893. Groups of Special-Equivalent Strings

Medium Array Hash Table String [Leetcode Link]

Problem Description

You're given an array of strings `words`, where each string is of the same length. Your goal is to understand how to determine the number of *special-equivalent* groups within this array.

A single *move* consists of swapping two characters within a string, but there's a catch: you can only swap either even indexed characters or odd indexed characters. Two strings `words[i]` and `words[j]` are considered *special-equivalent* if you can make such swaps to make one string transform into the other.

To elaborate further, consider you have a string `words[i] = "zzxy"` and another string `words[j] = "xyzz"`. You can swap the first and the third characters (both even indices) to form "xzzy", and then swap the second and the fourth characters (both odd indices) to form "xyzz". Therefore, the two strings are *special-equivalent*.

A group of special-equivalent strings is essentially a collection of strings from the `words` array that can all be transformed into one another through these special swaps. Each group needs to be as large as possible, meaning that there shouldn't be any string outside the group that would be special-equivalent to every string within the group.

The task is to calculate and return the number of these groups of special-equivalent strings in the provided `words` array.

Intuition

To determine if two strings are special-equivalent, we need to consider the characters at even and odd indices separately. Because we can swap even with even and odd with odd indices without affecting the other, the actual order of even and odd characters within the string doesn't matter for the purpose of determining if two strings are special-equivalent. As long as the frequency of characters at the even and odd positions matches, they can be considered special-equivalent.

Given this understanding, a practical solution is to represent each string in a standardized form where the even and odd indexed characters are sorted separately and concatenated. If two strings are special-equivalent, their standardized forms will be identical.

By creating such a standardized form for each string and adding them to a set (which inherently removes duplicates), we can simply count the unique elements in this set as each represents a unique group of special-equivalent strings.

The given Python code creates standardized forms by sorting the characters at the even and odd indices of every string and concatenating them. These forms are placed in a set to eliminate duplicates, and then the length of the set is returned, thus giving the number of unique special-equivalent groups.

Solution Approach

The implementation of this solution involves creating a unique representation for each string that reflects its *special-equivalent* potential. This is done in a very straightforward and elegant manner, leveraging Python's powerful list slicing and sorting features.

Here is a step-by-step explanation of what the solution code does:

1. Initialize an empty set `s`. Sets in Python are an unordered collection of unique elements. In this case, the set is used to store the unique representations of the special-equivalent classes.

2. Iterate over each word in the `words` array. For each word, two operations are performed:

   a. Slice the word into characters at even indices: `word[::2]`. This uses Python's slicing syntax where `word[start:end:step]` is used to take every `step`-th character between `start` and `end`. A `step` of 2 starting at index 0 gets all the even-indexed characters.

   b. Slice the word into characters at odd indices: `word[1::2]`. This time, the slicing starts from index 1 to get all the odd-indexed characters.

   c. Sort both the even and odd character lists individually. Sorting ensures that regardless of the original order of the characters in the word, if two words have the same characters in even and odd places, their sorted sequences will match after sorting.

3. Then, concatenate the sorted sequences of even and odd indexed characters to form a single string. This concatenated string serves as the standardized form for special-equivalence.

4. Add each standardized form to the set `s`. If the string is already present (meaning another word had the same standardized form and thus is special-equivalent), nothing happens due to the nature of sets. Otherwise, a new entry is created.

5. After all words have been processed, return the count of unique items in the set `s` with `len(s)`. This count directly represents the number of unique groups of special-equivalent strings because each unique entry in `s` corresponds to a different group.

Thus, the key algorithm used here is sorting, along with the characteristic property of sets to hold only unique elements. By applying these data structures, a complex problem of grouping special-equivalent strings is reduced to a simple operation of counting unique standardized representations.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Consider the array of strings `words`:

```
words = ["abcd", "cbad", "bacd", "dacb"]
```

Each string has the same length. We need to determine how many special-equivalent groups are there.

Let's apply the solution approach:

1. Initialize an empty set `s`.

2. Iterate over each word in the `words` array. The processing for each word goes like this:

   - "abcd"
     - Even indices characters: "ac" (at position 0 and 2)
     - Odd indices characters: "bd" (at position 1 and 3)
     - Sort each list and concatenate: "ac" + "bd" = "acbd"
     - Add "acbd" to the set `s`.
   - "cbad"
     - Even indices characters: "ca" (at position 0 and 2)
     - Odd indices characters: "bd" (at position 1 and 3)
     - Sort each list and concatenate: "ac" + "bd" = "acbd"
     - Add "acbd" to the set `s` (already present, so no change).
   - "bacd"
     - Even indices characters: "bc" (at position 0 and 2)
     - Odd indices characters: "ad" (at position 1 and 3)
     - Sort each list and concatenate: "bc" + "ad" = "bcad"
     - Add "bcad" to the set `s`.
   - "dacb"
     - Even indices characters: "db" (at position 0 and 2)
     - Odd indices characters: "ac" (at position 1 and 3)
     - Sort each list and concatenate: "bd" + "ac" = "bdac"
     - Add "bdac" to the set `s`.

3. After processing, the set `s` looks like this: `{"acbd", "bcad", "bdac"}`

4. Return the count of unique items in the set `s`. In this case, the count is 3.

Thus, the number of special-equivalent groups in the array `words` is 3.

Python Solution

```python
from typing import List

class Solution:
    def numSpecialEquivGroups(self, words: List[str]) -> int:
        # Create a set to store unique representations of words
        unique_representation = set()

        # Loop through each word in the words list
        for word in words:
            # Sort the characters in even positions, and join them to form a string
            even_chars_sorted = ''.join(sorted(word[::2]))
            # Sort the characters in odd positions, and join them to form a string
            odd_chars_sorted = ''.join(sorted(word[1::2]))
            # Concatenation of sorted even and odd characters gives the word's representation
            representation = even_chars_sorted + odd_chars_sorted
            # Add the unique representation to the set
            unique_representation.add(representation)

        # The number of special-equivalent groups is the number of unique representations
        return len(unique_representation)

```

Java Solution

```java
class Solution {
    // Function to count the number of special equivalent groups in the array of words
    public int numSpecialEquivGroups(String[] words) {
        // Use a set to store unique transformed words
        Set<String> uniqueTransformedWords = new HashSet<>();
        for (String word : words) {
            // Add the transformed word to the set
            uniqueTransformedWords.add(transform(word));
        }
        // The size of the set represents the number of unique special equivalent groups
        return uniqueTransformedWords.size();
    }

    // Helper function to transform a word into its special equivalent form
    private String transform(String word) {
        // Use two lists to separate characters by their index parity
        List<Character> evenChars = new ArrayList<>();
        List<Character> oddChars = new ArrayList<>();

        // Iterate over the characters of the word
        for (int i = 0; i < word.length(); i++) {
            char ch = word.charAt(i);
            // If index is even, add to evenChars, else add to oddChars
            if (i % 2 == 0) {
                evenChars.add(ch);
            } else {
                oddChars.add(ch);
            }
        }

        // Sort both lists to normalize the order of chars
        Collections.sort(evenChars);
        Collections.sort(oddChars);

        StringBuilder transformedString = new StringBuilder();

        // Append the sorted characters to the string builder
        for (char c : evenChars) {
            transformedString.append(c);
        }
        for (char c : oddChars) {
            transformedString.append(c);
        }

        // Return the string representation of the transformed string
        return transformedString.toString();
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to count the number of special-equivalent groups
    int numSpecialEquivGroups(std::vector<std::string>& words) {
        // Set to store unique representations of special-equivalent words
        std::unordered_set<std::string> uniqueGroups;

        // Iterate through each word in the input vector
        for (auto& word : words) {
            std::string oddChars = "";
            std::string evenChars = "";

            // Divide characters of the word into odd and even indexed sub-strings
            for (int i = 0; i < word.size(); ++i) {
                if (i & 1) {
                    // If the index is odd, add to the oddChars string
                    oddChars += word[i];
                } else {
                    // If the index is even, add to the evenChars string
                    evenChars += word[i];
                }
            }

            // Sort the characters within the odd and even indexed sub-strings
            std::sort(oddChars.begin(), oddChars.end());
            std::sort(evenChars.begin(), evenChars.end());

            // Concatenate the sorted odd and even indexed strings and insert into set
            // This serves as the unique representation for this special-equivalent word
            uniqueGroups.insert(evenChars + oddChars);
        }

        // The number of unique groups is the size of our set
        return uniqueGroups.size();
    }
};

```

Typescript Solution

```typescript
// Importing necessary utilities from 'lodash' library
import _ from 'lodash';

// Function to count the number of special-equivalent groups
function numSpecialEquivGroups(words: string[]): number {
    // Set to store unique representations of special-equivalent words
    const uniqueGroups: Set<string> = new Set();

    // Iterate through each word in the input array
    words.forEach((word) => {
        let oddChars: string = "";
        let evenChars: string = "";

        // Divide characters of the word into odd and even indexed substrings
        for (let i = 0; i < word.length; i++) {
            if (i % 2 === 0) {
                // If the index is even, add to the evenChars string
                evenChars += word[i];
            } else {
                // If the index is odd, add to the oddChars string
                oddChars += word[i];
            }
        }

        // Sort the characters within the odd and even indexed substrings
        const sortedOddChars = _.sortBy(oddChars.split('')).join('');
        const sortedEvenChars = _.sortBy(evenChars.split('')).join('');

        // Concatenate the sorted odd and even indexed strings and insert into the set
        // This serves as the unique representation for this special-equivalent word
        uniqueGroups.add(sortedEvenChars + sortedOddChars);
    });

    // The number of unique groups is the size of our set
    return uniqueGroups.size;
}

// Example use
const exampleWords = ["abc","acb","bac","bca","cab","cba"];
const specialGroupsCount = numSpecialEquivGroups(exampleWords);
console.log(specialGroupsCount);  // Outputs: 3

```

Time and Space Complexity

The given Python code snippet defines a function `numSpecialEquivGroups` which calculates the number of special equivalent groups within a list of strings. To understand the computational complexity, let's analyze the time complexity and space complexity separately.

Time Complexity:

The time complexity of the function is determined by several operations within the set comprehension:

- Two `sorted` calls for each word: `sorted(word[::2])` and `sorted(word[1::2])`
- Concatenation of the sorted results
- Insertion into the set `s`

The sorting operations have a complexity of `O(m log m)` each, where `m` is the maximum length of a word divided by 2 (since we are only sorting half of the characters each time). The concatenation can be considered `O(m)`. These operations are performed for each of the `n` words in the list.

As set insertion in Python has an average case of `O(1)` complexity, mainly limited by the hashing function of the strings, the overall time complexity is governed by the sorting and concatenation steps done `n` times. Therefore, the total time complexity is approximately `O(n * m log m)`.

Space Complexity:

The space complexity is determined by:

- The space used by the temporary variables which hold the sorted substrings.
- The set `s` which, in the worst case, could hold `n` different strings.

The temporary variables used for storing sorted substrings do not significantly impact the overall space complexity since they are only transient storage used during the processing of each word.

Considering that each word can produce a string of maximum length `2m` after concatenation of the sorted halves, and the set `s` can contain at most `n` such strings, the space complexity can be considered as `O(n * m)`.

Conclusion:

The time complexity of the function is `O(n * m log m)` and the space complexity is `O(n * m)`, where `n` is the number of words in the input list and `m` is the maximum length of a word divided by 2.