1170. Compare Strings by Frequency of the Smallest Character

Medium Array Hash Table String Binary Search Sorting

Leetcode Link

Problem Description

The problem presents a function `f(s)` that calculates the frequency of the lexicographically smallest character within a non-empty string `s`. For instance, if `s = "dcce"`, then `f(s)` equals 2 because the smallest character 'c' appears twice in the string.

Given two arrays, one containing strings `words` and the other containing query strings `queries`, the task is to determine how many words in `words` have a higher `f` value than the `f` value of each query string in `queries`. Specifically, for each query `queries[i]`, you need to count the number of strings in `words` where `f(queries[i])` is less than `f(W)` for each string `W` in `words`.

The expected output is an array `answer`, where `answer[i]` corresponds to the count for `queries[i]`. Essentially, the output tells us for each query, the count of words in `words` with a smaller lexicographically smallest character frequency.

Intuition

The intuition behind the solution involves first understanding what the function `f` does in terms of string processing. It counts the number of times the smallest lexicographical character appears in a string.

Approaching the solution, we consider these steps:

1. Precompute the `f` values for all strings in `words` since we will need to compare these with each query's `f` value. This precomputation helps to avoid recalculating these values for each query, which would be inefficient.
2. Sort the precomputed `f` values of the `words` array. With a sorted list of `f` values, we can efficiently determine where a particular value would be placed; all elements larger than this value would come after it in the sorted array.
3. For each query, calculate its `f` value. Then, to find out how many `f` values in `words` are larger than that of the query, we use binary search to find the insertion point in the sorted list of `f` values of `words`. This gives us the position where the `f` value of the query would fit if it were to be inserted into the list, effectively giving us the count of `f` values that are greater because all values to the right of this point are larger.
4. The `bisect_right` function from Python's `bisect` module is used to perform the binary search. It returns the insertion point (index) in the sorted `nums` list, which allows us to subtract this index from the total number of words to get the count of words with a larger `f` value than the query.
5. The final result for each query is calculated by taking the total number of words `n` and subtracting the insertion index. This count is added to the result list, which is returned at the end.

The process leverages binary search for efficiency, ensuring that an overall time complexity better than O(n*2) is achieved for the solution, where n is the length of the longest array between `queries` and `words`.

Solution Approach

The implementation of the solution to this problem uses Python's `Counter` class from the `collections` module, the `ascii_lowercase` string from the `string` module, and the `bisect_right` function from the `bisect` module. Here's how these are used step-by-step:

1. **Define the function `f(s)`:** The function `f(s)` is where the calculation of the smallest character's frequency is implemented. It uses the `Counter` class to count the occurrences of each character in string `s`. `Counter(s)` returns a dictionary-like object where characters are keys, and their counts are values.
2. **Find the smallest character's frequency:** Once we have the counts, we iterate over `ascii_lowercase` (a string containing all lowercase letters in alphabetical order) to find the smallest character present in `s`. As soon as we find a character `c` that is in `s` (i.e., `cnt[c]` is not zero), we return its count `cnt[c]`. Using `ascii_lowercase` ensures we check characters in lexicographical order.
3. **Precompute `f` for `words`:** We create a list called `nums` that holds the `f` value for each word in `words`. This is done with a list comprehension `[f(w) for w in words]`, calculating `f(w)` for each `w` in `words`.
4. **Sort `nums`:** `nums` is sorted so that binary search can be used to efficiently find positions within it. We use Python's built-in sorted method for this purpose.
5. **Find each query's count:** For each query `p` in `queries`, we calculate `f(p)` and then find the index at which `f(p)` would be inserted while maintaining the order in `nums`. This is where `bisect_right` comes in. It returns the insertion point to the right of existing values of `f(p)`. This index tells us how many words in `words` have a larger `f` value than `f(p)`.
6. **Prepare the result:** Knowing the index from `bisect_right`, the number of words with a larger `f` value is found by subtracting this index from the total number of words, which is `len(words)`. The comprehension `[n - bisect_right(nums, f(q)) for q in queries]` creates the result list by performing this calculation for each query.
7. **Return the result**: The resultant list is returned, which contains the count for each query indicating how many words in `words` have a higher `f` value than that of the query.

In conclusion, the solution makes use of a combination of string processing, precomputation, binary search, and efficient sorting to arrive at the result. It leverages the properties of sorted arrays and binary search (`bisect_right`) to perform queries in logarithmic time, after an initial sort cost, allowing for an efficient and scalable approach to solve the given problem.

Example Walkthrough

Let's go through an example to illustrate the solution approach described above. Suppose we have a list of words, `words = ["a", "aa", "aaa", "aaaa"]` and a list of queries, `queries = ["a", "aa", "aaa", "aaaa"]`. Our goal is to count how many words have a higher frequency of the lexicographically smallest character for each query. Let's walk through each step of the approach.

1. **Define and calculate `f` for each word**: We first define the function `f(s)` to calculate the frequency of the lexicographically smallest character in a given string `s`. For the `words` list, since all words consist of the character 'a', the `f` values are simply the lengths of the strings. Thus, `f("a") = 1`, `f("aa") = 2`, and so on.
2. **Precompute `f` values for `words`:** We precompute the `f` values for the `words`. This gives us `nums = [1, 2, 3, 4]`.
3. **Sort the `nums` list**: We sort the precomputed `f` values of `words`. Our `nums` list is already sorted: `nums = [1, 2, 3, 4]`.
4. **Find each query's count**: Now, we need to find out for each query how many `f` values in `words` are larger than the query's `f` value. For `queries[i] = "a"`, `f("a")` is `1`. Using binary search, we determine the position where `1` would fit in the sorted `nums` list (which is index `1`, right after the first occurrence of `1`). Since there are a total of 4 words, and `1` would be inserted at index `1`, there are `4 - 1 = 3` words with a larger `f` value.

   Repeating this for `queries[1] = "aa"` with `f("aa") = 2`, binary search tells us that `2` would be inserted at index `2` in `nums`, leaving `4 - 2 = 2` words with a larger `f` value.

   Lastly, for `queries[2] = "aaa"`, `f("aaa") = 3`, binary search gives us an insertion index of `3`, so there are `4 - 3 = 1` word with a larger `f` value.
5. **Prepare the result**: The counts we determined in the previous step give us our final result array: `[3, 2, 1]`.
6. **Return the result**: The result for our query is `[3, 2, 1]`, indicating for each query in `queries`, how many words in `words` have a higher `f` value.

The example confirms our approach, showing how we can effectively calculate the counts required using precomputation, sorting, and binary search.

Python Solution

```python
from collections import Counter
from bisect import bisect_right
from typing import List

class Solution:
    def numSmallerByFrequency(self, queries: List[str], words: List[str]) -> List[int]:
        # Helper method to calculate the frequency of the smallest character
        def calculate_frequency(s: str) -> int:
            # Count the occurrences of each character
            char_count = Counter(s)
            # Iterates over lowercase ascii characters to find the smallest character and return its count
            for c in 'abcdefghijklmnopqrstuvwxyz':
                if c in char_count:
                    return char_count[c]
            return 0  # In case the string is empty, though the problem assumes non-empty strings

        # Calculates the frequency for each word and sort the results
        word_frequencies = sorted(calculate_frequency(word) for word in words)
        # List to hold the result for each query
        results = []
        # Calculate the number of words with higher frequency than each query
        for query in queries:
            query_frequency = calculate_frequency(query)
            # Use binary search to find the number of words with higher frequency
            num_of_higher_frequency_words = len(word_frequencies) - bisect_right(word_frequencies, query_frequency)
            # Append the result to the results list
            results.append(num_of_higher_frequency_words)
        return results  # Returns the final list of results
```

Java Solution

```java
class Solution {

    // Function to count how many strings in `words` have a frequency smaller than the frequency of each string in `queries`
    public int[] numSmallerByFrequency(String[] queries, String[] words) {
        int[] answer = new int[queries.length];
        int[] frequencies = new int[words.length];

        // Calculate frequency of smallest char for each word
        for (int i = 0; i < words.length; i++) {
            frequencies[i] = smallestCharFrequency(words[i]);
        }

        // Sort the frequencies
        Arrays.sort(frequencies);

        // For each query, count how many words have a smaller frequency
        int queriesLength = queries.length;
        for (int i = 0; i < queriesLength; i++) {
            int queryFrequency = smallestCharFrequency(queries[i]);
            // Binary search to find the number of words with a larger frequency
            int left = 0;
            int right = words.length;
            while (left < right) {
                int mid = (left + right) >> 1;
                if (frequencies[mid] > queryFrequency) {
                    right = mid;
                } else {
                    left = mid + 1;
                }
            }
            // The result for this query is the number of frequencies greater than the query frequency
            answer[i] = words.length - left;
        }

        return answer;
    }

    // Helper function to calculate the frequency of the smallest character in a string
    private int smallestCharFrequency(String s) {
        // Count the occurrences of each character (there are 26 lowercase English letters)
        int[] charCounts = new int[26];
        for (int i = 0; i < s.length(); ++i) {
            charCounts[s.charAt(i) - 'a']++;
        }
        // Find the smallest non-zero frequency
        for (int count : charCounts) {
            if (count > 0) {
                return count;
            }
        }
        return 0; // Return 0 if the string was empty (though this should not happen given the problem constraints)
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm> // Needed for sorting and upper_bound

using std::vector;
using std::string;
using std::sort;
using std::upper_bound;

class Solution {
public:
    // This method will take a vector of query strings and a vector of word strings
    // and will return a vector with the count of words that have a higher frequency
    // of the smallest character than the frequency of the smallest character in each query string.
    vector<int> numSmallerByFrequency(vector<string>& queries, vector<string>& words) {
        // Lambda function to calculate the frequency of the smallest character in a string
        auto f = [](const string& s) {
            int count[26] = {0}; // Initialize an array to store the count of each character from 'a' to 'z'
            // Count how many times each character appears in the string
            for (char c : s) {
                count[c - 'a']++;
            }
            // Find and return count of the first non-zero frequency (smallest character)
            for (int frequency : count) {
                if (frequency) {
                    return frequency;
                }
            }
            return 0;
        };

        int numWords = words.size(); // Cache the number of words for efficiency
        vector<int> wordFrequencies(numWords); // This array will hold the frequencies of the smallest char in each word

        // Calculate the frequency of each word's smallest character
        for (int i = 0; i < numWords; ++i) {
            wordFrequencies[i] = f(words[i]);
        }

        // Sort the frequencies in non-decreasing order to perform efficient lookups later
        sort(wordFrequencies.begin(), wordFrequencies.end());

        // Prepare the answer vector to hold the results
        vector<int> answer;

        // Iterate the queries to find out the respective counts
        for (const string& query : queries) {
            int queryFrequency = f(query); // Get frequency of current query

            // Find how many words have a greater frequency than the current query frequency
            // The number of words with a greater frequency is the total number of words minus
            // the number of words with a frequency less than or equal to the query frequency
            int greaterFrequencyCount = numWords - (upper_bound(wordFrequencies.begin(), wordFrequencies.end(), queryFrequency) - wordFrequencies.begin());

            // Add the count to the answer vector
            answer.push_back(greaterFrequencyCount);
        }

        // Return the filled answer vector
        return answer;
    }
};
```

Typescript Solution

```typescript
// Function to count the smallest character frequency in a given string.
function frequencyOfSmallestChar(s: string): number {
    const charCountArr = new Array(26).fill(0); // Initialize an array for alphabets.
    // Count frequency of each character in the string.
    for (const char of s) {
        frequencyCounter[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }
    // Find and return the frequency of the smallest character.
    return frequencyCounter.find(count => count !== 0);
}

// The main function that applies the frequencyOfSmallestChar function on queries compared to words.
function numSmallerByFrequency(queries: string[], words: string[]): number[] {
    // Map the 'words' array into an array of smallest character frequencies
    // and sort it in ascending order.
    const sortedWordsFrequencies = words.map(frequencyOfSmallestChar).sort((a, b) => a - b);
    const answer = new Array(queries.length - 1); // Initialize an array to store the results.

    // Process each query string.
    for (let i = 0; i < queries.length; i++) {
        // Find the frequency of the smallest character in the current query string.
        const queryFrequency = frequencyOfSmallestChar(queries[i]);
        let leftIndex = 0,
            rightIndex = sortedWordsFrequencies.length;

        // Perform a binary search to find the count of elements in 'sortedWordsFrequencies'
        // that are greater than 'queryFrequency'.
        while (leftIndex < rightIndex) {
            const midIndex = Math.floor((leftIndex + rightIndex) / 2);
            if (sortedWordsFrequencies[midIndex] > queryFrequency) {
                rightIndex = midIndex;
            } else {
                leftIndex = midIndex + 1;
            }
        }

        // Push the count (length of the array minus the final leftIndex) to the result array.
        answer[i] = sortedWordsFrequencies.length - leftIndex;
    }
    return answer; // Return the final result array.
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code involves several components:

1. Calculating the frequency of the smallest character in each word using the `f` function. This operation involves creating a Counter for each word, which has a time complexity of $O(k)$, where k is the average string length. This is done for all words, so for n words, this part would have a time complexity of $O(n * k)$.
2. Sorting the frequencies of words. The sorted function has a time complexity of $O(n \log n)$, where n is the length of the `words` list.
3. For each query in `queries`, the function calculates the frequency of the smallest character and uses binary search (`bisect_right`) to find the position in the sorted list of word frequencies. The frequency calculation is $O(w)$, and binary search in a sorted list of size n is $O(\log n)$. Therefore, for q queries, this part has a time complexity of $O(q * (w + \log n))$.

Putting it all together, the total time complexity is $O(n * m) + O(n \log n) + O(q * (w + \log n))$.

Space Complexity

The space complexity is the additional space used by the algorithm:

1. Storing the frequencies of all words, which is $O(n)$ space.
2. The sorted frequency list, which again takes $O(n)$ space.
3. The counter created for each word, which in the worst case takes $O(k)$ space; however, since the counters are not stored and are used one at a time, this is not multiplied by n.
4. The final result list which will contain q elements, resulting in $O(q)$ space.

So, the total space complexity is $O(n + q)$, since n and q space requirements dominate the $O(k)$ space needed for the counter for each word or query string.