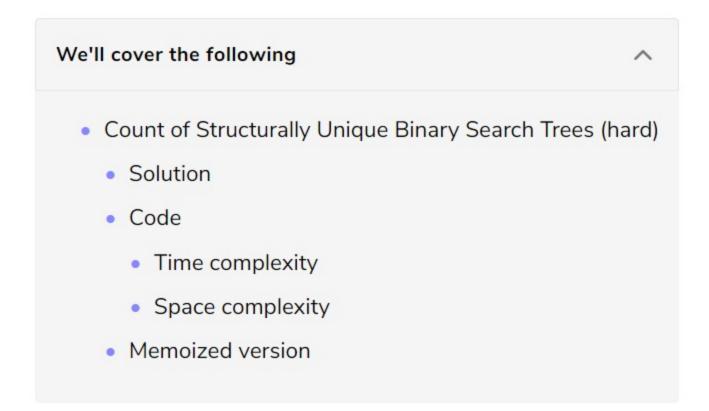


Solution Review: Problem Challenge 3



Count of Structurally Unique Binary Search Trees (hard)#

Given a number 'n', write a function to return the count of structurally unique Binary Search Trees (BST) that can store values 1 to 'n'.

₿

Example 1:

```
Input: 2
Output: 2
Explanation: As we saw in the previous problem, there are 2 unique BSTs storing numbers from 1-2.

Example 2:
```

Explanation: There will be 5 unique BSTs that can store numbers from 1 to 3.

Solution

Input: 3

Output: 5

This problem is similar to Structurally Unique Binary Search Trees. Following a similar approach, we can iterate from 1 to 'n' and consider each number as the root of a tree and make two recursive calls to count the number of left and right sub-trees.

Code

Here is what our algorithm will look like:

```
Python3
                          G C++
                                      JS JS
Java
 2 class TreeNode:
      def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
 9 def count_trees(n):
      if n <= 1:
        return 1
11
12
      count = 0
      for i in range(1, n+1):
13
       # making 'i' root of the tree
14
        countOfLeftSubtrees = count_trees(i - 1)
15
        countOfRightSubtrees = count_trees(n - i)
        count += (countOfLeftSubtrees * countOfRightSubtrees)
17
18
      return count
21
22 def main():
      print("Total trees: " + str(count_trees(2)))
      print("Total trees: " + str(count_trees(3)))
25
26
27 main()
30
                                                                                                                ::3
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 $O(2^n)$ but the actual will be ($O(4^n/\sqrt{n})$.

Memoized version

Our algorithm has overlapping subproblems as our recursive call will be evaluating the same sub-expression multiple times. To resolve this, we can use memoization and store the intermediate results in a **HashMap**. In each function call, we can check our map to see if we have already evaluated this sub-expression before. Here is the memoized version of our algorithm, please see highlighted changes:

```
Python3
                          G C++
                                       JS JS
Java
   class TreeNode:
      def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
    def count_trees(n):
      return count_trees_rec({}, n)
12 def count_trees_rec(map, n):
14
       return map[n]
15
      if n <= 1:
17
        return 1
18
      count = 0
      for i in range(1, n+1):
        # making 'i' the root of the tree
        countOfLeftSubtrees = count_trees_rec(map, i - 1)
21
        countOfRightSubtrees = count_trees_rec(map, n - i)
22
        count += (countOfLeftSubtrees * countOfRightSubtrees)
23
24
      map[n] = count
25
      return count
27
    def main():
      print("Total trees: " + str(count_trees(2)))
      print("Total trees: " + str(count trees(3)))
Run
                                                                                               Save
                                                                                                        Reset
```

The time complexity of the memoized algorithm will be $O(n^2)$, since we are iterating from '1' to 'n' and ensuring that each sub-problem is evaluated only once. The space complexity will be O(n) for the memoization map.

