

# 1910. Remove All Occurrences of a Substring

MediumString

## Problem Description

The task described in the LeetCode problem is about string manipulation. You are given two strings `s` and `part`. Your goal is to repeatedly find the leftmost occurrence of the string `part` in `s` and remove it. You should keep doing this operation until `part` can no longer be found within `s`. To clarify, a substring is a sequence of characters that appear in consecutive order within another string. The operation is only complete when there are no more sequences of characters in `s` that match `part`. The output should be the resulting string `s` after all possible removals of the substring `part` have been conducted.

## Intuition

To solve the problem, one can use the built-in string methods available in python. The intuition lies in searching for the substring `part` within the string `s` and removing the leftmost occurrence of it. This can be done iteratively using a `while` loop, which continues as long as `part` is found in `s`.

To implement this:

1. Check if `part` is a substring of `s` using the `in` keyword.
2. If it is, use the `replace` method of the string object, which replaces the first occurrence of `part` in `s` with an empty string (effectively removing it), but make sure to limit the replacement to just one occurrence by passing `1` as the second argument to `replace`.
3. Return the modified string `s` once there are no more occurrences of `part` in it.

## Solution Approach

The implementation of the solution is straightforward and relies primarily on Python's string processing capabilities. Here's a step-by-step explanation of the approach using algorithms and data structures:

- **Algorithm: Iterative Removal**
  - The algorithm uses a loop to repeatedly search and remove the substring `part` from `s`. It continues to do so until the `part` can no longer be found within `s`.
- **Data Structure: String**
  - Strings are the primary data structure used in this problem. In Python, strings are immutable, meaning a new string is created each time you modify it.

Given the Python code snippet:

```
class Solution:
    def removeOccurrences(self, s: str, part: str) -> str:
        while part in s:
            s = s.replace(part, '', 1)
        return s
```

Here's an explanation of the code:

- **While Loop:**
  - The `while` loop checks whether `part` is still a substring of `s`. The condition `part in s` returns a boolean value - `True` if `part` is found in `s` and `False` otherwise.
- **String Replacement:**
  - Inside the loop, the `replace` method is called on `s`. The first argument is the substring `part` we are looking for, the second argument is an empty string to which the found `part` will be replaced, signifying its removal. The third argument `1` ensures that only the first instance of `part` is replaced, which corresponds to the "leftmost occurrence".
- **Return value:**
  - Once the loop terminates (when `part` is not found in `s`), the final version of `s` is returned, which no longer contains any occurrences of `part`.

No complex patterns or sophisticated algorithms are needed beyond the basic iterative approach. The solution leverages Python's built-in string methods to achieve the result with a clean and easy-to-understand implementation.

## Example Walkthrough

To illustrate the solution approach, let's go through a small example. Imagine we have the string `s = "axbxcx"` and the substring `part = "x"`. Our goal is to remove the leftmost occurrence of `x` from `s` until `x` can no longer be found in `s`.

Here's how the implementation works step by step:

- **Initial:** `s = "axbxcx"`, `part = "x"`
- **First Iteration:**
  1. Check if `part` is in `s`: `x` is in `"axbxcx"`
  2. Replace the first occurrence of `part` with an empty string:
    - Before: `s = "axbxcx"`
    - After: `s = "abxcx"` (we removed the first `x`)
- **Second Iteration:**
  1. Check if `part` is in `s` again: `x` is in `"abxcx"`
  2. Replace the first occurrence of `part` with an empty string:
    - Before: `s = "abxcx"`
    - After: `s = "abcx"` (we removed the second `x`)
- **Third Iteration:**
  1. Check if `part` is in `s` again: `x` is in `"abcx"`
  2. Replace the first occurrence of `part` with an empty string:
    - Before: `s = "abcx"`
    - After: `s = "abc"` (we removed the last `x`)
- **Final Check:**
  - `part` is no longer in `s`: `x` is not in `"abc"`

The resulting string `s` is `"abc"` because all instances of `part` have been removed. This gives us the output of the function. Using the provided code snippet, the final return statement `return s` would give us `"abc"` as the solution to our input example.

## Solution Implementation

### Python

```
class Solution:
    def removeOccurrences(self, string: str, part: str) -> str:
        # Repeatedly search for the 'part' in 'string' and remove its first occurrence
        while part in string:
            # Find the index of the first occurrence of 'part'
            index = string.find(part)
            # Remove 'part' by slicing the string before and after 'part'
            string = string[:index] + string[index+len(part):]

        # Return the modified string after removing all occurrences of 'part'
        return string

# Example usage:
# sol = Solution()
# new_string = sol.removeOccurrences("daabcbaabcbc", "abc")
# print(new_string) # Output would be "dab"
```

### Java

```
class Solution {

    /**
     * Removes all occurrences of the substring 'part' from the string 's'.
     *
     * @param s The original string from which occurrences of 'part' will be removed.
     * @param part The substring to be removed from 's'.
     * @return The modified string with all occurrences of 'part' removed.
     */
    public String removeOccurrences(String s, String part) {
        // Keep removing 'part' from 's' while 's' contains 'part'
        while (s.contains(part)) {
            // Replace the first occurrence of 'part' in 's' with an empty string
            s = s.replaceFirst(part, "");
        }
        return s;
    }
}
```

### C++

```
class Solution {
public:
    // Function to remove all occurrences of a substring 'part' from the string 's'
    string removeOccurrences(string s, string part) {
        // Get the size of the substring 'part'
        int partSize = part.size();

        // Find the first occurrence of 'part' in 's'
        size_t position = s.find(part);

        // Continue looping as long as 'part' is found in 's'
        while (position != string::npos) {
            // Erase 'part' from 's'
            s.erase(position, partSize);
            // Find the next occurrence of 'part' in 's'
            position = s.find(part);
        }

        // Return the modified string with all 'part' occurrences removed
        return s;
    }
};
```

### TypeScript

```
/**
 * Removes all occurrences of a specified substring from the given string.
 * @param {string} str - The original string from which to remove occurrences.
 * @param {string} part - The substring to remove from the original string.
 * @returns {string} The modified string with all occurrences of the substring removed.
 */
function removeOccurrences(str: string, part: string): string {
    // Continue to look for the substring `part` in `str` until it cannot be found
    while (str.includes(part)) {
        // Replace the first occurrence of `part` in `str` with an empty string
        str = str.replace(part, '');
    }
    // Return the modified string with all occurrences of `part` removed
    return str;
}

class Solution:
    def removeOccurrences(self, string: str, part: str) -> str:
        # Repeatedly search for the 'part' in 'string' and remove its first occurrence
        while part in string:
            # Find the index of the first occurrence of 'part'
            index = string.find(part)
            # Remove 'part' by slicing the string before and after 'part'
            string = string[:index] + string[index+len(part):]

        # Return the modified string after removing all occurrences of 'part'
        return string

# Example usage:
# sol = Solution()
# new_string = sol.removeOccurrences("daabcbaabcbc", "abc")
# print(new_string) # Output would be "dab"
```

## Time and Space Complexity

### Time Complexity

The provided function's time complexity can be analyzed based on the `while` loop and the `str.replace()` method used within it.

- The `while` loop runs as long as the substring `part` is found within the string `s`.
- `str.replace()` is called each time the `part` is found, and it has a complexity of  $O(n)$  in the worst case, where `n` is the length of string `s`, since in the worst case it has to scan the entire string to replace the occurrences.

If `m` is the length of the substring `part`, the worst-case scenario occurs when `part` is found in `s` multiple times and the positions of `part` in `s` are distributed such that most of the string has to be scanned for each replacement. Therefore, the complexity can approximately be  $O((n - m + 1) * n)$  because it could take up to  $(n - m + 1)$  searches through the string `s`.

Hence, the worst-case time complexity is  $O((n - m + 1) * n)$ .

### Space Complexity

The space complexity of the function arises from the storage required for the input string `s` and the additional strings created during the replacements.

- The input string `s` has a space complexity of  $O(n)$ , where `n` is its length.
- Each time a replacement is performed, a new string is created, and this new string could have a length up to  $n - m$ .

Given that only one replacement string exists at a time (the previous is discarded when the new one is created), the additional space required for the algorithm as a result of replacements is also  $O(n)$ .

Therefore, the overall space complexity of the algorithm is  $O(n)$ .