2781. Length of the Longest Valid Substring

Hard | Array | Hash Table | String | Sliding Window   Leetcode Link

Problem Description

You're given a string `word` and an array of strings `forbidden`. The goal is to find the length of the longest substring within `word` that doesn't contain any sequence of characters present in `forbidden`. A substring is simply a contiguous sequence of characters within another string and it can even be an empty string.

To make a string "valid" in the context of this problem, none of its substrings can match any string present in `forbidden`. It's your task to figure out what's the maximum length of such a "valid" substring within the given `word`.

Intuition

The intuition behind the solution revolves around the concept of dynamic window sliding coupled with efficient substring checking. The brute force method of checking every possible substring would be too slow, especially since string `word` can be quite large. Instead, we need to find a quicker way of validating substrings against the `forbidden` list.

We create a sliding window that expands and contracts as needed while scanning the `word` from left to right. The set `s` allows for constant-time checks if a substring is in `forbidden`—this is much faster than repeatedly scanning a list.

Our approach is as follows:

- Start with two pointers `i` and `j`, initializing `i` to the beginning of the word and `j` to increment over each character.
- At every step, check the current window from `j` down to `i` to see if any substring is in `forbidden`. If it is, move `i` past the forbidden substring's start to make a valid window again.
- Keep track of the maximum valid window seen so far as `ans`.

One key optimization lies in the knowledge that the forbidden substrings have a maximum possible length. Thus the inner loop doesn't need to check further back than this maximum length from `j`, which prevents needless comparisons and significantly speeds up the algorithm.

This solution effectively balances between the thoroughness of checking all relevant substrings and the speed of skipping unnecessary checks, achieving the task within acceptable time complexity.

Solution Approach

In the provided solution, a set data structure is utilized to store the `forbidden` strings. The choice of a set is crucial because it provides O(1) lookup time complexity, which means checking if a substring is in `forbidden` is very fast, regardless of the size of `forbidden`.

Here is the step-by-step explanation of the code:

1. The set `s` is created from the `forbidden` array to take advantage of fast lookups.
2. Two pointers are introduced:
   - `i` is initialized to 0, pointing to the beginning of the current valid substring.
   - `j` is used to iterate through each character in `word`.
3. A variable `ans` is used to keep track of the length of the longest valid substring found so far.
4. We iterate over the string `word` with the help of `j`.
   - During each iteration, we look for any potential `forbidden` substrings within a window --- the substring from `[k : j + 1]`, where `k` will iterate backwards from `j` but not further back than 10 places (owing to the maximum length of forbidden substrings) or before `i`, whichever is lesser.
   - The use of `max(j - 10, i - 1)` ensures that we don't check beyond the necessary window backwards from the current character.
5. If a forbidden substring is found, `i` is moved to one character past the start of the forbidden substring. This effectively skips the forbidden part, searching for the next valid starting point.
6. `ans` is updated to be the maximum of its current value or `j - i + 1`, which represents the length of the new valid substring from the current position `j` back to `i`.
7. The iteration continues until the end of `word` is reached, with `ans` being updated as necessary.
8. At the end, `ans` holds the length of the longest valid substring, and it is returned as the solution.

The key algorithmic pattern here is the sliding window, employed to dynamically adjust the substring being inspected. Effective use of data structuring and algorithmic patterns are what make this solution both elegant and efficient.

Example Walkthrough

Let's take an example to understand how this solution works. Suppose we have `word = "abcdefgh"` and `forbidden = ["bc", "fg"]`. We want to find the longest substring of `word` that doesn't contain any sequence of characters from `forbidden`.

We start by creating a set `s` containing `forbidden` strings for quick lookup: `s = {"bc", "fg"}`. We initialize two pointers: `i` at index 0, and `j` which will traverse through `word`.

The `ans` variable starts at 0 to keep track of the length of the longest valid substring found so far. Now, we begin iterating over `word`:

1. With `j` at 0, our window is ["a"], and since it doesn't match any forbidden strings, we move on.

2. With `j` at 1, our window is ["ab"], which is also not forbidden. We continue.

3. With `j` at 2, our window is ["abc"], and checking from the previous 10 characters (or from `i` if less than 10), we find a match with `s` because "bc" is in the window. We update `i` to one position after where "bc" started, so `i` becomes 2.

4. Now `j` moves to 3, and our new window starts from `i` (2) to `j` (3), which is ["cd"]. It is valid, so we continue.

5. With `j` at 4, our window is ["cde"], still valid.

6. With `j` moving up to 5, our window is now ["cdef"], which is still not in `forbidden`.

7. At `j` = 6, our window becomes ["cdefg"], which contains "fg". We update `i` to move past the start of "fg", making `i` = 6.

8. As `j` moves to 7, our window is ["h"], since `i` was just updated to 6. This window is valid.

Throughout the process, we update `ans` to be the maximum length of valid substrings we have found. The sequence "cde" was the longest valid substring before we encountered "fg", with a length of 3. Therefore, `ans` is 3.

In this case, the longest substring of `word` without any `forbidden` sequences would be "cde" which has a length of 3. This is the value we return.

Python Solution

```python
class Solution:
    def longestValidSubstring(self, word: str, forbidden: list[str]) -> int:
        # Create a set from the `forbidden` list for O(1) access time during checks.
        forbidden_set = set(forbidden)

        # Initialize the maximum length of the valid substring (ans) and the starting index (start) of the current substring.
        ans = start = 0

        # Iterate over each character index `end` of the `word`.
        for end in range(len(word)):
            # Check all substrings that end at `end` and do not exceed 10 characters in length.
            # `max(end - 10, start - 1)` ensures we only consider substrings of up to 10 characters
            # and do not reexamine substrings already invalidated by a forbidden word.
            for check_start in range(end, max(end - 10, start - 1), -1):
                # If the substring from `check_start` to `end` (inclusive) is in the forbidden set,
                # update the `start` index to one position after `check_start`
                # and break from the loop to start checking the next end position.
                if word[check_start:end + 1] in forbidden_set:
                    start = check_start + 1
                    break

            # Calculate the length of the current valid substring and update `ans` if it's greater than the previous maximum.
            ans = max(ans, end - start + 1)

        # Return the length of the longest valid substring that doesn't contain a forbidden word.
        return ans

```

Java Solution

```java
class Solution {
    public int longestValidSubstring(String word, List<String> forbiddenSubstrings) {
        // A set to store the forbidden substrings for faster lookup
        Set<String> forbiddenSet = new HashSet<>(forbiddenSubstrings);

        // Initialize variables for the answer and the length of the word
        int maxValidLength = 0;
        int wordLength = word.length();

        // Two pointers for iterating through the string
        // i - start of the current valid substring
        // j - end of the current valid substring
        for (int i = 0, j = 0; j < wordLength; ++j) {
            // Check substrings within the current window from [i,j]
            for (int k = j; k > Math.max(j - 10, i - 1); --k) {
                // If a forbidden substring is found within this window
                if (forbiddenSet.contains(word.substring(k, j + 1))) {
                    // Move the start index after the forbidden substring
                    i = k + 1;
                    break;
                }
            }
            // Calculate the length of the current valid substring
            // And update the maximum if this is the longest so far
            maxValidLength = Math.max(maxValidLength, j - i + 1);
        }

        // Return the length of the longest valid substring found
        return maxValidLength;
    }
}

```

C++ Solution

```cpp
#include <string>
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    /**
     * Finds the longest valid substring that does not contain any forbidden substrings.
     * @param word The main string in which to look for the substring.
     * @param forbidden A vector of forbidden substrings.
     * @return The length of the longest valid substring.
     */
    int longestValidSubstring(string word, vector<string>& forbidden) {
        // Convert the forbidden vector to a set for efficient lookup
        unordered_set<string> forbiddenSet(forbidden.begin(), forbidden.end());

        // Variable to store the answer, i.e., the length of the longest valid substring
        int maxLength = 0;
        // Variable to store the length of the word
        int wordLength = word.size();

        // Two pointers technique to iterate through the string
        // `start` denotes the start index of the current valid substring
        // `end` denotes the end index of the current valid substring
        for (int start = 0, end = 0; end < wordLength; ++end) {
            // Check all possible forbidden substrings within the last 10 characters
            for (int k = end; k > max(end - 10, start - 1); --k) {
                // If a forbidden substring is found, move the start pointer past it
                if (forbiddenSet.count(word.substr(k, end - k + 1))) {
                    start = k + 1;
                    break;
                }
            }
            // Update the maximum length with the length of the current valid substring
            maxLength = max(maxLength, end - start + 1);
        }
        // Return the length of the longest valid substring found
        return maxLength;
    }
};

```

Typescript Solution

```typescript
// Defines a function to find the length of the longest substring without any forbidden words.
function longestValidSubstring(word: string, forbidden: string[]): number {
    // Create a Set to efficiently check if a substring is forbidden.
    const forbiddenSet: Set<string> = new Set(forbidden);
    // Get the length of the word to iterate through.
    const wordLength = word.length;
    // Initialize the answer to store the length of the longest valid substring.
    let maxLength = 0;

    // Initialize two pointers for the sliding window technique.
    // 'start' tracks the start index of the current substring.
    // 'end' tracks the end index of the current substring.
    for (let start = 0, end = 0; end < wordLength; ++end) {
        // Check all possible substrings from the current 'end' to 'start',
        // but we limit the substring length to a maximum of 10 characters.
        for (let k = end; k > Math.max(end - 10, start - 1); --k) {
            // If a forbidden substring is found, move 'start' past the forbidden word,
            // effectively shrinking the current window and continue searching.
            if (forbiddenSet.has(word.substring(k, end + 1))) {
                start = k + 1;
                break;
            }
        }
        // Update the length of the longest valid substring found.
        maxLength = Math.max(maxLength, end - start + 1);
    }
    // Return the length of the longest valid substring.
    return maxLength;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed by going through the nested loops and operations within them. The outer loop runs for `len(word)` times, where 'word' is the input string and its length is denoted by 'n'. The inner loop at worst case runs for a maximum of 10 iterations since it looks back from the current position `j` only up to 10 characters because of `max(j - 10, i - 1)` condition.

Hence, in the worst-case scenario, for each character we might look at the next 10 characters to check if the substring is in the forbidden set. Since checking whether a substring is in a set is O(1) operation due to hashing, the worst case would consider these constant time checks up to 10 times for each character.

Combining both loops, the worst-case time complexity can therefore be simplified to `O(n)`, where each character is checked against a constant number of potential forbidden substrings.

Space Complexity

The space complexity consists of the space required for the set 's' and a few integer variables. Since 's' contains the forbidden substrings, its space complexity will be O(m), where 'm' is the total length of all forbidden substrings combined.

The other variables, 'ans', 'i', and 'j' are constant space, adding O(1) to the total space complexity.

Therefore, the total space complexity of the code is `O(m + 1)` which simplifies to `O(m)`, denoting the space taken up by the set of forbidden substrings.