823. Binary Trees With Factors Medium (Array) Hash Table Dynamic Programming Sorting Leetcode Link

Problem Description In this problem, you are provided with an array of unique integers called arr, where each integer arr[1] is strictly greater than 1. You

for constructing the trees is that each non-leaf node (i.e., internal node) must have a value equal to the product of the values of its left and right child nodes. What's interesting is that you're not being asked to construct the trees per se, but rather compute the count of different binary trees

are tasked with creating binary trees using these integers, where integers can be used multiple times across different trees. The rule

that can be formed following this product rule. Due to the potentially large number of trees, the final count should be returned modulo 1009 + 7, which is a practice used to keep the answer within the bounds of typical integer storage limits in computational systems. As an example, consider arr = [2, 4]. We can make trees with root values of 2 and 4 directly, each as a single node. Additionally,

we can form a tree with 4 as the root and two children, both with the value 2, since 2 * 2 equals 4. Thus, several trees can be formed with different structures based on the given integers. Intuition

The solution leverages dynamic programming to count the number of possible binary trees. The primary intuition is that for any

Here is the step-by-step thinking process to arrive at this solution: 1. Sorting the Array: The algorithm begins by sorting the array, which allows us to consider pairs (b, c) that can be factors of any

a in an orderly manner. Since b * c = a, once a is fixed, if b is also fixed, c is determined. Sorting ensures we don't miss any

number a within the array, if it can be factored into two numbers b and c such that a = b * c and both b and c exist within the array,

- binary trees that can be made with arr[1] as the root node. 4. Building the DP Array: The algorithm then iterates through each element a in the array. For each a, it checks every element b
- means a can be the root of a binary tree with children b and c. The count of such trees is contributed by the product of counts of b and c's binary trees, which have been stored in f[j] and f[idx[c]] respectively.
- 6. Summarizing the Result: Finally, the total count of binary trees we can form is the sum of counts for all f[i], taken modulo 10^9 + 7.

5. Avoiding Large Numbers: As it computes the counts, the algorithm continually takes modulos of intermediate results by 10^9 +

the large number under control by modulus operation.

This approach efficiently computes the total count of binary trees by considering all possible pairs of factors for each element in the

1. Sort the Array: 1 arr.sort() The array arr is sorted to consider potential factor pairs in a systematic manner. Sorting is crucial because we want to ensure

An index map idx is created, mapping each value v in arr to its index i. This map is later used to quickly find the DP array index corresponding to a specific integer value from arr.

1 f = [1] * n

factors:

represents the fact that each individual number can form a binary tree with just one node. 4. Build the DP Array: The implementation uses a nested for-loop to build the DP array f based on the possible combinations of

A dynamic programming array f is initialized with the length equal to the number of elements in arr, filled with 1's. This

6. Sum and Modulo for the Final Result:

1 return sum(f) % mod

then immediately checks if c exists in the idx map.

if a % b == 0 and (c := (a // b)) in idx:

f[i] = (f[i] + f[j] * f[idx[c]]) % mod

1 for i, a in enumerate(arr):

for j in range(i):

3. Initialize the Dynamic Programming Array:

```
5. Modulo Operation:
  1 % mod
```

Modulo operation is used to ensure the count values in the DP array f do not exceed the integer storage limits.

stay within the storage limits. This is the total count of binary trees that can be created from the array arr.

not impactful in this specific case but is essential in the algorithm to consider all pairs in order.

For a = 2 (f[0]): There are no factors of 2 in the array before it, so f[0] remains 1.

This allows us to find the index of any integer present in arr.

(previous) + 2 (count for 4) * 1 (count for 2) = 5.

MODULO = 10**9 + 7 # The modulo value to keep the numbers in a range

index_by_value = {value: index for index, value in enumerate(arr)}

c = a // b # Calculate the other factor

Integer indexC = indexMap.get(valueC);

// Sum up the tree counts for all values and return the result modulo MODULO

if (indexC != null) {

Creating a dictionary to map array values to their indices for O(1) access

to make each element the root of a binary tree. Start with 1 for each element

can be children of a binary tree with 'a' as the root

Take the modulo to manage large numbers

Initialize a dp (dynamic programming) array to store the number of ways

n = len(arr) # Length of input array

arr.sort() # Sorting the array

as each can be a tree by itself.

Loop through each element 'a' in arr

if c in index_by_value:

dp = [1] * n

and take the modulo

return sum(dp) % MODULO

Note the syntax (c := (a // b)) in idx, which is a walrus operator introduced in Python 3.8 that assigns the value to c and

programming for optimization, and ensuring the numbers are managed within a reasonable range using the modulo operation.

To get the final result, the counts from the DP array f are summed up, and the sum is returned with one final modulo operation to

2. Create an Index Map: We create an index map for easy lookup: 1 $idx = \{2: 0, 4: 1, 8: 2\}$

node.

unique integers: arr = [2, 4, 8].

a binary tree with root 4 and children 2 and 2. So, f[1] becomes 1 (initial) + 1 (count for 2) * 1 (count for 2) = 2. o For a = 8 (f[2]): First, we consider the factor 2, 8 % 2 == 0 and c = 8 // 2 = 4 is in arr, so we can form a tree with root 8 and children 2 and 4. The count f[2] becomes 1 (initial) + 1 (count for 2) * 2 (count for 4) = 3. Then, for the

5. Modulo Operation: Since all the numbers obtained are smaller than 10^9 + 7, there is no need for a modulo operation in this

∘ For a = 4 (f[1]): We have one factor 2 which comes before 4. Since 4 % 2 == 0 and c = 4 // 2 = 2 is in arr, we can have

factor 4, 8 % 4 == 0 and c = 8 // 4 = 2 is also in arr, adding another tree with root 8 and children 4 and 2: f[2] becomes 3

example. However, in the algorithm, we apply % mod at each update for safety. 6. Sum and Modulo for the Final Result: We sum the counts in the dynamic programming array: 1 sum(f) % mod

Which results in (1 + 2 + 5) % mod = 8. Hence, there are a total of 8 unique binary trees that can be formed using the integers

16 for i, a in enumerate(arr): # Try to form trees by checking all the elements 'b' that are factors of 'a' (a % b == 0) # Here we are only checking for factors up to the current element 'a' 19 for j in range(i): 20 b = arr[j]if a % b == 0: # Check if 'b' is a factor of 'a' 21

If the other factor 'c' is in the index_by_value, it means that 'b' and 'c'

by adding the product of the number of ways to create 'b' and 'c'

 $dp[i] = (dp[i] + dp[j] * dp[index_by_value[c]]) % MODULO$

Finally, sum up all the possible ways to form binary trees for each element as the root

// Function to calculate the number of possible binary trees formed by the elements of the array,

Update the number of ways to create a binary tree with 'a' as the root

```
// where each node is equal to the product of two other nodes in the tree (if they exist in the array).
        public int numFactoredBinaryTrees(int[] arr) {
 6
            final int MODULO = 1_000_000_007; // The value to mod the result with to prevent overflow
            Arrays.sort(arr); // Sort the array in ascending order
 8
           Map<Integer, Integer> indexMap = new HashMap<>(); // Map each value to its index in the array
 9
           int arrayLength = arr.length;
10
            // Populate the indexMap with array elements
11
            for (int i = 0; i < arrayLength; i++) {</pre>
12
13
                indexMap.put(arr[i], i);
14
15
            long[] treeCounts = new long[arrayLength]; // Initialize tree count array for each node
            Arrays.fill(treeCounts, 1); // Set initial tree count to 1 for each node
16
17
18
           // Iterate over the sorted array to count the number of trees
19
            for (int i = 0; i < arrayLength; i++) {</pre>
20
                int valueA = arr[i]; // Current value for which we are counting trees
21
                for (int j = 0; j < i; j++) {
                    int valueB = arr[j]; // Possible factor of valueA
22
23
                    // Check if valueB is a factor of valueA
24
                    if (valueA % valueB == 0) {
                        int valueC = valueA / valueB; // The other factor
25
26
                        // Check if the other factor exists in the array using indexMap
```

// Update tree count for valueA by adding the product of tree counts for valueB and valueC

treeCounts[i] = (treeCounts[i] + treeCounts[j] * treeCounts[indexC]) % MODULO;

return (int)sum; // Cast sum to int type as final result is within int range due to modulo operation

39 40 41 42

8

9

10

Typescript Solution

```
const treeCounts: number[] = new Array(arrayLength).fill(1); // Initialize tree count to 1 for each node
 12
 13
 14
         // Iterate over the sorted array to count the number of trees
 15
         for (let i = 0; i < arrayLength; ++i) {</pre>
 16
             const valueA = arr[i]; // Current value for which we are counting trees
 17
             for (let j = 0; j < i; ++j) {
                 const valueB = arr[j]; // Possible factor of valueA
 18
 19
                 // Check if valueB is a factor of valueA
 20
                 if (valueA % valueB === 0) {
                     const valueC = valueA / valueB; // The other factor
 21
 22
                     // Check if the other factor exists in the array using indexMap
 23
                     if (indexMap.has(valueC)) {
 24
                         const indexC = indexMap.get(valueC)!; // Index of the other factor
 25
                         // Update tree count for valueA by adding the product of tree counts for valueB and valueC
                         treeCounts[i] = (treeCounts[i] + treeCounts[j] * treeCounts[indexC]) % MODULO;
 26
 27
 28
 29
 30
 31
         // Sum up the tree counts for all values and return the result modulo MODULO
 32
         return treeCounts.reduce((sum, count) => (sum + count) % MODULO, 0);
 33 }
 34
Time and Space Complexity
The provided code defines a function numFactoredBinaryTrees to calculate the number of binary trees where every node is a factor
of its children.
```

1 // Function to calculate the number of possible binary trees formed by the elements of the array

const MODUL0 = 10 ** 9 + 7; // The value to mod the result with to prevent overflow

function numFactoredBinaryTrees(arr: number[]): number {

// Populate the indexMap with array elements

for (let i = 0; i < arrayLength; ++i) {</pre>

const arrayLength = arr.length;

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

arr.sort((a, b) => a - b); // Sort the array in ascending order

2 // where each node is equal to the product of two other nodes in the tree (if they exist in the array).

const indexMap: Map<number, number> = new Map(); // Map each value to its index in the array

2. Building a dictionary (idx): This operation is linear with respect to the number of elements in arr, leading to a time complexity of 0(n). 3. Nested Loops:

The outer loop runs n times, once for each element in arr.

○ As a result, in the worst case, the total iterations of the nested loop can be calculated by summing up from 1 to n-1, leading to a complexity of $O(n^2/2)$ which simplifies to $O(n^2)$. Combining these, the dominant term is the nested loops with $0(n^2)$, leading to an overall time complexity of $0(n \log n + n + n^2)$,

◦ For each element in the outer loop, the inner loop runs at most i times, where i ranges from 0 to n-1.

The space complexity is determined by the additional space allocated for the computations, not including the input: 1. Dictionary (idx): Space complexity of O(n) for storing indices of elements in arr.

2. Array (f): Space complexity of O(n) for storing the number of possible binary trees for each element in arr.

Therefore, the total space complexity of the algorithm is O(n + n), which simplifies to O(n). In conclusion, the algorithm has a time complexity of $O(n^2)$ and a space complexity of O(n).

factor pairs due to their relative ordering. 2. Creating an Index Map: An index map (idx) is created to easily find the index of any number in arr after it's been sorted. This index is important because we will use it to look up the count of potential subtrees from the dynamic programming array f. 3. Dynamic Programming (DP) Array: A dynamic programming array f is initialized, where f[1] will eventually hold the count of

it can serve as a root of a binary tree whose left child is b and right child is c.

7 to ensure the numbers do not exceed the storage limits of integer types.

that comes before a in the sorted order. If a is perfectly divisible by b, and the quotient c = a / b also exists in the array, it

array, multiplying the counts of trees that can be formed from those factor elements, and cumulatively adding them up while keeping Solution Approach The following is a detailed walkthrough of the implementation of the solution which includes the algorithms, data structures, or

that we consider all potential child nodes for a given root value a in an ascending order.

1 idx = {v: i for i, v in enumerate(arr)}

2. Create an Index Map:

patterns used:

For each number a in arr, we check each number b that comes before a in the array (due to the previous sorting). If a is divisible by b and the division result c is also in arr, then a can be constructed as a root node with b and c as children. The current count of trees with root a is updated by adding the product of counts of trees with roots b and c.

Example Walkthrough

Let's take a small example and walk through the solution approach described above. Suppose we are given the following array of

1. Sort the Array: We start by sorting the array, resulting in arr remaining the same since it is already sorted: [2, 4, 8]. Sorting is

3. Initialize the Dynamic Programming Array: We initialize f to [1, 1, 1]. Each number can at least form a binary tree as a single

The algorithm efficiently calculates the count by exploiting the mathematical properties of factors and products, using dynamic

from arr.

8

9

10

11

12

13

14

15

22

23

24

25

26

27

28

29

30

31

32

33

34

35

4. Build the DP Array:

for each number in the array, updating the count in the dynamic programming array, and continually using modulo operations to manage large numbers. Python Solution class Solution: def num_factored_binary_trees(self, arr: List[int]) -> int:

This demonstrates how the algorithm would count the unique binary trees that can be formed by considering all possible factor pairs

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

C++ Solution

#include <vector>

2 #include <unordered_map>

Java Solution

import java.util.*;

public class Solution {

long sum = 0;

for (long count : treeCounts) {

sum = (sum + count) % MODULO;

```
#include <algorithm>
   class Solution {
   public:
        int numFactoredBinaryTrees(std::vector<int>& arr) {
            const int MOD = 1e9 + 7; // Modulo to avoid integer overflow
            std::sort(arr.begin(), arr.end()); // Sort the array for orderly processing
10
            std::unordered_map<int, int> indexMap; // Map to store the index of each element
11
            int n = arr.size(); // Number of elements in the array
12
            // Populate the index map with the elements from the array
13
            for (int i = 0; i < n; ++i) {
                indexMap[arr[i]] = i;
14
15
            std::vector<long> dp(n, 1); // Dynamic programming array to store number of trees
16
17
18
           // Iterate over each element to compute the number of trees with the element as root
19
            for (int i = 0; i < n; ++i) {
                int rootValue = arr[i];
20
21
                // Iterate over potential left children
22
                for (int j = 0; j < i; ++j) {
23
                   int leftChildValue = arr[j];
24
                    // Check if the current element can be factored by the left child
25
                   if (rootValue % leftChildValue == 0) {
26
                        int rightValue = rootValue / leftChildValue;
27
                        // Check if right child is present in the array
28
                        if (indexMap.count(rightValue)) {
29
                            int rightChildIndex = indexMap[rightValue];
30
                            // Multiply the number of trees from left and right child and add to dp array for root
31
                            dp[i] = (dp[i] + dp[j] * dp[rightChildIndex]) % MOD;
32
33
34
35
36
37
            long answer = 0; // Initialize the answer variable to store the final result
38
           // Sum up all possible trees for each element to get the final result
            for (long value : dp) {
                answer = (answer + value) % MOD;
            return answer; // Return the final answer
43
44
   };
45
```

Time Complexity: The algorithm's time complexity is mainly determined by the double nested loop. Here are the steps, broken down: 1. Sorting: The arr.sort() operation has a time complexity of O(n log n), where n is the length of the input list arr.

which simplifies to $O(n^2)$ since n^2 is the most significant term for large n. **Space Complexity:**