833. Find And Replace in String

Medium Array String Sorting Leetcode Link

Problem Description

In this problem, you are given a string `s` and tasked with performing a series of replacement operations on it. These operations are specified by three parallel arrays `indices`, `sources`, and `targets`, each of length `k`, representing the `k` operations.

For each operation `i`:

1. You have to check if the substring `sources[i]` is found in the string `s` exactly at the position `indices[i]`.
2. If the substring `sources[i]` does not exist at the specified index, you do nothing for that particular operation.
3. If the substring is found, you replace it with the string `targets[i]`.

All the replacements are to be done simultaneously, which means:

- They don't affect each other's indexing (you should consider the original indices while replacing).
- There will be no overlap among the replacements (no two substrings `sources[i]` and `sources[j]` will replace parts of `s` that overlap).

A *substring* is defined as a sequence of characters that are contiguous in a string.

Intuition

The key insight for solving this problem is understanding that all replacements happen independently and simultaneously, based on the original string `s`. This calls for a mapping from each `indices[i]` to its corresponding replacement (if valid), without disturbing the original indexing as subsequent replacements are planned according to the original positions.

We approach the solution by creating a mapping, in this case using an array `d` with the same length as the string `s`, and initialize it with a default value indicating an index with no operation. This array `d` is filled with the operation index `k` at position `indices[i]` only if the substring `sources[i]` is verified to be at the original position `indices[i]` in `s`.

While constructing the result:

1. We iterate through the original string `s`.
2. At each index `i`, we check the mapping array `d`.
3. If there is no replacement to be done (indicated by a default value), we simply add the current character `s[i]` to the result.
4. If a replacement is needed, we append the corresponding target string `targets[d[i]]` instead and increment `i` by the length of the source substring, effectively skipping over the entire substring that has been replaced.
5. We do this until we have processed the entire string.

By following this strategy, we can ensure that all replacements are based on the original indices, thereby meeting the constraint that the replacements do not alter each other's indexing, finally returning the correct modified string.

Solution Approach

The implementation of the solution follows a fairly straightforward process that can be broken down into the following steps:

1. Initialize an array `d` with the same length as the string `s` which contains default values (`-1` in this case). This array serves as a direct mapping to check if there's a valid replacement operation for each index in the string `s`.

2. Iterate over pairs of `indices` and `sources` using the `enumerate` function to keep track of the operation index `k`.

3. For each pair `(i, src)`, use `s.startswith(src, i)` to check if the substring `src` exists starting exactly at index `i` in the string `s`. If it does exist, set `d[i]` to `k` indicating that a replacement operation is mapped to this index.

4. Create an empty list `ans` to store the characters (and substrings) that will make up the resulting string after all operations have been performed.

5. Iterate over the string `s` with index `i`. If `i` is marked in `d` as a valid replacement index (`d[i] != -1`), append the corresponding `targets[d[i]]` to `ans`. Then increment `i` by the length of `sources[d[i]]` since you've just processed the entire substring that's been replaced.

6. If `i` is marked as having no operation (`d[i] == -1`), simply add `s[i]` to `ans` and increment `i` by 1 to continue processing the string.

7. Once the iteration is complete, combine the contents of `ans` using `"".join(ans)` to get the final string which reflects the result after applying all the replacement operations.

The data structure used in this approach is primarily an array (`d`) for mapping operations. The algorithm leverages the fact that operations are independent and simultaneous, which simplifies to updating indices directly without considering the impact of previous replacements—a classic case of 'direct addressing' where each index corresponds to a particular bit of information (here, the presence and index of a replacement operation).

The Python `startswith` method is used for string comparison to verify occurrences of substrings which is crucial for determining valid operations. The use of this method avoids writing additional code to manually compare substring characters.

The iteration over the string is done in a single pass, and the list `ans` builds up the result in a dynamic fashion, making the solution efficient in terms of both time and space complexity.

Overall, this approach capitalizes on the simultaneous nature of the operations and utilizes efficient string methods and direct mapping techniques provided by the language to achieve the desired outcome.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Suppose we have the string `s` which is `"abcd"` and we want to perform the following operations:

- `indices`: [0, 2]
- `sources`: ["a", "cd"]
- `targets`: ["z", "x"]

This corresponds to two operations:

1. Replace the substring "a" at index 0 with "z".
2. Replace the substring "cd" at index 2 with "x".

Step 1: Initialize the array `d` of the same length as `s` (`4` in this example) with default values `-1`.

`d`: [-1, -1, -1, -1]

Step 2: Iterate over pairs of `indices` and `sources` (`enumerate` is not needed here since we're only doing a 1-to-1 mapping).

Checking Operation 1:

- We check if `s.startswith("a", 0)`, which is `True`.
- Thus, we set `d[0]` to the operation index `0` (since 0 is the first index in `indices`).

`d`: [0, -1, -1, -1]

Checking Operation 2:

- We check if `s.startswith("cd", 2)`, which is `True`.
- Thus, we set `d[2]` to the operation index `1` (since 1 is the second index in `indices`).

`d`: [0, -1, 1, -1]

Step 3: Create an empty list `ans`.

`ans`: []

Step 4: Iterate over the string `s` with index `i`.

- `i = 0`: Since `d[0]` is `0`, we append `targets[0]` ("z") to `ans` and skip to the index after the replaced substring (since "a" has length `1`, we increment `i` by `1`).

`ans`: ["z"]

- `i = 1`: Since `d[1]` is `-1`, there's no operation. We add `s[1]` ("b") to `ans` and increment `i` by `1`.

`ans`: ["z", "b"]

- `i = 2`: Since `d[2]` is `1`, we append `targets[1]` ("x") to `ans` and skip to the index after the replaced substring (since "cd" has length `2`, we increment `i` by `2`.

`ans`: ["z", "b", "x"]

- Since we've reached the end of string `s`, the iteration stops.

Step 5: Combine the list `ans` into the final string.

Result: `"zbx"`

And that's the final string after performing all the replacement operations. The algorithm efficiently applies replacements based on the original string, resulting in the desired output.

Python Solution

```python
class Solution:
    def findReplaceString(self, s: str, indices: list[int], sources: list[str], targets: list[str]) -> str:
        # Initialize the length of the original string.
        length_of_string = len(s)

        # Create a list to keep track of valid source index matches (-1 means no match).
        match_tracker = [-1] * length_of_string

        # Loop through indices and sources to fill the match_tracker with correct target indices.
        for idx, (index, source) in enumerate(zip(indices, sources)):
            # Check if the source matches the substring in s starting at index.
            if s.startswith(source, index):
                match_tracker[index] = idx

        # Create a list to construct the answer.
        answer_components = []

        # Initialize the index for traversing the string.
        i = 0
        while i < length_of_string:
            # If there's a valid source match at current index, replace with target string.
            if match_tracker[i] != -1:
                answer_components.append(targets[match_tracker[i]])
                # Skip the length of the source that was replaced.
                i += len(sources[match_tracker[i]])
            else:
                # If no valid source match, keep the original character.
                answer_components.append(s[i])
                # Move to the next character.
                i += 1

        # Join all components to form the final string and return it.
        return "".join(answer_components)

```

Java Solution

```java
class Solution {

    // Method to replace substrings in 's' according to indices 'indices', with replacements from 'targets'
    public String findReplaceString(String s, int[] indices, String[] sources, String[] targets) {
        // Find the length of the string 's'
        int lengthOfString = s.length();
        // Array to keep track of the valid replacement indices
        int[] replacementIndices = new int[lengthOfString];
        // Initialize the replacementIndices array with -1 indicating no replacement initially
        Arrays.fill(replacementIndices, -1);

        // Loop through indices to find valid replacements
        for (int index = 0; index < indices.length; ++index) {
            int replaceAt = indices[index];
            // Check if the current source string is present in 's' starting at the index 'replaceAt'
            if (s.startsWith(sources[index], replaceAt)) {
                // Mark the valid replacement index
                replacementIndices[replaceAt] = index;
            }
        }

        // Using StringBuilder for efficient string manipulation
        StringBuilder resultBuilder = new StringBuilder();

        // Iterate through the original string 's'
        for (int i = 0; i < lengthOfString;) {
            // If there is a valid replacement at the current index 'i'
            if (replacementIndices[i] >= 0) {
                // Append the target replacement string to resultBuilder
                resultBuilder.append(targets[replacementIndices[i]]);
                // Increment 'i' by the length of the source at this valid index to skip replaced part
                i += sources[replacementIndices[i]].length();
            } else {
                // No valid replacement, append the current character and move to the next
                resultBuilder.append(s.charAt(i++));
            }
        }
        // Convert the StringBuilder object to a String before returning
        return resultBuilder.toString();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Method to perform find and replace in a string given specific indices, sources, and targets.
    string findReplaceString(string str, vector<int>& indices, vector<string>& sources, vector<string>& targets) {
        int strSize = str.size();
        // Create a lookup array to associate indices in the main string with their replacement index, initialized as -1.
        vector<int> replacementIndex(strSize, -1);

        // Loop through each index provided to calculate the replacement index.
        for (int i = 0; i < indices.size(); ++i) {
            int index = indices[i];
            // Only set the replacement index if the source matches the substring starting at index.
            if (str.compare(index, sources[i].size(), sources[i]) == 0) {
                replacementIndex[index] = i;
            }
        }

        string result; // Initialize the result string which will accumulate the final output.

        // Iterate through the original string by character.
        for (int i = 0; i < strSize;) {
            // If the current index has a valid replacement index, concatenate the target string.
            if (replacementIndex[i] != -1) {
                result += targets[replacementIndex[i]];
                // Move the index forward by the length of the source that was replaced.
                i += sources[replacementIndex[i]].size();
            } else {
                // If there's no replacement, just append the current character to the result.
                result += str[i++];
            }
        }

        // Return the modified string after all replacements are done.
        return result;
    }
};

```

Typescript Solution

```typescript
function findReplaceString(
    originalString: string,
    indexArray: number[],
    sourceArray: string[],
    targetArray: string[],
): string {
    // The length of the original string.
    const stringLength: number = originalString.length;

    // An array to keep track of which indices in the original string have valid replacements.
    const replacementIndexArray: number[] = Array(stringLength).fill(-1);

    // Iterate through the array of indices to find valid replacements.
    for (let i = 0; i < indexArray.length; ++i) {
        const index = indexArray[i];
        const source = sourceArray[i];

        // If the source string is found at the specified index, update the replacement index array.
        if (originalString.startsWith(source, index)) {
            replacementIndexArray[index] = i;
        }
    }

    // An array to build the new string with replacements.
    const resultArray: string[] = [];

    // Iterate through the original string while applying replacements.
    let currentIndex = 0;
    while (currentIndex < stringLength) {
        // If there is a valid replacement at the current index, add the target string to the result array.
        if (replacementIndexArray[currentIndex] >= 0) {
            const replacementIndex = replacementIndexArray[currentIndex];
            resultArray.push(targetArray[replacementIndex]);

            // Move the current index ahead by the length of the source string that was replaced.
            currentIndex += sourceArray[replacementIndex].length;
        } else {
            // Otherwise, add the current character to the result array and move to the next character.
            resultArray.push(originalString[currentIndex++]);
        }
    }

    // Join the result array into a single string and return it.
    return resultArray.join('');
}

```

Time and Space Complexity

The given Python code snippet is designed to replace parts of a string `s` with alternate strings provided in the `targets` list. The replacements are conditional on the substrings in `s` starting at indices found in `indices` matching the corresponding strings in `sources`. Here is the computational complexity analysis of the code:

Time Complexity

The time complexity of the function is determined by several operations:

1. **Iterating Over `indices` and `sources`:** There is an initial loop that iterates through the zipped `indices` and `sources`. This takes `O(m)` time, where `m` is the number of elements in `indices` (and also `sources` and `targets`).

2. **String Matching with `startswith`:** Inside the loop, there is a call to the `startswith` function. This has a worst-case time complexity of `O(len(src))` for each invocation, which can be up to `O(n)` in the worst case (where `n` is the length of the string `s`). Therefore, in the worst case, this part of the loop could have a time complexity of `O(m * n)`.

3. **Building the Result String:** After the initial loop, the function iterates through string `s` and constructs the answer. In the worst case, each character could potentially be copied individually (when there are no matches), resulting in a time complexity of `O(n)`.

4. **Appending to the `ans` List and Join Operation:** The append operation is `O(1)` for each character or replacement string, but the `join` operation at the end is `O(n)` since it iterates over the entire list of characters and concatenates them into a new string.

Considering these parts together, the total time complexity is the sum of the complexities of these steps, which is `O(m * n) + O(n) + O(n)`. `O(m * n)` is likely the dominating term here, so the overall time complexity can be considered `O(m * n)`.

Space Complexity

The space complexity of the function is determined by the additional memory space used, apart from the input. The main extra storage used is:

1. **Array `d`:** The array `d` has the same length as the input string `s`, i.e., `O(n)`.

2. **List `ans`:** The list `ans` is used to construct the resulting string. In the worst case, it could hold `n` characters plus the length of all `targets` strings if every source is found and replaced. Assuming the sum of the lengths of all `targets` is `t`, the space used by `ans` could be up to `O(n + t)`.

Therefore, the overall space complexity is the largest of the space used by `d` and `ans`, leading to a total complexity of `O(n + t)` since `t` can be larger than `n`.