2810. Faulty Keyboard

Easy String Simulation

Leetcode Link

Problem Description

In this problem, we're dealing with a string that is being typed on a laptop with a faulty keyboard. Every time the character 'i' is typed, instead of simply adding 'i' to the string, the entire string typed so far is reversed. Other characters are added to the string normally. We're given a string `s`, which represents the sequence of characters being typed, and we have to determine what the string looks like after typing all characters on the faulty keyboard.

The string `s` is 0-indexed, meaning we start counting positions of the characters in the string from 0. For example, in string `abc`, 'a' is at index 0, 'b' is at index 1, and 'c' is at index 2. The goal is to process the string character by character as per the keyboard's faulty behavior and to return the resulting string after the complete sequence has been typed out.

Intuition

When considering how to simulate the typing on this faulty keyboard, one approach is to iterate through the string and handle each character according to the described rules. Since characters can either be appended or cause a reversal of the current string, we can use a data structure that efficiently supports these operations. An array or list can serve this purpose. We can simulate typing by iterating over each character and manipulating the array accordingly.

Here's the intuitive breakdown of the process:

1. Initialize an empty array `t`, which will keep track of the characters typed so far.
2. Iterate over each character `c` in the given string `s`.
3. If `c` is 'i', reverse the array `t`. In Python, this is conveniently done using the slicing operation `t[::-1]`, which returns a new array with the elements of `t` reversed.
4. If `c` is not 'i', simply append `c` to the array `t`.
5. After processing all characters, convert the array `t` back to a string using the `join` method and return it.

The algorithm is straightforward and has a linear time complexity with respect to the length of the string because each character is processed once, and each operation is done in constant time (reversal with slicing is done in linear time, but it does not increase the overall complexity of the algorithm).

Solution Approach

The solution uses a simple algorithm that makes use of Python's list data structure for its ability to dynamically add elements and reverse the contents efficiently.

Here's the step-by-step implementation:

1. An empty list `t` is created: `t = []`. This list will be used to represent the current state of the string, simulating the typed characters.
2. A `for` loop iterates over each character `c` in the input string `s`. With each iteration, we perform one of two actions depending on whether `c` is 'i' or not:
   - If the character is not 'i', we append it to the end of the list `t` using `t.append(c)`. This simulates typing a character normally.
   - If the character is 'i', we reverse the list `t` in place. The slicing syntax `t[::-1]` creates a reversed copy of the list, and we assign this reversed copy back to `t`. The syntax `[::-1]` is a commonly used Python idiom to reverse a list or a string.
3. After the loop finishes processing all the characters in `s`, we convert the list `t` into a string using `"".join(t)`. The `join` method takes all the elements of the list and concatenates them into a single string, with each element being joined without any additional characters (since an empty string `""` is used as the separator).
4. Finally, the `finalString` method returns this resulting string which represents the final state of the string as it would appear on the faulty laptop screen.

The overall complexity of the algorithm is O(n), where n is the length of the input string `s`. Although reversing a list is an O(n) operation by itself, the algorithm performs a constant number of operations per character in the input string, and the complexity is linear with respect to the length of `s`.

This solution approach doesn't use any complex patterns or data structures. It capitalizes on the flexibility of Python lists and utilizes Python's slicing feature to perform the reversal operation succinctly.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose our input string `s` is "abici".

We follow the steps outlined in the Solution Approach:

1. **Initialize List**: We start by creating an empty list `t`.

```
t = []
```

2. **Iterate over `s`**: Now, we process each character in the input string "abici".
   - First character: 'a'
     - 'a' is not 'i', so we append 'a' to `t`: `t` becomes ['a'].
   - Second character: 'b'
     - 'b' is not 'i', so we append 'b' to `t`: `t` becomes ['a', 'b'].
   - Third character: 'i'
     - 'i' triggers the reversal, so we reverse `t`: before reversing `t` is ['a', 'b']; after reversing, `t` becomes ['b', 'a'].
   - Fourth character: 'c'
     - 'c' is not 'i', so we append 'c' to `t`: `t` becomes ['b', 'a', 'c'].
   - Fifth character: 'i'
     - 'i' triggers the reversal again, so we reverse `t`: before reversing `t` is ['b', 'a', 'c']; after reversing, `t` becomes ['c', 'a', 'b'].
3. **Join List into String**: After processing all characters, we concatenate the elements in `t` to form the final string.

```
final_string = "".join(t)  # This produces 'cab'
```

4. **Return Result**: The resulting string is 'cab', which is the state of the string as it appears after typing "abici" on the faulty keyboard.

And that's our final output. The input "abici" results in "cab" after all the operations are performed.

Python Solution

```python
class Solution:
    def finalString(self, input_string: str) -> str:
        # Initialize an empty list to hold the characters.
        transformed_list = []

        # Iterate over each character in the input string.
        for character in input_string:
            # If the character is 'i', reverse the transformed list.
            if character == 'i':
                transformed_list = transformed_list[::-1]
            # Otherwise, append the current character to the transformed list.
            else:
                transformed_list.append(character)

        # Join the characters in the transformed list to form the final string.
        return "".join(transformed_list)
```

Java Solution

```java
class Solution {

    /**
     * Processes the input string to form a final string.
     * Each occurrence of the character 'i' in the input string
     * will cause the current result to be reversed.
     *
     * @param s The input string to be processed.
     * @return The final processed string.
     */
    public String finalString(String s) {
        // StringBuilder is used for efficient string manipulation
        StringBuilder resultBuilder = new StringBuilder();

        // Iterate over each character in the input string
        for (char currentChar : s.toCharArray()) {
            // If the current character is 'i', reverse the current result
            if (currentChar == 'i') {
                resultBuilder.reverse();
            } else {
                // Otherwise, append the current character to the result
                resultBuilder.append(currentChar);
            }
        }

        // Convert the StringBuilder to String and return the final result
        return resultBuilder.toString();
    }
}

```

C++ Solution

```cpp
#include <string>
#include <algorithm> // include algorithm for std::reverse

class Solution {
public:
    // Function to process the string according to the given rules
    std::string finalString(std::string s) {
        std::string result; // Create an empty string to store the result

        // Iterate through each character in the input string
        for (char ch : s) {
            // Check if the current character is 'i'
            if (ch == 'i') {
                // Reverse the string stored in 'result' so far
                std::reverse(result.begin(), result.end());
            } else {
                // Append the current character to the 'result' string
                result.push_back(ch);
            }
        }

        // Return the final processed string
        return result;
    }
};
```

Typescript Solution

```typescript
// Function to process a given string according to specific rules
// Whenever the letter 'i' is encountered, the accumulated characters are reversed
// Other characters are simply added to the accumulator
function finalString(s: string): string {
    // Initialize an empty array to store characters
    const accumulatedCharacters: string[] = [];

    // Iterate over each character of the input string
    for (const char of s) {
        // Check if the current character is 'i'
        if (char === 'i') {
            // Reverse the accumulated characters if 'i' is encountered
            accumulatedCharacters.reverse();
        } else {
            // Add the current character to the accumulator if it is not 'i'
            accumulatedCharacters.push(char);
        }
    }

    // Combine the accumulated characters into a single string and return
    return accumulatedCharacters.join('');
}

```

Time and Space Complexity

The given Python code takes an input string `s` and reverses the list `t` whenever an 'i' is encountered, otherwise appends the character to `t`. The finally joined `t` is returned as the final string.

Time Complexity

Let's denote `n` as the length of the input string `s`.

- For each character in the string, the code checks if it is an 'i' or not, which is an O(1) operation.
- If the character is not an 'i', appending to list `t` is generally an O(1) operation.
- Reversing a list using `t[::-1]` creates a new copy of the list `t` in reverse order. This operation is O(n) where `n` is the current length of the list `t` at the time the reverse operation is performed.

Since the reversing operation could potentially occur for each 'i' in the string, in the worst case where the input string is composed of 'i's, the time complexity would be O(k*n) with 'k' being the number of 'i's and 'n' being the length of `t` at that point. In other words, the time complexity is quadratic with respect to the number of 'i's.

To be more precise, let `m` be the frequency of `i` in `s`, the complexity is the sum of an arithmetic sequence:

```
O(1) + O(2) + ... + O(n-m) = O((n-m)(n-m+1)/2) = O((n-m)^2/2)
```

Similarly, if 'i's are evenly distributed, the time complexity would still be high, though not strictly quadratic.

Thus, the worst-case time complexity is O(n^2) if we consider that `i`s are uniformly distributed or at the start of the string, but could potentially be less depending on the distribution of 'i's.

Space Complexity

- The list `t` is the additional data structure which, in the worst case, will be as long as the input string `s`. Thus, the space required by `t` is O(n).
- The list reversal operation `t[::-1]` does not happen in place, it creates a new list each time which requires up to O(n) space. However, this space is temporary and each reversed list is only present until the next reversal or append operation.

Therefore, considering the input string's length, the overall space complexity is O(n).