2000. Reverse Prefix of Word

Easy · Two Pointers · String

Leetcode Link

Problem Description

The task here involves manipulating a given string `word` based on the presence of a specified character `ch`. We are required to find the first occurrence of the character `ch` within `word`. Once found, we reverse the substring of `word` that starts from the beginning (index `0`) and ends at the position where `ch` is found (this position is included in the segment to be reversed). In case the character `ch` does not exist in `word`, no changes should be made to the original string. The main objective is to return the modified string as a result. The problem stresses that the input string `word` is 0-indexed, which means the first character is at position 0.

For example, consider `word = "abcdefd"` and `ch = "d"`. Since the first occurrence of "d" is at index 3, we reverse the substring from index 0 to 3, resulting in `"dcbaefd"` which is the output.

Intuition

The intuition behind the solution is to first locate the index of the character `ch` within the string `word` using a string search function. In Python, this is typically done using the `find()` method, which returns the lowest index of the substring if it is found, or `-1` if it is not found. If the character `ch` isn't found, `i` will be `-1` and the solution will simply return the original string as no action is needed.

If the character is found, we need to reverse the segment of the string up to and including that character. In Python, this can be efficiently accomplished using slicing. The slice `word[i::-1]` produces a new string that consists of the characters from the start of `word` up to `i`, reversed. After reversing the desired segment, we concatenate it with the remaining part of `word` that comes after the character `ch` (`word[i + 1 :]`) to form the final string.

The approach is direct and utilizes Python's powerful slicing capabilities, enabling the operation to be performed in a single line of code and with clear readability.

Solution Approach

The solution approach for this problem leverages a straightforward algorithm, which goes as follows:

1. **Find the Character 'ch':** Firstly, we use the `find()` method available in Python to search for the first occurrence of the character `ch` in the string `word`. The `find()` method returns the index of 'ch' if it is found, and `-1`. The syntax `word.find(ch)` executes this search.

2. **Check Character Presence:** We then check if `ch` is present in `word` by examining if the result from `find()` is not `-1`. If `ch` isn't present (i.e., if `find()` returns `-1`), we do nothing and return the original `word` unmodified.

3. **Reverse String Segment:** If `ch` is found, we proceed to reverse the segment of `word` from the start to the index where `ch` is found. Python strings support slicing, which we use to reverse a substring. The slice operation `word[i::-1]` is used, where `i` is the index returned by the `find()` operation. This operation reverses the string from the start up to index `i`.

4. **Concatenate the Segments:** The last step is to concatenate the reversed segment with the rest of the string that follows after 'ch'. This is done using `word[i + 1 :]`, which gives us the substring from the character immediately after 'ch' to the end of `word`.

5. **Return the Result:** The final step is to combine the two substrings— the reversed segment and the remaining part of the original string— to form the modified string and return it as the result.

The overall solution is efficient, requiring only a single pass to find the character and another pass to create the reversed segment. It does not use any additional data structures, and the operations used (string search, slicing, and concatenation) are all native to Python and designed for performance. The provided code encapsulates this logic concisely:

```python
class Solution:
    def reversePrefix(self, word: str, ch: str) -> str:
        i = word.find(ch)
        return word if i == -1 else word[i::-1] + word[i + 1 :]
```

This implementation highlights the effectiveness of Python's string manipulation features, allowing for an elegant expression of the solution.

Example Walkthrough

Let's go through a small example to illustrate the solution approach:

Consider the `word` to be "example" and the `ch` to be "p". The task is to find the first occurrence of the character "p" in the word "example" and reverse the string from the start up to and including the character "p".

1. **Find the Character 'ch':** We start by finding the index of "p" using `word.find(ch)`. In our example, `word = "example"` and `ch = "p"`. The character "p" is at index 4 in "example".

2. **Check Character Presence:** We check if "p" is found in "example". Since the `find()` method returned 4, we know "p" is present (it didn't return `-1`).

3. **Reverse String Segment:** We then reverse the string segment from the start to index 4, which includes "p". Using Python slicing, we write this operation as `word[4::-1]`, which gives us "elpmaxe".

4. **Concatenate the Segments:** Now we concatenate the reversed segment with the rest of the string after "p". The remaining part of the string is obtained with `word[5:]`, which is "le". So, appending it to the reversed segment, we get "elpmaxe" + "le" = "elpmaxele".

5. **Return the Result:** Finally, the modified string "elpmaxele" is returned, representing the string "example" with the segment up to and including the first occurrence of "p" reversed.

By following these steps, we used the given solution approach on the word "example" with the target character "p", resulting in "elpmaxele".

Python Solution

```python
class Solution:
    # Function to reverse the prefix of a word up to a certain character
    def reversePrefix(self, word: str, ch: str) -> str:
        # Find the index of the character ch in the word
        index_of_char = word.find(ch)

        # If ch is not found, the index will be -1 and the original word is returned
        if index_of_char == -1:
            return word

        # If ch is found, reverse the substring from start to the index of ch (inclusive)
        # then concatenate with the remaining substring starting from the element right after ch's index
        # The [::-1] slice reverses the substring
        reversed_prefix = word[:index_of_char+1][::-1]
        remaining_word = word[index_of_char+1:]

        return reversed_prefix + remaining_word

```

Java Solution

```java
class Solution {
    public String reversePrefix(String word, char ch) {
        // Find the index of the first occurrence of the character 'ch' in the 'word'
        int index = word.indexOf(ch);

        // If 'ch' is not found, return the original 'word'
        if (index == -1) {
            return word;
        }

        // Convert the string into a char array for in-place reversal
        char[] charArray = word.toCharArray();

        // Reverse the prefix of the word up to the index of 'ch'
        for (int i = 0; i < index; ++i, --index) {
            // Swap characters at position i and index
            char temp = charArray[i];
            charArray[i] = charArray[index];
            charArray[index] = temp;
        }

        // Convert the char array back to a string and return
        return String.valueOf(charArray);
    }
}

```

C++ Solution

```cpp
#include <algorithm> // Required for std::reverse
#include <string>    // Required for std::string

class Solution {
public:
    // This function takes a string and a character.
    // It reverses the order of characters in the string up to (and including) the first occurrence of the char.
    // Input: a string 'word' and a character 'ch'
    // Output: the modified string with the prefix reversed, if the character is found.
    string reversePrefix(string word, char ch) {
        // Find the first occurrence of the character 'ch' in 'word'.
        int index = word.find(ch);

        // If the character is found (i.e., find() doesn't return string::npos), reverse the substring.
        if (index != string::npos) {
            // Reverse from the beginning of 'word' to the character 'ch' (inclusive).
            reverse(word.begin(), word.begin() + index + 1);
        }

        // Return the modified string.
        return word;
    }
};

```

Typescript Solution

```typescript
// Reverses the prefix of a given word up to and including the first occurrence of a specified character.
// @param {string} word - The original string to be processed.
// @param {string} ch - The character to search for in the string.
// @return {string} - The string with the prefix reversed up to and including the first occurrence of the character.
function reversePrefix(word: string, ch: string): string {
    // Find the index of the character in the string.
    const index = word.indexOf(ch) + 1;

    // If the character is not found, return the original word.
    if (index === 0) {
        return word;
    }

    // Slice the word up to and including the found character, reverse this part,
    // and then join it back with the rest of the original word.
    return [...word.slice(0, index)].reverse().join('') + word.slice(index);
}

```

Time and Space Complexity

Time Complexity

The main operation in the given piece of code is finding the first occurrence of the character `ch` within the string `word` and then reversing the prefix up to that character.

1. The time complexity for the string `find` operation is `O(n)` where `n` is the length of the string, since in the worst case, it needs to scan the entire string.

2. Slicing the string to reverse the prefix takes `O(k)` where `k` is the index of the found character (or `n` in the worst case if the character is at the end of the string). While string slicing is typically `O(n)` in Python, in this case we are considering the slice operation up to the first occurrence of `ch`. Therefore, the cost is relative to the position of `ch`.

3. Concatenating the reversed prefix with the rest of the string is also an `O(n)` operation because it creates a new string from the two substrings.

Since these operations are done sequentially and we're interested in the worst-case scenario, the overall time complexity is dominated by the O(n) operations, making the total time complexity of the solution `O(n)`.

Space Complexity

The space complexity of the code is primarily dependent on the storage needed for the output, which is the reversed prefix concatenated with the rest of the string.

1. Since the reversed prefix is derived from slicing the existing string, it creates a new string object with the maximum length of `n`.

2. Concatenating the reversed prefix with the rest of the string generates another new string, but since strings are immutable in Python, this is 'destructive' concatenation and also takes up to `n` space.

However, it is important to note that the space used for the output does not count towards the extra space usage, as it is considered the space required to hold the input and output of the function. No additional data structures are used that scale with the size of the input, therefore, the extra space complexity is `O(1)` (constant space for the index `i` and temporary substrings during the execution).

In summary, the space complexity for additional space required is `O(1)`, while the overall space complexity (including space for inputs and outputs) is `O(n)`.