# 186. Reverse Words in a String II

Medium Two Pointers String

Leetcode Link

## Problem Description

The problem provided is one of reversing the order of words within a character array `s`. Each word within the array is defined as a contiguous sequence of characters without spaces, and each word is separated by a single space. The key constraint is that the reversal of the word order must be done in-place, which means the solution cannot use additional memory to store a new array or other data structure.

## Intuition

To solve this problem, the intuition is to think about the desired end state: we want the words in reverse order, but each word itself should remain in its original order. An efficient way to achieve this is by applying a two-step process:

1. Reverse each individual word.
2. Reverse the entire character array.

Step 1 ensures that when we perform Step 2, each word will appear in the correct order since they were individually reversed in the first step.

Implementing Step 1 involves iterating over the character array and reversing characters within each word boundary, defined by spaces. When a space is encountered, it signals the end of the current word, prompting a reversal of all characters from the start of that word to the character immediately before the space.

For the last word, since there's no trailing space, a check is needed to trigger the reversal when reaching the end of the array.

After all words are individually reversed, Step 2 simply reverses the entire array from start to end. As each word is already correctly ordered internally, this step places the words in the reverse order.

The solution employs helper method `reverse` that takes a subsection of the array `s` between indices `i` and `j`, swapping the elements at `i` and `j`, then moving `i` forward and `j` backward, repeating this process until the subsection is completely reversed.

## Solution Approach

The solution approach follows a systematic process:

1. **Initialize Pointers**: We start by initializing two pointers, `i` and `j`, along with `n` that holds the length of the character array `s`. Here, `i` will serve as the start index of a word, and `j` will iterate through the array to find the end of a word.
2. **Iterate Through `s`**: We begin iterating over the array with `j`. The loop continues until `j` has reached the end of the array.
3. **Reverse Individual Words**: As `j` encounters a space, we realize that it marks the end of a word. At this point, we call the `reverse` helper function with the current start index `i` and the end index `j - 1`. After the reversal of the current word, we move `i` past the space to the beginning of the next word (`i = j + 1`).
   - The `reverse` helper function takes two indices and reverses the subsection of `s` between these indices. It does this by swapping the elements at `i` and `j`, then moves `i` forward and `j` backward, repeating the process until `i` meets or passes `j`.
4. **Edge Case for Last Word**: Since there's no trailing space after the last word, the condition `j == n - 1` checks if `j` has reached the end of the array, signaling the end of the last word, and a reversal is performed for this last word segment.
5. **Reverse Entire Array**: Once all words are reversed, the final step is to reverse the entire character array. This is again done by calling the `reverse` function with the indices `0` and `n - 1`.

This algorithm does not use any extra space and operates entirely in-place, modifying the original array to achieve the desired result. The data structure used is simply the input array, and the algorithm leverages the two-pointer technique to identify and reverse each word, followed by the reversal of the entire array. The reversing of elements within the array is based on the classic pattern of using a temporary variable to swap two elements, a fundamental operation in many sorting and reversing algorithms.

### Example Walkthrough

Let's use a simple example to illustrate the solution approach. Suppose we have an array `s` representing the sentence: `"the sky is blue"`

The initial state of the array `s`:

```
1 ['t', 'h', 'e', ' ', 's', 'k', 'y', ' ', 'i', 's', ' ', 'b', 'l', 'u', 'e']
```

Now, let's walk through the process:

1. **Initialize Pointers**:
   - We initialize a pointer `i` at `0` and `n` as the length of the array `s`, which is `15`.
2. **Iterate Through `s`**:
   - We start iterating forward through the array with a pointer `j`. Initially, `j` is also at `0`.
3. **Reverse Individual Words**:
   - As `j` moves forward, it encounters the first space at `j = 3`. This means the first word `"the"` (from `i = 0` to `j - 1 = 2`) needs to be reversed. Using the `reverse` helper function, the array now looks like this:
     ```
     1 ['e', 'h', 't', ' ', 's', 'k', 'y', ' ', 'i', 's', ' ', 'b', 'l', 'u', 'e']
     ```
   - `i` is then updated to `j + 1`, which is `4`, the start of the next word `"sky"`.
4. **Continue Iterating and Reversing Words**:
   - This process is repeated for the remaining words. After reversing `"sky"` and `"is"`, the array looks like:
     ```
     1 ['e', 'h', 't', ' ', 'y', 'k', 's', ' ', 's', 'i', ' ', 'b', 'l', 'u', 'e']
     ```
   - When `j` reaches the end of the array after `"blue"`, `j = n - 1`, it triggers the last reversal from `i = 11` to `j = 14`, resulting in:
     ```
     1 ['e', 'h', 't', ' ', 'y', 'k', 's', ' ', 's', 'i', ' ', 'e', 'u', 'l', 'b']
     ```
5. **Reverse Entire Array**:
   - Finally, we reverse the entire array from `0` to `n - 1`. After this, the array `s` looks like:
     ```
     1 ['b', 'l', 'u', 'e', ' ', 'i', 's', ' ', 's', 'k', 'y', ' ', 't', 'h', 'e']
     ```

Now, the character array `s` correctly represents the sentence `"blue is sky the"`, with the words in reverse order, and each word internally in the correct order. All this has been achieved in-place, without using any additional memory for a new array or data structure.

## Python Solution

```python
class Solution:
    def reverseWords(self, s: List[str]) -> None:
        """
        This method reverses the words in the input list in-place.

        Parameters:
        s (List[str]): A list of characters representing a string with words
                       separated by single spaces.
        """

        def reverse_partial(section: List[str], start: int, end: int) -> None:
            """
            Helper function that reverses a substring of the list in-place.

            Parameters:
            section (List[str]): The list of characters which substring to be reversed
            start (int): The starting index of the substring
            end (int): The ending index of the substring
            """
            while start < end:
                section[start], section[end] = section[end], section[start]
                start += 1
                end -= 1

        length = len(s)
        # Initialize the starting index of the current word
        word_start = 0
        # Traverse the list of characters
        for idx in range(length):
            # If a space is found, reverse the word before the space
            if s[idx] == ' ':
                reverse_partial(s, word_start, idx - 1)
                # Update the starting index for the next word
                word_start = idx + 1
            # If end of the list is reached, reverse the last word
            elif idx == length - 1:
                reverse_partial(s, word_start, idx)

        # Reverse the whole modified list to get words in correct order
        reverse_partial(s, 0, length - 1)

```

## Java Solution

```java
class Solution {

    // Function to reverse words in place within a character array
    public void reverseWords(char[] str) {
        int n = str.length;

        // First, reverse each word in the array
        for (int start = 0, end = 0; end < n; ++end) {
            if (str[end] == ' ') {
                // When we find a space, reverse the previous word
                reverse(str, start, end - 1);
                // Update start to the start of the next word
                start = end + 1;
            } else if (end == n - 1) {
                // If this is the end of the last word, reverse it
                reverse(str, start, end);
            }
        }

        // After all words are reversed, reverse the entire array to put words into the correct order
        reverse(str, 0, n - 1);
    }

    // Helper function to reverse a section of the character array from index i to j
    private void reverse(char[] str, int i, int j) {
        // Swap characters from the starting and ending indices until they meet in the middle
        while (i < j) {
            char temp = str[i];
            str[i] = str[j];
            str[j] = temp;
            i++;
            j--;
        }
    }
}
```

## C++ Solution

```cpp
#include <vector>
#include <algorithm>  // for std::swap

class Solution {
public:
    // Function to reverse words in a given character array.
    void reverseWords(std::vector<char>& s) {
        int n = s.size(); // Get the size of the character array
        // Iterate through the array to find words and reverse them
        for (int start = 0, end = 0; end < n; ++end) {
            if (s[end] == ' ') {
                // A word is found, reverse it
                reverse(s, start, end - 1);
                // Update 'start' to the next word's starting index
                start = end + 1;
            } else if (end == n - 1) {
                // Last word is found, reverse it
                reverse(s, start, end);
            }
        }
        // After reversing all the words individually, reverse the entire array to get the final result
        reverse(s, 0, n - 1);
    }

    // Helper function to reverse characters in the array between indices i and j.
    void reverse(std::vector<char>& s, int i, int j) {
        // Swap characters from both ends moving towards the center
        while (i < j) {
            std::swap(s[i], s[j]);
            ++i;
            --j;
        }
    }
};
```

## Typescript Solution

```typescript
// Function to reverse a portion of the array between indices start and end.
function reverse(arr: string[], start: number, end: number): void {
    while (start < end) {
        // Swap elements at indices start and end.
        let temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

// Function to reverse the words in a given character array.
function reverseWords(s: string[]): void {
    const n: number = s.length; // Get the size of the character array

    // Iterate through the array to find words and reverse them.
    let start: number = 0;
    for (let end = 0; end <= n; end++) {
        // Check if the current character is a space or if we're at the end of the array.
        if (s[end] === ' ' || end === n - 1) {
            // Calculate the position to end the reversal (consider the last word case).
            let reverseEnd = s[end] === ' ' ? end - 1 : end;
            // Reverse the current word.
            reverse(s, start, reverseEnd);
            // Update 'start' to the next word's starting index.
            start = end + 1;
        }
    }

    // After reversing all the words individually, reverse the entire array to get the final result.
    reverse(s, 0, n - 1);
}

```

## Time and Space Complexity

The given Python code is designed to reverse the words in a string in-place. Here's the analysis of its complexity:

**Time Complexity:**

To analyze the time complexity, we observe each part of the code:

- The `reverse` function is a part of the total operation, which is called for each word and once for the entire string. It takes $O(k)$ time for each word, where $k$ is the length of that word, and $O(n)$ for the entire string, where $n$ is the total length of the input string.
- The `while` loop runs through each character in the string, which takes $O(n)$ time.

Combining these, each character is involved in two operations: once in the main loop, and once when its word is reversed. Including the final reversal of the entire string, we have:

- For each word of length $k$: $O(k)$
- For the entire string: $O(n)$

Since the sum of the lengths $k$ of all words is $n$, we can order these operations as $O(n) + O(n) = O(2n)$. However, in Big O notation, we would drop the constant to simplify the expression to $O(n)$.

**Space Complexity:**

- No additional space is needed except for a few variables ($i$, $j$, $n$), since the reversal is done in place.
- The `reverse` function does not use any additional space and is performed in place.

Hence, the space complexity is $O(1)$, which indicates constant space usage.

In conclusion, the code has a time complexity of $O(n)$ and a space complexity of $O(1)$.