Code Jam to I/O for Women 2020 - Code Jam to I/O for Women

Analysis: Interleaved Output: Part 1

Test Set 1

One way of approaching this test set is to try all ways of assigning pairs of characters in the string (an uppercase `I` or lowercase `i` followed by an uppercase `O` or lowercase `o`) to particular computers, and in each case, recursively check whether the remaining string could have been produced. If, at any point during our check, our string starts with an uppercase `O` or lowercase `o`, then the string is invalid and we can abandon that branch of the search.

For example, if we have `IoiO`, we can try assigning the uppercase `I` and the uppercase `O` to an `IO` computer, then recursively check the remaining `oi` and find that it's invalid, then try instead assigning the uppercase `I` and lowercase `o` to an `Io` computer, and so on. Finally, find the largest possible number of advertisements of `IO`, across all valid assignments.

Test Set 2

The above strategy is too slow for Test Set 2 because there are too many possible pairs to check. Let's look for a simpler strategy.

As we scan through a string, whenever we encounter an uppercase or lowercase O, we must find a previous unused uppercase or lowercase I to pair it with (implicitly claiming that a particular computer printed both of those characters). Intuitively, to find an interpretation that maximizes the number of times `IO` is advertised, we would like to do the following as much as possible:

- Preferentially pair an uppercase `O` with an uppercase `I`, rather than with a lowercase `i`
- Preferentially pair a lowercase `o` with a lowercase `i`, rather than with an uppercase `I`

However, we may not always be able to get what we want! For example, suppose we are scanning through the input `IoiO`. When we reach the lowercase `o`, we have no choice but to pair it with the preceding `I`.

We can prove, though, that if we adhere to the above preferences whenever we have a choice, we will find the correct answer. Suppose we are carrying out this method, and we reach an uppercase `O` — call it O_1 — that we can match to either some previous unmatched uppercase `I` — call it I_{prev} — or some previous unmatched lowercase `i` — call it i_{prev}. Suppose we choose to match O_1 with i_{prev}. Then we must eventually match I_{prev} with some later uppercase or lowercase O. (It is guaranteed that at least one of these exists because the input satisfies a balanced parentheses constraint, as outlined in the Limits section.) Call that later O O_2. We will obtain 1 "point" (i.e. we will be able to claim that an `IO` computer advertised its event once) if O_2 is uppercase, and 0 points if O_2 is lowercase.

But then observe that we could have instead matched the uppercase O_1 with I_{prev} (scoring 1 point *for sure*), and matched the lowercase i_{prev} with the same O_2. This argument holds true no matter which O_2 we picked. So preferentially matching with the I_{prev} is no worse, and may be better.

A similar argument holds for the situation in which we reach a lowercase `o` that we can match to either a previous uppercase `I` or a previous lowercase `i`, and it tells us to preferentially match

with the i_{prev}.

We have covered the only two cases in which we can make a decision. Since no other strategy is better in either case, our strategy is an optimal one. It is possible that we may have a choice of e.g. two previous uppercase `I`s and one previous lowercase `i`, but then it does not matter which of the uppercase `I`s we pick, since whichever one we don't use will still be "previous" for the purposes of later decisions.

A simpler way to frame this strategy is as follows: scan through the string from left to right, keeping two counts: C_I, the number of unmatched uppercase `I`s seen so far, and C_i, the number of unmatched lowercase `i`s seen so far. When we encounter an uppercase `O`, score one point and decrement C_I if C_I is positive, or otherwise decrement C_i. When we encounter a lowercase `o`, decrement C_i if C_i is positive, or otherwise decrement C_I. Notice that the rules of the problem guarantee that these counts will never simultaneously become zero at the time we encounter an uppercase `O` or lowercase `o`.