344. Reverse String

Easy | Two Pointers | String

Problem Description

The problem asks for a function that takes an array of characters `s` and reverses it. It's important to note that the reversal must be done in-place, which means the original array must be modified without using any extra space for another array. The challenge specifically requires that the solution adhere to `O(1)` extra memory usage, which means you cannot allocate additional memory that scales with the input size.

Intuition

The intuition behind the solution involves the realization that to reverse an array, you can swap elements from the beginning and the end moving inward. Picture the string as a line of people standing shoulder to shoulder, and you want them to face the other way. Instead of asking everyone to turn around, which would be like creating a new reversed array, you decide to have the person on one end swap places with the person on the other end, and so on, until you reach the middle.

That is exactly what the given solution does:

1. It starts with two pointers, `i` and `j`. Pointer `i` is positioned at the start of the array (index 0), and `j` is positioned at the end of the array (index `len(s) - 1`).
2. While `i` is less than `j`, the while loop continues to run.
3. Inside the loop, the characters at index `i` and index `j` are swapped. This can be thought of as two people in the line stepping out, passing by each other, and then stepping back in at each other's original positions.
4. After the swap, the `i` index is incremented, and the `j` index is decremented. This moves the pointers closer to the center of the array.
5. The loop stops once `i` is no longer less than `j`, which means the pointers have met in the middle or passed each other, indicating that the entire array has been reversed.

By only using the existing array to store the characters, the solution uses `O(1)` extra space, since the memory used does not depend on the size of the input array but only on a fixed number of index variables.

Solution Approach

The implementation of the provided solution employs a two-pointer technique as the core algorithm. This approach doesn't need any additional data structures, hence it sticks to the `O(1)` extra memory constraint. Here's the step-by-step walk-through:

1. **Initialization of Pointers**: Two pointers are initialized: `i` starts at index `0` (the beginning of the `s` array), and `j` starts at `len(s) - 1` (the end of the `s` array).
2. **Loop Until Pointers Meet**: We enter a loop that keeps running while `i < j`. The `<` condition ensures that we don't swap elements that have already been swapped or try to swap the middle element with itself in an odd-length array.
3. **Swapping Elements**: Inside the loop, we perform the swap. `s[i], s[j] = s[j], s[i]` is a Pythonic way to swap the values at two indices in a list.
4. **Moving Pointers**: After the swap, we need to move the pointers. The `i` pointer is moved one step forward (toward the middle) by doing `i += 1`, and the `j` pointer is moved one step back (away from the end) by doing `j -= 1`.
5. **Termination of Loop**: The loop finishes when `i >= j`, meaning the pointers have met or crossed, and the entire array `s` has been reversed.

In terms of data structures, the input and output are the same array `s`. We're not using any auxiliary data structures. This is purely an in-place transformation, an efficient pattern for situations where minimizing memory usage is crucial.

The pattern is simple and elegant, often used for reversing arrays or strings, and is a staple in a programmer's toolbox. It is also a good example of how to manipulate elements within a single data structure without additional memory overhead.

Example Walkthrough

To illustrate the solution approach, consider a simple array of characters `s = ['h', 'e', 'l', 'l', 'o']`. The goal is to reverse this array in place, resulting in `s = ['o', 'l', 'l', 'e', 'h']`.

Here's how the algorithm works on this small example:

1. **Initialization of Pointers**: Start with two indices, `i = 0` and `j = len(s) - 1` which is `j = 4`. So `i` points to `'h'` and `j` points to `'o'`.
2. **Loop Until Pointers Meet**: The condition `i < j` is `True` because `0 < 4`. Therefore, we go into the loop.
3. **Swapping Elements**: We perform the swap: `s[i], s[j]` becomes `s[j], s[i]`. After swapping, the array looks like this: `s = ['o', 'e', 'l', 'l', 'h']`.
4. **Moving Pointers**: We increment `i` by 1 and decrement `j` by 1. Now `i = 1` and `j = 3`. The pointer `i` now points to `'e'` and `j` points to the second `'l'`.
5. **Are Pointers Meeting?**: Yes, `i < j` still holds true (since `1 < 3`), so we repeat the swap.
6. **Second Swapping and Pointer Update**: Swap the characters at index 1 and 3. The array now looks like this: `s = ['o', 'l', 'l', 'e', 'h']`. Then, `i` is incremented to `2` and `j` is decremented to `2`.
7. **Termination of Loop**: Now `i` is equal to `j` (`2 = 2`), so the loop condition `i < j` is no longer satisfied. The loop stops.

At this point, we have completed the reversal of the array. The array began with the order `['h', 'e', 'l', 'l', 'o']` and after the process, it has been reversed in place to `['o', 'l', 'l', 'e', 'h']`. Using this two-pointer approach, the reversal was done in-place without requiring extra space.

Python Solution

```python
from typing import List

class Solution:
    def reverseString(self, string_list: List[str]) -> None:
        """
        Reverses the order of characters in the input string list in-place.

        :param string_list: List of characters representing the string to be reversed.
        :return: None, modifies the list in-place.
        """

        # Initialize two pointers.
        left_pointer = 0
        right_pointer = len(string_list) - 1

        # Loop until the two pointers meet in the middle.
        while left_pointer < right_pointer:
            # Swap the characters at the left and right pointers.
            string_list[left_pointer], string_list[right_pointer] = string_list[right_pointer], string_list[left_pointer]

            # Move the pointers closer to the center.
            left_pointer += 1
            right_pointer -= 1

```

Java Solution

```java
class Solution {
    // Method to reverse a string represented as a character array.
    public void reverseString(char[] str) {
        // Initialize two pointers. One starting from the beginning (left) and
        // the other from the end of the array (right).
        int left = 0, right = str.length - 1;

        // Loop until the two pointers meet in the middle.
        while (left < right) {
            // Swap the characters at the left and right pointers.
            char temp = str[left];
            str[left] = str[right];
            str[right] = temp;

            // Move the pointers towards each other.
            left++;
            right--;
        }
        // After the loop, the string is fully reversed in place.
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the header needed for std::vector
#include <algorithm> // Include the header for the std::swap function

// Solution class containing the method to reverse a string
class Solution {
public:
    // Function to reverse the characters in the vector 'str'
    void reverseString(vector<char>& str) {
        int leftIndex = 0; // Start index for the left side
        int rightIndex = str.size() - 1; // Start index for the right side

        // Continue swapping elements until the middle of the string is reached
        while (leftIndex < rightIndex) {
            // Swap elements at leftIndex and rightIndex
            std::swap(str[leftIndex], str[rightIndex]);

            // Move indices towards the middle
            ++leftIndex;
            --rightIndex;
        }
    }
};

```

Typescript Solution

```typescript
/**
 * Reverses an array of strings in place.
 * @param {string[]} strArray - The array of strings to be reversed.
 */
function reverseString(strArray: string[]): void {
    // Initialize two pointers, one at the start and one at the end of the array.
    let startIdx = 0;
    let endIdx = strArray.length - 1;

    // Continue swapping elements until the start index is greater or equal to the end index.
    while (startIdx < endIdx) {
        // Destructuring assignment to swap elements at startIdx and endIdx.
        [strArray[startIdx], strArray[endIdx]] = [strArray[endIdx], strArray[startIdx]];

        // Move the start index forward and the end index backward.
        startIdx++;
        endIdx--;
    }
    // The strArray is modified in place, so no return statement is necessary.
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the string `s`. This is because the code uses a while loop that iterates over each element of the string once to swap the characters.

The space complexity of the code is `O(1)` as the reversal is done in place, and no additional space is used that grows with the input. Only two pointers `i` and `j` are utilized to make the swaps.