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

Analysis: Interleaved Output: Part 2

An important difference between this problem and the Part 1 version is that in this version, for each of the events, there is only a single computer that can print it. That is, there are only four computers printing simultaneously for the four events, `IO`, `io`, `Io` and `iO`. Our optimal greedy strategy from the Part 1 analysis is not compatible with this constraint.

For example, consider the sample input `IiOioIoO`. Our Part 1 strategy would find an interpretation in which `IO` was advertised twice. But this interpretation requires two separate computers to have printed `io`, since one computer cannot print an `i` followed by another `i`... and in Part 2, we only have one `io` computer.

Let us find a way to turn this new constraint into an advantage. Regardless of how many times each computer ends up advertising its event, each computer can be in only one of two states at any given time: either it is about to print its first character, or about to print its second character. So, at any point, the system of 4 computers is in one of $2^4 = 16$ states. This situation, in which we have only a limited number of states to consider, is ideal for a dynamic programming approach in which we step through the string and keep track of the best option seen so far for each of those states.

Let us describe our 16 states in terms of which computers are about to print their *second* characters. For example, the state {`IO`, `iO`} means that the `IO` and `iO` computers need to print their second character (`O`) next, and the `io` and `Io` computers need to print their first character (`I` or `i`) next.

We will also define the "score" of a prefix of the input string p and a state s as the number of completed `IO` events designated after processing p ending in state s. Since scores only increase, we only care about the maximum score for each combination. The final answer to the problem is the score for p = **S** and s = {}, that is, when we have processed the entire string and no computer has printing pending. We will also define the "score" of a partial string as the number of completed `IO` events designated so far in that string.

Consider the input string `IiOioIoO`. When we process the first `I`, we must decide which computer printed that character — that is, after that choice, we can either be in the state {`IO`} or the state {`Io`}. With dynamic programming, we simultaneously keep track of all of these possible worlds; we are not actually committing to a particular choice. After we process the next character, depending on our interpretation, we can be in one of four states: {`IO, iO`}, {`IO, io`}, {`Io, iO`}, or {`Io, io`}. Then, moving on to the next `O`, we find that {`Io, io`} is inconsistent, and the possible states become {`IO`} (with a score of 0 so far), {`iO`} (with a score of 1), {`io`} (with a score of 0), or {`Io`} (with a score of 0). We continue this until we have finished processing the string, and then we have our answer.

Our dynamic programming approach has O(L) stages (where L is the input string length), and each stage has a constant processing time since there are no more than 16 states to consider. This results in an overall time complexity of O(L), but notice that we have only been able to wave away a constant factor because the number of events/computers was small. With E possible events, the complexity would be $O(L \times 2^E)$.)