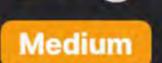
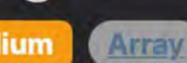
2023. Number of Pairs of Strings With Concatenation Equal to Target





Problem Description

Medium Array String Leetcode Link

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

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,

To solve this problem, the insight is to use the properties of strings and hash tables. We know that if the concatenation of two

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(m² + n), where m is the length of target and n is the number of strings in nums,

Solution Approach

since we are iterating through the target string and using a hash table for constant time lookups.

1. Initialize a Counter: A Counter from Python's collections module is initiated to store the occurrences of each string in nums. This

paired.

data structure allows us to query in constant time whether a string is present in nums and, if so, how many times.

The solution approach can be summarized in the following steps:

- 2. 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. 3. Calculate Pair Combinations: For a given split (a, b): o 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
- If a is equal to b, one instance of a is subtracted from the total count before multiplication to avoid pairing a number with

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

itself as i cannot be equal to j.

- The final answer, stored in ans, accumulates the count of valid pairs through all iterations.
- 1 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

else: ans += cnt[a] * (cnt[a] - 1) # Adjust if `a` and `b` are the same 10

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

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'

```
return ans # Return the final count of pairs
11
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
```

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

would be applied to find the count of pairs whose concatenation equals target. 1. Initialize Counter: The Counter will count occurrences of all strings in nums.

Let's consider a small example where nums = ["1","11","11","011"] and target = "1111". Here's how the solution approach

This allows for constant-time queries of occurrences.

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.

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

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

• For the split "111 1":

For the split "11 11":

For the split "1 111":

■ The count of "11" in nums is 1.

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

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

3. 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".

num_counter = Counter(nums)

if prefix != suffix:

Return the total number of pairs found

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

answer += countA * (countB - 1);

// Return the total number of valid pairs found

answer += countA * countB;

if (!a.equals(b)) {

} else {

pair_count = 0

from collections import Counter

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

Create a counter to hold the frequency of each number in nums

Initialize a variable to count the number of valid pairs

10 11 # Iterate through the target string and split it at different points for i in range(1, len(target)): 12 prefix, suffix = target[:i], target[i:] # Split target into prefix and suffix 13

If prefix and suffix are different, multiply their counts directly

```
pair_count += num_counter[prefix] * num_counter[suffix]
17
               else:
18
                   # If prefix and suffix are the same, we must avoid counting the pair (num, num) twice
19
                    pair_count += num_counter[prefix] * (num_counter[prefix] - 1)
20
```

Python Solution

class Solution:

9

14

15

16

21

22

21

22

23

24

26

27

28

29

30

31

32

```
23
           return pair_count
24
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
10
11
           int answer = 0;
12
13
           // Loop through the target string, excluding its first and last characters
           for (int i = 1; i < target.length(); ++i) {</pre>
14
               // Split the target into two substrings ("a" and "b") at the current position i
15
               String a = target.substring(0, i);
16
17
               String b = target.substring(i);
18
               // Retrieve the frequency of each substring from the map
19
               int countA = countMap.getOrDefault(a, 0);
20
```

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

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

33 return answer; 34 35 } 36

```
C++ Solution
1 #include <string>
2 #include <vector>
   #include <unordered_map>
  class Solution {
   public:
       // Function to count the number of pairs of strings in 'nums' that can be concatenated to form the 'target' string.
       int numOfPairs(std::vector<std::string>& nums, std::string target) {
           // Using a hashmap to count the frequency of each string in 'nums'.
9
           std::unordered_map<std::string, int> frequencyMap;
10
           for (auto &num : nums) {
               ++frequencyMap[num]; // Increment frequency count for each string.
12
13
14
15
           int pairCount = 0; // This will store the number of valid pairs found.
16
           // Iterate over all possible splits of 'target' into two non-empty substrings 'leftPart' and 'rightPart'.
17
           for (int i = 1; i < target.size(); ++i) {</pre>
18
               std::string leftPart = target.substr(0, i);
19
20
               std::string rightPart = target.substr(i);
21
22
               int leftCount = frequencyMap[leftPart], rightCount = frequencyMap[rightPart];
23
               // When 'leftPart' and 'rightPart' are different, multiply their frequencies directly.
24
25
               // Otherwise, if they are the same (e.g., 'a' and 'a'), pairs are counted by forming combinations
26
               // of two different indices from the frequency of that string; hence, the (rightCount - 1).
               if (leftPart != rightPart) {
27
28
                   pairCount += leftCount * rightCount;
29
               } else {
                   pairCount += leftCount * (rightCount - 1);
30
31
32
33
34
           return pairCount; // Return the total number of pairs.
35
```

Typescript Solution

type StringFrequencyMap = Record<string, number>;

36 };

37

```
function numOfPairs(nums: string[], target: string): number {
       // Creating a map to keep the frequency of each string in the array.
       const frequencyMap: StringFrequencyMap = nums.reduce((acc: StringFrequencyMap, num: string) => {
           acc[num] = (acc[num] | | 0) + 1;
           return acc;
 9
       }, {});
10
11
       let pairCount = 0; // This will hold the total number of valid pairs found.
12
       // Iterate over all possible non-empty prefixes and suffixes of the target string.
13
       for (let i = 1; i < target.length; i++) {</pre>
14
           const leftPart = target.slice(0, i);
15
           const rightPart = target.slice(i);
16
17
           const leftCount = frequencyMap[leftPart] || 0;
18
           const rightCount = frequencyMap[rightPart] || 0;
19
20
           // If leftPart and rightPart are different, compute the product of their counts.
           // If they are the same, we must choose different elements, hence the product of leftCount and (rightCount - 1).
22
23
           if (leftPart !== rightPart) {
24
               pairCount += leftCount * rightCount;
           } else {
25
26
               pairCount += leftCount * (rightCount - 1);
27
28
29
30
       // Return the computed number of pairs.
31
       return pairCount;
32 }
33
Time and Space Complexity
```

// Function to count the number of pairs of strings in the array that can be concatenated to form the target string.

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

• 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 0(1) operations.
- Therefore, the overall time complexity is O(n + m).

Space Complexity

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

over all elements once to count the frequencies.

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.