2734. Lexicographically Smallest String After Substring Operation

Medium Greedy String

Leetcode Link

Problem Description

In the given problem, you are provided with a string `s` which consists only of lowercase English letters. You are allowed to perform a single operation which involves selecting a non-empty substring of `s` (it could be the entire string), and then replacing each character in that substring with the character that comes before it in the English alphabet. So, 'b' becomes 'a', 'c' becomes 'b', and interestingly, 'a' loops back around to become 'z'. Your task is to determine the lexicographically smallest string that can be obtained by performing this operation exactly once.

Recall that a substring is just a sequence of consecutive characters from the string, and that one string is lexicographically smaller than another if, at the first position where they differ, the character in the first string comes before the character in the second string in the alphabet.

Intuition

In solving this problem, we want to use the operation in such a way that we make the string as small as possible in lexographic order. The strategy here is to identify the first non-'a' character of the string because converting 'a' to 'z' would actually make the string lexicographically larger. When we find a non-'a' character, we want to shift it and all subsequent characters that are not 'a' so that the resulting string is lexicographically smallest.

The reason we stop at the next 'a' character after starting the shift is because changing any 'a' to a 'z' will make the string larger instead of smaller. For example, if we have the string "abc", changing it to "zbc" would make it larger lexicographically, whereas changing it to "aac" makes it smaller.

So, to break down the approach:

1. Scan the string from the beginning and locate the first character that is not 'a'.
2. Continue scanning until you reach an 'a' or the end of the string.
3. Replace all characters from the first non-'a' character to the character before the 'a' you've reached, by shifting them one place back in the alphabet.
4. If the entire string consists of 'a's, then simply change the very last character to 'z'.

This algorithm ensures that we make the minimal required changes to obtain the lexicographically smallest string possible by performing the operation exactly once.

Solution Approach

The solution follows a simple straight-forward approach which includes a single pass through the string and basic string manipulation operations. No complex data structures or algorithms are necessary for this implementation. The steps taken by the code can be outlined as follows:

1. Initialize an index variable `i` to 0. This variable will be used to find the position of the first non-'a' character in the string.
2. Use a `while` loop to skip any 'a's at the start of the string. The loop goes on until a non-'a' character is encountered or until we've reached the end of string. In each iteration, `i` is incremented by 1.
3. Checking if `i` is equal to the length of the string `n`. If yes, it means the string consists entirely of 'a's, so we replace the last character with 'z' and return the modified string.
4. If the string does not consist solely of 'a's, then initialize another variable `j` with the value of `i` and proceed to find the end of the contiguous non-'a' substring by incrementing `j` until an 'a' is found or the string ends.
5. Use string slicing to separate the string into three parts: the unchanged prefix `s[:i]`, the modified middle `[... for c in s[i:j]]` where each character is shifted back in the alphabet, and the unchanged suffix `s[j:]`.
6. For the middle part, a list comprehension is used along with the `chr` and `ord` functions to shift each character. The `ord` function gets the ASCII number for each character and then 1 is subtracted from that number because we want to replace each character by its predecessor in the English alphabet. After that, the resulting ASCII number is converted back to a character using the `chr` function.
7. Finally, the three parts of the string are concatenated and the resultant string is returned.

In summary, the solution walks through the string to find the optimal point at which to apply the operation, and then uses simple character manipulation to perform the operation and construct the resulting lexicographically smallest string.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the string "abcde".

1. We initialize `i` to 0. The string `s` is "abcde", and we are looking for the first occurrence of a non-'a' character.
2. We skip the first character since it's an 'a', so `i` becomes 1. The second character is 'b', which is not 'a', so we stop the while loop.
3. Since `i` is now 1 and not equal to the length of the string (which is 5), we know the string does not consist only of 'a's.
4. We then initialize `j` to 1 and start scanning for the next 'a'. We find that 'c', 'd', and 'e' are also not 'a', so `j` continues to increment.
5. We get to the end of the string and have not found another 'a'. So, `j` stops at the string length (5).
6. Now, we slice the string into three parts: `s[:1]` gives us "a", which is unchanged. We will modify `s[1:j]`, which is "bcde". `s[j:]` is an empty string since `j` is at string end.
7. For each character in "bcde", we convert it to its predecessor: 'b' to 'a', 'c' to 'b', 'd' to 'c', 'e' to 'd'. This is done using ASCII values, where we subtract 1 from each character's ASCII value to get the previous character.
8. We then concatenate the parts together, so "a" + "abcd" + "" becomes "aabcd".

Therefore, by following the steps of the solution approach, the lexicographically smallest string we can obtain from "abcde" by performing the operation exactly once is "aabcd".

Python Solution

```python
class Solution:
    def smallestString(self, s: str) -> str:
        # Length of the input string
        length_of_string = len(s)
        # Initialize index to start from the beginning
        index = 0

        # Skip all the 'a' characters from the start
        while index < length_of_string and s[index] == "a":
            index += 1

        # If the string is made up entirely of 'a's, then replace the last 'a' with 'z'
        if index == length_of_string:
            return s[:-1] + "z"

        # Find the index where 'a' appears after the consecutive sequence of non 'a' characters
        next_a_index = index
        while next_a_index < length_of_string and s[next_a_index] != "a":
            next_a_index += 1

        # Decrease every character in the substring from 'index' to 'next_a_index' by one
        # and replace that part of the string with the new substring
        return (s[:index] +
                "".join(chr(ord(char) - 1) for char in s[index:next_a_index]) +
                s[next_a_index:])

# Here's a brief explanation of how the function operates:
# 1. The function finds the first sequence of non 'a' characters.
# 2. It then reduces each character in this sequence by 1 lexicographically.
# 3. If the whole string contains only 'a' characters, it turns the last 'a' into a 'z'.
# 4. The updated string is returned as the result.
```

Java Solution

```java
class Solution {
    public String smallestString(String s) {
        int stringLength = s.length();
        int firstNonAIndex = 0;

        // Find the first instance of a character that is not 'a'
        while (firstNonAIndex < stringLength && s.charAt(firstNonAIndex) == 'a') {
            firstNonAIndex++;
        }

        // If there's no character other than 'a', replace the last 'a' with 'z'
        if (firstNonAIndex == stringLength) {
            return s.substring(0, stringLength - 1) + "z";
        }

        // Convert the string to a character array for manipulation
        char[] chars = s.toCharArray();

        // Start decreasing the value of characters until an 'a' is reached
        int reduceIndex = firstNonAIndex;
        while (reduceIndex < stringLength && chars[reduceIndex] != 'a') {
            chars[reduceIndex] = (char) (chars[reduceIndex] - 1);
            reduceIndex++;
        }

        // Return the new string constructed from the character array
        return String.valueOf(chars);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to construct the lexicographically smallest string
    // by changing characters to 'a' or 'z' as per the conditions given
    string smallestString(string s) {
        int strSize = s.size(); // Get the size of the string
        int index = 0;

        // Iterate through the string and skip all 'a'
        while (index < strSize && s[index] == 'a') {
            ++index;
        }

        // If the string is composed of 'a' only, change the last character to 'z'
        if (index == strSize) {
            s[strSize - 1] = 'z'; // Changing the last character to 'z'
            return s;
        }

        // Iterate through the string starting from the first character that was not 'a'
        // and decrement each character until we find 'a' or reach the end of the string
        while (index < strSize && s[index] != 'a') {
            s[index] = s[index] - 1; // Change character to previous one in alphabetical order
            ++index;
        }

        // Return the modified string which should be the lexicographically smallest string possible
        return s;
    }
};
```

Typescript Solution

```typescript
// Function to construct the lexicographically smallest string
// by changing characters to 'a' or 'z' according to the conditions given
function smallestString(s: string): string {
    let strSize: number = s.length; // Get the length of the string
    let index: number = 0;

    // Iterate through the string and skip all 'a'
    while (index < strSize && s[index] === 'a') {
        ++index;
    }

    // If the string is composed of 'a' only, change the last character to 'z'
    if (index === strSize) {
        s = s.substring(0, strSize - 1) + 'z'; // Changing the last character to 'z'
        return s;
    }

    // Iterate through the string starting from the first character that was not 'a'
    // and decrement each character until we find 'a' or reach the end of the string
    let chars: string[] = s.split(''); // Split the string into an array of characters
    while (index < strSize && chars[index] !== 'a') {
        chars[index] = String.fromCharCode(chars[index].charCodeAt(0) - 1); // Change character to previous one in alphabetical order
        ++index;
    }

    // Rejoin the array of characters into a modified string which should be the lexicographically smallest string possible
    return chars.join('');
}

// Example usage:
// let result: string = smallestString("abcde");
// console.log(result); // Outputs: "abbde"
```

Time and Space Complexity

Time Complexity:

The time complexity of this code is `O(n)`, where `n` is the length of the input string `s`.

- The first `while` loop runs in `O(n)` in the worst case when all characters are `"a"`. At best, it exits immediately if the first character is not `"a"`.
- The second `while` loop also runs in `O(n)` in the worst case, if there are no `"a"` characters following the first non-`"a"` character. At best, it exits immediately if the next character is `"a"`.
- The line with `join` and `chr(ord(c) - 1)` inside list comprehension again runs in `O(n)` because it iterates through the substring `s[i:j]`. This substring can potentially be the entire string `s` in the worst case.

These loops are sequential and not nested, so the time complexity remains `O(n)`.

Space Complexity:

The space complexity of the code is also `O(n)`.

- This is because the code creates a new string with the `join` operation, which can potentially contain as many characters as the original string `s` in the worst case.
- The space used to store indexes `i` and `j` is constant and does not scale with the size of the input string, therefore their contribution to space complexity is `O(1)`.

To sum up, the space complexity of the code is dominated by the space required for the new string generated in the `join` operation, which is `O(n)`.