Added material: Error detection in decimal codes

(ie detecting errors in numbers that are expressed in the usual way).

The goal of these notes is to describe a technique commonly used to catch errors resulting from common mistakes by humans.

E.g.:
- Single digit errors, where "x" is replaced with "y", like 687 becomes 627
- Transposition errors, like "xy" becoming "yx", such as 687 becoming 678.
- Phonetic errors, like "16" ⟶ "60" ("sixteen" sounds like "sixty").

Example: Suppose we want to send an n-digit number, such as

$$a_0 a_1 a_2 \cdots a_{n-1}$$

where $a_i \in \{0, 1, 2, .., 9\}$. One way of adding a bit of "error detecting ability" to the system would be to send n+1 digits

$$a_0 a_1 a_2 \cdots a_n$$

where $a_n = a_0 + a_1 + a_2 + \cdots + a_{n-1}$, $\underline{\text{mod} 10}$. Then if any single-digit error is made in $a_0, \ldots, a_{n-1}$, our recipient will know there is a problem.

For concreteness, say Alice wants to send 12345 to Bob. She computes

$$1+2+3+4+5 = 15 = 5 \mod 10$$

and sends 123455 to Bob.

However there's a mistake introduced somewhere along the way, and Bob receives 223455. He checks the sum:

$$2+2+3+4+5 = 16 \mod 10$$
$$= 6$$

and realizes there must be a mistake, so asks her to send it again. <u>Note</u>: This scheme will detect single-digit errors, but detects <u>no</u> transposition errors (that's bad).

<u>Remark</u>: If you are wondering how this connects to our previous formalism of codes, error-detecting, etc., here is what is going on. In the above example, we want to send n-digit strings $a_0 \cdots a_{n-1}$, and we agree to do this using the code

$$C = \{a_0 \cdots a_n \mid a_n = a_0 + \ldots + a_{n-1}\}$$

consisting of codewords of length n, in the alphabet $\{0, 1, 2, \ldots, 9\}$. Our previous definition of "error detecting" no longer applies, since it used Hamming distance (and Hamming distance is not defined for decimal codes).

So now, we are using the phrase

"C detects single digit errors" to mean that any change of a single digit of a string $s \in C$ results in a string $s' \notin C$. In general, we'll use the phrase

"C detects errors of type X" to mean that a change of a certain type to any $s \in C$ results in $s \notin C$.

Example: UPC codes and error detecting.

A UPC code on a product in a store looks like:

BAR CODE

digit → x $a_0 a_1 a_2 a_3 a_4 \quad a_5 a_6 a_7 a_8 a_9$ y ← digit

two batches of 5 digits.

So overall, there are 12 digits. The "check digit" is y.

The check digit y is computed by:

$$y = -\left[3(x + a_1 + a_3 + a_5 + a_7 + a_9) + (a_0 + a_2 + a_4 + a_6 + a_8)\right] \bmod 10$$

Note the minus sign!

For example, the Wikipedia article on barcodes shows

BAR CODE

0 36000 29145 2

We can observe the "check digit" at work here:

$$3(0+6+0+2+1+5) + (3+0+0+9+4)$$
$$= 3(14) + 16 = 58$$

Then $-58 \bmod 10 = 2$, which is the final digit of the UPC code.

This error detection scheme is <u>better</u> than the last. It catches <u>all</u> single digit errors. To see this, Suppose that x or a_i is accidentally changed to the number $x+d$ or a_i+d, where $d \neq 0$ is the difference. Then if $i = 0, 2, 4, 6, 8$, this change will change the checksum "y" by $-d \bmod 10 \neq 0$. If $a_i = 1, 3, 5, 7, 9$ or if the change happens to "x" then y changes by $-3d \bmod 10 \neq 0$.

It also catches <u>most</u> transposition errors (89% of them). I.e. if we swap a_i and a_{i+1}, then the checksum y will change 89% of the time. Specifically: swapping two adjacent numbers changes the checksum y whenever the difference of the swapped integers is not 5.

<u>Example</u>: The "IBM check" used on almost

everything, including credit cards:

First, use the notation $x \# y$ to denote the operation "multiply y by x and add the decimal digits of the result".

So e.g. $2 \# 413 = 8 + 2 + 6 = 14$

Then we can check an arbitrary n-digit string for errors by caclulating:

If the string is $a_0 a_1 \cdots a_{n-1}$ we require

$$(a_0 + 2\#a_1 + a_2 + 2\#a_3 + a_4 + \ldots) \bmod 10 = 0.$$

The order of operations is to do all "#" first, then add. An alternative is to think of a_0 as a "check digit" that is given by the equation:

$$a_0 = (-2\#a_1 - a_2 - 2\#a_3 - \ldots) \bmod 10.$$

So if this scheme were being used on bank account numbers, and you had an account number of the form

X 5 5 2 3 9

then X would have to be:

$$X = -(1+0) - 5 - (4) - 3 - (1+8) = -22 \bmod 10 = \underline{\underline{8.}}$$

(with $(1+0)$ from $2\#5$, (4) from $2\#2$, $(1+8)$ from $2\#8$)

We will not discuss why, but this scheme is great: It detects all single-digit errors and 97.7% of transposition errors — unfortunately it detects all transposition errors except for changing 09 to 90, which is 88 out of 90 possible transpositions.

Natural question:

Does there exist a checking scheme that catches all single-digit and transposition errors?

Or, in a more formal language (using our definition of code, etc:

Does there exist a code C consisting of length n codewords $(n > 2)$ using the alphabet $\{0, 1, 2, \ldots, 9\}$ that has the following properties:

(1): For every possible string of length $n-1$, say $a_0 a_1 \cdots a_{n-2}$, there's an $x \in \{0, \ldots, 9\}$ such that $a_0 a_1 \cdots a_{n-2} x \in C$ (so we can encode any length $n-1$ string by adding one check digit)

(2) If $s \in C$ and s' differs from s by a single digit error, then $s' \notin C$

(3) If $s \in C$ and s' differs from s by a single transposition error, then $s' \notin C$.

If such a C exists, then we could say that "C detects all single-digit substitution and transposition errors".

Ans: Such a code exists. It was discovered by Verhoeff in 1969.

Verhoeff's code:

Verhoeff's code not only catches all single digit errors and transpositions, but does very well in catching phonetic errors and other kinds of errors as well!

To explain the IBM check, we needed a new notation of "$x \# y$" as shorthand for a very complicated operation. We will need the same thing here.

Use the notation "$x * y$" to mean the number that you find in row x, column y of the following multiplication table:

*	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	0	6	7	8	9	5
2	2	3	4	0	1	7	8	9	5	6
3	3	4	0	1	2	8	9	5	6	7
4	4	0	1	2	3	9	5	6	7	8
5	5	9	8	7	6	0	4	3	2	1
6	6	5	9	8	7	1	0	4	3	2
7	7	6	5	9	8	2	1	0	4	3
8	8	7	6	5	9	3	2	1	0	4
9	9	8	7	6	5	4	3	2	1	0

So, e.g. $6 * 4 = 7$ and $3 * 5 = 8$. An important thing to notice is that $x * y \neq y * x$ in general! We see things like $6 * 4 = 7$ while $4 * 6 = 5$.

A reasonable question at this point would be: What on Earth is going on in this multiplication table? What is $x * y$?

The table above is actually a description of the symmetries of a pentagon. Explicitly, replace each number i with a function f_i, then in order for composing the f_i's to give the multiplication table above we can describe each as follows:

Suppose we start with a pentagon P:

Then f_0 is the function which "does nothing" to P, ie $f_0(P) = P$, it's the identity.

The functions f_1, f_2, f_3, f_4 are rotations of the pentagon by one, two, three and four "clicks" of $2\pi/5$, respectively. I.e.

$f_1(P) = $ [P rotated one click] $\qquad f_3(P) = $ [P rotated three clicks]

$f_2(P) = $ [P rotated two clicks] $\qquad f_4(P) = $ [P rotated four clicks]

Note that this already makes the top left portion of the multiplication table look sensible. In the table, we see things like:

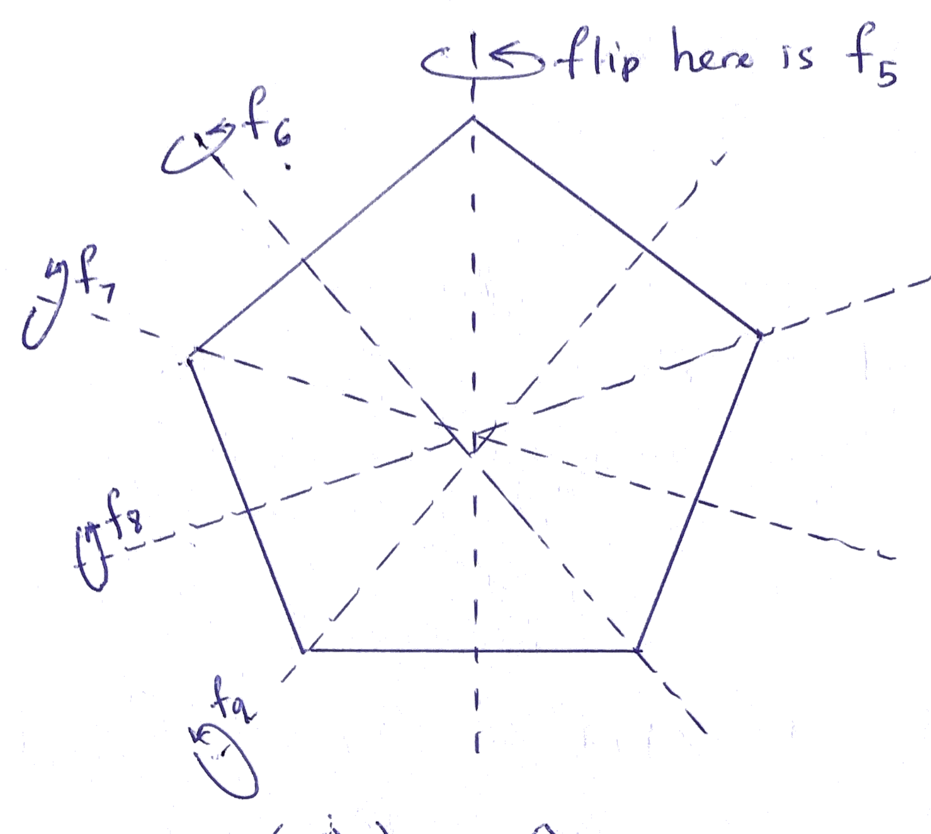
$1 * 1 = 2$ (because rotating one "click", then another, is the same as rotating two)

same as $f_1 \circ f_1 = f_2$

$3 * 3 = 1$ (because rotating six "clicks" is the same as going all the way around once (ie doing nothing) and then doing one more "click").

same as $f_3 \circ f_3 = f_1$

Then the functions f_5, f_6, f_7, f_8, f_9 are "flips" in the axes of symmetry of P:



So, e.g. f_5 (p) = q

and f_6 (P) = a .

From this we can see why certain other entries in the multiplication table work out as they do, e.g.

$1 * 5 = 6$ because

$f_1 \circ f_5$ (p) = f_1 (q) = a

and f_6 (P) = a ← same!

This explanation also explains why $x * y \neq y * x$ in general. It's because <u>order matters</u> when you are thinking about composing flips/rotations of a geometric figure

ASIDE

For the mathematically inclined, what we have just described is the dihedral group D_5.

ASIDE

Back to Verhoeff's code. His idea was to choose a permutation $\sigma : \{0, 1, \ldots, 9\} \longrightarrow \{0, 1, \ldots, 9\}$ and for every decimal string of length n

$$a_0 a_1 a_2 \cdots a_{n-1}, \quad \text{where } a_i \in \{0, \ldots, 9\}$$

define an $(n+1)$st number a_n by

$$a_n = \left[a_0 * \sigma(a_1) * \sigma^2(a_2) * \sigma^3(a_3) * \ldots * \sigma^{n-1}(a_{n-1})\right]^{-1}$$

Then transmit the decimal string

$$a_0 a_1 a_2 \cdots a_n.$$

Note the inverse here! This means we want $a_n * X = 0$, where X is everything in square brackets on the RHS

This code detects all single digit errors, all transposition errors, and (for good choices of σ), many phonetic errors.

In particular Verhoeff found that the best permutation σ for detecting most errors was:

$$\sigma = (1\ 5\ 8\ 9\ 4\ 2\ 7\ 0)(3\ 6)$$

(here we're using disjoint cycle notation).

As an example, if we wanted to compute the check digit (ie a_n) using σ above and starting with

$$a_0 a_1 a_2 a_3 = 2\ 4\ 6\ 8$$

then we would first do:

$$2 * \sigma(4) * \sigma^2(6) * \sigma^3(8)$$
$$= 2 * 2 * 6 * 2$$
$$= 4 * 6 * 2$$
$$= 5 * 2$$
$$= 8$$

Then last we compute 8^{-1} by looking in the multiplication table for x such that $8 * x = 0$. We find $x = 8$. So $a_4 = 8$ and we transmit:

$$2\ 4\ 6\ 8\ 8.$$

↑ check digit.