Code Jam 2016 - Round 1B

Analysis: Close Match

Close Match: Analysis

Small dataset

A brute force algorithm will work just fine for the Small. For each case, we can try all possible ways to fill in the `?`s and see which way produces the smallest absolute difference. It is important to break any ties first by the first score, or, if necessary, by the second score. Even in the worst case for brute force, `??? ???`, there are only 10^6 = one million different possibilities to check.

Large dataset

Many Code Jam problems look complex and challenging, but have a surprisingly simple solution. But sometimes we throw in just the opposite sort of problem! In this case, it's easy to come up with a tantalizing greedy approach that will sometimes fail. We'll present an approach that gets greedy only when it's safe to do so.

We will investigate various different ways to fill in the `?`s, and we will constantly keep track of the best set of scores (using tiebreaker rules) seen so far. We will look at the pair of characters in the first position of each string, then the pair of characters in the second position of each string, and so on.

When we look at a pair of characters, we must decide how to fill in any `?`s. We may be able to make those two digits of the final scores be the same, either by filling in `?`s appropriately, or because the two characters are already the same digit and we have no choice. If we're lucky, and we never encounter a position where the two strings have different digits, we can make every pair of digits in the final scores the same! This is as good as it gets; our difference will be 0.

For example, for the case `?1? 2??`, we can make all the digits equal to get `210 210`. We choose `0`s for any pair of `?`s to satisfy the tiebreaker rules.

However, we won't have this option in every test case. If the strings have different digits at some position, then there's no way the score difference can be 0. Moreover, we may even need to introduce our own difference before that first existing point of difference! For instance, in the case `??0 ?99`, we don't want to just make the digits equal until we get to the first point of difference, because that would leave us with `090 099`, whereas we can do better with `100 099`.

So, if we have two different digits at any position in the strings, then either we can make everything equal up to that first existing point of difference, or we can introduce a first point of difference earlier. That is, we will start from the left, and for some number of positions (possibly zero), we will make the digits in the final scores equal. Then we will encounter (or make) our first point of difference.

Once we've reached our first point of difference, we know something crucial: *which of the two final scores is larger*. Because the scores have been equal up until the first point of difference, the score with the larger digit in this position is guaranteed to be larger, no matter how we fill in any `?`s beyond this point.

Moreover, we know exactly how to fill in those `?`s to minimize the difference between the two scores! We have no reason to make the larger number any larger, so all of its `?`s should become `0`s. And we want to make the smaller number as large as possible, so all of its `?`s should become `9`s.

So, as we step through our strings from left to right, for every position of the strings, we'll do the following:

1. If we encounter two identical digits, move on.
2. If we encounter two different digits, fill in all remaining `?`s, compare those scores with the best we've seen so far, and stop. We're done!
3. If we encounter a digit and a `?`:
   - If the digit is not `9`: try changing the `?` to 1 more than that digit. This would introduce a first point of difference, so we know how we would fill in all remaining `?`s. Compare those scores with the best we've seen so far. Then...
   - If the digit is not `0`: try changing the `?` to 1 less than that digit. This would introduce a first point of difference, so we know how we would fill in all remaining `?`s. Compare those scores with the best we've seen so far. Then...
   - Change the `?` to that digit and move on. (Why didn't we try changing the `?` to all possible digits? We can get away with doing this, but it's not necessary. If we have control over the digits at the first point of difference, there's no reason to make them differ by more than 1; this would just make the overall difference larger, regardless of what we do with the rest of the strings.)
4. If we encounter two `?`s:
   - Try changing the first `?` to a `0` and the second `?` to a `1`. Figure out the scores and compare. Then...
   - Try changing the first `?` to a `1` and the second `?` to a `0`. Figure out the scores and compare. Then...
   - Change both `?`s to 0s and move on. (Again, there is no reason to make the `?`s differ by more than `1`.)

We must make sure our implementation also checks the case in which there is no point of difference. Once we're done, the best scores will be our answer.

This approach has some room for improvement, but it solves the Large very quickly. It makes one full pass through the strings, and at each position, it might make up to 2 additional partial passes, so the worst case running time is O($\mathbf{N}^2$), where **N** is the length of **C** (and **J**). We leave an O(**N**) approach as an exercise!