Contest Analysis

Fractiles: Analysis

This problem is more about analyzing an existing algorithm than writing a new one. Once you understand how more complex artwork depends on the original sequence, you can solve the problem with a short piece of code.

The first thing to notice is that if the original sequence is all `L`s, the artwork will be all `L`s, no matter what the value of **C** is. If we choose some set of tiles that all turn out to be `L`s for some original sequence other than all `L`s, then our solution is invalid, because we won't be able to tell whether the artwork was based on that original sequence or on an original sequence of all `L`s. This means we have to come up with a set of positions to check out such that for any original sequence besides all `L`s, we will see at least one `G`.

Small dataset

In the Small dataset, since we can check as many tiles as the length of the original sequence, we may be tempted to try to reconstruct it in full. And while this is possible (we'll get there in a moment), there is an easier alternative. The simplest solution, as it turns out, is to always output the integers 1 through **K**. It can be easily proved that it works with the following two-case analysis. Let us call the original sequence O, and let A_i be the artwork of complexity i for a fixed O.

1. Suppose that O starts with an `L`. Let us prove that each A_i starts with O. This is trivially true for $A_1 = O$. Now, if A_i starts with O, it also starts with an `L`, and since the transformation maps that first `L` into a copy of O, A_{i+1} starts with O. By induction, each A_i starts with O. Then, by checking positions 1 through **K**, we are checking a copy of the original sequence O, so if there are any `G`s in O, we will see a `G`.

2. Suppose instead that O starts with a `G`. Let us prove that each A_i starts with a `G`. This is trivially true for $A_1 = O$. Now, if A_i starts with a `G`, then A_{i+1} also starts with a `G`, since the transformation maps that `G` at the start of A_i to **K** `G`s at the start of A_{i+1}. By induction, each A_i starts with `G`. Then, since we are checking position 1, we will see a `G`.

Since we will see at least one `G` for any original sequence that is not all `L`s, and only `L`s for the original sequence that is all `L`s, we have answered the question successfully. Notice that this also proves that there is no impossible case in the Small dataset.

The proofs above hint at another possible solution for the Small dataset that gets enough information from the tiles to know the entire O. We will explain it not only because it is interesting, but also because it is a stepping stone towards a solution for the Large dataset.

We have seen that position 1 of any A_i is always equal to position 1 of O. Is there any position in A_i that is always equal to position

2 of O? It turns out that there is, and the same is true for any position of O.

Consider position 2 of O as an example. It is position 2 in $A_1 = O$. When A_2 is produced from A_1, the tile at position 2 of A_1 determines which tiles will appear at positions **K** + 1 through **K** + **K** of A_2. In particular, the second of those tiles, the tile at position **K** + 2 of A_2, is the same as the tile at position 2 of A_1. Then, it follows that position **K** + 2 of A_2 generates positions **K***(**K** + 2 - 1) + 1 through **K***(**K** + 2) of A_3, and the second of those tiles, at position **K***(**K** + 2 - 1) + 2 of A_3, is also a copy of position 2 of O. You can follow this further to discover which position of $A_\mathbf{C}$ is equal to position 2, or you can write a program to do it for you. Similarly, for each position of O there is exactly one "fixed point" position in $A_\mathbf{C}$ that is always equal in value, and you can get those with a program by generalizing the procedure described for position 2. If you check out all of those positions, you obtain a different result for every possible O, which makes the solution valid.

Large dataset

The reasoning that we just used to find fixed points will help us solve the Large. Each position in A_i generates **K** positions in A_{i+1}. So, indirectly, each position in A_i also generates $\mathbf{K}^2$ positions in A_{i+2}, $\mathbf{K}^3$ positions in A_{i+3}, and so on. Let us say that a position in $A_{i+\mathrm{d}}$ is a descendant of a position p in A_i if it was generated from a position in A_{i+d-1} generated from a position in A_{i+d-2} ... generated from position p in A_i. Notice that a `G` in any given position of any A_i implies a `G` in all descendant positions. However, if there is an `L` in position p of A_i, a descendant position (p - 1)***K**+d (with 1 ≤ d ≤ **K**) of A_{i+1} will be equal to position d of O. So, position (p - 1)***K**+d of A_{i+1} is an `L` if and only if both position p of A_i and position d of O are `L`s. If we take this further, we arrive at a key insight: any position of any A_i is an `L` if and only if a particular set of positions in O are `L`s.

We can find those positions by thinking about the orders in which the descendants at each level were produced. For instance, for **K**=3, position 8 of A_3 is descendant number 2 of position 3 of A_2, which in turn is descendant number 3 of position 1 of A_1. That means that position 8 of A_3 is `L` if and only if positions 2, 3 and 1 of O are all `L`s. So, just by looking at position 8 of A_3, we know whether the original sequence had a `G` in at least one of those three positions.

Generalizing this, if we start at position p_1 of $A_1 = O$, and take its p_2-th descendant in A_2, and then take its p_3-th descendant in A_3, and so on, until taking the $p_\mathbf{C}$-th descendant in $A_\mathbf{C}$, we have a single position that tells us whether the original sequence has a `G` in positions p_1, p_2, ..., $p_\mathbf{C}$. And, conversely, for any position in $A_\mathbf{C}$, we can find a corresponding sequence of **C** positions that lead to it. So, each position we check on $A_\mathbf{C}$ can cover up to **C** positions of O, and will cover exactly **C** positions if we make the right choice. Since we need to cover all **K** positions of the original sequence, that means the impossible cases are exactly those where **S***__C__ < **K** — that is, where getting **C** positions out of every one of our **S** tile choices is still not enough. For the rest, we can assign a list of positions [1, 2, ..., **C**] to tile choice 1, [**C**+1, **C**+2, ..., 2**C**] to tile choice 2, and so on until we get to **K**. If the last tile choice has a list shorter than **C**, we can fill it up with copies of any integer between 1 and **K**. Now all we need to do is match each of these lists to a position in $A_\mathbf{C}$, which we can do by following the descendant path (descendants of position p are always positions (p - 1)***K**+1 through (p - 1)***K**+**K**). This simple Python code represents this idea:

```
def Solve(k, c, s):
  if c*s < k:
    return []  # returns an empty list for impossibl
  tiles = []
  # the list for the last tile choice is filled with
  # i is the first value of the list of the current 
  for i in xrange(1, k + 1, c):
    p = 1
    # j is the step in the current list [i, i+1, ...
    for j in xrange(c):
      # the min fills the last tile choice's list wi
      p = (p - 1) * k + min(i + j, k)
    tiles.append(p)
  return tiles
```