899. Orderly Queue

Hard Math String Sorting

Problem Description

You are provided with two inputs: a string `s` and an integer `k`. The task is to form the lexicographically smallest string possible by repeatedly performing the following operation: choose one of the first `k` letters from the string `s`, and append it to the end of the same string. This operation can be performed as many times as you wish.

Intuition

To solve this problem, we look at two separate cases based on the value of `k`:

- Case where `k` is equal to 1: The only operation allowed in this case is to take the first character of the string and append it to the end. To find the lexicographically smallest string, we can simulate this operation for each character in the string. By doing this, we try all possible rotations of the string to find the smallest one.
- Case where `k` is greater than 1: When `k` is greater than 1, it means we have the ability to move not just the first character but any of the first `k` characters to the end. This effectively allows us to sort the string to achieve the lexicographically smallest string possible. The reasoning is that we can move the smallest character in the first `k` characters to the front, followed by the next smallest, forming an increasing sequence, which is the definition of a sorted string.

Thus, our solution approach is:

1. If `k` is 1, we simulate rotating the string and keep track of the smallest string seen so far, and then return that string.
2. If `k` is greater than 1, we simply return the sorted string.

Solution Approach

The solution follows a straightforward approach that depends on whether `k` equals 1 or is greater than 1. Let's discuss the implementation step by step:

1. The case where `k` equals 1:
   - The implementation begins by setting `ans` to the original string `s`. This `ans` variable will keep track of the current smallest string.
   - Then, we perform a loop iteration for every character in the string except the last one (`len(s) - 1` times).
     - In each iteration, the string `s` is modified by removing its first character and appending it at the end using `s = s[1:] + s[0]`.
     - After modifying the string, we compare it with `ans` using the `min()` function, and if the modified string `s` is smaller, we update `ans`.
   - After testing all possible single-character rotations, we return the smallest string found, `ans`.
2. The case where `k` is greater than 1:
   - This case is handled by one line of code: `return "".join(sorted(s))`.
     - The `sorted()` function is used to sort the characters of the string `s` into lexicographical order.
     - The `"".join()` function is then used to concatenate the sorted characters back into a string.
   - Since we can reorder the first `k` characters to eventually get any permutation of the string, we can always achieve a fully sorted string. So this method will return the lexicographically smallest possible string.

By using Python's built-in functions for sorting and string manipulation, the solution is both efficient and concise, leveraging the language's strengths in handling such operations.

Example Walkthrough

Let's illustrate the solution approach with an example:

Suppose the input string is `s = "bca"` and `k = 2`.

Case where `k > 1`:

Since `k` is greater than 1, we are allowed to choose any of the first `k` letters and move it to the end of the string. This means we can eventually sort the entire string `s`. There are no limitations on how many times this operation can be performed, so the desired output is the lexicographically smallest sorted string. For the given string `"bca"`, the sorted characters would be `["a", "b", "c"]`. Joining these characters, we get the output `"abc"`.

Let's detail the steps for this case:

1. We call the `sorted()` function on the string `s`, which returns a list of the characters in `s` sorted in lexicographical order: `['a', 'b', 'c']`.
2. Then we concatenate these characters back into a string with the `join()` method: `"".join(['a', 'b', 'c'])` results in `"abc"`.
3. The final output `"abc"` is the lexicographically smallest string possible, so we return this string.

Now, let's consider the case when the input string `s = "bca"` and `k = 1`.

Case where `k == 1`:

Here, we can only move the first character to the end of the string. We have to try every possible rotation to find the lexicographically smallest string:

1. The initial value of `ans` is the original string `s`, so `ans = "bca"`.
2. We perform a single-character rotation:
   1. Rotate once: remove the first character and append it to the end, `s = "cab"`.
   2. Compare `s` with `ans`, `s` is lexicographically smaller, so `ans = "cab"`.
3. Repeat the rotation:
   1. Rotate again: `s = "abc"` (from the previous "cab").
   2. Compare `s` with `ans`, `s` is lexicographically smaller, so `ans = "abc"`.
4. No more rotations are needed because we've gone through the string once and `ans` now holds the smallest string after all rotations, `ans = "abc"`.

In both scenarios, the output for the given example is `"abc"`, but the approach to getting the result differs based on the value of `k`. If `k` is greater than 1, we sort the string; if `k` is 1, we perform rotations and keep track of the smallest result.

Solution Implementation

Python

```python
class Solution:
    def orderlyQueue(self, string: str, k: int) -> str:
        # If k is equal to 1, we can only rotate the string.
        if k == 1:
            # Initialize the answer with the original string.
            answer = string
            # Iterate over the string, excluding the last character.
            # since the last rotation would return the string to the original state.
            for _ in range(len(string) - 1):
                # Rotate the string by moving the first character to the end.
                string = string[1:] + string[0]
                # Update the answer if the new string is lexicographically smaller.
                answer = min(answer, string)
            # Return the lexicographically smallest string after all rotations.
            return answer
        # If k is greater than 1, we can sort the string to get the smallest possible string.
        else:
            # Convert the string into a sorted list of characters and join them back into a string.
            return "".join(sorted(string))
```

Java

```java
public class Solution {
    public String orderlyQueue(String s, int k) {
        // If k is greater than 1, we can achieve any permutation by rotation,
        // so we return the characters of the string sorted in alphabetical order.
        if (k > 1) {
            char[] chars = s.toCharArray();
            Arrays.sort(chars);
            return new String(chars);
        }

        // If k is 1, we can only rotate the string.
        // We need to find the lexicographically smallest string possible by rotation.

        // Store the length of the string for convenience.
        int stringLength = s.length();
        // Initialize the minimum (lexicographically smallest) string as the original string.
        String minimumString = s;

        // Iterate through each possible rotation of the string.
        for (int i = 1; i < stringLength; i++) {
            // Create a rotated string by taking the substring from the current index to the end,
            // and concatenating it with the substring from the start to the current index.
            String rotatedString = s.substring(i) + s.substring(0, i);
            // If the rotated string is lexicographically smaller than our current minimum,
            // we update the minimum string.
            if (rotatedString.compareTo(minimumString) < 0) {
                minimumString = rotatedString;
            }
        }

        // Return the lexicographically smallest string after checking all possible rotations.
        return minimumString;
    }
}
```

Please make sure to include the necessary imports at the beginning of your Java file:

```java
import java.util.Arrays;
```

C++

```cpp
class Solution {
public:
    // Function to return the lexicographically smallest string formed by rotating the string 's' if k=1,
    // or returning the lexicographically smallest permutation of 's' if k>1.
    string orderlyQueue(string s, int k) {
        // If only one rotation is allowed at a time (k==1).
        if (k == 1) {
            // 'smallestString' will keep track of the lexicographically smallest string encountered.
            string smallestString = s;
            // Rotate the string and check each permutation.
            for (int i = 0; i < s.size() - 1; ++i) {
                // Rotate string left by one character.
                s = s.substr(1) + s[0];
                // Update 'smallestString' if the new rotation results in a smaller string.
                if (s < smallestString) {
                    smallestString = s;
                }
            }
            // Return the lexicographically smallest string after rotating 's' by each possible number of positions.
            return smallestString;
        } else {
            // If more than one rotation is allowed (k>1), return the smallest permutation.
            // Sorting the string gives us the lexicographically smallest permutation.
            sort(s.begin(), s.end());
            return s;
        }
    }
};
```

TypeScript

```typescript
function orderlyQueue(s: string, k: number): string {
    // If k is greater than 1, sort the characters of the string alphabetically and return it,
    // as we can achieve any permutation of the string by rotating it k times.
    if (k > 1) {
        return [...s].sort().join('');
    }

    // If k is 1, we can only rotate the string. Hence, we must find the lexicographically
    // smallest string by rotating.

    // Store the length of the string for convenience.
    const stringLength = s.length;
    // Initialize the minimum (lexicographically smallest) string as the original string.
    let minimumString = s;

    // Iterate through each possible rotation of the string.
    for (let i = 1; i < stringLength; i++) {
        // Create a rotated string by taking the substring from current position to the end,
        // and concatenating it with the substring from start to the current position.
        const rotatedString = s.slice(i) + s.slice(0, i);
        // If the rotated string is lexicographically smaller than our current minimum,
        // update the minimum string.
        if (rotatedString < minimumString) {
            minimumString = rotatedString;
        }
    }

    // Return the lexicographically smallest string after checking all rotations.
    return minimumString;
}
```

```python
class Solution:
    def orderlyQueue(self, string: str, k: int) -> str:
        # If k is equal to 1, we can only rotate the string.
        if k == 1:
            # Initialize the answer with the original string.
            answer = string
            # Iterate over the string, excluding the last character,
            # since the last rotation would return the string to the original state.
            for _ in range(len(string) - 1):
                # Rotate the string by moving the first character to the end.
                string = string[1:] + string[0]
                # Update the answer if the new string is lexicographically smaller.
                answer = min(answer, string)
            # Return the lexicographically smallest string after all rotations.
            return answer
        # If k is greater than 1, we can sort the string to get the smallest possible string.
        else:
            # Convert the string into a sorted list of characters and join them back into a string.
            return "".join(sorted(string))
```

Time and Space Complexity

Time Complexity

The given code block has two conditions based on the value of `k`:

- When `k` equals `1`, the time complexity is `O(n^2)`. This is because there is a loop that runs `n-1` times (where `n` is the length of string `s`), and in each iteration, a string concatenation operation that takes `O(n)` time is performed. Since string concatenation is done inside the loop, the overall time complexity for this case is `O(n) * O(n) = O(n^2)`.
- When `k` is greater than `1`, the time complexity is determined by the sorting operation, which is `O(n log n)` for typical sorting algorithms, where `n` is the length of the string `s`.

Space Complexity

- When `k` equals `1`, the space complexity is `O(n)` because a new string `ans` of maximum length `n` is created to store the interim and final results.
- When `k` is greater than `1`, the space complexity is also `O(n)` since a sorted copy of the string `s` is being created and returned.