, index+1, j)
                    for Lnum in Lresult:
                        for Rnum in Rresult:
                            if S[index] == "+":
                                result.append(Lnum + Rnum)
                            elif S[index] == "-":
                                result.append(Lnum - Rnum)
                            else:
                                result.append(Lnum * Rnum)


            #Base case
            if len(result) == 0:
                result.append(int(S[i:j+1]))

            memo[(i,j)] = result
            return result

        return helper(expression, 0, len(expression)-1)

```