07.01.2025

Problem: String Matching in an Array

Problem Statement:
Given an array of strings `words`, return all strings in `words` that are substrings of another word.

- Return the answer in **any order**.
- A **substring** is a contiguous sequence of characters within a string.

Examples:

Example 1:

```
Input: words = ["mass","as","hero","superhero"]
Output: ["as","hero"]
Explanation: "as" is a substring of "mass" and "hero" is a substring of
"superhero".
```

Example 2:

```
Input: words = ["leetcode","et","code"]
Output: ["et","code"]
Explanation: "et" and "code" are substrings of "leetcode".
```

Example 3:

```
Input: words = ["blue","green","bu"]
Output: []
Explanation: No word is a substring of another word.
```

Approach: Nested Loops with String Matching (Brute Force)

Steps to Solve:

1. **Initialize a result list.**
2. **Iterate through each word** and compare it with every other word in the list.
3. For each pair `(words[i], words[j])` where `i != j`:

   a. Check if `words[i]` is a substring of `words[j]` using the `find()` method.

4. If `words[i]` is a substring of `words[j]`, add it to the result list and break the loop to avoid duplicates.
5. Return the result list.

Code (C++):

```
class Solution {
public:
    vector<string> stringMatching(vector<string>& words) {
        vector<string> res;
        for(int i = 0; i < words.size(); i++) {
            for(int j = 0; j < words.size(); j++) {
                if(i == j) continue;
                // Check if words[i] is a substring of words[j]
                if(words[j].find(words[i]) != string::npos) {
                    res.push_back(words[i]);
                    break;
                }
            }
        }
        return res;
    }
};
```

Explanation:

For `words = ["mass","as","hero","superhero"]`:

1. `"as"` is checked and found in `"mass"`. ✅
2. `"hero"` is checked and found in `"superhero"`. ✅
3. `"mass"` is not a substring of any word. ❌
4. `"superhero"` is not a substring of any word. ❌

Output: `["as", "hero"]`

Complexity:

- **Time Complexity:** `O(n^2 * m)`
   a. `n^2` for the nested loop.
   b. `m` for substring matching using `find`.
- **Space Complexity:** `O(n)` for the result list.

Edge Cases Handled:

- If `words` has only one element → Return an empty list.
- If no word is a substring → Return an empty list.
- All words are substrings of each other → Return all except the longest.

Optimized Approach (Using Sorting + String Matching):

1. **Sort** the array by string length (ascending order).
2. Iterate through the sorted array and check each word against all longer words.

This can improve efficiency slightly by reducing unnecessary checks.