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.