<https://leetcode.com/problems/the-number-of-good-subsets/description/>

You are given an integer array nums. We call a subset of nums **good** if its product can be represented as a product of one or more **distinct prime** numbers.

For example, if nums = [1, 2, 3, 4]:

[2, 3], [1, 2, 3], and [1, 3] are **good** subsets with products 6 = 2\*3, 6 = 2\*3, and 3 = 3 respectively.

[1, 4] and [4] are not **good** subsets with products 4 = 2\*2 and 4 = 2\*2 respectively.

Return *the number of different****good****subsets in*nums***modulo***109 + 7.

A **subset** of nums is any array that can be obtained by deleting some (possibly none or all) elements from nums. Two subsets are different if and only if the chosen indices to delete are different.

**Example 1:**

**Input:** nums = [1,2,3,4]

**Output:** 6

**Explanation:** The good subsets are:

- [1,2]: product is 2, which is the product of distinct prime 2.

- [1,2,3]: product is 6, which is the product of distinct primes 2 and 3.

- [1,3]: product is 3, which is the product of distinct prime 3.

- [2]: product is 2, which is the product of distinct prime 2.

- [2,3]: product is 6, which is the product of distinct primes 2 and 3.

- [3]: product is 3, which is the product of distinct prime 3.

**Example 2:**

**Input:** nums = [4,2,3,15]

**Output:** 5

**Explanation:** The good subsets are:

- [2]: product is 2, which is the product of distinct prime 2.

- [2,3]: product is 6, which is the product of distinct primes 2 and 3.

- [2,15]: product is 30, which is the product of distinct primes 2, 3, and 5.

- [3]: product is 3, which is the product of distinct prime 3.

- [15]: product is 15, which is the product of distinct primes 3 and 5.

**Constraints:**

1 <= nums.length <= 105

1 <= nums[i] <= 30

**Attempt 1: 2025-09-03**

**Solution 1: Bit Manipulation + DFS + Memoization (60 min)**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numberOfGoodSubsets(int[] nums) {

        // List of primes up to 30

        int[] primes = new int[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

        // numMasks[i] = bitmask representation of prime factors for number i

        int[] numMasks = new int[31];

        // Frequency array: freq[i] = count of number i in input

        int[] freq = new int[31];

        // Memoization cache: key -> result, where key = mask \* 31 + num

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

        for(int num : nums) {

            freq[num]++;

        }

        for(int num = 2; num <= 30; num++) {

            // Skip numbers not present in input

            if(freq[num] != 0) {

                // Skip numbers with repeated prime factors (4, 8, 9, 12, 16, 18, 20, 24, 25, 27, 28)

                if(num % 4 != 0 && num % 9 != 0 && num % 25 != 0) {

                    // Create bitmask for this number's prime factors

                    int mask = 0;

                    for(int i = 0; i < primes.length; i++) {

                        if(num % primes[i] == 0) {

                            // Set the bit for this prime

                            mask |= (1 << i);

                        }

                    }

                    numMasks[num] = mask;

                }

            }

        }

        // Start DFS from initial state: no primes used (mask=0), starting from number 2

        long total = helper(0, 2, primes, numMasks, freq, memo);

        // Subtract 1 to exclude the empty subset (which is included in DFS result)

        total = (total - 1 + MOD) % MOD;

        // Multiply by 2^(count of 1s) since 1s can be added to any subset without affecting prime factors

        long ones = pow(2, freq[1]);

        total = total \* ones % MOD;

        return (int) total;

    }

    private long helper(int mask, int num, int[] primes, int[] numMasks, int[] freq, Map<Integer, Long> memo) {

        // When num exceeds 30, return 1 (meaning we've considered all numbers,

        // and this path has one way to form a subset—though we'll subtract the

        // empty subset later).

        if(num > 30) {

            return 1;

        }

        // Create unique key for memoization: combines mask and current number

        int key = mask \* 31 + num;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        // Option 1: Skip the current number

        long result = helper(mask, num + 1, primes, numMasks, freq, memo);

        // Option 2: Include the current number if valid and compatible

        if(freq[num] > 0 && numMasks[num] != 0) {

            int numMask = numMasks[num];

            // Check if this number's primes don't overlap with already used primes

            if((mask & numMask) == 0) {

                // Include this number: multiply by frequency and recurse

                result = (result + freq[num] \* helper(mask | numMask, num + 1, primes, numMasks, freq, memo)) % MOD;

            }

        }

        memo.put(key, result);

        return result;

    }

    // Efficient exponentiation with modulo

    private long pow(long base, long exponent) {

        long result = 1;

        while(exponent > 0) {

            if(exponent % 2 == 1) {

                result = result \* base % MOD;

            }

            base = base \* base % MOD;

            exponent /= 2;

        }

        return result;

    }

}

Time Complexity: The DFS approach has a time complexity of (O(2^m \* n)),

where (m) is the number of primes (10) and (n) is the number of valid

numbers (up to 30). While manageable for small (m), the recursion overhead

and memoization using maps can be less efficient than iterative DP.

**Solution 2: Bit Manipulation + 2D DP (120 min)**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numberOfGoodSubsets(int[] nums) {

        int[] primes = new int[] {2,3,5,7,11,13,17,19,23,29};

        // Frequency array: freq[i] = count of number i in input

        int[] freq = new int[31];

        // numMasks[i] = bitmask representation of prime factors for number i

        int[] numMasks = new int[31];

        for(int num : nums) {

            freq[num]++;

        }

        // Precompute the prime masks for numbers 2 to 30

        for(int num = 2; num <= 30; num++) {

            if(freq[num] > 0) {

                // Skip numbers with repeated prime factors (4, 8, 9, 12, 16, 18, 20, 24, 25, 27, 28)

                if(num % 4 != 0 && num % 9 != 0 && num % 25 != 0) {

                    int mask = 0;

                    for(int i = 0; i < primes.length; i++) {

                        if(num % primes[i] == 0) {

                            mask |= (1 << i);

                        }

                    }

                    numMasks[num] = mask;

                }

            }

        }

        int totalMasks = 1 << primes.length;

        // DP array: dp[mask][num] represents the number of good subsets from numbers 'num' to 30 with prime mask 'mask'

        // We use indices for num from 2 to 31 (inclusive). So we need a 2D array of size [totalMasks][32].

        // Note: num=31 is the base case (equivalent to num>30 in DFS).

        // [mask][num], num index: 2 to 31 (0-31 unused for num=0,1)

        long[][] dp = new long[totalMasks][32];

        // Base case: for all masks, when num=31 (i.e., beyond 30), there is 1 way (empty subset)

        for(int mask = 0; mask < totalMasks; mask++) {

            dp[mask][31] = 1;

        }

        // Iterate from num=30 down to num=2 (reverse order of DFS recursion)

        for(int num = 30; num >= 2; num--) {

            // Get the precomputed mask for this number

            int numMask = numMasks[num];

            // 为什么内层循环需要遍历所有mask ?

            // 在DFS中，对于每个num，递归调用会处理所有可能的mask状态（通过递归树展开）。

            // 在2D DP中，内层循环显式地遍历所有mask值，以确保计算每个num下所有mask状态的结果。

            // 这避免了递归的开销，但保持了相同的逻辑。

            for(int mask = 0; mask < totalMasks; mask++) {

                // Start by skipping the current number: value from num+1

                dp[mask][num] = dp[mask][num + 1];

                // If the current number is valid and available, consider including it

                if(freq[num] != 0 && numMask != 0) {

                    // Check if the current number's primes do not conflict with the current mask

                    if((mask & numMask) == 0) {

                        int newMask = mask | numMask;

                        // Include the current number: multiply by frequency and add the value from num+1 with newMask

                        dp[mask][num] = (dp[mask][num] + freq[num] \* dp[newMask][num + 1]) % MOD;

                    }

                }

            }

        }

        // The result for all numbers from 2 to 30 is stored in dp[0][2] (which includes the empty subset)

        long total = dp[0][2];

        // Subtract the empty subset

        total = (total - 1 + MOD) % MOD;

        // Multiply by 2^(count of 1s) since 1s can be added arbitrarily

        long ones = pow(2, freq[1]);

        total = total \* ones % MOD;

        return (int) total;

    }

    private long pow(long base, long exponent) {

        long result = 1;

        while(exponent > 0) {

            if(exponent % 2 == 1) {

                result = result \* base % MOD;

            }

            base = base \* base % MOD;

            exponent /= 2;

        }

        return result;

    }

}

Time Complexity: O(totalMasks × maxNum) = O(1024 × 30) = O(30,720)

Space Complexity: O(totalMasks × (maxNum + 1)) = O(1024 × 31) = O(31,744)

**Solution 3: Bit Manipulation + 1D DP (120 min)**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numberOfGoodSubsets(int[] nums) {

        int[] primes = new int[] {2,3,5,7,11,13,17,19,23,29};

        // Frequency array: freq[i] = count of number i in input

        int[] freq = new int[31];

        // numMasks[i] = bitmask representation of prime factors for number i

        int[] numMasks = new int[31];

        for(int num : nums) {

            freq[num]++;

        }

        // Precompute the prime masks for numbers 2 to 30

        for(int num = 2; num <= 30; num++) {

            if(freq[num] > 0) {

                // Skip numbers with repeated prime factors (4, 8, 9, 12, 16, 18, 20, 24, 25, 27, 28)

                if(num % 4 != 0 && num % 9 != 0 && num % 25 != 0) {

                    int mask = 0;

                    for(int i = 0; i < primes.length; i++) {

                        if(num % primes[i] == 0) {

                            mask |= (1 << i);

                        }

                    }

                    numMasks[num] = mask;

                }

            }

        }

        int totalMasks = 1 << primes.length;

        // DP array: dp[mask][num] represents the number of good subsets from numbers 'num' to 30 with prime mask 'mask'

        // We use indices for num from 2 to 31 (inclusive). So we need a 2D array of size [totalMasks][32].

        // Note: num=31 is the base case (equivalent to num>30 in DFS).

        // [mask][num], num index: 2 to 31 (0-31 unused for num=0,1)

        long[] dp = new long[totalMasks];

        // 初始化：空集有一种方案（mask=0）

        dp[0] = 1;

        // Iterate from num=30 down to num=2 (reverse order of DFS recursion)

        for(int num = 30; num >= 2; num--) {

            // Get the precomputed mask for this number

            int numMask = numMasks[num];

            // 为什么内层循环需要遍历所有mask ?

            // 在DFS中，对于每个num，递归调用会处理所有可能的mask状态（通过递归树展开）。

            // 在1D DP中，内层循环显式地遍历所有mask值，以确保计算每个num下所有mask状态的结果。

            // 这避免了递归的开销，但保持了相同的逻辑。

            // 在压缩2D DP到1D DP的过程中，关键：逆序遍历所有mask（从大到小），以避免覆盖当前数字处理前的状态

            // 因为newMask = mask | numMask 总是大于或等于mask，逆序确保在更新dp[newMask]时，dp[mask]是未更新的值

            for(int mask = totalMasks - 1; mask >= 0; mask--) {

                // If the current number is valid and available, consider including it

                if(freq[num] != 0 && numMask != 0) {

                    // Check if the current number's primes do not conflict with the current mask

                    if((mask & numMask) == 0) {

                        int newMask = mask | numMask;

                        // 状态转移：选择当前数字，newMask的方案数增加 freq[num] \* dp[mask]

                        dp[newMask] = (dp[newMask] + freq[num] \* dp[mask]) % MOD;

                    }

                }

            }

        }

        // The result for all numbers from 2 to 30 is stored in dp[0][2] (which includes the empty subset)

        // 计算所有非空好子集的数量（排除mask=0的空集）

        long total = 0;

        for (int i = 1; i < totalMasks; i++) {

            total = (total + dp[i]) % MOD;

        }

        // Multiply by 2^(count of 1s) since 1s can be added arbitrarily

        long ones = pow(2, freq[1]);

        total = total \* ones % MOD;

        return (int) total;

    }

    private long pow(long base, long exponent) {

        long result = 1;

        while(exponent > 0) {

            if(exponent % 2 == 1) {

                result = result \* base % MOD;

            }

            base = base \* base % MOD;

            exponent /= 2;

        }

        return result;

    }

}

Time Complexity: O(totalMasks × maxNum) = O(1024 × 30) = O(30,720)

Space Complexity: O(totalMasks × (maxNum + 1)) = O(1024 × 31) = O(31,744)

**Refer to Deepseek**

**🔍 Step 1: Understand the Proble**m

A **good subset** is defined as a subset of nums where the product of all its elements can be expressed as a product of **distinct prime numbers** (i.e., no prime number appears more than once in the prime factorization of the product). For example:

✅ [2, 3] is good because (2 \times 3 = 6) (primes 2 and 3 are distinct).

❌ [4] is not good because (4 = 2^2) (prime 2 appears twice).

The number **1** is special: it doesn't contribute any prime factors, so it can be included in any good subset without affecting the product.

**🧠 Step 2: Key Observations for DFS Approac**h

**Prime Factors and Bitmasks**:

Since numbers in nums are between 1 and 30, we only need primes up to 30: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29].

Each number (except 1) can be represented as a **bitmask** of its prime factors. For example:

(6 = 2 \times 3) → bitmask 0b11 (bits 0 and 1 set for primes 2 and 3).

(15 = 3 \times 5) → bitmask 0b110 (bits 1 and 2 set for primes 3 and 5).

**Invalid Numbers**:

Numbers with repeated prime factors (like (4 = 2^2), (9 = 3^2), (25 = 5^2)) cannot be part of any good subset and are skipped.

**DFS State**:

We define a state by a **bitmask** usedMask that represents the set of primes already used in the current subset.

The goal is to explore all subsets where no prime is used more than once (i.e., no overlapping bits in usedMask and the current number's mask).

**Recursive DFS**:

For each number, we decide whether to **include** it (if it doesn't share any prime with the current usedMask) or **exclude** it.

Base case: after processing all numbers, we count the subset if it is good.

**🌳 Step 3: DFS Solution with Memoizatio**n

We use **memoization** to avoid recomputing states with the same usedMask.

import java.util.\*;

class Solution {

// Modulo value for large numbers

private static final int MOD = 1000000007;

// List of primes up to 30

private static final int[] PRIMES = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

// Frequency array: freq[i] = count of number i in input

private int[] freq;

// numMasks[i] = bitmask representation of prime factors for number i

private int[] numMasks;

// Memoization cache: key -> result, where key = mask \* 31 + num

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

public int numberOfGoodSubsets(int[] nums) {

// Initialize frequency array for numbers 1-30

freq = new int[31];

for (int num : nums) {

if (num <= 30) {

freq[num]++;

}

}

// Precompute prime bitmasks for numbers 2-30

numMasks = new int[31];

for (int num = 2; num <= 30; num++) {

if (freq[num] == 0) continue; // Skip numbers not present in input

// Skip numbers with repeated prime factors (4, 8, 9, 12, 16, 18, 20, 24, 25, 27, 28)

if (num % 4 == 0 || num % 9 == 0 || num % 25 == 0) continue;

// Create bitmask for this number's prime factors

int mask = 0;

for (int i = 0; i < PRIMES.length; i++) {

if (num % PRIMES[i] == 0) {

mask |= (1 << i); // Set the bit for this prime

}

}

numMasks[num] = mask;

}

// Start DFS from initial state: no primes used (mask=0), starting from number 2

long total = dfs(0, 2);

// Subtract 1 to exclude the empty subset (which is included in DFS result)

total = (total - 1 + MOD) % MOD;

// Multiply by 2^(count of 1s) since 1s can be added to any subset without affecting prime factors

long ones = pow(2, freq[1]);

total = total \* ones % MOD;

return (int) total;

}

/\*\*

\* Recursive DFS function to count valid subsets

\* @param mask Bitmask representing primes already used in current subset

\* @param num Current number being considered (from 2 to 30)

\* @return Number of valid subsets from current state

\*/

private long dfs(int mask, int num) {

// Base case: processed all numbers (2-30)

if (num > 30) {

return 1; // Return 1 to count this path (includes empty subset)

}

// Create unique key for memoization: combines mask and current number

int key = mask \* 31 + num;

if (memo.containsKey(key)) {

return memo.get(key); // Return cached result if available

}

// Option 1: Skip the current number

long res = dfs(mask, num + 1);

// Option 2: Include the current number if valid and compatible

if (freq[num] > 0 && numMasks[num] != 0) {

int numMask = numMasks[num];

// Check if this number's primes don't overlap with already used primes

if ((mask & numMask) == 0) {

// Include this number: multiply by frequency and recurse

res = (res + freq[num] \* dfs(mask | numMask, num + 1)) % MOD;

}

}

// Store result in memoization cache

memo.put(key, res);

return res;

}

/\*\*

\* Efficient exponentiation with modulo

\* @param base Base number

\* @param exponent Exponent

\* @return base^exponent mod MOD

\*/

private long pow(long base, long exponent) {

long result = 1;

while (exponent > 0) {

if (exponent % 2 == 1) {

result = result \* base % MOD;

}

base = base \* base % MOD;

exponent /= 2;

}

return result;

}

}

**📝 Explanation of DFS Solution**:

**Memoization**: The memo map stores the number of good subsets for a given usedMask to avoid recomputation.

**State**: usedMask is a bitmask where each bit indicates if a prime has been used.

**Recursion**:

**Skip current number**: Proceed to next number without changing usedMask.

**Include current number**: If the number's prime mask doesn't overlap with usedMask, multiply by its frequency and recurse with updated usedMask.

**Base Case**: When all numbers (2-30) are processed, return 1 (count one subset).

**Handling 1s**: After DFS, multiply result by (2^{\text{count of 1s}}) since 1s can be added arbitrarily.

**Key Points in the DFS Solution:**

**Excluding the Empty Subset**: The raw DFS included the empty subset in its count. The update involves subtracting 1 from the result to exclude the empty subset.

**State Management**: The DFS now uses a composite key for memoization that combines the current bitmask and the number being processed. This ensures correct state tracking and avoids overcounting.

**Handling Number 1**: The number 1 is handled separately by multiplying the result by 2^count of 1s at the end, accounting for all combinations of including 1s without affecting the product.

**⚡ Step 4: Mapping DFS to 2D DP Solution**

根据您提出的“顶底之术”转换逻辑，我将LeetCode 1994的DFS解法转换为符合该逻辑的DP解法。在DFS中，递归调用从num=2开始（顶），递归到num>30时返回1（底）。在DP中，迭代方向应与递归方向相反，即从底（num=31）开始，向顶（num=2）迭代。

转换后的DP解法代码

class Solution {

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

public int numberOfGoodSubsets(int[] nums) {

// List of primes up to 30

int[] primes = new int[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

int nPrimes = primes.length;

int totalMasks = 1 << nPrimes;

// numMasks[i] = bitmask representation of prime factors for number i

int[] numMasks = new int[31];

// Frequency array: freq[i] = count of number i in input

int[] freq = new int[31];

for (int num : nums) {

if (num <= 30) {

freq[num]++;

}

}

// Precompute the prime masks for numbers 2 to 30

for (int num = 2; num <= 30; num++) {

if (freq[num] == 0) continue;

// Skip numbers with repeated prime factors (4, 8, 9, 12, 16, 18, 20, 24, 25, 27, 28)

if (num % 4 == 0 || num % 9 == 0 || num % 25 == 0) continue;

int mask = 0;

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

if (num % primes[i] == 0) {

mask |= (1 << i);

}

}

numMasks[num] = mask;

}

// DP array: dp[mask][num] represents the number of good subsets from numbers 'num' to 30 with prime mask 'mask'

// We use indices for num from 2 to 31 (inclusive). So we need a 2D array of size [totalMasks][32].

// Note: num=31 is the base case (equivalent to num>30 in DFS).

long[][] dp = new long[totalMasks][32]; // [mask][num], num index: 2 to 31 (0-31 unused for num=0,1)

// Base case: for all masks, when num=31 (i.e., beyond 30), there is 1 way (empty subset)

for (int mask = 0; mask < totalMasks; mask++) {

dp[mask][31] = 1;

}

// Iterate from num=30 down to num=2 (reverse order of DFS recursion)

for (int num = 30; num >= 2; num--) {

int numMask = numMasks[num]; // get the precomputed mask for this number

// 为什么内层循环需要遍历所有mask ?

// 在DFS中，对于每个num，递归调用会处理所有可能的mask状态（通过递归树展开）。

// 在2D DP中，内层循环显式地遍历所有mask值，以确保计算每个num下所有mask状态的结果。

// 这避免了递归的开销，但保持了相同的逻辑。

for (int mask = 0; mask < totalMasks; mask++) {

// Start by skipping the current number: value from num+1

dp[mask][num] = dp[mask][num + 1];

// If the current number is valid and available, consider including it

if (freq[num] > 0 && numMask != 0) {

// Check if the current number's primes do not conflict with the current mask

if ((mask & numMask) == 0) {

int newMask = mask | numMask;

// Include the current number: multiply by frequency and add the value from num+1 with newMask

dp[mask][num] = (dp[mask][num] + freq[num] \* dp[newMask][num + 1]) % MOD;

}

}

}

}

// The result for all numbers from 2 to 30 is stored in dp[0][2] (which includes the empty subset)

long total = dp[0][2];

total = (total - 1 + MOD) % MOD; // subtract the empty subset

// Multiply by 2^(count of 1s) since 1s can be added arbitrarily

long ones = pow(2, freq[1]);

total = total \* ones % MOD;

return (int) total;

}

private long pow(long base, long exponent) {

long result = 1;

while (exponent > 0) {

if (exponent % 2 == 1) {

result = result \* base % MOD;

}

base = base \* base % MOD;

exponent /= 2;

}

return result;

}

}

**解释符合“顶底之术”的部分**

**DFS基础情况（底）**：在DFS中，当num > 30时返回1。在DP中，我们初始化dp[mask][31] = 1 for all mask，直接对应基础情况。

**DFS递归方向**：DFS中递归调用num + 1，即从num=2向num=31方向递归。在DP中，迭代方向相反，从num=30向下迭代到num=2，即从底向顶计算。

**状态转移**：DP中的状态转移完全镜像DFS的递归逻辑：

跳过当前数字：dp[mask][num] = dp[mask][num+1]

包含当前数字：如果数字有效且质数不冲突，则dp[mask][num] += freq[num] \* dp[newMask][num+1]

**初始状态（顶）**：DFS的初始调用是helper(0, 2, ...)，对应DP的最终结果dp[0][2]。

这种解法直接遵循了“顶底之术”的转换原则，通过反向迭代填充DP表，确保子问题先被计算。虽然空间复杂度较高（O(1024 \* 32)），但清晰体现了DFS到DP的转换逻辑。

**关于2D DP解法下面这个部分里面内层的for循环for (int mask = 0; mask < totalMasks; mask++) {...}对应DFS解法中的哪一部分逻辑？**

在2D DP解法中，内层循环for (int mask = 0; mask < totalMasks; mask++) {...} 对应DFS解法中的递归逻辑，具体来说，它模拟了DFS函数helper(int mask, int num, ...)中对所有可能mask状态的处理。以下是详细解释：

**DFS解法中的关键逻辑**

在DFS解法中，helper函数对于给定的mask和num执行以下操作：

**基础情况**：如果num > 30，返回1（表示空集）。

**跳过当前数字**：递归调用helper(mask, num + 1, ...)，表示不选择当前数字num。

**包含当前数字**：如果当前数字num有效（即freq[num] > 0且numMasks[num] != 0）且与当前mask不冲突（即(mask & numMask) == 0），则递归调用helper(mask | numMask, num + 1, ...)，并将结果乘以freq[num]（因为数字num可能出现多次）。

**2D DP解法中的内层循环对应关系**

在2D DP解法中，内层循环遍历所有可能的mask值（从0到totalMasks - 1），对于每个mask，它计算dp[mask][num]（即状态(mask, num)的结果）。具体代码行与DFS逻辑的对应关系如下：

dp[mask][num] = dp[mask][num + 1];
这对应DFS中**跳过当前数字**的递归调用helper(mask, num + 1, ...)。这里直接使用dp[mask][num + 1]的值，因为DP表已经计算了num+1的状态（由于DP从num=30向下填充，num+1的状态已先计算）。

接下来的if语句：

if (freq[num] > 0 && numMask != 0) {

if ((mask & numMask) == 0) {

int newMask = mask | numMask;

dp[mask][num] = (dp[mask][num] + freq[num] \* dp[newMask][num + 1]) % MOD;

}

}

这对应DFS中**包含当前数字**的逻辑。如果当前数字有效且与mask不冲突，则计算新masknewMask = mask | numMask，并添加freq[num] \* dp[newMask][num + 1]到结果中。这对应DFS中的freq[num] \* helper(mask | numMask, num + 1, ...)。

**为什么内层循环需要遍历所有mask？**

在DFS中，对于每个num，递归调用会处理所有可能的mask状态（通过递归树展开）。在2D DP中，内层循环显式地遍历所有mask值，以确保计算每个num下所有mask状态的结果。这避免了递归的开销，但保持了相同的逻辑。

**总结**

内层循环for (int mask = 0; mask < totalMasks; mask++) 本质上是在模拟DFS中对每个固定num处理所有mask状态的过程。它直接映射到DFS递归函数的主体部分，包括跳过当前数字和包含当前数字的分支。DP通过表格存储子问题结果，避免了重复计算，从而提高了效率。

**2D DP解法的时间复杂度和空间复杂度分析**

**基本参数定义**

nPrimes = 10（30以内的质数数量）

totalMasks = 1 << nPrimes = 1024（所有可能的质数组合状态数）

maxNum = 30（处理的数字范围）

**时间复杂度：O(totalMasks × maxNum) = O(1024 × 30) = O(30,720)**

**详细分析：**

for (int num = 30; num >= 2; num--) { // 循环29次 (30-2+1)

for (int mask = 0; mask < totalMasks; mask++) { // 循环1024次

// 常数时间操作：比较、位运算、赋值

dp[mask][num] = dp[mask][num + 1]; // O(1)

if (freq[num] > 0 && numMask != 0) { // O(1)

if ((mask & numMask) == 0) { // O(1)

int newMask = mask | numMask; // O(1)

dp[mask][num] = (dp[mask][num] + freq[num] \* dp[newMask][num + 1]) % MOD; // O(1)

}

}

}

}

**计算：**

外层循环：29次（数字从30到2）

内层循环：1024次（所有mask状态）

每次内层循环的操作：常数时间（O(1)）

总时间复杂度：29 × 1024 × O(1) = O(29,696) ≈ O(30,000)

**空间复杂度：O(totalMasks × (maxNum + 1)) = O(1024 × 31) = O(31,744)**

**详细分析：**

long[][] dp = new long[totalMasks][32]; // [1024][32]

**存储需求：**

DP表维度：1024行 × 32列

每元素大小：long类型占用8字节

总空间：1024 × 32 × 8 bytes = 262,144 bytes ≈ 0.26 MB

**预处理阶段复杂度**

**预处理阶段：**

// 频率统计：O(n)，其中n是输入数组长度

for (int num : nums) {

if (num <= 30) freq[num]++;

}

// 掩码预处理：O(30 × 10) = O(300)

for (int num = 2; num <= 30; num++) {

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

// 质数检查

}

}

预处理阶段的时间复杂度为O(n + 300)，相对于主DP循环可以忽略不计。

**复杂度总结**

|  |  |  |
| --- | --- | --- |
| 复杂度类型 | 数值 | 大O表示法 |
| 时间复杂度 | 29 × 1024 = 29,696 次操作 | O(30,000) |
| 空间复杂度 | 1024 × 32 = 32,768 个元素 | O(32,768) |
| 实际内存使用 | 262,144 bytes | ≈ 0.26 MB |

**与其他解法的比较**

|  |  |  |  |
| --- | --- | --- | --- |
| 解法类型 | 时间复杂度 | 空间复杂度 | 特点 |
| 2D DP解法 | O(30,000) | O(32,768) | 直观，易于理解 |
| 1D DP解法 | O(30,000) | O(1,024) | 空间优化，但逻辑稍复杂 |
| DFS + Memoization | O(30,000) | O(32,768) + 递归栈 | 递归开销，可能栈溢出 |

**实际性能考虑**

**时间效率**：30,000次操作在现代计算机上可以在毫秒级完成

**空间效率**：0.26MB的内存占用完全可以接受

**可读性**：2D DP解法比1D DP更容易理解和调试

**适用性**：对于这个问题规模，2D DP是完全可行的解决方案

这种复杂度在LeetCode等编程竞赛中是完全可接受的，特别是考虑到问题的约束条件（数字范围1-30，输入长度最多10^4）。

**💡 Step 5: 2D DP to 1D DP Solutio**n

接下来可以把2D DP压缩成1D DP吗？需要详细的注释和对for loop的方向，初始值的设定之类的解释

**根据前面分析的空间压缩原理，你能重新修正一下下面这个错误的1D DP吗？**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numberOfGoodSubsets(int[] nums) {

        // 质数列表（最多10个质数）

        int[] primes = new int[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

        int nPrimes = primes.length;

        int totalMasks = 1 << nPrimes; // 所有可能的质数组合状态数

        // 预处理：计算每个数字的质因数掩码

        int[] numMasks = new int[31];

        int[] freq = new int[31];

        // 统计每个数字的出现频率

        for (int num : nums) {

            if (num <= 30) freq[num]++;

        }

        // 为每个数字计算质因数掩码

        for (int num = 2; num <= 30; num++) {

            if (freq[num] == 0) continue;

            // 跳过有重复质因数的数字（平方数等）

            if (num % 4 == 0 || num % 9 == 0 || num % 25 == 0) continue;

            int mask = 0;

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

                if (num % primes[i] == 0) {

                    mask |= (1 << i);

                }

            }

            numMasks[num] = mask;

        }

        // 1D DP数组：dp[mask]表示当前状态下好子集的数量

        long[] dp = new long[totalMasks];

        // 初始化：对应DFS中的base case（num > 30）

        // 当没有数字可选时，只有空集一种可能

        dp[0] = 1; // 空集

        // 处理数字1（特殊处理，因为它不包含任何质因数）

        long ones = pow(2, freq[1]); // 1可以任意加入或不加入任何子集

        // 按数字从大到小处理（遵循"顶底之术"的逆序原则）

        // 注意：这里从30开始向下到2，与DFS递归方向相反

        for (int num = 30; num >= 2; num--) {

            if (freq[num] == 0 || numMasks[num] == 0) continue;

            int numMask = numMasks[num];

            // 关键：必须逆序遍历mask，以避免覆盖还需要使用的状态

            // 这是因为dp[mask]的计算依赖于dp[newMask]，其中newMask = mask | numMask

            // 逆序遍历确保在计算dp[mask]时，dp[newMask]还是上一轮（num+1）的状态

            for (int mask = totalMasks - 1; mask >= 0; mask--) {

                // 只有当当前数字的质因数与已选质因数不冲突时，才能选择该数字

                if ((mask & numMask) == 0) {

                    int newMask = mask | numMask;

                    // 状态转移方程：

                    // 不选当前数字：保持原状态dp[mask]

                    // 选当前数字：增加freq[num] \* dp[newMask]种可能

                    dp[mask] = (dp[mask] + freq[num] \* dp[newMask]) % MOD;

                }

            }

        }

        // 计算结果：dp[0]包含空集，需要减去

        long result = (dp[0] - 1 + MOD) % MOD;

        // 乘以1的可能组合数

        result = result \* ones % MOD;

        return (int) result;

    }

    // 快速幂算法：计算base^exponent mod MOD

    private long pow(long base, long exponent) {

        long result = 1;

        while (exponent > 0) {

            if (exponent % 2 == 1) {

                result = result \* base % MOD;

            }

            base = base \* base % MOD;

            exponent /= 2;

        }

        return result;

    }

}

**之前的错误在于：**

Input

nums = [1,2,3,4]

Output = 0

Expected =6

**要求是维持for loop从num = 30到num = 2，之前修正过的答案改变了for loop的方向，num变成了从2到30，并且符合你所说的压缩规则，并给出完整解释**

首先，用户提供了一个1D DP解法，但该解法对于输入nums = [1,2,3,4]输出0，而期望是6。用户要求维持for循环从num = 30到num = 2的方向，但之前的修正改变了for循环的方向（从2到30）。用户希望我基于空间压缩原理重新修正这个1D DP解法，同时保持num从30到2的循环方向。

原解法错误的原因在于状态转移方程。在原解法中，当处理数字num时，对于每个mask，如果mask与num的掩码不冲突，则更新dp[mask] += freq[num] \* dp[newMask]，其中newMask = mask | numMask。这种更新方式是不正确的，因为dp[newMask]表示的是在包含当前数字num之后的状态，但在这个循环方向上，dp[newMask]可能还没有被更新为包含后续数字的状态。

回顾空间压缩原理：在DP中，当我们压缩空间时，需要确保在更新当前状态时，所依赖的状态是来自上一轮（即处理当前数字之前）的状态。在正确的解法中，我们通常是从小到大处理数字，并且逆序遍历mask，以避免覆盖还需要使用的状态。

但用户要求保持num从30到2的循环方向。这意味着我们需要重新思考状态转移的逻辑。

在DFS解法中，递归是从num=2开始，到num=31结束（base case）。在DP中，如果我们从num=30向下到2，这实际上是逆DFS的递归方向。在2D DP中，我们确实这样做了：从num=30到2，对于每个num，我们计算dp[mask][num]依赖于dp[mask][num+1]和dp[newMask][num+1]。

在1D DP中，我们想要压缩掉num维度，所以我们需要确保在处理num时，dp数组存储的是处理num+1之后的状态。也就是说，当我们处理num时，dp数组已经包含了从num+1到30的所有数字的处理结果。

因此，对于num从30到2的循环，我们需要更新dp数组，使得在处理num之后，dp[mask]表示从num到30的数字中，形成mask的好子集数量。

正确的状态转移方程应该是：

对于每个num，从大到小处理mask（逆序mask），对于每个mask，如果mask与numMask不冲突，则newMask = mask | numMask，然后更新dp[newMask] += freq[num] \* dp[mask]。但注意，这里dp[mask]是处理num之前的状态，即从num+1到30的状态。

然而，在原解法中，更新的是dp[mask]而不是dp[newMask]。这是错误的。

正确的做法应该是：当我们处理数字num时，我们想要将num加入到可能的好子集中。所以，对于每个mask，如果当前数字num可以加入到mask中（即不冲突），那么newMask = mask | numMask，那么newMask的方案数应该增加freq[num] \* dp[mask]（其中dp[mask]是未加入num时的方案数）。

但由于我们是从大到小处理num，并且dp数组初始化为base case（只有dp[0]=1），我们需要在处理每个num时，更新那些包含num的新状态。

所以，修正后的步骤：

初始化dp[0] = 1，其他为0。

对于num从30下降到2：

如果num无效，跳过。

获取num的掩码numMask。

逆序遍历mask（从totalMasks-1到0）：

如果dp[mask] > 0 且 (mask & numMask) == 0，则newMask = mask | numMask，然后更新dp[newMask] += freq[num] \* dp[mask]

最后，dp[0]包含空集，所以计算所有非空好子集时，需要求和dp[1]到dp[totalMasks-1]，然后乘以2的freq[1]次方。

但注意：在循环中，我们更新的是dp[newMask]，而不是dp[mask]。这样，当我们处理num时，我们实际上是在现有状态（包含num+1到30的数字）的基础上，添加num数字，从而得到新的状态。

让我们用nums = [1,2,3,4]测试一下。

数字2、3、4的有效性：

2: 有效，掩码为1（假设质数2对应第0位）

3: 有效，掩码为2（质数3对应第1位）

4: 无效，因为4%4==0，跳过。

所以只处理数字2和3。

初始化: dp[0]=1

处理num=3:

掩码numMask=2

逆序遍历mask:

mask=3: 冲突? 3&2=2≠0, 跳过

mask=2: 冲突? 2&2=2≠0, 跳过

mask=1: 冲突? 1&2=0, 不冲突 -> newMask=1|2=3 -> dp[3] += freq[3]*dp[1] = 1*0 =0

mask=0: 冲突? 0&2=0, 不冲突 -> newMask=0|2=2 -> dp[2] += freq[3]*dp[0] =1*1=1

所以处理后: dp[0]=1, dp[1]=0, dp[2]=1, dp[3]=0

处理num=2:

掩码numMask=1

逆序遍历mask:

mask=3: 冲突? 3&1=1≠0, 跳过

mask=2: 冲突? 2&1=0, 不冲突 -> newMask=2|1=3 -> dp[3] += freq[2]*dp[2] =1*1=1

mask=1: 冲突? 1&1=1≠0, 跳过

mask=0: 冲突? 0&1=0, 不冲突 -> newMask=0|1=1 -> dp[1] += freq[2]*dp[0] =1*1=1

所以处理后: dp[0]=1, dp[1]=1, dp[2]=1, dp[3]=1

然后，计算所有非空好子集：求和dp[1]到dp[3] = 1+1+1=3

然后乘以2的freq[1]次方：freq[1]=1（因为nums=[1,2,3,4]中有一个1），所以2^1=2，结果=3\*2=6，符合预期。

因此，修正后的代码应该更新dp[newMask]而不是dp[mask]。

同时，注意在逆序遍历mask时，我们需要确保dp[mask]是处理当前num之前的状态，所以逆序mask是必要的，因为newMask大于mask，如果我们顺序遍历mask，可能会先更新小的mask，然后更新大的mask时使用了已经更新过的dp[mask]，导致重复计算。逆序mask可以避免这个问题。

所以，修正后的代码：

保持num从30到2的循环，但内部更新dp[newMask]。

另外，初始化dp[0]=1。

最后，结果需要求和所有非零mask的dp值（除了dp[0]），然后乘以2的freq[1]次方。

现在，编写修正后的代码。

根据您的需求，我重新修正了1D DP解法，保持了num从30到2的循环方向。修正的核心在于状态转移方程：原来错误地更新了dp[mask]，而正确的做法是更新dp[newMask]，其中newMask = mask | numMask。这样，当处理每个数字num时，我们将num添加到现有的子集状态中，从而得到新的状态。同时，内层循环逆序遍历mask以确保在更新dp[newMask]时，所依赖的dp[mask]是处理当前数字之前的状态。

以下是修正后的代码，包含详细注释：

class Solution {

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

public int numberOfGoodSubsets(int[] nums) {

// 质数列表：30以内的质数共有10个

int[] primes = new int[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};

int nPrimes = primes.length;

int totalMasks = 1 << nPrimes; // 所有可能的质数组合状态数（2^10 = 1024）

// numMasks[i]：数字i的质因数掩码（bitmask）

int[] numMasks = new int[31];

// freq[i]：数字i在输入数组中的出现频率

int[] freq = new int[31];

// 统计每个数字的出现频率（只处理1到30）

for (int num : nums) {

if (num <= 30) {

freq[num]++;

}

}

// 预处理：计算每个数字（2到30）的质因数掩码

for (int num = 2; num <= 30; num++) {

if (freq[num] == 0) continue; // 跳过未出现的数字

// 跳过有重复质因数的数字（如4、8、9、12、16、18、20、24、25、27、28）

if (num % 4 == 0 || num % 9 == 0 || num % 25 == 0) continue;

int mask = 0;

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

if (num % primes[i] == 0) {

mask |= (1 << i); // 设置对应质数的位

}

}

numMasks[num] = mask;

}

// DP数组：dp[mask]表示形成质数组合为mask的好子集的数量

long[] dp = new long[totalMasks];

// 初始化：空集有一种方案（mask=0）

dp[0] = 1;

// 处理数字30 down to 2：保持从大到小的顺序（与DFS递归方向相反）

for (int num = 30; num >= 2; num--) {

if (freq[num] == 0 || numMasks[num] == 0) continue; // 跳过无效数字

int numMask = numMasks[num];

// 关键：逆序遍历所有mask（从大到小），以避免覆盖当前数字处理前的状态

// 因为newMask = mask | numMask 总是大于或等于mask，逆序确保在更新dp[newMask]时，dp[mask]是未更新的值

for (int mask = totalMasks - 1; mask >= 0; mask--) {

// 如果当前状态mask有方案（dp[mask] > 0）且与当前数字的质因数不冲突

if (dp[mask] != 0 && (mask & numMask) == 0) {

int newMask = mask | numMask;

// 状态转移：选择当前数字，newMask的方案数增加 freq[num] \* dp[mask]

dp[newMask] = (dp[newMask] + freq[num] \* dp[mask]) % MOD;

}

}

}

// 计算所有非空好子集的数量（排除mask=0的空集）

long total = 0;

for (int i = 1; i < totalMasks; i++) {

total = (total + dp[i]) % MOD;

}

// 处理数字1：数字1不包含任何质因数，可以任意添加到任何子集中而不影响质因数组合

// 因此，总方案数乘以2的freq[1]次方（每个1都可以选择加入或不加入）

long ones = pow(2, freq[1]);

total = total \* ones % MOD;

return (int) total;

}

// 快速幂算法：计算 base^exponent mod MOD

private long pow(long base, long exponent) {

long result = 1;

while (exponent > 0) {

if (exponent % 2 == 1) {

result = result \* base % MOD;

}

base = base \* base % MOD;

exponent /= 2;

}

return result;

}

}

**修正说明：**

**状态转移方程**：原解法错误地更新了dp[mask]，修正后更新dp[newMask]，这符合逻辑：当选择数字num时，从状态mask转移到状态newMask，方案数增加freq[num] \* dp[mask]。

**循环方向**：保持num从30到2的循环方向，内层循环逆序遍历mask，以确保状态正确更新。

**测试验证**：针对输入nums = [1,2,3,4]，计算过程如下：

数字4无效（跳过），处理数字3和2。

处理数字3后，dp[2] = 1（状态2表示包含数字3）。

处理数字2后，dp[1] = 1（状态1表示包含数字2），dp[3] = 1（状态3表示包含数字2和3）。

非空好子集数量为dp[1] + dp[2] + dp[3] = 3，乘以2^(freq[1]) = 2，结果为6，符合预期。

此解法已修正原错误，同时保持了您要求的循环方向。

**你看看这篇网页上写的是否有帮助？**

<https://afteracademy.com/blog/when-to-convert-a-2d-dp-array-to-1d-dp-array-and-how/>

你提供的这篇网页文章对理解何时以及如何将2D DP数组转换为1D DP数组非常有帮助。它很好地补充了我们之前讨论的内容，特别是关于**降维的条件**和**逆序更新**的关键点。

结合该文章和我之前的解答，以下是核心总结：

**✅ 何时可以降维（条件）**

当DP状态转移方程中，计算dp[i][j]时**仅依赖于**：

同一行的前一个状态（例如dp[i][j-1]）

上一行的状态（例如dp[i-1][j]、dp[i-1][k] (k <= j)）

这意味着当前行（或当前阶段）的状态只与**上一行（或前一阶段）的状态**和**当前行已计算的状态**有关。一旦处理完某个状态，上一行对应的状态在后续计算中就不再需要，因此可以覆盖它，从而将空间复杂度从 O(N^2) 降低到 O(N)。

**🔁 如何降维（关键：逆序更新**）

降维最常用**滚动数组**技巧。关键在于**内层循环往往需要逆序（从大到小）更新**。

**为什么逆序？**

为了防止覆盖后续计算还需要用到的**上一行（或旧状态）** 的值。

**顺序更新（从小到大）**：当你计算dp[j]（新值）时，你可能会覆盖掉dp[j]（旧值，即上一行的dp[i-1][j]）。然而，在计算后面的dp[k]（k > j）时，可能还需要用到原来的dp[i-1][j]（但现在已经被覆盖成了dp[i][j]），从而导致错误。

**逆序更新（从大到小）**：当你计算dp[j]（新值）时，你所依赖的dp[k]（k < j）如果在本轮已经计算过，则是**本轮的新值**（通常是允许的，如果状态转移依赖本轮已更新的状态），而你所依赖的dp[m]（m >= j）的值**还是上一行的旧值**（因为你在逆序更新，还没碰到它们），这正好符合原始2D DP方程的依赖关系，保证了正确性。

**📊 降维前后对比（以经典0-1背包为例**）

|  |  |  |
| --- | --- | --- |
| 特征 | 2D DP | 1D DP (空间优化后) |
| **状态定义** | dp[i][j]  处理前i件物品，容量j的最大价值 | dp[j]  容量j的最大价值 |
| **初始化** | dp[0][j] = 0 | dp[j] = 0 |
| **状态转移方程** | dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) | dp[j] = max(dp[j], dp[j - w[i]] + v[i]) |
| **更新方向** | 外层i正序，内层j正序或逆序均可 | **外层i正序，内层j必须逆序**  （从大到小） |
| **空间复杂度** | O(N \* W) | O(W) |
| **依赖关系** | 清晰呈现所有状态 | 需要理解  **逆序**  是为了保护  **上一行**  的未更新状态 |

**🧮 逆序更新在LeetCode 1994中的体**现

在你提供的LeetCode 1994的1D DP解法中：

for (int num = 2; num <= 30; num++) {

if (freq[num] == 0 || numMasks[num] == 0) continue;

int numMask = numMasks[num];

for (int mask = totalMasks - 1; mask >= 0; mask--) { // 注意：内层循环逆序

if (dp[mask] != 0 && (mask & numMask) == 0) {

int newMask = mask | numMask;

dp[newMask] = (dp[newMask] + freq[num] \* dp[mask]) % MOD;

}

}

}

这里的**逆序更新 (mask从大到小)** 至关重要。

**目的**：确保在更新dp[newMask]时，所读取的dp[mask]值是**处理当前数字num之前**的值（即“上一轮”循环中对应于更小num的值）。

**原因**：newMask是由mask和当前数字的质因数掩码numMask做**或运算**得到的，因此newMask >= mask。如果顺序遍历mask，当计算到较大的mask时，其依赖的较小mask可能已经被当前数字num更新过了，这会导致重复计算。逆序遍历可以避免这个小mask被当前num提前更新，保证了数据的正确性。

**⚠️ 注意事项**

**可读性**：降维后的DP代码通常不如2D DP直观。在理解、调试或与他人交流时，有时先写出2D版本再优化是更好的选择。

**并非所有2D DP都可降维**：如果状态转移方程依赖**同一行**的多个后续状态或更复杂的历史状态，可能无法简单降维到1D。

**变量覆盖**：降维的核心是**安全地覆盖**不再需要的数据。逆序循环是保证覆盖安全性的常用手段。

希望这些解释能帮助你更好地理解DP降维的时机和方法。

**Refer to**

[L1125.Smallest Sufficient Team (Ref.L77,L1986,L2397)](note://WEBdc520ac7caa3b8b82dc86eee72337a3c)

[L2305.Fair Distribution of Cookies (Ref.L410,L473,L698)](note://WEBccd4ba98d71625f4c769ccee45f02d79)

[When to Convert a 2-D DP Array To 1-D DP Array And How](note://74A2DD972C0F4106B2A5D870091EBE07)