Reverse Character Arrays LeetCode

The problem is to reverse the order of characters in a given vector of characters.

Example: Input: **{'h', 'e', 'l', 'l', 'o'}** Output: **{'o', 'l', 'l', 'e', 'h'}**

Explanation: The given vector **{'h', 'e', 'l', 'l', 'o'}** is reversed to **{'o', 'l', 'l', 'e', 'h'}**. The first character 'h' swaps with the last character 'o', the second character 'e' swaps with the second-to-last character 'l', and so on. The result is a reversed vector.

Approach 1: Function to reverse the vector iteratively

The **reverseIteratively** function uses a two-pointer approach to reverse the vector iteratively. It starts with two pointers, **start** pointing to the first element and **end** pointing to the last element. It swaps the characters at the **start** and **end** positions and then increments **start** and decrements **end**. This process continues until **start** becomes greater than or equal to **end**, which means the entire vector has been reversed.

Time Complexity: The time complexity of the iterative reversal approach is O(N), where N is the number of elements in the vector. The function iterates through half of the vector length, swapping characters at each iteration.

Space Complexity: The iterative reversal approach has a space complexity of O(1) since it performs in-place reversal, modifying the given vector directly without using any extra space.

Approach 2: Function to reverse the vector recursively

The **reverseRecursively** function uses a recursive approach to reverse the vector. It takes the start and end indices as parameters. In each recursive call, it swaps the characters at the **start** and **end** indices and then recursively calls itself with updated indices (**start + 1** and **end - 1**). The recursion stops when **start** becomes greater than or equal to **end**.

Time Complexity: The time complexity of the recursive reversal approach is also O(N), where N is the number of elements in the vector. The function makes N/2 recursive calls, each performing a constant amount of work.

Space Complexity: The recursive reversal approach has a space complexity of O(N) due to the recursive function calls. Each recursive call adds a new frame to the call stack, and in the worst case scenario, the maximum depth of the call stack is N/2, resulting in O(N) space usage.

Which Approach to Use: Both approaches, iterative and recursive, provide the same result. The choice of approach depends on personal preference or specific requirements of the problem. The iterative approach may be slightly more efficient in terms of space complexity as it uses O(1) space, whereas the recursive approach has O(N) space complexity due to the call stack usage. However, the performance difference is likely to be negligible for small input sizes.