408. Valid Word Abbreviation

Easy Two Pointers String Leetcode Link

Problem Description

The problem presents a method of abbreviating strings by replacing non-adjacent, non-empty substrings with the length of those substrings. The objective is to evaluate whether a given abbreviated string, `abbr`, correctly represents the abbreviation of another string, `word`. It's important to note that a valid abbreviation:

- replaces non-empty and non-adjacent substrings,
- does not have leading zeros in the numerical representations of the lengths,
- is not created by replacing adjacent substrings.

The goal is to determine if the given `abbr` represents a valid abbreviation of the `word`.

Here are some examples:

- `"s10n"` is a valid abbreviation of `"substitution"` because the substring of 10 characters, "ubstitutio", is replaced by its length, without leading zeros.
- `"s55n"` is not valid because it implies adjacent substrings were replaced, which is against the rules.

We are tasked with writing a function that takes `word` and `abbr` and returns `True` if `abbr` is a valid abbreviation of `word`, and `False` otherwise.

Intuition

The solution is implemented by simultaneously iterating through both `word` and `abbr` and checking for matches and valid abbreviations. The approach is as follows:

- Iterate over each character in the `word` using a pointer `i` and a pointer `j` for iterating over the `abbr`.
- If the current characters in `word` and `abbr` match, continue to the next characters of both strings.
- If the current character in `abbr` is a digit, it indicates an abbreviated part of `word`. We then:
  - Check for any leading zeros or if the number is '0', which are not valid.
  - Parse the complete number indicating the length of the abbreviated substring.
  - Skip ahead in `word` by the length of the abbreviated part.
- After the loop:
  - If `i` is at the end of `word` and `j` is at the end of `abbr`, the abbreviation is valid, and we return `True`.
  - If either `i` is not at the end of `word` or `j` is not at the end of `abbr`, the abbreviation is not valid, and we return `False`.

We leverage the facts that letters must match exactly and numbers must accurately represent the length of the skipped substrings in the `word`.

This approach efficiently determines if the abbreviation is valid by inspecting each character only once, resulting in an O(n) time complexity, where n is the length of the `word`.

Solution Approach

The solution uses a straightforward approach based on two-pointer iteration and string manipulation techniques without the need for additional data structures. Let's walk through the implementation and the reasoning behind it:

- Two integer pointers, `i` and `j`, are initialized to `0`. These act as indexes to traverse `word` and `abbr` respectively.
- Two variables, `m` and `n`, hold the lengths of `word` and `abbr`. They help in checking whether we have reached the end of the strings during iteration.

The algorithm then enters a while-loop, with the condition that `i` is less than `m`, letting us loop until the end of `word`:

- The first `if` statement within the loop checks whether `j` has reached the end of `abbr`. If it has, then `abbr` cannot be a valid abbreviation of `word` because there are unmatched characters in `word`, so the function returns `False`.

- If `word[i]` is equal to `abbr[j]`, we have a direct character match, and we advance both pointers.

- If `word[i]` is not equal to `abbr[j]`, the algorithm checks for a number in `abbr` by entering another while-loop which continues as long as `abbr[k]` is a digit. This is determined by the `isdigit()` method, effectively parsing an integer from the abbreviation.

- After exit from the inner while-loop, a substring `t` from `abbr[j]` to `abbr[k]` holds the number.

- The validity of the number is then checked:

  - It should be a digit and not have leading zeros.
  - It should not be `0` since an abbreviation can't represent a substring of `0` length.
- If `t` is valid, the length `int(t)` is added to pointer `i` to skip the matching characters in `word`. Pointer `j` is set to `k` to continue checking `abbr` after the number.

Finally, after the while-loop:

- The function returns `True` only if `i == m` and `j == n`, meaning that both the word and abbreviation were completely matched.

Overall, the solution uses an iterative approach and basic string operations without requiring additional memory, i.e., in O(1) space complexity, and time complexity is O(n), where n is the length of the `word`.

Example Walkthrough

Let's consider a small example to illustrate the solution approach described in the content above. Suppose we have the word `"international"` and the abbreviation `"i11l"`. We want to determine if `"i11l"` is a correct abbreviation for `"international"`.

Here's the step-by-step process of checking the validity of the abbreviation:

1. Initialize two pointer variables, `i` and `j`, to `0`. They will iterate over the characters in `"international"` and `"i11l"`, respectively.

2. Iterate while `i < len("international")`.

   - **Step 1:** Both `i` and `j` point to `'i'`. Since the characters match, we move `i` and `j` forward to `1`.

   - **Step 2:** `j` now points to `'1'`. This signifies an abbreviation. Since `'1'` is a digit and not `'0'`, there are no leading zeros, and so it is initially considered valid.

   - **Step 3:** We continue to move `j` forward to gather the complete number representing the length of the abbreviation. We find that `j` is `'1'` at index `1` and `'1'` at index `2`. This represents the number `11`.

   - **Step 4:** After parsing the complete number `11`, we move the `i` pointer forward in `"international"` by `11` characters, landing it on `'l'`.

   - **Step 5:** We set `j` to the index after the parsed number, which is `3`, pointing to `'l'`.

3. Now, both `i` and `j` point to `'l'` again, which match, so we move both `i` and `j` forward by `1`.

4. At this point, `i == len("international")` and `j == len("i11l")`, indicating that we have reached the end of both the word and the abbreviation. Since all characters have been successfully matched, and the numerical abbreviations have been valid, we return `True`.

So, `"i11l"` is a valid abbreviation for `"international"`.

Python Solution

```python
class Solution:
    def validWordAbbreviation(self, word: str, abbr: str) -> bool:
        # Initialize pointers for word and abbreviation
        word_index = 0
        abbr_index = 0

        # Get lengths of word and abbreviation
        word_length = len(word)
        abbr_length = len(abbr)

        # Loop through the characters in the word
        while word_index < word_length:
            # Check if abbreviation index has become out of bound
            if abbr_index >= abbr_length:
                return False

            # If current characters match, move to the next ones
            if word[word_index] == abbr[abbr_index]:
                word_index += 1
                abbr_index += 1
                continue

            # Check if abbreviation character is not a digit
            if not abbr[abbr_index].isdigit():
                return False

            # Calculate the number representing the skipped characters
            num_start_index = abbr_index
            while abbr_index < abbr_length and abbr[abbr_index].isdigit():
                abbr_index += 1
            num_str = abbr[num_start_index:abbr_index]

            # Leading zero or invalid number check
            if num_str[0] == '0':
                return False

            # Move the word index forward by the number of skipped characters
            word_index += int(num_str)

        # If we've reached the end of both the word and the abbreviation, the abbreviation is valid
        return word_index == word_length and abbr_index == abbr_length

```

Java Solution

```java
class Solution {
    public boolean validWordAbbreviation(String word, String abbreviation) {
        int wordLength = word.length(), abbreviationLength = abbreviation.length();
        int wordIndex = 0, abbreviationIndex = 0;

        // Iterate through the characters of the word
        while (wordIndex < wordLength) {
            // If the abbreviation index exceeds its length, return false
            if (abbreviationIndex >= abbreviationLength) {
                return false;
            }
            // If characters in word and abbreviation match, move to the next character
            if (word.charAt(wordIndex) == abbreviation.charAt(abbreviationIndex)) {
                wordIndex++;
                abbreviationIndex++;
                continue;
            }
            // If the abbreviation character is not a digit, return false
            if (!Character.isDigit(abbreviation.charAt(abbreviationIndex))) {
                return false;
            }
            int start = abbreviationIndex;

            // Find the end of the digit sequence in the abbreviation
            while (start < abbreviationLength && Character.isDigit(abbreviation.charAt(start))) {
                ++start;
            }
            // Get the numerical value for the abbreviation part
            String numString = abbreviation.substring(abbreviationIndex, start);
            // Leading zeroes are not valid, and a substring of "0" is also invalid
            if (abbreviationIndex == start || numString.charAt(0) == '0') {
                return false;
            }
            // Convert the numbers string to the actual number
            int skipCount = Integer.parseInt(numString);

            // Move the word index forward by the numerical value
            wordIndex += skipCount;

            // Set the abbreviation index to the end of the current numerical sequence
            abbreviationIndex = start;
        }
        // Ensure both word and abbreviation have been fully processed
        return wordIndex == wordLength && abbreviationIndex == abbreviationLength;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    bool validWordAbbreviation(string word, string abbr) {
        int wordIndex = 0, abbrIndex = 0; // Initiate indices to iterate over word and abbreviation strings
        int wordLength = word.size(), abbrLength = abbr.size(); // Get lengths of both word and abbreviation

        // Iterate over the entire word to check if it matches with abbreviation
        while (wordIndex < wordLength) {
            if (abbrIndex >= abbrLength) {
                return false; // If abbreviation is shorter than the word part processed, return false
            }

            // If the current characters match, move to the next character in both strings
            if (word[wordIndex] == abbr[abbrIndex]) {
                ++wordIndex;
                ++abbrIndex;
                continue;
            }

            // Find the next non-digit character in abbreviation to extract the numeric part
            int numStart = abbrIndex;
            while (numStart < abbrLength && isdigit(abbr[numStart])) {
                ++numStart;
            }

            // Get the numeric part as a substring
            string numStr = abbr.substr(abbrIndex, numStart - abbrIndex);

            // If there's no numeric part or it starts with a zero, the abbreviation is invalid
            if (numStart == abbrIndex || numStr[0] == '0') {
                return false;
            }

            // Convert numeric string to integer
            int num = stoi(numStr);

            // Advance the word index by the numeric value and update the abbreviation index
            wordIndex += num;
            abbrIndex = numStart;
        }

        // After processing, both indices should be at the end of their respective strings
        return wordIndex == wordLength && abbrIndex == abbrLength;
    }
};

```

Typescript Solution

```typescript
let wordIndex: number = 0;
let abbrIndex: number = 0;
let wordLength: number;
let abbrLength: number;

function isValidWordAbbreviation(word: string, abbr: string): boolean {
    wordIndex = 0;
    abbrIndex = 0;
    wordLength = word.length;
    abbrLength = abbr.length;

    while (wordIndex < wordLength) {
        if (abbrIndex >= abbrLength) {
            return false;
        }

        if (word[wordIndex] === abbr[abbrIndex]) {
            wordIndex++;
            abbrIndex++;
            continue;
        }

        let numStart = abbrIndex;

        while (numStart < abbrLength && isDigit(abbr[numStart])) {
            numStart++;
        }

        let numStr = abbr.substring(abbrIndex, numStart);

        if (numStart === abbrIndex || numStr.startsWith('0')) {
            return false;
        }

        let num = parseInt(numStr, 10);

        wordIndex += num;
        abbrIndex = numStart;
    }

    return wordIndex === wordLength && abbrIndex === abbrLength;
}

function isDigit(char: string): boolean {
    return !isNaN(parseInt(char, 10));
}

```

Time and Space Complexity

Time Complexity

The provided algorithm iterates over each character of the `word` and `abbr` strings once. On each iteration, while checking if a part of `abbr` is a number, it may iterate through consecutive digits. However, each character in both strings is looked at mostly once because, after processing the digits, the index jumps from the beginning to the end of the numerical sequence.

Therefore, the time complexity of the algorithm can be represented as `O(m + n)`, where `m` is the length of `word` and `n` is the length of `abbr`.

Space Complexity

The algorithm uses a constant amount of extra space for variables `i`, `j`, `k`, `t`, `m`, and `n`. It doesn't utilize any additional structures dependent on the input sizes.

Thus, the space complexity is `O(1)`, which represents constant space.