📝 Longest Uncommon Subsequence II – Documentation

1. Problem Statement

You are given an array of strings strs. You need to return the length of the longest uncommon subsequence (LUS) among them.

An uncommon subsequence is defined as a string that is a subsequence of one string only, and not a subsequence of any other string in the array.

If no such string exists, return -1.

2. Intuition

- The longest uncommon subsequence must be a string that:
  - Exists in the list.
  - Is not a subsequence of any of the other strings.
- The longer the string, the better chance it has of being the answer (if it's unique), since shorter subsequences are more likely to appear in other strings.

3. Key Observations

- A string is always a subsequence of itself.
- Any string that appears more than once cannot be an LUS.
- If a string is a subsequence of any other string in the list, it is disqualified.
- Sorting strings by length (descending) helps us check longest candidates first.

4. Approach

- Sort the strings in descending order by length.
- For each string s, check if s is not a subsequence of any other string in the array.
- If such a string is found, return its length.

- If no valid string is found, return -1.

We use a helper function is_subsequence(a, b) to check if string a is a subsequence of b.

5. **Edge Cases**

- All strings are the same → return -1.
- One string is a subsequence of all others → return -1.
- Multiple strings, but all are subsequences of longer strings → return -1.

6. **Complexity Analysis**

□ Time Complexity

- Sorting: $O(n \log n)$ where n is the number of strings.
- Checking subsequences:
  - At most $O(n^2 * l)$, where l is the max string length (≤ 10).
  - Each subsequence check is $O(l)$.

➡ Overall: $O(n^2 * l)$

🖫 Space Complexity

- $O(1)$ extra space (no data structures except variables).
- Sorting is in-place (neglecting sorting memory).

7. **🔄 7. Alternative Approaches**

- Brute Force: Generate all subsequences and count frequency → too slow.
- Hash Maps: Count string occurrences, but we still need to check for subsequences.

The current approach is efficient and elegant given the constraints ($n \leq 50$ and len ≤ 10).

8. **Test Cases**

```
sol = Solution()
    # Test case 1
    print(sol.findLUSlength(["aba", "cdc", "eae"]))  # Output: 3

# Test case 2
    print(sol.findLUSlength(["aaa", "aaa", "aa"]))   # Output: -1

# Test case 3 (unique string)
    print(sol.findLUSlength(["abc", "def", "gh"]))   # Output: 3

# Test case 4 (all same)
    print(sol.findLUSlength(["same", "same", "same"])) # Output: -1

# Test case 5 (nested subsequences)
    print(sol.findLUSlength(["a", "ab", "abc", "abcd"])) # Output: 4
```

9. **Final Thoughts**

- This problem tests understanding of subsequences and string comparison.
- Sorting by length optimizes checking for the longest uncommon subsequence early.
- Efficient due to small constraints (max length = 10, max strings = 50).
- Great use case for nested loops with careful conditional logic.