Day 29 / 100 :

Topic - Binary Tree

1 Problem statement: All possible full binary Tree (Medium)

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
Output:
[[0,0,0,null,null,0,0,null,null,0,0],[0,0,0,null,null,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,null,null,null,null,0,0],[0,0,0,0,0,null,null,0,0]]

Example 2:
Input: n = 3
Output: [[0,0,0]]

Solutions :

Approach 1 - Dynamic Problem

1. **Base cases**

- If n is 0 or a multiple of 2 : in both cases, a full binary tree isn't possible, so we return empty vector.
- If n is 1 we return a vector with just one node with val = 0.

- This base case comes from Memorization (DP) : if we have seen/stored the vector of Full Binary Trees/roots already for some n, just return that. I used a map for memorization.

2. **Main Part/Recursive Part**

- Firstly get all the possible node combinations that exist in the left and right subtree. For example: for n = 7: the combinations can be (1,5);(3,3);(5,1). Use recursion to get all the possible combinations.
- Now for all the combinations, find all the possible permutations using nested loops, and push them into ans vector. Finally return the vector, but remember to memorize it.

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr),
right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr),
right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right)
: val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    // global map
    unordered_map<int, vector<TreeNode *>> memo;
```

```
    // recursion is very lengthy: memorization can help
a little bit
    vector<TreeNode *> allPossibleFBT(int n) {
        // vector that stores ans
        vector<TreeNode *> ans;
        // base case
        if (n < 1 || n % 2 == 0) {
            // if n is less than 1 or a multiple of 2:
cannot have a FULL binary tree
            return ans;
        }
        if (memo.find(n) != memo.end()) {
            // memo map has something
            return memo[n];
        }
        if (n == 1) {
            // if n is eql to 1 simply return
            ans.push_back(new TreeNode(0));
            memo[1] = ans;
            return ans;
        }


        // main part: including the recursive part
        // get all the possible number of node
combinations possible on the left and right sub tree
        for (int i = 1; i < n; i += 2) {
```

```
            // incremented i by 2 because if we
increment it by 1 : it'll automatically get rejected by
the base condition
            // make 2 vectors that store the number of
nodes in left and right sub tree specifically
            vector<TreeNode *> left = allPossibleFBT(i);
            vector<TreeNode *> right = allPossibleFBT(n
- 1 - i);

            // make trees from all possible combinations
of left and right subtrees
            for (int j = 0; j < left.size(); j++) {
                for (int k = 0; k < right.size(); k++) {
                    TreeNode *root = new TreeNode(0);
                    root->left = left[j];
                    root->right = right[k];
                    ans.push_back(root);
                }
            }
        }
        memo[n] = ans;  // memorize the ans for this
integer
        return ans;
    }
};
```

Solutions :

Approach 2 - Memorization

Intuition:

The recursion works by first generating all possible full binary trees with i nodes, and then all possible full binary trees with n - i - 1 nodes. For each pair of trees, a new tree is created with the first tree as the left subtree and the second tree as the right subtree. This process is repeated until all possible trees with n nodes have been generated.

Approach:

- Initialize a hash table to store the generated trees.
- If n is even, return an empty list.
- If n is 1, add a new tree to the hash table.
- For i from 1 to n - 1, do:
- Generate all possible full binary trees with i nodes.
- Generate all possible full binary trees with n - i - 1 nodes.
- For each pair of trees, create a new tree with the first tree as the left subtree and the second tree as the right subtree.
- Add the new tree to the hash table.
- Return the list of trees in the hash table.

Complexity:

- Time complexity: O(2^n)
- Space complexity: O(n)

```
class Solution {
    static Map<Integer, List<TreeNode>> saved = new HashMap<>();

    public List<TreeNode> allPossibleFBT(int n) {
        if (n%2==0)
            return new ArrayList<>();

        if (!saved.containsKey(n)) {
            List<TreeNode> list = new ArrayList<>();
```

```
            if (n==1)
                list.add(new TreeNode(0));
            else {
                for (int i=1; i<=n-1; i+=2) {
                    List<TreeNode> lTrees = allPossibleFBT(i);
                    List<TreeNode> rTrees = allPossibleFBT(n-i-1);

                    for (TreeNode lt: lTrees) {
                        for (TreeNode rt: rTrees) {
                            list.add(new TreeNode(0, lt, rt));
                        }
                    }
                }
            }

            saved.put(n, list);
        }
        return saved.get(n);
    }
}
```

2 Problem statement: Divide 2 integers (Medium)

Given two integers dividend and divisor, divide two integers without using multiplication, division, and mod operator.
The integer division should truncate toward zero, which means losing its fractional part. For example, 8.345 would be truncated to 8, and -2.7335 would be truncated to -2.
Return the quotient after dividing dividend by divisor.
Note: Assume we are dealing with an environment that could only store integers within the 32-bit signed integer range: [−231, 231 − 1]. For this problem, if the quotient is strictly greater than 231 - 1, then return 231 - 1, and if the quotient is strictly less than -231, then return -231.

Example 1:
Input: dividend = 10, divisor = 3
Output: 3
Explanation: 10/3 = 3.33333.. which is truncated to 3.

Example 2:
Input: dividend = 7, divisor = -3
Output: -2
Explanation: 7/-3 = -2.33333.. which is truncated to -2.

Solutions :

Approach 1 - Bit manipulation

Explanation:

- The key observation is that the quotient of a division is just the number of times that we can subtract the divisor from the dividend without making it negative.

- Suppose dividend = 15 and divisor = 3, 15 - 3 > 0. We now try to subtract more by shifting 3 to the left by 1 bit (6). Since 15 - 6 > 0, shift 6 again to 12. Now 15 - 12 > 0, shift 12 again to 24, which is larger than 15. So we can at most subtract 12 from 15. Since 12 is obtained by shifting 3 to left twice, it is 1 << 2 = 4 times of 3. We add 4 to an answer variable (initialized to be 0). The above process is like 15 = 3 * 4 + 3. We now get part of the quotient (4), with a remaining dividend 3.

- Then we repeat the above process by subtracting divisor = 3 from the remaining dividend = 3 and obtain 0. We are done. In this case, no shift happens. We simply add 1 << 0 = 1 to the answer variable.

- This is the full algorithm to perform division using bit manipulations. The sign also needs to be taken into consideration. And we still need to handle one overflow case: dividend = INT_MIN and divisor = -1.

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (dividend == INT_MIN && divisor == -1) {
            return INT_MAX;
        }
        long dvd = labs(dividend), dvs = labs(divisor), ans = 0;
        int sign = dividend > 0 ^ divisor > 0 ? -1 : 1;
        while (dvd >= dvs) {
```

```
            long temp = dvs, m = 1;
            while (temp << 1 <= dvd) {
                temp <<= 1;
                m <<= 1;
            }
            dvd -= temp;
            ans += m;
        }
        return sign * ans;
    }
};
```