2325. Decode the Message

Easy Hash Table String

Problem Description

The given problem presents a scenario where a secret message needs to be decoded using a cipher key. The cipher key is a string that contains all 26 lowercase English letters, where the first occurrence of each letter represents its position in the cipher table. The process of decoding involves creating a substitution table based on the first appearance of each letter in the key and then mapping that to the regular English alphabet in order (a-z).

To decode the message, each letter in the message is substituted with the corresponding letter from the substitution table. It is crucial to note that spaces remain unchanged. The goal is to apply this substitution process to the entire message to retrieve the original text.

Intuition

The intuition behind the solution is to map each letter from the key to its position in the alphabet sequence. This is the basis for creating the substitution table. In the Python solution, a dictionary is used to store this mapping. The steps to arrive at the solution include:

1. Initialize a dictionary that will hold the mapping (`d = {" ": " "}`) allowing spaces to be mapped to themselves as per the problem statement.
2. Iterate over each character in the key string and fill the dictionary with unique letters, assigning them to the corresponding order in the English alphabet (using the `ascii_lowercase[i]`). The variable `i` is used to keep track of the position in the alphabet.
3. Use the dictionary to translate each character in the message. If the character is a space, it maps to a space; otherwise, it will be substituted according to the key-to-alphabet mapping.
4. Finally, the translated characters are joined to form the decoded message which is returned as the solution.

Solution Approach

The solution leverages a few key concepts, primarily dictionary mapping and string iteration in Python.

1. **Dictionary for Mapping:** The use of a dictionary, `d`, is central to this approach. Dictionaries in Python are key-value pairs that allow for quick look-ups, insertions, and updates. Given that spaces are mapped to themselves (`d = {" ": " "}`), the dictionary serves as the substitution table.

2. **String Iteration:** The code iterates over the `key` string using a `for` loop. For each character `c` found in `key`, it checks whether the character is already in the dictionary. If it is not, it means this is the first occurrence of that character and, thus, should be added to the dictionary.

3. **Substitution Logic:** The mapping to the English alphabet is handled by `ascii_lowercase[i]`, which returns the `i`-th letter from the English alphabet (contained in the `string` module from Python's standard library). The variable `i` is incremented only when a new mapping is created. This ensures that each unique letter in `key` is associated with a unique letter of the alphabet in order.

4. **Message Decoding:** Next, the code decodes `message` by iterating over every character in it. It uses the dictionary `d` to find the substitution for each character, compiles these using a list comprehension, and joins them with `"".join(d[c] for c in message)` to form the final decoded message string.

5. **Ignoring Duplicate Letters:** Since only the first occurrence of each letter is considered, subsequent ones are ignored, effectively skipping them during the dictionary mapping process.

This solution is efficient as mapping and look-up operations in dictionaries are on average O(1) in complexity. The overall decoding process depends on the lengths of the `key` and `message` strings, making the time complexity linear with respect to the size of the input, or O(N+M) where N is the length of `key` and M is the length of `message`.

Example Walkthrough

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

Suppose we are given the following cipher key:

`key = "thequickbrownfoxjumpsoverlazydg"`

and we want to decode this message:

`message = "vkbs bs t suepr"`

Here's how the solution would work step by step:

1. **Dictionary for Mapping:** We initialize a dictionary `d` that will hold the mapping of each character. We start with mapping spaces to themselves: `d = {" ": " "}`.

2. **String Iteration (Creating the Substitution Table):**

   - We start by iterating over the `key` string.
   - For each character `c` in the key, we check if it's already in the dictionary `d`.
   - Our dictionary starts as `d = {" ": " "}`.

   Iterating through the `key`:

   - The first character is `t`, which is not in `d`, so we add it: `d = {" ": " ", "t": "a"}`.
   - The second character is `h`, not in `d` either, so we add it: `d = {" ": " ", "t": "a", "h": "b"}`.
   - This process continues with each unique letter until the dictionary is filled with the first occurrences: `... {"x": "m", "j": "n", "u": "o", "m": "p", "p": "q", "s": "r", "o": "s", "v": "t", "e": "u", "r": "v", "l": "w", "a": "x", "z": "y", "y": "z"}`.
   - We ignore any subsequent occurrences of letters we've seen before.
3. **Substitution Logic:**

   - We have the `ascii_lowercase` variable which is a string `"abcdefghijklmnopqrstuvwxyz"`.
   - Each time we add a new key-value pair to the dictionary, the value is the next letter from `ascii_lowercase` that has not yet been used.
4. **Message Decoding:**

   - Now, we decode the `message` by looking at each character and using our dictionary `d` to find what it maps to.
   - Translating our example message: `"vkbs bs t suepr"`
     - 'v' becomes 't'
     - 'k' becomes 'h'
     - 'b' maps to 'i'
     - 's' maps to 'r'
     - and so on.
   - The decoded message is formed by substituting each character in the original `message` with its mapped value from `d`.
5. **Resulting Output:**

   - After applying the substitution to each character, we get the final message: `"this is a super"`
   - The message has been successfully decoded!

This example clearly demonstrates how the dictionary is created based on the first unique occurrence of each letter in the cipher key and how the message characters are then substituted according to this dictionary to reveal the original text.

Solution Implementation

Python

```python
from string import ascii_lowercase

class Solution:
    def decodeMessage(self, key: str, message: str) -> str:
        # Dictionary that maps each unique letter in 'key' to the corresponding letter in the alphabet
        # Initial mapping for space character is included as it maps to itself
        char_mapping = {" ": " "}

        # Index to keep track of the next letter in the alphabet to be used for mapping
        next_alpha_index = 0

        # Iterate through each character in the 'key'
        for char in key:
            # Check if the character is not already in the mapping dictionary
            if char not in char_mapping:
                # Map the character to the next available letter in the alphabet
                char_mapping[char] = ascii_lowercase[next_alpha_index]
                # Move to the next letter in the alphabet
                next_alpha_index += 1

        # Convert the message using the generated mapping by replacing each character
        # with its corresponding mapped character
        decoded_message = "".join(char_mapping[char] for char in message)

        return decoded_message
```

Java

```java
class Solution {

    // Decodes a message using a substitution cipher provided by the 'key'.
    public String decodeMessage(String key, String message) {
        // Array to hold the substitution cipher mapping.
        char[] decoder = new char[128];
        // Preserving the space character.
        decoder[' '] = ' ';

        // Initialize variables for tracking indices in 'key' and 'decoder'.
        for (int keyIndex = 0, decoderIndex = 0; keyIndex < key.length(); ++keyIndex) {
            // Get the current character from the key.
            char currentChar = key.charAt(keyIndex);
            // If current character is not already present in the 'decoder' array, add it.
            if (decoder[currentChar] == 0) {
                // Map the current character to the next available character in the alphabet.
                decoder[currentChar] = (char) ('a' + decoderIndex++);
            }
        }

        // Convert the message into a char array for in-place decoding.
        char[] decodedMessage = message.toCharArray();

        // Loop through the message and decode each character.
        for (int messageIndex = 0; messageIndex < decodedMessage.length; ++messageIndex) {
            // Substitute the character using the decoder array.
            decodedMessage[messageIndex] = decoder[decodedMessage[messageIndex]];
        }

        // Return the decoded message as a string.
        return String.valueOf(decodedMessage);
    }
}
```

C++

```cpp
#include <string>
using namespace std;

class Solution {
public:
    string decodeMessage(string key, string message) {
        // Create a dictionary to hold the character mapping.
        // Initialize a lookup array to zero, assuming ASCII values.
        char dictionary[128] = {};

        // Map space to itself since it is not to be encoded.
        dictionary[' '] = ' ';

        // Start with the first lowercase letter for substitution.
        char substitution_letter = 'a';

        // Iterate through the key and fill the dictionary for encoding.
        for (char& current_char : key) {
            // Check if character is already mapped; if not, map it
            if (!dictionary[current_char] && current_char >= 'a' && current_char <= 'z') {
                dictionary[current_char] = substitution_letter++;
                // Stop if all letters have been mapped
                if(substitution_letter > 'z') break;
            }
        }

        // Decode the message using the filled dictionary.
        for (char& current_char : message) {
            // Replace each character in the message with the mapped character.
            current_char = dictionary[current_char];
        }

        // Return the decoded message.
        return message;
    }
};
```

TypeScript

```typescript
// Function to decode a message using a substitution cipher provided by a key
function decodeMessage(key: string, message: string): string {
    // Dictionary to hold the key-value pairs for decoding
    const decoderMap = new Map<string, string>();

    // Loop through each character in the key to build the decoder map
    for (const char of key) {
        // If the character is a space or already exists in the map, skip it
        if (char === ' ' || decoderMap.has(char)) {
            continue;
        }

        // Map the character to its corresponding decoded alphabet
        // The decoded character is determined by the current size of the decoder map
        // 'a'.charCodeAt(0) converts the letter 'a' to its ASCII code,
        // then the size of the decoderMap is added to get the new character
        decoderMap.set(char, String.fromCharCode('a'.charCodeAt(0) + decoderMap.size));
    }

    // Ensure space is mapped to itself in the decoder map
    decoderMap.set(' ', ' ');

    // Transform the message: Split the message into characters, decode each character, and join the decoded characters
    return [...message].map(char => decoderMap.get(char)).join('');
}
```

```python
from string import ascii_lowercase

class Solution:
    def decodeMessage(self, key: str, message: str) -> str:
        # Dictionary that maps each unique letter in 'key' to the corresponding letter in the alphabet
        # Initial mapping for space character is included as it maps to itself
        char_mapping = {" ": " "}

        # Index to keep track of the next letter in the alphabet to be used for mapping
        next_alpha_index = 0

        # Iterate through each character in the 'key'
        for char in key:
            # Check if the character is not already in the mapping dictionary
            if char not in char_mapping:
                # Map the character to the next available letter in the alphabet
                char_mapping[char] = ascii_lowercase[next_alpha_index]
                # Move to the next letter in the alphabet
                next_alpha_index += 1

        # Convert the message using the generated mapping by replacing each character
        # with its corresponding mapped character
        decoded_message = "".join(char_mapping[char] for char in message)

        return decoded_message
```

Time and Space Complexity

The time complexity of the provided code is primarily determined by two operations: iterating through the `key` string, and building the decoded `message` string.

1. Iterating through the `key` string involves checking each character to see if it is already in the dictionary `d`. Each check is an O(1) operation due to the hash table underlying Python dictionaries. There are at most 26 unique letters to insert into the dictionary (ignoring spaces as they are hardcoded), so that part of the algorithm is O(26), which simplifies to O(1) since we ignore constants and non-dominant terms in Big O notation.

2. Converting the `message` involves a single pass through the message characters, each look-up in the dictionary `d` is O(1), and we perform a constant time operation for each character in the `message`. Therefore, if the message length is `n`, this part of the algorithm is O(n).

The combined time complexity is therefore O(n), where `n` is the length of the `message`, since this is the dominant term.

The space complexity is determined by the additional space used by the algorithm. Here, it is the space required to store the `d` dictionary and the output string.

- The dictionary `d` stores a mapping of characters to characters, and there are at most 26 letter mappings plus one space mapping, so it requires O(27) space, which simplifies to O(1).

- The space complexity for the output string is O(n), where `n` is the length of the `message`, because we're building a new string with the same length as the input message.

Therefore, the overall space complexity of the algorithm is O(n), where `n` is the length of the `message`.