Problem 823: Binary Trees With Factors

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem 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

10

9

+ 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] <= 10

9

All the values of

arr

are

unique

.

Code Snippets

C++:

```
class Solution {
public:
int numFactoredBinaryTrees(vector<int>& arr) {

}
};
```

Java:

```
class Solution {
public int numFactoredBinaryTrees(int[] arr) {

}
}
```

Python3:

```
class Solution:
def numFactoredBinaryTrees(self, arr: List[int]) -> int:
```

Python:

```
class Solution(object):
def numFactoredBinaryTrees(self, arr):
"""
:type arr: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} arr
* @return {number}
*/
var numFactoredBinaryTrees = function(arr) {

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public int NumFactoredBinaryTrees(int[] arr) {

}
}
```

C:

```
int numFactoredBinaryTrees(int* arr, int arrSize) {

}
```

Go:

```
func numFactoredBinaryTrees(arr []int) int {

}
```

Kotlin:

```
class Solution {
fun numFactoredBinaryTrees(arr: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func numFactoredBinaryTrees(_ arr: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn num_factored_binary_trees(arr: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} arr
# @return {Integer}
def num_factored_binary_trees(arr)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $arr
* @return Integer
*/
function numFactoredBinaryTrees($arr) {

}
}
```

Dart:

```
class Solution {
int numFactoredBinaryTrees(List<int> arr) {

}
}
```

Scala:

```
object Solution {
def numFactoredBinaryTrees(arr: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec num_factored_binary_trees(arr :: [integer]) :: integer
def num_factored_binary_trees(arr) do

end
end
```

Erlang:

```
-spec num_factored_binary_trees(Arr :: [integer()]) -> integer().
num_factored_binary_trees(Arr) ->
.
```

Racket:

```
(define/contract (num-factored-binary-trees arr)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Binary Trees With Factors
* Difficulty: Medium
* Tags: array, tree, dp, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int numFactoredBinaryTrees(vector<int>& arr) {

}
};
```

Java Solution:

```
/**
* Problem: Binary Trees With Factors
* Difficulty: Medium
* Tags: array, tree, dp, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int numFactoredBinaryTrees(int[] arr) {

}
}
```

Python3 Solution:

```
"""
Problem: Binary Trees With Factors
Difficulty: Medium
Tags: array, tree, dp, hash, sort

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def numFactoredBinaryTrees(self, arr: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def numFactoredBinaryTrees(self, arr):
"""
:type arr: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Binary Trees With Factors
* Difficulty: Medium
* Tags: array, tree, dp, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
```

```
* @param {number[]} arr
* @return {number}
*/
var numFactoredBinaryTrees = function(arr) {

};
```

TypeScript Solution:

```
/**
* Problem: Binary Trees With Factors
* Difficulty: Medium
* Tags: array, tree, dp, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

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

};
```

C# Solution:

```
/*
* Problem: Binary Trees With Factors
* Difficulty: Medium
* Tags: array, tree, dp, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int NumFactoredBinaryTrees(int[] arr) {

}
}
```

C Solution:

```
/*
* Problem: Binary Trees With Factors
* Difficulty: Medium
* Tags: array, tree, dp, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int numFactoredBinaryTrees(int* arr, int arrSize) {

}
```

Go Solution:

```
// Problem: Binary Trees With Factors
// Difficulty: Medium
// Tags: array, tree, dp, hash, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func numFactoredBinaryTrees(arr []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun numFactoredBinaryTrees(arr: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func numFactoredBinaryTrees(_ arr: [Int]) -> Int {


}
}
```

Rust Solution:

```
// Problem: Binary Trees With Factors
// Difficulty: Medium
// Tags: array, tree, dp, hash, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn num_factored_binary_trees(arr: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} arr
# @return {Integer}
def num_factored_binary_trees(arr)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $arr
* @return Integer
*/
function numFactoredBinaryTrees($arr) {

}
}
```

Dart Solution:

```
class Solution {
int numFactoredBinaryTrees(List<int> arr) {

}
}
```

Scala Solution:

```
object Solution {
def numFactoredBinaryTrees(arr: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec num_factored_binary_trees(arr :: [integer]) :: integer
def num_factored_binary_trees(arr) do

end
end
```

Erlang Solution:

```
-spec num_factored_binary_trees(Arr :: [integer()]) -> integer().
num_factored_binary_trees(Arr) ->
.
```

Racket Solution:

```
(define/contract (num-factored-binary-trees arr)
(-> (listof exact-integer?) exact-integer?)
)
```