Code Jam 2020 - Round 3

Analysis: Naming Compromise

Test Set 1

In Test Set 1, the limits are small enough that we might be tempted to just try lots of possible outputs and keep the best one. However, since there is no a priori limit on the output length, and we can use any of the English alphabet letters in the output, we need to think a little bit first. In addition, we have to actually implement the calculation of edit distance, which we can learn about via various online resources, including that Wikipedia link.

The distance between two strings of length up to 6 is at most 6, since we could just replace every character in the longer string (or an arbitrary one, if they are of the same length) to match the other string, and delete any excess. Moreover, the distance between two strings is never less than the absolute difference in length between them, because we would need to add at least that many characters to the shorter string (if one is indeed shorter) just to make it as long as the other string. It follows that the output should never be longer than 12 characters.

In addition, if the output N contains a letter that is not present in either of the input strings, exchanging that letter in N for a letter contained in one of the inputs cannot increase either distance, which can be proven by induction by inspecting the recursive definition of edit distance. Therefore, we can limit our search to strings of length up to 12 over the alphabet {`X`, `Y`, `Z`}. There are less than a million candidates, and calculating the edit distance for such short strings is really efficient, so this is fast enough to solve Test Set 1.

It is possible to prove tighter limits on the output, and this can drastically reduce the number of candidates and make the solution even faster. We leave the details to the reader, but consider whether we ever really need an answer longer than 6 letters. If we have `ABCDFG` and `ACDEFG`, for example, then instead of compromising on the seven-letter `ABCDEFG` by adding `E` to the first name and `B` to the second name, we could just as easily delete `B` from the first name and `E` from the second name, compromising on the shorter `ACDFG` instead.

Test Set 2

For Test Set 2 we need to use some additional insights. The most important one is noticing that edit distance is actually a distance, and has typical properties of distances like being reflexive and satisfying the triangle inequality. Let e(s, t) denote the edit distance between two strings s and t. For an output N, we want e(**C**, N) + e(**J**, N) = e(**C**, N) + e(N, **J**) to be minimal. By the triangle inequality, e(**C**, **J**) is a lower bound on this quantity, and an achievable one. For example, we can set N equal to **C** since e(**C**, **C**) = 0.

By the reasoning above, we need to find an N such that e(**C**, N) + e(N, **J**) = e(**C**, **J**) and |e(**C**, N) - e(N, **J**)| is as small as possible. Luckily, the definition of edit distance hints at a way of doing that. If the edit distance between **C** and **J** is d, it means that there is a path of d valid operations on **C** that results in it turning into **J**. Formally, there are d - 1 intermediate strings S_1, S_2, ..., S_{d-1} such that e(**C**, S_i) = i and e(S_i, **J**) = d - i. Therefore, it suffices to pick $S_{d/2}$. If d is odd, both rounding up and down work, since for both possibilities, |e(**C**, N) - e(N, **J**)| = 1.

To find S_1, S_2, ..., S_{d-1}, we can use a common technique to reconstruct the path that achieves an optimal result as found via dynamic programming (DP).

For example, let $\mathbf{C}_a$ be the prefix of length a of **C** and $\mathbf{J}_b$ be the prefix of length b of **J**. The usual algorithm to compute edit distance is based on a recursive definition of a function f(a, b) that returns e($\mathbf{C}_a$, $\mathbf{J}_b$) (see the link from the previous section for details). Similarly, we can define g(a, b, k) as a fuction that returns e($\mathbf{C}_a$, $\mathbf{J}_b$) and also a string S such that e($\mathbf{C}_a$, S) = k and e($\mathbf{J}_b$, S) = e($\mathbf{C}_a$, $\mathbf{J}_b$) - k. The new function g can be defined with a similar recursion as f, and then memoized for efficiency. The details are left as an exercise for the reader.