831. Masking Personal Information

Medium String Leetcode Link

Problem Description

This problem involves designing an algorithm to mask personal information. Personal information can be an email address or a phone number, and each has specific rules for masking.

Email Address Masking Rules:

- Transform all uppercase letters in the name and domain parts of the email address to lowercase.
- Mask all but the first and last letter of the name part with five asterisks (`*****`).

Phone Number Masking Rules:

- Remove all separation characters (which may include `+`, `-`, `(`, `)`, and spaces) to only retain digits.
- Format the number based on the leftover digit count after the local number (last 10 digits). The formatting is as follows:
  - If there is no country code, the format is `***-***-XXXX`, where `XXXX` is the last four digits of the local number.
  - If there is a country code, it should be represented by the same number of asterisks as digits, followed by the formatted local number `***-***-XXXX`.

The algorithm must identify whether the provided string is an email or a phone number and mask it accordingly.

Intuition

The intuition behind solving this problem starts with identifying if the given `s` is an email address or a phone number. This can usually be identified by the presence of alphabetic characters and the special character `@`. If `s[0]` is an alphabetic character, it is an email, otherwise it's a phone number.

For an **email address**:

1. Convert the entire string to lowercase to satisfy the case-insensitivity requirement.
2. Replace the middle part of the name with five asterisks, but maintain the first character, the character before the `@`, and the complete domain after the `@`.

For a **phone number**:

1. Remove all separation characters by filtering out non-digit characters, leaving only the digits.
2. The number of characters remaining after the last 10 gives us the number of digits in the country code. Depending on this count, mask the country code with asterisks.
3. The last 4 digits of the local number remain unmasked, while the others are replaced by asterisks, keeping consistent with the `***-***-` prefix.

The provided solution uses Python comprehensions to filter and the string methods to replace and format according to the rules. The code is compact and only uses conditional logic to distinguish the masking process between an email address and a phone number.

Solution Approach

The implementation of the masking algorithm can be broken down into several steps and patterns used throughout the code:

1. **Determining the Type of Personal Information**: The algorithm first checks if the given string `s` represents an email address or a phone number. This is done by checking the first character of the string:

   ```python
   if s[0].isalpha():
   ```

   If the first character is an alphabetical letter, it is an email address; otherwise, it is a phone number.

2. **Email Masking**:

   - All characters of the email address are converted to lowercase using `lower()` to ensure case insensitivity.
   - The first and the last character of the name part, along with the domain, are preserved.
   - The middle part of the name is replaced with five asterisks (`*****`).

   ```python
   return s[0] + '*****' + s[s.find('@') - 1 :]
   ```

3. **Phone Number Masking**:

   - All non-digit characters are filtered out, leaving only digits in the string:

     ```python
     s = ''.join(c for c in s if c.isdigit())
     ```

   - The algorithm calculates the length of the country code by subtracting 10 from the total length of the filtered digits. This is because the local number always consists of the last 10 digits.
   - A switch-like statement, using if-elif, is avoided. Instead, a more elegant formatting via an f-string and repetition of asterisk characters is used to handle different lengths of country codes:

     ```python
     cnt = len(s) - 10
     suf = '***-***-' + s[-4:]
     return suf if cnt == 0 else f'+{"*" * cnt}-{suf}'
     ```

In summary, the solution makes use of string methods (like `lower()` and `find()`), list comprehensions (to filter characters), and string concatenation to assemble the masked personal information. It uses the compactness of Python's string formatting to keep the code concise and readable. There is no explicit usage of complex data structures as the operations are basic string manipulations. This approach is efficient because it goes through the string at most two times: once to either lower the case of the characters (for emails) or filter digits (for phone numbers), and once to construct the masked string. This is essentially O(n) complexity, where n is the length of the input string.

Example Walkthrough

Let's walk through a small example illustrating the solution approach for both an email and a phone number.

Email Example

Given the email `ExampleEmail@LeetCode.com`, let's apply the solution approach:

1. Since the first character 'E' is an alphabetical letter, we identify the string as an email address.
2. Convert the email to lowercase: `exampleemail@leetcode.com`.
3. Preserve the first and the last character of the name part and the entire domain: 'e' and 'l' are the two characters we preserve from `exampleemail`.
4. Replace the middle part of the name with five asterisks: `e*****l@leetcode.com`.

After following these steps, we get the masked email address: `e*****l@leetcode.com`.

Phone Number Example

Given the phone number `+1 (234) 567-8901`, let's apply the solution approach:

1. The first character '+' indicates it's a phone number.
2. Remove all non-digit characters, leaving only the digits: `12345678901`.
3. Identify the country code, which in this case is '1', by counting the digits beyond the last 10, which gives us 1 digit here.
4. Replace the country code with an asterisk and apply the standard mask for the remaining number, preserving the last four digits: `* ***-***-8901`.

After following these steps, we get the masked phone number: `* ***-***-8901`.

These two examples demonstrate how the algorithm distinguishes between email addresses and phone numbers and how it applies the respective masking rules to conceal personal information while following the specified format.

Python Solution

```python
class Solution:
    def maskPII(self, string: str) -> str:
        # Check if the string contains an email address by looking for an alphabetic character at the start
        if string[0].isalpha():
            string = string.lower() # Convert the email to lowercase
            # Build the masked email with the first character, five asterisks, and the domain part
            masked_email = string[0] + '*****' + string[string.find('@') - 1:]
            return masked_email
        # If it's not an email, assume it's a phone number
        else:
            # Remove all non-digit characters from the string
            digits_only = ''.join(c for c in string if c.isdigit())
            # Count the number of digits beyond the local 10-digit format
            extra_digits_count = len(digits_only) - 10
            # Create a suffix for the masked phone number
            phone_suffix = '***-***-' + digits_only[-4:]
            # If there are no extra digits, return the standard format
            if extra_digits_count == 0:
                return phone_suffix
            # If there are extra digits (international format), add the country code mask
            else:
                return f'+{"*" * extra_digits_count}-{phone_suffix}'

# Example usage:
# solution = Solution()
# print(solution.maskPII("John.Doe@example.com"))  # Output: "j*****e@example.com"
# print(solution.maskPII("1(234)567-890"))         # Output: "***-***-7890"
# print(solution.maskPII("+1(234)567-890"))        # Output: "+*-***-***-7890"

```

Java Solution

```java
class Solution {

    public String maskPII(String input) {
        // Check if the first character is a letter to determine if it's an email
        if (Character.isLetter(input.charAt(0))) {
            // Convert the entire input string to lower case for uniformity
            input = input.toLowerCase();
            // Find the index of the '@' symbol in the email
            int atIndex = input.indexOf('@');
            // Create a masked email with the first character, five stars, and the domain part
            return input.charAt(0) + "*****" + input.substring(atIndex - 1);
        }

        // StringBuilder to hold only the digits from the input (phone number)
        StringBuilder digitBuilder = new StringBuilder();
        // Loop through the characters in the input
        for (char c : input.toCharArray()) {
            // Append only if the character is a digit
            if (Character.isDigit(c)) {
                digitBuilder.append(c);
            }
        }

        // Convert the StringBuilder to a string containing only digits
        String digits = digitBuilder.toString();
        // Calculate the number of digits that are in the international code
        int internationalCodeLength = digits.length() - 10;
        // Create a masked string for the last 7 digits of the phone number
        String maskedSuffix = "***-***-" + digits.substring(digits.length() - 4);

        // Depending on whether there is an international code, mask the phone number appropriately
        if (internationalCodeLength == 0) {
            // If there's no international code, return just the masked U.S. number
            return maskedSuffix;
        } else {
            // If there's an international code, mask it and append the U.S. number
            String starsForInternational = "+";
            for(int i = 0; i < internationalCodeLength; i++) {
                starsForInternational += "*";
            }
            return starsForInternational + "-" + maskedSuffix;
        }
    }
}

```

C++ Solution

```cpp
#include <cctype> // include ctype for using tolower, isdigit functions
#include <string> // include string for using string class
using namespace std;

class Solution {
public:
    /**
     * @param s: The original string, either an email or a phone number.
     * @return: The masked string with PIIs (Personally Identifiable Information) hidden.
     */
    string maskPII(string s) {
        // Find the position of '@', which indicates an email.
        size_t atPosition = s.find('@');
        // If "@" is found, mask it as an email.
        if (atPosition != string::npos) {
            string maskedEmail;
            // Convert first character to lowercase and append.
            maskedEmail += tolower(s[0]);
            // Append 5 asterisks to hide part of the local name.
            maskedEmail += "*****";
            // Convert from before '@' to lowercase and append the domain.
            for (size_t j = atPosition - 1; j < s.size(); ++j) {
                maskedEmail += tolower(s[j]);
            }
            return maskedEmail;
        }

        // If "@" is not found, it's a phone number.
        // Extract all digits from the string.
        string digits;
        for (char c : s) {
            if (isdigit(c)) {
                digits += c;
            }
        }

        // Determine how many digits are in the country code.
        int countryCodeLength = digits.size() - 10;
        // Create the suffix for the standard 10-digit mask.
        string suffix = "***-***-" + digits.substr(digits.size() - 4);

        // If there's no country code, return just the suffix.
        if (countryCodeLength == 0) {
            return suffix;
        } else {
            // If there's a country code, prepend it with masks.
            string countryCodeMask = "+" + string(countryCodeLength, '*') + "-";
            return countryCodeMask + suffix; // Full masked string with country code.
        }
    }
};

```

Typescript Solution

```typescript
function maskPII(s: string): string {
    // Find the index of '@' to determine if it's an email.
    const atIndex: number = s.indexOf('@');
    if (atIndex !== -1) {
        // If it's an email, mask the characters between the first character and the '@' symbol.
        let maskedEmail: string = s[0].toLowerCase() + '*****';
        for (let j: number = atIndex - 1; j < s.length; j++) {
            maskedEmail += s.charAt(j).toLowerCase();
        }
        return maskedEmail;
    }

    // If it's not an email, build a string containing only the digits from the input.
    let numericString: string = '';
    for (const char of s) {
        if (/\d/.test(char)) { // Check if the character is a digit.
            numericString += char;
        }
    }

    // Compute the count of digits to be replaced by '*' in the case there are country codes.
    const countryCodeStarsCount: number = numericString.length - 10;

    // The suffix for the phone number hiding the first 6 digits, revealing only the last 4.
    const maskedPhoneSuffix: string = `***-***-${numericString.substring(numericString.length - 4)}`;

    // Return the masked phone number with country code as '*' if there is a country code.
    return countryCodeStarsCount === 0 ? maskedPhoneSuffix : `+${'*'.repeat(countryCodeStarsCount)}-${maskedPhoneSuffix}`;
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the `maskPII` function is determined by several operations that the function performs:

1. Checking whether the first character is alphabetical: `O(1)`, since it's a constant-time check on a single character.

2. Converting the string to lower case if it's an email: `O(n)`, where `n` is the length of the string `s`.

3. Finding the position of '@' in the string: `O(n)` in the worst case, since it requires scanning the string in case of an email address.

4. Stripping non-digit characters and joining digits into a new string: `O(n)`, because each character of the original string `s` is checked once.

5. Constructing the masked string: The actual construction is `O(1)` because we are appending fixed strings and characters to a masked result.

Combining these, the time complexity is dictated by the string operations, which are linear in the length of the input string `s`. Therefore, the time complexity is `O(n)`.

Space Complexity:

The space complexity is determined by the additional space used by the function:

1. A new lower case email id string if `s[0].isalpha()`: `O(n)` space for the new string.

2. The list comprehension that filters digits: `O(n)` space for the new list of characters before it is joined into a string.

3. The final masked string has a constant size related to the format of the phone number or email (`O(1)` space), not dependent on the input size.

Thus, the dominating factor is the creation of new strings, which takes `O(n)` space. Hence, the space complexity is `O(n)`.