09.01.2025

Problem: Counting Words with a Given Prefix

Problem Statement:
You are given an array of strings `words` and a string `pref`. Return the number of strings in `words` that contain `pref` as a prefix.

Example 1:

```
Input: words = ["pay","attention","practice","attend"], pref = "at"
Output: 2
Explanation: "attention" and "attend" start with "at".
```

Example 2:

```
Input: words = ["leetcode","win","loops","success"], pref = "code"
Output: 0
Explanation: No word starts with "code".
```

Approach 1: Using Trie (Efficient for multiple queries)

A **Trie (Prefix Tree)** is useful when you need to perform multiple prefix-based searches efficiently.

Steps:

1. **Create a TrieNode structure.**
2. **Implement insertion:** Each word is inserted letter by letter.
3. **Implement search for a prefix:** Search if the prefix exists and count how many words have that prefix.
4. **Count words with the prefix by traversing the Trie.**

C++ Code (Using Trie)

```cpp
struct TrieNode {
    TrieNode* children[26];
    int count;
    bool endofword;
};
```

```cpp
// Function to create a new TrieNode
TrieNode* getNode() {
    TrieNode* newNode = new TrieNode();
    newNode->endofword = false;
    newNode->count = 0;
    for (int i = 0; i < 26; i++) {
        newNode->children[i] = nullptr;
    }
    return newNode;
}

class Trie {
public:
    TrieNode* root;

    Trie() {
        root = getNode();
    }

    // Insert a word into the Trie
    void insertNode(const string& word) {
        TrieNode* ptr = root;
        for (char ch : word) {
            int index = ch - 'a';
            if (ptr->children[index] == nullptr) {
                ptr->children[index] = getNode();
            }
            ptr = ptr->children[index];
            ptr->count++;
        }
        ptr->endofword = true;
    }

    // Search for the prefix and return the count of words with that prefix
    int searchAndPrefixCount(const string& prefix) {
        TrieNode* ptr = root;
        for (char ch : prefix) {
            int index = ch - 'a';
            if (ptr->children[index] == nullptr) {
                return 0;
            }
            ptr = ptr->children[index];
        }
        return ptr->count;
    }
};

class Solution {
public:
    int prefixCount(vector<string>& words, string pref) {
        Trie trie;

        // Insert all words into the Trie
        for (const string& word : words) {
            trie.insertNode(word);
        }
```

```
        // Return the count of words with the given prefix
        return trie.searchAndPrefixCount(pref);
    }
};
```

Approach 2: Using Simple String Matching (Direct Search)

This approach checks if each word starts with the given prefix using the `substr()` function.

C++ Code:

```
class Solution {
public:
    int prefixCount(vector<string>& words, string pref) {
        int count = 0;
        for (const string& word : words) {
            // Check if the prefix matches the start of the word
            if (word.substr(0, pref.length()) == pref) {
                count++;
            }
        }
        return count;
    }
};
```

Explanation:

- **Input:** `words = ["pay","attention","practice","attend"]`, `pref = "at"`
- **Output:** `2` because `"attention"` and `"attend"` start with `"at"`.

Complexity:

Trie Approach:

- **Time Complexity:** `O(n * m + p)`
  a. `n` = number of words
  b. `m` = average length of the words
  c. `p` = length of the prefix
- **Space Complexity:** `O(n * m)`

String Matching Approach:

- **Time Complexity:** `O(n * p)`

- **Space Complexity:** `O(1)`

Edge Cases Handled:

- If `words` contains only one word.
- If `pref` is longer than any word in `words`.
- If no word starts with `pref`.

✅ Key Takeaways:

- Use a **Trie** when multiple prefix searches are required.
- Use **direct string matching** for a simpler approach when only one search is required.

Would you like me to optimize further or explain the Trie data structure more? 😊