1711. Count Good Meals

Medium · Array · Hash Table · Leetcode Link

Problem Description

In this problem, we are given an array called `deliciousness` where each element represents the deliciousness level of a specific food item. We are tasked with finding combinations of exactly two different food items such that their total deliciousness equals a power of two. These combinations are called "good meals". To clarify, two food items are considered different if they are at different indices in the array, even if their deliciousness values are identical.

The output should be the number of good meals that we can create from the given list, and because this number could be very large, we are instructed to return it modulo `10^9 + 7`. A modular result is a standard requirement in programming challenges to avoid overflow issues with high numbers.

Intuition

To solve this problem, we can use a hash map (in Python, this is a dictionary) to store the frequency of each deliciousness value. We iterate over all possible powers of two (up to the 21st power since the input constraint is `2^20`), and within this iteration, we check each unique deliciousness value. For each of these values, say `a`, we look for another value `b` such that `a + b` equals the current power of two we're checking against. This value `b` must be `2^i - a`.

Here's the step-by-step breakdown of our approach:

1. Initialize a `Counter` for the array `deliciousness` to keep track of the number of occurrences of each value of deliciousness.
2. Initialize a variable `ans` to keep track of the total number of good meals.
3. Loop through all the powers of two up to `2^21`. This covers the range of possible sums of the two deliciousness values.
4. For each deliciousness value `a` found in the hash map we created, calculate `b = 2^i - a`.
5. If `b` is also in the hash map and `a != b`, then we have found a pair of different food items whose deliciousness sums to a power of two.
   - In this case, we add to `ans` the product of the number of times `a` appears and the number of times `b` appears.
6. If `a == b`, we have found a pair of the same food items, and we add to `ans` the product of the number of times `a` appears with `m - 1` because you cannot count the same pair twice.
7. Since each pair will be counted twice during this process (once for each element as `a` and once as `b`), we must divide the total answer by 2 to get the correct count.
8. Finally, take the modulo of the count by `10^9 + 7` to get our answer within the required range.

It is important to note that we use a bit manipulation trick—`1 << i`—to quickly find the i-th power of two, which greatly reduces the time complexity.

Solution Approach

The implementation uses a `Counter` from the Python `collections` module, which is essentially a hash map or dictionary designed to count the occurrences of each element in an iterable. This data structure is ideal for keeping track of the frequency of deliciousness in the given `deliciousness` array.

Here's a step-by-step guide to how the algorithm and data structures are used in the solution:

1. First, a `Counter` object named `cnt` is created to store the frequency of each value of deliciousness from the array.

2. We set `ans` to 0 as an accumulator for the total number of good meals.

3. We loop through all possible powers of two up to `2^21` (specified as `22` in the `range(22)` because range goes up to but does not include the end value in Python). We need to cover `2^20` since it's the maximum sum according to the problem constraints regarding deliciousness.

4. Inside this loop, we calculate `s = 1 << i`, which is a bit manipulation operation that left-shifts the number 1 by `i` places, effectively calculating `2^i`.

5. With each power of two, we iterate over the items in the `Counter` object, where `a` is a deliciousness value from the array, and `m` is its frequency (the number of times it appears).

6. We then calculate `b = s - a`, to find the complementary deliciousness value that would make a sum of `s` with `a`.

7. We check if `b` is present in our `Counter`. If it is, we have a potential good meal. However, we must be mindful of counting pairs correctly:

   - If `a` equals `b`, then we increment `ans` by `m * (m - 1)` because we can't use the same item twice, hence we consider the combinations without repetition.

   - If `a` does not equal `b`, then we increment `ans` by `m * cnt[b]`, considering all combinations between the occurrences of `a` and `b`.

8. After the loop, since every pair is counted twice (once for each of its two items, as both `a` and `b`), we divide `ans` by 2 to obtain the actual number of good meals.

9. Lastly, we apply modulo `10^9 + 7` to our result to handle the large numbers and prevent integer overflow issues as per the problem's requirement.

By utilizing a hash map (Counter) and iterating over the powers of two, the solution effectively pairs up food items while avoiding nested loops that would significantly increase the time complexity. This allows for an efficient solution to the problem.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the array `deliciousness = [1, 3, 5, 7, 9]`.

1. We first create a `Counter` object from the `deliciousness` array, which will count the frequency of each value. In this case, all values are unique and appear once, so our counter (`cnt`) would look like this: `{1:1, 3:1, 5:1, 7:1, 9:1}`.

2. We initialize `ans` to 0 to begin counting the number of good meals.

3. Now, we loop through all possible powers of two up to `2^21`. For simplicity, consider that we just check up to `2^3` (or 8) for this example. Our powers of two are therefore `[1, 2, 4, 8]`.

4. For the power of 2 (say `s = 2^i`), we loop through the `Counter` object items. Let's first choose `s = 4` and consider the entries in our counter. We have `a` as the key and `m` as the frequency (always 1 in this case).

5. For each `a`, we calculate `b = s - a` to find the complementary deliciousness value.

6. We check if `b` exists in our counter. If it does, and `a` is not equal to `b`, then we found a good meal pair and increment `ans` by the product of their frequencies (since `m` is always 1, it would just be incremented by 1).

7. However, if `a` equals `b`, then we increment `ans` by `m * (m - 1) / 2` which is zero in this case, as there's only one of each item.

8. We continue this process for all powers of two. Given our example and `s = 4`, we notice that pairs `(1, 3)` and `(3, 1)` form good meals because `1+3=4`, which is a power of two. Both pairs are counted separately, so `ans` is incremented twice.

9. At the end of the loop, assuming we found no other pairs for other powers of two, `ans` would be 2 (since we found the `(1, 3)` pair twice). We then divide it by 2 to correct for the double counting, leaving us with a final `ans` of 1.

10. Lastly, we apply modulo `10^9 + 7` to our result. Since our `ans` is much less than `10^9 + 7`, it remains unchanged.

Our final answer is that there is 1 good meal combination in the `deliciousness` array `[1, 3, 5, 7, 9]` when considering powers of two up to `2^3`. If we extended it to `2^21`, there may be more combinations available.

Python Solution

```python
from collections import Counter

class Solution:
    def countPairs(self, deliciousness: List[int]) -> int:
        # Define the modulus for the final answer due to large numbers
        mod = 10**9 + 7

        # Create a counter to count occurrence of each value in deliciousness
        count = Counter(deliciousness)

        # Initialize the answer to zero
        answer = 0

        # Iterate through powers of two from 2^0 to 2^21
        for i in range(22):
            power_two_sum = 1 << i  # Calculate the power of two for current i

            # Iterate through each unique value in deliciousness
            for value, frequency in count.items():
                complement = power_two_sum - value  # Find the complement

                # If the complement is also in deliciousness
                if complement in count:

                    # If value and complement are the same, choose pairs from the same number (frequency choose 2)
                    if value == complement:
                        answer += frequency * (frequency - 1) // 2
                    else:
                        # If they are different, we count all unique pairs (frequency_a * frequency_b)
                        answer += frequency * count[complement]

        # Divide by 2 because each pair has been counted twice
        answer //= 2

        # Return the final answer modulo 10^9 + 7
        return answer % mod

```

Java Solution

```java
class Solution {
    // Define the modulus value for large numbers to avoid overflow
    private static final int MOD = (int) 1e9 + 7;

    // Method to count the total number of pairs with power of two sums
    public int countPairs(int[] deliciousness) {
        // Create a hashmap to store the frequency of each value in the deliciousness array
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int value : deliciousness) {
            frequencyMap.put(value, frequencyMap.getOrDefault(value, 0) + 1);
        }

        long pairCount = 0; // Initialize the pair counter to 0

        // Loop through each power of 2 up to 2^21 (because 2^21 is the closest power of 2 to 10^9)
        for (int i = 0; i < 22; ++i) {
            int sum = 1 << i; // Calculate the sum which is a power of two
            for (var entry : frequencyMap.entrySet()) {
                int firstElement = entry.getKey();   // Key in the map is a part of the deliciousness pair
                int firstCount = entry.getValue();   // Value in the map is the count of that element
                int secondElement = sum - firstElement;   // Find the second element of the pair

                // Check if the second element exists in the map
                if (!frequencyMap.containsKey(secondElement)) {
                    continue; // If it doesn't, continue to the next iteration
                }

                // If the second element exists, increment the pair count
                // If both elements are the same, we must avoid counting the pair twice
                pairCount += (long) firstCount * (firstElement == secondElement ? firstCount - 1 : frequencyMap.get(secondElement));
            }
        }

        // Divide the result by 2 because each pair has been counted twice
        pairCount >>= 1;

        // Return the result modulo MOD to get the answer within the range
        return (int) (pairCount % MOD);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    const int MOD = 1e9 + 7;

    int countPairs(vector<int>& deliciousness) {
        // Create map to store the frequency of each deliciousness value
        unordered_map<int, int> countMap;
        // Populate the frequency map
        for (int& value : deliciousness) {
            ++countMap[value];
        }
        long long totalPairs = 0;  // Using long long to prevent overflow

        // Iterate over all possible powers of two up to 2^21
        for (int i = 0; i < 22; ++i) {
            int sum = 1 << i;  // Current sum target (power of two)
            // Iterate over the frequency map to check for pairs
            for (auto& [deliciousValue, frequency] : countMap) {
                int complement = sum - deliciousValue;  // Complement to make a power of two
                // Check if complement exists in the map
                if (!countMap.count(complement)) continue;
                // If it's the same number, pair it with each other (except with itself)
                // Else multiply frequencies of the two numbers
                totalPairs += deliciousValue == complement ?
                              static_cast<long long>(frequency) * (frequency - 1) :
                              static_cast<long long>(frequency) * countMap[complement];
            }
        }

        totalPairs >>= 1;  // Each pair is counted twice, so divide by 2
        return totalPairs % MOD;  // Modulo operation to avoid overflow
    }
};

```

Typescript Solution

```typescript
// Define MOD constant for modulus operation to avoid overflow
const MOD = 1e9 + 7;

// Function to count pairs with sum that are power of two
function countPairs(deliciousness: number[]): number {
    // Create map to store the frequency of each deliciousness value
    const countMap = new Map<number, number>();

    // Populate the frequency map with deliciousness counts
    for (const value of deliciousness) {
        const count = countMap.get(value) || 0;
        countMap.set(value, count + 1);
    }

    let totalPairs = 0; // Using long to prevent overflow

    // Iterate over all possible powers of two up to 2^21
    for (let i = 0; i < 22; ++i) {
        const sum = 1 << i; // Current sum target (power of two)

        // Iterate over the frequency map to check for pairs
        for (const [deliciousValue, frequency] of countMap.entries()) {
            const complement = sum - deliciousValue; // Complement to make a power of two

            if (!countMap.has(complement)) continue; // Continue if complement does not exist

            // Calculate the total pairs
            // If it's the same number, combine it with each other (except with itself)
            // Else multiply frequencies of the two numbers
            totalPairs += deliciousValue === complement
                ? frequency * (frequency - 1)
                : frequency * (countMap.get(complement) as number);
        }
    }

    totalPairs /= 2; // Each pair is counted twice, so divide by 2

    // Return the number of pairs mod MOD to prevent overflow
    return totalPairs % MOD;
}

```

Time and Space Complexity

Time Complexity

The provided code has two nested loops. The outer loop is constant, iterating 22 times corresponding to powers of two up to `2^21`, as any pair of meals should have a sum that is a power of two for a maximum possible pair value of `2^(20+20) = 2^40`, and the closest power of two is `2^41`.

The inner loop iterates through every element `a` in the `deliciousness` list once. So, if `n` is the length of `deliciousness`, the inner loop has a time complexity of `O(n)`.

The `if` condition inside the inner loop checks if `b` exists in `cnt`, which is a `Counter` (essentially a dictionary), and this check is `O(1)` on average. The increment of `ans` is also `O(1)`.

So, multiplying the constant `22` by the `O(n)` complexity of the inner loop gives the total time complexity:

```
T(n) = 22 * O(n) = O(n)
```

Space Complexity

The `cnt` variable is a `Counter` that stores the occurrences of each item in `deliciousness`. At worst, if all elements are unique, `cnt` would be the same size as `deliciousness`, so the space used by `cnt` is `O(n)` where `n` is the length of `deliciousness`.

```
S(n) = O(n)
```

There is a negligible additional space used for the loop indices, calculations, and single-item `b`, which does not depend on the size of `deliciousness` and thus does not affect the overall space complexity.