2085. Count Common Words With One Occurrence

Easy Array Hash Table String Counting

Problem Description

The problem requires you to find the number of unique strings that appear exactly once in each of the two provided string arrays, namely `words1` and `words2`. To solve this, you need to determine the strings that are not repeated within their own arrays and then check how many of these non-repeated strings are common to both arrays.

Intuition

To approach this problem effectively, count the occurrences of each string in both arrays independently. This can be achieved by using data structures that map a string to its frequency (like a dictionary in Python). The standard choice for counting objects in Python is the `Counter` class from the `collections` module, which does exactly that: it creates a frequency map where each key is a string from the array and each value is the count of occurrences of that string.

Solution Approach

The reference solution approach is straightforward and cleverly utilizes Python's `Counter` class from the `collections` module to count the frequency of each word in both input arrays `words1` and `words2`.

Here are the steps followed in the implementation:

- Firstly, two `Counter` objects, `cnt1` and `cnt2`, are created for `words1` and `words2` respectively. These objects map each word to its frequency in the respective arrays.
- Then, we use a list comprehension combined with the `sum` function to iterate through the items of `cnt1`. We only consider the keys `k` (words) that have a value `v` (count) of exactly one since we are looking for unique occurrences.
- For each of these keys with a count of one in `cnt1`, we check if the corresponding count in `cnt2` is also exactly one, using the expression `cnt2[k] == 1`. This filters down to words that are unique in both arrays.
- If both conditions are satisfied (the word appears exactly once in both `cnt1` and `cnt2`), then the condition evaluates to `True`, which is implicitly interpreted as `1` in the summation.
- The `sum` function adds up all the `1`s, effectively counting the number of strings that meet the specified criteria.
- The result of the `sum` is the final answer and is returned by the `countWords` method.

This approach is efficient because it leverages hash maps (via the `Counter` object in Python) to count the frequency of elements with O(n) complexity, where n is the size of the input array. Consequently, the overall time complexity of the algorithm is O(n + m), where n is the length of `words1` and m is the length of `words2`, since each word in both arrays is processed exactly once.

Example Walkthrough

Let's take an example to illustrate how the solution approach works.

Suppose we have two string arrays:

```
words1 = ["apple", "banana", "cherry", "date"]
words2 = ["banana", "cherry", "apple", "elderberry", "fig", "cherry"]
```

Following the steps outlined in the solution approach:

- **Step 1:** Use the Counter class to map each word to its frequency in both arrays.

```
from collections import Counter

cnt1 = Counter(words1) # Counter({'apple': 1, 'banana': 1, 'cherry': 1, 'date': 1})
cnt2 = Counter(words2) # Counter({'banana': 1, 'cherry': 2, 'apple': 1, 'elderberry': 1, 'fig': 1})
```

We observe that in `cnt1`, every word has a frequency of 1, while in `cnt2` "cherry" has a frequency of 2 (which means it's not unique) and all others have a frequency of 1.

- **Step 2:** Using a list comprehension, we iterate through the items of `cnt1` and check if the corresponding count in `cnt2` is also 1.

```
unique_in_both = sum(1 for word in cnt1 if cnt1[word] == 1 and cnt2[word] == 1)
```

- **Step 3:** In our example, "apple" and "banana" are the words that have a count of one in both `cnt1` and `cnt2`.
  - For "apple", `cnt1["apple"]` is 1 and `cnt2["apple"]` is also 1.
  - For "banana", `cnt1["banana"]` is 1 and `cnt2["banana"]` is also 1.
  - "cherry" and "date" do not meet the criteria because "cherry" is not unique in `cnt2`, and "date" does not exist in `cnt2` at all.
- **Step 4:** So, only "apple" and "banana" are counted, giving us a sum of 2.

Therefore, the `countWords` method would return `2`, which is the number of strings that appear exactly once in each of the arrays `words1` and `words2`.

The example demonstrates the effectiveness of the approach by filtering unique occurrences efficiently through the use of Counter objects and a summation over a conditional check. The final result correctly reflects the number of unique words appearing once in both arrays.

Solution Implementation

Python

```
from collections import Counter

class Solution:
    def countWords(self, words1: List[str], words2: List[str]) -> int:
        # Count the occurrences of each word in the first list
        count_words1 = Counter(words1)

        # Count the occurrences of each word in the second list
        count_words2 = Counter(words2)

        # Sum the total number of words that appear exactly once in each list
        return sum(count_words2[word] == 1 for word, count in count_words1.items() if count == 1)
```

Java

```
class Solution {
    public int countWords(String[] words1, String[] words2) {
        // Count the occurrences of each word in both arrays
        Map<String, Integer> countWords1 = countOccurrences(words1);
        Map<String, Integer> countWords2 = countOccurrences(words2);

        int result = 0; // Initialize the result to count the words that appear exactly once in both arrays
        for (String word : words1) {
            // For each word in words1, check if it occurs exactly once in both words1 and words2
            if (countWords1.getOrDefault(word, 0) == 1 && countWords2.getOrDefault(word, 0) == 1) {
                result++; // Increment the result count for each such word
            }
        }
        return result; // Return the final count of words that appear exactly once in both arrays
    }

    // Helper method to count occurrences of each word in a given array
    private Map<String, Integer> countOccurrences(String[] words) {
        Map<String, Integer> countMap = new HashMap<>(); // Map to store word counts
        for (String word : words) {
            // Update the count of each word in the map
            countMap.put(word, countMap.getOrDefault(word, 0) + 1);
        }
        return countMap; // Return the map containing counts of each word
    }
}
```

C++

```
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int countWords(vector<string>& words1, vector<string>& words2) {
        // Create two hash maps to store the frequency of each word in words1 and words2.
        unordered_map<string, int> freqWords1;
        unordered_map<string, int> freqWords2;

        // Iterate through the first list of words and count their occurrences.
        for (const auto& word : words1) {
            freqWords1[word]++;
        }
        // Iterate through the second list of words and count their occurrences.
        for (const auto& word : words2) {
            freqWords2[word]++;
        }

        int count = 0; // This will hold the number of words that appear exactly once in both lists.
        // Iterate through the first list's frequency map.
        for (const auto& [word, freq] : freqWords1) {
            // Increment count if the word occurs exactly once in both lists.
            if (freq == 1 && freqWords2[word] == 1) {
                count++;
            }
        }
        // Return the final count of words that appear exactly once in each list.
        return count;
    }
};
```

TypeScript

```
// Import the necessary TypeScript types.
import { string, number } from 'typescript';

// Define a function to count words that appear exactly once in both lists.
function countWords(words1: string[], words2: string[]): number {
    // Create two maps to store the frequency of each word in words1 and words2.
    const freqWords1: Record<string, number> = {};
    const freqWords2: Record<string, number> = {};

    // Iterate through the first list of words and count their occurrences.
    for (const word of words1) {
        freqWords1[word] = (freqWords1[word] || 0) + 1;
    }

    // Iterate through the second list of words and count their occurrences.
    for (const word of words2) {
        freqWords2[word] = (freqWords2[word] || 0) + 1;
    }

    let count = 0; // Variable to hold the number of words that appear exactly once in both lists.

    // Iterate through the frequency map of the first list.
    for (const word in freqWords1) {
        // Increment count if the word occurs exactly once in both lists.
        if (freqWords1[word] === 1 && freqWords2[word] === 1) {
            count++;
        }
    }

    // Return the final count of words that appear exactly once in each list.
    return count;
}

// Example usage:
const wordsList1 = ['apple', 'banana', 'cherry'];
const wordsList2 = ['banana', 'apple', 'durian'];
const uniqueWordsCount = countWords(wordsList1, wordsList2);
console.log(`Number of words appearing exactly once in both lists: ${uniqueWordsCount}`);
```

```
from collections import Counter

class Solution:
    def countWords(self, words1: List[str], words2: List[str]) -> int:
        # Count the occurrences of each word in the first list
        count_words1 = Counter(words1)

        # Count the occurrences of each word in the second list
        count_words2 = Counter(words2)

        # Sum the total number of words that appear exactly once in each list
        return sum(count_words2[word] == 1 for word, count in count_words1.items() if count == 1)
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by several factors:

- Creating the first counter `cnt1` for `words1`: This involves iterating over all elements in the `words1`, so it takes `O(n)` time where `n` is the number of elements in `words1`.
- Creating the second counter `cnt2` for `words2`: Similarly, this takes `O(m)` time where `m` is the number of elements in `words2`.
- Iterating over the `cnt1` items and summing: We iterate over the counter `cnt1` items once, which takes `O(u)` time where `u` is the number of unique words in `words1`. The conditional check for `cnt2[k] == 1` is an `O(1)` operation because of the hash table lookup in the counter.

Therefore, the total time complexity is `O(n + m + u)`. In the worst case, where all the words are unique, `u` can be equal to `n`, simplifying it to `O(n + m)`.

Space Complexity

The space complexity is dominated by the two counters `cnt1` and `cnt2`:

- The counter `cnt1` stores each unique word in `words1`, which takes `O(u)` space where `u` is the number of unique words in `words1`.
- The counter `cnt2` stores each unique word in `words2`, taking `O(v)` space where `v` is the number of unique words in `words2`.

The total space complexity is `O(u + v)`. In the worst case, where all the words are unique, `u` can be equal to `n` and `v` equal to `m`, which makes the space complexity `O(n + m)`.