Code Jam 2021 - Qualification Round

Analysis: Moons and Umbrellas

Test Set 1

In Test Set 1, the size of the mural is small enough that we can test every possible final mural. Let ℓ be the length of $\mathbf{S}$. We can either check for each string of length ℓ consisting only on `C`s and `J`s whether it agrees with $\mathbf{S}$ in the non-`?` spots, or brute-force only the `?`s directly. For each valid way of finishing the mural, we can calculate the total cost to Cody-Jamal, and keep a running minimum to output at the end. Since we check at most 2^ℓ final murals, and for each we need only an additional linear pass to check the score, this algorithm takes $O(2^\ell \cdot \ell)$ time, which is fast enough for Test Set 1.

Test Set 2

In Test Set 2, ℓ can be up to 1000, so an exponential algorithm will not do. However, a simple observation can yield a much faster algorithm: since $\mathbf{X} > 0$ and $\mathbf{Y} > 0$, we would like to avoid inserting `CJ` or `JC` into the string. So if all letters surround a consecutive substring of `?`s are the same letter a, then making all those `?`s into a adds no `CJ`s or `JC`s. If the substring is surrounded by a on the left and b on the right, with $a \neq b$, on the other hand, at some point there will be a change from a to b, so we will be forced into an occurrence of ab. Making all the `?`s into as (or bs) makes sure that we only add that one forced occurrence and nothing else. This yields a greedy algorithm that can be implemented in multiple ways, some taking as little as $O(\ell)$ time. Notice that the cost is the same on the resulting string than on the string where `?`s are removed instead of replaced, which leads to a 1-line implementation:

```
S.replace('?', '').count('CJ') * X + S.replace('?', '').count('JC') * Y
```

Extra credit: Test Set 3

In the solution for Test Set 2 we assume that minimizing the number of occurrences of `CJ` and/or `JC` is best, which is true only when $\mathbf{X}$ and $\mathbf{Y}$ are non-negative. This means that Test Set 3 needs a completely different solution. In this case, a technique that would not normally show up in the second problem in a Qualification Round: dynamic programming.

We can take the function $f(s)$ that we need to compute and define it recursively as follows:

- $f(\texttt{?}s) = \min(f(\texttt{C}s), f(\texttt{J}s))$
- $f(a\texttt{?}s) = \min(f(a\texttt{C}s), f(a\texttt{J}s))$ for $a \in \{\texttt{C}, \texttt{J}\}$
- $f(aas) = f(as)$ for $a \in \{\texttt{C}, \texttt{J}\}$
- $f(\texttt{CJ}s) = \mathbf{X} + f(\texttt{J}s)$
- $f(\texttt{JC}s) = \mathbf{Y} + f(\texttt{C}s)$
- $f(s) = 0$ if the length of $s \leq 1$

You can verify that the cases above form a partition and properly define the recursion. We have a constant number of cases, each of which can be computed in constant time not counting the recursion calls, so the time complexity of the solution is $O(D)$ where D is the size of the domain of the function. You can check that every value of f that is ever required to calculate $f(\mathbf{S})$ can be defined by a suffix of the input $\mathbf{S}$ and at most two extra characters at the beginning. This means D is linear in the length of $\mathbf{S}$, and so is the time complexity of the algorithm. Notice that we need to represent each element in the domain in constant space for this to work out exactly

as mentioned, for example representing suffixes of $\mathbf{S}$ by just their length or an index into $\mathbf{S}$. However, for the low limits of this problem, even a larger representation using copies of the full suffix is fast enough.