<https://leetcode.com/problems/count-pairs-of-similar-strings/description/>

You are given a **0-indexed** string array words.

Two strings are **similar** if they consist of the same characters.

For example, "abca" and "cba" are similar since both consist of characters 'a', 'b', and 'c'.

However, "abacba" and "bcfd" are not similar since they do not consist of the same characters.

Return *the number of pairs*(i, j)*such that*0 <= i < j <= word.length - 1*and the two strings*words[i]*and*words[j]*are similar*.

**Example 1:**

**Input:** words = ["aba","aabb","abcd","bac","aabc"]

**Output:** 2

**Explanation:** There are 2 pairs that satisfy the conditions:

- i = 0 and j = 1 : both words[0] and words[1] only consist of characters 'a' and 'b'.

- i = 3 and j = 4 : both words[3] and words[4] only consist of characters 'a', 'b', and 'c'.

**Example 2:**

**Input:** words = ["aabb","ab","ba"]

**Output:** 3

**Explanation:** There are 3 pairs that satisfy the conditions:

- i = 0 and j = 1 : both words[0] and words[1] only consist of characters 'a' and 'b'.

- i = 0 and j = 2 : both words[0] and words[2] only consist of characters 'a' and 'b'.

- i = 1 and j = 2 : both words[1] and words[2] only consist of characters 'a' and 'b'.

**Example 3:**

**Input:** words = ["nba","cba","dba"]

**Output:** 0

**Explanation:** Since there does not exist any pair that satisfies the conditions, we return 0.

**Constraints:**

1 <= words.length <= 100

1 <= words[i].length <= 100

words[i] consist of only lowercase English letters.

**Attempt 1: 2025-03-31**

**Solution 1: Normalize String + Hash Table (30 min)**

class Solution {

    public int similarPairs(String[] words) {

        Map<String, Integer> freq = new HashMap<>();

        for(String word : words) {

            String normalizedWord = normalize(word);

            freq.put(normalizedWord, freq.getOrDefault(normalizedWord, 0) + 1);

        }

        int count = 0;

        for(int f : freq.values()) {

            count += f \* (f - 1) / 2;

        }

        return count;

    }

    private String normalize(String word) {

        boolean[] seen = new boolean[26];

        for(char c : word.toCharArray()) {

            seen[c - 'a'] = true;

        }

        StringBuilder sb = new StringBuilder();

        for(int i = 0; i < 26; i++) {

            if(seen[i]) {

                sb.append((char)(i + 'a'));

            }

        }

        return sb.toString();

    }

}

Time Complexity: O(n \* m), where n is the number of words and m is the average word length

Space Complexity: O(n), for storing the frequency map

**Solution 2: Bit Manipulation (30 min)**

class Solution {

    public int similarPairs(String[] words) {

        Map<Integer, Integer> freq = new HashMap<>();

        for(String word : words) {

            int mask = 0;

            for(char c : word.toCharArray()) {

                mask |= 1 << (c - 'a');

            }

            freq.put(mask, freq.getOrDefault(mask, 0) + 1);

        }

        int count = 0;

        for(int f : freq.values()) {

            count += f \* (f - 1) / 2;

        }

        return count;

    }

}

Time Complexity: O(n \* m), where n is the number of words and m is the average word length

Space Complexity: O(n), for storing the frequency map

**Refer to Deepseek**

This problem requires counting the number of pairs of similar strings in an array, where two strings are considered similar if they contain the exact same set of characters (order doesn't matter, and duplicates don't count).

**Approach**

Normalize Strings: Convert each string to its unique character set representation (sorted and deduplicated)

Count Frequencies: Use a frequency map to count occurrences of each normalized string

Count Pairs: For each normalized string that appears n times, the number of possible pairs is n \* (n - 1) / 2

**Solution Code**

class Solution {

public int similarPairs(String[] words) {

Map<String, Integer> freqMap = new HashMap<>();

int count = 0;

for (String word : words) {

// Get unique characters and sort them

String normalized = normalize(word);

freqMap.put(normalized, freqMap.getOrDefault(normalized, 0) + 1);

}

// Calculate pairs for each normalized string

for (int freq : freqMap.values()) {

count += freq \* (freq - 1) / 2;

}

return count;

}

private String normalize(String word) {

boolean[] seen = new boolean[26];

for (char c : word.toCharArray()) {

seen[c - 'a'] = true;

}

StringBuilder sb = new StringBuilder();

for (int i = 0; i < 26; i++) {

if (seen[i]) {

sb.append((char)('a' + i));

}

}

return sb.toString();

}

}

**Explanation**

Normalization: The normalize method converts each string to a sorted string of unique characters (e.g., "aba" becomes "ab")

Frequency Counting: We count how many times each normalized string appears in the input array

Pair Counting: For each normalized string with frequency n, we calculate the number of possible pairs using the combination formula n choose 2

**Complexity Analysis**

Time Complexity: O(n \* m), where n is the number of words and m is the average word length

Space Complexity: O(n), for storing the frequency map

**Alternative Solution (Using Bitmask)**

For better performance, we can represent character sets as bitmasks:

class Solution {

public int similarPairs(String[] words) {

Map<Integer, Integer> freqMap = new HashMap<>();

int count = 0;

for (String word : words) {

int mask = 0;

for (char c : word.toCharArray()) {

mask |= 1 << (c - 'a');

}

freqMap.put(mask, freqMap.getOrDefault(mask, 0) + 1);

}

for (int freq : freqMap.values()) {

count += freq \* (freq - 1) / 2;

}

return count;

}

}

This bitmask approach:

Represents each character set as an integer (each bit represents a character)

Has the same time complexity but is more efficient in practice

Reduces space usage as integers are smaller than strings

**Explain how below BitMask section works ?**

int mask = 0;

for (char c : word.toCharArray()) {

mask |= 1 << (c - 'a');

}

The bitmask section is a clever way to represent the set of characters in a string using a single integer. Here's how it works step by step:

**Understanding the Bitmask Concept**

Integer as a Set of Flags:

We use a 32-bit integer where each bit represents whether a particular character exists in the string

For English lowercase letters (a-z), we only need 26 bits (one for each letter)

Initialization:

int mask = 0;

Starts with all bits set to 0 (no characters present)

Setting Bits:

mask |= 1 << (c - 'a');

For each character in the string:

c - 'a' converts the character to a 0-based index (a=0, b=1, ..., z=25)

1 << (c - 'a') creates a bit pattern with 1 at the character's position

|= (bitwise OR assignment) sets that bit in the mask

**Example Walkthrough**

For the word "abc":

'a' (ASCII 97):

'a' - 'a' = 0

1 << 0 = 1 (binary: 000...0001)

mask becomes 1

'b' (ASCII 98):

'b' - 'a' = 1

1 << 1 = 2 (binary: 000...0010)

mask becomes 1 | 2 = 3 (binary: 000...0011)

'c' (ASCII 99):

'c' - 'a' = 2

1 << 2 = 4 (binary: 000...0100)

mask becomes 3 | 4 = 7 (binary: 000...0111)

Final mask: 7 (binary 000...0111), representing characters a, b, and c

**Why This Works for the Problem**

Unique Representation:

Any combination of characters produces a unique bit pattern

"aab" and "ab" both produce the same mask (3) since duplicates don't matter

Efficient Comparison:

We can compare character sets by simply comparing integers

Much faster than comparing strings or sets of characters

Memory Efficient:

Storing an integer (4 bytes) is more efficient than storing strings or sets

Particularly beneficial when dealing with many words

This technique is commonly used in problems involving character sets or when we need to quickly compare/represent combinations of elements.

**Refer to**

[L451.P14.5.Sort Characters By Frequency (Ref.L347)](note://512A6121D907491D8CF8FDD9119E5CC3)

[L1684.Count the Number of Consistent Strings (Ref.L2506)](note://WEB7aacce78a919644996a3b28207cb96c4)

[L2421.Number of Good Paths (Ref.L2506)](note://WEBe0a7e9243acd024b92c4cd8a899fb0d3)