916. Word Subsets

Medium Array Hash Table String

Leetcode Link

Problem Description

The problem provides two arrays of strings, `words1` and `words2`. The goal is to determine which strings from `words1` are considered "universal." A string is universal if every string in `words2` is a subset of it. A string `b` is a subset of another string `a` if `a` contains all the letters of `b` in the same number or more. That means for every character and its count in `b`, this character must appear at least the same number of times in `a`. For example, `'wrr'` is a subset of `'warrior'` but not of `'world'`. The task is to return an array of universal strings from `words1`, and the order of results does not matter.

Intuition

To find the universal strings efficiently, rather than comparing each string in `words1` with all the strings in `words2`, we can create a letter frequency count that represents the maximum frequency of each character across all strings in `words2`. This way, we only need to compare each string in `words1` with this combined frequency count.

Firstly, we create a `Counter` (which is a dictionary subclass for counting hashable objects) to hold the maximum frequency of characters required from `words2`. For each string in `words2`, we count its characters and update our maximum frequency `Counter`. This counter will tell us the least number of times each character should appear in any string from `words1` for it to be considered universal.

After this, we iterate through each string in `words1` and create a frequency count for it. Then, we check if this count satisfies the requirements of our precomputed maximum frequency counter. The `all()` function helps in determining whether all conditions are true for each character's count.

If a string `a` from `words1` has at least as many occurrences of each character as computed in the maximum frequency `Counter`, then it is universal. We append such strings to our answer list `ans`.

This solution approach minimizes the number of total comparisons needed to identify the universal strings, and therefore is optimized for the task.

Solution Approach

To solve this problem, the code implementation makes use of Python's `Counter` class from the `collections` module. The `Counter` class is a specialized dictionary used for counting hashable objects, in our case, characters in strings. It's particularly useful because it allows us to easily compare the frequencies of letters in different strings.

Here's a step-by-step walkthrough of the implementation:

1. Initialize a new `Counter` object called `cnt`, which will be used to store the maximum frequency requirements for each letter, derived from `words2`.

```python
cnt = Counter()
```

2. Loop over each string `b` in the array `words2`. For each string:

   a. Create a temporary counter `t` from `b`.

   b. Update the `cnt` counter with the maximum frequency of characters from `t`. This means `cnt[c] = max(cnt[c], v)` for each character `c` and its count `v` in `t`. This ensures that `cnt` holds the maximum number of times any character needs to appear for a string from `words1` to be universal.

```python
for b in words2:
    t = Counter(b)
    for c, v in t.items():
        cnt[c] = max(cnt[c], v)
```

3. Initialize an empty list `ans`, which will hold all the universal strings from `words1`.

```python
ans = []
```

4. Loop over each string `a` in `words1`. For each string:

   a. Create a temporary counter `t` from `a`.

   b. Use the `all` function to check if `a` contains at least as many of each character as required by the `cnt` counter. The comparison `v <= t[c] for c, v in cnt.items()` ensures that `a` meets the criteria for all characters `c` with their respective counts `v` in `cnt`.

   c. If `a` meets the criteria, append it to the `ans` list.

```python
for a in words1:
    t = Counter(a)
    if all(v <= t[c] for c, v in cnt.items()):
        ans.append(a)
```

5. Finally, return the list `ans`, which now contains all the universal strings from `words1`.

```python
    return ans
```

The primary algorithm patterns used in this solution include frequency counting and iterating with condition checking. By using the `Counter` class with maximum frequency logic, the code efficiently reduces what could be an O(n * m) problem (checking each of n strings in `words1` against each of m strings in `words2`) to a more manageable O(n + m) problem by minimizing repetitive checks.

Example Walkthrough

Consider the following example to illustrate the solution approach:

Let `words1` be `["ecology", "universal", "computer"]` and `words2` be `["cool", "envy", "yon", "yes"]`.

1. The maximum frequency counters (`cnt`) are determined from `words2`. For "cool", it will be `{'c': 1, 'o': 2, 'l': 1}`, for "envy", it will be `{'e': 1, 'n': 1, 'v': 1, 'y': 1}` and so forth. Ultimately, after comparing all `words2` strings, `cnt` would be:

```python
cnt = {'c': 1, 'o': 2, 'l': 1, 'e': 1, 'n': 1, 'v': 1, 'y': 2, 's': 1}
```

   This `cnt` means for a string in `words1` to be universal; it must have at least one 'c', two 'o's, one 'l', one 'e', one 'n', one 'v', two 'y's, and one 's'.

2. Now we check each word in `words1` against `cnt`.

   - "ecology": It's counter is `{'e': 1, 'c': 1, 'o': 1, 'l': 1, 'g': 1, 'y': 1}`. While this word has all the required characters, it fails to meet the count for 'o' and 'y'. Thus, "ecology" is not universal.
   - "universal": Counter is `{'u': 1, 'n': 1, 'i': 1, 'v': 1, 'e': 1, 'r': 1, 's': 1, 'a': 1, 'l': 1}`. Despite having all characters, it does not have enough 'o's and 'y's. So, "universal" is not universal.
   - "computer": Counter is `{'c': 1, 'o': 1, 'm': 1, 'p': 1, 'u': 1, 't': 1, 'e': 1, 'r': 1}`. Even though "computer" meets or exceeds the count for 'c', 'e', 't', 'r', it doesn't contain any 'l', 'v', 'y', or sufficient 'o's. Therefore, it's not universal either.

3. Applying this logic, none of the words in `words1` are universal, as none meet the frequency criteria established by `words2`. The final answer `ans` list would be empty.

The result is an efficient check that precisely tells us that there are no strings in `words1` that are universal with respect to `words2`. Thus, the function would return an empty list `[]`.

Python Solution

```python
from collections import Counter

class Solution:
    def wordSubsets(self, words1: List[str], words2: List[str]) -> List[str]:
        # Create a counter to store the maximum frequency of each character
        # across all words in words2
        max_freq_counter = Counter()
        for word in words2:
            word_freq_counter = Counter(word)
            for char, freq in word_freq_counter.items():
                # Update the counter for each character to the maximum frequency
                max_freq_counter[char] = max(max_freq_counter[char], freq)

        # Initialize a list to keep all words from words1 that meet the criteria
        universal_words = []
        # Iterate through each word in words1 to check if it is a universal word
        for word in words1:
            word_freq_counter = Counter(word)
            # Check if word has at least as many of each character as needed
            is_universal = all(freq <= word_freq_counter[char] for char, freq in max_freq_counter.items())
            # If the word meets the criteria, add it to the universal words list
            if is_universal:
                universal_words.append(word)

        # Return the list of universal words
        return universal_words

```

Java Solution

```java
class Solution {
    public List<String> wordSubsets(String[] universalSet, String[] subsetWords) {
        // This array will keep the max frequency of each letter required by subsetWords
        int[] maxSubsetFreq = new int[26];

        // Calculate the max frequency of each character across all words in subsetWords
        for (String subsetWord : subsetWords) {
            // Temporary array to store frequency of each character in the current word
            int[] tempFreq = new int[26];
            for (char ch : subsetWord.toCharArray()) {
                // Increment character frequency
                tempFreq[ch - 'a']++;

                // Update the maxSubsetFreq array with the maximum frequency needed for this character
                maxSubsetFreq[ch - 'a'] = Math.max(maxSubsetFreq[ch - 'a'], tempFreq[ch - 'a']);
            }
        }

        // This will store our final result
        List<String> result = new ArrayList<>();

        // Loop through each word in universalSet
        for (String word : universalSet) {
            // Temporary array to store frequency of each character in the current word
            int[] wordFreq = new int[26];
            for (char ch : word.toCharArray()) {
                // Increment character frequency
                wordFreq[ch - 'a']++;
            }

            // Check if the current word contains all the required characters in proper frequency
            boolean isUniversal = true;
            for (int i = 0; i < 26; ++i) {
                if (maxSubsetFreq[i] > wordFreq[i]) {
                    // If any character is found in less frequency than required,
                    // mark word as non-universal, and break the loop
                    isUniversal = false;
                    break;
                }
            }

            // If the word is universal, add it to the result list
            if (isUniversal) {
                result.add(word);
            }
        }

        // Return the list of all universal words
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <string>
#include <cstring>

class Solution {
public:
    // Determines the words from words1 that are universal for words2.
    vector<string> wordSubsets(vector<string>& words1, vector<string>& words2) {
        int maxCharFrequencies[26] = {0}; // Array to store the maximum frequency of each character required across all words in wo
        int currentWordFrequencies[26]; // Array to store the frequency of characters in the current word

        // Calculate the maximum frequency of each character across all words in words2
        for (const auto& wordB : words2) {
            memset(currentWordFrequencies, 0, sizeof(currentWordFrequencies));
            for (const char &ch : wordB) {
                currentWordFrequencies[ch - 'a']++;
            }
            // Update maxCharFrequencies with the maximum frequency for each character
            for (int i = 0; i < 26; ++i) {
                maxCharFrequencies[i] = std::max(maxCharFrequencies[i], currentWordFrequencies[i]);
            }
        }

        vector<string> universalWords; // Vector to store the universal words

        // Iterate over each word in words1 to determine if it is universal.
        for (const auto& wordA : words1) {
            memset(currentWordFrequencies, 0, sizeof(currentWordFrequencies));
            for (const char &ch : wordA) {
                currentWordFrequencies[ch - 'a']++;
            }
            // Check if wordA has at least the maximum frequency of each character required by words in words2
            bool isUniversal = true;
            for (int i = 0; i < 26; ++i) {
                if (maxCharFrequencies[i] > currentWordFrequencies[i]) {
                    isUniversal = false;
                    break;
                }
            }
            // If the word is universal, add it to the resulting vector.
            if (isUniversal) {
                universalWords.emplace_back(wordA);
            }
        }

        return universalWords; // Return the vector containing all universal words
    }
};

```

Typescript Solution

```typescript
// Type definition for a frequency array representing characters 'a' to 'z'
type CharFrequencyArray = number[];

// Calculates character frequencies in a given word
function calculateCharFrequencies(word: string): CharFrequencyArray {
    const frequencies: CharFrequencyArray = new Array(26).fill(0);
    for (const char of word) {
        frequencies[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }
    return frequencies;
}

// Determine the words from words1 that are universal for words2
function wordSubsets(words1: string[], words2: string[]): string[] {
    let maxCharFrequencies: CharFrequencyArray = new Array(26).fill(0); // Store max frequency of each char across all words in wor

    // Calculate max frequency of each character required by words in words2
    for (const word of words2) {
        const currentWordFrequencies = calculateCharFrequencies(word);
        // Update max frequencies
        for (let i = 0; i < 26; i++) {
            maxCharFrequencies[i] = Math.max(maxCharFrequencies[i], currentWordFrequencies[i]);
        }
    }

    const universalWords: string[] = []; // Storage for universal words

    // Check each word in words1 for universality
    for (const word of words1) {
        const currentWordFrequencies = calculateCharFrequencies(word);
        let isUniversal = true; // Assume word is universal until proven otherwise

        // If word has fewer of any character than required, not universal
        for (let i = 0; i < 26; i++) {
            if (maxCharFrequencies[i] > currentWordFrequencies[i]) {
                isUniversal = false;
                break;
            }
        }

        // Add universal words to the result list
        if (isUniversal) {
            universalWords.push(word);
        }
    }

    return universalWords; // Return the list of all universal words
}

```

Time and Space Complexity

Time Complexity

The given code has two main sections contributing to the time complexity:

1. **Building the Universal Counter**:
   - Iterating over each word in `words2` and updating the `cnt` counter for each character.
   - If `M` is the average length of words in `words2` and `N2` is the number of words in `words2`, this part is `O(M * N2)` in the worst case.
2. **Checking if Words in `words1` are Universal**:
   - For each word in `words1`, a counter is created and checked against the `cnt` counter.
   - If `L` is the average length of words in `words1` and `N1` is the number of words in `words1`, and if `K` is the number of unique characters in `cnt` (bounded by the alphabet size), then checking each word has complexity `O(L + K)`, and doing this for all words in `words1` gives `O(N1 * (L + K))`.

Combining the two we get a total time complexity of `O(M * N2) + O(N1 * (L + K))`.

- Note that `K` is bounded by the alphabet size which can be considered a constant, hence it sometimes can be omitted from the Big-O notation, simplifying to `O(M * N2 + N1 * L)`.

Space Complexity

The space complexity of the code comprises the following factors:

1. **The Universal Counter `cnt`**:
   - The maximum space required is the size of the alphabet, let's denote it as `a`, which is constant. Hence, this is `O(a)`.
2. **Temporary Counter for Each Word in `words1`**:
   - At most, the size of the alphabet `a` for each word. As this is temporary and not used simultaneously, it's still `O(a)`.
3. **The `ans` List**:
   - At most, it can be as big as `N1` in case all `words1` are universal. So the space for it would be `O(N1)`.

Considering all, the total space complexity is `O(a) + O(N1)` which simplifies to `O(N1)` because `a` is constant and generally considered negligible compared to `N1`.