Solution Review: Problem Challenge 2

We'll cover the following ∧

- Structurally Unique Binary Search Trees (hard) #
- Solution #
- Code #
  - Time complexity #
  - Space complexity #
- Memoized version #

Structurally Unique Binary Search Trees (hard)

Given a number 'n', write a function to return all structurally unique Binary Search Trees (BST) that can store values 1 to 'n'?
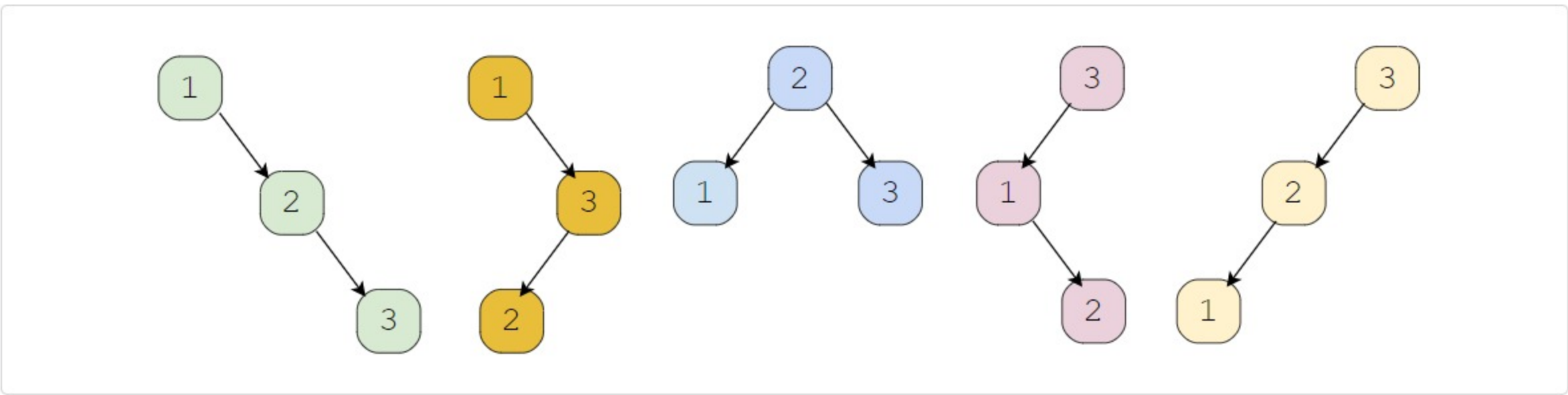
Example 1:

```
Input: 2
Output: List containing root nodes of all structurally unique BSTs.
Explanation: Here are the 2 structurally unique BSTs storing all numbers from 1 to 2:
```
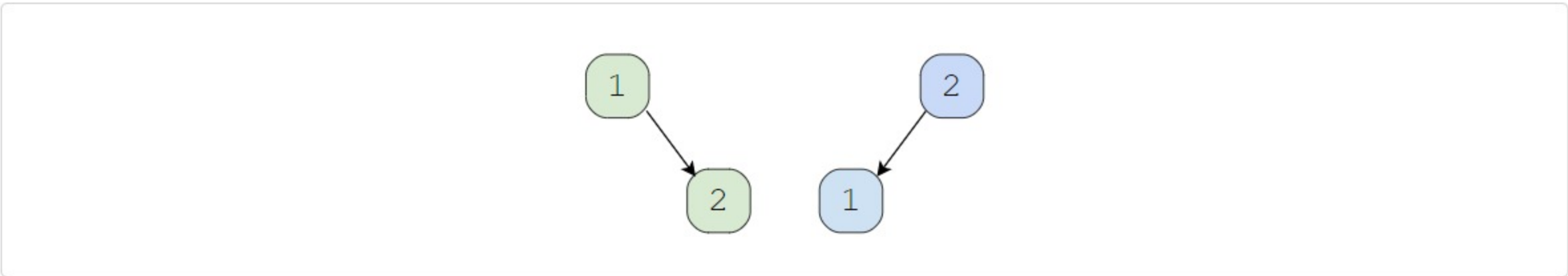

Example 2:

```
Input: 3
Output: List containing root nodes of all structurally unique BSTs.
Explanation: Here are the 5 structurally unique BSTs storing all numbers from 1 to 3:
```


Solution

This problem follows the Subsets pattern and is quite similar to Evaluate Expression. Following a similar approach, we can iterate from 1 to 'n' and consider each number as the root of a tree. All smaller numbers will make up the left sub-tree and bigger numbers will make up the right sub-tree. We will make recursive calls for the left and right sub-trees

Code

Here is what our algorithm will look like:

| Java | Python3 | C++ | JS |

```python
 1
 2  class TreeNode:
 3    def __init__(self, val):
 4      self.val = val
 5      self.left = None
 6      self.right = None
 7
 8
 9  def find_unique_trees(n):
10    if n <= 0:
11      return []
12    return findUnique_trees_recursive(1, n)
13
14
15  def findUnique_trees_recursive(start, end):
16    result = []
17    # base condition, return 'None' for an empty sub-tree
18    # consider n = 1, in this case we will have start = end = 1, this means we should have only one tree
19    # we will have two recursive calls, findUniqueTreesRecursive(1, 0) & (2, 1)
20    # both of these should return 'None' for the left and the right child
21    if start > end:
22      result.append(None)
23      return result
24
25    for i in range(start, end+1):
26      # making 'i' the root of the tree
27      leftSubtrees = findUnique_trees_recursive(start, i - 1)
28      rightSubtrees = findUnique_trees_recursive(i + 1, end)
29      for leftTree in leftSubtrees:
30        for rightTree in rightSubtrees:
31          root = TreeNode(i)
```

[Run]   [Save] [Reset]

Time complexity

The time complexity of this algorithm will be exponential and will be similar to Balanced Parentheses. Estimated time complexity will be $O(n * 2^n)$ but the actual time complexity ($O(4^n/\sqrt{n})$) is bounded by the Catalan number and is beyond the scope of a coding interview. See more details here.

Space complexity

The space complexity of this algorithm will be exponential too, estimated at $O(2^n)$, but the actual will be ($O(4^n/\sqrt{n})$).

Memoized version

Since our algorithm has overlapping subproblems, can we use memoization to improve it? We could, but every time we return the result of a subproblem from the cache, we have to clone the result list because these trees will be used as the left or right child of a tree. This cloning is equivalent to reconstructing the trees, therefore, the overall time complexity of the memoized algorithm will also be the same.

☑ Mark as Completed

⊘ Report an Issue   ？ Ask a Question