<https://leetcode.com/problems/binary-trees-with-factors/description/>

Given an array of unique integers, arr, where each integer arr[i] is strictly greater than 1.

We make a binary tree using these integers, and each number may be used for any number of times. Each non-leaf node's value should be equal to the product of the values of its children.

Return *the number of binary trees we can make*. The answer may be too large so return the answer **modulo** 109 + 7.

**Example 1:**

Input: arr = [2,4]

Output: 3

Explanation: We can make these trees: [2], [4], [4, 2, 2]

**Example 2:**

Input: arr = [2,4,5,10]

Output: 7

Explanation: We can make these trees: [2], [4], [5], [10], [4, 2, 2], [10, 2, 5], [10, 5, 2].

**Constraints:**

1 <= arr.length <= 1000

2 <= arr[i] <= 109

All the values of arr are **unique**.

**Attempt 1: 2025-05-05**

**Solution 1: DFS (30 min, TLE 42/48)**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numFactoredBinaryTrees(int[] arr) {

        Set<Integer> set = new HashSet<>();

        for(int a : arr) {

            set.add(a);

        }

        long total = 0;

        for(int a : arr) {

            total = (total + helper(set, a)) % MOD;

        }

        return (int) total;

    }

    private long helper(Set<Integer> set, int num) {

        // count the tree with just this node

        long count = 1;

        for(int factor : set) {

            if(num % factor == 0 && set.contains(num / factor)) {

                count = (count + helper(set, factor) \* helper(set, num / factor)) % MOD;

            }

        }

        return count;

    }

}

Time Complexity: O(3^n)

Space Complexity: O(n)

**Solution 2: Memoization (10 min)**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numFactoredBinaryTrees(int[] arr) {

        Set<Integer> set = new HashSet<>();

        for(int a : arr) {

            set.add(a);

        }

        Map<Integer, Long> memo = new HashMap<>();

        long total = 0;

        for(int a : arr) {

            total = (total + helper(set, a, memo)) % MOD;

        }

        return (int) total;

    }

    private long helper(Set<Integer> set, int num, Map<Integer, Long> memo) {

        if(memo.containsKey(num)) {

            return memo.get(num);

        }

        // count the tree with just this node

        long count = 1;

        for(int factor : set) {

            if(num % factor == 0 && set.contains(num / factor)) {

                count = (count + helper(set, factor, memo) \* helper(set, num / factor, memo)) % MOD;

            }

        }

        memo.put(num, count);

        return count;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Solution 3: DP (30 min)**

class Solution {

public int numFactoredBinaryTrees(int[] arr) {

// Define the modulo constant as per problem requirements

final int MOD = (int)1e9 + 7;

int n = arr.length;

// Sort the array to enable bottom-up DP approach

// We process numbers from smallest to largest to ensure when we process arr[i],

// all possible factor pairs (smaller numbers) have already been processed

Arrays.sort(arr);

// DP array where dp[i] represents the number of binary trees with arr[i] as root

// Initialize to 1 because each number can form at least one tree (itself as leaf node)

long[] dp = new long[n];

Arrays.fill(dp, 1);

// Create a value-to-index map for O(1) lookups

// This helps quickly find if a required factor exists in the array

Map<Integer, Integer> indexMap = new HashMap<>();

for(int i = 0; i < n; i++) {

indexMap.put(arr[i], i);

}

// Main DP processing loop

for(int i = 0; i < n; i++) {

// Check all possible smaller numbers that could be factors of arr[i]

for(int j = 0; j < i; j++) {

// Check if arr[j] is a factor of arr[i]

if(arr[i] % arr[j] == 0) {

// Calculate the complementary factor needed

int quotient = arr[i] / arr[j];

// Check if the complementary factor exists in our array

if(indexMap.containsKey(quotient)) {

// Get the index of the complementary factor

int k = indexMap.get(quotient);

// Update dp[i]: total trees = existing count +

// (trees with arr[j] as left child) \* (trees with quotient as right child)

// Using modulo to prevent integer overflow

dp[i] = (dp[i] + dp[j] \* dp[k]) % MOD;

}

}

}

}

// Sum up all possible binary trees that can be formed

long result = 0;

for(long num : dp) {

result += num;

result %= MOD; // Apply modulo at each step to keep numbers manageable

}

return (int) result;

}

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Refer to Deepseek**

This problem asks us to find the number of binary trees we can make using numbers from an array where each non-leaf node's value is the product of its children's values.

**Approach**

Sort the array: This allows us to process numbers in increasing order.

Dynamic Programming (DP): For each number, we'll check all possible pairs of factors in the array that multiply to it.

Modulo operation: Since the result can be large, we'll use modulo 10^9 + 7 as required.

**Solution Code**

class Solution {

public int numFactoredBinaryTrees(int[] arr) {

int MOD = 1\_000\_000\_007;

int n = arr.length;

Arrays.sort(arr);

long[] dp = new long[n];

Arrays.fill(dp, 1); // Each element can be a tree by itself

Map<Integer, Integer> indexMap = new HashMap<>();

for (int i = 0; i < n; i++) {

indexMap.put(arr[i], i);

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < i; j++) {

if (arr[i] % arr[j] == 0) {

int right = arr[i] / arr[j];

if (indexMap.containsKey(right)) {

int k = indexMap.get(right);

dp[i] = (dp[i] + dp[j] \* dp[k]) % MOD;

}

}

}

}

long result = 0;

for (long num : dp) {

result += num;

result %= MOD;

}

return (int) result;

}

}

**Explanation**

Sorting: We start by sorting the array to process numbers in ascending order.

Initialization: We initialize a DP array where each element represents the number of binary trees that can be formed with that number as the root. Initially, each number can form at least one tree (itself as a leaf node).

Factor Checking: For each number arr[i], we check all smaller numbers arr[j] to see if they are factors of arr[i]. If they are, we check if the complementary factor arr[i]/arr[j] exists in the array.

DP Update: If both factors exist, we update dp[i] by adding the product of the counts of trees for both factors.

Result Calculation: Finally, we sum up all values in the DP array to get the total number of possible binary trees.

This approach efficiently counts all possible binary trees using dynamic programming and careful factor checking, with a time complexity of O(n²) where n is the number of elements in the array.

**Can you write solutions: 1.pure DFS, 2.Memoization ?**

**1. Pure DFS Solution (Brute Force)**

This is a straightforward recursive solution that explores all possible factor combinations, but it will be inefficient for larger inputs due to repeated calculations.

class Solution {

private int MOD = 1\_000\_000\_007;

public int numFactoredBinaryTrees(int[] arr) {

Set<Integer> set = new HashSet<>();

for (int num : arr) {

set.add(num);

}

long total = 0;

for (int num : arr) {

total = (total + dfs(num, set)) % MOD;

}

return (int) total;

}

private long dfs(int num, Set<Integer> set) {

long count = 1; // count the tree with just this node

for (int factor : set) {

if (num % factor == 0 && set.contains(num / factor)) {

count = (count + dfs(factor, set) \* dfs(num / factor, set)) % MOD;

}

}

return count;

}

}

**2. Memoization Solution (Optimized DFS)**

This solution adds memoization to cache already computed results, significantly improving performance by avoiding redundant calculations.

class Solution {

private int MOD = 1\_000\_000\_007;

public int numFactoredBinaryTrees(int[] arr) {

Set<Integer> set = new HashSet<>();

for (int num : arr) {

set.add(num);

}

Map<Integer, Long> memo = new HashMap<>();

long total = 0;

for (int num : arr) {

total = (total + dfsWithMemo(num, set, memo)) % MOD;

}

return (int) total;

}

private long dfsWithMemo(int num, Set<Integer> set, Map<Integer, Long> memo) {

if (memo.containsKey(num)) {

return memo.get(num);

}

long count = 1; // count the tree with just this node

for (int factor : set) {

if (num % factor == 0 && set.contains(num / factor)) {

count = (count + dfsWithMemo(factor, set, memo) \* dfsWithMemo(num / factor, set, memo)) % MOD;

}

}

memo.put(num, count);

return count;

}

}

**Explanation**

**Pure DFS Solution:**

Base Case: Each number can form at least one binary tree (itself as a leaf node)

Recursive Case: For each number, find all factor pairs (a, b) where a × b = num

Count Trees: For each valid factor pair, recursively count the trees that can be formed with those factors as children

Sum Results: Sum up all possible trees for each number in the array

**Memoization Solution:**

Memoization Cache: Stores already computed results to avoid redundant calculations

Check Cache First: Before computing, check if result for current number exists in cache

Store Results: After computing, store the result in cache for future use

Same Logic: Otherwise follows the same recursive logic as the pure DFS solution

The memoization solution is much more efficient as it avoids recalculating the number of trees for numbers we've already processed, reducing the time complexity from exponential to polynomial.

**How the pure DFS solution connect to the DP solution ?**

**Connecting the Pure DFS Solution to the DP Solution**

The pure DFS solution and the DP solution for this problem are fundamentally solving the same problem but with different approaches. Let me explain how they connect and how the DP solution essentially optimizes the DFS approach.

**Key Connections Between DFS and DP**

Same Recursive Relationship:

**Both solutions use the same core logic: a number can form trees either as a leaf or as a root with its factor pairs as children**

**The recursive formula is identical: count[num] = 1 + sum(count[a] \* count[b] for all a\*b=num)**

DFS Represents Top-Down Approach:

The pure DFS is a top-down approach where we start with the target number and recursively break it down into factors

This naturally follows the problem's recursive structure but leads to repeated calculations

DP Represents Bottom-Up Approach:

The DP solution is a bottom-up approach where we build solutions for smaller numbers first

This ensures each subproblem is only solved once by processing numbers in sorted order

**How DP Optimizes DFS**

Memoization → DP Table:

The memoization in DFS (storing computed results) directly translates to the DP table

memo[num] in DFS becomes dp[i] in DP where arr[i] = num

Order of Computation:

Pure DFS computes values on demand, potentially many times

DP systematically computes values in order (smallest to largest) ensuring each is computed exactly once

Factor Lookup Optimization:

DFS checks all possible factors for each number

DP uses the sorted array and index map to efficiently find valid factor pairs

**Transformation Steps (DFS to DP)**

Add Memoization:

First optimization is to add memoization to avoid redundant calculations

Change Evaluation Order:

Instead of starting from each number and decomposing, process numbers in sorted order

This ensures when computing dp[i], all possible factor pairs (arr[j], arr[k]) have already been computed

**Replace Recursion with Iteration:**

**The recursive calls become nested loops in DP**

**The base case (leaf nodes) becomes initializing all dp[i] = 1**

**Example Comparison**

For input [2,4,5,10]:

DFS with memoization:

Computes count[2], count[4], etc. as needed

Might compute count[2] multiple times without memoization

Evaluation order depends on recursion path

DP solution:

Sorts array to [2,4,5,10]

Processes in order:

dp[0] = 1 (for 2)

dp[1] = 1 + dp[0]\*dp[0] (for 4 = 2×2)

dp[2] = 1 (for 5)

dp[3] = 1 + dp[0]\*dp[2] (for 10 = 2×5)

**Why DP is More Efficient**

Guaranteed Single Computation:

Each subproblem (count for a number) is solved exactly once

No risk of stack overflow from deep recursion

Better Cache Locality:

Sequential array access is more cache-friendly than recursive calls

Structured Factor Pair Search:

The nested loop structure makes it easier to optimize factor pair finding

The DP solution can be viewed as the most optimized version of the DFS approach, where we've:

Made the evaluation order explicit and optimal

Replaced recursion with iteration

Used additional data structures for efficient factor lookups

Maintained the same fundamental counting logic but in a more efficient framework

**What's the time and space complexity for pure DFS solution and memoization solution and DP solution ?**

Let's analyze the time and space complexity for each of the three solutions to LeetCode 823 (Binary Trees With Factors):

**1. Pure DFS Solution (Brute Force)**

Time Complexity: O(3^n)

**In the worst case, each number can be decomposed into factors in multiple ways**

For a number with all possible factor pairs, we get branching recursion

The upper bound is exponential since we might recompute the same subproblems many times

Space Complexity: O(n)

The recursion stack depth can go up to n (if we have a chain of factors)

The HashSet for storing numbers takes O(n) space

**2. Memoization Solution (Optimized DFS)**

Time Complexity: O(n^2)

Each of the n numbers is computed exactly once

For each number, we check up to n possible factors (though in practice fewer)

With memoization, we avoid recomputing subproblems

Space Complexity: O(n)

The memoization cache stores results for all n numbers

The recursion stack depth is limited by the number of factors

The HashSet still takes O(n) space

**3. DP Solution (Bottom-up)**

Time Complexity: O(n^2)

Sorting the array takes O(n log n)

The double nested loop dominates with O(n^2) operations

Each factor pair check is O(1) thanks to the hash map

Space Complexity: O(n)

The DP array takes O(n) space

The hash map for indices takes O(n) space

No recursion stack overhead

**Key Observations:**

Exponential → Polynomial:

Pure DFS is unusable for larger inputs (n > 20)

Memoization and DP both achieve polynomial time

Same Complexity Class:

Both memoization and DP are O(n^2) time and O(n) space

DP typically has better constant factors (no recursion overhead)

Practical Differences:

Memoization might be slightly easier to implement correctly

DP is more cache-friendly and often faster in practice

DP doesn't risk stack overflow for large inputs

Factor Pair Checking:

The O(n^2) comes from checking all possible factor pairs

In practice, with optimizations, it's often less than n^2

Here's a comparison table:

|  |  |  |  |
| --- | --- | --- | --- |
| Solution | Time Complexity | Space Complexity | Notes |
| Pure DFS | O(3^n) | O(n) | Unusable for n > 20 |
| Memoization | O(n^2) | O(n) | Top-down, recursive |
| DP (Bottom-up) | O(n^2) | O(n) | Bottom-up, iterative |

In interviews, the DP solution is generally preferred as it's the most efficient and doesn't risk stack overflow, though explaining how it connects to the recursive formulation (via memoization) shows deeper understanding.

**Refer to**