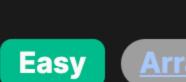
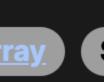
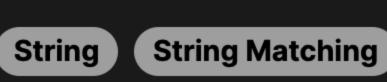
# 1408. String Matching in an Array





**Problem Description** 



In this problem, we are given an array of string words. Our task is to find all the strings in the array that are a substring of another string within the same array. In other words, we are looking for any string w1 from the words array that can be found within another string w2 in the same array, where w1 and w2 are not the same string. A substring, by definition, is a contiguous sequence of characters within a string; it could be as short as one character or as long as the string itself minus one character. Our final output is an array that includes these substrings, and the order of the substrings in the output doesn't matter.

## Intuition

achieve this by comparing each word with every other word in the array. This can be done through a nested loop where for every word w1, we look through the whole array to check if there's another word w2 that contains w1 as a substring. If such a w2 is found, we can conclude that w1 is a substring of another word in the array. We use an inner loop and an outer loop to iterate through the array of words. For each word in the outer loop (w1), we iterate

The intuition behind the solution is straightforward: we need to find if any word is part of another word in the array. We can

through all other words in the array in the inner loop (w2). If we find that w1 is contained within w2 (w1 is a substring of w2) and w1 is not the same as w2 (to avoid comparing a word with itself), we can add w1 to the answer list. Once w1 has been found as a substring, we don't need to check the rest of the words for w1, so we use a break statement to move on to the next word in the outer loop. This approach ensures that we check all pairs of words to determine which ones are substrings of another. Once all checks are

completed, we return the list of substrings we found. Solution Approach

### The solution approach takes advantage of a simple brute force method to solve the problem, utilizing two nested for loops to compare every pair of words.

**Algorithms and Patterns** 

### 1. We iterate through the words array using an outer loop. Each word encountered in this loop will be referred to as w1.

**Brute Force Comparison** 

#### 2. For each w1, we use another inner loop to iterate through all other words in the array, which we refer to as w2. 3. We compare w1 to every w2, ensuring not to compare a word with itself by checking that the indices i and j are not equal (i != j). If w1 is the

- 1. The check w1 in w2 is used to determine if w1 is a substring of w2. This is a built-in operation in Python that checks for the presence of a
- **Substring Check**

same as w2, it doesn't count as a substring, so we only want to find instances where w1 is contained in a different w2.

3. After appending w1 to the answer list, we don't need to check it against the remaining w2 words in the loop, as we have already confirmed it is a substring. This is where we break the inner loop.

# **Code Representation**

**Data Structures** 

class Solution: def stringMatching(self, words: List[str]) -> List[str]:

• A list ans is used to store the words that satisfy the condition of being a substring of another word.

#### ans = []for i, w1 in enumerate(words): for j, w2 in enumerate(words):

if i != j and w1 in w2: ans.append(w1)

The solution in Python looks like this:

sequence of characters within another sequence.

2. If the condition is met, we append w1 to our answer list ans.

```
break
      return ans
In this code snippet, enumerate is a handy function that provides a counter (i and j) to the loop which we use to ensure we are
not comparing the same word with itself. When we find a match, we immediately break out of the inner loop to avoid unnecessary
checks, which is a minor optimization within the brute force approach.
The idea of breaking out of the loop as soon as we find a word is a substring ensures that the algorithm doesn't do excessive
```

each character within those words for being a substring. By returning the ans list after the loops have completed, we have solved the problem by collecting all strings in the input array that are substrings of another word.

work. However, it's worth noting that the algorithm's time complexity is O(n^2 \* m), where n is the number of words and m is the

average length of a word. This is because, in the worst case, for each outer loop word, we potentially check all other words and

Let's consider a small example to illustrate the solution approach. Suppose we have the following array of strings: words = ["mass", "as", "hero", "superhero", "cape"]

We want to find all strings that are a substring of another string in this list. Using the brute force method, we follow these steps:

## We start with the outer loop, taking w1 = "mass".

We move on to w1 = "as".

Next is w1 = "hero".

determine our answer.

from typing import List

class Solution:

Solution Implementation

**Example Walkthrough** 

 When j = 0, we compare w1 with "mass"; since they are the same, we continue. When j = 1, we compare w1 with "as"; the word "mass" does not contain "as" as a substring, we continue.

○ When j = 0, comparing "as" to "mass", we find that w1 is a substring of w2. We add "as" to the answer list and break the loop.

def string\_matching(self, words: List[str]) -> List[str]:

# Check if word1 is a substring of word2

matching\_substrings.append(word1)

# Initialize an empty list for the answer

for index\_outer, word1 in enumerate(words):

if word1 in word2:

break

// Return the list of matched strings

// Iterate through the words again for comparison

if (word !== targetWord && word.includes(targetWord)) {

return substrings; // Return the array containing all found substrings

check, the overall time complexity of this algorithm would be  $0(n^2 * m^2)$ .

for (const word of words) {

# Initialize an empty list for the answer

return matchedStrings;

# Return the list of matching substrings

The same goes for "superhero" and "cape", no matches are found.

When j = 2, we compare w1 with "hero", we continue since there's no match.

We enter the inner loop and compare w1 with every w2.

- No matches for "mass", "as", but when we reach "superhero", we find that w1 is a substring of w2. We add "hero" to the answer list and break the loop.
- substrings of "mass" and "superhero" respectively. By looping through the entire list of words and comparing each pair, we have effectively applied the brute force solution to

For w1 = "superhero" and w1 = "cape", we find no substrings, as they are not contained in any other word in the list.

Now that we have gone through each word, our answer list ans contains ["as", "hero"]. These two words were identified as

**Python** 

#### # Compare the current word (word1) with every other word in the list for index\_inner, word2 in enumerate(words): # Make sure not to compare the word with itself if index\_outer != index\_inner:

matching\_substrings = []

# Iterate over the list of words

```
return matching_substrings
Java
import java.util.List;
import java.util.ArrayList;
class Solution {
    // Method to find all strings in an array that are substrings of another string
    public List<String> stringMatching(String[] words) {
       // Initialize an empty list to hold the answer
       List<String> matchedStrings = new ArrayList<>();
       // Get the number of words in the array
       int numberOfWords = words.length;
       // Iterate through each word in the array
        for (int i = 0; i < numberOfWords; ++i) {</pre>
           // Inner loop to compare the current word with others
            for (int j = 0; j < numberOfWords; ++j) {</pre>
                // Check if words are different and if the current word is contained within another word
                if (i != j && words[j].contains(words[i])) {
                    // If the condition is true, add the current word to the list of matched strings
                    matchedStrings.add(words[i]);
                    // Break out of the inner loop, as we already found a matching word
                    break;
```

# If it is, add to the answer list and break to avoid duplicates

```
C++
class Solution {
public:
    // Function to find all strings in 'words' that are substrings of other strings
    vector<string> stringMatching(vector<string>& words) {
        vector<string> result; // To store the substrings
        int numWords = words.size(); // Number of words in the input vector
        // Loop through each word in the vector
        for (int i = 0; i < numWords; ++i) {</pre>
           // Nested loop to compare the current word with every other word in the vector
            for (int j = 0; j < numWords; ++j) {</pre>
                // Check if the current word is a substring of any other word, but not the same word
                if (i != j && words[j].find(words[i]) != string::npos) {
                    result.push_back(words[i]); // If it's a substring, add to the result vector
                    break; // No need to check other words, break out of inner loop
        return result; // Return the vector containing all substrings
};
TypeScript
function stringMatching(words: string[]): string[] {
    const substrings: string[] = []; // Initialize an array to hold our result of substrings
   // Iterate through each word in the provided array
    for (const targetWord of words) {
```

// Check if the current word is not the targetWord and includes the targetWord as a substring

break; // Break out of the inner loop as we have found the targetWord as a substring

substrings.push(targetWord); // If conditions met, add the targetWord to our result array

```
from typing import List
class Solution:
   def string_matching(self, words: List[str]) -> List[str]:
```

```
matching substrings = []
       # Iterate over the list of words
       for index_outer, word1 in enumerate(words):
           # Compare the current word (word1) with every other word in the list
           for index_inner, word2 in enumerate(words):
               # Make sure not to compare the word with itself
               if index_outer != index_inner:
                   # Check if word1 is a substring of word2
                   if word1 in word2:
                       # If it is, add to the answer list and break to avoid duplicates
                       matching_substrings.append(word1)
                       break
       # Return the list of matching substrings
       return matching_substrings
Time and Space Complexity
Time Complexity
  The time complexity of the given code can be analyzed by looking at the two nested loops where it iterates over all possible pairs
```

### of strings in the list, except when they are the same string (i.e., where i != j). For each pair, the code checks if one string is a substring of another by using in operation.

If we assume n is the number of words and m the maximum length of a word, the check w1 in w2 has a worst-case complexity of 0(m^2) because in the worst case, every character in w1 might be checked against every character in w2 before a match is found or the end of w2 is reached. Given that we have two nested loops each going through n elements, and assuming the worst-case scenario for the substring

**Space Complexity** 

The space complexity of the code is determined by the additional memory we use, aside from the input. Here, the only extra

space that we use is the ans list, which in the worst case may contain all the original strings, if all strings are substrings of at least

one other string. Thus the space required for the ans list is 0(n \* m), as it can store at most n strings, with each string having a length at most m.

The final space complexity for the algorithm is 0(n \* m) since this is the most significant additional space the algorithm uses.