2506. Count Pairs Of Similar Strings

Easy Array Hash Table String

Leetcode Link

Problem Description

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 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.

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]` are similar.

Intuition

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:

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

Doing this allows us to efficiently calculate the number of similar pairs without having to directly compare every pair of strings, resulting in a more time-efficient algorithm.

Solution Approach

The implementation of the solution follows a bit manipulation approach to efficiently count similar string pairs. Here is a step-by-step walk-through of the algorithm, with reference to the given solution code:

1. **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()
```

2. **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 = 0
    for c in w:
        v |= 1 << (ord(c) - ord("A"))
```

   - Inside the loop for each word `w`, we initialize `v` to 0.
   - 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 `v` using a bitwise OR assignment (`|=`).
3. **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 same bit vector).
   - We add this count to `ans`, which stores the total number of similar pairs.
4. **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
```

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

```
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 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 following array of words:

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

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

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

```
cnt = Counter()
```

2. **Processing Words**: We iterate through each word in `words` and construct a bit vector `v`.
   - 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'))`.
     - Repeat the process for `'b'` and `'c'`.
     - After processing "abc", `v` will be a number with bits 0, 1, and 2 set.
     - Update the Counter: `cnt[v] += 1`.
   - For the second word "bca", we repeat the process:
     - 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]`.
     - Update the Counter: `cnt[v] += 1`.
   - Continue the same process for "dab", "bac", and "bad".
3. **Counting Similar Pairs**: As we move through the array, the `Counter` helps us to keep track of the number of similar strings 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.
   - 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 this pattern.
4. **Updating the Counter**: After each word, we updated our `Counter` with the new bit vector or incremented the existing one if the bit vector was seen before.
5. **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 would be `3`.

By using the bit vectors, we avoided comparing each pair of strings directly, which would have been more time-consuming. This illustrates the effectiveness of the bit manipulation approach for this problem.

Python Solution

```
from typing import List
from collections import Counter

class Solution:
    def similarPairs(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()

        # Iterate through each word in the list
        for word in words:
            # Start with a bit vector of 0 for each word
            bit_vector = 0

            # Iterate through each character in the word
            for char in word:
                # Shift 1 to the left by the position of the character in the alphabet
                # 'A' would correspond to bit 0, 'B' to bit 1, and so on.
                bit_vector |= 1 << (ord(char) - ord('A'))

            # Add the current bit vector pattern's existing count to similar_pairs_count
            similar_pairs_count += bit_pattern_counter[bit_vector]

            # 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
```

Java Solution

```
import java.util.HashMap;
import java.util.Map;

class Solution {

    // 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
        for (String word : words) {
            // Initialize a variable to store the unique combination of letters as a bitmask
            int bitmaskValue = 0;

            // Iterate over the characters of the word
            for (int i = 0; i < word.length(); ++i) {
                // Create the bitmask by 'or'-ing with the bit representation for the current letter
                // The bitmask represents which letters are present in the word
                bitmaskValue |= 1 << (word.charAt(i) - 'a');
            }

            // Update the answer with the count of the current bitmask in our map if it exists
            answer += letterCombinationCount.getOrDefault(bitmaskValue, 0);

            // Increment the count for this bitmask in our map
            letterCombinationCount.put(bitmaskValue, letterCombinationCount.getOrDefault(bitmaskValue, 0) + 1);
        }

        // Return the number of similar pairs
        return answer;
    }
}
```

C++ Solution

```
#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
    }
};
```

Typescript Solution

```
function similarPairs(words: string[]): number {
    let pairCount = 0;
    const wordBitmaskCount: Map<number, number> = new Map();

    // Iterates over each word in the input array
    for (const word of words) {
        let bitmask = 0;
        // Converts each character of the word into a bitmask
        // Each bit in the integer represents the presence of a character ('a' -> 0th bit, 'b' -> 1st bit, ...)
        for (let i = 0; i < word.length; ++i) {
            bitmask |= 1 << (word.charCodeAt(i) - 'a'.charCodeAt(0));
        }

        // If a bitmask has already been seen, add its count to the answer since
        // it represents a word with a matching set of characters
        pairCount += wordBitmaskCount.get(bitmask) || 0;

        // Increment the count for this bitmask representation of a word
        wordBitmaskCount.set(bitmask, (wordBitmaskCount.get(bitmask) || 0) + 1);
    }

    return pairCount;
}
```

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 representation of word characters has been seen.

Time Complexity

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.
- 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 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 words, giving O(n).
- The variable `v` is an integer that represents a set of characters. The space for this is O(1).

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