2741. Special Permutations

Medium · Bit Manipulation · Array · Bitmask   Leetcode Link

Problem Description

In this problem, we are given a zero-indexed array `nums` that consists of `n` distinct positive integers. We are asked to find the number of permutations of `nums` that are "special." A permutation is considered special if it meets the following condition: for every adjacent pair of elements in the permutation `(nums[i], nums[i+1])` where `0 <= i < n - 1`, the pair must satisfy either `nums[i] % nums[i+1] == 0` or `nums[i+1] % nums[i] == 0`. In other words, each adjacent pair must be such that one number is a divisor of the other. We need to return the total number of such special permutations modulo `10^9 + 7` to handle the large number that might result from the calculation.

Intuition

To solve this problem, we can use dynamic programming with bitmasking to handle the permutations efficiently. We consider each bit in a bitmask to represent whether a particular element of the array `nums` has already been used in the permutation or not. Since `nums` contains `n` distinct elements, we can represent the state of the permutation using `n` bits, which leads to a total of `2^n` states (`m` in the code).

The main idea behind the solution is the following steps:

1. Initialize a 2D array `f` with dimensions `m * n` to store the number of ways to achieve a valid permutation using some of the elements of `nums`. The row represents the bitmask (which elements have been used), and the column `j` represents the last element used in the permutation.

2. Loop through all possible bitmask states `i`, and for each, we calculate the number of valid special permutations ending with each element `x` (where `i` has the `j`th bit set, indicating that `x` is the last element of the permutation). To do this:

   - Find `ii` by subtracting the current element `x` from the bitmask, resulting in the bitmask of the rest of the elements.

   - If `ii` equals zero, it means that this is the starting element of the permutation; set `f[i][j]` to 1.

   - Otherwise, iterate over the array again using index `k` and element `y` representing the second-to-last element in the permutation, if the current element `x` can divide `y` or vice versa, increment the count in `f[i][j]` by the number of ways you can form a permutation that ends with `y`; update it as `f[i][j] = (f[i][j] + f[ii][k]) % mod`.

3. After filling up the table, the sum of counts in the last row of `f` will give the total number of special permutations, because the last row represents the state where all elements have been used.

This dynamic programming approach effectively counts all valid permutations while ensuring that we do not revisit configurations we have already considered, which greatly optimizes the solution.

Solution Approach

The solution provided uses dynamic programming and bitmasking to find the number of special permutations.

Here's the breakdown of the implementation:

- First, the `mod` variable is set to `10^9 + 7` to ensure all calculations are done under modulo arithmetic to prevent integer overflow.

- The variable `n` is assigned the length of the `nums` array. And `m` is calculated as `1 << n`, which equals `2^n` and represents the total number of states each element can be in (used or unused).

- The `f` array is initialized as a two-dimensional list of size `m * n`, with all elements set to zero. This array holds the number of special permutations possible for each combination of elements.

- The outermost loop iterates over all possible states, represented by the bitmask `i`. The bitmask `i` has `n` bits, with each bit corresponding to whether a particular number in `nums` has been included in the permutation (`1`) or not (`0`).

- Within the outer loop, we iterate over all elements `x` in `nums` with their corresponding index `j`. If the `j`th bit of bitmask `i` is set (`i >> j & 1`), it indicates that number `x` is being considered as the last number in the permutation.

- For each such `x`, we calculate `ii`, which is the bitmask state without `x` (done by `i ^ (1 << j)`).

- If `ii` is zero (`ii == 0`), it means we are considering permutations where `x` is the only number so far. Thus, there is only one way to form such a permutation and `f[i][j]` is set to `1`.

- If `ii` is not zero, we look at all other numbers `y` in `nums` that could come before `x` in the permutation. We use the second-to-last number's `k` index to check divisibility between `y` and `x` (either `x % y == 0` or `y % x == 0`). If the divisibility condition is met, it means `y` can come before `x` in a permutation. We add the count of special permutations ending with `y` (stored in `f[ii][k]`) to the count of permutations ending with `x` (`f[i][j]`), applying modulo every time to keep numbers within bounds.

- After completing both loops, the array `f[-1]` contains counts of all special permutations for each `nums` element as the last number. The sum of all these counts gives us the total number of special permutations. The final result is taken modulo `10^9 + 7` to get the answer within the required bounds.

This algorithm is efficient because it takes advantage of the structure of permutations and divisibility to prune the search space and count only valid special permutations, avoiding the need to generate and test every permutation explicitly, which would be impractical for large `n`.

Example Walkthrough

Consider an array `nums` with distinct positive integers like `[1, 2, 3]`. Let's walk through the solution approach using this set of numbers to understand how the dynamic programming and bitmasking technique is applied.

1. First, we set `n` equal to the length of `nums`, which in this case is `3`. We calculate `m` as `1 << n` (which is `2^n`), amounting to `8` for this example because there are `2^3` possible states for three distinct numbers.

2. Initialize the DP array `f` with size `m * n`. In this case, `f` will be an 8 by 3 array filled initially with zeros. The state of `f` will look as follows after initialization, where `f[i][j]` represents the number of ways to form permutations of size `i` with `j` as the last number:

```
f = [
  [0, 0, 0],  // state 0: no elements used
  [0, 0, 0],  // state 1: only num[0] used
  [0, 0, 0],  // state 2: only num[1] used
  [0, 0, 0],  // state 3: num[0] and num[1] used
  [0, 0, 0],  // state 4: only num[2] used
  [0, 0, 0],  // state 5: num[0] and num[2] used
  [0, 0, 0],  // state 6: num[1] and num[2] used
  [0, 0, 0],  // state 7: all numbers used
]
```

3. Start iterating over all possible states (`i`) from `1` to `m - 1`. For each state `i`, we look for all numbers `x` in `nums` that could potentially be the last number in the permutation.

4. For each number `x` (with index `j`), if `x` is included in state `i` (checked by `i >> j & 1`), we calculate the state `ii` by removing `x` from `i` (via `i ^ (1 << j)`).

5. If `ii` is 0, we know that `x` is at the start of the permutation, hence `f[i][j]` is set to `1`.

6. If `ii` is not 0, check all numbers `y` (with index `k`) that are not `x` and occur before `x` in the permutation. If `x` divides `y` or `y` divides `x`, add the number of permutations that end with `y` (`f[ii][k]`) to the current permutation count (`f[i][j]`).

Applying this process to our example, let's consider the state where `i = 3` (binary `011`), which means `nums[0]` (value `1`) and `nums[1]` (value `2`) are included. Since `1` is a divisor of `2`, we can form a permutation `[1, 2]`. Thus, `f[3][1]` (where 3 represents the state `011` and 1 represents index of number `2` in `nums`) will increment by 1. Iterate this procedure for each state and number pair.

Once the table is filled, sum up the last row's values to get the total number of special permutations. For our example, `f[7]` (which represents all numbers being used) yields the count for each number as the last in a special permutation. Since all numbers in `nums` are divisible by 1, any permutation where `1` is the first element will be valid, leading to the total count being the sum of the last row of `f` modulo `10^9 + 7`.

This small example demonstrates the use of dynamic programming and bitmasking for efficiently calculating the number of special permutations. Each state is systematically built upon the previous states, ensuring that all permutations are accounted for and that each permutation is valid according to the given divisibility condition.

Python Solution

```python
class Solution:
    def specialPerm(self, nums: List[int]) -> int:
        mod = 10**9 + 7  # Modulus for keeping values within integer limits
        num_count = len(nums)  # Total number of elements in nums
        bitmask_limit = 1 << num_count  # Compute the limit for bitmask (2^n combinations)
        dp = [[0] * num_count for _ in range(bitmask_limit)]  # Initialize the dynamic programming table

        # Build the dynamic programming table
        for bitmask in range(1, bitmask_limit):  # Loop over all possible combinations
            for j, x in enumerate(nums):  # Iterate over numbers in nums with index j
                if bitmask >> j & 1:  # Check if element at index j is included in the combination
                    prev_bitmask = bitmask ^ (1 << j)  # Calculate bitmask of the previous state
                    if prev_bitmask == 0:  # If it's the first number in the combination
                        dp[bitmask][j] = 1  # There is one way to pick it
                        continue
                    # Calculate number of special permutations ending with the number at index j
                    for k, y in enumerate(nums):  # Iterate over all other numbers with index k
                        if x % y == 0 or y % x == 0:  # Check if x and y are in a special relation
                            dp[bitmask][j] = (dp[bitmask][j] + dp[prev_bitmask][k]) % mod
        # The answer is the sum of all permutations ending with any number in nums
        return sum(dp[-1]) % mod  # Return the sum of permutations for the final state

```

Java Solution

```java
class Solution {
    public int specialPerm(int[] nums) {
        final int MOD = (int) 1e9 + 7; // Define the modulus for the output to prevent overflow
        int n = nums.length; // Store the length of the input array
        int m = 1 << n; // Calculate 2^n, which will be used for creating subsets
        int[][] dp = new int[m][n]; // Create a dp table for memoization

        // Iterate over all subsets of the given array
        for (int i = 1; i < m; ++i) {
            // Iterate over each number in the current subset
            for (int j = 0; j < n; ++j) {
                // Check if the j-th number is in the current subset
                if ((i >> j & 1) == 1) {
                    int prevSubset = i ^ (1 << j); // Create the previous subset by removing j-th number
                    if (prevSubset == 0) {
                        dp[i][j] = 1; // If the subset is empty, initialize with 1
                        continue;
                    }
                    // Iterate over each number in the previous subset
                    for (int k = 0; k < n; ++k) {
                        // Apply the given condition (nums[j] % nums[k] == 0 or nums[k] % nums[j] == 0)
                        if (nums[j] % nums[k] == 0 || nums[k] % nums[j] == 0) {
                            // Update the dp table using the results from the previous subset
                            dp[i][j] = (dp[i][j] + dp[prevSubset][k]) % MOD;
                        }
                    }
                }
            }
        }

        int answer = 0; // Initialize the answer

        // Sum up all possibilities for the full set
        for (int x : dp[m - 1]) {
            answer = (answer + x) % MOD;
        }

        return answer; // Return the calculated answer
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int specialPerm(vector<int>& nums) {
        const int MOD = 1e9 + 7; // Constant to hold the modulo value
        int n = nums.size(); // Size of the input vector
        int upperBound = 1 << n; // Equivalent to 2^n, decides the upper bound for bitmasking
        int dp[upperBound][n]; // 2D array for dynamic programming
        memset(dp, 0, sizeof(dp)); // Initialize the array with zeros

        // Loop through all bitmask possibilities (from 1 to 2^n - 1)
        for (int mask = 1; mask < upperBound; ++mask) {
            // Iterate over each number in nums
            for (int j = 0; j < n; ++j) {
                // Check if the j-th number is included in the current combination (mask)
                if ((mask >> j & 1) == 1) {
                    int prevMask = mask ^ (1 << j); // Calculate previous mask by removing j-th bit

                    if (prevMask == 0) {
                        dp[mask][j] = 1; // Base case for DP: only one number in the sequence
                        continue;
                    }

                    // Iterate over each possibility for the previous number in the sequence
                    for (int k = 0; k < n; ++k) {
                        // Check if the current and previous numbers are compatible (divisible)
                        if (nums[j] % nums[k] == 0 || nums[k] % nums[j] == 0) {
                            // If they are, increment the DP value for the current combination and number
                            dp[mask][j] = (dp[mask][j] + dp[prevMask][k]) % MOD;
                        }
                    }
                }
            }
        }

        // Calculate the final answer by summing up the valid sequences ending with each number
        int answer = 0;
        for (int val : dp[upperBound - 1]) { // Iterate over the last row of the DP array
            answer = (answer + val) % MOD; // Accumulate values considering the modulo
        }

        // Return the computed answer
        return answer;
    }
};

```

Typescript Solution

```typescript
// Defining constants and utility functions
const MOD = 1e9 + 7;

// Utility function to count the set bits in a bitmask
function countSetBits(mask: number): number {
  let count = 0;
  while (mask) {
    count += mask & 1;
    mask >>= 1;
  }
  return count;
}

// Dynamic programming function to calculate the special permutation count
function specialPerm(nums: number[]): number {
  const n: number = nums.length;
  const upperBound: number = 1 << n;
  let dp: number[][] = new Array(upperBound).fill(null).map(() => new Array(n).fill(0));

  // Iterate through all bitmask possibilities (from 1 to 2^n - 1)
  for (let mask = 1; mask < upperBound; ++mask) {
    // Iterate over each number in nums
    for (let j = 0; j < n; ++j) {
      // Check if the j-th number is included in the current combination (mask)
      if ((mask >> j & 1) === 1) {
        let prevMask = mask ^ (1 << j); // Calculate previous mask by removing j-th bit

        if (prevMask === 0) {
          dp[mask][j] = 1; // Base case for DP: only one number in the sequence
          continue;
        }

        // Iterate over each possibility for the previous number in the sequence
        for (let k = 0; k < n; ++k) {
          // Check if the current and previous numbers are compatible (divisible)
          if (nums[j] % nums[k] === 0 || nums[k] % nums[j] === 0) {
            // If they are, increment the DP value for the current combination and number
            dp[mask][j] = (dp[mask][j] + dp[prevMask][k]) % MOD;
          }
        }
      }
    }
  }

  // Calculate the final answer by summing up the valid sequences ending with each number
  let answer = 0;
  for (let val of dp[upperBound - 1]) {
    answer = (answer + val) % MOD; // Accumulate values considering the modulo
  }

  // Return the computed answer
  return answer;
}

// Example usage:
// const nums = [1, 2, 3];
// console.log(specialPerm(nums)); // Output the number of special permutations

```

Time and Space Complexity

The given Python function `specialPerm` calculates a special permutation count under certain constraints. We'll analyze the time and space complexity of this function.

Time Complexity

The function involves multiple nested loops:

- The outermost loop runs through all subsets of indices formed from the `nums` array, having a total of `2^n` iterations (`n` is the length of `nums`), because `m = 1 << n` which is `2^n`.
- The second loop iterates over each element in `nums`, contributing a factor of `n`.
- The innermost loop runs conditionally when the bit at position `j` is set in the current subset `i` (`i >> j & 1`). When this condition is met, it attempts to iterate over `nums` again to update `f[i][j]`. However, this does not bring another factor of `n` because it will only iterate up to `n` times in the worst-case scenario, not for each subset or combination.

Considering the bit manipulation operations and modulo operations inside the loops are constant time, the time complexity is derived primarily from these loops.

Therefore, the total time complexity is `O(n * 2^n)` since the two significant factors are the iterations over the subsets (`2^n`) and the elements in `nums` (`n`).

Space Complexity

The space usage in the function comes from:

- The list `f`: a 2D list of dimensions `2^n` by `n`. It holds the count of permutations that end with each element `j` for all subsets of `nums`.
- Auxiliary space: negligible constant space used for loop indices and temporary variables.

The space complexity is thus determined by the size of `f`, which is `O(n * 2^n)`.

In conclusion, the function has a **time complexity** of `O(n * 2^n)` and a **space complexity** of `O(n * 2^n)`.