2416. Sum of Prefix Scores of Strings

You are given an array `words` of size `n` consisting of **non-empty** strings.

We define the **score** of a string `term` as the **number** of strings `words[i]` such that `term` is a **prefix** of `words[i]`.

- For example, if `words = ["a", "ab", "abc", "cab"]`, then the score of `"ab"` is `2`, since `"ab"` is a prefix of both `"ab"` and `"abc"`.

Return *an array* `answer` *of size* `n` *where* `answer[i]` *is the* ***sum*** *of scores of every* ***non-empty*** *prefix of* `words[i]`.

Note that a string is considered as a prefix of itself.

Example 1:

```
Input: words = ["abc","ab","bc","b"]
Output: [5,4,3,2]
Explanation: The answer for each string is the following:
- "abc" has 3 prefixes: "a", "ab", and "abc".
- There are 2 strings with the prefix "a", 2 strings with the prefix "ab", and 1
string with the prefix "abc".
The total is answer[0] = 2 + 2 + 1 = 5.
- "ab" has 2 prefixes: "a" and "ab".
- There are 2 strings with the prefix "a", and 2 strings with the prefix "ab".
The total is answer[1] = 2 + 2 = 4.
- "bc" has 2 prefixes: "b" and "bc".
- There are 2 strings with the prefix "b", and 1 string with the prefix "bc".
The total is answer[2] = 2 + 1 = 3.
- "b" has 1 prefix: "b".
- There are 2 strings with the prefix "b".
The total is answer[3] = 2.
```

Example 2:

```
Input: words = ["abcd"]
Output: [4]
Explanation:
"abcd" has 4 prefixes: "a", "ab", "abc", and "abcd".
Each prefix has a score of one, so the total is answer[0] = 1 + 1 + 1 + 1 = 4.
```

Constraints:

- `1 <= words.length <= 1000`
- `1 <= words[i].length <= 1000`
- `words[i]` consists of lowercase English letters.

2416. Sum of Prefix Scores of Strings

```
/*
   Hash map - TLE

   Time complexity: O(nm+np)
   Space complexity: O(p+p)
   n: size of words' array
   m: the average length of the strings in words array
   p: number of all prefixes formed by every word in words array
*/
class Solution {
   public:
      std::vector<std::string> get_all_prefixes(std::string& s){
         int m=s.size();
         std::string pre="";
         std::vector<std::string> prefixes;
         for(int i=0;i<m;++i){
            pre+=s[i];
            prefixes.push_back(pre);
         }
         return prefixes;
      }

      std::vector<int> sumPrefixScores(std::vector<std::string>& words){
        std::unordered_map<std::string,int> prefixes_counts;
        for(auto& word: words){
            std::vector<std::string> prefixes=get_all_prefixes(word);
            for(auto& prefix: prefixes) {
               prefixes_counts[prefix]++;
            }
        }

        std::vector<int> ans;
        for(auto& word: words){
            int cnt=0;
            std::vector<std::string> prefixes=get_all_prefixes(word);
            for(auto& prefix: prefixes){
               cnt+=prefixes_counts[prefix];
            }
            ans.push_back(cnt);
        }
        return ans;
      }
};
```

2416. Sum of Prefix Scores of Strings

/*
Prefix tree (Trie)
Time complexity: O(nm)
Space complexity: O(nm)
n: size of words' array
m: the average length of the strings in words
*/

```
class Trie{
  private:
    class TrieNode{
      public:
        TrieNode* children[26]={nullptr};
        int count=0;
    };
    TrieNode* root;
  public:
    Trie(){
      root=new TrieNode();
    }
    void insert(std::string& s){
      TrieNode* cur=root;
      for(auto& c: s){
        int i=c-'a';
        TrieNode* node=cur->children[i];
        if(!node){
          node=new TrieNode();
          cur->children[i]=node;
        }
        cur->children[i]->count+=1;
        cur=node;
      }
    }
int compute(std::string& s){
      TrieNode* cur=root;
      int ans=0;
      for(auto& c: s){
        int i=c-'a';
        TrieNode* node=cur->children[i];
        if(node){
          ans+=cur->children[i]->count;
          cur=node;
        }
      }
      return ans;
    }
};
```

```
class Solution {
    public:
        std::vector<int> sumPrefixScores(std::vector<std::string>& words){
            Trie trie=Trie();
            for(auto& word: words) trie.insert(word);

            std::vector<int> ans;
            for(auto& word: words){
                ans.push_back(trie.compute(word));
            }
            return ans;
        }
};
```