1554. Strings Differ by One Character

Medium Hash Table String Hash Function Rolling Hash

Leetcode Link

Problem Description

In this problem, you are given a list of strings called `dict`, where each string is of the same length. Your task is to determine whether there are at least two strings in the list that differ from each other by exactly one character, and this difference must be at the same position in both strings. If such a pair of strings exists, you should return `true`. If no such pairs exist, you should return `false`.

Intuition

The core idea behind the solution is to use hashing to efficiently check for the string pair with the one-character difference. To compare strings while allowing for one difference, we mask each position in the strings one at a time and see if this version of strings has been seen before.

Here's the sequence of thoughts leading to the solution approach:

1. If we could somehow ignore one character in each string and then compare them, we could easily identify if only one character was different.
2. Hashing is efficient for quickly searching a collection of items.
3. We can iterate over each string and temporarily replace each character one by one with a placeholder (in this case, an asterisk `'*'`), creating a masked version of the string.
4. We then check if this masked string has already been encountered i.e., present in our hash set.
5. If we find the masked string in the set, it means there's another word in the list that could match this string with exactly one character difference.
6. If not, we add this masked string to the set and continue the process with the next string.
7. Finally, if we never find a matching masked string, it means there are no such pairs, and we return `false`.

This approach is clever because it avoids the need to compare every string with every other string directly, which would be time-consuming, especially as the list size grows.

Solution Approach

The Python code provided for the solution utilizes a set data structure to keep track of all the masked versions of the strings encountered so far. A set is chosen for its efficient `O(1)` average time complexity for adding elements and checking for membership.

Let's go through the algorithm step by step:

1. Initialize an empty set `s` to hold the masked versions of the strings.
2. Iterate through each `word` in the given `dict`.
3. For each `word`, iterate over the length of the word using a range loop to get each index `i`.
4. In each iteration, construct a new string `t` by concatenating:
   - The substring of `word` from the beginning up to but not including `i` (denoted by `word[:i]`).
   - A placeholder asterisk `'*'` which acts as a mask for the character at position `i`.
   - The substring of `word` from just after `i` to the end (denoted by `word[i + 1 :]`).
5. Check if the new masked string `t` is already in the set `s`.
   - If `t` is already in the set, this means there is another string in the list which differs from the current string by exactly one character at the same index. Thus, we return `True`.
6. If `t` is not in the set, add this new masked version of the string to the set `s` for future comparisons.
7. Continue this process until all words have been processed or until a match is found.
8. If no match is found after processing all words, return `False`.

Using this approach, the time complexity of comparing the strings becomes `O(N*M)` where `N` is the number of strings and `M` is the length of each string, because for each word we iterate over its length once and each operation inside the loop is `O(1)` due to the nature of set operations.

Example of how the masking works with an input list `["abcd", "accd", "bccd"]`:

- For "abcd", we'll add to the set: `"*bcd"`, `"a*cd"`, `"ab*d"`, `"abc*"`
- For "accd", we'll add and compare: `"*ccd"`, `"a*cd"`. As `"a*cd"` is already in the set, we detect that "accd" differs from "abcd" by one character ('b' vs 'c'), thus we can return `True`.

By applying the masking technique, we save time by not having to compare each string with every other string in a brute-force manner.

Example Walkthrough

Let's take an example list of strings `["pine", "sine", "ping", "cling", "singe"]`, where we want to determine if there's at least one pair of strings that only differ by one character at the same position.

Following the solution approach step by step:

1. Initialize an empty set: `s = {}`
2. We process the first word `"pine"`.
   - Create masks: *"ine"*, *"pne"*, *"pie"*, *"pin"*
   - Add these to the set: `s = {"*ine", "*pne", "pi*e", "pin*"}`
3. Now, process the second word `"sine"`.
   - Create masks: *"ine"*, *"sne"*, *"sie"*, *"sin"*
   - Checking these against the set:
     - `"*ine"` is found in the set (matching "pine" masked as `"*ine"`), meaning "pine" and "sine" differ by one character.
     - Since we found a match, we return `True`.
     - There is no need to process further as we've found at least one pair of strings meeting the criteria.

If we had not found a match for "sine", we would then add its masked versions to the set and continue with the next word.

In this example, we quickly identified a pair without having to compare every word to every other word, thus demonstrating the efficiency of the solution.

Python Solution

```python
class Solution:
    def differByOne(self, dict: List[str]) -> bool:
        # Initialize a set to store modified words
        seen = set()

        # Iterate over each word in the dictionary
        for word in dict:
            # Iterate over each character in the word
            for i in range(len(word)):
                # Create a new word by replacing the current character with a '*'
                temp_word = word[:i] + '*' + word[i + 1:]
                # Check if the modified word is already in the set (seen)
                if temp_word in seen:
                    # If found, return True since two words differ by exactly one character
                    return True
                # Otherwise, add the modified word to the set
                seen.add(temp_word)

        # Return False if no such pair of words is found in the dictionary
        return False

```

Java Solution

```java
class Solution {

    /**
     * Checks if there are two strings in the provided array that differ by exactly one character.
     *
     * @param dict An array of strings.
     * @return true if there are two strings differing by one character, otherwise false.
     */
    public boolean differByOne(String[] dict) {
        // Create a HashSet to store unique patterns of the words
        Set<String> patterns = new HashSet<>();

        // Iterate over each word in the dictionary
        for (String word : dict) {
            // Replace each character one by one with '*' to create patterns
            for (int i = 0; i < word.length(); ++i) {
                // Generate a new pattern by replacing the character at index 'i' with '*'
                String pattern = word.substring(0, i) + "*" + word.substring(i + 1);

                // If the pattern already exists in the set, return true
                if (patterns.contains(pattern)) {
                    return true;
                }

                // Add the new pattern to the set
                patterns.add(pattern);
            }
        }

        // If no pattern has two matching strings, return false
        return false;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function checks if any two strings in the given dictionary differ by exactly one character
    bool differByOne(std::vector<std::string>& dict) {
        // Create an unordered set to keep track of unique patterns
        std::unordered_set<std::string> patterns;

        // Iterate through each word in the dictionary
        for (const auto& word : dict) {
            // Iterate through each character in the word
            for (size_t i = 0; i < word.size(); ++i) {
                // Make a copy of the word to create a pattern
                std::string pattern = word;
                // Replace the i-th character with a wildcard symbol '*'
                pattern[i] = '*';

                // Check if the pattern is already in the set
                if (patterns.count(pattern)) {
                    // If found, two words in the dict differ by one character
                    return true;
                }
                // If not found, insert the new pattern into the set
                patterns.insert(pattern);
            }
        }
        // If no such pair of words found, return false
        return false;
    }
};

```

Typescript Solution

```typescript
// A variable to keep track of unique patterns
const patterns: Set<string> = new Set<string>();

// Function checks if any two strings in the given array differ by exactly one character
function differByOne(dict: string[]): boolean {
    // Iterate through each word in the array
    for (const word of dict) {
        // Iterate through each character in the word
        for (let i = 0; i < word.length; ++i) {
            // Make a copy of the word to create a pattern
            let pattern = word.substring(0, i) + '*' + word.substring(i + 1);

            // Check if the pattern is already in the set
            if (patterns.has(pattern)) {
                // If found, two strings in the array differ by one character
                return true;
            }
            // If not found, insert the new pattern into the set
            patterns.add(pattern);
        }
    }
    // If no such pair of strings is found, return false
    return false;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the two nested loops: the outer loop iterates over each word in the dictionary, and the inner loop iterates over each character in a word to create a new string pattern with a wildcard character `'*'`. This new string pattern has the same length as the original word, but with one of the characters replaced.

If `n` is the number of words in the dictionary and `m` is the average length of a word, then the outer loop runs `n` times, and the inner loop runs `m` times for each word. Therefore, the total number of iterations is `n * m`. Inside the inner loop, there's a string concatenation operation which takes `O(m)` time since it involves creating a new string of length `m`. Then it checks the presence of this pattern in the set and possibly adds it to the set. Both of these set operations take `O(1)` time on average.

Combining these factors, the overall time complexity is `O(n * m^2)`, where `n` is the number of words and `m` is the length of each word.

Space Complexity

The space complexity is mainly due to the set `s` that stores all unique word patterns with the wildcard. In the worst case, we store `n * m` different patterns since each word can lead to `m` different patterns. Since each pattern is of length `m`, they can be thought to occupy `m` space each.

Therefore, the space complexity is `O(n * m^2)`, as we need to store `n * m` patterns, each of length `m`.