Strong password checker

We want to return the minimum steps required to make s strong ⇒ each change should fix as many problems as possible.

First we identify the problems & how to solve them.

Let len be the length of the string. We identify the solution as operation made (insert, replace, delete) - position in the string where operation in made.

1. Length problem

- if len < 6 ⇒ **insert - any position.**
- if len > 20 ⇒ **delete - any position.**

2. Missing characters problem (uppercase, lowercase, digit)

- to correct this we either **insert - any position** or we **replace - any position.**

3. Repeating characters problem

- for this problem any operation is allowed but we are constrained by the position.
- we see that for larger repeating sequences, replacing is the best operation since it makes fewer steps to solve the problem than insertion or deletion.

Now, we're interested to overlap the changes in order to fix as much as possible.

We split the problem in 2 cases:

1. **Len < 6**

For this case we notice that by inserting a character we can solve all three problems since we have an overlapping in the operations.

We notice that in this case the number of fixes for the first two problems always outnumber the third one, so fixing the first two takes care of the last one as well.

To solve the first one we can insert any character, for the second one we need that specific type of character ⇒ we assume the insertions made are first of the missing characters and if we need more we insert any character.

The solution to this first case will be max(missing_characters, 6 - len).

2. **Len ≥ 6**

Here, we try to overlap the first & third problem, second & third problem by "delete" and "replace".

First, we compute how many characters we need to delete. Take 0 if none and len ≤ 20. We add this to changes since no mather what we do next, it's a requirment to delete these characters.

We initialize an array that tells us the number of repeating characters that start at the coressponding position in the string.

We notice that by transforming these number in the form 3m+2 (where this 3m+2 is the largest number < the number is in the array) the replacing operation will be at its best performance.

By transforming we also keep count of the number of characters we need to delete.

⇒ while we still have characters to delete we substract from the repetitions 1 if the number is a multiple of 3 or min(2, no. characters to delete) if the number % 3 == 1

- we substract the minimum of those 2 because we need to make sure we still have characters to delete.

At this point we have 2 cases:

- we still have characters to delete so we delete as much as possible from the numbers ≥3 (repeating sequence)
- no more characters to delete so we just count how many replacing we need to fix the repeating characters problem

At the end we return changes (that holds the number of deleted characters) + max(missing_characters, no._of_replacing_weNeed) since by replacing we can solve the missing characters problem.