1002. Find Common Characters

Easy Array Hash Table String

Problem Description

The given problem requires us to find the common characters across all strings in the given array of strings `words`. The characters need to be counted with their frequency, which means if a character appears twice in all strings, it should appear twice in the resulting array. The characters in the resulting array can be in any order, and we need to take into account duplicates as well.

Intuition

To solve this problem, we can take the following approach:

1. Start with counting the frequency of characters in the first string. This gives us a starting point and includes all characters, as we are looking for characters that are common in all strings.
2. Traverse through the remaining strings and, for each string:
   - Count the frequency of characters in the current string.
   - Compare the frequency of each character with the frequency in our current count (initialized from the first string).
   - Update the count to the minimum frequency found between the current count and the frequency in the current string. This step ensures that we only keep the count of characters that are common across all strings processed so far.
3. Once we have the final count of characters that are common to all strings in `words`, we can create our resulting array. We add each character to the array based on the frequency recorded in our count.

The implementation begins by using a `Counter` from the `collections` module in Python, which is a subclass of `dict`. It's specifically designed to count hashable objects, in our case, characters in a string.

Here's the step-by-step breakdown of the code:

1. Initialize the counter with the characters of the first word:

   ```
   cnt = Counter(words[0])
   ```

   This gives us a dictionary-like object where keys are characters and values are their frequencies in the first string.

2. Iterate over all the other words in the `words` list:

   ```
   for w in words:
       ccnt = Counter(w)
   ```

   For each word `w`, we create its character counter `ccnt`.

3. For every character in `cnt` (which contains the character count from the first word), we take the minimum frequency found in both the `cnt` and the character counter for the current word `ccnt`:

   ```
   for c in cnt.keys():
       cnt[c] = min(cnt[c], ccnt[c])
   ```

   This step essentially updates `cnt` so that it only contains characters that are present in every word encountered so far and in the smallest frequency among them. It's the intersection part of the algorithm.

4. After iterating through all the strings and updating the counter, we have our final `cnt` which includes only the common characters. Lastly, we need to convert this count into a list of characters.

   ```
   ans = []
   for c, v in cnt.items():
       ans.extend([c] * v)
   ```

   For each character and its count in `cnt`, we extend our answer list `ans` with that character repeated `v` times. This repeats a character as many times as it appears in all strings.

This solution ensures that we only keep characters that are common to all words and respects their minimum frequency across the words. The use of `Counter` and dictionary operations makes the implementation concise and efficient.

Example Walkthrough

Let's walk through a small example to understand the solution approach presented. Suppose our `words` array is:

```
words = ["bella", "label", "roller"]
```

We want to find the common characters across all these words, with their exact frequency. Let's apply our intuition step by step:

Step 1: First Word's Counter: We initialize the counter with the first word `bella`. The counter would be something like this:

```
cnt = Counter("bella")  # cnt is {'b': 1, 'e': 1, 'l': 2, 'a': 1}
```

Step 2: Traverse Remaining Words Now we go through the other words in the list: "label" and "roller". Let's take them one by one.

- For the second word, `label`:

  ```
  ccnt = Counter("label")  # ccnt is {'l': 2, 'a': 1, 'b': 1, 'e': 1}
  ```

- Now for every character in our initial counter `cnt`, we update the counts to the minimum we find in `ccnt`. For the 'l' character, both counters have a count of 2, so we keep it as 2 in `cnt`. We do the same for other characters:

  ```
  # After updating with 'label'
  cnt = {'b': 1 ('bella' had 1, 'label' had 1),
         'e': 1 ('bella' had 1, 'label' had 1),
         'l': 2 ('bella' had 2, 'label' had 2),
         'a': 1 ('bella' had 1, 'label' had 1)}
  ```

- Next, for the third word, `roller`:

  ```
  ccnt = Counter("roller")  # ccnt is {'r': 2, 'o': 1, 'l': 2, 'e': 1}
  ```

- We again update the `cnt` to keep only the minimum count for each character found in both counters:

  ```
  # After updating with 'roller'
  cnt = {'b': 0 ('bella' had 1, 'roller' had 0),
         'e': 1 ('bella' had 1, 'roller' had 1),
         'l': 2 ('bella' had 2, 'roller' had 2),
         'a': 0 ('bella' had 1, 'roller' had 0)}
  ```

Characters 'b' and 'a' are not present in 'roller', so their count is updated to zero.

Step 3: Building the Result Array Finally, we create the resulting array by adding each character from `cnt` its counted number of times:

```
ans = []
for c, v in cnt.items():
    ans.extend([c] * v)  # ['e', 'l', 'l']
```

After traversing all the strings and updating the counter, we only kept 'e' and 'l', each character appearing respectively once and twice, since those are the only ones common to all the strings in the exact minimum frequency.

Conclusively, our resulting array for common characters across all strings in `words` would be:

```
['e', 'l', 'l']
```

This illustrates how the described algorithm works to find common characters among an array of strings taking into account their frequency.

Solution Implementation

Python

```python
from collections import Counter  # Importing Counter class from collections module

class Solution:
    def commonChars(self, words: List[str]) -> List[str]:
        # Initialize a Counter for the first word to keep track of letter frequencies
        char_count = Counter(words[0])

        # Iterate through all words in the input list
        for word in words:
            # Create a Counter for the current word
            current_count = Counter(word)

            # Check all the characters in the initial word's counter
            for char in list(char_count):
                # Update the character's count to be the minimum of the current and the first word's count
                # This ensures we only keep as many of a character as is common to all words so far
                char_count[char] = min(char_count[char], current_count[char])

        # Initialize an empty list to hold the common characters
        common_characters = []

        # Iterate through the items in the final char_count
        for char, count in char_count.items():
            # For each character, add it to the list as many times as its count
            common_characters.extend([char] * count)

        # Return the list of common characters
        return common_characters
```

Java

```java
import java.util.List;
import java.util.Arrays;
import java.util.ArrayList;

class Solution {
    public List<String> commonChars(String[] words) {
        // Initialize a count array to track the minimum frequency of each letter
        int[] letterCount = new int[26];

        // Start by setting each letter's frequency to a high value
        Arrays.fill(letterCount, Integer.MAX_VALUE);

        // Iterate over each word in the input array
        for (String word : words) {
            // Initialize a temporary array to store the frequency of each letter in the current word
            int[] currentWordCount = new int[26];
            // Loop through each character in the current word and increment its frequency
            for (char letter : word.toCharArray()) {
                currentWordCount[letter - 'a']++;
            }

            // Update the letterCount array to keep the minimum frequency among the words processed so far
            for (int i = 0; i < 26; ++i) {
                letterCount[i] = Math.min(letterCount[i], currentWordCount[i]);
            }
        }

        // Prepare the result list
        List<String> result = new ArrayList<>();

        // Add the common characters to the result list, based on the letterCount frequencies
        for (int i = 0; i < 26; ++i) {
            while (letterCount[i]-- > 0) {
                // Append the character to the result list
                result.add(String.valueOf((char) (i + 'a')));
                // Decrement the count for this letter
                letterCount[i]--;
            }
        }
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<string> commonChars(vector<string>& words) {
        // Initialize a count array for 26 letters with a high number to represent infinity.
        int letterCount[26];
        memset(letterCount, 0x3f, sizeof(letterCount));

        // Loop through each word in the words vector.
        for (auto& word : words) {
            // Local count for letters in the current word.
            int wordLetterCount[26] = {0};

            // Count each letter in the current word.
            for (char letter : word) {
                ++wordLetterCount[letter - 'a'];
            }

            // Compare counts for each letter with the global count and take the minimum.
            for (int i = 0; i < 26; ++i) {
                letterCount[i] = min(letterCount[i], wordLetterCount[i]);
            }
        }

        // Prepare the result vector to store common letters.
        vector<string> result;
        for (int i = 0; i < 26; ++i) {
            // Add the appropriate number of the current letter to the result.
            while (letterCount[i]-- > 0) {
                result.emplace_back(1, static_cast<char>(i + 'a'));
                --letterCount[i];
            }
        }

        return result; // Return the final result.
    }
};
```

TypeScript

```typescript
function commonChars(words: string[]): string[] {
  // Initialize a frequency array to keep track of each character's minimum occurrence across all words
  const minFrequencies: number[] = new Array(26).fill(Infinity);

  // Iterate over each word in the input array
  for (const word of words) {
    // Temporary frequency array for the current word
    const wordFrequencies: number[] = new Array(26).fill(0);

    // Populate the frequency array for the current word
    for (const char of word) {
      const charIndex = char.charCodeAt(0) - 'a'.charCodeAt(0);
      wordFrequencies[charIndex]++;
    }

    // Update the minFrequencies array with the minimum frequency of each character
    for (let i = 0; i < 26; ++i) {
      minFrequencies[i] = Math.min(minFrequencies[i], wordFrequencies[i]);
    }
  }

  // The result array to hold common characters
  const result: string[] = [];

  // Build the result array using characters that appear across all words based on min frequencies
  for (let i = 0; i < 26; ++i) {
    // Continue appending the character to the result as long as the frequency is greater than 0
    while (minFrequencies[i]-- > 0) {
      result.push(String.fromCharCode(i + 'a'.charCodeAt(0)));
      minFrequencies[i]--;
    }
  }

  return result; // Return the array containing all common characters
}
```

```python
from collections import Counter  # Importing Counter class from collections module

class Solution:
    def commonChars(self, words: List[str]) -> List[str]:
        # Initialize a Counter for the first word to keep track of letter frequencies
        char_count = Counter(words[0])

        # Iterate through all words in the input list
        for word in words:
            # Create a Counter for the current word
            current_count = Counter(word)

            # Check all the characters in the initial word's counter
            for char in list(char_count):
                # Update the character's count to be the minimum of the current and the first word's count
                # This ensures we only keep as many of a character as is common to all words so far
                char_count[char] = min(char_count[char], current_count[char])

        # Initialize an empty list to hold the common characters
        common_characters = []

        # Iterate through the items in the final char_count
        for char, count in char_count.items():
            # For each character, add it to the list as many times as its count
            common_characters.extend([char] * count)

        # Return the list of common characters
        return common_characters
```

Time and Space Complexity

The given code snippet defines a function `commonChars` in the `Solution` class, which finds common characters among all strings in a given list.

Time Complexity:

Let `n` be the number of strings in the `words` list and `k` be the average length of these strings.

1. We initialize a counter `cnt` with the first word in the list, which takes $O(k)$ time.
2. Then, we iterate over each word in the `words` list.
   - For each word, we initialize a counter `ccnt`, which also takes $O(k)$ time.
   - We iterate over each key in `cnt` to update it with the minimum frequency found in `ccnt`, and since `cnt` never has more characters than there are in the alphabet, let's denote the alphabet size as `a` and this nested loop runs in $O(a)$ time for each word.

Overall, the time complexity looks like this:

For the loop over words: $O(n * (O(k) + O(a))$ We take O(a) from inside as it is constant and does not depend on `n` or `k`.

Since $O(a)$ is constant and typically $a <= 26$ (for English alphabet), we can simplify the expression:

So, the total time complexity is: $O(nwk + nwa)$ which simplifies to $O(nwk)$ because `nwk` will typically dominate `nwa`.

Space Complexity:

1. The counter `cnt` uses $O(a)$ space.
2. For each word, a temporary counter `ccnt` is created, which also uses $O(a)$ space, however since it is not preserved after each iteration, it doesn't add up with the space complexity.
3. The answer list `ans` in the worst case can contain all characters from all strings if all characters are the same which is $O(nwk)$.

Thus the total space complexity is $O(a + nwk)$. Since `a` is a constant and typically $a <= 26$, the main factor here is `nwk`.

In conclusion, the space complexity of the code is $O(nwk)$ (since we're considering the space for the output, which can be as large as the input in the worst case).