1813. Sentence Similarity III

Medium Array Two Pointers String

Leetcode Link

Problem Description

A sentence in this context is a collection of words that are separated by a single space and do not have any spaces at the beginning or end. These words consist solely of uppercase and lowercase English letters. Two sentences are described as "similar" if you can insert any arbitrary sentence (which might even be empty) into one of them such that both sentences are exactly the same in the end. This problem asks us to determine if two given sentences are similar according to this definition.

Intuition

To determine if two sentences are similar, we can look for a common starting sequence of words and a common ending sequence of words between them. If we have these sequences, whatever is left in the middle of one sentence can potentially be the insert needed to make the sentences similar.

We begin by splitting both sentences into two arrays of words. Then, we compare the words at the beginning of both arrays until we find a pair that doesn't match. We then do the same from the end of both arrays. If sufficiently many words from the beginning and end overlap (the sum of the number of matching words from the beginning and end is greater than or equal to the total words in the shorter sentence), it means all the words in the shorter sentence are included in the longer sentence in the right order, and the sentences are similar.

Let's break this down further:

- We split the sentences into words and compare them from the start.
- We keep a count of how many words are the same from the start.
- Next, we compare the sentences from the end.
- We then count how many words are the same from the end.
- If the sum of these two counts is greater than or equal to the length of the shorter sentence, we can conclude that the sentences are indeed similar.

This approach works because we essentially check if the words of the shorter sentence can be found in the longer sentence in the same order, allowing for an arbitrary insertion in the longer one that doesn't disrupt the sequence of the common words.

Solution Approach

The implementation of the solution uses a two-pointer technique and basic list operations provided by Python. The code first splits the given sentences into two lists (`words1` and `words2`) using `split()`, which is a string method that divides a string into a list at each space.

With two pointers (`i` and `j` initialized to `0`), the algorithm operates in two main stages:

1. Check from the beginning of both lists for similar words. Increment pointer `i` as long as `words1[i]` is equal to `words2[i]`. This loop continues until a mismatch is encountered or the end of one of the lists is reached.
2. Check from the end of both lists for similar words. Just like the first stage, increment pointer `j` as long as the last elements of both lists match (`words1[e - 1 - j]` is equal to `words2[n - 1 - j]`). The index calculation here is adjusted for zero-based indexing and goes in reverse because of the backwards check.

The lengths of the lists (`e` for `words1` and `n` for `words2`) are taken into account, and if `words1` is shorter than `words2`, they are swapped. This is to make sure that we are always trying to fit the shorter list into the longer one.

Finally, the condition is checked: if the sum of `i` and `j` is greater than or equal to `n` (the length of the shorter list after a potential swap), the sentences are determined to be similar and `True` is returned; else `False` is returned.

By using the two-pointer technique, we avoid unnecessary checks and manage to achieve the comparison with a linear time complexity relative to the length of the sentences.

Example Walkthrough

To illustrate the solution approach, let's consider the following two sentences:

- Sentence A: `I have a fast car`
- Sentence B: `I have a very fast car today`

First, we need to split both sentences into arrays of words to compare them word by word. So, Sentence A becomes `["I", "have", "a", "fast", "car"]` and Sentence B becomes `["I", "have", "a", "very", "fast", "car", "today"]`.

Next, the algorithm uses two pointers. We'll describe each step:

1. **From the Beginning**: Starting with pointer `i` at index 0, we compare the words at the current index in both sentences.
   - `words1[0]` is `"I"`, and `words2[0]` is also `"I"`. Since they are the same, we increment `i` to 1.
   - This process repeats until `words1[i]` does not equal `words2[i]`.
   - In our case, they match until `i` is 3 (pointing to the word `fast`), and at index 4 they diverge: `words1[4]` is `car` whereas `words2[4]` is `very`.
2. **From the End**: Now, we initialize pointer `j` to 0, and we'll compare the words starting from the end of both sentences.
   - `words1[4 - 0]` (`"car"`) is the same as `words2[6 - 0]` (`"today"`). It's not a match, so `j` remains 0 for now.
   - We then check the next word from the end, so we increment `j` to 1 and compare `words1[4 - 1]` (`"fast"`) with `words2[6 - 1]` (`"car"`).
   - They still don't match, so we increment `j` again.
   - Now, `words1[4 - 2]` (`"a"`) matches with `words2[6 - 2]` (`"fast"`). There's still no match.
   - Continuing this process, we find that `words1[4 - 3]` (`"have"`) matches with `words2[6 - 3]` (`"very"`), and so on. However, for this example, no other matches from the end will be found.

After this process, we have `i` equal to 3 (since 3 words matched from the start) and `j` equal to 0 (since no words matched from the end).

We compare the sum of `i` and `j` to the length of the shorter list, which is `len(words1)` or 5.

- `i + j` is 3 + 0, which is 3.
- Since 3 is less than 5, the condition `i + j >= len(words1)` is not met.

Therefore, the sentences are not similar according to the definition, and the algorithm would return `False` in this case. If instead, B was "I have a fast car today", the result would have been `True`, as all words in A could be found in B with the addition of "today" at the end.

Python Solution

```python
class Solution:
    def areSentences_similar(self, sentence1: str, sentence2: str) -> bool:
        # Split the sentences into lists of words.
        words1, words2 = sentence1.split(), sentence2.split()

        # Determine the length of each list.
        len_words1, len_words2 = len(words1), len(words2)

        # Ensure that words1 is the longer list.
        if len_words1 < len_words2:
            words1, words2 = words2, words1
            len_words1, len_words2 = len_words2, len_words1

        # Initialize two pointers to compare words from the beginning and end.
        start_index = end_index = 0

        # Compare words from the beginning of both lists.
        while start_index < len_words2 and words1[start_index] == words2[start_index]:
            start_index += 1

        # Compare words from the end of both lists.
        while end_index < len_words2 and words1[len_words1 - 1 - end_index] == words2[len_words2 - 1 - end_index]:
            end_index += 1

        # Check if all words have been matched when appending the shorter list at any position in the longer list.
        return start_index + end_index >= len_words2
```

Java Solution

```java
class Solution {
    public boolean areSentencesSimilar(String sentence1, String sentence2) {
        // Split both sentences into arrays of words
        String[] words1 = sentence1.split(" ");
        String[] words2 = sentence2.split(" ");

        // Ensure words1 is the longer array
        if (words1.length < words2.length) {
            String[] temp = words1;
            words1 = words2;
            words2 = temp;
        }

        // Initialize lengths and indices
        int length1 = words1.length; // Length of the longer array
        int length2 = words2.length; // Length of the shorter array
        int startMatchCount = 0; // Count how many words match from the beginning
        int endMatchCount = 0; // Count how many words match from the end

        // Count matching words from the start
        while (startMatchCount < length2 && words1[startMatchCount].equals(words2[startMatchCount])) {
            startMatchCount++;
        }

        // Count matching words from the end
        while (endMatchCount < length2 && words1[length1 - 1 - endMatchCount].equals(words2[length2 - 1 - endMatchCount])) {
            endMatchCount++;
        }

        // Check if the total matching words are at least the number of words in the shorter sentence
        return startMatchCount + endMatchCount >= length2;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Determine if two sentences are similar according to the problem's definition.
    bool areSentencesSimilar(std::string sentence1, std::string sentence2) {
        // Split the sentences into words.
        std::vector<std::string> words1 = split(sentence1, ' ');
        std::vector<std::string> words2 = split(sentence2, ' ');

        // Ensure that words1 is the longer sentence to simplify the logic.
        if (words1.size() < words2.size()) {
            std::swap(words1, words2);
        }

        int length1 = words1.size(); // Length of the longer sentence.
        int length2 = words2.size(); // Length of the shorter sentence.

        int prefixMatch = 0; // Number of matching words from the start.
        int suffixMatch = 0; // Number of matching words from the end.

        // Count the number of matching words from the start of both sentences.
        while (prefixMatch < length2 && words1[prefixMatch] == words2[prefixMatch]) {
            ++prefixMatch;
        }

        // Count the number of matching words from the end of both sentences.
        while (suffixMatch < length2 && words1[length1 - 1 - suffixMatch] == words2[length2 - 1 - suffixMatch]) {
            ++suffixMatch;
        }

        // If the sum of matches is greater than or equal to length2, the sentences are similar.
        return prefixMatch + suffixMatch >= length2;
    }

private:
    // Utility function to split a string by a delimiter, returning a vector of words.
    std::vector<std::string> split(std::string& s, char delimiter) {
        std::stringstream stream(s);
        std::string item;
        std::vector<std::string> result;

        while (std::getline(stream, item, delimiter)) {
            result.emplace_back(item);
        }

        return result;
    }
};
```

Typescript Solution

```typescript
function areSentencesSimilar(sentence1: string, sentence2: string): boolean {
    // Split the sentences into arrays of words.
    const words1 = sentence1.split(' ');
    const words2 = sentence2.split(' ');

    // If the first sentence has fewer words than the second, swap them to standardize the processing.
    if (words1.length < words2.length) {
        return areSentencesSimilar(sentence2, sentence1);
    }

    // Record the lengths of the word arrays.
    const words1Length = words1.length;
    const words2Length = words2.length;

    // Initialize pointers for traversing the words from the start and from the end.
    let forwardIndex = 0;
    let backwardIndex = 0;

    // Move the forward pointer as long as the words are similar from the start.
    while (forwardIndex < words2Length && words1[forwardIndex] === words2[forwardIndex]) {
        forwardIndex++;
    }

    // Move the backward pointer as long as the words are similar from the end.
    while (
        backwardIndex < words2Length &&
        words1[words1Length - 1 - backwardIndex] === words2[words2Length - 1 - backwardIndex]
    ) {
        backwardIndex++;
    }

    // Determine if the combined length of similar words from start and end covers the shorter sentence.
    // Return true if all words from the shorter sentence are covered in the sequence, thus similar.
    return forwardIndex + backwardIndex >= words2Length;
}
```

Time and Space Complexity

The time complexity of the given `areSentencesSimilar` function is `O(L)` where `L` is the sum of the lengths of the two input sentences. This is because the primary computation in the function involves splitting the sentences into words and then iterating over the word lists, which happens linearly relative to the size of the sentences.

The space complexity of the function is also `O(L)` since it creates two arrays `words1` and `words2` to store the words from `sentence1` and `sentence2`, respectively. The size of these arrays is directly proportionate to the length of the input sentences, hence the linear space complexity.