1055. Shortest Way to Form String

Medium Topics Companies Hint String

Problem Description

The problem involves finding how to form a given `target` string by concatenating subsequences of a `source` string. A subsequence can be obtained by deleting zero or more characters from a string without changing the order of the remaining characters. We need to determine the minimum number of these subsequences from `source` needed to create the `target` string. If it's not possible to form the `target` from subsequences of `source`, the function should return `-1`.

To visualize the problem, think about how you can create the word "cat" using letters from the word "concentrate". You can select `c`, omit `o`, pick `n`, skip `n`, `c`, `e`, `n`, and then pick `t` and skip `r`, `a`, `t`, `e`. That forms one subsequence "cat". Moreover, if you had a target like "catat", you'd need two subsequences from "concentrate" to form it—"cat" and "at".

Intuition

The intuition for the solution is both an idea that we can iterate over the `source` and `target` strings simultaneously, matching characters from `target` and moving through `source`. We do this in a loop that continues until we've either created the `target` string or determined it's impossible.

For each iteration (which corresponds to constructing a single subsequence), we do the following:

1. Start from the beginning of the `source` string and look for a match for the current character in `target`.
2. Each time we find a match, we move to the next character in `target` but continue searching from the current position in `source`.
3. If we reach the end of the `source` without finding the next character in `target`, we start over from the beginning of `source` and increment our subsequence count.
4. If when restarting the `source` string we don't make any progress in `target` (meaning we didn't find even the next character in the target), we conclude that the `target` cannot be formed and return `-1`.

The concept is similar to using multiple copies of the `source` string side by side, and crossing out characters as we match them to `target`. Whenever we reach the end of a `source` string copy and still have characters left to match in `target`, we move on to the next copy of `source`, symbolizing this with an increment in our subsequence counter. This continues until we've matched the entire `target` string or have verified that it's impossible.

Solution Approach

The solution uses a two-pointer technique to iterate through both the `source` and `target` strings. One pointer (`i`) traverses the `source` string, while the other pointer (`j`) iterates over the `target` string. Here's a step-by-step breakdown of the key components of the solution:

1. **Function `f(i, j)`:** This is a helper function that takes two indices, `i` for the `source` and `j` for the `target`. The purpose of `f` is to try to match as many characters of `target` starting from index `j` with the `source` starting from index `i` until we reach the end of `source`. The function runs a `while` loop until either `i` or `j` reaches the end of their respective strings. Inside this loop:
   - If the characters at `source[i]` and `target[j]` match, increment `j` to check for the next character in `target`.
   - Whether or not there is a match, always increment `i` because we can skip characters in `source`.
   - The function returns the updated index `j` after traversing through `source`.
2. **Main Algorithm:** Once we have our helper function, the main algorithm proceeds as follows:
   - We initialize two variables, `ans` and `i` as the lengths of `source` and `target` respectively, and `ans` and `j` to keep track of the number of subsequences needed and the current index in `target`.
   - We use a `while` loop that continues as long as `j < n`, meaning there are still characters in `target` that have not been matched.
   - Inside the `while` loop, we call our helper function `f(0, j)` which tries to match `target` starting from the current `j` index with `source` starting from `0`. If the returned index `k` is the same as `j`, it means no further characters from `target` could be matched and we return `-1` as it's impossible to form `target`.
   - If `k` is different from `j`, this means we've managed to match some part of `target`, and we update `j` to `k` and increment `ans` to signify the creation of another subsequence from `source`.
   - The process repeats until all characters of `target` are matched.
3. **Return Value:** The loop ends with two possibilities; either we were able to form `target` successfully, hence we return the `ans` which is the count of subsequences needed, or we determined that `target` cannot be formed from `source` and returned `-1`.

Complexity Analysis

- **Time Complexity:** $O(n * m)$, where n is the length of `source` and m is the length of `target`. In the worst case, we iterate through the entire `source` for every character in `target`.
- **Space Complexity:** $O(1)$, we only use a fixed amount of extra space for the pointers and the `ans` variable regardless of the input size.

By thoroughly understanding the definition of a subsequence and carefully managing the iteration through both strings, this solution efficiently determines the minimum number of subsequences of `source` required to form `target` or establishes that it's not possible.

Example Walkthrough

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

Example:

- `source`: "abcab"
- `target`: "abccba"

Walkthrough:

1. Initialize the count of subsequences (`ans`) needed to 0 and the index `j` in the `target` to 0.
2. Since `j < n` (where `n` is the length of `target`), start the iteration and call the helper function `f` with `f(0, 0)`.
3. Inside `f(0, 0)`, iterate over `source` and `target`. For each character in `source`, check if it matches the current `target[j]`.
   - For `source[0]` = 'a' and `target[0]` = 'a', there's a match, increment `j` to 1 (next character in `target`).
   - Continue to `source[1]` = 'b', which matches `target[1]` = 'b', increment `j` to 2.
   - Continue to `source[2]` = 'c', which matches `target[2]` = 'c', increment `j` to 3.
   - The character `source[3]` = 'a' does not match `target[3]` = 'c', so just increment `i`.
   - The character `source[4]` = 'b' does not match `target[3]` either, increment `i` again and now `i` reaches the end of `source`.
4. The `f` function returns `j` which is now 3. Since `j` has increased from 0 to 3, one subsequence "abc" has been matched from `source`.
   - Increment `ans` to 1 and start matching the next subsequence with `f(0, 3)`.
5. In the second call to `f(0, 3)`, we iterate from the start of `source` again:
   - Skip `source[0]` = 'a', since it doesn't match `target[3]` = 'c'.
   - Skip `source[1]` = 'b', since it doesn't match `target[3]` either.
   - `source[2]` matches `target[3]`, so increment `j` to 4.
   - `source[3]` matches `target[4]`, increment `j` to 5.
   - No more characters in `source` match `target[5]` = 'a', but once we reach the end of `source`, `f` returns `j` which is now 5.
6. Increment `ans` to 2 and start matching the last character with `f(0, 5)`.
7. In the third call to `f(0, 5)`, the first character `source[0]` = 'a' matches the last character `target[5]` = 'a'. The `j` is incremented to 6, which is the length of `target`, so the entire target string has been matched.
8. Increment `ans` to 3 which is our final answer.

Return:

The function would return `3` as it takes three subsequences of `source` to form the `target` string "abccba".

This example collapses the entire iteration into a concise explanation, demonstrating how the algorithm works in practice and matches subsequences in the source to form the target string.

Solution Implementation

Python

```
class Solution:
    def shortestWay(self, source: str, target: str) -> int:
        # Helper function to find the first unmatched character in 'target'
        # starting from 'target_index' by iterating through 'source'.
        def find_unmatched_index(source_index, target_index):
            # Iterate over both 'source' and 'target' strings.
            while source_index < len_source and target_index < len_target:
                # If the current characters match, move to the next character in 'target'.
                if source[source_index] == target[target_index]:
                    target_index += 1
                # Move to the next character in 'source'.
                source_index += 1
            # Return the index in 'target' where the characters stop matching.
            return target_index

        # Initialize the length variables of 'source' and 'target'.
        len_source, len_target = len(source), len(target)

        # Initialize 'subsequences_count' to 0 to count the subsequences of 'source' needed.
        subsequences_count = 0

        # Initialize 'target_index' to keep track of progress in the 'target' string.
        target_index = 0

        # Main loop to iterate until the entire 'target' string is checked.
        while target_index < len_target:
            # Find the index of the first unmatched character after 'target_index'.
            unmatched_index = find_unmatched_index(0, target_index)

            # Check if 'target_index' did not move forward; if so, 'target' cannot be constructed.
            if unmatched_index == target_index:
                return -1

            # Update 'target_index' to the index of the first unmatched character.
            target_index = unmatched_index

            # Increment the count of subsequences used.
            subsequences_count += 1

        # Return the total number of subsequences from 'source' needed to form 'target'.
        return subsequences_count
```

Java

```
class Solution {
    // Method to find the minimum number of subsequences of 'source' which concatenate to form 'target'
    public int shortestWay(String source, String target) {
        // 'sourceLength' is the length of 'source', 'targetLength' is the length of 'target'
        int sourceLength = source.length(), targetLength = target.length();
        // 'numSubsequences' will track the number of subsequences used
        int numSubsequences = 0;
        // 'targetIndex' is used to iterate through the characters of 'target'
        int targetIndex = 0;

        // Continue until the whole 'target' string is covered
        while (targetIndex < targetLength) {
            // 'tempTargetIndex' is used to iterate through characters of 'source'
            int sourceIndex = 0;
            // 'isSubsequenceFound' flags if a matching character was found in the current subsequence iteration
            boolean isSubsequenceFound = false;

            // Loop through both 'source' and 'target' strings to find subsequence matches
            while (sourceIndex < sourceLength && targetIndex < targetLength) {
                // If the characters match, move to the next character in 'target'
                if (source.charAt(sourceIndex) == target.charAt(targetIndex)) {
                    isSubsequenceFound = true; // A match in the subsequence was found
                    targetIndex++;
                }
                // Always move to the next character in 'source'
                sourceIndex++;
            }

            // If no matching subsequence has been found, it's not possible to form 'target'
            if (!isSubsequenceFound) {
                return -1;
            }

            // A subsequence that contributes to 'target' was used, so increment the count
            numSubsequences++;
        }

        // Return the minimum number of subsequences needed to form 'target'
        return numSubsequences;
    }
}
```

C++

```
class Solution {
public:
    // Function to find the minimum number of subsequences of 'source' required to form 'target'.
    int shortestWay(string source, string target) {
        int sourceLength = source.size(), targetLength = target.size(); // Source and target lengths
        int subsequencesCount = 0; // Initialize the count of subsequences needed
        int targetIndex = 0; // Pointer for traversing the target string

        // Loop until the entire target string is covered
        while (targetIndex < targetLength) {
            int sourceIndex = 0; // Reset source pointer for each subsequence iteration
            bool subsequenceFound = false; // Flag to check if at least one matching character is found in this iteration

            // Traverse both source and target to find the subsequence
            while (sourceIndex < sourceLength && targetIndex < targetLength) {
                // If the characters match, move pointer in target string to find the next character
                if (source[sourceIndex] == target[targetIndex]) {
                    subsequenceFound = true;
                    ++targetIndex;
                }
                ++sourceIndex; // Always move to the next character in the source string
            }

            // If no matching character was found, it's impossible to form target from source
            if (!subsequenceFound) {
                return -1;
            }

            ++subsequencesCount; // A new subsequence is found for this iteration
        }

        // Return the total count of subsequences required
        return subsequencesCount;
    }
};
```

TypeScript

```
// Function to find the minimum number of subsequences of 'source' required to form 'target'.
function shortestWay(source: string, target: string): number {
    let sourceLength: number = source.length; // Source length
    let targetLength: number = target.length; // Target length
    let subsequencesCount: number = 0; // Initialize the count of subsequences needed
    let targetIndex: number = 0; // Pointer for traversing the target string

    // Loop until the entire target string is covered
    while (targetIndex < targetLength) {
        let sourceIndex: number = 0; // Reset source pointer for each subsequence iteration
        let subsequenceFound: boolean = false; // Flag to check if at least one matching character is found in this iteration

        // Traverse both source and target to find the subsequence
        while (sourceIndex < sourceLength && targetIndex < targetLength) {
            // If the characters match, move pointer in target string to find the next character
            if (source.charAt(sourceIndex) === target.charAt(targetIndex)) {
                subsequenceFound = true;
                targetIndex++; // Move to the next character in target
            }
            sourceIndex++; // Always move to the next character in the source string
        }

        // If no matching character was found, it's impossible to form target from source
        if (!subsequenceFound) {
            return -1;
        }

        subsequencesCount++; // A new subsequence is found for this iteration
    }

    // Return the total count of subsequences required
    return subsequencesCount;
}


class Solution:
    def shortestWay(self, source: str, target: str) -> int:
        # Helper function to find the first unmatched character in 'target'
        # starting from 'target_index' by iterating through 'source'.
        def find_unmatched_index(source_index, target_index):
            # Iterate over both 'source' and 'target' strings.
            while source_index < len_source and target_index < len_target:
                # If the current characters match, move to the next character in 'target'.
                if source[source_index] == target[target_index]:
                    target_index += 1
                # Move to the next character in 'source'.
                source_index += 1
            # Return the index in 'target' where the characters stop matching.
            return target_index

        # Initialize the length variables of 'source' and 'target'.
        len_source, len_target = len(source), len(target)

        # Initialize 'subsequences_count' to 0 to count the subsequences of 'source' needed.
        subsequences_count = 0

        # Initialize 'target_index' to keep track of progress in the 'target' string.
        target_index = 0

        # Main loop to iterate until the entire 'target' string is checked.
        while target_index < len_target:
            # Find the index of the first unmatched character after 'target_index'.
            unmatched_index = find_unmatched_index(0, target_index)

            # Check if 'target_index' did not move forward; if so, 'target' cannot be constructed.
            if unmatched_index == target_index:
                return -1

            # Update 'target_index' to the index of the first unmatched character.
            target_index = unmatched_index

            # Increment the count of subsequences used.
            subsequences_count += 1

        # Return the total number of subsequences from 'source' needed to form 'target'.
        return subsequences_count
```

Time and Space Complexity

Time Complexity

The primary function of the algorithm, `shortestWay`, iterates over the `target` string while repeatedly scanning the `source` string to find subsequences that match the `target`. The function `f(i, j)` is called for each subsequence found and runs in a while loop that continues until either the end of the `source` or `target` string is reached. The worst-case scenario occurs when every character in the `source` has to be visited for every character in the `target`.

Given:

- n is the length of `source`
- m is the length of `target`

The worst-case time complexity can be roughly bounded by $O(n + m)$ since, in the worst case, the substring search could traverse the entire source string for each character in the target string.

Space Complexity

The space complexity of the algorithm is $O(1)$ as it only uses a fixed number of integer variables `n`, `m`, `ans`, `j`, and `i`, and does not allocate any additional space proportional to the input size.