Solution Review: Problem Challenge 2

We'll cover the following

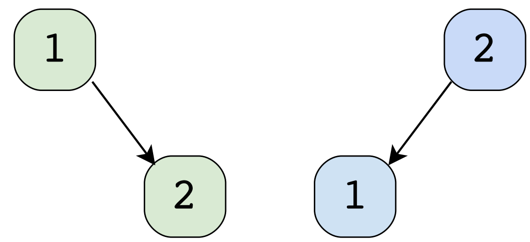
- Structurally Unique Binary Search Trees (hard)
- Solution
- Code
  - Time complexity
  - Space complexity
- Memoized version

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’?

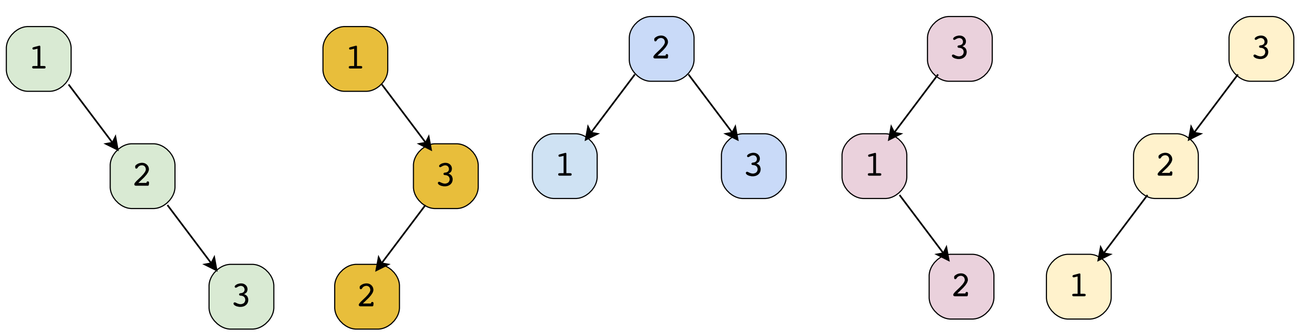
Example 1:

```
Input: 2
Output: List containing root nodes of all structurally unique BS
Ts.
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 BS
Ts.
Explanation: Here are the 5 structurally unique BSTs storing all
 numbers from 1 to 3:
```



Solution

This problem follows the Subsets (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5670249378611200) pattern and is quite similar to Evaluate Expression (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5712272949248000/). 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

```
class TreeNode {
  constructor(val, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}


function find_unique_trees(n) {
  if (n <= 0) {
    return [];
  }
  return findUnique_trees_recursive(1, n);
}

function findUnique_trees_recursive(start, end) {
  const result = [];
  // base condition, return 'null' for an empty sub-tree
  // consider n = 1, in this case we will have start = end = 1, this means we sho
  // we will have two recursive calls, findUniqueTreesRecursive(1, 0) & (2, 1)
  // both of these should return 'null' for the left and the right child
  if (start > end) {
    result.push(null);
    return result;
  }

  for (let i = start; i < end + 1; i++) {
    // making 'i' the root of the tree
    const leftSubtrees = findUnique_trees_recursive(start, i - 1);
    const rightSubtrees = findUnique_trees_recursive(i + 1, end);
    for (let p = 0; p < leftSubtrees.length; p++) {
      for (let q = 0; q < rightSubtrees.length; q++) {
        const root = new TreeNode(i, leftSubtrees[p], rightSubtrees[q]);
        result.push(root);
      }
    }
  }

  return result;
}


console.log(`Total trees: ${find_unique_trees(2).length}`);
console.log(`Total trees: ${find_unique_trees(3).length}`);
```

Run Save Reset

Time complexity

The time complexity of this algorithm will be exponential and will be similar to Balanced Parentheses (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5753264117121024/). 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 (https://en.wikipedia.org/wiki/Catalan_number) and is beyond the scope of a coding interview. See more details here (https://en.wikipedia.org/wiki/Central_binomial_coefficient).

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.

Back

Problem Challenge 2

Next

Problem Challenge 3

Mark as Completed

Report an Issue