<https://leetcode.com/problems/count-nice-pairs-in-an-array/description/>

You are given an array nums that consists of non-negative integers. Let us define rev(x) as the reverse of the non-negative integer x. For example, rev(123) = 321, and rev(120) = 21. A pair of indices (i, j) is nice if it satisfies all of the following conditions:

0 <= i < j < nums.length

nums[i] + rev(nums[j]) == nums[j] + rev(nums[i])

Return the number of nice pairs of indices. Since that number can be too large, return it modulo 10^9 + 7.

**Example 1:**

**Input:** nums = [42,11,1,97]

**Output:** 2

**Explanation:** The two pairs are:

- (0,3) : 42 + rev(97) = 42 + 79 = 121, 97 + rev(42) = 97 + 24 = 121.

- (1,2) : 11 + rev(1) = 11 + 1 = 12, 1 + rev(11) = 1 + 11 = 12.

**Example 2:**

**Input:** nums = [13,10,35,24,76]

**Output:** 4

**Constraints:**

1 <= nums.length <= 10^5

0 <= nums[i] <= 10^9

**Attempt 1: 2024-01-19**

**Solution 1: Harsh Table (10 min)**

class Solution {

    public int countNicePairs(int[] nums) {

        int MOD = (int)1e9 + 7;

        int result = 0;

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

        // nums[i] + rev(nums[j]) == nums[j] + rev(nums[i])

        // -> nums[i] - rev(nums[i]) == nums[j] - rev(nums[j])

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

            nums[i] -= reverse(nums[i]);

            result = (result + map.getOrDefault(nums[i], 0)) % MOD;

            map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

        }

        return result;

    }

    private int reverse(int num) {

        int result = 0;

        while(num > 0) {

            result = result \* 10 + num % 10;

            num /= 10;

        }

        return result;

    }

}

Time Complexity: O(N\*logM), where M is the value of the integer

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/count-nice-pairs-in-an-array/solutions/1140639/java-c-python-straight-forward/>

**Explanation**

A[i] + rev(A[j]) == A[j] + rev(A[i])

A[i] - rev(A[i]) == A[j] - rev(A[j])

B[i] = A[i] - rev(A[i])

Then it becomes an easy question that,

how many pairs in B with B[i] == B[j]

**Complexity**

Time O(nloga)

Space O(n)

**Java**

public int countNicePairs(int[] A) {

int res = 0, mod = (int)1e9 + 7;

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

for (int a : A) {

int b = rev(a), v = count.getOrDefault(a - b, 0);

count.put(a - b, v + 1);

res = (res + v) % mod;

}

return res;

}

public int rev(int a) {

int b = 0;

while (a > 0) {

b = b \* 10 + (a % 10);

a /= 10;

}

return b;

}

**Refer to**

<https://algo.monster/liteproblems/1814>

**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) 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.

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

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.

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

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

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.

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

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 and to prevent overflow issues.

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:

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

        def rev(x):

            y = 0

            while x:

                y = y \* 10 + x % 10

                x //= 10

            return y

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

        mod = 10\*\*9 + 7

        return sum(v \* (v - 1) // 2 for v in cnt.values()) % mod

In 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 of nice pairs as required.

**Example Walkthrough**

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:

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

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

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

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

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

Use a hash table to map transformed values to frequencies:

{ 18: 2, -18: 2 }

Calculate the number of nice pairs using combination formula:

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

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

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

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

class Solution {

public int countNicePairs(int[] nums) {

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

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

// Iterate through the array of numbers

for (int number : nums) {

// Calculate the difference between the number and its reverse

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

return (int) answer;

}

// Helper function to reverse a given integer

private int reverse(int number) {

// Set initial reversed number to 0

int reversed = 0;

// Loop to reverse the digits of the number

while (number > 0) {

// Append the last digit of number to reversed

reversed = reversed \* 10 + number % 10;

// Remove the last digit from number

number /= 10;

}

// Return the reversed integer

return reversed;

}

}

**Time and Space Complexity**

**Time Complexity**

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

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

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 number of digits in x, which is O(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 O(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.

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

**Space Complexity**

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