Code Jam 2016 - Round 1B

Analysis: Getting the Digits

Getting the Digits: Analysis

Small dataset

Since the string can be no longer than 20 letters in the Small dataset, the phone number can be no longer than six digits (since the digits with the shortest spelled-out length, `ONE` and `TWO`, have three letters, and 7 * 3 > 20). A key observation is that this means there are no more than a million possible answers; in fact, there are far fewer than that, since only phone numbers with digits in nondecreasing order are valid.

So, you can look at all possible phone numbers from `0` to `9`, `00` to `99` (note that `00` is different from `0`), and so on, up to `000000` through `999999`, and save the numbers that have their digits in nondecreasing order. (In fact, for that last range, you only need to go up to `222222`.) Then, turn each of those into letters. There are multiple possible orderings for those letters — `1` could be given as `EON`, `ENO`, or `NEO`, among others — but you can just take the alphabetically ordered version. `0` becomes `EORZ`, `1` becomes `ENO`, ..., `01` becomes `EENOORZ`, and so on.

Then you can make a dictionary (hash table) with these strings as keys and the phone numbers as values. (It turns out that each of those keys maps to only one phone number — more on that later.) To solve a test case, take the string your friend gave you, sort it alphabetically, and then look that up in the dictionary. (Your friend could have given you the same string in any order, so you can permute it however you want without changing the identity of the underlying phone number.)

There's no need to generate this dictionary every time you solve a test case; you can do it just once, before your program starts processing test cases. You can even produce the entire dictionary before you download an input file and store it in your source code or in a separate file that your source code reads, as long as this does not cause your source code to exceed the standard limit of 100kB.

Large dataset

The dictionary for all 1000-digit phone numbers would be too huge to generate and store beforehand, let alone produce on the fly.

One tempting approach is to greedily remove spelled-out digits: for instance, keep taking the letter set `NINE` (one `E`, one `I`, and two `N`s) away until you no longer can, then keep taking the letter set `EIGHT` away, and so on. But this won't necessarily work — what if there are no `9`s in the actual phone number, and the first `NINE` you try to take away is really the `N` from a `SEVEN`, the `I` from a `SIX`, and the `NE` from a `ONE`? This will eventually leave you with a mess of letters from which no digit's letters can be removed!

It turns out that you can make this method work... as long as you pick the right order in which to remove the digits! For instance, `FOUR` is the only digit out of the ten that contains a `U`. So, if you see three `U`s in the string, there must be three instances of `FOUR`. You can safely remove three `F`s, three `O`s, three `U`s, and three `R`s, and record that the phone number has three `4`s. Once all the `FOUR`s are gone, `FIVE` is the only remaining digit with an `F`, so you can remove as many `FIVE`s as there are `F`s, and so on. If you had tried to do this in the other order, starting by

removing as many FIVEs as there were Fs, it might not have worked, depending on how many FOURs, SIXes, SEVENs, etc. were in the number!

Here is one safe order in which to remove digits, based on their letters that are unique at the time of removal: **Z**ERO, SI**X**, EI**G**HT, T**W**O, FO**U**R, **F**IVE, SE**V**EN, **T**HREE, N**I**NE, **O**NE. Note, for example, that even though all of the letters in ONE appear in other digits besides ONE, by the time we get to removing ONEs, there are only ONEs left.

The existence of an ordering like this also explains why we didn't have to worry about two different phone numbers having the same alphabetized string in our dictionary for the Small solution. Two different phone numbers cannot produce the same string because no subset of digit words is a linear combination of other digit words. This would *not* necessarily hold for arbitrary sets of words, though. For example, in a language in which the digit words are AB, AC, BD, and CD, given the string ABCD, it is impossible to know whether it was formed from one AB and one CD, or from one AC and one BD.

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 $O(\mathbf{N})$ approach as an exercise!

Code Jam 2016 - Round 1B

Analysis: Technobabble

Technobabble: Analysis

Small dataset

Each topic on the list could be "faked" or "un-faked." One natural brute force solution is to enumerate all possible orderings of the topics, and pick the one that has the most faked topics.

It's easy to check whether a topic could have been faked: simply check whether the first word of the topic appears as a first word earlier in the list, and the second word of the topic appears as a second word earlier in the list. Note that marking a topic as faked does not change whether or not topics later in the list can be marked as faked, so using this strategy, it is optimal to always count a topic as faked if possible.

However, there are **N**! possible orderings, which is too large to enumerate even for the small dataset where **N** ≤ 16 (16! is on the order of 21 trillion). We need a better approach. Rather than trying to maximize the number of faked topics, let's think about the reverse problem: trying to minimize the number of un-faked topics.

The key observation is that any possible set of un-faked topics must contain every first word at least once and every second word at least once — otherwise, there would be a faked topic that contained a word that was not available for the faker to use. Conversely, any set containing every first word at least once and every second word at least once could be a possible set of un-faked topics — simply put all the topics from the un- faked set at the top of the list. So, the question we need to answer is this: *What is the smallest set of topics that contains every first word at least once and every second word at least once?*

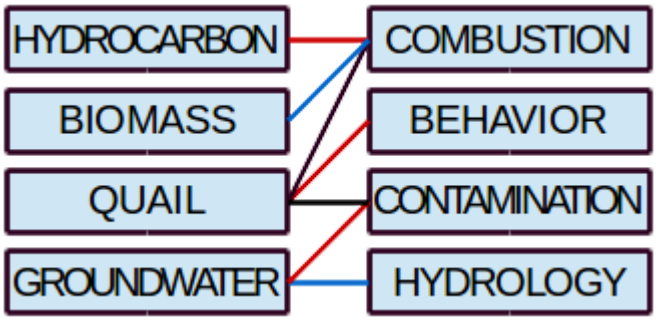
A brute-force approach, which works for the Small dataset, is to enumerate all subsets of topics and pick the subset with the fewest topics that covers every first word at least once and every second word at least once. Since there are $2^{\mathbf{N}}$ subsets, this solution runs in exponential time, which is fine for the Small dataset (2^{16} = 65,536).

Large dataset

For the large dataset, the exponential time solution will not work. It turns out that there is a polynomial time solution to the problem. We will illustrate the solution using graph theory.

Let each word be a vertex in a bipartite graph in which each topic is an edge connecting two vertices. The sample input below corresponds to the following graph (we'll explain the colors of the edges in a moment).

```
HYDROCARBON COMBUSTION
BIOMASS COMBUSTION
QUAIL COMBUSTION
QUAIL BEHAVIOR
QUAIL CONTAMINATION
GROUNDWATER CONTAMINATION
GROUNDWATER HYDROLOGY
```



The problem we're trying to solve on this graph is the *minimum edge cover*; that is, finding the smallest set of edges such that each vertex is connected to at least one edge. This corresponds to the smallest set of topics containing every first word at least once and every second word at least once.

The minimum edge cover problem is related to finding a *maximum matching* of a graph (the largest set of edges without any common vertices): the two will always have the same number of connected components. If it's not immediately obvious why this is the case, convince yourself by drawing some graphs on paper and trying to come up with a counter-example. We can additionally observe that every vertex left out of a maximum matching must be connected to a vertex in the maximum matching; otherwise, we could have added that pair to the maximum matching. With these facts, we can use a two-step algorithm to compute a minimum edge cover on our bipartite graph:

1. Find a maximum cardinality bipartite matching of the graph, which can be done in polynomial time using an approach such as the Ford-Fulkerson algorithm or the Hopcroft–Karp algorithm. One such matching is shown above in red.
2. Iterate over the remaining edges, and greedily add edges that connect to an unused vertex. The edges added by this step are shown above in blue.

In fact, all we really need to know is the *size* of a minimum edge cover: the number of edges in a maximum matching plus the number of vertices not included in a maximum matching. The solution to the problem is then simply the total number of edges (topics) minus the size of a minimum edge cover.