1961. Check If String Is a Prefix of Array

Easy Array String

Problem Description

The problem gives us two primary inputs: a string `s` and an array of strings `words`. The core task is to determine whether the string `s` can be constructed by concatenating the first `k` strings from the `words` array, where `k` is a positive integer and cannot exceed the length of `words`. If `s` can be created in this way, we refer to it as a *prefix string* of `words`.

Intuition

To approach this problem, our main goal is to check if `s` matches with a concatenation of the beginning subset of `words`. To do that, we can iterate through the `words` array and keep concatenating words until the total length of concatenated words equals the length of `s`. The solution approach is straightforward:

- We maintain a running total of the lengths of the strings we have taken from the `words` array.
- For each word in `words`, we increase this running total by the length of the current word.
- If at any point the running total matches the length of `s`, we then check if the concatenation of the words thus far equals `s`.
- If the concatenation is a match, we return `true` indicating `s` is a prefix string.
- If we traverse all of `words` without finding a match, we return `false`.

Solution Approach

The solution implementation uses a simple algorithmic approach utilizing elementary control structures found in Python:

- We initialize a variable `t` to 0. This variable serves as the running total of lengths of strings from the `words` array that have been concatenated.
- We iterate over each word `w` in the `words` array using a `for` loop, and with each iteration, we add the length of `w` to our running total `t`.
- For each word, after updating `t`, we compare `t` with the length of `s` to check if we've reached or exceeded the length of `s`.
  - If the length `t` is equal to the length of `s`, we concatenate all `words` up to the current `i` index (through slicing `words[: i + 1]`) and compare the concatenated result with `s`.
  - If they match, `s` is indeed a prefix string, and we return `true`.
  - If the lengths match but the strings do not, or if we finish the loop without ever matching lengths, we return `false`.
- The code exploits Python's list slicing and string joining capabilities to achieve this without needing a separate string builder or array.

Here is a step-by-step breakdown of the algorithm:

- Set `t = 0`.
- Loop `i, w in enumerate(words)`:
  1. `t += len(w)` increases total concatenated length.
  2. Check if `len(s) == t` to see if lengths match.
     - If they do, verify string match with `''.join(words[: i + 1]) == s`.
     - If match is true, return `true`.
- If no match is found after the loop completes, return `false`.

This algorithm uses a linear approach, iterating over the array once and stopping early if a match is found. On a complexity level, it operates in O(n) time with respect to the number of words, and potentially O(m) with respect to the length of `s` because of the string comparison step.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have a string `s = "applepenapple"` and an array of strings `words = ["apple", "pen"]`. We want to determine if `s` can be formed by concatenating the entire `words` array.

Here are the steps we would follow:

- We initialize `t` to 0. This is our running total of the lengths of the strings we've concatenated from the `words` array.
- We begin a loop over `words` with `i, w in enumerate(words)`:
  1. For the first iteration, `w = "apple"` and `t` is updated to `t += len(w)` which gives `t = 5`.
  2. We then compare `t` with `len(s)`. Since `len(s) = 13` and `t = 5`, they do not match, so we continue without checking for a string match.
- In the second iteration:
  1. `w = "pen"` and now `t` is updated to `t += len(w)` which results in `t = 8`.
  2. We compare `t` with `len(s)` again. Since `len(s) = 13` and `t = 8`, they still do not match, so we continue.
- Since we've reached the end of the `words` array without `t` equaling `len(s)`, we continue to concatenate the words from the beginning of the `words` array until `t` equals `len(s)`, if possible.
- The concatenated result of `words` is `"applepen"`, which does not match the entire string `s = "applepenapple"`.
- Even if we repeat the `words` array once more (which is not part of this solution but just for understanding), the concatenated result would be `"applepenapplepen"`, which exceeds `s`.
- Since we cannot form `s` by concatenating the `words` array, the final result would be `false`.

In this example, `s` cannot be a prefix string of `words` because the concatenation of all the strings in `words` does not equal `s`, and furthermore, there is remaining substring in `s` that can't be matched by elements within `words`. Thus, following the solution approach, we return `false`.

Solution Implementation

Python

```python
from typing import List  # This line is to import the List type from the typing module

class Solution:
    def isPrefixString(self, s: str, words: List[str]) -> bool:
        # Initialize total_length to 0 to keep track of the combined length of words in "words"
        total_length = 0

        # Iterate over the words list with an index i and word w
        for i, word in enumerate(words):
            # Update total_length by adding the length of the current word
            total_length += len(word)

            # Check if the combined length of words matches the length of s
            if len(s) == total_length:
                # Check if the concatenated words from the begin up to the current word equals s
                # If it matches, our string s is a prefix and we return True
                return ''.join(words[: i + 1]) == s

        # If no prefix match is found after iterating through all words, return False
        return False
```

Java

```java
class Solution {
    public boolean isPrefixString(String s, String[] words) {
        // Initialize a StringBuilder to construct the prefix from words array
        StringBuilder prefix = new StringBuilder();

        // Iterate through each word in the words array
        for (String word : words) {
            // Append the current word to the prefix
            prefix.append(word);

            // Check if the length of the constructed prefix is same as the input string
            if (s.length() == prefix.length()) {
                // Compare the constructed prefix with the input string
                return s.equals(prefix.toString());
            }
        }

        // Return false if no prefix matches the input string
        return false;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to determine if the given string 's' is a prefix string
    // created by concatenating elements from 'words' array.
    bool isPrefixString(string s, vector<string>& words) {
        string currentConcatenation = ""; // This will store the concatenation of words so far.

        // Iterate over the words in the given words vector.
        for (const string& word : words) {
            // Add the current word to our concatenation string.
            currentConcatenation += word;

            // Check if the length of the current concatenation matches the length of 's'.
            // If they match, perform a value comparison to see if they are equal.
            if (currentConcatenation.size() == s.size()) {
                return currentConcatenation == s; // Return true if they're exactly the same.
            }
        }
        // If we exit the loop, the prefix condition has not been met.
        return false;
    }
};
```

TypeScript

```typescript
// Function to determine if the given string 's' is a prefix string
// created by concatenating elements from 'words' array.
function isPrefixString(s: string, words: string[]): boolean {
    let currentConcatenation: string = ""; // This will store the concatenation of words so far.

    // Iterate over the words in the given words array.
    for (const word of words) {
        // Add the current word to our concatenation string.
        currentConcatenation += word;

        // Check if the length of the current concatenation matches the length of 's'.
        // If they match, perform a value comparison to see if they are equal.
        if (currentConcatenation.length === s.length) {
            return currentConcatenation === s; // Return true if they're exactly the same.
        }
    }

    // If we exit the loop, any prefix created doesn't match 's' exactly.
    return false;
}
```

```python
from typing import List  # This line is to import the List type from the typing module

class Solution:
    def isPrefixString(self, s: str, words: List[str]) -> bool:
        # Initialize total_length to 0 to keep track of the combined length of words in "words"
        total_length = 0

        # Iterate over the words list with an index i and word w
        for i, word in enumerate(words):
            # Update total_length by adding the length of the current word
            total_length += len(word)

            # Check if the combined length of words matches the length of s
            if len(s) == total_length:
                # Check if the concatenated words from the begin up to the current word equals s
                # If it matches, our string s is a prefix and we return True
                return ''.join(words[: i + 1]) == s

        # If no prefix match is found after iterating through all words, return False
        return False
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n*k)` where `n` is the number of words in the list and `k` is the average length of the words. This is because in the worst case, the code iterates through each word in the `words` list (`O(n)`) and for each word it adds the length of the word to `t` (`O(k)` since in the worst case the length of the word can be as much as the length of `s`). Moreover, joining the words is another `O(n*k)` because all words might be joined in the worst case (when `s` is exactly the concatenation of all words in the list).

Space Complexity

The space complexity is `O(s)`, where `s` is the length of the input string `s`. In the worst case, when `s` is a prefix string formed by concatenating all of the words in `words`, the space taken by `''.join(words[:i + 1])` will be `O(s)` since it needs to store the entire concatenated string equivalent to `s` in memory temporarily.