Coding Challenge

Welcome!

We're glad you're interested in joining the Kin Insurance team! But first, a challenge.

The coding challenge below presents problem statements and requirements for a specific business scenario. Your goal: to implement a solution for this scenario and show off your coding chops.

Your submission will be assessed subjectively and relative to your experience. In other words, there's not a strict pass/fail measurement we have in mind. The aim is to learn more about your thought process and foster a discussion. You are not necessarily expected to finish all four User Stories. Complete as much as you can. If you are selected for a technical interview, your interviewer will pick up wherever you left off in a pairing session.

While there are particular business rules in each system, you may have to make some product decisions of your own. Requirements may also seem unclear, but we prefer not to provide additional information while your work is in progress. If something seems nebulous, make your own call and provide rationale.

Your submission should showcase your capabilities. Be creative and play to your strengths. Ruby, our core programming language, is strongly preferred.

How to Submit

Take your time completing this assignment, but don't spend too long on it. We want to see your work under normal conditions and understand that you may have other commitments

In order to submit this challenge *we require you configure a private GitHub repository and invite @oelbrenner, @kevingreene and @quigebo as collaborators*. We will fork your submission once it's complete. No changes will be accepted after the fork so if there's a last minute change just save it for discussion. Please include instructions for how to install and run your code. You must also list any system dependencies (eg. Ruby 2.3, Erlang runtime, JDK8, etc.).

Please do not open your repo to the public or share your solution.

The Challenge: Kinsurance OCR

User Story 1

Kin has just recently purchased an ingenious machine to assist in reading policy report documents. The machine scans the paper documents for policy numbers, and produces a file with a number of entries which each look like this:

```
 _  _     _  _  _  _  _
| _| _||_||_ |_   ||_||_|
||_  _|  | _||_|  ||_| _|
```

Each entry is 4 lines long, and each line has 27 characters. The first 3 lines of each entry contain a policy number written using pipes and underscores, and the fourth line is blank. Each policy number should have 9 digits, all of which should be in the range 0-9. A normal file contains around 500 entries.

Your first task is to write a program that can take this file and parse it into actual numbers.

User Story 2

Having done that, you quickly realize that the ingenious machine is not in fact infallible. Sometimes it goes wrong in its scanning. So the next step is to validate that the numbers you read are in fact valid policy numbers. A valid policy number has a valid checksum. This can be calculated as follows:

```
policy number:  3  4  5  8  8  2  8  6  5
position names:  d9 d8 d7 d6 d5 d4 d3 d2 d1

checksum calculation:
(d1+(2*d2)+(3*d3)+...+(9*d9)) mod 11 = 0
```

Your second task is to write some code that calculates the checksum for a given number, and identifies if it is a valid policy number.

User Story 3

Your boss is keen to see your results. They ask you to write out a file of your findings, one for each input file, in this format:

```
457508000
664371495 ERR
86110??36 ILL
```

ie the file has one policy number per row. If some characters are illegible, they are replaced by a ?. In the case of a wrong checksum (ERR), or illegible number (ILL), this is noted in a second column indicating status.

Third task: write code that creates this file in the desired output.

User Story 4

It turns out that often when a number comes back as ERR or ILL it is because the scanner has failed to pick up on one pipe or underscore for one of the figures. For example

```
    _  _  _  _  _  _     _
|_||_|| ||_   |  |  ||_
  | _||_||_|  |  |  | _|
```

The 9 could be an 8 if the scanner had missed one |. Or the 0 could be an 8. Or the 1 could be a 7. The 5 could be a 9 or 6. So your next task is to look at numbers that have come back as ERR or ILL, and try to guess what they should be, by adding or removing just one pipe or underscore. If there is only one possible number with a valid checksum, then use that. If there are several options, the status should be AMB. If you still can't work out what it should be, the status should be reported ILL.

Your final task is to write code that does the guess work described above to remove as many ERR and ILL as can safely be done.

Things to watch for

We ask that you find a way to write out 3x3 cells on 3 lines in your code, so they form an identifiable digits. Even if your code actually doesn't represent them like that internally. We'd much rather read

```
"    " +
"|_|" +
"  |"
```

than

```
"    |_|   |"
```

any day.

Some gotchas to avoid:

- Be very careful to read the definition of checksum correctly. It is not a simple dot product, the digits are reversed from what you expect.
- The spec does not list all the possible alternatives for valid digits when one pipe or underscore has been removed or added
- Don't forget to try to work out what a `?` should have been by adding or removing one pipe or underscore.

Good Luck!