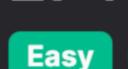
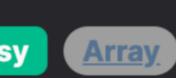
# 2744. Find Maximum Number of String Pairs







**Problem Description** 



Simulation

In this problem, you are given an array words where each element is a distinct string. A pair of strings (words[i] and words[j]) can be formed if one string is the reverse of the other and the indices i and j meet the condition that i < j. The goal is to find the maximum number of such unique pairs from the array words. It's important to note that a string can only be used in one pair, meaning once a string has been paired, it cannot be part of another pair.

**Leetcode Link** 

## Intuition

To solve the problem, we need to keep track of strings that we can form a pair with. As the condition requires one string to be the reverse of another, for any given string, we can look for its reverse in the array. The approach involves iterating through the array, checking if we have previously encountered the reverse of the current string. If so, we have found a viable pair, and we increment our count of pairs.

We utilize a counter (a special dictionary in Python that counts the occurrences of each element in an iterable) to keep track of how

many times we have encountered the reversed strings. Each time we find a string, we look up its reverse in the counter. If the reverse is present, that means we can form a pair, so we increment our answer by the count of the reverse (which, due to the problem's constraints, will always be 1), and then we increment the counter for the reverse of the current string (which prepares us for future potential pairs). This solution ensures that we count each possible pair exactly once and efficiently solves the problem in linear time.

## The implementation of the solution uses a Counter, which is a subclass of dictionary provided by the Python collections module. It's specifically designed to count hashable objects in an iterable. In the context of this problem, it is used to keep track of the

found.

**Solution Approach** 

occurrence of each string's reverse within the words array. Here's a step-by-step approach to how the code works:

1. We initialize a Counter object named cnt and a variable named ans set to 0, which will keep track of the total number of pairs

- 2. The for loop iterates through each word in the given words array.
- It increments the answer ans by the current count of the word in the counter. This count represents the number of times the
- reverse of this word has been encountered before in the array. Because of the problem constraints (distinct strings and each

3. Each iteration performs two operations:

- string can be part of at most one pair), this condition indicates that we've found a pair. • It then increments the count of the reversed word in the counter by 1. This will help to identify a pair when we meet the corresponding reverse in subsequent iterations.
- 4. Finally, after processing all the words, we return ans, which holds the maximum number of unique pairs we can form. In summary, the algorithm utilizes the hash-based lookups provided by the Counter to check for potential pairs efficiently. This
- approach is essentially a one-pass solution since each word in the array is looked at exactly once, making the algorithm O(n) where n is the number of words in the array.

Example Walkthrough Let's walk through a small example to illustrate the solution approach using the following words array:

First, we initialize a Counter named cnt to keep track of the reversed words we encounter and a variable ans to count the number of

1 cnt = Counter()

current word.

pairs.

```
2 \text{ ans} = 0
```

1 words = ["abcd", "dcba", "lls", "s", "sssll"]

1 cnt["dcba"] = 1 2 ans = 0

As we iterate through the words array:

2. The next word "dcba" has a reverse "abcd", which is already in cnt. It's a match! Increment ans by 1 and update cnt for the reverse "badc".

1. For the first word "abcd", its reverse is "dcba", which is not yet in cnt, so no pair is formed. Update cnt with the reverse of the

2 ans = 13. The third word "lls" does not have its reverse in cnt, so no new pair is formed yet.

1 cnt["sll"] = 1

2 ans = 1

1 cnt["badc"] = 1

4. For the single character word "s", there is no reverse in cnt. No pair is formed.

5. The last word is "sssll". Its reverse "llsss" is not in cnt, so no final pair is formed.

```
2 ans = 1
```

1 cnt["llsss"] = 1

1 cnt["s"] = 1

2 ans = 1

```
because each string can be part of at most one pair. Thus, ans = 1 is returned as the final output of the algorithm.
Python Solution
```

from collections import Counter

number of pairs = 0

for word in words:

int totalPairs = 0;

for (String word : words) {

for (auto& word : words) {

// of a previously encountered word

// Reverse the current word in-place

reverse\_word\_count[word]++;

std::reverse(word.begin(), word.end());

max\_pair\_count += reverse\_word\_count[word];

// Iterate through each word in the array.

// Reverse the current word.

// number of times the reverse has appeared.

// the existing count is incremented by 1.

totalPairs += reverseCountMap.getOrDefault(word, 0);

from typing import List

11

12

13

16

17

12

14

15

16

18

20

21

23

24

25

26

27

28

class Solution: def maximumNumberOfStringPairs(self, words: List[str]) -> int: # Initialize a counter to keep track of palindrome pairs palindrome\_pairs\_counter = Counter() # Initialize a variable to count the number of palindrome pairs

After iterating through the array, we find that the maximum number of unique pairs that can be formed is 1, with the paired words

being "abcd" and "dcba". The other words did not pair with any other words, either because their reverses weren't present or

```
# Increment the counter for the reverse of the current word,
18
               # to account for future potential pairs.
19
               palindrome_pairs_counter[word[::-1]] += 1
20
           # Return the total count of palindrome pairs
23
           return number_of_pairs
24
Java Solution
   class Solution {
       /**
        * This method calculates the maximum number of string pairs where one string
        * is the reverse of the other in a given array of strings.
        * @param words An array of strings to find the maximum number of reversible string pairs.
        * @return The maximum number of reversible string pairs.
       public int maximumNumberOfStringPairs(String[] words) {
           // A map to count occurrences of the reversed strings.
```

Map<String, Integer> reverseCountMap = new HashMap<>(words.length);

// Counter to keep track of the total number of reversible string pairs.

String reversedWord = new StringBuilder(word).reverse().toString();

# For each word, check if its reverse is already in the counter,

# which would make a palindrome pair, and if so, increment the pairs count.

# Iterate over each word in the input list of words

number\_of\_pairs += palindrome\_pairs\_counter[word]

```
29
               reverseCountMap.merge(reversedWord, 1, Integer::sum);
30
31
           // Return the computed total number of reversible string pairs.
33
           return totalPairs;
34
35 }
36
C++ Solution
1 #include <algorithm>
2 #include <string>
3 #include <unordered_map>
   #include <vector>
  class Solution {
   public:
       // This function computes the maximum number of pairs of strings
       // where one is the reverse of the other
10
       int maximumNumberOfStringPairs(std::vector<std::string>& words) {
           // Create a hash map to count occurrences of each reversed word
11
12
           std::unordered_map<std::string, int> reverse_word_count;
           // Initialize count of valid pairs to zero
13
           int max_pair_count = 0;
14
15
           // Loop through each word in the vector
16
```

// Increment the count of pairs for each word that is a reverse

// Increment the count of the reversed word in our hash map

// If the word has appeared as a reverse before, increment the total pairs count by the

// Update the reverse count map by incrementing the count of the reverse of the current word.

// If this reversed word is not in the map, it's added with a count of 1. If it is already there,

### 27 28 29 // Answer is the total count of valid pairs found 30

20

21

22

23

24

25

26

```
return max_pair_count;
31
32 };
33
Typescript Solution
   function maximumNumberOfStringPairs(words: string[]): number {
       // Create a map to count the occurrences of each reversed word.
       const reversedWordCount: Map<string, number> = new Map();
       let pairCount = 0; // Initialize the count of pairs to 0.
       // Iterate over the array of words.
       for (const word of words) {
           // Check if the current word has a reverse counterpart already counted.
           pairCount += reversedWordCount.get(word) || 0;
           // Find the reversed string of the current word.
11
           const reversedWord = word.split('').reverse().join('');
12
           // Update the count of the reversed word in the map, incrementing the current value or setting to 1 if not present.
14
           reversedWordCount.set(reversedWord, (reversedWordCount.get(reversedWord) | | 0) + 1);
15
16
17
       // Return the total number of pairs found.
       return pairCount;
```

# Time and Space Complexity

### 18 19 20 } 21

map update is 0(1).

Time Complexity The time complexity of the provided code is primarily determined by the loop that iterates through each word in the input list words. The operations within the loop are constant-time lookups and updates to the counter, cnt, which is a hash map. For n words:

- The ans += cnt[w] operation is 0(1) because it is a hash map lookup. • The cnt[w[::-1]] += 1 operation involves reversing the word which costs 0(k) where k is the length of the word, and the hash
- Since the reversal of the word is the most expensive operation within the loop, and it happens for each word, the overall time complexity is O(nk), where n is the number of words and k is the average length of the words.

**Space Complexity** 

The space complexity is determined by the space required to store the counter, cnt. In the worst-case scenario, if all words and their reverses are unique, the counter would need to store an entry for each unique word and its reversed counterpart. Therefore, the space complexity is O(n) where n is the number of unique words in the list.