02.01.2025

Problem: Count Vowel Strings in Ranges

Problem Statement:

You are given a **0-indexed** array of strings `words` and a 2D array of integers `queries`.

Each query `queries[i] = [li, ri]` asks you to find the number of strings in the range `[li, ri]` (both inclusive) of `words` that **start and end with a vowel**.

Vowels:
`a, e, i, o, u`

Example 1:

Input:
`words = ["aba","bcb","ece","aa","e"], queries = [[0,2],[1,4],[1,1]]`
Output: `[2,3,0]`

Explanation:

- Strings starting and ending with vowels: `"aba"`, `"ece"`, `"aa"`, `"e"`.
- Query `[0,2]` → `"aba"` and `"ece"` → 2 strings
- Query `[1,4]` → `"ece"`, `"aa"`, `"e"` → 3 strings
- Query `[1,1]` → `"bcb"` → 0 strings

Example 2:

Input:
`words = ["a","e","i"], queries = [[0,2],[0,1],[2,2]]`
Output: `[3,2,1]`

Approach: Prefix Sum Technique

Key Idea:

- To efficiently handle multiple queries, use **prefix sum** to store cumulative counts of vowel strings up to each index.
- This allows us to answer each query in **O(1)** time after the prefix sum array is built.

Code:

```
class Solution {
public:
    // Helper function to check if a character is
a vowel
    bool isVowel(char& ch){
        return (ch == 'a' || ch == 'e' || ch ==
'i' || ch == 'o' || ch == 'u');
    }

    vector<int> vowelStrings(vector<string>&
words, vector<vector<int>>& queries) {
        int n = words.size();
        int q = queries.size();

        vector<int> res(q);               //
Result array for storing answers
```

```
        vector<int> prefixSum(n);          //
Prefix sum array
        int sum = 0;                       // To
keep track of cumulative count of valid strings

        // Step 1: Build the prefix sum array
        for (int i = 0; i < n; i++) {
            if (isVowel(words[i].front()) &&
isVowel(words[i].back())) {
                sum++; // If the word starts and
ends with a vowel, increment sum
            }
            prefixSum[i] = sum; // Store the
cumulative sum
        }

        // Step 2: Answer each query using the
prefix sum array
        for (int i = 0; i < q; i++) {
            int start = queries[i][0];
            int end = queries[i][1];
            // To get the count of vowel strings
in the range [start, end]
            res[i] = prefixSum[end] - ((start > 0)
? prefixSum[start - 1] : 0);
        }

        return res;
    }
};
```

Explanation:

Step 1: Check for Vowel Strings

- Use a helper function `isVowel()` to check if a character is a vowel.
- For each word, check if both the **first** and **last** characters are vowels.

Step 2: Build Prefix Sum Array

- If a word meets the condition, increment the cumulative count `sum`.
- Store this cumulative count in `prefixSum[i]`.

Step 3: Answer Queries

- For each query `[start, end]`:
  a. If `start > 0`, result = `prefixSum[end] - prefixSum[start-1]`
  b. If `start == 0`, result = `prefixSum[end]`

Example Walkthrough:

Input: `words = ["aba","bcb","ece","aa","e"]`, `queries = [[0,2],[1,4],[1,1]]`

Step 1: Build the `prefixSum` array

- `"aba"` → starts and ends with a vowel → `prefixSum[0] = 1`
- `"bcb"` → does not meet condition → `prefixSum[1] = 1`
- `"ece"` → meets condition → `prefixSum[2] = 2`
- `"aa"` → meets condition → `prefixSum[3] = 3`
- `"e"` → meets condition → `prefixSum[4] = 4`

`prefixSum = [1, 1, 2, 3, 4]`

Step 2: Answer Queries

- Query `[0,2]`: `prefixSum[2] - 0 = 2`
- Query `[1,4]`: `prefixSum[4] - prefixSum[0] = 4 - 1 = 3`
- Query `[1,1]`: `prefixSum[1] - prefixSum[0] = 1 - 1 = 0`

Final Answer: `[2, 3, 0]`

Complexity Analysis:

- **Time Complexity:**
  a. Building prefix sum: `O(n)`
  b. Answering each query: `O(1)`
  c. Total: `O(n + q)`
- **Space Complexity:**
  a. `O(n)` for the prefix sum array.
  b. `O(q)` for storing results.

Edge Cases Handled:

- If all strings have no vowels → Correctly returns `0` for each query.
- If `words` contains a single string → Query result depends on whether it satisfies the condition.

Key Takeaways:

1. **Prefix Sum:** Efficiently handles multiple queries by preprocessing the data.
2. **Helper Function:** Simplifies checking for vowels.
3. **Optimization:** `O(n + q)` is optimal compared to a brute-force approach of `O(n * q)`.