1528. Shuffle String

Easy Array String

Problem Description

You are provided with two inputs: a string `s` and an integer array `indices`. The length of both `s` and `indices` is equal, implying that for every character in `s`, there is a corresponding element in `indices`. The problem requires you to 'shuffle' or rearrange the characters in `s` based on the values in `indices`. Specifically, each character from the original string `s` should be placed at a new position in a resulting string, such that if the `i`th character in the original string is `s[i]`, it will be moved to the position `indices[i]` in the shuffled string. The output should be the shuffled string obtained after rearranging the characters of `s` as stated.

For example, if `s = "abc"` and `indices = [1, 2, 0]`, the shuffled string would rearrange `a` to index `1`, `b` to index `2`, and `c` to index `0`, resulting in the string `"cab"`.

Intuition

To solve this problem, the intuition is to simulate the shuffling process described. Since you have a mapping from the original indices of `s` to the shuffled indices via the `indices` array, you can create a new list, initially filled with placeholders (for example, zeroes), with the same length as `s`. You can then iterate over the string `s` and for each character `s[i]`, place it at the index specified by `indices[i]` in the new list. This directly implements the shuffling process. Once this operation is completed for all characters, you simply need to join the list's elements to form the shuffled string.

The reason why using a list for the output is beneficial is because strings in Python are immutable, meaning you cannot change characters at specific indices without creating a new string. On the other hand, lists are mutable, allowing you to assign new values at specific indices, which is perfect for this shuffling algorithm. After the shuffling is done, the `join` method is used to convert the list of characters back into a string, because the final expected output is a string, not a list.

Solution Approach

The solution presented uses a straightforward array manipulation approach to implement the shuffle. Let's walk through the implementation step by step.

1. **Creating a placeholder list**: The first step in the solution is to create a new list `ans` with the same length as the input string `s`. This list is populated with zeros or placeholders, but the exact values don't matter since they'll be overwritten. The purpose of this list is to store the characters of `s` at their new positions. This is done by:

```
ans = [0] * len(s)
```

2. **Enumerating the original string**: We then use a loop to go through each character and its corresponding original index in the string `s`. This is achieved with the `enumerate()` function, which gives us each character `c` and its index `i`.

```
for i, c in enumerate(s):
```

3. **Placing each character at the new index**: Within the loop, we access the shuffling index for the current character from the `indices` list using `indices[i]`. The character `c` is then assigned to the placeholder list `ans` at this new index. The line of code that does this is:

```
    ans[indices[i]] = c
```

   By assigning to `ans[indices[i]]`, we ensure that each character from the original string `s` is placed according to the shuffled indices provided by the `indices` list.

4. **Converting the list back to a string**: After the loop completes, all of the characters are in their correct shuffled positions in the list `ans`. The last step is to join the list of characters into a single string, which gives us the shuffled string required by the problem. The `join()` method is perfectly suited for this, as it concatenates a list of strings into a single string without any separators (since we're dealing with single characters).

```
return ''.join(ans)
```

No additional data structures, patterns, or complex algorithms are necessary for this solution. By leveraging the mutable nature of lists and the ability to directly assign to indices, the task is done efficiently and in a manner that is easy to read and understand.

Example Walkthrough

Let's illustrate the solution approach by walking through a small example. Assume we have the following inputs: `s = "Leet"`, and `indices = [2, 0, 3, 1]`

Following the solution approach:

1. **Creating a placeholder list**: We create an empty list `ans` with the same length as `s`, filled with zeros (which will be placeholders in this case).

```
ans = [0] * len(s)  # ans = [0, 0, 0, 0]
```

2. **Enumerating the original string**: Using the `enumerate()` function to loop through `s`, we get index `i` and character `c` for each iteration.

   The `enumerate()` function would provide the following pairs: `(0, 'L')`, `(1, 'e')`, `(2, 'e')`, `(3, 't')`.

3. **Placing each character at the new index**: For each index-character pair `(i, c)`, we place character `c` at the index given by `indices[i]` in the list `ans`. The loop would look like the following:

   For `(0, 'L')`:

```
ans[indices[0]] = 'L'  # ans[2] = 'L', now ans = [0, 0, 'L', 0]
```

   For `(1, 'e')`:

```
ans[indices[1]] = 'e'  # ans[0] = 'e', now ans = ['e', 0, 'L', 0]
```

   For `(2, 'e')`:

```
ans[indices[2]] = 'e'  # ans[3] = 'e', now ans = ['e', 0, 'L', 'e']
```

   For `(3, 't')`:

```
ans[indices[3]] = 't'  # ans[1] = 't', now ans = ['e', 't', 'L', 'e']
```

   After this step, each character from the string `s` has been moved to its new index specified by `indices`.

4. **Converting the list back to a string**: All characters are now in the correct position in the list `ans`. We convert `ans` back to a string using the `join()` method:

```
shuffled_string = ''.join(ans)  # shuffled_string = "etLe"
```

The final output `shuffled_string` is `"etLe"`, which is the shuffled result of the string `"Leet"` according to the indices `[2, 0, 3, 1]`.

Solution Implementation

Python

```
class Solution:
    def restoreString(self, s: str, indices: list[int]) -> str:
        # Create a list to hold the characters in their restored positions
        restored_string = [''] * len(s)

        # Loop through each character and its intended index
        for char_index, character in enumerate(s):
            target_index = indices[char_index]
            # Place the character in the correct index of the restored string
            restored_string[target_index] = character

        # Join all characters to form the restored string and return it
        return ''.join(restored_string)

# The class can then be used as follows:
# solution = Solution()
# restored = solution.restoreString("codeleet", [4,5,6,7,0,2,1,3])
# print(restored)  # Output will be: "leetcode"
```

Java

```
class Solution {
    // Method to restore a string based on given indices
    public String restoreString(String inputString, int[] indices) {
        // Determine the length of the input string
        int stringLength = inputString.length();
        // Create a char array to store the rearranged characters
        char[] rearrangedCharacters = new char[stringLength];

        // Iterate through the indices array
        for (int i = 0; i < stringLength; ++i) {
            // Place the character from the input string at the correct position
            // as specified by the current element in the indices array
            rearrangedCharacters[indices[i]] = inputString.charAt(i);
        }

        // Convert the character array back to a string and return it
        return new String(rearrangedCharacters);
    }
}
```

C++

```
#include <string>
#include <vector>

class Solution {
public:
    // Function to restore a string based on the given indices
    std::string restoreString(std::string s, std::vector<int>& indices) {
        // Determine the length of the string
        int strLength = s.size();

        // Create an output string initialized with 'strLength' number of zeros
        std::string restoredString(strLength, '0'); // Use a character '0' for clear initialization

        // Iterate over each character in the original string
        for (int i = 0; i < strLength; ++i) {
            // Place the current character at the correct position in 'restoredString'
            restoredString[indices[i]] = s[i];
        }

        // Return the restored string
        return restoredString;
    }
};
```

TypeScript

```
/**
 * Function to restore a string based on given indices.
 * @param {string} str - The original string to be shuffled.
 * @param {number[]} indices - The array indicating the new order of letters.
 * @returns {string} - The restored string after shuffling.
 */
function restoreString(str: string, indices: number[]): string {
    // Create an array to hold the restored characters.
    const restoredStringArray: string[] = [];

    // Loop through each character in the input string.
    for (let i = 0; i < str.length; i++) {
        // Place the character at the correct index in the restored array.
        restoredStringArray[indices[i]] = str.charAt(i);
    }

    // Join the array elements to form the restored string and return it.
    return restoredStringArray.join('');
}
```

```
class Solution:
    def restoreString(self, s: str, indices: list[int]) -> str:
        # Create a list to hold the characters in their restored positions
        restored_string = [''] * len(s)

        # Loop through each character and its intended index
        for char_index, character in enumerate(s):
            target_index = indices[char_index]
            # Place the character in the correct index of the restored string
            restored_string[target_index] = character

        # Join all characters to form the restored string and return it
        return ''.join(restored_string)

# The class can then be used as follows:
# solution = Solution()
# restored = solution.restoreString("codeleet", [4,5,6,7,0,2,1,3])
# print(restored)  # Output will be: "leetcode"
```

Time and Space Complexity

Time Complexity

The time complexity of the provided function is `O(n)`, where `n` is the length of the string `s`. The function iterates through each character of the string exactly once, which means the number of operations grows linearly with the size of the input string.

Space Complexity

The space complexity of the function is also `O(n)`, since it creates a list `ans` of the same length as the input string to store the rearranged characters. This means the amount of additional memory used by the program is directly proportional to the input size.