<https://leetcode.com/problems/number-of-pairs-of-strings-with-concatenation-equal-to-target/description/>

Given an array of digit strings nums and a digit string target, return the number of pairs of indices (i, j) (where i != j) such that the concatenation of nums[i] + nums[j] equals target.

**Example 1:**

**Input:** nums = ["777","7","77","77"], target = "7777"

**Output:** 4

**Explanation:** Valid pairs are:

- (0, 1): "777" + "7"

- (1, 0): "7" + "777"

- (2, 3): "77" + "77"

- (3, 2): "77" + "77"

**Example 2:**

**Input:** nums = ["123","4","12","34"], target = "1234"

**Output:** 2

**Explanation:** Valid pairs are:

- (0, 1): "123" + "4"

- (2, 3): "12" + "34"

**Example 3:**

**Input:** nums = ["1","1","1"], target = "11"

**Output:** 6

**Explanation:** Valid pairs are:

- (0, 1): "1" + "1"

- (1, 0): "1" + "1"

- (0, 2): "1" + "1"

- (2, 0): "1" + "1"

- (1, 2): "1" + "1"

- (2, 1): "1" + "1"

**Constraints:**

2 <= nums.length <= 100

1 <= nums[i].length <= 100

2 <= target.length <= 100

nums[i] and target consist of digits.

nums[i] and target do not have leading zeros.

**Attempt 1: 2024-01-24**

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

class Solution {

    public int numOfPairs(String[] nums, String target) {

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

        for(String num : nums) {

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

        }

        int result = 0;

        for(int i = 1; i <= target.length(); i++) {

            String a = target.substring(0, i);

            String b = target.substring(i);

            int count\_a = map.getOrDefault(a, 0);

            int count\_b = map.getOrDefault(b, 0);

            if(!a.equals(b)) {

                result += count\_a \* count\_b;

            } else {

                // If "a" and "b" are the same, each instance of "a" could

                // pair with all other instances of "b", but not with itself

                // e.g nums = ["777","7","77","77"], target = "7777"

                // - (2, 3): a = "77" + b = "77"

                // - (3, 2): b = "77" + a = "77"

                // since switch order recognize as 2 pair, so we have

                // count\_a \* (count\_a - 1) / 2 \* 2 = count\_a \* (count\_a - 1)

                result += count\_a \* (count\_a - 1);

            }

        }

        return result;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/number-of-pairs-of-strings-with-concatenation-equal-to-target/solutions/1503157/c-simple-and-easy-solution-with-detailed-explanation/>

**Idea:**

First, we store in a map the frequencies of the strings, so that we can find easily which strings we have and how many.

Now, we iterate through the freq map.

For every string:

We check if it's a prefix of our target.

If yes, first case is that the target is exactly twice the prefix. If so, we add frq\*(frq-1) to res.

Otherwise we look in the map if we have the suffix, if so - we add the product of their frequencies to res.

**Time Complexity:** O(n)

**Space Complexity:** O(n)

**Refer to**

<https://algo.monster/liteproblems/2023>

**Problem Description**

The given problem involves finding the total number of unique pairs of indices (i, j) from an array of digit strings nums such that when nums[i] and nums[j] are concatenated (joined together in the order nums[i] + nums[j]), the result equals the given digit string target. The constraint is that i and j must be different, i.e., you cannot use the same index twice in a pair. The task is to return the count of such pairs.

**Intuition**

To solve this problem, the insight is to use the properties of strings and hash tables. We know that if the concatenation of two strings, a and b, produces the target, then a must be a prefix of target, and b must be a suffix. Furthermore, a and b together must cover the entire target string without overlap, except when a and b are equal.

One approach would be to iterate over all possible pairs of strings in nums and check if their concatenation equals target. However, this approach would have a time complexity of O(n^2 \* m), where n is the number of strings in nums and m is the length of target. We can optimize this by using a hash table (a Counter in Python) to store counts of all the strings in nums. This allows us to efficiently look up how many times a specific string occurs without iterating through the array again.

The solution iterates through each possible split point in target, effectively dividing the target into a prefix a and a suffix b. For each such pair (a, b), the product of the number of occurrences of a and b in nums is added to the answer. If a and b are the same, we adjust the count since we cannot use the same index twice; this is done by subtracting one from the count of a before multiplying.

This approach results in a time complexity of O(m^2 + n), where m is the length of target and n is the number of strings in nums, since we are iterating through the target string and using a hash table for constant time lookups.

**Solution Approach**

The solution approach can be summarized in the following steps:

Initialize a Counter: A Counter from Python's collections module is initiated to store the occurrences of each string in nums. This data structure allows us to query in constant time whether a string is present in nums and, if so, how many times.

Iterate through Target Splits: The approach then involves iterating through each possible index in the target string to split it into two parts, a and b. The indices chosen range from 1 to the length of the target string minus one. This ensures that a and b are non-empty and cover the whole target string when concatenated.

Calculate Pair Combinations: For a given split (a, b):

If a is not equal to b, the number of valid pairs is the product of the occurrences of a and b in nums, since they can be freely paired.

**If a is equal to b, one instance of a is subtracted from the total count before multiplication to avoid pairing a number with itself as i cannot be equal to j.**

The final answer, stored in ans, accumulates the count of valid pairs through all iterations.

class Solution:

    def numOfPairs(self, nums: List[str], target: str) -> int:

        cnt = Counter(nums)  # Step 1: Initialize a Counter

        ans = 0  # Initialize the count of pairs to zero

        for i in range(1, len(target)):  # Step 2: Iterate through Target Splits

            a, b = target[:i], target[i:]  # Split the `target` into `a` and `b`

            if a != b:  # Step 3: Calculate Pair Combinations

                ans += cnt[a] \* cnt[b]  # Multiply the counts if `a` and `b` are not equal

            else:

                ans += cnt[a] \* (cnt[a] - 1)  # Adjust if `a` and `b` are the same

        return ans # Return the final count of pairs

The current implementation is efficient because it avoids the brute-force checking of all pairs in nums, instead taking advantage of the hashing capability of the Counter to look up counts quickly.

**Example Walkthrough**

Let's consider a small example where nums = ["1","11","111","011"] and target = "1111". Here's how the solution approach would be applied to find the count of pairs whose concatenation equals target.

Initialize Counter: The Counter will count occurrences of all strings in nums.

Counter({'1': 1, '11': 1, '111': 1, '011': 1})

This allows for constant-time queries of occurrences.

Iterate through Target Splits: The target "1111" has several possible splits: "1|111", "11|11", and "111|1".

For the split "1|111":

a is "1", and b is "111".

The count of "1" in nums is 1, and the count of "111" is also 1.

Since a != b, we multiply their counts: 1 \* 1 = 1.

**For the split "11|11":**

**a is "11", and b is also "11".**

**The count of "11" in nums is 1.**

**But a == b, so we use the adjusted count: 1 \* (1 - 1) = 0.**

For the split "111|1":

a is "111", and b is "1".

The count of "111" in nums is 1, and the count of "1" is 1.

Since a != b, we multiply their counts: 1 \* 1 = 1.

Calculate Pair Combinations: Adding the results of all splits, we get 1 + 0 + 1 = 2.

So, there are 2 unique pairs of indices in nums that can be concatenated to form the target "1111".

**Java Solution**

class Solution {

    public int numOfPairs(String[] nums, String target) {

        // Create a map to store the frequency of each number (string) in the nums array

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

        for (String num : nums) {

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

        }

        // Initialize a variable to keep track of the number of valid pairs

        int answer = 0;

        // Loop through the target string, excluding its first and last characters

        for (int i = 1; i < target.length(); ++i) {

            // Split the target into two substrings ("a" and "b") at the current position i

            String a = target.substring(0, i);

            String b = target.substring(i);

            // Retrieve the frequency of each substring from the map

            int countA = countMap.getOrDefault(a, 0);

            int countB = countMap.getOrDefault(b, 0);

            // If "a" and "b" are different, multiply their counts since they can form distinct pairs

            if (!a.equals(b)) {

                answer += countA \* countB;

            } else {

                // If "a" and "b" are the same, each instance of "a" could pair with all other instances of "b", but not with itself

                answer += countA \* (countB - 1);

            }

        }

        // Return the total number of valid pairs found

        return answer;

    }

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code is composed of two parts: the creation of the counter and the loop that goes through the possible splits of the target string.

Constructing cnt as a Counter object takes O(n) time, where n is the number of elements in nums, because it needs to iterate over all elements once to count the frequencies.

For the loop that checks all the possible splits, the number of iterations is proportional to the length of the target string because it iterates through every possible split index. This is O(m), where m is the length of the target string.

The operations within the loop take constant time since dictionary access and multiplication are O(1) operations.

Therefore, the overall time complexity is O(n + m).

**Space Complexity**

The space complexity is primarily influenced by the storage requirements of the Counter object.

The Counter object cnt stores each unique element from nums. In the worst case, all elements are unique, so the space required is O(n), where n is the number of elements in nums.

Thus, the space complexity of the code is O(n).