08.01.2025

Problem: Count Prefix and Suffix Pairs I

Problem Statement:
Given a 0-indexed string array `words`, return the number of pairs `(i, j)` such that:

- `i < j` and
- `words[i]` is both a **prefix** and a **suffix** of `words[j]`.

Examples:

Example 1:

```
Input: words = ["a","aba","ababa","aa"]
Output: 4
Explanation:
- "a" is a prefix and suffix of "aba", "ababa", and "aa".
- "aba" is a prefix and suffix of "ababa".
Total pairs: 4
```

Example 2:

```
Input: words = ["pa","papa","ma","mama"]
Output: 2
Explanation:
- "pa" is a prefix and suffix of "papa".
- "ma" is a prefix and suffix of "mama".
```

Example 3:

```
Input: words = ["abab","ab"]
Output: 0
Explanation: No pairs satisfy the condition.
```

Approach: Using Trie Data Structure

The idea here is to use a **Trie** (Prefix Tree) to efficiently check if a word is a prefix and suffix.

Steps:

1. **Initialize Two Tries:**
   a. One for storing the **original strings**.

   b. One for storing the **reversed strings** (to handle suffixes).

2. **Insert words[i] into both Tries.**
3. **Iterate through all pairs `(i, j)` where `i < j`.**
4. **Check:**
   a. If `words[i]` is a **prefix** of `words[j]`.

   b. If `words[i]` is a **suffix** of `words[j]`.

5. **Count such pairs.**

Code (C++):

```
struct trieNode {
    trieNode* children[26];
    bool isend;
};

// Function to create a new Trie Node
trieNode* getNode() {
    trieNode* newNode = new trieNode();
    newNode->isend = false;
    for (int i = 0; i < 26; i++) {
        newNode->children[i] = NULL;
    }
    return newNode;
}

class Trie {
public:
    trieNode* root;
    Trie() { root = getNode(); }

    // Function to insert a word into the trie
    void insertNode(string& word) {
        trieNode* ptr = root;
        for (char& ch : word) {
            int index = ch - 'a';
            if (ptr->children[index] == NULL) {
                ptr->children[index] = getNode();
            }
            ptr = ptr->children[index];
        }
        ptr->isend = true;
    }

    // Function to search for a prefix in the trie
    bool searchPrefix(string prefix) {
        trieNode* ptr = root;
        for (char& ch : prefix) {
            int index = ch - 'a';
            if (ptr->children[index] == NULL) {
                return false;
            }
            ptr = ptr->children[index];
        }
        return true;
    }
};

class Solution {
public:
    int countPrefixSuffixPairs(vector<string>& words) {
        int n = words.size();
        int count = 0;
        for(int i = 0; i < n; i++){
            Trie prefixTrie;
            Trie suffixTrie;

            // Insert the current word into both tries
            prefixTrie.insertNode(words[i]);
            string reversedString = words[i];
            reverse(reversedString.begin(), reversedString.end());
            suffixTrie.insertNode(reversedString);

            for(int j = 0; j < i; j++){
                if(words[j].length() > words[i].length()) {
                    continue;
                }
                string reversedString_2 = words[j];
                reverse(reversedString_2.begin(), reversedString_2.end());

                // Check both prefix and suffix conditions
                if(prefixTrie.searchPrefix(words[j]) &&
suffixTrie.searchPrefix(reversedString_2)) {
                    count++;
                }
            }
        }
        return count;
    }
};
```

Explanation:

For `words = ["a","aba","ababa","aa"]`:

1. Insert `"a"` into both Tries.
2. Insert `"aba"` and compare it with `"a"` → True (prefix and suffix match).
3. Insert `"ababa"` and compare with `"a"` and `"aba"` → Matches.

4. Insert `"aa"` and compare with `"a"` → Matches.

Output: `4`

Complexity:

- **Time Complexity:** `O(n^2 * m)` where `n` is the number of words and `m` is the maximum word length.
- **Space Complexity:** `O(n * m)` for storing the Trie nodes.

✅ Edge Cases Handled:

- If no valid pairs exist, return `0`.
- If only one word exists, return `0`.
- Words can overlap as prefixes and suffixes.

Alternate Approach (Without Trie)

- Directly compare substrings using nested loops for smaller constraints.
- Time Complexity: `O(n^2 * m)` but simpler to code.

Would you like me to provide the alternative simpler version without a Trie? 😊