Kick Start 2021 - Round H

Analysis: Silly Substitutions

In this problem, each replacement shortens the string by one character, so the number of replacements is no more than $\mathbf{N}$. It is the complexity of each replacement that may bother us.

Test Set 1

The basic step that we are asked to do is the "replace all" operation; that is, to find all the occurrences of a given substring within a given string, and replace them with another string.

Many programming languages have a built-in function that does exactly this; for example, in both Python and Java it is called `replace()`. In other languages you may implement the "replace all" operation by using a few basic operations or completely from scratch. Either way, the typical implementation of "replace all" would take *linear* time, because it has to:

- search the entire string for all occurrences of the substring, and
- (if found) build the resulting string.

And then, we are asked to apply "replace all" until there is nothing to change. This would look like (pseudo-code):

```
while True:
  previous = s
  s = s.replace_all('01', '2').replace_all('12', '3')....replace_all('90', '1')
  if s == previous:
    return s
```

This loop makes up to $\mathbf{N}$ iterations, each iteration attempts the "replace all" operation 10 times, and, as said, each attempt takes $O(\mathbf{N})$.

Therefore, *complexity:* $O(\mathbf{N}^2)$ *per test case.*

Test Set 2

To reduce the complexity we should implement our own "replace all" and optimize the two things that it does:

- Instead of searching the entire string every time, we will maintain a set of "locations of interest". More accurately, 10 sets: all locations of the substring `01`, all locations of the substring `12`, and so on. How to define a location in a string that keeps changing? The next bullet solves this.
- To quickly replace characters, we will convert the string to a bidirectional linked list where each node holds a character. Thus, a location in the string is actually a pointer to a node, and each local change, such as removing two nodes or inserting a node, takes $O(1)$ time.

A "replace all" operation might still take linear time in the optimized version. For example, if $\mathbf{S} =$ `010101010101` then the first replacement (of all the `01`s to `2`s) will have to replace $\mathbf{N}/2$ occurrences. But then, the next operations will be either few enough or fast enough to compensate for the long one. This is known as amortized analysis.

More accurately, whenever a location becomes interesting, one of the future replacement operations will have to handle it (as said, by $O(1)$ manipulations of the linked list). How many times can locations become interesting? In the initial string, maybe all the $\mathbf{N}$ characters (except the last one) start as interesting. Later on, whenever we replace a pair of characters with a new one, that new character may turn one or both its neighbors to be interesting. For example, when changing `1013` to `123` it creates two interesting substrings: `12` and `23`. Finer analysis can show that we only need to mark the `12` as interesting, but it does not really matter for our conclusion.

So, since we replace characters at most $\mathbf{N}$ times, the event of a location becoming interesting can happen at most $3\mathbf{N}$ times (or $2\mathbf{N}$ with finer analysis).

Therefore, *complexity:* $O(\mathbf{N})$ *per test case*

Notes:

1. We can even mark "false interesting locations". That is, mark *all* the initial locations and *both* neighbors of every modification as interesting, even if they do not really start a pair that should be replaced (like `01`). Of course, our "replace all" operation will need to check if each "interesting location" should indeed be replaced, but this is a good practice anyway. The best part is that our analysis with the $3\mathbf{N}$ bound will still be valid.
2. The *set* of locations of interest is indeed a set - each location should be marked once and the order that we handle them does not matter. However, if your programming language makes it harder (higher complexity) to maintain a set, you may as well store the locations of interest in any other structure, like a vector or a list.