151. Reverse Words in a String

Medium Two Pointers String

Leetcode Link

Problem Description

In this problem, we are presented with a string `s` that consists of words separated by spaces. Our objective is to reverse the order of these words and return the resulting string with the words placed in opposite order from that of the input. A word is defined as a sequence of non-space characters, meaning that punctuation and letters are considered part of a word but spaces are not. Additionally, the given string `s` could have leading or trailing spaces and could also contain multiple spaces between words.

The crux of the problem is to treat the string as a sequence of words rather than individual characters, thus seeing the string as a list where each element is a word. After identifying the words, we need to reverse this list and then reconstruct the string from these reversed words. Importantly, the result must not contain any extra spaces, so only a single space should separate the words, and no leading or trailing spaces should be included.

Intuition

The first solution approach involves using the built-in functions of the language, in this case, Python. Since Python has powerful string manipulation capabilities, we can solve the problem in a few succinct steps. We start by using `split()` on the input string `s`, which splits the string into a list of words based on spaces while automatically removing any excess spaces. We then reverse this list of words using the `reversed()` function. Finally, we join these reversed words back into a string using the `join()` function, with a space as the separator. This method is straightforward and efficient, leveraging Python's abilities to handle the reversing and joining of the words with minimal code.

Another approach could use two pointers to iterate over the string and identify the words. This is a bit more manual but doesn't rely on Python's built-in functions as heavily. We would advance the pointers to find the beginning and end of each word, add that word to a list, and once we've gone through the whole string and captured all the words, we would reverse the list of words. We then join these words into a final result string with single spaces between them.

Both methods aim to manipulate the words as a sequence and then reverse that sequence to construct the desired output. Each method has its nuances, but they both focus on the core idea of treating the string as a list of words for the purpose of reversing the order.

Solution Approach

The reference solution approach provides two potential methods for solving the problem, both involving different use of algorithms, data structures, and language features.

Solution 1: Use Language Built-in Functions

We start by using Python's `split()` function, which will iterate through the input string `s` and break it into a list of words. This function intelligently ignores any additional spaces beyond the first, so if there is more than one space between words or there are leading and trailing spaces, these won't affect the splitting and won't appear in the final list of words.

Once we have a list of words, we use the `reversed()` function, which takes in a sequence and returns an iterator that accesses the given sequence in the reverse order. This is where the reversal of the word order takes place. Since `reversed()` returns an iterator, we pass it to the `join()` function which concatenates them into a single string with a space as the separator. This gives us the final output.

The time complexity for this approach is `O(n)` because `split()` and `join()` both require a pass through the string or the list of words (which will collectively have a length equivalent to `n`, where `n` is the length of the string). The space complexity is also `O(n)` as we save the list of words separately from the original string.

Solution 2: Two Pointers

While not explicitly implemented here, the two-pointer approach would involve initializing two pointers, usually named `i` and `j`, which would handle the traversal of the string to identify the start and end of each word. The key steps in this approach would be:

- Initialize `i` and `j` to the start of the string.
- Increment `j` to find the end of a word, which is indicated by a space or the end of the string.
- Extract the word and add it to a result list.
- Reset `i` to `j + 1` and find the next word.
- After traversing the entire string and capturing all words into the result list, reverse the list.
- Join the reverse list of words as in the first approach.

The two-pointer technique gives you precise control over the traversal and extraction of words, and efficiently manages spaces. It's a common pattern used in string manipulation problems where you need to parse through and process sections of strings based on certain criteria (like non-space characters).

In either case, the goal is the same: to rearrange the words in the string in reverse order while managing and minimizing spaces appropriately. The tools and approaches may vary depending on language features or personal preference, but the underlying logic of word reversal remains central.

Example Walkthrough

Let's use the sentence `"Hello World from LeetCode"` to illustrate the solution approach using language built-in functions.

1. We start by applying the `split()` function to our example string. This turns our string into a list of words by breaking it at spaces.

```
1 Input string: "Hello World from LeetCode"
2 After split() function: ["Hello", "World", "from", "LeetCode"]
```

2. Next, we apply the `reversed()` function to the list of words. This returns an iterator that will go through the words in reverse order.

```
1 Using reversed() function on our list gives us: ["LeetCode", "from", "World", "Hello"]
```

3. Finally, we use the `join()` function with a space as a separator to turn our list of reversed words back into a single string.

```
1 Result after join() function: "LeetCode from World Hello"
```

By following these steps, we can see that our input string has been transformed such that the words are in the opposite order, and our sentence is grammatically correct without any leading, trailing, or excess in-between spaces.

Now suppose we were to take the two-pointer approach on the example string `"Hello World from LeetCode"`, the steps would look something like this:

1. Initialize the pointers. In this case, let's call them `start` and `end`.
2. Move `end` forward until it hits a space or the end of the string which signifies the end of a word.
3. Take the word from `start` to `end` and add it to a list.

```
1 After the first iteration:
2 "Hello" is captured and added to the list.
3 The list of words: ["Hello"]
```

4. Advance `end` to skip any spaces and set `start` to the `end`'s new location.
5. Repeat steps 2-4 until the end of the string is reached.

```
1 After completing the iterations with capturing:
2 The list of words: ["Hello","World","from","LeetCode"]
```

6. Reverse the list of collected words.

```
1 Reversing the list:
2 The list of words: ["LeetCode", "from", "World", "Hello"]
```

7. Join the reversed list with single spaces into a final result string.

```
1 Final result: "LeetCode from World Hello"
```

This example clarifies how both approaches achieve the same result but through different stages of string and list manipulation.

Python Solution

```python
class Solution:
    def reverseWords(self, s: str) -> str:
        # Split the input string on spaces to get a list of words
        words = s.split()

        # Reverse the list of words
        reversed_words = reversed(words)

        # Join the reversed list of words back into a string with spaces
        reversed_sentence = ' '.join(reversed_words)

        # Return the reversed sentence
        return reversed_sentence
```

Java Solution

```java
class Solution {

    // Method to reverse the words in a given string 's'
    public String reverseWords(String s) {
        // Trim the input string to remove leading and trailing whitespaces
        // and split it into an array of words based on one or more whitespace characters
        String[] wordsArray = s.trim().split("\\s+");

        // Convert the array of words into a list for easy manipulation
        List<String> wordsList = new ArrayList<String>(Arrays.asList(wordsArray));

        // Reverse the order of the words in the list
        Collections.reverse(wordsList);

        // Join the reversed list of words into a single string separated by a single space
        String reversed = String.join(" ", wordsList);

        // Return the reversed string
        return reversed;
    }
}
```

C++ Solution

```cpp
#include <algorithm>    // Include algorithm header for the std::reverse function
#include <string>       // Include string header for using the std::string class

class Solution {
public:
    // This method reverses the words in the string and trims any extra spaces.
    std::string reverseWords(std::string s) {
        int start = 0;                  // Start index of the word
        int end = 0;                    // Index used to copy characters
        int length = s.size();          // Total size of the string

        while (start < length) {
            // Skip any spaces at the beginning of the current segment.
            while (start < length && s[start] == ' ') {
                ++start;
            }

            if (start < length) {
                // Put a space before the next word if it's not the first word.
                if (end > 0) {
                    s[end++] = ' ';
                }
                int tempStart = start;
                // Copy the next word.
                while (tempStart < length && s[tempStart] != ' ') {
                    s[end++] = s[tempStart++];
                }
                // Reverse the word that was just copied.
                std::reverse(s.begin() + end - (tempStart - start), s.begin() + end);
                // Move start index to the next segment of the string.
                start = tempStart;
            }
        }

        // Erase any trailing spaces from the string.
        s.erase(s.begin() + end, s.end());
        // Reverse the entire string to put the words in the original order.
        std::reverse(s.begin(), s.end());

        return s;  // Return the modified string.
    }
};
```

Typescript Solution

```typescript
// Reverses the order of words in a given string.
function reverseWordsInputString(inputString: string): string {
    // Trim the input string to remove leading and trailing whitespaces.
    const trimmedString: string = inputString.trim();

    // Split the trimmed string into an array of words using regular expression to match one or more spaces.
    const wordsArray: string[] = trimmedString.split(/\s+/);

    // Reverse the array of words to get the words in reverse order.
    const reversedWordsArray: string[] = wordsArray.reverse();

    // Join the reversed words array into a single string, separated by a single space.
    const reversedString: string = reversedWordsArray.join(' ');

    // Return the reversed string.
    return reversedString;
}
```

Time and Space Complexity

The given Python function `reverseWords` takes a string `s` and reverses the order of the words within it.

Time Complexity:

The time complexity of the function is `O(n)`. This is because each operation within the function has a linear time complexity related to the length of the string `n`. Here's the breakdown:

- `s.split()`: This operation traverses the string once and splits it based on spaces, taking `O(n)` time.
- `reversed(...)`: The `reversed` function takes an iterable and returns an iterator that goes through the elements in reverse order. This is a linear operation in the number of items to reverse, but since it's just an iterator, the act of reversing itself doesn't consume time for a list, which would rather be encountered during iteration.
- `' '.join(...)`: Joining the words back together involves concatenating them with a space in between, which will also take `O(n)` time because it has to combine all characters back into a single string of approximately the same length as the input.

Since these operations are performed sequentially, the time complexity does not exceed `O(n)`.

Space Complexity:

The space complexity is `O(n)` as well. This arises from several factors:

- `s.split()`: This creates a list of all words, which, in the worst case, would be roughly `n/2` (if the string was all single characters separated by spaces), so this consumes `O(n)` space.
- The list produced by `reversed(...)`: When used with `join()`, the reversed iterator is realized into a list in memory, therefore an additional `O(n)` space is necessary.
- The output string: Since we're creating a new string of approximately the same length as the input, this also requires `O(n)` space.

Since we need to store the intermediate word list and the final string, `O(n)` space is the overall space requirement for this function.