522. Longest Uncommon Subsequence II

Medium Array Hash Table Two Pointers String Sorting

Leetcode Link

Problem Description

Given an array of strings named `strs`, the task is to find the length of the longest uncommon subsequence among the strings. A string is considered an uncommon subsequence if it is a subsequence of one of the strings in the array, but not a subsequence of any other strings in the array.

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. For example, `abc` is a subsequence of `aebdc` because you can remove the characters `e` and `d` to obtain `abc`.

If no such uncommon subsequence exists, the function should return `-1`.

Intuition

To solve this problem, the key observation is that if a string is not a subsequence of any other string, then it itself is the longest uncommon subsequence. We can compare strings to check whether one is a subsequence of another. We'll perform this comparison for each string in the array against all other strings.

Here's the approach:

1. We will compare each string (`strs[i]`) to every other string in the array to determine if it is a subsequence of any other string.
2. If `strs[i]` is found to be a subsequence of some other string, we know it cannot be the longest uncommon subsequence, so we move on to the next string.
3. If `strs[i]` is not a subsequence of any other strings, it is a candidate for the longest uncommon subsequence. We update our answer with the length of `strs[i]` if it is longer than our current answer.
4. We continue this process for all strings in the array, and in the end, we return the length of the longest uncommon subsequence found. If we don't find any, we return `-1`.

The reason why comparing subsequence relations works here is because a longer string containing the uncommon subsequence must itself be uncommon if none of the strings is a subsequence of another one. This means that the longest string that doesn't have any subsequences in common with others is effectively the longest uncommon subsequence.

Solution Approach

The solution approach follows a brute-force strategy to check each string against all others to see if it is uncommon. Let's break it down step by step:

1. A helper function `check(a, b)` is defined to check if string `a` is a subsequence of string `a`. It iterates through both strings concurrently using two pointers, `j` for `a` and `i` for `b`. If characters match (`a[i] == b[j]`), it moves the pointer `j` forward. The function returns `True` if it reaches the end of string `b`, meaning `b` is a subsequence of `a`.
2. The main function `findLUSlength` initializes an answer variable `ans` with value `-1`. This will hold the length of the longest uncommon subsequence or `-1` if no uncommon subsequence exists.
3. We iterate through each string `strs[i]` in the input array `strs` and for each string, we again iterate through the array to compare it against every other string.
4. During the inner iteration, we compare `strs[i]` to every other string `strs[j]`. If a match is found (meaning `strs[i]` is a subsequence of `strs[j]`), we break out of the inner loop as `strs[i]` cannot be an uncommon subsequence.
5. If `strs[i]` is not found to be a subsequence of any other string (`j` reaches `n`, the length of `strs`), it means `strs[i]` is uncommon. At this point, we update `ans` with the maximum of its current value or the length of `strs[i]`.
6. After completing the iterations, we return the final value of `ans` as the result.

This solution uses no additional data structures, relying on iterations and comparisons to find the solution. The solution's time complexity is O(n^2 * m), where `n` is the number of strings and `m` is the length of the longest string. The space complexity is O(1) as no additional space is required besides the input array and pointers.

Example Walkthrough

Let's use the following small example to illustrate the solution approach:

Consider `strs = ["a", "b", "aa", "c"]`

Here is the step-by-step walkthrough of the above solution:

1. We will start with `strs[0]` which is `"a"`. We compare `"a"` with every other string in `strs`. No other string is `"a"`, so `"a"` is not a subsequence of any string other than itself. The answer `ans` is updated to max(-1, length("a")), which is 1.
2. Next, we look at `strs[1]`, which is `"b"`. Similar to the first step, compare `"b"` with every other string. Since it is unique and not a subsequence of any other string, `ans` becomes max(1, length("b")), which remains 1, since the length of "b" is also 1.
3. We then move on to `strs[2]`, which is `"aa"`. Repeat the same procedure. When comparing `"aa"` with other strings, we realize `"aa"` is not a subsequence of `"a"`, `"b"`, or `"c"`. Therefore, update `ans` to max(1, length("aa")), which is 2 now.
4. Lastly, look at `strs[3]`, which is `"c"`. Again, since `"c"` isn't a subsequence of any other string in the array, we compare `ans` to the length of `"c"`. The `ans` value remains 2 because max(2, length("c")) still equals 2.

After completing the iterations, since we have found uncommon subsequences, the final answer `ans` is 2, which is the length of the longest uncommon subsequence, `"aa"` from our array `strs`.

So, the function `findLUSlength(["a", "b", "aa", "c"])` returns 2.

Python Solution

```python
from typing import List

class Solution:
    def findLUSlength(self, strs: List[str]) -> int:
        # Helper function to check if string b is a subsequence of string a.
        def is_subsequence(a: str, b: str) -> bool:
            i = j = 0
            while i < len(a) and j < len(b):
                if a[i] == b[j]:
                    j += 1  # Move to the next character in b if there's a match.
                i += 1  # Move to the next character in a.
            return j == len(b)  # Check if all characters in b are matched.

        num_strings = len(strs)
        longest_unique_length = -1  # Initialize with -1 as we may not find any unique strings.

        # Iterate over each string in 'strs'.
        for i in range(num_strings):
            j = 0
            # Compare the selected string with all other strings.
            while j < num_strings:
                # Skip if comparing the string with itself or if 'strs[j]' is not a subsequence of 'strs[i]'.
                if i == j or not is_subsequence(strs[i], strs[j]):
                    j += 1  # Move to the next string for comparison.
                else:
                    break  # 'strs[j]' is a subsequence of 'strs[i]', hence not unique.
            # If we reached the end after comparisons, 'strs[j]' is unique.
            if j == num_strings:
                # Update the maximum length with the length of this unique string.
                longest_unique_length = max(longest_unique_length, len(strs[i]))

        # Return the length of the longest uncommon subsequence.
        return longest_unique_length
```

Java Solution

```java
class Solution {

    // This method finds the length of the longest uncommon subsequence among the given array of strings.
    public int findLUSlength(String[] strs) {
        int longestLength = -1; // Initialize with -1 to account for no solution case.

        // We iterate over each string in the array to check if it's a non-subsequence of all other strings in the array.
        for (int i = 0, j = 0, n = strs.length; i < n; ++i) {
            j = 0;

            // We iterate over the strings again looking for a common subsequence.
            for (; j < n; ++j) {
                // We skip the case where we compare the string with itself.
                if (i == j) {
                    continue;
                }

                // If the current string (strs[i]) is a subsequence of strs[j], then break out of this loop.
                if (isSubsequence(strs[i], strs[j])) {
                    break;
                }
            }

            // If we've gone through all strings without breaking, then strs[i] is not a subsequence of any other string.
            if (j == n) {
                // We update the longestLength if strs[i]'s length is greater than the current longestLength.
                longestLength = Math.max(longestLength, strs[i].length());
            }
        }

        // Return the length of the uncommon subsequence. If there are none, return -1.
        return longestLength;
    }

    // This private helper method checks if string b is a subsequence of string a.
    private boolean isSubsequence(String a, String b) {
        int j = 0; // This will be used to iterate over string b.

        // Iterate over string a and string b to check if all characters of b are also in a in the same order.
        for (int i = 0; i < a.length() && j < b.length(); ++i) {
            // If we find a matching character, move to the next character in b.
            if (a.charAt(i) == b.charAt(j)) {
                ++j;
            }
        }

        // After the loop, if j equals the length of b, it means all characters of b are found in a in order.
        return j == b.length();
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

using std::vector;
using std::string;
using std::max;

class Solution {
public:
    // This function is to find the length of the longest uncommon subsequence among the strings.
    // An uncommon subsequence does not appear as a subsequence in any other string.
    int findLUSlength(vector<string>& strs) {
        int longestUncommonLength = -1; // Initialize the result with -1 to indicate no result.

        // Iterate over all strings in the array to find the longest uncommon subsequence.
        for (int i = 0, j = 0, n = strs.size(); i < n; ++i) {
            for (j = 0; j < n; ++j) {
                // If we're comparing the string with itself, continue.
                if (i == j) continue; // Skip comparing the string with itself.
                // If current string strs[i] is a subsequence of strs[j], break and move to the next string.
                if (isSubsequence(strs[i], strs[j])) break;
            }
            // If no subsequence is found in any other string, update the longest uncommon length.
            if (j == n) longestUncommonLength = max(longestUncommonLength, (int)strs[i].size());
        }
        return longestUncommonLength;  // Return the length of the longest uncommon subsequence.
    }

private:
    // This function checks if string 'b' is a subsequence of string 'a'.
    bool isSubsequence(const string& a, const string& b) {
        int indexB = 0;  // Index for iterating through string 'b'.
        // Iterate over string 'a' with 'i' and string 'b' with 'indexB'.
        for (int i = 0; i < a.size() && indexB < b.size(); ++i) {
            if (a[i] == b[indexB]) ++indexB;  // If current chars are equal, move to the next char in 'b'.
        }
        // If 'indexB' reached the end of 'b', it means 'b' is a subsequence of 'a'.
        return indexB == b.size();
    }
};
```

Typescript Solution

```typescript
// This function checks if string 'sub' is a subsequence of string 'main'.
function isSubsequence(main: string, sub: string): boolean {
    let subIndex: number = 0;  // Index for iterating through the subsequence 'sub'.

    // Iterate over the main string with 'i' and the subsequence with 'subIndex'.
    for (let i = 0; i < main.length && subIndex < sub.length; i++) {
        if (main.charAt(i) === sub.charAt(subIndex)) {
            subIndex++; // If the current characters are equal, move to the next character in 'sub'.
        }
    }

    // If 'subIndex' reached the end of 'sub', it means 'sub' is a subsequence of 'main'.
    return subIndex === sub.length;
}

// This function finds the length of the longest uncommon subsequence among the strings.
// An uncommon subsequence does not appear as a subsequence in any other string.
function findLUSlength(strs: string[]): number {
    let longestUncommonLength: number = -1; // Initialize the result with -1 to indicate no result.

    // Iterate over all strings in the array to find the longest uncommon subsequence.
    for (let i = 0, n = strs.length; i < n; i++) {
        let j: number;
        for (j = 0; j < n; j++) {
            if (i === j) continue; // Skip comparing the string with itself.

            // If current string strs[j] is a subsequence of strs[i], break and move to the next string.
            if (isSubsequence(strs[i], strs[j])) break;
        }
        // If no subsequence is found in any other string, update the longest uncommon length.
        if (j === n) {
            longestUncommonLength = Math.max(longestUncommonLength, strs[i].length);
        }
    }

    return longestUncommonLength; // Return the length of the longest uncommon subsequence.
}
```

Time and Space Complexity

The provided code snippet defines the `findLUSlength` function, which finds the length of the longest uncommon subsequence among an array of strings. The time complexity and space complexity analysis for the code is as follows:

Time Complexity

The time complexity of the algorithm is `O(n^2 * m)`, where `n` is the number of strings in the input list `strs`, and `m` is the length of the longest string among them.

Here's the justification for this complexity:

- There are two nested loops, with the outer loop iterating over all strings (`O(n)`) and the inner loop potentially iterating over all other strings (`O(n)`) in the worst case.
- Within the inner loop, the `check` function is called, which in worst-case compares two strings in linear time relative to their lengths. Since we're taking the length of the longest string as `m`, each check call could take up to `O(m)` time.

Hence, the multiplication of these factors leads to the `O(n^2 * m)` time complexity.

Space Complexity

The space complexity of the algorithm is `O(1)`.

The explanation is as follows:

- The extra space used in the algorithm includes constant space for variables `i`, `j`, `ans`, and the space used by the `check` function.
- The `check` function uses constant space aside from the input since it only uses simple counter variables (`i` and `j`), which don't depend on the input size.

Thus, the overall space complexity is constant, regardless of the input size.