2452. Words Within Two Edits of Dictionary

Medium Array String

Leetcode Link

Problem Description

In this problem, we are given two arrays of strings: `queries` and `dictionary`. Each string consists only of lowercase English letters, and every string in both arrays is of the same length.

We need to perform *edits* on the words in `queries`. An *edit* is defined as changing any single letter in a word to any other letter. Our objective is to determine which words from `queries` can be transformed into any word from `dictionary` using at most two such edits.

The output should be a list of words from `queries` that can be matched with any word in `dictionary` after performing no more than two edits. The returned list must preserve the original order of words as they appear in `queries`.

Intuition

The intuition behind the solution is to iterate over each word in `queries` and each word in `dictionary`, comparing them letter by letter. For each pair of words, we count the number of letters that are different between them. This count is the number of edits needed to transform the query word into the dictionary word.

Since we are allowed up to two edits, we look for pairs of words where the number of differing letters is less than or equal to two. If such a pair is found, the query word qualifies as a match, and we add it to our answer list.

We proceed with this process until we have checked the word from `queries` against all words in `dictionary`, ensuring we do not exceed two edits. The crucial observation is that any query word can be transformed into any dictionary word by changing at most two letters - signifying that the edit distance between them is less than or equal to two.

This approach ensures that we examine all possible pairs of words, and when a match is found, we immediately move to the next word in `queries` to maintain efficiency. We exit the inner loop early using `break` once we append the query word to the answer list, as we do not need any more comparisons for that word.

Solution Approach

The given Python code defines a class `Solution` with a method `twoEditWords`, which takes two parameters: `queries` and `dictionary`. These parameters are lists of strings representing the words we will be working with. The method returns a list of strings.

```
class Solution:
    def twoEditWords(self, queries: List[str], dictionary: List[str]) -> List[str]:
        ans = []
        for s in queries:
            for t in dictionary:
                if sum(a != b for a, b in zip(s, t)) < 3:
                    ans.append(s)
                    break
        return ans
```

The algorithm proceeds with the following steps:

1. Initialize an empty list `ans` to store the final matching words from `queries`.
2. Iterate over each word `s` from `queries`.
3. For each word `s`, iterate over each word `t` from `dictionary`.
4. Zip the two words `s` and `t` to compare their corresponding letters. The built-in `zip` function pairs up elements from two iterables, allowing us to iterate over them in parallel.
5. Use a generator expression inside the `sum` function to count the number of differing letters between `s` and `t`. We compare each pair of letters and count a difference whenever a pair does not match (`a != b`).
6. If the count is less than 3 (meaning we can turn `s` into `t` with at most two edits), we append `s` to our answer list. This is because we are only allowed a maximum of two edits.
7. Once a word from `queries` matches a word from `dictionary`, break out of the inner loop to avoid unnecessary comparisons and move to the next word in `queries`.
8. After going through all the words in `queries`, return the answer list `ans`.

This solution uses a brute-force approach and takes advantage of Python's concise syntax for list comprehension and the `sum` function. This approach works efficiently when the dataset is not prohibitively large since it explores all possible pairs of words from `queries` and `dictionary` and calculates the edit distance in a straightforward manner.

However, it's worth noting that this algorithm has a quadratic time complexity with respect to the number of words in `queries` and `dictionary`. Therefore, for very large datasets, the performance might be a concern, and more sophisticated algorithms could be required to optimize the search and comparison processes.

Example Walkthrough

Let's take a small example to illustrate the solution approach using the Python code provided.

Suppose we have the following arrays:

- `queries` = ["abc", "def", "ghi"]
- `dictionary` = ["abd", "def", "hij"]

The question tells us that we can make at most two edits on each word in `queries` to see if it can match any word in `dictionary`. Let's walk through each word:

1. Starting with the first word in `queries`, which is "abc":
    - When we compare "abc" to "abd" in `dictionary`, we notice that they differ by one character (`c != d`). Since only one edit is needed, "abc" can be transformed into "abd" with a single change. Therefore, "abc" satisfies the condition of two or fewer edits and is appended to our answer list.
    - We do not need to compare "abc" further with other words in `dictionary` because we found a match. We move to the next word in `queries`.
2. Now, look at the second word in `queries`, "def":
    - Comparing "def" with "abd" from `dictionary`, we find three different characters, meaning we need more than two edits, so we move to the next word in `dictionary`.
    - The next comparison is between "def" from `queries` and "def" from `dictionary`. They are identical, so zero edits are required, and "def" is added to our answer list. Then we exit the inner loop and proceed to the next query.
3. Finally, for the third word in `queries`, "ghi":
    - We compare "ghi" with "abd" from `dictionary`, and all three characters are different, requiring more than two edits. So we continue to the next word in `dictionary`.
    - We compare "ghi" with "def" from `dictionary`, but again all characters differ, so we move on to the last word.
    - The last comparison is between "ghi" from `queries` and "hij" from `dictionary`. Here we see that only one character differs (`g != h`), so we can make one edit to match them. "ghi" is then added to our answer list, and we finish reviewing `queries`.

The final returned list (ans) from the `twoEditWords` method is ["abc", "def", "ghi"], preserving the original order from `queries`. Each of these words can be transformed into a word in `dictionary` with no more than two edits.

Using this example, we can see how the solution approach successfully iterates over the `queries` and `dictionary`, compares the words, and keeps track of the number of edits necessary to determine if a word from `queries` can be turned into any word in `dictionary` within the allowed edits.

Python Solution

```
class Solution:
    def twoEditWords(self, queries: List[str], dictionary: List[str]) -> List[str]:
        # Initialize a list to hold the answer.
        result = []

        # Iterate through each word in the queries list.
        for query_word in queries:
            # Now, compare with each word in the dictionary.
            for dictionary_word in dictionary:
                # If the count of differing characters is less than 3,
                # it means they are within two edits of each other.
                if sum(1 for q_char, d_char in zip(query_word, dictionary_word) if q_char != d_char) < 3:
                    # We've found a word in the dictionary that is within two edits.
                    # Add the query word to the result list.
                    result.append(query_word)
                    # No need to check the rest of the dictionary for this word.
                    break

        # Return the list of words that are within two edits of any word in the dictionary.
        return result
```

Java Solution

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

class Solution {
    public List<String> twoEditWords(String[] queries, String[] dictionary) {
        // Initialize an ArrayList to store the result.
        List<String> result = new ArrayList<>();

        // Assume all strings in queries have the same length as the first string's length.
        int queryLength = queries[0].length();

        // Iterate through each string in queries.
        for (String query : queries) {
            // Compare each query with each string in the dictionary.
            for (String word : dictionary) {
                // Initialize a count to track the number of differing characters.
                int differenceCount = 0;

                // Check for character differences between the query and the dictionary word.
                for (int i = 0; i < queryLength; ++i) {
                    if (query.charAt(i) != word.charAt(i)) {
                        differenceCount++;
                    }
                }

                // If there are fewer than 3 differences, add the query to the result list.
                if (differenceCount < 3) {
                    result.add(query);
                    // Since we found a word in the dictionary that is within
                    // two edits of the query, we break out of the dictionary loop.
                    break;
                }
            }
        }

        // Return the list of queries that are within two edits of some dictionary word.
        return result;
    }
}
```

C++ Solution

```
#include <vector>
#include <string>

using std::vector;
using std::string;

class Solution {
public:
    // Function to return a vector of strings from queries that are at most two edits
    // away from any string in the dictionary.
    vector<string> twoEditWords(vector<string>& queries, vector<string>& dictionary) {
        vector<string> results;                     // To hold the result strings
        for (auto& query : queries) {               // Iterate through each query
            for (auto& word : dictionary) {         // Iterate through each word in the dictionary
                // Ensure the length difference is not greater than 2.
                if (std::max(query.size(), word.size()) - std::min(query.size(), word.size()) > 2) {
                    continue;
                }

                int count = 0;                      // Counter to track differences
                for (size_t i = 0; i < query.size() && i < word.size(); ++i) {
                    if (query[i] != word[i]) {
                        ++count;                    // Increment count if characters differ
                    }
                }

                // Add the length difference for any additional characters
                count += std::abs(static_cast<int>(query.size()) - static_cast<int>(word.size()));

                if (count < 3) {                    // If less than three edits
                    results.emplace_back(query);    // Add query to result
                    break;                          // Break since only one match is needed
                }
            }
        }

        return results;                             // Return the resulting vector
    }
};
```

Typescript Solution

```
// Check if each query string is at most two edits away from any string in the dictionary.
function twoEditWords(queries: string[], dictionary: string[]): string[] {
    // Assuming all queries are of the same length as stated by the first query's length.
    const queryLength = queries[0].length;

    // Filter and return only those queries that are within two edits of any dictionary word.
    return queries.filter(query => {
        // Iterate over each word in the dictionary to compare with the query.
        for (const word of dictionary) {
            let differences = 0; // Counter for character differences between query and dictionary word

            // Compare each character of the query with the dictionary word.
            for (let i = 0; i < queryLength; i++) {
                // Increment differences if characters do not match.
                if (query[i] !== word[i]) {
                    differences++;
                    // If differences exceed 2, break out of the loop as it is no longer a valid match.
                    if (differences > 2) {
                        break;
                    }
                }
            }

            // If the word in the dictionary is at most two edits away from the query, it's a match.
            if (differences <= 2) {
                return true;
            }
        }

        // If no words in the dictionary are within two edits of the query, filter it out.
        return false;
    });
}
```

Time and Space Complexity

Time Complexity

The given code has two nested loops; the outer loop iterates through each string in `queries`, while the inner loop iterates through each string in `dictionary`. Within the inner loop, there's a comparison of corresponding characters in two strings (from `queries` and `dictionary`) made using a generator expression with a `zip` function and `sum`. This comparison runs in O(min(len(s), len(t))) time, where `len(s)` and `len(t)` represent the lengths of the individual strings being compared.

To determine the time complexity of the entire code, we have to consider the lengths of the `queries` and `dictionary` lists and the average lengths of the strings within them. Let `n` denote the number of strings in `queries`, `m` denote the number of strings in `dictionary`, and `L` be the average length of the strings in both lists. The total time complexity is thus:

```
O(n * m * L)
```

For every string in `queries`, every string in `dictionary` is checked, and for each comparison, an iteration over the length of the strings occurs.

Space Complexity

The space complexity of the code consists of the space needed to store the `ans` list and the temporary space for the comparison operation. The `ans` list, in the worst case, will store all strings from `queries`. Therefore, its space complexity depends on the length of the output list, which is at most `n` (where `n` is the number of strings in `queries`).

The generator expression with `zip` and `sum` does not create a list of differences but rather creates an iterator, which takes constant space.

Hence, the space complexity is:

```
O(n)
```

This represents the space needed to store the `ans` list. The rest of the operations use constant additional space (ignoring the overhead of the input and the internal working storage of Python's function calls).