2434. Using a Robot to Print the Lexicographically Smallest String

Medium Stack Greedy Hash Table String Leetcode Link

Problem Description

In this problem, you are given a string `s` and you are to simulate the behavior of a robot that is generating a new string `t` with the ultimate goal of writing down the lexicographically smallest string possible onto paper. The robot achieves this by performing operations until both `s` and `t` are empty. There are two operations possible:

1. Take the first character from the string `s` and append it to the string `t` that the robot holds.
2. Take the last character from the string `t` and write it down on the paper.

The constraints are that you can only write to paper from `t` and you can only add to `t` from `s`. The challenge is to determine the sequence of operations that lead to the smallest possible string on the paper.

Intuition

To arrive at the lexicographically smallest string, we need to write to the paper in a way that prioritizes placing smaller characters before larger ones. However, we are limited by the arrangement of characters in `s` and the operations we can perform.

The solution involves keeping track of the minimum(characters in remaining string `s`) while processing characters from `s` to `t`. We use a stack to replicate the behavior of `t`, appending characters from `s` and popping them when they are ready to be written to the paper. The idea is to always pop the stack and write to the paper whenever the top of the stack has a character that is less than or equal to the current minimum we can still encounter in the remainder of `s`.

Here's the approach broken down:

- We use a counter to keep track of the frequency of characters in `s`.
- We iterate through `s`, decrementing the count for each character as we go.
- We maintain a variable `mi` which is set to the smallest character `a` initially, and it is incremented whenever the count for the current `mi` becomes zero, i.e., we have used all occurrences of the current smallest character.
- We push characters onto a stack, and whenever the top character is less than or equal to `mi` (meaning we can write this character without worrying about missing the opportunity to write a smaller character later), we pop it from the stack and append it to the result string `ans`.
- We repeat the above steps until all characters from `s` have been pushed to the stack and the stack is emptied by writing its characters onto `ans`.

The stack (simulating `t`) essentially acts as a holding area where characters have to wait if a smaller character can appear later from `s`. Only when we're sure that we're at the smallest possible character that is on top of the stack given the future possibilities, we pop it to `ans`.

This strategy guarantees that we are writing the smallest possible characters as early as possible without violating the rules of operation, thus achieving a lexicographically smallest result.

Solution Approach

The solution makes use of a stack data structure, a counter to keep track of character frequencies, and a variable to store the smallest character that can still appear in the string `s`.

Here's a step-by-step explanation of the implementation:

- First, we create a frequency counter for the characters in `s` using Python's `Counter` from the `collections` module. This lets us know how many times each character appears in `s`.

  ```
  cnt = Counter(s)
  ```

- We initialize an empty list `ans` that will serve as the paper onto which the robot writes the characters. We also initialize a stack `stk` to simulate the behavior of holding the characters from `s` before they are written to the paper.

  ```
  ans = []
  stk = []
  ```

- We keep track of the smallest character we have not used up in `s`. Initially, this is `'a'`, the smallest possible character.

  ```
  mi = 'a'
  ```

- We then iterate over each character `c` in our string `s`:

  - For each character `c`, we decrement its count in the frequency counter because we are moving it from `s` to `t` (simulated by the stack).

    ```
    cnt[c] -= 1
    ```

  - We update the `mi` variable if the character count for `mi` has reached zero (meaning all of its occurrences have been used), incrementing it to the next character.

    ```
    while mi < 'z' and cnt[mi] == 0:
        mi = chr(ord(mi) + 1)
    ```

  - The character `c` is then pushed onto the stack as part of the simulation of moving it from `s` to `t`.

    ```
    stk.append(c)
    ```

  - We check the stack's top element and compare it with `mi`. If the top (last) character in the stack is less than or equal to `mi`, it is safe to write this character to paper (append it to `ans`), so we pop it from the stack. This check is done in a `while` loop because multiple characters might satisfy this condition.

    ```
    while stk and stk[-1] <= mi:
        ans.append(stk.pop())
    ```

- Finally, after the loop has processed all characters from `s`, the stack will be empty, and the list `ans` will hold the lexicographically smallest string that the robot could write on the paper.

- The resulting smallest string is obtained by joining the elements of `ans` and returning it.

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

This algorithm's correctness relies on the fact that at each step, we are writing the smallest possible character to the paper. Using the stack allows us to temporarily hold characters that might not be optimal to write immediately and wait until we are certain no smaller character will come before them.

Example Walkthrough

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

Suppose we are given the following string `s`: `"cba"`.

Now, let's walk through the solution:

1. Initialize the frequency counter, the stack (simulating `t`), and the `ans` list (simulating the paper).

   ```
   cnt = Counter("cba")  # cnt = {'c': 1, 'b': 1, 'a': 1}
   ans = []
   stk = []
   mi = 'a'  # Smallest character
   ```

2. We iterate over each character in `s`.

   - For the first character `c`:

     ```
     cnt['c'] -= 1  # cnt = {'c': 0, 'b': 1, 'a': 1}
     stk.append('c')  # stk = ['c']
     ```

     We don't update `mi` because `cnt['a']` is not zero. The stack's top `c` is not less than or equal to `mi` ('a'), so we don't pop from the stack.

   - For the second character `b`:

     ```
     cnt['b'] -= 1  # cnt = {'c': 0, 'b': 0, 'a': 1}
     stk.append('b')  # stk = ['c', 'b']
     ```

     We now update `mi` as 'b' has been used up, but `mi` remains 'a' because `cnt['a']` is not zero. Similarly to the previous step, we do not write to `ans` because `stk[-1]` (`b`) is not less than or equal to `mi` (`a`).

   - For the third character `a`:

     ```
     cnt['a'] -= 1  # cnt = {'c': 0, 'b': 0, 'a': 0}
     stk.append('a')  # stk = ['c', 'b', 'a']
     ```

     `mi` remains 'a' since all characters have been used at least once. Now the condition `stk[-1] <= mi` holds true, so we write to `ans`:

     ```
     # We enter the while loop because stk[-1] ('a') is less than or equal to 'a' (mi).
     ans.append(stk.pop())  # ans = ['a']
     # We continue in the loop, now 'b' is at the top of the stack and mi is 'a'
     # Since 'b' is greater than 'a', we exit the loop.
     ```

   After steps 1-3, we have two letters in `stk`, one letter in `ans`, and `mi` remains 'a'. The loop ends since we traversed the entire `s`. We now have remaining characters in our stack, which need to be appended to `ans` in reverse order (because we pop them from the stack).

   So we pop `b` and then `c` from `stk`, appending each to `ans`:

   ```
   ans.append(stk.pop())  # ans = ['a', 'b']
   ans.append(stk.pop())  # ans = ['a', 'b', 'c']
   ```

3. After processing all the characters, we join the `ans` list to form the result string.

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

This example demonstrates how the given solution approach results in the lexicographically smallest string by strategically moving characters from `s` to `t` (stack `stk`) and then deciding the right time to pop them onto the paper (`ans`). The final output of this process is the string `"abc"`, which is the smallest possible arrangement of the given string `"cba"`.

Python Solution

```python
from collections import Counter

class Solution:
    def robotWithString(self, s: str) -> str:
        # Counter to keep track of how many of each character are left in the string
        char_count = Counter(s)

        # Output list to store the characters for the final result
        result = []

        # Stack to simulate the robot's hand
        stack = []

        # Variable to keep track of the smallest character not exhausted in the string
        min_char = 'a'

        # Iterate over the characters in the string
        for char in s:
            # Decrement the count for the current character
            char_count[char] -= 1

            # Update the smallest character ('min_char') if the current one runs out
            while min_char < 'z' and char_count[min_char] == 0:
                min_char = chr(ord(min_char) + 1)

            # Append the current character to the stack (robot's hand)
            stack.append(char)

            # Append characters to the result from the stack if they are smaller
            # or equal to 'min_char'. This ensures the lexicographically smallest
            # result possible at each step.
            while stack and stack[-1] <= min_char:
                result.append(stack.pop())

        # Join the list of characters to form the final string result
        return ''.join(result)

```

Java Solution

```java
class Solution {
    public String robotWithString(String s) {
        int[] charCount = new int[26]; // Holds the count of each character in the string

        // Increment the count for each character in the string
        for (char ch : s.toCharArray()) {
            charCount[ch - 'a']++;
        }

        StringBuilder answer = new StringBuilder(); // For building the final string
        Deque<Character> stack = new ArrayDeque<>(); // To keep track of characters for processing
        char minChar = 'a'; // To keep track of the smallest character that hasn't been used up

        // Iterate through all characters in the string
        for (char ch : s.toCharArray()) {
            charCount[ch - 'a']--; // Decrease the count as we process each char

            // Update the minChar to the next available smallest character
            while (minChar < 'z' && charCount[minChar - 'a'] == 0) {
                minChar++;
            }

            stack.push(ch); // Add current character to the stack

            // Pop characters from the stack if they are smaller or equal to the current minChar
            while (!stack.isEmpty() && stack.peek() <= minChar) {
                answer.append(stack.pop());
            }
        }

        return answer.toString();
    }
}

```

C++ Solution

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

class Solution {
public:
    // Method that takes a string s and outputs a string based on certain logic
    string robotWithString(string s) {
        int charCount[26] = {0}; // Array to keep track of each character's frequency

        // Count the frequency of each character in the input string
        for (char ch : s) {
            ++charCount[ch - 'a'];
        }

        char minChar = 'a'; // Variable to keep track of the smallest character not used up
        string stack;       // Use a string as a stack to keep track of characters
        string answer;      // The resulting string that we'll build

        // Iterate over each character in the input string
        for (char ch : s) {
            // Decrement the count of the current character,
            // since we're going to process it
            --charCount[ch - 'a'];

            // Find the smallest character that still has remaining occurrences
            while (minChar < 'z' && charCount[minChar - 'a'] == 0) {
                ++minChar;
            }

            // Push the current character onto the stack
            stack.push_back(ch);

            // While the stack is not empty and the top of the stack is less than or equal
            // to the smallest character not used up, append it to the answer and pop it from the stack
            while (!stack.empty() && stack.back() <= minChar) {
                answer.push_back(stack.back());
                stack.pop_back();
            }
        }

        // Return the constructed answer string
        return answer;
    }
};

```

Typescript Solution

```typescript
function robotWithString(s: string): string {
    // Initialize an array to keep count of each character's occurrences
    let characterCount = new Array(128).fill(0);

    // Counting occurrences of each character in the string
    for (let character of s) {
        characterCount[character.charCodeAt(0)] += 1;
    }

    // Initialize with the charCode of 'a', intending to find the smallest lexicographical character
    let minCharCodeIndex = 'a'.charCodeAt(0);

    // The output string will be constructed into this array
    let resultArray = [];

    // Use a stack to keep track of characters processed
    let charStack = [];

    // Process each character in the given string
    for (let character of s) {
        // Decrease the count for this character as it is being processed
        characterCount[character.charCodeAt(0)] -= 1;

        // Find the next character that still has occurrences left
        while (minCharCodeIndex <= 'z'.charCodeAt(0) && characterCount[minCharCodeIndex] == 0) {
            minCharCodeIndex += 1;
        }

        // Push the current character onto the stack
        charStack.push(character);

        // As long as the stack is not empty and the last character on the stack is less
        // or equal to the current minimum, add it to the result array
        while (charStack.length > 0 && charStack[charStack.length - 1].charCodeAt(0) <= minCharCodeIndex) {
            resultArray.push(charStack.pop());
        }
    }

    // Join the result array into a string and return
    return resultArray.join('');
}

```

Time and Space Complexity

Time Complexity

The given Python function `robotWithString` has several key operations contributing to its time complexity:

1. The construction of the `Counter` object - This happens once and takes `O(n)` time, where `n` is the length of string `s`, since the `Counter` has to iterate through all characters in the string.

2. The first `for` loop - This loop runs for each character in the string `s`, hence it iterates `n` times.

3. The inner `while` loop to find the current minimum character `mi` - Since there are 26 English letters, in the worst case, it might check every character until 'z', taking `O(1)` time regardless of the size of `s` as it's not directly dependent on `n`.

4. The second inner `while` loop where elements are popped from `stk` and added to `ans` - Each character can be pushed and popped at most once, leading to `O(n)` operations overall.

Given these, we see that the main complexity comes from the operations linked directly to the length of `s`. No nested loops are dependent on the size of `s`, thus, the overall time complexity is `O(n)`.

Space Complexity

Space complexity considerations involve the additional memory used by the algorithm:

1. The `Counter` object `cnt` - Its space complexity is `O(n)` in the worst case when all characters in the `s` are unique.

2. The list `ans` - Worst case, this list will contain all characters from `s`, thus `O(n)` space.

3. The list `stk` - In the worst case, this can also grow to include all characters from `s`, hence `O(n)` space.

Adding all this up - we have the `Counter`, `ans`, and `stk` all contributing a linear amount of space complexity with respect to the length of string `s`. Therefore, the total space complexity is `O(n)`, where `n` is the length of the string `s`.