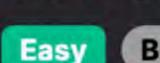
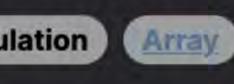
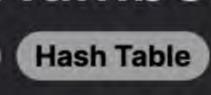
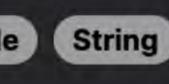
## 1684. Count the Number of Consistent Strings



Bit Manipulation







Leetcode Link

# Problem Description

string allowed. Additionally, we are given an array of strings called words. Our task is to determine how many strings in the array words are "consistent."

The problem provides us with a string allowed which is made up of distinct characters, meaning no character is repeated in the

character in a word that is not present in allowed, then that word is not considered consistent. The goal is to return the number of consistent strings in the array words.

A string is defined as consistent if every character in the string appears in the allowed string. In other words, if there's even one

Intuition

allowed string. To optimize this process, we convert the allowed string to a set. Sets in Python are collections that provide O(1) time complexity for checking if an element is present, which is faster than if allowed were a list or a string, where the time complexity would be O(n). Once we have the set of allowed characters, we iterate over each word in words. For each word, we check whether every character

To solve this problem, the intuitive approach is to check each word in words and ensure all of its characters are contained within the

word) are True (or if the iterable is empty). The expression (c in s for c in w) is a generator that yields True or False for each character c in the word w depending on whether c is in the set s or not. The all() function takes this generator as input and evaluates to True only if every character in the

is in our set of allowed characters. We use the all() function which returns True if all elements of the iterable (the characters in the

word is in the set of allowed characters. Finally, we use the sum() function to count how many words are consistent by adding up 1 for every True result from the all() check.

The result is the number of consistent strings in the words array, which is what we return.

Converting to a set is significant because set operations in Python are usually faster than list or string operations. Specifically,

The solution provided uses Python's set and comprehension features to perform the task efficiently.

## 1. The first step is to convert the allowed string into a set:

1 s = set(allowed)

Solution Approach

checking for membership using the in operator is very fast for sets, taking O(1) time on average.

2. Next, we use a list comprehension to iterate over each word in words:

This comprehensive line does several things:

1 sum(all(c in s for c in w) for w in words)

set s. The all() function checks if all values provided by this generator are True, meaning every character in the word is in

the allowed set.

• The entire all() expression will be True if the word is consistent and False otherwise.

o for w in words: We start by going over each word in the words list.

o sum(...): We are then using sum() to add up all the True values. Since True is equivalent to 1 in Python, sum() effectively counts the number of consistent strings. This algorithm is very concise and takes advantage of Python's high-level operations to solve the problem with both simplicity and

o all(c in s for c in w): For each word, we create a generator expression that yields True for each character c that is in the

Example Walkthrough

efficiency. The use of all() combined with a generator expression and sum() allows us to perform the entire operation in a single,

Let's walk through a small example to illustrate the solution approach. Suppose the string allowed contains the characters "abc" and we have an array words containing the words ["ab", "ac", "de", "abc", "xy"]. Now, let's apply the solution step by step:

### 1. We convert the allowed string into a set to expedite membership checks: 1 s = set('abc') # This creates the set {'a', 'b', 'c'}

readable line of code.

1 result = sum(all(c in s for c in w) for w in words)

• For the word "ac": the characters are 'a' and 'c', both of which are in the set s. The all() function will also return True.

2. Next, we iterate over each word in words and use the all() function to check if every character of a word is in the set s:

```
• For the word "ab": the characters are 'a' and 'b', both of which are in the set s. So all(c in s for c in 'ab') will yield True
```

for both characters, and all() will return True.

• For the word "de": the character 'd' is not in the set s, so all(c in s for c in 'de') will yield False when checking 'd', and all() will return False.

'x', and all() will return False.

Let's break this down for each word:

- For the word "abc": all the characters 'a', 'b', and 'c' are in the set s, and all() will return True. • For the word "xy": neither 'x' nor 'y' is in the set s, so all(c in s for c in 'xy') will yield False immediately when checking
- So, we get the following results for our words array:
  - "de": Not consistent "abc": Consistent

1 result = sum([True, True, False, True, False]) # This equals 3

allowed\_chars = set(allowed)

Therefore, the output would be 3, as there are three consistent strings in the array words.

class Solution:

11

10

11

12

13

14

15

16

17

18

19

32

33

34

35

29

30

31

32

33

34

35

36

37 };

**Python Solution** 

• "ab": Consistent

"ac": Consistent

"xy": Not consistent

# Count the words in which all the characters are in the allowed set # Summing up the boolean values where True is counted as 1, False as 0 consistent\_words\_count = sum(all(char in allowed\_chars for char in word) for word in words) return consistent\_words\_count # Return the total count of consistent words

def countConsistentStrings(self, allowed: str, words: list[str]) -> int:

# Convert the allowed string into a set for O(1) lookup times

public int countConsistentStrings(String allowed, String[] words) {

// Populate the isAllowed array with characters from the 'allowed' string

Finally, the sum() function would add up all the True values (represented as 1 in Python):

Java Solution 1 class Solution {

// Array to keep track of allowed characters

// Method to count number of consistent strings

boolean[] isAllowed = new boolean[26];

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

// Initialize count for consistent strings

if (isConsistent(word, isAllowed)) {

if (!isAllowed[word.charAt(i) - 'a']) {

// Iterate through each word in the array 'words'

// If the word is consistent, increment the count

isAllowed[c - 'a'] = true;

for (String word : words) {

return false;

if (isConsistent(word)) {

++consistentCount;

return consistentCount;

// Return the final count of consistent strings

count++;

int count = 0;

```
20
21
22
23
           // Return the total count of consistent strings
24
            return count;
25
26
27
       // Helper method to check if a word is consistent
28
       private boolean isConsistent(String word, boolean[] isAllowed) {
29
           // Iterate through each character in the word
            for (int i = 0; i < word.length(); i++) {</pre>
30
                // If the character is not in the allowed list, return false
31
```

```
36
37
           // Return true if all characters of the word are allowed
38
           return true;
39
40 }
41
C++ Solution
   #include <vector>
2 #include <string>
   #include <bitset>
   class Solution {
6 public:
       int countConsistentStrings(std::string allowed, std::vector<std::string>& words) {
           // Initialize a bitset to store whether each letter in the alphabet is allowed
           std::bitset<26> allowedLetters;
           for (char ch : allowed) {
10
               allowedLetters.set(ch - 'a');
11
12
13
           // Counter for the number of consistent strings
14
15
           int consistentCount = 0;
16
17
           // Lambda function to check if all characters in a word are allowed
           auto isConsistent = [&](std::string& word) {
18
               for (char ch : word) {
19
                   // If any character is not allowed, return false immediately
20
                   if (!allowedLetters.test(ch - 'a')) return false;
21
22
23
               // All characters in the word are allowed
24
               return true;
25
           };
26
           // Iterate through each word and increment the consistent count if the word is consistent
28
           for (std::string& word : words) {
```

#### // @param allowed - A string consisting of distinct lowercase English letters, representing the allowed characters. // @param words - An array of strings, where each string consists only of lowercase English letters. // @return The count of words from the 'words' array that are "consistent" - contain only characters from 'allowed'.

Typescript Solution

```
function countConsistentStrings(allowed: string, words: string[]): number {
       // Converts a string to a bitmask where each bit set represents the presence of a character in the string.
       // @param str - A string to convert to a bitmask.
       // @return A bitmask representing the characters of the input string.
       const convertToBitmask = (str: string): number => {
9
           let bitmask = 0;
           for (const char of str) {
11
               bitmask |= 1 << (char.charCodeAt(0) - 'a'.charCodeAt(0));
12
13
14
           return bitmask;
       };
15
16
17
       // Create a bitmask for the allowed characters.
       const allowedMask = convertToBitmask(allowed);
18
       let consistentCount = 0; // Initialize the count of consistent strings.
19
20
21
       // Loop through each word in the array.
       for (const word of words) {
           // If the bitmask of the word OR'd with the allowedMask equals the allowedMask,
           // it means the word contains only characters from 'allowed'.
24
25
           if ((allowedMask | convertToBitmask(word)) === allowedMask) {
               consistentCount++; // Increment count as the word is consistent.
26
27
28
29
       // Return the total count of consistent strings.
30
31
       return consistentCount;
Time and Space Complexity
Time Complexity
```

// Counts the number of words from the 'words' array that contain only characters from the 'allowed' string.

The time complexity of the function depends on two factors: the length of the words list meaning the number of words it contains

(let's denote this number as n) and the average length of the words (let's denote it as k). The function iterates over each word and

then over each character in the word to check if it is in the set s. Checking membership in a set is an O(1) operation on average.

# Therefore, the time complexity for checking one word is O(k), and for all the words it's O(n \* k).

Space Complexity The space complexity is O(s) where s is the number of unique characters in the allowed string because these are stored in a set. The other variable that could contribute to space complexity is win the generator expression, but since the space required for wis reused as the iteration proceeds and since the strings in words are input to the function and not duplicated by it, this does not add to the space complexity of the solution itself. The space complexity for the output of the sum function is 0(1) since it's accumulating the result into a single integer. Therefore, the overall space complexity is 0(s) + 0(1) which simplifies to 0(s).