1814. Count Nice Pairs in an Array

Math

Problem Description You are given an array called nums which is filled with non-negative integers. The challenge is to find all pairs of indices (i, j)

Hash Table

Medium <u>Array</u>

equal to the number at position j plus the reversal of the number at position i.

that meet a certain "nice" criterion. This criterion is defined by two conditions:

• The first condition is that the indices i and j must be different and i must be less than j.

Counting

- Now, because simply reversing a number isn't mathematically challenging, the real complexity of the problem lies in finding all
- such pairs efficiently. Since the number of nice pairs can be very large, you need to return the count modulo 10^9 + 7, which is a common technique in programming contests to avoid dealing with extraordinarily large numbers.

• The second condition is that when you take the number at position i and add it to the reversal of the number at position j, this sum must be

Intuition Let's look at the condition provided in the problem - nums[i] + rev(nums[j]) == nums[j] + rev(nums[i]). If we play around

with this equation a bit, we can rephrase it into nums[i] - rev(nums[i]) == nums[j] - rev(nums[j]). This observation is crucial

The intuition behind the problem is to count how many numbers have the same value after performing the operation number -

because it allows us to switch from searching pairs to counting the frequency of unique values of nums[i] - rev(nums[i]).

reversed number. If a certain value occurs k times, any two unique indices with this value will form a nice pair. The number of unique pairs that can be formed from k numbers is given by the formula k * (k - 1) / 2. We use a hash table (python's Counter class) to store the occurrence of each nums[i] - rev(nums[i]) value. Then, we calculate

the sum of the combination counts for each unique nums[i] - rev(nums[i]) value. The Combination Formula is used here to find the number of ways you can select pairs from a group of items. Finally, remember to apply modulo $10^9 + 7$ to our result to get the final answer.

Solution Approach

The solution uses a clever transformation of the check for a nice pair of indices. Instead of directly checking whether nums[i] +

rev(nums[j]) == nums[j] + rev(nums[i]) for each pair, which would be time-consuming, it capitalizes on the insight that if two

nums[i] have the same value after subtracting their reverse, rev(nums[i]), they can form a nice pair with any nums[j] that

shows the same characteristic. The following steps outline the implementation:

items from a set of v items without considering the order.

def countNicePairs(self, nums: List[int]) -> int:

v = v * 10 + x % 10

cnt = Counter(x - rev(x) for x in nums)

and to prevent overflow issues.

Define a rev function which, given an integer x, reverses its digits. This is accomplished by initializing y to zero, and then repeatedly taking the last digit of x by x % 10, adding it to y, and then removing the last digit from x using integer division

by 10.

using the Counter class from the collections module. Once the hash table is filled, iterate over the values in the hash table. For each value v, which represents the number of occurrences of a particular transformed value, calculate the number of nice pairs that can be formed with it using the combination formula v * (v - 1) / 2. This formula comes from combinatorics and gives the number of ways to choose 2

Iterate over all elements in nums and compute the transformed value nums[i] - rev(nums[i]) for each element. We use a

hash table to map each unique transformed value to the number of times it occurs in nums. In Python, this is efficiently done

Sum these counts for each unique transformed value to get the total number of nice pairs. Because the count might be very

large, the problem requires us to modulo the result by 10^9 + 7 to keep the result within the range of a 32-bit signed integer

By transforming the problem and using a hash table to track frequencies of the transformed values, we turn an O(n^2) brute force solution into an O(n) solution, which is much more efficient and suitable for larger input sizes. The code that accomplishes this: class Solution:

return sum(v * (v - 1) // 2 for v in cnt.values()) % modIn the provided Python code, rev is the function that reverses an integer, and Counter(x - rev(x) for x in nums) creates the

hash table mapping each nums[i] - rev(nums[i]) to its frequency. The final summation and modulo operation provide the count

Example Walkthrough

of nice pairs as required.

def rev(x):

V = 0

while x:

return y

mod = 10**9 + 7

x //= 10

```
Let's explain the solution using a small example. Suppose we have the following array:
nums = [42, 13, 20, 13]
 We want to find the count of all "nice" pairs, which means for any two different indices (i, j) with i < j, the condition nums[i]
 + rev(nums[j]) == nums[j] + rev(nums[i]) holds true. Following the steps defined in the solution:
    Define the reverse function: This function reverses the digits of a given number. For example, rev(42) returns 24 and
     rev(13) returns 31.
    Compute transformed values and frequency:
```

{ 18: 2, -18: 2 }

1 = 2. There's no need for the modulo operation in this small example as the result is already small enough.

Sum the counts and apply modulo: We add up the counts from the previous step to get the total count of nice pairs. So, 1 +

At this point, we notice that the transformed value 18 occurs twice and also -18 occurs twice.

Solution Implementation

def countNicePairs(self, nums: List[int]) -> int:

def reverse_number(x: int) -> int:

which simplifies to n * (n - 1) / 2

Python

Java

class Solution {

class Solution:

from collections import Counter

rev = 0

while x > 0:

return rev

mod = 10**9 + 7

return nice_pairs_count

for (int number : nums) {

return (int) answer;

int reversed = 0;

while (number > 0) {

number /= 10;

return reversed;

int reverseNumber(int num) {

int reversedNum = 0:

while (num > 0) {

return reversedNum;

C++

public:

#include <vector>

class Solution {

using namespace std;

#include <unordered map>

// Return the reversed integer

private int reverse(int number) {

// Helper function to reverse a given integer

// Loop to reverse the digits of the number

reversed = reversed * 10 + number % 10;

// Function to calculate the reverse of a given number

num /= 10; // Remove the last digit from num

// Create a map to count occurrences of differences

int difference = num - reverseNumber(num);

// Iterate through the map to calculate the pairs

// Increase the count of the current difference

// Calculate the difference between the number and its reverse

const int mod = 1e9 + 7; // Use modulo to avoid integer overflow

reversedNum = reversedNum * 10 + num % 10: // Append the last digit to the reversedNum

// Iterate over the digits of the number

// Function to count nice pairs in an array

unordered_map<int, int> differenceCount;

differenceCount[difference]++;

int countNicePairs(vector<int>& nums) {

// Iterate over the given numbers

for (int& num : nums) {

long long answer = 0:

// Remove the last digit from number

// Append the last digit of number to reversed

// Set initial reversed number to 0

public int countNicePairs(int[] nums) {

Hence, the count of nice pairs in this example is 2.

 \circ For -18, similarly, we calculate 2 * (2 - 1) / 2 = 1

 \circ For nums[0] = 42: 42 - rev(42) = 42 - 24 = 18

 \circ For nums[1] = 13: 13 - rev(13) = 13 - 31 = -18

 \circ For nums[2] = 20: 20 - rev(20) = 20 - 02 = 18

 \circ For nums[3] = 13 (again): 13 - rev(13) = 13 - 31 = -18

Use a hash table to map transformed values to frequencies:

Calculate the number of nice pairs using combination formula:

∘ For 18, the number of nice pairs is calculated as 2 * (2 - 1) / 2 = 1

```
# Create a counter to count the occurrences of differences between
# each number and its reversed version
difference_counter = Counter(x - reverse_number(x) for x in nums)
# Define the modulus for the answer to prevent overflow
```

Calculate the number of nice pairs using the formula:

Return the total count of nice pairs modulo 10^9 + 7

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

// Iterate through the array of numbers

// Create a HashMap to store the counts of each difference value

// Calculate the difference between the number and its reverse

v * (v - 1) // 2 for each count 'v' in the counter

x //= 10 # Remove the last digit from x

Define a helper function to reverse the digits of a number

rev = rev * 10 + x % 10 # Append the last digit of x to rev

The formula is derived from the combination formula C(n, 2) = n! / (2! * (n - 2)!)

nice_pairs_count = $sum(v * (v - 1) // 2 for v in difference_counter.values()) % mod$

```
int difference = number - reverse(number);
    // Update the count of the difference in the HashMap
    countMap.merge(difference, 1, Integer::sum);
// Define the modulo value to ensure the result fits within integer range
final int mod = (int) 1e9 + 7;
// Initialize the answer as a long to handle potential overflows
long answer = 0;
// Iterate through the values in the countMap
for (int count : countMap.values()) {
    // Calculate the number of nice pairs and update the answer
    answer = (answer + (long) count * (count - 1) / 2) % mod;
// Cast the answer back to an integer before returning
```

```
for (auto& kvp : differenceCount) {
            int value = kvp.second: // Extract the number of occurrences
            // Update the answer using the combination formula C(v, 2) = v! / (2! * (v - 2)!)
            // Simplifies to v * (v - 1) / 2
            answer = (answer + 1LL * value * (value - 1) / 2) % mod;
        return answer; // Return the final count of nice pairs
};
TypeScript
function countNicePairs(nums: number[]): number {
    // Helper function to reverse the digits of a number
    const reverseNumber = (num: number): number => {
        let rev = 0;
        while (num) {
            rev = rev * 10 + (num % 10);
            num = Math.floor(num / 10);
        return rev;
    };
    // Define the modulo constant to prevent overflow
    const MOD = 10 ** 9 + 7;
    // Map to keep count of each difference occurrence
    const countMap = new Map<number, number>();
    // Initialize the answer to be returned
    let answer = 0;
    // Loop through the array of numbers
    for (const num of nums) {
        // Calculate the difference of the original and reversed number
        const difference = num - reverseNumber(num);
        // Update the answer with the current count of the difference
        // If the difference is not vet encountered, it treats the count as 0
        answer = (answer + (countMap.get(difference) ?? 0)) % MOD;
        // Update the count of the current difference in the map
        countMap.set(difference, (countMap.get(difference) ?? 0) + 1);
    // Return the final answer
    return answer;
from collections import Counter
class Solution:
    def countNicePairs(self, nums: List[int]) -> int:
        # Define a helper function to reverse the digits of a number
        def reverse_number(x: int) -> int:
            rev = 0
            while x > 0:
                rev = rev * 10 + x % 10 # Append the last digit of x to rev
```

Time Complexity

return nice_pairs_count

Time and Space Complexity

return rev

mod = 10**9 + 7

2. Summing up all pairs for each unique difference (value in the counter object). The first operation depends on the number of digits for each integer in the nums list. Reversing an integer x is proportional to the

1. Calculating the reverse of each number and constructing the counter object.

The time complexity of the given code consists of two main operations:

x //= 10 # Remove the last digit from x

Define the modulus for the answer to prevent overflow

Calculate the number of nice pairs using the formula:

Return the total count of nice pairs modulo 10^9 + 7

v * (v - 1) // 2 for each count 'v' in the counter

each number and its reversed version

which simplifies to n * (n - 1) / 2

Create a counter to count the occurrences of differences between

difference_counter = Counter(x - reverse_number(x) for x in nums)

The formula is derived from the combination formula C(n, 2) = n! / (2! * (n - 2)!)

nice_pairs_count = $sum(v * (v - 1) // 2 for v in difference_counter.values()) % mod$

number of digits in x, which is 0(log M) where M is the value of the integer. Since we perform this operation for each element in the list, the time complexity of this part is 0(n * log M).

The second operation involves iterating over each value in the counter object and calculating the number of nice pairs using the formula v * (v - 1) // 2. As there are at most n unique differences (in the case that no two numbers have the same difference), iterating over each value in the counter will be O(n) in the worst case.

The space complexity is determined by the additional space used by the algorithm beyond the input size. In this case, it is the

Space Complexity

Hence, the overall time complexity is dominated by the first part, which is 0(n * log M).

space used to store the counter object. The counter object could have as many as n entries (in the worst case where each number's difference after reversals is unique). Therefore, the space complexity of the code is O(n).