**Solution - Reverse String**

In this article, we present a sample solution for the problem of [Reverse String](https://leetcode.com/explore/learn/card/recursion-i/250/principle-of-recursion/1440/).

The problem is not difficult, yet the trick part is that we have an additional **constraint** for the problem, *i.e.* one must modify the string with {\mathcal{O}(1)}O(1) extra space.

Let's define the problem as the function reverseString(str[0...n-1]), where str[0...n-1] is a list of characters with the first character denoted as str[0].

Below, we will discuss how we can solve this problem with recursion.

*First Attempt*

If we follow the idea of the problem of printing a string in reversed order, as we presented in [the first article](https://leetcode.com/explore/learn/card/recursion-i/250/principle-of-recursion/1439/) of this card, we might come up with the following algorithm:

1. take the leading character str[0] from the input string.
2. call the function itself on the remaining substring, *i.e.* reverseString(str[1...n-1]).
3. then append the leading character to the result returned in the step (2).

The above algorithm could work, except that it does not meet the constraint imposed on the problem. This is because one would need to keep the intermediate result in step **(2)** which is proportional to the input string (*i.e.* with at least {\mathcal{O}(N)}O(*N*) space complexity), which in no case could satisfy the constraint (use {\mathcal{O}(1)}O(1) space to modify the string).

*Another Divide-and-Conquer Solution*

Looking closer at the constraint imposed by the problem, if we put it into the context of recursion, we could interpret it as not having additional space consumption between two consecutive recursive calls, *i.e.* we should divide the problem into independent subproblems.

So one of the ideas about how to divide the problem would be reducing the input string at each step into two components: *1).* the leading and trailing characters. *2).* the remaining substring without the leading and trailing characters. We then can solve the two components independently from each other.

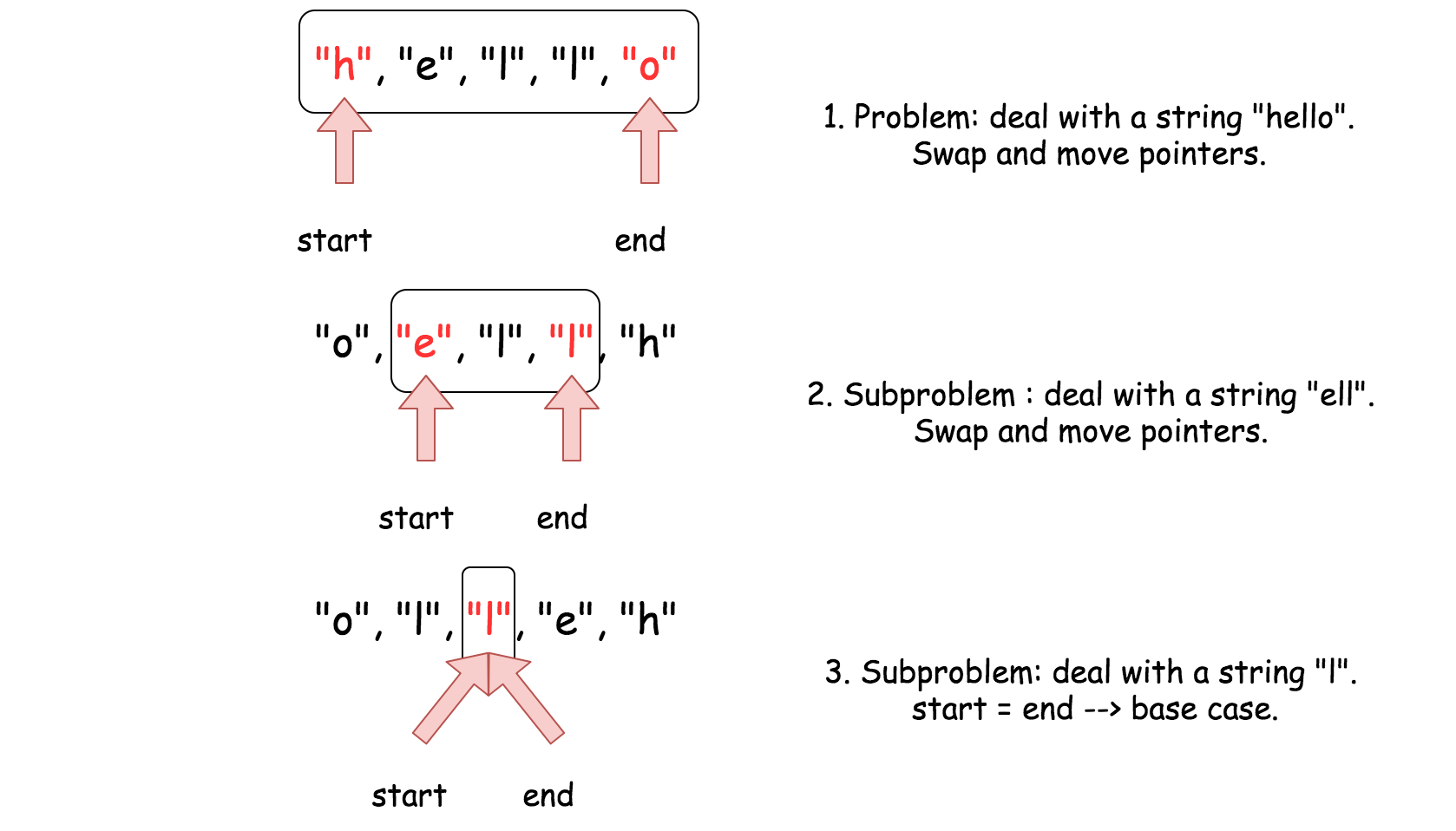
Following the above idea, we could come up the algorithm as follows:

1. Take the leading and trailing characters from the input string, *i.e.* str[0] and str[n-1].
2. Swap the leading and trailing characters in place.
3. Call the function recursively to reverse the remaining substring, *i.e.* reverseString(str[1...n-2]).

Note that you can actually swap the order of steps *(2)* and *(3)*, since they are independent tasks. Yet, it is better to keep them in this order, since this way we can use the optimization called [tail recursion](https://en.wikipedia.org/wiki/Tail_call). We'll shed more light on tail recursion in later chapters.

Here is an implementation of the above algorithm.

Given the input string ["h", "e", "l", "l", "o"], we illustrate how it can be divided and solved:



As one can see, we only need a constant memory in each recursive call in order to swap the leading and trailing characters. As a result, it meets the constraint of the problem.