Strong Password Checker Documentation

1. Problem Statement

A password is considered strong if it meets the following conditions:

- It has at least 6 characters and at most 20 characters.
- It contains at least one lowercase letter, at least one uppercase letter, and at least one digit.
- It does not contain three repeating characters in a row (e.g., "Baaabb0" is weak, but "Baaba0" is strong).

Given a string password, return the minimum number of steps required to make the password strong. If the password is already strong, return 0.

In one step, you can:

- Insert one character into the password,
- Delete one character from the password, or
- Replace one character of the password with another character.

Example 1:

- **Input:** password = "a"
- **Output:** 5
- **Explanation:** The password needs 5 more characters to meet the minimum length requirement and must include at least one uppercase letter and one digit.

Example 2:

- **Input:** password = "aA1"
- **Output:** 3
- **Explanation:** The password needs 3 more characters to meet the minimum length requirement.

Example 3:

- **Input:** password = "1337C0d3"
- **Output:** 0
- **Explanation:** The password is already strong.

Constraints:

- 1 <= password.length <= 50
- password consists of letters, digits, dot '.', or exclamation mark '!'.

2. Intuition

The problem requires us to ensure that the password meets specific criteria for strength. The main challenges are:

a. Ensuring the password length is between 6 and 20 characters.

b. Ensuring the password contains at least one lowercase letter, one uppercase letter, and one digit.

c. Ensuring there are no sequences of three or more repeating characters.

To solve this, we need to:

a. Identify missing character types (lowercase, uppercase, digit).

b. Identify sequences of repeating characters that need to be broken.

c. Determine the minimum number of steps (insertions, deletions, or replacements) required to fix the password.

3. Key Observations

a. Length Constraints:

- If the password is too short (<6), we need to add characters.
- If the password is too long (>20), we need to delete characters.
- If the password is within the valid range (6-20), we only need to fix missing types and repeating sequences.

b. Missing Character Types:

- We need to count how many types (lowercase, uppercase, digit) are missing.

c. Repeating Characters:

- Sequences of three or more repeating characters need to be broken by replacements or deletions.

d. Optimization:

- When the password is too long, deletions can help reduce the number of replacements needed for repeating sequences.

4. Approach

a. Check for Missing Character Types:

- Iterate through the password to check if it contains at least one lowercase letter, one uppercase letter, and one digit.
- Calculate the number of missing types.

b. Identify Repeating Sequences:

- Traverse the password to find sequences of three or more repeating characters.
- Count the number of replacements needed to break these sequences.

c. Handle Different Length Cases:

- ***Case 1:*** *Password is too short (<6):*
  - The number of steps required is the maximum of (6 - n) and the number of missing types.
- ***Case 2:*** *Password is within the valid range (6-20):*
  - The number of steps required is the maximum of the number of replacements needed and the number of missing types.
- ***Case 3:*** *Password is too long (>20):*
  - Calculate the number of excess characters.
  - Use deletions to reduce the number of replacements needed for repeating sequences.
  - The total steps required are the number of deletions plus the maximum of the remaining replacements and missing types.

5. Edge Cases

a. Empty Password:

- The password is too short, so we must add 6 characters, including at least one lowercase, uppercase, and digit.

b. Password with All Repeating Characters:

- For example, "aaaaaa" requires replacements to break the repeating sequence.

c. Password with No Repeating Characters but Missing Types:

- For example, "abcABC" is missing a digit, so one replacement is needed.

d. Password with Maximum Length (50):

- The password is too long, so deletions are required to bring it down to 20 characters.

6. Complexity Analysis

Time Complexity:

- O(n), where n is the length of the password.
- We traverse the password once to check for missing types and repeating sequences.

Space Complexity:

- O(1)
- We use a constant amount of extra space for variables.

7. Alternative Approaches

a. Dynamic Programming:

- Use dynamic programming to track the minimum number of steps required to fix the password. However, this approach is more complex and less efficient for this problem.

b. Greedy Algorithm:

- Use a greedy approach to prioritize fixing the most critical issues first (e.g., breaking long repeating sequences).

8. Test Cases

Test Case 1:

- ***Input:*** "a"
- ***Output:*** 5
- ***Explanation:*** The password needs 5 more characters to meet the minimum length requirement and must include at least one uppercase letter and one digit.

Test Case 2:

- ***Input:*** "aA1"
- ***Output:*** 3
- ***Explanation:*** The password needs 3 more characters to meet the minimum length requirement.

Test Case 3:

- ***Input:*** "1337C0d3"
- ***Output:*** 0
- ***Explanation:*** The password is already strong.

Test Case 4:

- ***Input:*** "aaaaaa"
- ***Output:*** 2
- ***Explanation:*** The password has a sequence of 6 repeating 'a's, which requires 2 replacements.

Test Case 5:

- ***Input:*** "abcABC"
- ***Output:*** 1
- ***Explanation:*** The password is missing a digit, so one replacement is needed.

9. Final Thoughts

The Strong Password Checker problem is a good example of how to balance multiple constraints in a string manipulation task. The key is to break down the problem into smaller subproblems (length, character types, repeating sequences) and handle each case efficiently. The provided solution ensures that we use the minimum number of steps to make the password strong while considering all edge cases.