917. Reverse Only Letters

Easy Two Pointers String

Problem Description

The task is to reverse a string `s`, but with specific constraints. Only the English letters (both lowercase and uppercase) in the string should be reversed; all other characters should stay in the same place as they were in the original string.

To illustrate, if we are given a string like "a-bC-dEf-ghIj", we want to reverse only the letters to get "j-Ih-gfE-dCba", keeping the hyphens (`-`) in their original positions.

Intuition

To address the given problem, we consider a two-pointer approach. We start with two indexes: one at the beginning of the string (`i`) and one at the end (`j`).

We then move the two indexes towards the center of the string, pausing each time we approach a non-letter character and skipping over it.

When both `i` and `j` point to English letters, we swap them. We continue this process, incrementing `i` and decrementing `j`, until `i` is no longer less than `j`.

This approach ensures that the letters are reversed in place while non-letter characters remain untouched, thus satisfying both conditions set by the problem constraints.

Solution Approach

The solution employs a two-pointer technique, which is often used when we want to traverse an array or string from both ends and possibly meet in the middle. Below is the breakdown of the implementation steps:

1. First, we convert the string `s` into a list because strings in Python are immutable, and we want to be able to swap letters in place.
2. We then initialize two pointers, `i` at the beginning of the list (`0`) and `j` at the end (`len(s) - 1`).
3. We use a `while` loop to iterate over the list as long as `i` is less than `j`.
4. Inside the loop, we use two more `while` loops to move the `i` pointer forward and the `j` pointer backward until they point to English letters. We use the `isalpha()` method to check if a character is an English letter:
   - The first inner `while` loop increments `i` if `s[i]` is not a letter.
   - The second inner `while` loop decrements `j` if `s[j]` is not a letter.
5. After both pointers `i` and `j` point to letters, we swap the characters at these positions.
6. We then increment `i` and decrement `j` to continue the traversal.
7. Once the `while` loop condition `i < j` is no longer satisfied (meaning we have either completed traversing the list or both pointers have met or crossed), we exit the loop.
8. Finally, we use `''.join(s)` to convert the list back into a string and return it as the final output. This string now contains all non-letter characters in their original positions, with the letters reversed relative to their positions in the original string.

This approach is efficient because it only requires a single pass through the string, with a time complexity of O(n) where n is the length of the string. The space complexity is also O(n) due to the conversion of the string to a list, which is necessary for in-place swaps.

Example Walkthrough

Let's go through the given solution approach using a shorter example string: "Ab3c-dE".

1. Convert the string `s` into a list: `["A", "b", "3", "c", "-", "d", "E"]`
2. Initialize our two pointers:
   - `i` starts at index `0` (pointing to "A")
   - `j` starts at index `6` (the last index, pointing to "E")
3. We enter the `while` loop since `i (0)` is less than `j (6)`.
4. We start the inner loops to increment `i` and decrement `j` while skipping non-letter characters:
   - `i` is at index `0`, pointing to "A", which is a letter, so we don't move it.
   - `j` is at index `6`, pointing to "E", which is also a letter, so we don't move it.
5. Both pointers are at English letters, so we swap them.
   - The list now looks like this: `["E", "b", "3", "c", "-", "d", "A"]`
6. Increment `i` to `1` and decrement `j` to `5`.
7. Now, `i` is pointing to "b" and `j` is pointing to "d".
8. Repeat steps 4-6:
   - Both `i` and `j` point to letters again, so we swap:
     - The list after the swap: `["E", "d", "3", "c", "-", "b", "A"]`
   - Increment `i` to `2` and decrement `j` to `4`.
9. Now `i` is pointing to "3" which is not a letter, so `i` moves forward to index `3` (pointing to "c").
10. For `j`, it is pointing to "-", which is not a letter, so `j` moves backward to index `3`.
    - Both `i` and `j` are now pointing to the same position, so there's no need for further swaps.
11. Since `i` is no longer less than `j`, the `while` loop condition is not satisfied. We exit the loop.
12. Finally, we convert the list back to a string: `''.join(s)` gives us "Ed3c-bA"

The final output "Ed3c-bA" shows that the letters have been reversed while the non-letter characters ("3" and "-") are in their original positions as expected.

Solution Implementation

Python

```python
class Solution:
    def reverseOnlyLetters(self, string: str) -> str:
        # Convert the input string into a list of characters for easy manipulation
        char_list = list(string)
        # Initialize two pointers, one at the beginning and one at the end of the char_list
        left_index, right_index = 0, len(char_list) - 1

        # Loop until the two pointers meet or pass each other
        while left_index < right_index:
            # Move the left_index forward if the current character is not a letter
            while left_index < right_index and not char_list[left_index].isalpha():
                left_index += 1
            # Move the right_index backward if the current character is not a letter
            while left_index < right_index and not char_list[right_index].isalpha():
                right_index -= 1
            # If both the current characters are letters, swap them
            if left_index < right_index:
                char_list[left_index], char_list[right_index] = char_list[right_index], char_list[left_index]
                # Move both pointers closer towards the center
                left_index, right_index = left_index + 1, right_index - 1

        # Join the list of characters back into a string and return it
        return ''.join(char_list)
```

Java

```java
class Solution {
    public String reverseOnlyLetters(String str) {
        // Convert the input string to a character array for easier manipulation.
        char[] characters = str.toCharArray();

        // Initialize two pointers.
        int left = 0; // The beginning of the string
        int right = str.length() - 1; // The end of the string

        // Use a while loop to iterate over the character array until the two pointers meet.
        while (left < right) {
            // Move the left pointer to the right as long as the current character isn't a letter.
            while (left < right && !Character.isLetter(characters[left])) {
                left++;
            }

            // Move the right pointer to the left as long as the current character isn't a letter.
            while (left < right && !Character.isLetter(characters[right])) {
                right--;
            }

            // Once both pointers are at letters, swap the characters.
            if (left < right) {
                char temp = characters[left];
                characters[left] = characters[right];
                characters[right] = temp;

                // Move both pointers towards the center.
                left++;
                right--;
            }
        }

        // Convert the manipulated character array back to a string and return it.
        return new String(characters);
    }
}
```

C++

```cpp
#include <string> // Include necessary header
using namespace std;

class Solution {
public:
    // Function to reverse only the letters in a string, leaving other characters in place
    string reverseOnlyLetters(string str) {
        int left = 0; // Initialize left pointer
        int right = str.size() - 1; // Initialize right pointer

        // Iterate over the string with two pointers from both ends
        while (left < right) {
            // Move left pointer to the right as long as it points to a non-letter
            while (left < right && !isalpha(str[left])) {
                ++left;
            }

            // Move right pointer to the left as long as it points to a non-letter
            while (left < right && !isalpha(str[right])) {
                --right;
            }

            // If both pointers are at letters, swap the letters and move pointers towards the center
            if (left < right) {
                swap(str[left], str[right]);
                ++left;
                --right;
            }
        }

        // Return the modified string with letters reversed
        return str;
    }
};
```

TypeScript

```typescript
function reverseOnlyLetters(s: string): string {
    const stringLength: number = s.length; // Length of the input string
    let leftIndex: number = 0;             // Initialize left pointer
    let rightIndex: number = stringLength - 1; // Initialize right pointer
    let reversedArray = [...s];            // Convert string to array for easy manipulation

    // Loop through the array to reverse only the letters
    while (leftIndex < rightIndex) {
        // Decrement left pointer if current character is not a letter, until it points to a letter
        while (!/[a-zA-Z]/.test(reversedArray[leftIndex]) && leftIndex < rightIndex) {
            leftIndex++;
        }

        // Decrement right pointer if current character is not a letter, until it points to a letter
        while (!/[a-zA-Z]/.test(reversedArray[rightIndex]) && leftIndex < rightIndex) {
            rightIndex--;
        }

        // Swap the letters at leftIndex and rightIndex
        [reversedArray[leftIndex], reversedArray[rightIndex]] = [reversedArray[rightIndex], reversedArray[leftIndex]];

        // Move pointers towards the center
        leftIndex++;
        rightIndex--;
    }

    // Join the array back into a string and return the result
    return reversedArray.join('');
}
```

```python
class Solution:
    def reverseOnlyLetters(self, string: str) -> str:
        # Convert the input string into a list of characters for easy manipulation
        char_list = list(string)
        # Initialize two pointers, one at the beginning and one at the end of the char_list
        left_index, right_index = 0, len(char_list) - 1

        # Loop until the two pointers meet or pass each other
        while left_index < right_index:
            # Move the left_index forward if the current character is not a letter
            while left_index < right_index and not char_list[left_index].isalpha():
                left_index += 1
            # Move the right_index backward if the current character is not a letter
            while left_index < right_index and not char_list[right_index].isalpha():
                right_index -= 1
            # If both the current characters are letters, swap them
            if left_index < right_index:
                char_list[left_index], char_list[right_index] = char_list[right_index], char_list[left_index]
                # Move both pointers closer towards the center
                left_index, right_index = left_index + 1, right_index - 1

        # Join the list of characters back into a string and return it
        return ''.join(char_list)
```

Time and Space Complexity

The given Python code defines a function `reverseOnlyLetters` that takes a string `s`, reverses only the alphabetic characters in it while leaving the other characters in their original positions, and then returns the modified string.

Time Complexity:

The time complexity of this function is `O(n)`, where `n` is the length of the string `s`. Here's the breakdown:

- The function initially converts the string into a list, which takes `O(n)` time.
- The while loop uses a two-pointer approach, with `i` starting at the beginning of the string and `j` at the end. The loop runs until `i` is less than `j`. Within the loop, there are operations of checking whether a character is alphabetical (`s[i].isalpha()` and `s[j].isalpha()`) and swapping the characters if both are letters. Both of these operations are `O(1)`.
- The loop will iterate at most `n/2` times because once `i` meets `j` in the middle, the process is complete. Each iteration has constant work (checking and swapping), thus the total time for the loop is `O(n/2)`, which simplifies to `O(n)`.

Therefore, combining the initial list conversion and the while loop, the overall time complexity remains `O(n)`.

Space Complexity:

The space complexity of the function is also `O(n)` for the following reasons:

- The function allocates space for a list of characters from the original string `s`, which is `O(n)` space.
- The space for the pointers `i` and `j` is negligible (constant space, or `O(1)`).
- The list is converted back to a string at the end, but this does not require extra space proportional to `n` as the list is transformed in place.

Hence, the additional space required is proportional to the size of the input, leading to a space complexity of `O(n)`.