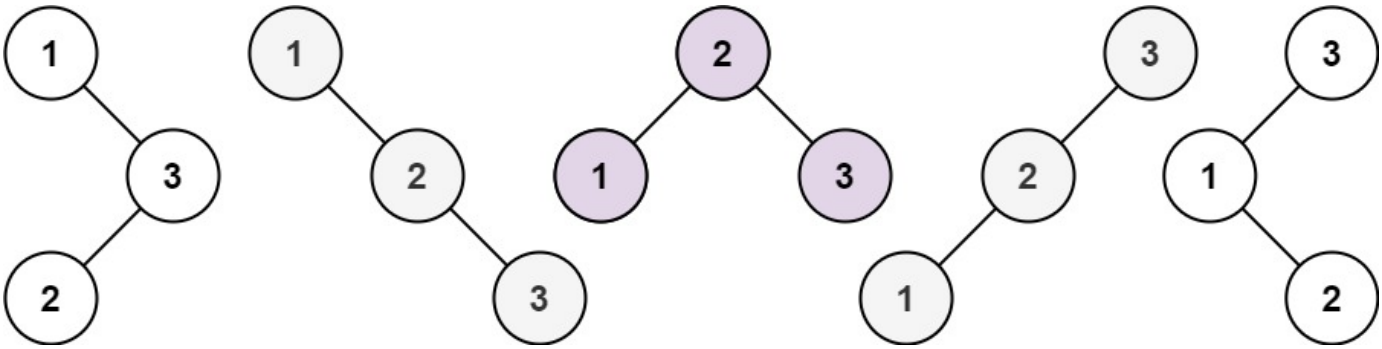
# 95. Unique Binary Search Trees II

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**.

**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]]
```

**Constraints:**

- `1 <= n <= 8`

```
class Solution:
    def generateTrees(self, n: int) -> List[Optional[TreeNode]]:
        if n==0:
            return []
        return self.helper(1,n)

    def helper(self,start,end):
        if start > end:
            return [None]
        ans = []
        for root in range(start,end+1):
            leftTrees = self.helper(start,root-1)
            rightTrees = self.helper(root+1,end)
```

```
            for leftNode in leftTrees:
                for rightNode in rightTrees:
                    node = TreeNode(root)
                    node.left = leftNode
                    node.right = rightNode
                    ans.append(node)
        return ans
```

**Short answer**

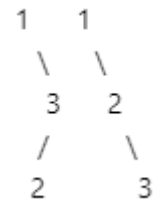
Create root nodes by cycling through [start,end] or [1,n]
Recursively generate all possible left subtrees. By rules of a BST, all values in a left subtree are less than the root. Thus, use the list [start, curRootVal - 1] to create the left subtrees
Recursively generate all possible right subtrees. By rules of a BST, all values in a right subtree are greater than the root. Thus, use the list [curRooVal + 1, end] to create the right subtrees.
For each combination of left and right subtrees, attach them to the current root to form a unique BST (or subtree)
Attach each unique BST to a list
Return list of unique BST

See code at the very bottom!

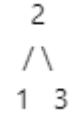
**Long answer**

The goal is to create every possible BST that contain the numbers [1,n]
To do this, we'll start by cycling through [1,n] to create a root node.

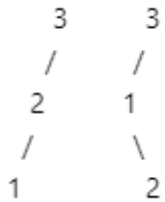
Take the list [1,3] as example, we first create every tree with 1 as a root:



Then every tree with 2 as a root:



Then every tree with 3 as the root:



In Python we could start this by writing:

```
for curRootVal in range(1, n+1): # n+1 since we want [1,n]
    curRoot = TreeNode(curRootVal)
```

Next, we can try to do the same and cycle through [1,n] for each of the children of `curRoot` But we have to obey the rules of a binary search tree which are:
nodes in a *left* subtree must be *less than* the root
and
nodes in a *right* subtree must be *greater than* the root

So it doesn't make sense to cycle through [1,n]. Instead, we make each possible left child by cycling through all values less than the current root or [1,curRootVal -1]. We make each possible right child by cycling through all values greater than the current root or [curRootVal + 1, n]

BUT

this only helps us get the left and right children of our unique roots. We really want each child to not just a single node, but rather the *root of a whole subtree*.
If we apply this strategy *recursively*, then when we make each child, it will be it's own root and repeat the process of cycling through all possible left and right children, which will in turn become their own root and so forth...

Thus, we modify our Python code to generalize for any root and consider all values from a specific start to end.
Our function starts to look like the code below. When reading it, don't follow the recursion just yet! The code is incomplete.

```
def helper(self, start, end):
    for curRootVal in range(start, end+1): # end + 1 because range stops
early
        left_subtree = self.helper(start, curRootVal - 1)
        right_subtree = self.helper(curRootVal + 1, end)

        curRoot = TreeNode(curRootVal)
        curRoot.left = left_subtree
        curRoot.right = right_subtree
```

So for `helper(1,n)`, we've created all possible roots using [1,n] in the for loop. We *somehow* made the left subtree using the list [1,curRootVal-1] (numbers less than curRootVal). We *somehow* made the right subtree using the list [curRootVal +1, n] (numbers greater than curRootVal). Then we make `left_subtree` the left child of `curRoot` and `right_subtree` the right child of `curRoot`.

From this code, should the recursive function return a single subtree? Take a look at the trees for roots 1 and 3 above, they have multiple possible subtrees! Thus, we should return *all possible left and right subtrees*. That is, the recursive function returns a list! I'll call this list `all_trees`.

One problem in our code above is the edge cases. `curRootVal` cycles through [start,end] and `helper(start, curRootVal-1)` is `helper(start, start-1)` when `curRootVal = start`. Similarly, `helper(curRootVal+1, end)` is `helper(end+1, end)` when `curRootVal = end`. In both cases, *within* `helper()`, `start` will be greater than `end` so the for loop will be skipped. Therefore, we shouldn't return any trees in this case. However, I don't want to return an empty list for a reason I'll explain in a second. Instead, I'll return `[None]`.

Now watch the code (I recommend not recurring in your head in your first read through):

**Solution**

```
class Solution:
    def generateTrees(self, n: int) -> List[TreeNode]:
        if n == 0:
            return []
        return self.helper(1, n)


    def helper(self, start, end):
        if start > end: # edge case, see exposition below
            return [None]

        all_trees = [] # list of all unique BSTs
        for curRootVal in range(start, end+1): # generate all roots using
list [start, end]
            # recursively get list of subtrees less than curRoot (a BST must
have left subtrees less than the root)
            all_left_subtrees = self.helper(start, curRootVal-1)

            # recursively get list of subtrees greater than curRoot (a BST
must have right subtrees greater than the root)
            all_right_subtrees = self.helper(curRootVal+1, end)

            for left_subtree in all_left_subtrees:   # get each possible
left subtree
                for right_subtree in all_right_subtrees: # get each possible
right subtree
                    # create root node with each combination of left and
right subtrees
                    curRoot = TreeNode(curRootVal)
                    curRoot.left = left_subtree
                    curRoot.right = right_subtree
```

```
                    # curRoot is now the root of a BST
                    all_trees.append(curRoot)

        return all_trees
```

Hopefully everything but the edge case part makes sense to you. Remember, this case happens when we `curRootValue = start` so we call `helper(start, start-1)`. Or when `curRootValue = end` and we call `helper(end+1, end)`. In both cases, *within* `helper()`, `start` is greater than `end` which will cause the for loop to not run. If the for loop is skipped, we return `all_trees` which is the empty list `[]`. Thus, if the line `all_left_subtrees = self.helper(start, start-1)` returns the empty list, `[]`, what happens when we reach the inner for loop?

```
for left_subtree in all_left_subtrees:
```

This loop won't run if `all_left_subtrees` is empty! But if we changed `all_left_subtrees` to be `[None]`, then `left_subtree` is `None`. It's perfectly safe to set `curRoot.left = None` and we avoid the problem of skipping the loop entirely. The same principle applies for the right subtree.