2135. Count Words Obtained After Adding a Letter

Medium | Bit Manipulation | Array | Hash Table | String | Sorting | Leetcode Link

Problem Description

You are provided with two lists of strings: `startWords` and `targetWords`. Each string is composed of lowercase English letters. Your task is to determine how many strings in `targetWords` can be formed from any string in `startWords` by performing a specific conversion operation.

The conversion operation consists of two steps:

1. Append a single lowercase letter that is not already in the string to its end. For example, if you have the string "abc", you can add "d", "e", "y", but not "a" to it, creating strings like "abcd".
2. Rearrange the letters of the newly formed string in any order. For instance, the string "abcd" could be rearranged into "acbd", "bacd", "cbda", etc.

You need to count the number of strings in `targetWords` that can be achieved by applying this operation on any of the strings in `startWords`.

It's important to note that `startWords` aren't actually altered during this process; the operation is only used to verify the possibility of transforming a `startWord` into a `targetWord`.

Intuition

The solution is based on a clever use of bitwise operations to track the presence of each letter in a given string. We use an integer (bitmask) to represent each string, where the ith bit (from right to left) is set to 1 if the letter 'a' + i is in the string. For instance, the string "abc" would result in the bitmask `0b111` (in binary), representing the presence of 'a', 'b', and 'c'.

By using this approach, we can easily check whether we can form a `targetWord` from any `startWord` by appending a letter and rearranging it. For each `targetWord`, we can generate its bitmask and then, for each letter in that `targetWord`, we toggle the corresponding bit (using XOR operation) to simulate the removal of the letter. We then check if the resulting bitmask is present in the set of bitmasks created from the `startWords`. If it's found, we know that we can form the `targetWord` from that `startWord`.

Let's walk through the steps of the solution:

1. Create an integer set `s`. For each word in `startWords`, calculate a bitmask representing the letters in the word and add it to `s`.
2. Initialize a counter `ans` to zero, which will keep track of the number of `targetWords` that can be formed.
3. For every word in `targetWords`, compute the bitmask in a similar way.
4. For each letter in the current `targetWord`, create a temporary bitmask by toggling (using XOR operation) the bit corresponding to the current letter. If the resulting bitmask is found in `s`, increment `ans` by one, and proceed to check the next `targetWord`.
5. Return the final count `ans`.

By representing strings as bitmasks, we switch from an O(N*26) character comparison problem to an O(N) integer comparison problem, which is much more efficient.

Solution Approach

The solution implements a bit manipulation strategy using Python's bitwise operations to encode each word as a bitmask, where each bit represents the presence of a corresponding letter from the alphabet.

Let's detail the steps followed in the given Python code:

1. **Bitmask Creation for `startWords`:**
   - Initialize an empty set `s` which will hold the unique bitmasks of all the words in `startWords`.
   - Iterate over each word in `startWords`, and for each word:
     - Initialize a variable `mask` with a value of 0. This `mask` will be the bitmask representation of the word.
     - For each character `c` in the word, calculate the difference `ord(c) - ord('a')` to find the position of the bit that corresponds to the character in the alphabet (0 for 'a', 1 for 'b', and so on).
     - Use the bitwise OR `mask |= 1 << (ord(c) - ord('a'))` operation to set the bit at the calculated position. This ensures that each bit in the `mask` represents whether a particular character is in the word.
   - Add the resulting `mask` to the set `s`.
2. **Verification for `targetWords`:**
   - Initialize the `ans` counter to 0, which will count the valid `targetWords`.
   - Repeat the similar process for each word in `targetWords` to create the bitmask.
   - For each character `c` in the current `targetWord`, generate a temporary bitmask `t` by toggling the bit corresponding to `c` in the word's bitmask. Toggling is done using the XOR operation `mask ^ (1 << (ord(c) - ord('a')))`, which effectively simulates removing the character `c` from the word.
   - If the temporary bitmask `t` is found in the set `s`, increment the `ans` because this indicates that the original `targetWord` can be formed by adding `c` to some `startWord` and rearranging the letters.
   - If a match is found in the set `s` for the current `targetWord`, break the inner loop to prevent counting the same `targetWord` multiple times, and continue with the next `targetWord`.
3. **Return the Result:**
   - After iterating over all the words in `targetWords`, return the final count `ans`.

The algorithm leverages bitwise operations for efficient comparison and the set data structure for constant-time lookup to check the possibility of formation of `targetWords`. The key patterns used here are bitwise encoding and set membership checks, which make the solution compact and efficient.

Example Walkthrough

To elaborate on the solution approach, let's walk through a small example:

Let's say we have:

- `startWords = ["go","bat"]`
- `targetWords = ["bago","atb","tabg"]`

Now, we will apply the solution approach on this example.

Bitmask Creation for `startWords`:

1. Initialize an empty set `s`. This will store the bitmasks of `startWords`.
2. For "`go`":
   - Start with bitmask `mask = 0`.
   - We add 'g' (bitmask `1 << (ord('g') - ord('a')) = 1 << 6`).
   - We add 'o' (bitmask `1 << (ord('o') - ord('a')) = 1 << 14`).
   - Final bitmask for "go" is `0 | 1 << 6 | 1 << 14`, which is `0b1000001000000`.
   - Add this bitmask to set `s`.
3. For "`bat`":
   - Start with bitmask `mask = 0`.
   - We add 'b' (bitmask `1 << (ord('b') - ord('a')) = 1 << 1`).
   - We add 'a' (bitmask `1 << (ord('a') - ord('a')) = 1 << 0`).
   - We add 't' (bitmask `1 << (ord('t') - ord('a')) = 1 << 19`).
   - Final bitmask for "bat" is `0 | 1 << 1 | 1 << 0 | 1 << 19`, which is `0b1000000000011`.
   - Add this bitmask to set `s`.

After processing `startWords`, our set `s` has bitmasks `{0b1000001000000, 0b1000000000011}`.

Verification for `targetWords`:

1. Initialize `ans` to 0.
2. Process `targetWords` similarly and compare with bitmasks in `s`.
3. For "`bago`":
   - Bitmask for "bago" is `0b1010001000010`.
   - Check each letter by toggling it in the bitmask:
     - Toggle 'b': We get mask `0b1010001000010 ^ 1 << 1 = 0b1010001000000` which is **not** in `s`.
     - Toggle 'a': We get mask `0b1010001000010 ^ 1 << 0 = 0b1010001000011` which is **in** `s`.
     - No need to check further, increment `ans` to `1`.
4. For "`atb`":
   - Bitmask for "atb" directly matches the bitmask of "bat" in `s`.
   - Since one letter has to be added, this shouldn't have happened; "atb" is the same length as "bat", thus, we skip and don't increment `ans`.
5. For "`tabg`":
   - Bitmask for "tabg" is `0b1000001000011`.
   - Check each letter by toggling it in the bitmask:
     - Toggle 't': We get mask `0b1000001000011 ^ 1 << 19 = 0b0000001000011`, which matches "ab" (not in `s`, and too short anyway).
     - Toggle 'a': We get mask `0b1000001000011 ^ 1 << 0 = 0b1000001000010`, which is **in** `s`.
     - No need to check further, increment `ans` to `2`.

Return the Result:

After examining all `targetWords`, the final `ans` is `2`, since the words "`bago`" and "`tabg`" in `targetWords` can be formed from the `startWords` "`go`" and "`bat`", respectively.

The key insights from this example that lead to the count of valid `targetWords` are the efficient bitmask representation of the words and constant-time set membership checks to verify the transformation possibilities.

Python Solution

```python
from typing import List

class Solution:
    def wordCount(self, start_words: List[str], target_words: List[str]) -> int:
        bit_masks = set()  # Set to store the unique bit masks for start words.

        # Preprocessing start words by converting them to bit masks
        for word in start_words:
            mask = 0
            for char in word:
                # Set the bit corresponding to the character in the mask
                mask |= 1 << (ord(char) - ord('a'))
            bit_masks.add(mask)  # Add the mask to the set

        valid_target_count = 0  # Counter for valid target words

        # Check if each target word can be formed by adding a single letter to a word in start_words
        for word in target_words:
            mask = 0
            for char in word:
                # Set the bit corresponding to the character in the mask
                mask |= 1 << (ord(char) - ord('a'))
            # Now try removing one character at a time and check if it matches a start word
            for char in word:
                # Toggle off the bit corresponding to the character to remove it
                temp_mask = mask ^ (1 << (ord(char) - ord('a')))
                # If the resulting mask is in the set, increment the valid target count.
                if temp_mask in bit_masks:
                    valid_target_count += 1
                    break  # Break, as we only need to find one such start word

        return valid_target_count  # Return the total count of valid target words
```

Java Solution

```java
class Solution {
    public int wordCount(String[] startWords, String[] targetWords) {
        // Initialize a set to store the bit masks of startWords
        Set<Integer> startWordMasks = new HashSet<>();

        // Convert each start word into a bit mask and add to the set
        for (String word : startWords) {
            int bitmask = 0;
            // For each character in the word, set the corresponding bit in the bitmask
            for (char ch : word.toCharArray()) {
                bitmask |= (1 << (ch - 'a'));
            }
            // Add the bitmask to the set
            startWordMasks.add(bitmask);
        }

        // Counter for the number of valid target words
        int validTargetCount = 0;

        // Check each target word against the bit masks of startWords
        for (String word : targetWords) {
            int targetBitmask = 0;
            // Calculate the bit mask for the target word
            for (char ch : word.toCharArray()) {
                targetBitmask |= (1 << (ch - 'a'));
            }
            // Try to find a start word that matches the target word by removing one letter
            for (char ch : word.toCharArray()) {
                // Create a new mask by flipping the bit corresponding to the current character
                int modifiedMask = targetBitmask ^ (1 << (ch - 'a'));
                // If the modified mask exists in the start words, we found a valid target
                if (startWordMasks.contains(modifiedMask)) {
                    validTargetCount++;
                    break; // We found a matching start word; move to the next target word
                }
            }
        }
        // Return the count of valid target words
        return validTargetCount;
    }
}
```

C++ Solution

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

class Solution {
public:
    int wordCount(vector<string>& startWords, vector<string>& targetWords) {
        // A set to store the unique representation of each startWord.
        unordered_set<int> wordMasks;

        // Convert each startWord into a bitmask representing the letters it contains.
        for (const auto& word : startWords) {
            int mask = 0;
            for (char c : word)
                mask |= (1 << (c - 'a')); // Set the bit corresponding to the letter.
            wordMasks.insert(mask); // Add the bitmask to the set.
        }

        int count = 0; // Counter for eligible target words.

        // Iterate over each target word.
        for (const auto& word : targetWords) {
            int mask = 0;
            for (char c : word)
                mask |= (1 << (c - 'a')); // Create a bitmask for the target word.

            // Try removing one letter from the target word to match a start word.
            for (char c : word) {
                int modifiedMask = mask ^ (1 << (c - 'a')); // Remove one letter from the target word.
                if (wordMasks.count(modifiedMask)) {
                    // If the modified target word's bitmask matches any start word's bitmask,
                    // increment the count and stop checking further letters.
                    ++count;
                    break;
                }
            }
        }

        return count;
    }
};
```

Typescript Solution

```typescript
// Define the function that counts the eligible target words.
function wordCount(startWords: string[], targetWords: string[]): number {
    // A set to store the unique representation of each start word.
    const wordMasks = new Set<number>();

    // Convert each start word into a bitmask representing the letters it contains.
    startWords.forEach(startWord => {
        let mask = 0;
        for (const char of startWord) {
            mask |= (1 << (char.charCodeAt(0) - 'a'.charCodeAt(0))); // Set the bit corresponding to the letter.
        }
        wordMasks.add(mask); // Add the bitmask to the set.
    });

    // Counter for eligible target words.
    let count = 0;

    // Iterate over each target word.
    targetWords.forEach(targetWord => {
        let mask = 0;
        for (const char of targetWord) {
            mask |= (1 << (char.charCodeAt(0) - 'a'.charCodeAt(0))); // Create a bitmask for the target word.
        }

        // Try removing one letter from the target word to match a start word.
        for (const char of targetWord) {
            const modifiedMask = mask ^ (1 << (char.charCodeAt(0) - 'a'.charCodeAt(0))); // Remove one letter from the target word.
            if (wordMasks.has(modifiedMask)) {
                // If the modified target word's bitmask matches any start word's bitmask,
                // increment the count and stop checking further letters.
                count++;
                break;
            }
        }
    });

    // Return the count of eligible target words.
    return count;
}
```

Time and Space Complexity

The time complexity of the given code is `O(L * (N + M))`, where `L` is the average length of the words across both `startWords` and `targetWords`, `N` is the number of `startWords`, and `M` is the number of `targetWords`. This complexity arises because the code iterates over each character of every word in `startWords` to create bit masks (`L * N` operations), and then for each word in `targetWords`, it does the same to get a bitmask and tries to find if there is any word in `startWords` that can be turned into this target word by adding one letter (`L * M` operations).

The space complexity is `O(N)` because we store bit masks of all `startWords` in a set `s`. If `k` is the size of the alphabet and all words are of length `L`, the actual bit masks would take up `L` bits each, but since a set of integers is used here and the letters are mapped to bit indices, it only matters how many different bit masks there are, which corresponds to the number of `startWords` (or `N`).