2506. Count Pairs Of Similar Strings Hash Table String Easy <u>Array</u>

## **Problem Description**

frequency of those characters. For a clear understanding, let's look at some examples: • "abca" and "caba" are similar because both contain the characters 'a', 'b', and 'c'. • On the other hand, "abacba" and "bcfd" are not similar because they contain different characters.

In this problem, we are provided with an array of strings named words. Our goal is to find the number of similar string pairs in this

array. A pair of strings is considered similar if both strings are comprised of the same characters, regardless of the order or

The task is to return the total count of pairs (i, j) where 0 <= i < j < words.length, and the strings words[i] and words[j]

for that vector by 1, since we have one more occurrence of this bit pattern.

Inside the loop for each word w, we initialize v to 0.

We add this count to ans, which stores the total number of similar pairs.

are similar.

To solve this problem, we can use a bit representation technique for the characters in each string to efficiently check for similarity. Here's how we arrive at the solution:

Intuition

• We will use a bit vector (an integer) to represent which characters are present in each string. For example, if a string contains the character 'A', we will set the 0th bit in the vector. If it contains 'B', we will set the 1st bit, and so on. • We represent each string in words as an integer v by iterating over each character in the string and setting the corresponding bit in v.

- Then, we use a Counter to keep track of how many times we have seen each bit vector representation so far. This is because if we have seen the same bit vector before, the current string is similar to all previous strings with that bit vector, forming similar pairs. • As we process each word, we add the current count of the identical bit vector from the Counter to our answer, then we increment the Counter
- Solution Approach

The implementation of the solution follows a bit manipulation approach to efficiently count similar string pairs. Here is a step-bystep walk-through of the algorithm, with reference to the given solution code: Counter Initialization: We use a Counter from Python's collections module to keep track of the frequencies of the unique

bit vectors representing each string. cnt = Counter()

same bit vector).

following array of words:

cnt = Counter()

this pattern.

would be 3.

**Python** 

Java

words = ["abc", "bca", "dab", "bac", "bad"]

Repeat the process for 'b' and 'c'.

Update the Counter: cnt[v] += 1.

Update the Counter: cnt[v] += 1.

the bit vector was seen before.

Solution Implementation

from collections import Counter

for word in words:

bit\_vector = 0

for char in word:

return similar\_pairs\_count

for (String word : words) {

int bitmaskValue = 0;

// Return the number of similar pairs

function similarPairs(words: string[]): number {

// Iterates over each word in the input array

for (let i = 0;  $i < word.length; ++i) {$ 

def similar pairs(self, words: List[str]) -> int:

# Iterate through each word in the list

# Initialize the number of similar pairs to zero

# Start with a bit vector of 0 for each word

# Iterate through each character in the word

bit\_vector |= 1 << (ord(char) - ord('A'))

similar\_pairs\_count += bit\_pattern\_counter[bit\_vector]

const wordBitmaskCount: Map<number, number> = new Map();

// Converts each character of the word into a bitmask

// it represents a word with a matching set of characters

pairCount += wordBitmaskCount.get(bitmask) || 0;

bitmask |= 1 << (word.charCodeAt(i) - 'a'.charCodeAt(0));</pre>

// Increment the count for this bitmask representation of a word

# Initialize a Counter to keep track of the different bit patterns

# Shift 1 to the left by the position of the character in the alphabet

# Add the current bit vector pattern's existing count to similar\_pairs\_count

# 'A' would correspond to bit 0, 'B' to bit 1, and so on.

// If a bitmask has already been seen, add its count to the answer since

wordBitmaskCount.set(bitmask, (wordBitmaskCount.get(bitmask) || 0) + 1);

return answer;

**TypeScript** 

let pairCount = 0:

return pairCount;

from typing import List

class Solution:

**Time Complexity** 

words, giving O(n).

from collections import Counter

similar\_pairs\_count = 0

for word in words:

bit\_vector = 0

for char in word:

bit\_pattern\_counter = Counter()

for (const word of words) {

let bitmask = 0;

// Iterate over the characters of the word

bitmaskValue |= 1 << (word.charAt(i) - 'a');</pre>

// Increment the count for this bitmask in our map

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

import java.util.HashMap;

import java.util.Map;

class Solution {

# Iterate through each word in the list

# Start with a bit vector of 0 for each word

# Iterate through each character in the word

bit\_pattern\_counter[bit\_vector] += 1

# Return the total count of similar pairs

bit\_vector |= 1 << (ord(char) - ord('A'))

similar\_pairs\_count += bit\_pattern\_counter[bit\_vector]

# Increment the count for this bit pattern in the counter

# Shift 1 to the left by the position of the character in the alphabet

# Add the current bit vector pattern's existing count to similar\_pairs\_count

// Initialize a variable to store the unique combination of letters as a bitmask

// Update the answer with the count of the current bitmask in our map if it exists

// The bitmask represents which letters are present in the word

answer += letterCombinationCount.getOrDefault(bitmaskValue, 0);

// Create the bitmask by 'or'-ing with the bit representation for the current letter

letterCombinationCount.put(bitmaskValue, letterCombinationCount.getOrDefault(bitmaskValue, 0) + 1);

# 'A' would correspond to bit 0, 'B' to bit 1, and so on.

from typing import List

■ After processing "abc", v will be a number with bits 0, 1, and 2 set.

For the second word "bca", we repeat the process:

Continue the same process for "dab", "bac", and "bad".

illustrates the effectiveness of the bit manipulation approach for this problem.

**Processing Words**: We iterate over each word in the words array. With each word, we intend to create a bit vector v that

uniquely represents the set of characters in the word. for w in words:

V = 0for c in w:  $v \mid = 1 << (ord(c) - ord("A"))$ 

Counting Similar Pairs: After getting the bit vector for the current word, we check how many times this bit pattern has occurred before by looking up v in the cnt Counter. ans += cnt[v]

• The value from cnt[v] gives us the number of similar strings encountered so far (since they have the same characters, and hence the

• For each character c in the word, we calculate the bit position based on its ASCII value (using ord(c) - ord("A") which gives a unique

number for each uppercase letter) and set the corresponding bit in the vector  $\mathbf{v}$  using a bitwise OR assignment ( $\mathbf{l} = \mathbf{l}$ ).

**Updating the Counter:** Lastly, we update the Counter by incrementing the count for the current bit vector, because we have one more string that represents this set of characters. cnt[v] += 1

return ans

The data structure used in this solution is a Counter, which is essentially a dictionary specialized for counting hashable objects.

The algorithm leverages bit manipulation to create a compact representation of each string's character set, which allows us to

quickly determine if two strings are similar without having to compare each character. This translates to an efficient solution in

Returning the Result: Once all words have been processed, we return ans as the total number of similar string pairs.

terms of both time and space complexity. **Example Walkthrough** 

Let's consider a small example using the solution approach to illustrate how this algorithm works. Assume we are given the

Counter Initialization: First, we initialize an empty Counter to keep track of the bit vector representations of the strings.

For the first word "abc": ■ Initialize v = 0. Iterate over each character: 'a', 'b', 'c'. For 'a', it corresponds to bit 0, so v |= 1 << (ord('a') − ord('A')).</p>

We want to find the number of similar string pairs in this array using the bit manipulation method.

**Processing Words**: We iterate through each word in words and construct a bit vector v.

■ The resulting v will be the same as for "abc" because it has the same unique characters.

Before updating the Counter, add the current count of v to ans: ans += cnt[v].

we've encountered. For each word that generates the same bit vector v, we keep incrementing the ans. When we process "bac", we will find that it has a similar bit vector to "abc", and hence our ans will be incremented by 1 again.

Counting Similar Pairs: As we move through the array, the Counter helps us to keep track of the number of similar strings

• Finally, when we get to "bad", we need to create a new bit pattern because 'd' introduces a new character. We then start a new counter for

Updating the Counter: After each word, we updated our Counter with the new bit vector or incremented the existing one if

Returning the Result: Once we have processed all words, the ans variable will give us the total number of similar string pairs.

In our example, the similar pairs are: ("abc", "bca"), ("abc", "bac"), ("bca", "bac"). So the final answer returned by our algorithm

By using the bit vectors, we avoided comparing each pair of strings directly, which would have been more time-consuming. This

class Solution: def similar pairs(self, words: List[str]) -> int: # Initialize the number of similar pairs to zero similar\_pairs\_count = 0 # Initialize a Counter to keep track of the different bit patterns bit\_pattern\_counter = Counter()

// Method to find the number of similar pairs in an array of words public int similarPairs(String[] words) { // Initialize the answer to zero int answer = 0; // A map to keep track of the count of the unique letter combinations for words Map<Integer, Integer> letterCombinationCount = new HashMap<>(); // Iterate over each word in the array

C++ #include <vector> #include <string> #include <unordered\_map> class Solution { public: // Function to count the number of similar pairs in the given vector of strings int similarPairs(std::vector<std::string>& words) { int similarPairsCount = 0; // Variable to store the count of similar pairs std::unordered\_map<int, int> bitmaskFrequencyMap; // Map to store frequency of each bitmask for (auto& word : words) { // Iterate through each word in the vector int bitmask = 0; // Initialize bitmask for this word // Create a bitmask for the word by setting bits corresponding to characters in the word for (auto& character : word) { bitmask |= 1 << (character - 'a'); // Set the bit for this particular character // Increment the count of similar pairs by the frequency of the current bitmask similarPairsCount += bitmaskFrequencyMap[bitmask]; // Increment the frequency of the current bitmask bitmaskFrequencyMap[bitmask]++; return similarPairsCount; // Return the final count of similar pairs **}**;

// Each bit in the integer represents the presence of a character ('a' -> 0th bit, 'b' -> 1st bit, ...)

# Increment the count for this bit pattern in the counter bit\_pattern\_counter[bit\_vector] += 1 # Return the total count of similar pairs return similar\_pairs\_count Time and Space Complexity The provided code snippet is designed to count pairs of words that are similar in the sense that they share the same character set. The Counter class from Python's collections module is used to maintain a count of how many times each unique

## • Inside the loop, there is an inner loop that iterates over each character c in the word w. The maximum length of a word can be denoted as k. The bitwise OR and shift operations inside the inner loop are constant time operations (O(1)). Therefore, the time complexity for processing each word is O(k), and since there are n words, the overall time complexity of the

The time complexity of the code can be analyzed as follows:

• The outer loop runs n times where n is the number of words in the words list.

• The variable v is an integer that represents a set of characters. The space for this is O(1).

representation of word characters has been seen.

- algorithm is O(nk).
- **Space Complexity** Analyzing the space complexity:

• A Counter object is used to count the instances of each unique character set which in the worst case could have as many entries as there are

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