<https://leetcode.com/problems/count-good-meals/description/>

A **good meal** is a meal that contains **exactly two different food items** with a sum of deliciousness equal to a power of two.

You can pick **any** two different foods to make a good meal.

Given an array of integers deliciousness where deliciousness[i] is the deliciousness of the i​​​​​​th​​​​​​​​ item of food, return **the number of different good meals you can make from this list modulo** 109 + 7.

Note that items with different indices are considered different even if they have the same deliciousness value.

**Example 1:**

**Input:** deliciousness = [1,3,5,7,9]

**Output:** 4

**Explanation:** The good meals are (1,3), (1,7), (3,5) and, (7,9).

Their respective sums are 4, 8, 8, and 16, all of which are powers of 2.

**Example 2:**

**Input:** deliciousness = [1,1,1,3,3,3,7]

**Output:** 15

**Explanation:** The good meals are (1,1) with 3 ways, (1,3) with 9 ways, and (1,7) with 3 ways.

**Constraints:**

1 <= deliciousness.length <= 10^5

0 <= deliciousness[i] <= 2^20

**Attempt 1: 2024-01-12**

**Solution 1: Hash Table + Bit Manipulation (180min)**

**Wrong Solution (71/72)**

**Failed on input is an array contains one hundred thousand 32**

Input: deliciousness = [32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,

32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,

32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,

32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,

32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,

32,32,32,32,32,32,32,32,32,32,32,...32]

The deliciousness array contains 100000's 32

Output: 499949968

Expect: 999949972

class Solution {

    public int countPairs(int[] deliciousness) {

        int MOD = (int)1e9 + 7;

        Map<Integer, Integer> freq = new HashMap<>();

        for(int num : deliciousness) {

            freq.put(num, freq.getOrDefault(num, 0) + 1);

        }

        long count = 0;

        for(int key : freq.keySet()) {

            for(int i = 0; i < 22; i++) {

                int sum = 1 << i;

                //for(int key : freq.keySet()) {

                if(freq.containsKey(sum - key)) {

                    if(sum - key == key) {

                        count += (long)freq.get(key) \* (freq.get(key) - 1) % MOD;

                    } else {

                        count += (long)freq.get(key) \* freq.get(sum - key) % MOD;

                    }

                }

            }

        }

        count >>= 1;

        return (int)count % MOD;

    }

}

**A correct version from chatGPT**

The issue with your code above might be related to the bit manipulation and handling powers of two. In particular, the condition **if (freq.containsKey(sum - key))** might be problematic for certain inputs.

I suggest modifying the code to handle the case where **sum - key** is a power of two, considering the constraints provided in the problem. Here's an updated version:

import java.util.HashMap;

import java.util.Map;

class Solution {

public int countPairs(int[] deliciousness) {

int MOD = (int) 1e9 + 7;

Map<Integer, Integer> freq = new HashMap<>();

// Find the maximum deliciousness value

int maxDeliciousness = 0;

for (int num : deliciousness) {

maxDeliciousness = Math.max(maxDeliciousness, num);

freq.put(num, freq.getOrDefault(num, 0) + 1);

}

long count = 0;

for (int key : freq.keySet()) {

for (int i = 1; i <= maxDeliciousness \* 2; i <<= 1) {

int complement = i - key;

if (complement >= key && freq.containsKey(complement)) {

if (complement == key) {

count = (count + (long) freq.get(key) \* (freq.get(key) - 1) / 2) % MOD;

} else {

count = (count + (long) freq.get(key) \* freq.get(complement)) % MOD;

}

}

}

}

return (int) count;

}

}

This version ensures that the condition **if (freq.containsKey(sum - key))** considers all powers of two within the given constraints. Additionally, it uses **(freq.get(key) \* (freq.get(key) - 1) / 2)** to calculate combinations for pairs with the same deliciousness.

**Correct Solution**

**Copy from** <https://algo.monster/liteproblems/1711>**, but seems not much different than above wrong solution, the most suspect part is**

**below section, relate to % MOD operation, but hard to debug, even switch the wrong solution inner and outer for loop order to make it same as correct solution, still failed on same test case**

// The MOD calculation suspect as most potential error out part

if(sum - key == key) {

    count += (long)freq.get(key) \* (freq.get(key) - 1) % MOD;

} else {

    count += (long)freq.get(key) \* freq.get(sum - key) % MOD;

}

**Full code**

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);

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/1711>

**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:

Initialize a Counter for the array deliciousness to keep track of the number of occurrences of each value of deliciousness.

Initialize a variable ans to keep track of the total number of good meals.

Loop through all the powers of two up to 2^21. This covers the range of possible sums of the two deliciousness values.

For each deliciousness value a found in the hash map we created, calculate b = 2^i - a.

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.

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.

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.

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:

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

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

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.

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.

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).

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

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.

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.

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].

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}.

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

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].

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).

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

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).

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.

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.

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.

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

**Java Solution**

class Solution {

    public int countPairs(int[] deliciousness) {

        int MOD = (int)1e9 + 7;

        Map<Integer, Integer> freq = new HashMap<>();

        for(int num : deliciousness) {

            freq.put(num, freq.getOrDefault(num, 0) + 1);

        }

        long count = 0;

        for(int key : freq.keySet()) {

            for(int i = 0; i < 22; i++) {

                int sum = 1 << i;

                //for(int key : freq.keySet()) {

                if(freq.containsKey(sum - key)) {

                    if(sum - key == key) {

                        count += (long)freq.get(key) \* (freq.get(key) - 1) % MOD;

                    } else {

                        count += (long)freq.get(key) \* freq.get(sum - key) % MOD;

                    }

                }

            }

        }

        count >>= 1;

        return (int)count % 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.