2299. Strong Password Checker II

Easy String

Leetcode Link

Problem Description

The problem presents a scenario where we need to validate a string, `password`, to determine if it qualifies as a **strong password**. For a password to be considered strong, it must meet all these criteria:

1. The length of the password must be at least `8` characters.
2. It must include at least one lowercase letter.
3. It must include at least one uppercase letter.
4. It must have at least one digit.
5. It must contain at least one special character, which must be from the set `!@#$%^&*()-+`.
6. It must not have more than one identical character in a row, meaning no two adjacent characters can be the same.

The goal is to write a function that returns `true` if the `password` meets all the above-described conditions, otherwise returns `false`.

Intuition

The intuition behind the solution is to go through the password character by character to check if it meets all the necessary criteria for being strong. We can do this by iterating through the string and using flags to mark if we've detected each type of required character. Here's the step-by-step process:

1. **Length Check**: First, we check if the password has at least 8 characters. If it's shorter, we immediately return `false`.
2. **Adjacency Check**: As we iterate, we check if the current character is the same as the previous one - if it is, we return `false` because this violates the non-adjacent character condition.
3. **Character Type Checks**: For every character, we need to check if it is a lowercase letter, an uppercase letter, a digit, or a special character. We can do this using the `.islower()`, `.isupper()`, `.isdigit()` methods, and by verifying if the character is in the specified special characters string.
4. **Aggregation with Bitmasking**: Instead of keeping four separate flags, we can use a bitmask (`mask`) to aggregate all the flags into a single integer. Bitwise OR operations `|=` are used to set the corresponding bits when we encounter lowercase letters, uppercase letters, digits, and special characters. Each character type corresponds to a different bit in the mask, so for example:
   - If we encounter a lowercase letter, we set the first bit (`mask |= 1`).
   - For an uppercase letter, the second bit (`mask |= 2`), and so on.
5. **Final Verification**: After going through every character of the password, we check if the mask equals `15` (binary `1111`). This means that all four bits are set, so every type of required character is included in the password at least once.

The code is efficient and compact. By using bit operations, it avoids the use of multiple boolean variables and reduces the number of conditions checked.

Solution Approach

The implementation of the solution involves a simple yet effective approach by scanning through each character in the password and using bitwise operations to track whether the password criteria have been met. Here's a detailed walk-through:

1. **Initial Length Check**: Immediately check if the password is less than 8 characters. If so, the function returns `false`.

   ```
   if len(password) < 8: return False
   ```

2. **Setup**: Initialize a variable `mask` to `0`. This will serve as a 4-bit mask where each bit represents the presence of a different character type (lowercase, uppercase, digit, special character).

3. **Iterate Through Password Characters**: By using a `for` loop with enumeration, we iterate over the password's characters, keeping track of each character and its index.

   ```
   for i, c in enumerate(password):
   ```

4. **Adjacency Check**: Inside the loop, we first check if the current character is the same as the previous one. If that's the case, we immediately return `false` since this violates the non-adjacent identical character condition.

   ```
   if i and c == password[i - 1]: return False
   ```

5. **Character Type Detection**: Still in the loop, we check the type of the current character:
   - If it's a lowercase letter (`c.islower()`), set the first bit of the mask (`mask |= 1`).
   - If it's an uppercase letter (`c.isupper()`), set the second bit of the mask (`mask |= 2`).
   - If it's a digit (`c.isdigit()`), set the third bit of the mask (`mask |= 4`).
   - If it's a special character (checked by seeing if it is not any of the above types), set the fourth bit of the mask (`mask |= 8`).
6. **Final Verification**: After the loop, we check if all the bits are set in the mask by comparing it to `15` (binary `1111`). This means all required character types are present in the password. The function returns `true` if `mask == 15`; otherwise, it returns `false`.
7. **Data Structures, Algorithms & Patterns**:
   - **Data Structure**: A single *integer* variable is used for tracking the presence of character types through a concept called bitmasking.
   - **Algorithms**: A single pass through the string is the main algorithmic component. All checks are done in this single pass, making the time complexity `O(n)` with `n` as the password length.
   - **Patterns**: The solution uses bitwise operations to aggregate checks into a single value, reducing the need for multiple variables.

The simplicity and efficiency of the bitwise operations are key in making the code concise and performant. The choice to use a bitmask over multiple boolean variables exemplifies a common pattern in problems where aggregating flags into a single integer helps optimize space and improve code readability.

Example Walkthrough

Let's consider the password string `password` as `Aa1!Aa1!`. To determine if this is a strong password according to the given criteria, our function will proceed as follows:

1. **Initial Length Check**: Our password `Aa1!Aa1!` is 8 characters long, so it passes the length requirement.
2. **Setup**: We initialize the `mask` to `0`. This mask will help us track whether we have encountered at least one lowercase letter, uppercase letter, digit, and special character.
3. **Iterate Through Password Characters**: We start looping through each character in the password.
4. **Adjacency Check**:
   - For the first character 'A', there is no previous character, so we move on to type checking.
   - The second character 'a' is different from the first, so no adjacency violation occurs.
   - This process continues, and since no adjacent characters are identical, no adjacency checks fail.
5. **Character Type Detection**: As we go through each character:
   - For 'A': It's uppercase, so we set the second bit of the mask (`mask |= 2`).
   - For 'a': It's lowercase, so we set the first bit of the mask (`mask |= 1`).
   - For '1': It's a digit, so we set the third bit of the mask (`mask |= 4`).
   - For '!': It's a special character, so we set the fourth bit of the mask (`mask |= 8`).
   - As we continue through each character, no additional bits are set since each type has already been encountered.
6. **Final Verification**: At the end, our `mask` is `1111` in binary, or `15` in decimal, which means all types of required characters were present at least once.
7. **Data Structures, Algorithms & Patterns**: The use of bitmasking to track character types in a single integer is an efficient data structure choice, and iterating through the password is the primary algorithm. No patterns are additional, and the bitwise operations signify a common pattern to optimize space and improve code readability.

The password `Aa1!Aa1!` is therefore confirmed to be a strong password by our implementation.

Python Solution

```python
class Solution:
    def strongPasswordCheckerII(self, password: str) -> bool:
        # Minimum password length required
        min_password_length = 8

        # Check if the password length is at least 8 characters
        if len(password) < min_password_length:
            return False

        # Initializing a variable to use as a bitmask to track the requirement fulfillment
        requirement_mask = 0

        # Loop through each character in the password
        for i, char in enumerate(password):
            # Check if the current character is the same as the previous character
            if i > 0 and char == password[i - 1]:
                return False  # Consecutive characters are not allowed

            # Check if the character is a lowercase letter
            if char.islower():
                requirement_mask |= 1  # Set the bit for lowercase letter
            # Check if the character is an uppercase letter
            elif char.isupper():
                requirement_mask |= 2  # Set the bit for uppercase letter
            # Check if the character is a digit
            elif char.isdigit():
                requirement_mask |= 4  # Set the bit for digit
            # Check if the character is a special character
            else:
                requirement_mask |= 8  # Set the bit for special character

        # Check if all 4 requirements are met, which is when all 4 bits are set (i.e., requirement_mask == 1111 binary, which is 15 i
        return requirement_mask == 15
```

Java Solution

```java
class Solution {
    // Method to check if a given password is strong according to specified rules
    public boolean strongPasswordCheckerII(String password) {
        // Requirement: password should be at least 8 characters long
        if (password.length() < 8) {
            return false;
        }

        // A mask to keep track of the types of characters found
        int characterTypesMask = 0;

        // Iterate through each character in the password
        for (int i = 0; i < password.length(); ++i) {
            // Current character being checked
            char currentChar = password.charAt(i);

            // Requirement: password should not contain consecutive identical characters
            if (i > 0 && currentChar == password.charAt(i - 1)) {
                return false;
            }

            // Identifying the type of the current character and updating the mask accordingly
            // If it is lowercase, set the first bit using OR operation with 1 (001)
            if (Character.isLowerCase(currentChar)) {
                characterTypesMask |= 1; // 001
            }
            // If it is uppercase, set the second bit using OR operation with 2 (010)
            else if (Character.isUpperCase(currentChar)) {
                characterTypesMask |= 2; // 010
            }
            // If it is a digit, set the third bit using OR operation with 4 (100)
            else if (Character.isDigit(currentChar)) {
                characterTypesMask |= 4; // 100
            }
            // If it is a special character, set the fourth bit using OR operation with 8 (1000)
            else {
                characterTypesMask |= 8; // 1000
            }
        }

        // Requirement: password must contain all types of characters (lowercase, uppercase, digit, special character)
        // This is true if, after going through the entire string, the mask equals 15 (1111)
        // which corresponds to having all four types of characters
        return characterTypesMask == 15;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if a given password meets strong password criteria.
    bool strongPasswordCheckerII(string password) {
        // The password must be at least 8 characters long.
        if (password.size() < 8) {
            return false;
        }

        // 'requirementsMet' will track the types of characters present in the password.
        // Each bit in 'requirementsMet' corresponds to a different requirement:
        // Bit 0 (1) represents the presence of a lowercase letter,
        // Bit 1 (2) represents the presence of an uppercase letter,
        // Bit 2 (4) represents the presence of a digit,
        // Bit 3 (8) represents the presence of a special character.
        int requirementsMet = 0;

        // Iterate over the password characters.
        for (int i = 0; i < password.size(); ++i) {
            char currentChar = password[i];

            // Check if the current character is the same as the previous one.
            if (i > 0 && currentChar == password[i - 1]) {
                return false; // Return false if two adjacent characters are the same.
            }

            // Check for different types of characters and update 'requirementsMet'.
            if (currentChar >= 'a' && currentChar <= 'z') {
                requirementsMet |= 1; // Presence of a lowercase letter.
            } else if (currentChar >= 'A' && currentChar <= 'Z') {
                requirementsMet |= 2; // Presence of an uppercase letter.
            } else if (currentChar >= '0' && currentChar <= '9') {
                requirementsMet |= 4; // Presence of a digit.
            } else {
                requirementsMet |= 8; // Presence of a special character.
            }
        }

        // Check if all four types of characters are present (binary 1111 is decimal 15).
        return requirementsMet == 15;
    }
};
```

Typescript Solution

```typescript
function strongPasswordCheckerII(password: string): boolean {
    // Length check - password must be at least 8 characters
    if (password.length < 8) {
        return false;
    }

    // Initialize a bitmask to keep track of character types encountered
    // bit 0 for lowercase, bit 1 for uppercase, bit 2 for digits, bit 3 for special characters
    let charTypesMask = 0;

    // Iterate over the characters of the password to validate the rules
    for (let i = 0; i < password.length; ++i) {
        const currentChar = password[i];
        // Check for consecutive identical characters
        if (i > 0 && currentChar === password[i - 1]) {
            return false;
        }

        // Check the type of character and update the bitmask accordingly
        if (currentChar >= 'a' && currentChar <= 'z') {
            charTypesMask |= 1; // Set bit 0 for lowercase
        } else if (currentChar >= 'A' && currentChar <= 'Z') {
            charTypesMask |= 2; // Set bit 1 for uppercase
        } else if (currentChar >= '0' && currentChar <= '9') {
            charTypesMask |= 4; // Set bit 2 for digit
        } else {
            charTypesMask |= 8; // Set bit 3 for special character
        }
    }

    // Check if all four character types are present by confirming all bits are set in the mask
    return charTypesMask === 15; // (binary 1111)
}
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the `password` string. This is because the function consists of a single for loop that iterates through each character of the password string exactly once, performing a constant number of operations per character.

The space complexity of the code is `O(1)`. The extra space used by the function is constant and does not depend on the size of the input password string. The variables used (`mask` and `c`) require a fixed amount of space and their size does not scale with the input.