1309. Decrypt String from Alphabet to Integer Mapping

Easy String

Leetcode Link

Problem Description

The given problem defines a unique mapping system where digits are used to represent English lowercase letters, with some digits followed by a '#' symbol to represent letters beyond 'i'. Specifically, the digits '1' to '9' correspond to the letters 'a' to 'i', and the representation for letters 'j' to 'z' is given as two digits followed by a '#', starting with '10#' for 'j' and ending with '26#' for 'z'.

Given a string `s` that is formed by these digits and '#' characters, the objective is to convert or map the string back to its English lowercase letter representation. For example, if the string `s` is "10#11#12", the output should be "jkl".

This mapping is bijective, meaning there will always be a unique solution to the mapping. One key detail stated in the problem is that test cases are generated in a way that ensures a unique mapping is always possible.

Intuition

To achieve the conversion from the given string `s` to the lowercase English letters, we need to traverse the string and determine if we are dealing with a one-digit number (which maps to 'a' to 'i') or a two-digit number followed by a '#' (which maps to 'j' to 'z').

Our approach is as follows:

1. We iterate over the string `s` starting from the beginning.
2. At each step, we check if there's a '#' character two positions ahead. If there is, it means the current and the next character form a two-digit number that corresponds to a letter from 'j' to 'z'. We then convert these two digits (and skip the '#' symbol) into the respective letter.
3. If there's no '#' two positions ahead, we treat the current character as a one-digit number and convert it to the respective letter from 'a' to 'i'.
4. We continue this until the end of the string, appending each translated letter to a result list.
5. Once the traversal is done, we join all elements of the result list to form the final string.

This ensures that we are able to sequentially convert each number or number combination into its corresponding character, thereby obtaining the lowercased English letter representation of the input string.

Solution Approach

The solution's approach is rather straightforward and employs a simple while loop to iterate over the string. Let's walk through the implementation based on the provided Python code:

1. Define a helper function named `get`, which takes a string `s` representing a number and converts it into a character. The function does this by first converting the string to an integer, then adding it to the ASCII value of 'a', and subtracting 1. This result is then converted back to a char using the `chr` function, aligning '1' with 'a', '2' with 'b', etc. The logic can be expressed by: `chr(ord('a') + int(s) - 1)`.
2. Initialize two variables, `i` and `n`, where `i` will be used to iterate over the string `s` and `n` holds the length of the string.
3. Start a while loop that continues as long as `i` is smaller than `n`. This loop will process each character or pair of characters followed by a '#' one at a time.
4. Inside the loop, there's an if statement that checks if the current position `i` plus 2 is within bounds of the string and if there is a '#' character at this position. This condition confirms that we have a two-digit number that needs to be mapped to a character from 'j' to 'z'.
5. If the condition is true, append the letter obtained from the helper function `get` by passing the substring from `i` to `i + 2`. Then increment `i` by 3 to account for the two digits and the '#' we just processed.
6. If the condition is false, it suggests we are currently handling a single digit. Append the result of the helper function `get` with the single character at position `i`. Increment `i` by 1 to move to the next character.
7. After the loop exits, use the `join` method on an empty string to concatenate all the elements in the result list `res`. This provides us with the final decoded string.
8. Return the decoded string.

Notice how the solution ensures we correctly interpret characters that are either single-digit numbers or two-digit numbers followed by '#' without the need for separate parsing steps. The use of array slicing and the helper function makes the code more readable and maintainable.

Here's a step-by-step explanation of how the algorithm works on the input "12#11#10":

- Initialize `i` to 0 and `n` to the length of the string (which is 8).
- Check the `i+2` position for a '#', which is true at `i = 0`. Use `get('12')` to map to 'l' and increment `i` by 3.
- Now `i` is 3, and `i+2` is "11#", so `get('11')` maps to 'k' and `i` becomes 6.
- Finally, `i` is 6, and `i+2` hits "10#", so `get('10')` maps to 'j' and `i` becomes 9, which breaks the loop.
- The result list `res` has ['l', 'k', 'j'], which joins to `"ljk"`.

By following this process, the solution efficiently decodes the entire string into the format we need.

Example Walkthrough

Let's apply the solution approach step-by-step on a smaller example input: `"2#5"`.

- We initialize `i` to 0 and `n` to 4, which is the length of the string.
- 1st iteration (`i = 0`):
  - We check if there is a '#' at position `i+2`. However, `i+2` is 2 and `s[2]` is not '#', it's '5'. Therefore, we process a single-digit number.
  - We call `get('2')`, which translates to 'b' using our custom function.
  - Increment `i` by 1 to move to the next character.
- 2nd iteration (`i = 1`):
  - Now `i` is 1. Again, we check if there is a '#' at position `i+2`. This time, `i+2` is 3 and `s[3]` is '#', indicating a two-digit number followed by '#'.
  - We call `get(s[1:1+2])`, which means `get('25')`, translating to 'y' using our custom function.
  - Increment `i` by 3 to move past the two-digit number and the '#'.
- The while loop exits as `i` (now 4) is not smaller than `n`.
- We finally join ['b', 'y'] to form the output string, resulting in `"by"`.

In this example, our first step was to convert the single digit '2' into 'b'. Then, we correctly identified '25#' as a representation of 'y' and completed our decoding process. The final result, for the input string `"2#5"`, using the provided solution approach, is `"by"`.

Python Solution

```python
class Solution:
    def freqAlphabets(self, string: str) -> str:
        # Helper function to convert a string number to its corresponding alphabet
        def decode_to_char(s: str) -> str:
            # The alphabet starts from 'a', hence the offset is 'a' + int(s) - 1
            return chr(ord('a') + int(s) - 1)

        # Initialize index and the length of the input string
        index, length_of_string = 0, len(string)
        # Initialize an empty list to store the resulting characters
        result = []

        # Iterate over the input string
        while index < length_of_string:
            # Check if a '#' follows two characters to detect if it's a two-digit number
            if index + 2 < length_of_string and string[index + 2] == '#':
                # If true, append the decoded character from the two-digit number to the result
                result.append(decode_to_char(string[index: index + 2]))
                # Move the index by 3 positions forward, skipping over the two digits and the '#'
                index += 3
            else:
                # If false, it's a single-digit number which is directly appended after decoding
                result.append(decode_to_char(string[index]))
                # Move the index by 1 position forward
                index += 1

        # Join the list of characters into a string and return it
        return ''.join(result)
```

Java Solution

```java
class Solution {

    // Function to decode the string s as per the given pattern
    public String freqAlphabets(String s) {
        int index = 0; // Initialize index to keep track of the current position in the string
        int strLength = s.length(); // Store the length of the input string for boundary checking
        StringBuilder decodedString = new StringBuilder(); // StringBuilder to append the decoded characters

        // Iterate through the string until all characters are processed
        while (index < strLength) {
            // Check if the current character is part of a two-digit number followed by a '#'
            if (index + 2 < strLength && s.charAt(index + 2) == '#') {
                // Decode the two-digit number and append the corresponding character
                decodedString.append(decode(s.substring(index, index + 2)));
                index += 3; // Skip the next two characters and the '#'
            } else {
                // Decode a single-digit number and append the corresponding character
                decodedString.append(decode(s.substring(index, index + 1)));
                index += 1; // Move to the next character
            }
        }

        // Convert the StringBuilder to a String and return it
        return decodedString.toString();
    }

    // Helper function to convert a numeric string to its respective alphabet character
    private char decode(String numericString) {
        // Subtract '1' because 'a' corresponds to '1', 'b' to '2', and so on.
        // Parse the numeric string to an integer, subtract 1, and then add it to the char 'a'
        return (char) ('a' + Integer.parseInt(numericString) - 1);
    }
}
```

C++ Solution

```cpp
#include <iostream>
#include <string>
#include <vector>

std::string freqAlphabets(const std::string& encoded_str) {
    // Get the length of the input string.
    size_t length = encoded_str.length();
    // Initialize a vector to build the decoded characters.
    std::vector<std::string> decoded_chars;
    // Initialize an index to iterate over the input string.
    size_t index = 0;

    // Iterate over the input string.
    while (index < length) {
        // Check if there is a '#' two characters ahead.
        if (index + 2 < length && encoded_str[index + 2] == '#') {
            // If so, take the two characters before '#' and push them onto the vector.
            decoded_chars.push_back(encoded_str.substr(index, 2));
            // Skip the next two characters and the '#' by incrementing the index by 3.
            index += 3;
        } else {
            // Otherwise, push the current character onto the vector.
            decoded_chars.push_back(encoded_str.substr(index, 1));
            // Move to the next character by incrementing the index by 1.
            index++;
        }
    }

    // Initialize the decoded string.
    std::string decoded_str;

    // Convert each encoded character or pair in the vector to the corresponding letter.
    for (const auto& encoded_char : decoded_chars) {
        // 'a' - 2 provides the offset necessary because 'a' starts at 97. Adding encoded_value gives us the ASCII value.
        char decoded_char = static_cast<char>('a' + std::stoi(encoded_char) - 1);
        // Add the decoded character to the decoded string.
        decoded_str += decoded_char;
    }

    // Return the final decoded string.
    return decoded_str;
}

// Example of using the function.
int main() {
    std::string encoded_string = "12#14#3#";
    std::string decoded = freqAlphabets(encoded_string);
    std::cout << "Decoded string: " << decoded << std::endl;
    return 0;
}
```

Typescript Solution

```typescript
function freqAlphabets(encodedStr: string): string {
    // Get the length of the input string.
    const length = encodedStr.length;
    // Initialize an array to build the decoded characters.
    const decodedChars: string[] = [];
    // Initialize an index to iterate over the input string.
    let index = 0;

    // Iterate over the input string.
    while (index < length) {
        // Check if there is a '#' two characters ahead.
        if (encodedStr[index + 2] === '#') {
            // If so, slice the two characters before '#' and add them to the array.
            decodedChars.push(encodedStr.slice(index, index + 2));
            // Skip the next two characters and the '#' by incrementing the index by 3.
            index += 3;
        } else {
            // Otherwise, add the current character to the array as is.
            decodedChars.push(encodedStr[index]);
            // Move to the next character by incrementing the index by 1.
            index += 1;
        }
    }

    // Map each character or pair in the array to its corresponding decoded character.
    // Convert each encoded character to its respective letter in the alphabet.
    return decodedChars.map(encodedChar =>
        String.fromCharCode('a'.charCodeAt(0) + Number(encodedChar) - 1)
    ).join(''); // Join the array of characters to form the final decoded string.
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed by looking at the number of iterations it makes over the string `s`. The code uses a while loop to iterate over each character of the string. The key operation within the loop is checking whether the next character is a special '#' and then either taking one or two characters to convert them into a letter. These operations are all constant time, and since the loop runs for each character or pair of characters marked with a '#', the time complexity is $O(n)$, where n is the length of the string `s`.

Space Complexity

The code uses additional memory for the `res` list to store the result. In the worst case, where there are no '#' characters, the `res` list would have as many characters as the original string `s`. Therefore, the space complexity is also $O(n)$, where n is the length of the string `s`. In the case of the given code, the space complexity does not grow more than $O(n)$ as we are only creating a result list that, at most, will have the same length as the input.