Code Jam 2016 - Round 3

Analysis: Go++

go++: Analysis

To make the analysis a tiny bit easier to follow, we'll typeset strings (**B**, **G**, and output from the Go++ programs) in monospace font, instructions from the first program in normal font, and instructions from the second program with an underline.

First, let's try to figure out when the problem is impossible to solve. It's clearly impossible when **B** is in **G**; that case is even in the sample input. What about otherwise?

Surprisingly, *if* **B** *is not in* **G***, the problem is always solvable*.

To prove this, I'll construct two Go++ programs that, when run as described in the problem statement, can produce *any* **L**-length string *except* **B**. That sounds harder than making one that just produces the strings in **G**, but sometimes, making the problem more difficult is a great way to gain the insight you need to solve the original problem. For this problem in particular, the set **G** is a red herring!

The first program

Given an input string **B**, construct the first program as follows:

1. Start with the inverse of every character of **B** (`0` ↦ 1, `1` ↦ 0).
2. Add a ? instruction after every 0 or 1 instruction.

For instance, if **B** = `111`, which could appear on the Small input, the first program is 0?0?0?. If **B** = `010`, which could only appear on the Large input, the first program is 1?0?1?.

Why does this work? Notice that by default, this program generates the opposite of **B**. In order to generate a different string, we need to override one or more of the ? instructions; that is, the programs need to be interleaved such that right before the ? from the first program, some other instruction from the second program changes what it prints. For instance, given the first program 1?0?1?, we can print `011` instead of `101` using the second program 0̲1̲, if it's interleaved as follows: 1**0̲**?0**1̲**?1?. In order to print **B**, we need to override all three ? instructions.

So how do we prevent that from happening? We write a second program that cannot override all three ? instructions. Our solution differs for the Small and Large inputs.

The second program (Small)

For the Small input, **B** is always a string of `1`s. That means the first program is always 0?, repeated **L** times. For instance, if **B** = `111`, the first program is 0?0?0?.

Our second program needs to be able to override any two of those ? instructions; that way, we can print any output with up to two `1`s. However, we can't override all three ? instructions, since that means our program might print **B**. So we need a second program that contains **L**-1 1̲s, but not **L** 1̲s. The solution to this is fairly simple: our second program simply consists of 1̲, repeated **L**-1 times. For instance, if **B** = `111`, the second program is 1̲1̲.

The second program (Large)

To solve the problem for arbitrary **B**, we need a second program that, when taken as a string, has every (**L**-1)-length string as a subsequence (so we can print any string except **B**), but doesn't have **B** as a subsequence. (Here, we use "subsequence" to mean any set of elements, ordered as they appear in the original sequence; they need not be adjacent.) We'll discuss how to produce such a program shortly.

Now, with our first program and this hypothetical second program, we can produce any **L**-length string except **B**. That's because the second program has every possible subsequence of length **L**-1, so we can override up to **L**-1 of the ? instructions of the first program. However, we cannot override all **L** of them to produce **B**, because the second program doesn't have **B** as a subsequence.

There are many ways to generate the second program. This one won't produce the shortest program, but it is perhaps the easiest to explain:

1. Start with a copy of **B**, excluding the last character.
2. Replace each character with a two-character sequence as follows: `0` ↦ 10, `1` ↦ 01.

For instance, if **B** = `010`, the second program is 1001. The first `0` becomes 10, the `1` becomes 01, and we ignore the last character of **B**.

We outlined the two requirements above for the second program to be valid:

- The second program must have every (**L**-1)-length string as a subsequence.
- The second program must not contain **B** as a subsequence.

Now, we will show that the steps above give us a second program that satisfies these requirements.

To get any (**L**-1)-length subsequence, split the second program into pairs of adjacent instructions. Each pair contains a 0 and a 1, since that's how we built the program. So we just pick one character from each pair, and we have any (**L**-1)-length string!

Now, let's try to get **B** as a subsequence of our second program. Say **B** = `010` and the second program is 1001. In order to get 010 as a subsequence, we start from finding the first 0 in 1001, which occurs at the second character. Next we find the first 1 after that, which is the fourth character. The pattern continues: because of how we've built the second program, we always go through two characters whenever we try to find **B** as a subsequence. And because the second program only has 2**L**-2 characters, we can't find all of **B** as a subsequence. Therefore, our second program is valid.