

# Cryptic Crossword

## Amateur Crypto and Reverse Engineering

### Introduction

Reverse engineering is a special subgenre of computer programming. It's about the closest that I as a programmer get to being a scientist. Gather data, formulate a hypothesis, test, refine, repeat: reverse engineering is basically applying the scientific method to a very, very small knowledge domain. If you've never tried to reverse-engineer a program before, you may be wondering how one goes about such a task. The following essay retraces one of the more colorful reverse-engineering problems that I've pursued.

How all this started was that a friend of mine was contributing to an iOS application for crossword puzzles. This was in the early days of the first iPhone, and they wanted to get their app into the store quickly, before this tiny niche was saturated. The program was working, but it was missing one very small feature — namely that it couldn't unscramble scrambled crossword puzzles.

Let me back up a bit and explain. There is a widely used file format for crossword puzzles, called the .puz file format. (Or at least that's what I called it; I don't know if it has a better name.) The file format was created back in the 1990s by a software company called Literate Software, and they used it in their own application, "Across Lite". that allowed you to both create and solve crossword puzzles in their file format. Apparently they had the good fortune to become a de facto standard, so it's pretty much the main file format to use, if you're a crossword application. As far as I know they never published a spec for their format; however, other people had taken the time to reverse-engineer it and make the description publicly available. My understanding is that the company actually made its money not from the software, but by licensing the right to sell crossword files made with its software. So, even though they may not have officially approved of the reverse engineering, they probably didn't mind, since it would only help cement their file format as a standard. I'm fuzzy on the historical details here, but the upshot is that the only available descriptions of this file format, which is pretty much the standard format for crossword puzzles, were written by people outside of the company.

Fortunately, the file format is pretty straightforward. As you might expect, it's a binary file format. The header includes things like the width and height of the grid, and the number of clues.

```
0000000: FC25 4143 524F 5353 2644 4F57 4E00 02EA  .%ACROSS&DOWN...
0000010: 4BD0 B00C ABE5 B845 312E 3200 0000 0000  K.....E1.2....
0000020: 0000 0000 0000 0000 0000 0000 0F0F 4E00  .....N.
0000030: 0100 0000 4641 5445 2E41 5741 5348 2E41  ....FATE.AWASH.A
0000040: 574F 4C4C 4945 532E 4355 5249 4F2E 5348  WOLLIES.CURIO.SH
0000050: 4F45 454C 4543 544F 5241 5445 2E53 495A  OEELECTORATE.SIZ
0000060: 4541 5353 2E45 5253 542E 4449 4554 4544  EASS.ERST.DIETED
0000070: 2E2E 2E43 454E 542E 484F 5354 4553 5352  ...CENT.HOSTESSR
```

After the header, the file provides all the strings — the first one being the completed grid, which you can see the beginning of in the sample here. Black squares are indicated with ASCII periods. After the answer grid would come the player's working grid, then the list of clues, and then some miscellaneous string data such as the crossword's title and author.

Here's a breakdown of the file header.

Description	Length	Details
File checksum	2	short
Magic	12	The ASCII text: "ACROSS&DOWN\0"
Base checksum	2	short
Masked checksums	8	short[4]
File Version	4	The ASCII text: "1.N\0" (N varies)
Unused	2	set to uninitialized garbage values
Unknown	2	zero unless scrambled

Reserved	12	set to uninitialized garbage values
Width	1	char
Height	1	char
Number of clues	2	short
Bitmask	2	normally set to 0x0001
Bitmask	2	0x0004 = scrambled

I don't know why anybody thought the file needed six different checksums. (Even better, four of the six checksums are masked: their value is XORed with the ASCII characters "ICHEATED". I'm guessing this was meant to discourage people from manually modifying existing files to make their own crossword files without using the software.) Fortunately, someone besides me had already figured this stuff out. There was one aspect of the file format, however, that was missing from the available descriptions.

One feature of the Across Lite program is that once you had created a crossword, you could opt to have it "scrambled". Normally, the completed grid (i.e. the crossword with all of the answers filled in) was stored in the file in plaintext. But that meant that a motivated user could examine the binary file contents to look up the answers (as we just did). Scrambling gave the puzzle author a way to prevent that.

After scrambling a puzzle, the file contents might look like this:

```
0000000: EDFB 4143 524F 5353 2644 4F57 4E00 04EA ..ACROSS&DOWN...
0000010: 4DF5 B00C AB05 B845 312E 3200 0000 8B6B M.....E1.2....k
0000020: 0000 0000 0000 0000 0000 0000 0F0F 4E00 .....N.
0000030: 0100 0400 4846 4A49 2E46 4241 4746 2E50 ....HFJI.FBAGF.P
0000040: 5A44 4C49 4342 5A2E 534D 4549 492E 485A ZDLICBZ.SMEII.HZ
0000050: 4F45 514C 564E 4E50 4E4B 4D56 2E54 414D OEQLVNNPNKMV.TAM
0000060: 5258 5358 2E4F 5557 4F2E 5848 5951 4C56 RXSX.OUWO.XHYQLV
0000070: 2E2E 2E46 5144 4C2E 5952 4659 4F4D 4157 ...FQDL.YRFYOMAW
```

When you ask to have a puzzle scrambled, the application provides you with a key in the form of a four-digit number. (In this particular case, the key is 5274.) If someone later asks to unscramble the puzzle, the Across Lite application will prompt them for the key. If they don't provide the correct key, they can't unscramble the solution.

The reverse-engineered description of the .puz file format had absolutely nothing to say about the nature of scrambled files. And this was my friend's problem. Without that information, their app wouldn't be able to unscramble a file, at all. The user could still work on the crossword, of course: the scrambling doesn't affect the grid's layout, or the clues. But users wouldn't be able to validate their answers.

```
F A T E   A W A S H   A W O L
L I E S   C U R I O   S H O E
E L E C T O R A T E   S I Z E
A S S   E R S T   D I E T E D
      C E N T   H O S T E S S
R E F I T S   J E W I S H
A R I T H   K E R N S   O A F
N I L E   A N N E S   D U P E
D E I   O V E N S   L O S E R
      B O D I L Y   R A C E R S
G L U T E A L   P E P S
R E S I S T   S L U E   S K I
O T T O   R E P U B L I C A N
O M E S   I R A T E   R A M S
```



```
H F J I   F B A G F   P Z D L
I C B Z   S M E I I   H Z O E
Q L V N N P N K M V   T A M R
X S X   O U W O   X H Y Q L V
      F Q D L   Y R F Y O M A
W M I V L G   I E X A G C
Y M U X J   E X Z Q G   H H Z
M H I A   D G F K V   O J I I
S C Y   R C S E A   J X E W Q
      X I A M K Z   X M U H Y P
W G U C F A L   K Q O I
X L R I L V   M U I L   O B U
F C A Z   B W F C J C S D H Z
V W W U   D C A Z R   X G L T
```

M E R E X E N O N A B E T

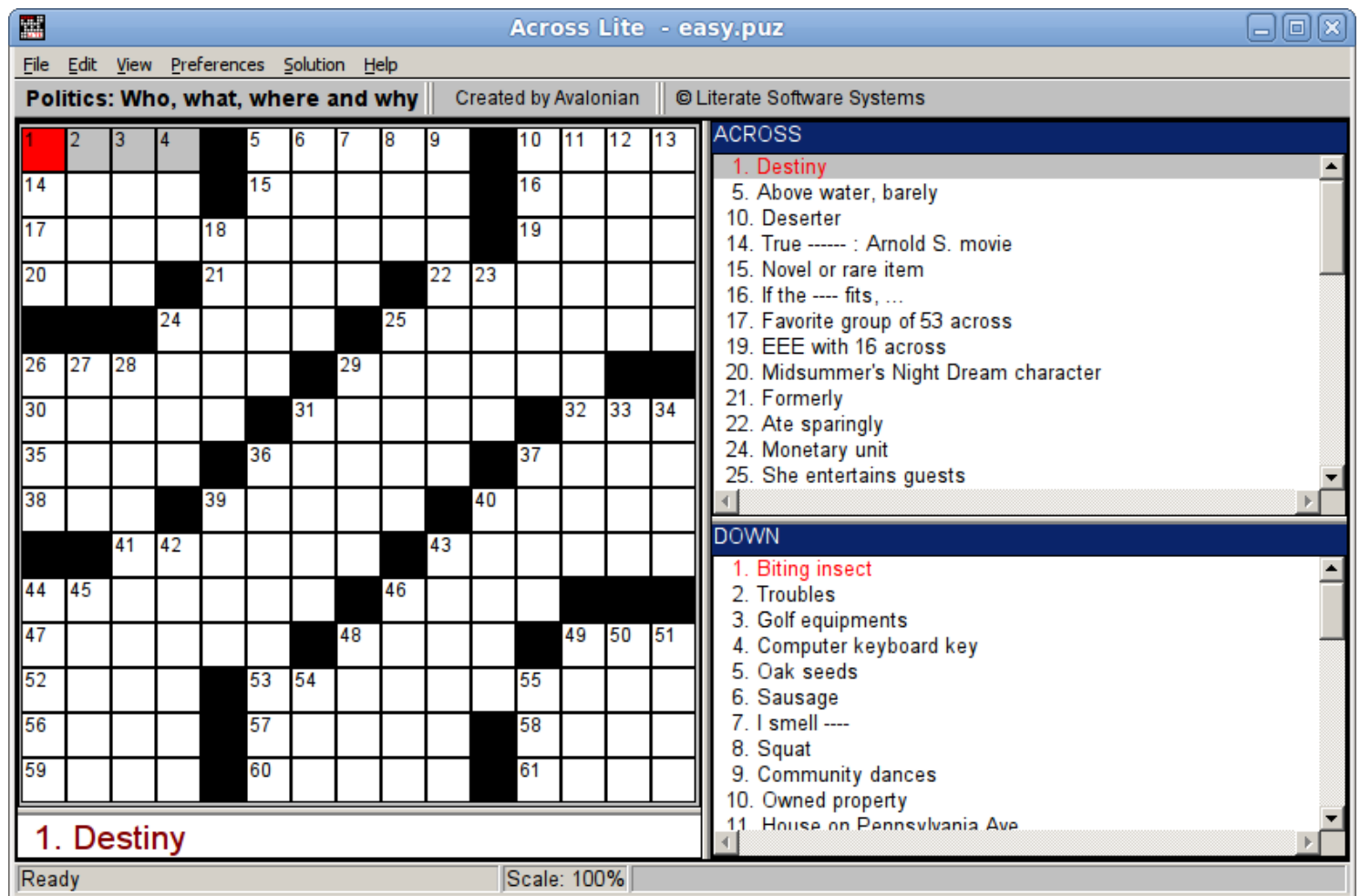
T I C T B Z D N M B H M T

One of the few advantages that crossword apps have over paper crosswords is that at the end, you can check your answer, and the program can tell you whether or not it's correct. Or it can tell you how many letters are wrong, or it can highlight the squares that are wrong, or just one wrong square, etcetera etcetera. But their app couldn't do that with a scrambled file, *even if* the user had the four-digit key!

As it happened, the majority of crossword publishers didn't bother to scramble their files. So they were tempted to just release their app without proper support for that feature. Unfortunately, one of the few publishers that did use scrambled files was the New York Times. (They would publish the scrambled file, and then publish the four-digit key the next day, on the same schedule as the print version published the answer grids.) Having a feature that works on everyone's crosswords except the New York Times is kind of like a wedding band that can play everything except the Wedding March. For a lot of potential users, that would be a deal-breaker.

So, my friend contacted me and described the situation, and asked: Do you think you might be able to reverse-engineer this scrambling algorithm? My response was: maybe. Hard to say, but I'm willing to try. Privately, though, my reaction was THIS IS MY DREAM PROJECT AND THERE IS NO WAY I'M NOT SPENDING ALL AVAILABLE FREE TIME ON THIS.

## Getting Started



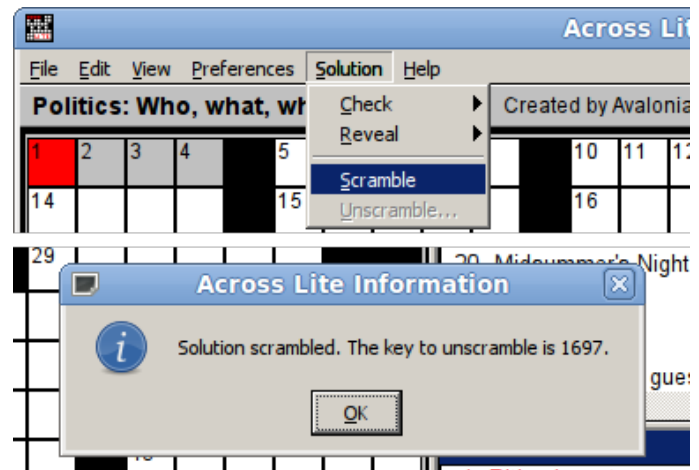
The first step, as it turned out, was just getting the Across Lite program running for myself. To my surprise, they actually had a Linux binary available. Unfortunately, it had a library dependency on a remarkably ancient version of the C++ standard library that pre-dated ANSIfication. After an unsuccessful attempt to track down a copy of this library that would run on a modern kernel, I wound up just using the Windows version of the program, running it under Wine. It was a bit slow to start up, but otherwise ran

fine.

So this is what it looks like when you bring up the Across Lite program with a .puz file. The main focus of this interface is for working on solving the crossword, but this program also provides the ability to scramble the crossword, as you can see from the opened menu.

When you use it, the program selects a four-digit unscrambling key, which it gives to you in a message box. You can then save the scrambled .puz file to disk. The key is not available after you close that message box, so it's up to you to make a note of it.

Once that was working, my next step was to write my own program to create unscrambled .puz files as input. I could have used someone else's code, but since I was already studying the file format, it was easy enough for me to slap together a script that generated a .puz file from some basic input.



With that preparation out of the way, I was ready to actually start work. After brainstorming for a bit on where to begin, here are some of the things that I initially tried:

- Conduct a thorough web search. Though I really wanted to solve this one entirely by myself, I knew that if anyone else had already solved it, I shouldn't be wasting other people's time. So I spent the better part of a day going through various web searches. The closest I came was a forum in which some people were discussing scrambled .puz files. One person there claimed to have cracked the scrambling algorithm, but had since lost the source code. Could be BS, but if not, then at least I knew that it could be done.
- Look for easy-to-calculate invariants between a scrambled grid and its original contents. For example, is there a checksum or a parity value for a grid that remains the same after scrambling? Something like this could reveal possible mechanisms used in the scrambling algorithm. I tried a handful of possibilities; none of them worked.
- Look for a way to control the selection of the four-digit key. The annoying aspect of the scrambling feature is that the program selects the key for you; you don't get to choose what it will be. If I could compare two or more grids that were scrambled with the same key, that might be a good place to start.
- Finally, one approach that may seem obvious which I explicitly did *not* pursue: examining the Across Lite program directly with a debugger and/or disassembler. I took this approach off the list of available options right from the start, due to the project's ultimate goal. The law around reverse engineering is murky (though IANAL and things may well have changed since I last looked into it), but black-box reverse engineering generally appears to be on safer legal ground. By "black box", I mean figuring out how a program works by studying it from the outside, examining only those aspects that the program explicitly makes visible. By contrast, looking at a program's disassembly is a bit too similar to looking at source code, and I imagine this could be a thorny distinction to have to mount a defensive legal case with. Since the ultimate goal was to incorporate this into a program that would be sold for profit in Apple's online store, it only made sense that I should avoid potential gray areas as much as possible.

Something that my friend had noticed was that when we scrambled a puzzle twice in a row, the two keys would be different, but only in the first half. The third and fourth digits were the same. At first I thought that this might be due to scrambling the same grid, but further exploration suggested that it was entirely due to temporal proximity. So naturally, I tried running two instances of the Across Lite program at the same time, and hit Alt-S S on both of them as quickly as possible. In this way I obtained two grids scrambled with the same key.

With this technique, I had my first inroad, a way to start making some actual progress in the investigation. I could now create two crossword grids that differed in some specific way, scramble them both with the same key, and then compare the results, seeing directly how a change in input affected the scrambled output. (In fact, I found I could do up to four grids at once, and still have about a 50% chance of all four being assigned the same key.)

## Initial Discoveries

One of the first things I tested was scrambling a grid of all As and an otherwise identical grid of all Bs.

```

A A A A   A A A A   A A A
A A A A   A A A A   A A A
A A A A A A A A A   A A A
A A A   A A A A A   A A A
A A A A A       A A A A A
A A A A   A A A A A A A A
      A A A A A A A
A A A A A A A A   A A A A
A A A A A       A A A A A
A A A   A A A A A   A A A
A A A   A A A A A A A A A
A A A   A A A A   A A A A
A A A   A A A A   A A A A

```



```

S O W S   S N O U   V A N
N S O T   W X S W   W S X
X K P L Q Y S K O   R W S
S X K   U Q U X P   T W X
U Z V P M           K P U Q V
V U Z V   R V Z V V M R Q
      V V M N U Z Q
N S R W X T R S   S V M R
R N S R S           R T V W M
Y P U   Q X X N S   S X T
T Y P   L P T P X L Q P X
U Q V   N Q U Y   P L Q P
P U Q   A S P Q   Q Q Q Q

```

```

B B B B   B B B B   B B B
B B B B   B B B B   B B B
B B B B B B B B B   B B B
B B B   B B B B B   B B B
B B B B B       B B B B B
B B B B   B B B B B B B B
      B B B B B B B
B B B B B B B B   B B B B
B B B B B       B B B B B
B B B   B B B B B   B B B
B B B   B B B B B B B B B
B B B   B B B B   B B B B
B B B   B B B B   B B B B

```



```

T P X T   T O P V   W B O
O T P U   X Y T X   X T Y
Y L Q M R Z T L P   S X T
T Y L   V R V Y Q   U X Y
V A W Q N           L Q V R W
W V A W   S W A W W N S R
      W W N O V A R
O T S X Y U S T   T W N S
S O T S T           S U W X N
Z Q V   R Y Y O T   T Y U
U Z Q   M Q U Q Y M R Q Y
V R W   O R V Z   Q M R Q
Q V R   B T Q R   R R R R

```

The results were exactly what I had been hoping (though not really expecting): the scrambled all-B grid's contents were exactly one letter ahead of the contents of the scrambled all-A grid (with Z wrapping around to A). This shows that the grid's *contents* were not an input into the scrambling process, except at the very end. The scrambling therefore can only depend on the shape of the grid (i.e. its size and the placement of black squares), and of course on the four-digit key. While that's still a big space to explore, it's nowhere near as big as it would be if each letter in the grid could contribute to the scrambling of the other letters.

This meant that from now on, I could examine nothing but all-A grids, and not have to worry that I might be overlooking some important factor in the scrambling process. This was the first point that I was willing to say aloud that I thought that I would be able to solve it. I still couldn't say how long it would take, but I felt confident in predicting that it was ultimately doable.

The next thing to try, of course, was scrambling two grids of the same size but with different shapes.

```

A A A A   A A A A   A A A
A A A A   A A A A   A A A
A A A A A A A A A   A A A
A A A   A A A A A   A A A
A A A A A       A A A A A
A A A A   A A A A A A A
      A A A A A A A
A A A A A A A A   A A A A
A A A A A       A A A A A
A A A   A A A A A   A A A
A A A   A A A A A A A A A
A A A   A A A A   A A A A
A A A   A A A A   A A A A

```



```

S O W S   S N O U   V A N
N S O T   W X S W   W S X
X K P L Q Y S K O   R W S
S X K   U Q U X P   T W X
U Z V P M       K P U Q V
V U Z V   R V Z V V M R Q
      V V M N U Z Q
N S R W X T R S   S V M R
R N S R S       R T V W M
Y P U   Q X X N S   S X T
T Y P   L P T P X L Q P X
U Q V   N Q U Y   P L Q P
P U Q   A S P Q   Q Q Q Q

```

```

A A A A A A A   A A A A A
A A A A A A A   A A A A A
A A A A A A A   A A A A A
A A A   A A A A   A A A
A A A A   A A A   A A A
      A A A A   A A A A A A A
      A A A A A A A
A A A A A A A   A A A A
A A A   A A A   A A A A
A A A   A A A A   A A A
A A A A A   A A A A A A A
A A A A A   A A A A A A A
A A A A A   A A A A A A A

```



```

S P U Q U W S   Q X W S X
N O W S M Y N   U P R W S
X S O T V Q X   W V T W X
S K P       R S S O   U Q V
U X K L   M U K       M R Q
      Z V P X   V X P Q V M R
      V S T N Z K S
V U Z V Q X R   V T V W
N S R       P X U   L S X M
R N S   L Q T S       Q P T
Y P U W N   U N Z P L Q X
T Y P R A   P P R Q Q Q P
U Q V Q S   O Y S V A N Q

```

The total number of white and black squares are the same in each grid — just their positions are different. As you can see, this time I was not so lucky: the two grids produced completely different encryptions, despite both of them containing all As. Or mostly different — as you might be able to tell, there seem to be a lot of similarities between the two scrambled grids, even though it's not clear that there's an actual pattern present.

To study this further, I created a series of very small grids (experimenting showed that the minimum grid size was twelve letters) with only minor variations in their layout. Once I had succeeded in scrambling all of them with the same key, I soon found the method for how the scrambling algorithm was reading the layout. Can you spot the rule?

A A A A

Z S M O

A A A A  
A A A A



U T P S  
S Y S P

A A A A  
A A A A  
A A A A



Z S Y S  
U S M O  
T P S P

A A A A  
A A A A  
A A A A



Z S T P  
U S Y S  
M O S P

A A A A  
A A A A  
A A A A



Z S T M  
U S Y P  
S O S P

It is simply that the letters are read from the grid vertically, not horizontally. Top to bottom, then left to right. The letters are assembled into a plain old one-dimensional string, scrambled, and then the result is then put back into the grid the same way — top to bottom, left to right. The actual positioning of the black squares is completely unimportant.

(This was further vindicated when I returned to considering the unidentified value in the header. The header contains a run of 8 values, each two bytes long, that are unused — and in fact many files contain random values here. Except, that is, for the second entry: This entry (labeled "Unknown" in the table up above) is always zero for normal .puz files, and non-zero for scrambled files. What its value meant, though, was an open question. I had guessed that it was a checksum representing the unscrambled grid, since Across Lite could tell if you tried to unscramble a grid with an incorrect key. But the number didn't actually match the checksum of the unscrambled grid, so I had set aside that hypothesis for the time being. Now, though, I tried taking the checksum of the unscrambled grid contents rearranged as a one-dimensional string, top-to-bottom left-to-right, and this matched the mysterious header value perfectly.)

So, I now knew that the main scrambling process was determined entirely by the number of letters in the grid and the four-digit key. No other aspect of the grid's contents or arrangement was an important factor. Again, this was a huge decrease in the potential solution space to explore.

The next test, though, was a point where things turned out to be less simple than I had expected. I scrambled a series of grids that differed in size by only one letter:

A A A A A A A A A A A A A A



T Q N N X W W U M V V P T

A A A A A A A A A A A A A A



N X W N Q W O V M V S P P T

A A A A A A A A A A A A A A A



T Q T Q L C X X Q U M P Q X P

Although there are clear hints of shared patterns, the basic fact is that changing the grid size can affect every single letter in the scrambled grid. I had been hoping, now that I understood the ordering of the scrambled grids' contents, that I would find that even the size of the grid wasn't actually an important factor, and that the contents of smaller grids would just prove to be a subset of the larger ones. No such luck.

The next test also showed me that things were still more complicated than I had been expecting. Here are four crosswords, all scrambled with the same key, in which each grid differs from the previous one by only one letter. (Colors indicate the changed letter.)

```

A A A A   A A A A   A A A
A A A A   A A A A   A A A
A A A A A A A A A   A A A
A A A   A A A A A   A A A
A A A A A       A A A A A
A A A A   A A A A A A A A
      A A A A A A A
A A A A A A A A   A A A A
A A A A A       A A A A A
A A A   A A A A A   A A A
A A A   A A A A A A A A A
A A A   A A A A   A A A A
A A A   A A A A   A A A A

```



```

S O W S   S N O U   V A N
N S O T   W X S W   W S X
X K P L Q Y S K O   R W S
S X K   U Q U X P   T W X
U Z V P M       K P U Q V
V U Z V   R V Z V V M R Q
      V V M N U Z Q
N S R W X T R S   S V M R
R N S R S       R T V W M
Y P U   Q X X N S   S X T
T Y P   L P T P X L Q P X
U Q V   N Q U Y   P L Q P
P U Q   A S P Q   Q Q Q Q

```

```

B A A A   A A A A   A A A
A A A A   A A A A   A A A
A A A A A A A A A   A A A
A A A   A A A A A   A A A
A A A A A       A A A A A
A A A A   A A A A A A A A
      A A A A A A A
A A A A A A A A   A A A A
A A A A A       A A A A A
A A A   A A A A A   A A A
A A A   A A A A A A A A A
A A A   A A A A   A A A A
A A A   A A A A   A A A A

```



```

S O X S   S N O U   V A N
N S O T   W X S W   W S X
X K P L Q Y S K O   R W S
S X K   U Q U X P   T W X
U Z V P M       K P U Q V
V U Z V   R V Z V V M R Q
      V V M N U Z Q
N S R W X T R S   S V M R
R N S R S       R T V W M
Y P U   Q X X N S   S X T
T Y P   L P T P X L Q P X
U Q V   N Q U Y   P L Q P
P U Q   A S P Q   Q Q Q Q

```



```

B A A A   A A A A   A A A
B A A A   A A A A   A A A
A A A A A A A A A   A A A
A A A   A A A A A   A A A
A A A A A       A A A A A
A A A A   A A A A A A A A
      A A A A A A A
A A A A A A A A   A A A A
A A A A A       A A A A A
A A A   A A A A A   A A A
A A A   A A A A A A A A A
A A A   A A A A   A A A A
A A A   A A A A   A A A A

```



```

S O X S   S N O U   V A N
N S O T   W X S W   W S X
X K P L Q Y S K O   R W S
S X K   U Q U X P   T W X
U Z V P M       K P U Q V
V U Z W   R V Z V V M R Q
      V V M N U Z Q
N S R W X T R S   S V M R
R N S R S       R T V W M
Y P U   Q X X N S   S X T
T Y P   L P T P X L Q P X
U Q V   N Q U Y   P L Q P
P U Q   A S P Q   Q Q Q Q

```

```

B A A A   A A A A   A A A
B A A A   A A A A   A A A
B A A A A A A A A   A A A
A A A   A A A A A   A A A
A A A A A       A A A A A
A A A A   A A A A A A A A
      A A A A A A A
A A A A A A A A   A A A A
A A A A A       A A A A A
A A A   A A A A A   A A A
A A A   A A A A A A A A A
A A A   A A A A   A A A A
A A A   A A A A   A A A A

```



```

S O X S   S N O U   V A N
N S O T   W X S W   W S X
X K P L Q Z S K O   R W S
S X K   U Q U X P   T W X
U Z V P M       K P U Q V
V U Z W   R V Z V V M R Q
      V V M N U Z Q
N S R W X T R S   S V M R
R N S R S       R T V W M
Y P U   Q X X N S   S X T
T Y P   L P T P X L Q P X
U Q V   N Q U Y   P L Q P
P U Q   A S P Q   Q Q Q Q

```

As you can see, in addition to being encrypted, the letters of the grid are also being reordered. So there was (at least) two steps to the scrambling process. My hunch was that the letters were encrypted first, and then scrambled. But that meant that I would need to be able to undo the scrambling step before I could even start to tackle the decryption. Of course it was entirely possible that I was wrong and the scrambling was done first, or that there were multiple interleaved steps of encryption and reordering. But as usual, we start by assuming that things are simple until they are proven to be otherwise.

## Collecting Data: Script Everything

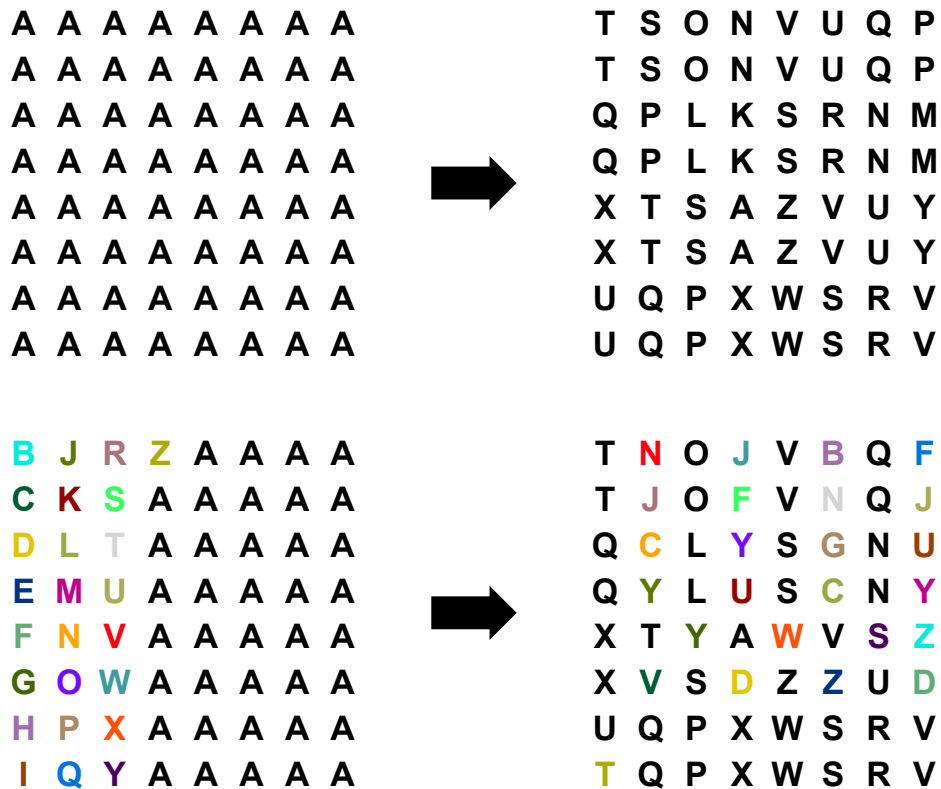
The other thing I found is that trying to obtain four separate files scrambled with the same key was annoying. I had to launch four separate windows, and then quickly select the scrambling menu option in each of them. Even if I didn't make any slips, I found that it would fail almost half the time, leaving some of the files with one key and the rest with a second key. By now I was doing lots of different comparisons, in search of more data, and so I realized that I needed a more reliable technique.

I found a handy command-line program called `xdotool`, which allows one to identify windows by their title text and inject mouse and keyboard events. A simple shell script then allowed me to reliably scramble two or more files in parallel.

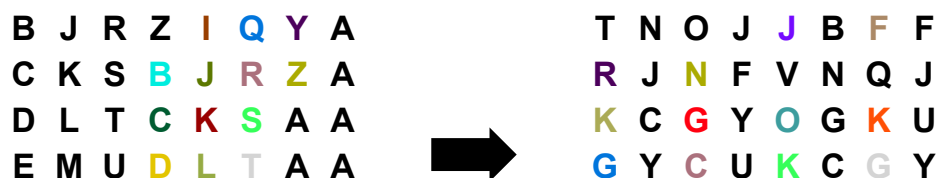
```
for w in `xdotool search --title 'across lite'` ; do
  xdotool windowactivate $w
  sleep 0.1
  xdotool key alt+s s 2> /dev/null
  sleep 0.2
done
```

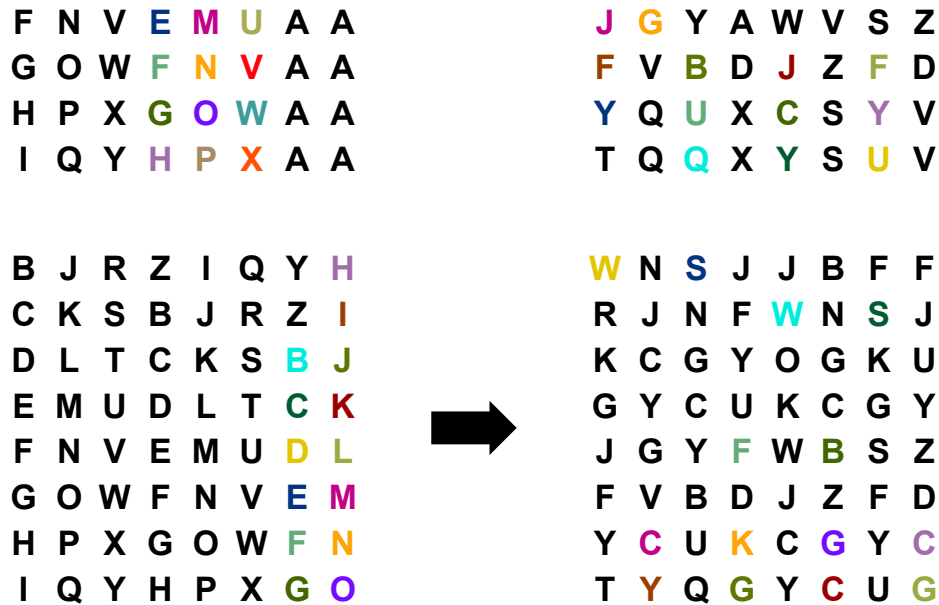
This wasn't perfect — every once in a while I would still get a split of two different keys, but it was much, much rarer. And when it did happen, I just deleted the output files and ran it again.

With the ability to scramble four or more files with the same key easily, I realized I now had a way to fully expose the reordering that had been done on a grid. It works like this. Make one grid with all As. Make another grid with the first 25 letters replaced with B, C, D, on up to Z. Make a third grid with the next 25 letters replaced with B through Z. Make a fourth grid that replaces the next set of 25 letters, and so on until every square is set to something other than A in exactly one file. Scramble them all at once so they have the same key. Thanks to the additive nature of the encryption, you can compare each of the B-through-Z files with the all-As file to identify where each of the B-through-Z letters were reordered to.



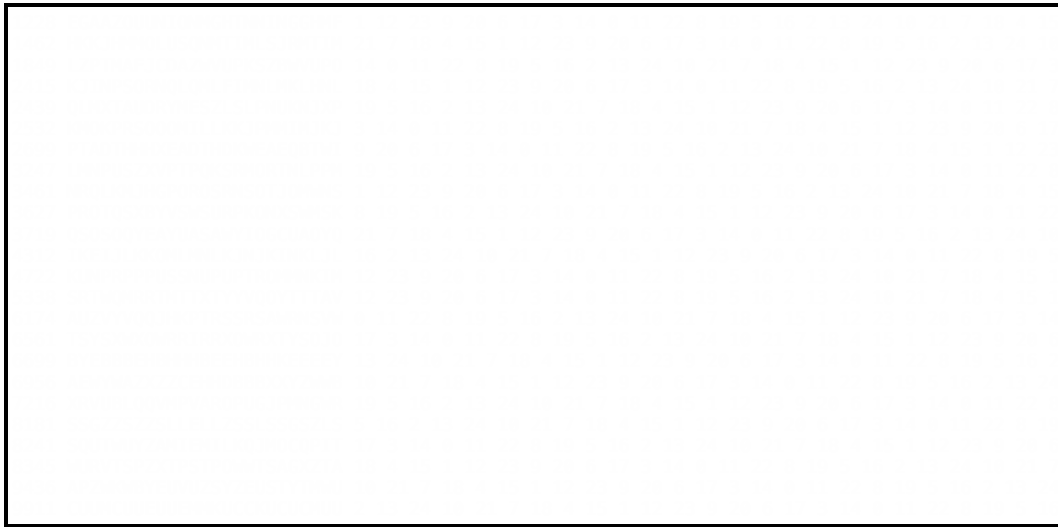
Thus, for example, looking at the two scrambled grids on the right, the Y in the middle of the rightmost column of the top grid is a Z in the second grid. A difference of 1 means that it must match the square that went from A to B in the unscrambled grid. The S at the top of the second column becomes an N in the next grid, for a difference of 21. This corresponds to the square that contains a V (i.e. the letter shifted 21 from A) in the unscrambled grid. And finally the U in the bottom left corner that becomes a T corresponds to the A that was replaced with Z.





By scrambling enough grids simultaneously, I could figure out where every position in the original grid went to in the scrambled grid. (And it was necessary to scramble them simultaneously; I verified that grids assigned different four-digit keys were scrambled completely differently.)

So this meant that I could completely separate out the encryption and reordering steps, and study them independently. I cobbled together another script that would take a set of such files, locate all of the reordered letters, and then spit out the encrypted string in its unscrambled order, along with the scrambled ordering sequence, all in a single line of text. This had become easy enough to do that it was little effort to accumulate multiple examples of the same size grid scrambled with various keys. With this, I could then start looking more widely for patterns.



I stored my collected information in files like this. Each file was devoted to a specific grid size. (This one is of grids of size 25.) This made it easy to write scripts to iterate over the grids of a particular size, to search for specific patterns, or to verify hypothesized invariants, or just to display information for me to stare at. Any time I had a new idea, I could quickly produce a script to try it out.

## The Reordering Process: A Working Hypothesis

It didn't take much looking before some patterns started to become clear in the reordering. I noticed that, typically, a letter in the grid would wind up about 16 positions away from the previous letter (in the original grid), wrapping around from the end to the beginning in the usual fashion. If you examine the numbers on the right in one of the lines in the sample of grids shown above, and pick a 0 on one of the above lines, and then count 16 entries from there, you'll find yourself on the 1. Count 16 again to get to the 2, and so on.

The pattern of intervals — or "strides", as I wound up calling them — wasn't obvious until I started focusing on grids larger than size 32, but once I did it leaped out. In fact, the most common stride was exactly 16, with 17 and 15 coming in a distant second and third place. Looking at more examples showed that the stride could range all the way from 7 to 30, but extreme examples never appeared more than once or twice in a given grid. Almost all of a grid's strides would be in the range of 14–18.

As I looked at a grids of various sizes to see if the pattern continued to hold, I found that the reordering behavior was different for even-sized grids and odd-sized grids. If the grid size was odd, then the stride was exactly 16, every time, reliably. Irregular strides only occurred in even-sized grids. I was surprised to see that the code was making a distinction like that, but in a way it makes a kind of sense. Because 16 is a power of 2, it will always be coprime with a odd number, and therefore a stride of 16 is guaranteed to visit every square in an odd grid. With an even-sized grid, you have to take at least one odd stride in order to avoid revisiting a square before the grid is filled.

In any case, I found it strange that the reordering of even grids was so much more complicated, but at least I could say that I had actually figured out one small piece of the puzzle.

(None of these patterns applied to the initial position, by the way. From what I could see, the first letter could go anywhere in the grid, with equal probability.)

## Collecting Data: No Seriously, Script Everything

As I realized that I needed to collect grids scrambled with as many different keys as possible, I wrote scripts to automate more and more of the collection process. Eventually I had all of it scripted except for one part — namely, retrieving the four-digit key. The key was never output anywhere; it was simply displayed to the user in a message box. There wasn't a way to copy the key to the clipboard or anything like that. So, I would run scripts that would do everything up to the scrambling of the files, then I would read the displayed key and enter it on the terminal command-line (or, if some of the keys failed to match, abort and start over), and then the second set of scripts would grab the scrambled files, extract the grids, determine the reordering and store the information in the appropriate data file. I lived with this for a while, until I was forced to acknowledge that this just wasn't going to work for the amount of data that I wanted to collect. So ... I started investigating OCR programs for Linux.

```
for w in `xdotool search --title 'across lite info' | tac` ; do
  xdotool windowactivate $w
  sleep 0.3
  k=`xwd -id $w -silent | xwdtopnm --quiet | gocr - |
    sed -ne 'y/I/1/;s/ //g;s/.*keytounscrambleis\([1-9]\{4\}\).*\/1/p`
  if test "${key:=$k}" != "$k" ; then
    test $quiet || echo Warning: inconsistent keys \($key vs $k\). >&2
    unset key
    break
  fi
  xdotool key Return 2> /dev/null
  sleep 0.2
done
```

I found a very simple one called `gocr` that worked on the command line. It took a pixmap file as input and returned plain text on standard output. The venerable `xwd` utility has the ability to select the window to capture by window ID, which could be turned into a pixmap via one of the ImageMagick utilities, which `gocr` could then turn back into text.

(Side note: I don't hate GUIs per se. What I hate are GOUIs: graphical-only user interfaces. They are the real blight. The above snippet demonstrates one of the things I love about being a Unix programmer: even when faced with a needless GOUI, there is always some way to turn it back into a text-based interface. And then we are unstoppable.)

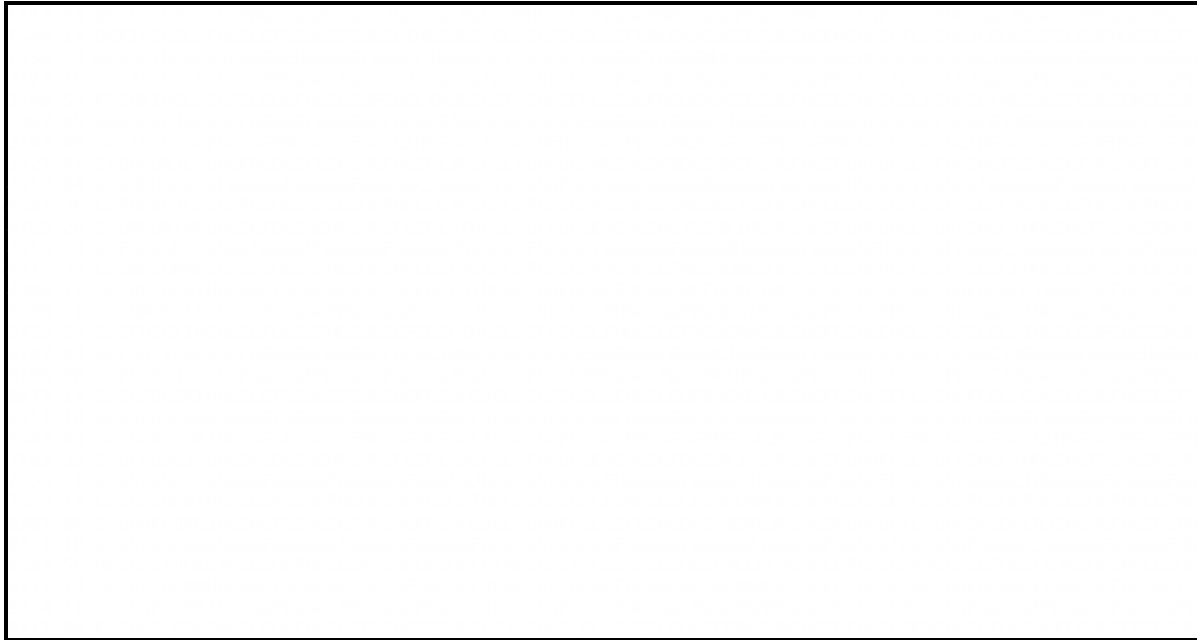
So: I now had a fully automated system that could collect scrambled grids without oversight. I couldn't use the computer for anything else while it was running, of course, since it depended on screenshots and simulated keyboard events. But in the morning

I would set it running in an infinite loop, collecting grids of a specific size, and when I came home from work I would find hundreds of new examples in my data file. A veritable mother lode.

## The Reordering Process: Visualizing Strides

Looking at the rows of raw numbers and hoping to notice a pattern was futile, I quickly realized. So I started thinking about how I could display them in ways that would make patterns show up more clearly. Since I was storing all the data in easy-to-parse text files, it was a simple matter to write a script that could display the contents in different ways.


The first thing I wanted to do was display the strides instead of the absolute positions. In order to pack the information more densely, I displayed the stride value as a single character in base 36 — i.e. A being 10, B being 11, C being 12, and so on. (The four-digit key is shown on the far left, and next to it is displayed the initial position in base 16.)



G, representing 16, is clearly the most common value, as I already knew. But other than that, it's hard to see much of anything. You can tell that the non-G values tend to cluster in various places, but that's about it. Obviously, this wasn't the right way to visualize my data.

My next thought was to use color to highlight values. Part of me disliked having to use colorized output, because it pretty much makes it impossible to pipe the output to anything else: it makes the program an endpoint of any pipeline. But I was desperate, so I did it anyway. This time, instead of displaying the stride's absolute value, I displayed the stride's offset from 16. I chose to display a positive offset in bright blue, and a negative offset in magenta. Zero would be displayed in dark blue, so that the non-zero values would stand out.

Immediately I knew that, whatever problems it introduced, colorizing the output was the right thing to do.



I quickly began noticing patterns in the data. I realized that there was a "seam" of large positive numbers running down the left-hand edge, with values like +7, +8, and +12. And down the middle of the sequence list was a seam of large negative values, usually -8. No other numbers in the stride sequence went outside the range of  $\pm 5$ .

Going down the list, I noticed that the positive number seam slowly moved away from the edge, and that that the negative number seam moved at the exact same rate, so that they were always halfway across the sequence from each other. Closer inspection suddenly brought forth the realization: The position of the positive number seam was equal to the first digit in the key. So, a key of 2358 would mean that the maximum-width stride would be between the second and third square. No exceptions.

Furthermore, I saw that the largest positive values, +11 and +12, only occurred with keys with a second digit of 1. Any other second digit, and the big positive value stayed in the range +7 to +10, with another +4 somewhere nearby. Eventually I realized that the +12 was actually a separate +8 event and a +4 event that had coincided. There was another +4 event near the halfway point, not far away from the -8 event.

It appeared that the placement of these exceptions to the 16-stride rule was partly based on the size of the grid and partly based on the individual digits of the key. By looking at a larger grid size, I was able to see the individual events more clearly:

```

861000000010000002100000001000000421000000010000002100000001000080610000000100000021000000010000002100000001000000010000
80403000000001000000010000000100000040300000000100000001000000010008004300000000100000001000000043000000010000000210000000100
0C2100000001000000210000000100000040300000001000000010000000100008421000000010000002100000001000000421000000010000002100000001000
0C0300000001000000010000000100000040300000001000000010000000100008421000000010000002100000001000000403000000010000000100000001000
08421000000001000000210000000010000004210000000010000002100000000100008043000000001000000010000000421000000010000002100000001000
00506100000001000000210000000100000043000000010000000100000001000080430000000100000001000000043000000010000000100000001000
00800610000000100000021000000010000002100000001000000210000000100080043000000010000000100000001000000421000000010000002100000001
0005402100000000100000021000000001000040210000000100000021000000010008402100000001000000210000000100004003000000010000000100000001
00034030000000100000001000000010000004030000000100000001000000010000804300000001000000010000000421000000010000002100000001
10080043000000010000000210000000100000421000000010000002100000001008004210000000100000021000000010000040300000001000000010000000
0000861000000001000000210000000010000002100000000100000001000086100000000100000021000000010000000430000000100000002100000001
1000806100000001000000201000000010000042100000001000000210000000100080061000000010000002100000001000000021000000010000000210000000
010080421000000010000002100000001000004210000000100000021000000010008006100000001000000210000000100000043000000010000000210000000
0100804030000000100000001000000010000040300000001000000010008004300000001000000010000000210000000100000042100000001000000210000000
00108004030000000100000001000000010000040300000001000000010000000100800403000000010000000100000001000000402100000001000000210000000
00108004030000000100000001000000010000040300000001000000021000000010000000100800403000000010000000100000001000000403000000010000000
00108000421000000010000002100000000100000042100000001000000210000000100800061000000010000002100000001000000430000000100000001000000
00180000403000000001000000021000000010000040210000000100000021000000018000042100000001000000210000000100000403000000010000000100000
10000842100000001000000210000000100000421000000010000002100000001000084030000000100000001000000010000040300000001000000010000000
001008004300000001000000010000000100000430000000100000002100000001008004210000000100000021000000010000042100000001000000210000000
000018004021000000010000002100000001000040030000000100000001000000010800040300000001000000010000000402100000001000000210000
0100008430000000100000001000000010000000100000001000000010000008421000000010000002100000001000000010000004030000000100000001000000
0001000840210000000100000021000000010000040210000000100000021000000010008042100000001000000210000000100000403000000010000000100000
0000010080421000000010000002100000001000004210000000100000021000000010080006100000001000000210000000100000001000000210000000100000020100
000100008070000000100000001000000010000000100000002100000001000080430000000100000001000000010000000100000040300000001000000010000000
00000100804210000000100000021000000010000042100000001000000210000000100080042100000001000000210000000100000421000000010000002100
0000010080403000000010000000100000001000004030000000100000002100000001008004030000000100000001000000010000004030000000100000001000
00000100800610000000100000021000000010000004300000001000000010000000100080043000000010000000100000001000000421000000010000002100
00000100800403000000010000000100000001000004021000000010000002100000001008004030000000100000001000000403000000010000000100000010
0000001800040300000001000000021000000010000402100000001000000210000000180004021000000010000002100000001000040030000000100000001

```

I now could see that there was a consistent pattern of one +8 event, two +4 events, four +2 events, and eight +1 events. These were balanced by an equal number of negative events, offset for maximum distance from the matching positive events, more or less. And the events could overlap each other, in which case they would just add together, like waves passing through each other.

I found myself imagining the underlying mechanism using physical imagery. Specifically, I imagined the four digits of the key as being like four timing wheels that would turn with each step of the reordering process, periodically firing to create the positive and negative changes in the stride. I therefore wound up referring to the events as "firings". The +8 and -8 wheels fire once, the ±4 wheels fire twice, etc.

I labelled the four separate digits of the key **a**, **b**, **c**, and **d**, and set out to create formulas that gave the location of every firing. The +8 firing occurred at position **a**. The -8 firing could be expressed as **a** + **N**/2 (where **N** stood for the size of the grid). The +4 mechanism produced two firings, and it didn't take long to determine that they took place **b**/2 positions after the ±8 firings. Although if **b** was odd, then the second firing would actually be at (**b** + 1)/2. Or, put another way, the value was (**b** + *i*)/2, where *i* was 0 or 1 depending on the firing. (Note that the slash specifically represents integer division, where any remainders are discarded.)

The two -4 firings proved to be a little more complicated, but not intractable. Encouraged by these results, I dove headfirst into the ±2 and ±1 firings. When I was done, I had this:

$$F^{+8} = \mathbf{a}$$

$$F^{-8} = \mathbf{a} + \mathbf{N}/2$$

$$F_i^{+4} = F^{+8} + (\mathbf{b} + i)/2$$

$$F_i^{-4} = F^{+8} + (\mathbf{b} + (\mathbf{N}/2)\%2 + i)/2$$

$$F_i^{+2} = F^{+4} + (\mathbf{c} + (2 \cdot (\mathbf{b}\%2) + i \cdot (2 \cdot ((\mathbf{N}/2)\%2) + 1))\%4)/4$$

$$F_i^{-2} = F^{+4} + (\mathbf{c} + (\mathbf{N}/2)\%4 + (2 \cdot (\mathbf{b}\%2) + i \cdot (2 \cdot ((\mathbf{N}/2)\%2) + 1))\%4)/4$$

$$F_i^{+1} = F^{+2} + (\mathbf{d} + (2 \cdot ((2 \cdot (\mathbf{b}\%2) + \mathbf{c})\%4) + i \cdot (\mathbf{N}\%8 + 1))\%8)/8$$

$$F_i^{-1} = F^{+2} + (\mathbf{d} + (\mathbf{N}/2)\%8 + (2 \cdot ((2 \cdot (\mathbf{b}\%2) + \mathbf{c})\%4) + i \cdot (\mathbf{N}\%8 + 1))\%8)/8$$

As you can see, it got a lot uglier pretty quickly. And I'm skipping over a bunch of time here. There were some failed attempts that only worked some of the time, and there were formulas that were much more complicated than these. But this was the first set of formulas that accurately predicted the reordering of every single grid of size 128 that I had collected so far. I then tried it on a larger grid size, size 140, and again it correctly predicted every single one. I also tried it on a smaller grid size, size 122 — and it got exactly one grid wrong, in one place.

I wanted to cry.

But this wasn't an anomaly. The more I looked at the smaller grids, the more divergences I found from my model. By focusing on the larger grid, I had apparently not been able to see some further set of complications to my model. Not that this model was simple, mind you. But apparently it was still too simple. After all, it did work consistently for larger grids. That couldn't be accidental. So the fact that it worked sometimes but not all the time seemed to indicate that the real answer was more complicated still.

In parallel with this, I had been trying to figure out how the initial position was selected. Since that was just a single number, it was a little easier to explore. I looked for scrambled grids where the keys differed from each other by only one number, and could see some obvious patterns. In fact, I soon realized that, once again, the odd-sized grids followed a relatively simple rule:

$$IP = 15 - 2 \cdot (8a + 4b + 2c + d) \% N$$

But even-sized grids followed a slightly more complicated rule, one that usually produced similar results, but with occasional disruptions. (Sound familiar?)

```
IP = -2 · (8a + 4b + 2c + d)
parity = 1
while IP < 0
    IP += N
    parity = 1 - parity
IP += parity
```

Oh and also when I tried to apply the rule to smaller grids, I found more exceptions where these rules stopped working.

## Interlude: Epicycles

At this point, I felt like I was inventing epicycles. Are you familiar with epicycles? They were circles-upon-circles that earlier astronomers theorized to explain the motion of the sun, moon, and planets — specifically to explain why they periodically sped up and slowed down, even to the point of going backwards at times. I mean, if they were rotating at constant speed in perfect circles, how could there be all these variations? Epicycles were the theory that the celestial bodies were really attached to a secondary circle that rotated around the point of attachment to the main circle. A slight complication to the original picture. It did a great job of explaining the orbit of the sun and moon, for example. But when you applied the same approach to the planets, this simple model isn't quite enough. More epicycles are needed, but also things like deferents and equants. In the end the (known) solar system requires dozens of epicycles to cover all the observed motions.

Despite the complexity, this system did a pretty good job of predicting the motion of the heavenly bodies. But even so, there were hints that it wasn't the right answer. For example, all the planets have completely independent motions, yet why do the orbits of the two fastest planets, Mercury and Venus, never take them very far away from the position of the sun? Under the epicycle model, this could only be considered a coincidence. Like the Ptolemaic astronomer, I found myself unable to explain certain features of my stride equations. You'll notice that certain terms, like  $2 \cdot (b\%2)$ , crop up in several places, while most terms only appear once. That suggests that there's something in there that hasn't been factored out correctly. But there's something else that's much more troubling. Would you guess, just from looking at these equations and knowing how they're used to guide the reordering step, that it would always assign exactly one letter to every position in the output, never accidentally trying to put two letters into the same position? I mean, it just so happens to be true — but it's far from obvious. Usually when a sensible person writes something like a reordering algorithm, a necessary feature like that will usually be a clear feature of the code.

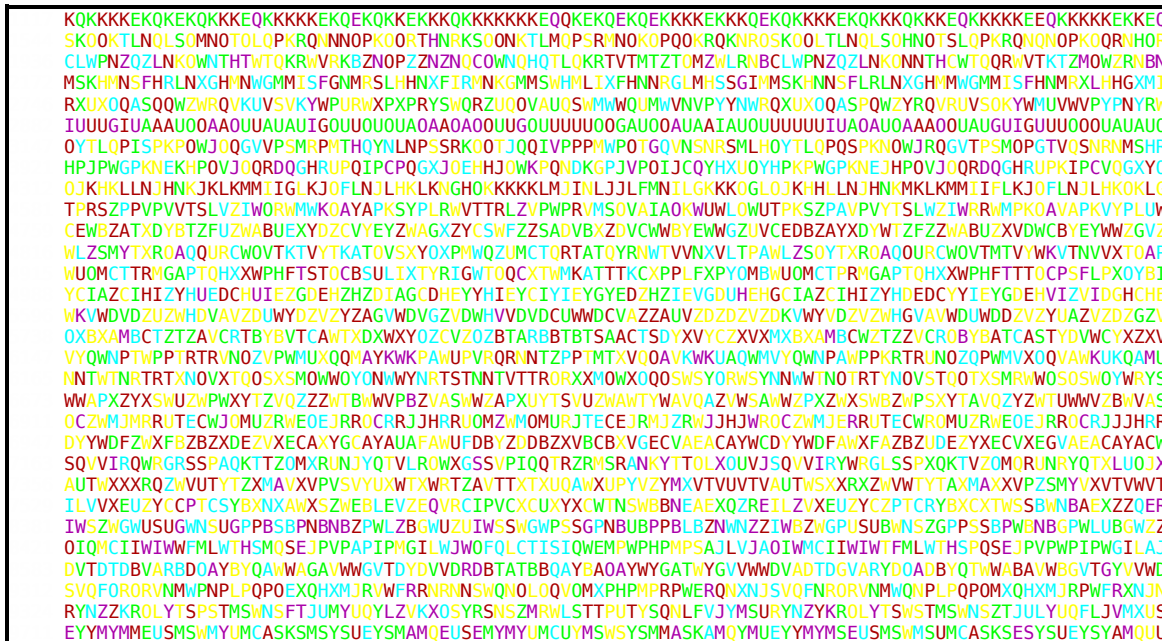
Things like this convinced me that what I had created was an effective system of epicycles. But then the next question was: how do I go from epicycles to ellipses? If this system of stride formulas isn't the right answer, then how do I get from here to a better answer?

Of course, during this time I wasn't just looking at the reordering process. There was also the actual encryption. I worked on both in parallel, so that when I got frustrated with one I would switch over to the other. So, let's set the reordering problem aside for now, rewind a bit, and cover the other half of the scrambling process.



## The Encryption Process: Various Visualizations

Remembering how color had helped me to see patterns in the reordering process, I started by writing scripts to colorize the encrypted grids in various ways, looking for an entry point. One of the first I tried was to color each letter based on how distant it was from the previous letter in grid.



Red for closest, violet for most distant. Since the plaintext letter is always A, any variance or pattern from letter to letter should be entirely due to the encryption process. Note, however, that this display is showing the grids as they appeared in the scrambled output file. In contrast, here is the same colorization filter applied to the encrypted grids after undoing the reordering step:



As you can see, this output has more suggestion of patterns being present, in the form of short columns of colors as patterns align across grids. I had originally guessed, just as a hunch, that the encryption step came first, followed by the reordering step. This output seemed to confirm that my hunch was correct.

I also couldn't help noticing that a few of the entries had long sections of red, and that they happened to be entries with repeated digits in the key. So I tried an experiment where instead of sorting my grids by the key's numerical order, I sorted it so that keys with digits that were close together came earlier than keys with widely varying digits.



This output confirmed that general pattern, with red colors at the top slowly shifting across the spectrum into more blue and violet colors at the bottom.

Perhaps the most striking example was a single grid I had managed to find in which the key was 8888 — a single repeated digit. The scrambled grid for this key was all Gs. That suggested a number of possible underlying mechanisms for the encryption process, which unfortunately proved not to hold up. For example, you might naively expect that a key of three repeated digits might tend to have three out of every four letters be identical, but this was not the case. (Although such grids did tend to have a lot of repeated letters, demonstrating that this idea likely wasn't entirely wrong, just oversimplified.)

Still, you can see that there does seem to be an erratically recurring rhythm of fours in the grids. With the key being four digits long, there was definitely an attraction to the idea of the encryption process being based on a cycle of four steps, rotating through the digits of the key in some way or another. I found myself imagining a series of rotors, sort of like parts of the Enigma machine, for those of you who've read about that.

So the next colorizing display I tried out specifically indicated when a sequence of four letters was repeated.



In this display, the bland cyan is the background color. Green shows when a sequence of four letters is repeated once. Magenta indicates a "three-peat", i.e. another repetition, followed by yellow, then red, and so on. Once again, we see that there are vague patterns occurring across grids, and roughly moving towards the right as the digits in the key get larger.

However, what eventually caught my eye in this display was the stumpy little column of green down in the lower left. It surprised me because, unlike the other patterns, it was exactly vertical, straight up and down. And it was so close to the left edge, showing that these grids had a repeating pattern of four right from the start.

Upon studying this anomaly more closely, I realized that the column began right when the leftmost digit of the key increased to 8. Looking at the first eight letters of grids with a key starting with 7 revealed that they repeated three of the first four letters, but then the last letter invariably diverged. And likewise, keys starting with 6 suggested that they had the repeating-fours pattern as well, but interrupted after the sixth letter in the grid. Even more exciting, I noticed that when the key started with 4, then first four letters of the grid was repeated at the *end* of the grid.

Based on these observations, I theorized that this "quartet" encryption pattern actually ran all the way through the grid, from beginning to end. Mostly it was obscured with another layer of encryption, but it was visible along the leftmost edge, as long as the first digit of the key was greater than or equal to 4. I further theorized that if I isolated the underlying quartet pattern, I could subtract it out of the encrypted grid, and hopefully what remained would be simplified as a result. So instead of thinking of rotor wheels, I began to imagine the encryption process as a set of layers, each one applied in turn atop the other.

If I could completely associate the pattern of the underlying quartet to the four-digit key (and possibly the grid size), that would be a real milestone. In order to do that, I needed more data.

## Interlude: Epicycles, Revisited

A while ago I drew comparisons with the my reordering model and the Ptolemaic system of planetary orbits, and I asked rhetorically, how does one go from epicycles to ellipses? I mentioned some of the facts that hinted at the incompleteness of the older model — Mercury and Venus, and the fixed stars — but really those were hints that the geocentric model was wrong. Those hints led to the heliocentric model, but heliocentrism alone doesn't actually get rid of epicycles. You still need them to match the orbits of the planets, because you're still using circles for everything. In fact when Nicolaus Copernicus advanced his heliocentric system, he required more epicycles, not fewer, because he used them instead of equants, which geocentrism made necessary. Copernicus advocated for heliocentrism in part because it explained the aforementioned coincidences, but mainly because it allowed him to get rid of the equant points, which were hard to calculate with and, in Copernicus's opinion, aesthetically ugly. They sullied the perfection of the circles.

But if you're looking to ditch epicycles — if you want to make the leap to elliptical orbits, as Johannes Kepler did — those facts aren't enough. So: what did Kepler have that Copernicus didn't, that led him to leave circles behind completely, and consider a completely different model in search of simplicity? Probably more than anything, he had Tycho Brahe's data. Tycho Brahe compiled some of the most precise and thorough astronomical observations made before the invention of the telescope. It was by studying his numbers, and the story that they told, that Kepler could see the flaws in even the best epicyclical model, and was forced to abandon that system entirely, and to finally consider the possibility of orbits in the shape of imperfect, lopsided ellipses.

## Collecting Data: Time For A New Approach

Like Kepler, I needed more data. In particular, I was now occasionally wanting to see the grid for a specific key. For example, I had noticed that most of the grids where my reordering algorithm failed were ones that had 1 or 2 for the first digit and 8 or 9 for the remaining digits. But I only had a few example of these. I needed more of them to see if this pattern held, or if it was just a coincidence. In order to do that, I either had to be extremely patient, or I had to figure out how to control which four-digit key was selected. Knowing that it was somehow related to the current time, I sat down and started doing some experiments. And in very little time I worked out the exact process by which the key was selected:

1. Take the current Unix time as a decimal number.
2. Read the digits from right to left.
3. Skip over the first (lowest) digit, and any zeros.
4. Stop once four digits have been obtained.

For example:

```
time(0) = 1 2 1 8 8 6 8 0 9 5  →  9 8 6 8
```



So now I knew how to pick a current time in order to get Across Lite to use a specific key. I could have set the computer clock just before scrambling a grid, but of course there's a much easier way. You may already be familiar with Unix's `$LD_PRELOAD` environment variable. By providing a path to a shared-object library in this variable, you can force it to be loaded ahead of any other dynamic libraries when a program is run — even before system libraries like `libc`. It's sometimes used to replace `malloc()` et al. with debugging versions, but a very common use, at least at one time, was to change the system time for a single program, typically demo programs that were set to expire at the end of some trial period. I did some poking around online and quickly found sample code that showed how to create a library that provided a replacement `gettimeofday()` system call. With the help of `$LD_PRELOAD`, I set it up so that this library was feeding the current time to my wine process, and thus by extension to the Across Lite windows program.

(Interesting aside: My first version of this library simply always returned the same time every time it was invoked, but to my surprise this caused the Across Lite program to malfunction, producing a negative value for the scrambling key. So I then tried just having my library set the initial time, but then count time normally afterwards. However, wine could sometimes be slow to initialize, and I found that starting too many wine processes simultaneously could cause one of them to crash, presumably due to some race-condition bug. So finally I modified my library so that it advanced the clock one microsecond every time `gettimeofday()` was invoked, and that worked perfectly. I also discovered that just starting a single wine process running Across Lite and then immediately shutting it down caused `gettimeofday()` to be invoked over 3000 times.)

Since I was already making changes to how I collected my data, I took this opportunity to do a major overhaul on the whole process. As proud as I was of my Rube Goldberg solution of cobbled-together scripts and obscure utilities, it really was a complicated and brittle solution. I have some experience with Windows code, and I knew that a Windows program would be able to handle all of the interactions with the Across Lite application, and it would be able to do so in a much more direct manner, for example not having to use OCR in order to read the text in a message box.

```
HWND wnd = NULL;
int counter = 0;
for (;;) {
    wnd = FindWindowEx(NULL, wnd, AL_WINDOW_CLASS, NULL);
    if (!wnd)
        break;
    char buf[AL_WINDOW_TITLE_PREFIX_LEN + 1];
```

```

GetWindowText(wnd, buf, sizeof buf);
if (strcmp(buf, AL_WINDOW_TITLE_PREFIX))
    continue;
if (!PostMessage(wnd, WM_COMMAND, AL_IDM_SCRAMBLE, 0)) {
    warn("PostMessage to window %04X: %s", wnd, GetLastError());
    ++failures;
    break;
}
++counter;
}

```

Here's a brief sample from the C program I wrote, which selects the "scramble" menu command on all running instances of Across Lite. Not only was this faster and more reliable, it was also much simpler.

Armed with my Windows program and my time-setting script, I set up a new data collection process. Now, instead of just fishing for grids at random, my scripts specifically gathered grids for every key in order (skipping over keys that were already in my collection). And this turned out to be extremely important. Because once I had long stretches of adjacent keys, I finally was able to see the patterns in explicit detail.

## The Encryption Process: Layers Upon Layers

Here's an example of what some of the new data looked like:

The image shows a dense block of text where each character is color-coded. The colors represent the relationship between the current letter and the letter above it in the previous grid. Cyan indicates a letter that's the same as the one above it. Green marks a letter that's exactly one after the one above it. Yellow if it jumps by two, magenta if jumps by three. Dark blue if it appears to be unrelated. The grids making those dark blue dividing lines correspond, as you can see, to the points where the rightmost digits rolls back from 9 to 1, and the second-to-last digit increments.

Since I now had every single key value in sequence, I introduced a different coloring process. Here, the colors of each letter indicates its relationship with the same letter in the previous grid. Cyan indicates a letter that's the same as the one above it. Green marks a letter that's exactly one after the one above it. Yellow if it jumps by two, magenta if jumps by three. Dark blue if it appears to be unrelated. The grids making those dark blue dividing lines correspond, as you can see, to the points where the rightmost digits rolls back from 9 to 1, and the second-to-last digit increments.

Look at this. It's abundantly clear here, that when only the rightmost digit is changing, you can easily see its effect. And the effect remains consistent as long as the other digits are consistent. (The effect is not the same regardless of the other digits, though. It's not quite that simple.)



Anyway, the original motivation for collecting such complete data was trying to study the "underlying quartet" pattern. With this extra data, all it took was a single all-nighter, and I managed to work out the formulas that completely described the underlying quartet for all grid sizes.

$$\begin{aligned} &\text{if } N \% 4 == 0 \text{ or } 1: \\ & z = a \cdot (2 - b \% 2 - c \% 2) + c \cdot (0 + b \% 2 + c \% 2) \\ &\text{else if } N \% 4 == 2 \text{ or } 3: \\ & z = a \cdot (0 + b \% 2 + c \% 2) + c \cdot (2 - b \% 2 - c \% 2) \end{aligned}$$

$$\begin{aligned} &\text{then, if } N \% 4 == 0 \text{ or } 3: \\ & q_0 = z + a \cdot (2 - a \% 2) + c \cdot (0 + a \% 2) \\ & q_1 = z + a \cdot (0 + a \% 2) + c \cdot (1 - a \% 2) + b \\ & q_2 = z + a \cdot (1 - a \% 2) + c \cdot (1 + a \% 2) \\ & q_3 = z + a \cdot (0 + a \% 2) + c \cdot (1 - a \% 2) + d \\ &\text{else if } N \% 4 == 1 \text{ or } 2: \\ & q_0 = z + a \cdot (1 + a \% 2) + c \cdot (1 - a \% 2) \\ & q_1 = z + a \cdot (1 - a \% 2) + c \cdot (0 + a \% 2) + b \\ & q_2 = z + a \cdot (0 + a \% 2) + c \cdot (2 - a \% 2) \\ & q_3 = z + a \cdot (1 - a \% 2) + c \cdot (0 + a \% 2) + d \end{aligned}$$

Plugging in the grid size for  $N$  and the four digits of the key for  $a$   $b$   $c$   $d$ , this spits out four numbers which consistently matched the underlying quartet for all grids that I could examine.

With this in hand, I wrote a filter script that would compute the underlying quartet for a key and then subtract it from the grid. This would then let me see the remaining steps of the encryption looked like in isolation:

When I first saw this, I felt pretty good about myself. The resulting encrypted grids were clearly simpler with the underlying quartet removed. The remaining patterns appeared to be larger and slower to change, which made them easier to see and isolate. So I picked another pattern, came up with a set of formula that described it in isolation, and wrote another filter script to subtract it as well.

Once again, the remaining patterns in the encrypted grids were blockier and easier to see. Hot on these successes, I continued to identify patterns, describe them, and subtract them out. Before long, there was nothing left but some odd patches:

And when I subtracted those out, all that was left was a sea of As that represented the original, unencrypted grids:

I now had a sequence of steps that described the encryption process as the union of five or six simpler processes. I thought of them as layers, each one overlaid on top of the others. Like the reordering formulas, it seemed a little too complicated to be the actual process, but nonetheless it worked.

Except, that is, when it didn't:



Odd-sized grids proved to be different in ways that I couldn't just capture as a minor variation (as I had with the formulas for the underlying quartet layer), and I wound up identifying three more layers that only appeared on odd grids. Worse yet, even after adding these, I found that smaller-sized grids turned up more exceptions, just as it had with my description of the reordering algorithm.

Let me switch gears again to the reordering process. All of my original attempts to describe the reordering were done with scattered data points. Now that I had a full set of data for several different grid sizes, I had a different view of the stride patterns.

<http://www.muppetlabs.com/~breadbox/txt/acre.html> Page 25 of 39

Comparing with some of the earlier images of the where the strides appeared, it was now easier to find commonalities in the places where my current algorithm got the placement of the firings wrong. Notice, for example, the four lines in the above output where a +1 firing occurs at the very first position (in bright blue). An unusual thing, and not something that my equations currently allowed for at all. What's really happening, though, is that the rightmost +1 firing is wrapping around to the front again. If you look closely, you can see that +1 firing disappearing off the right edge and reappearing on the left for those values. Notice that those grids are ones with high values for **c** and **d**. And, notice that those grids also have very different values for the initial position. (The initial position is indicated by the two-digit hex value immediately after the key.) Now what's interesting here is that my current equation for calculating the initial position correctly predicted the wraparound here, but its values were actually off by one. If you look at the initial position values immediately before the wraparound, you can see that they diminish by two as the value of **d** increases. But then, at the point of wraparound the value suddenly changes from odd to even, as the value diminishes by three instead of two. Why? I couldn't say. Once again, this only happened for even grids; on odd grids, a simple modulus operations captured the wraparound. But I did my best to capture this behavior in another equation and added it as a further refinement to my reordering algorithm. Some random spot checking suggested that I now could correctly predict the reordering of all grids that were larger than 55 or so. Below that, errors started to appear, getting more numerous as the grid size got smaller and smaller. (At the minimum grid size of 12, almost every grid had at least one error.)

So here is where things stood for me. On the one hand, I had a complete unscrambling algorithm, covering both encryption and reordering, that worked correctly for something like 90% of all possible grids, including almost all real-word crosswords. (Even a small crossword puzzle typically has over 100 white squares.) On the other hand, the algorithm was ridiculously complicated, spread out over several scripts, and it was obviously not correct, since its errors got worse and worse as the grid size shrank. There was no way that the algorithm I had, for all of its successes, looked anything the real unscrambling algorithm.

I felt strongly that I had come to a turning point. The fact that both my reordering and my encryption algorithms failed as the grid decreased in size felt too compelling to be a coincidence. I felt that I was at the point where I had to turn my back on epicycles and embrace ellipses. But I still had no idea what that meant; I had no idea what my ellipses were. But my gut told me that collecting more data wasn't going to help any more. I was at the point where I had to start making mental leaps.

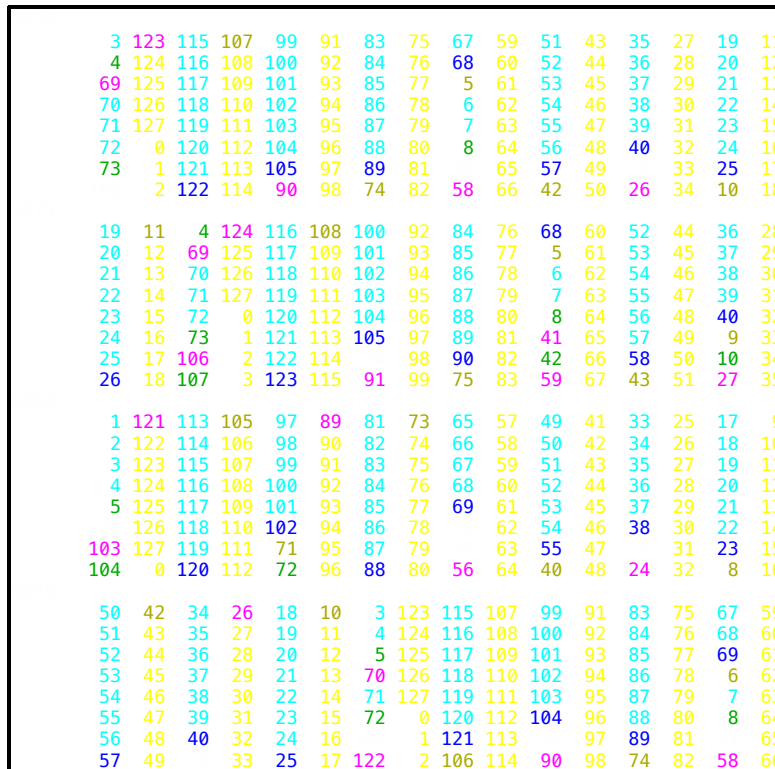
Part of the problem, of course, was that I had collected all these scripts that provided visualizations based on my current approach. All the ways I had of looking at my data were colored by my current model (quite literally, in most cases). So I started forcing myself to get away from my computer and think about alternate ways to model the behavior.

I spent some time floundering like this, toying with alternative models. Despite the effort I put into it, I only came up with one idea that appealed to me at all.

This idea was to view the grid not as a single row, but as a sequence of rows, each 16 positions long. With the rows stacked up from top to bottom, this would create a set of 16 columns. This was attractive for the reordering process, because it meant that most of the letters would be filled in by simply going down the columns. My so-called firings would then appear as points in which the sequence jumped from one column to a different column. I hoped that looking at it this way might cause the firings (or rather, the column changes) to fall into a more visible pattern.

## The Reordering Process: Sixteen Columns

So, after a period of time in which I failed to come up with a better idea for a new model, I sat down and started writing some new scripts to display my collected data in this way. Right off I found it frustrating because, since I was using a two-dimensional shape to display the grids, I couldn't fit nearly as many grids into a single window. But after looking at them for a while, I began to see some promising avenues.



Since the strides are now displayed spatially instead of numerically, the contents of the grid are just indicated by the numbers 0 through 127 (since this shows grids of size 128). The brightly-colored numbers show the column switches, where the sequence switches from one column to another. Beyond that the columns are color-coded for even and odd columns, just as a minor visual aid.

Right away I started noticing patterns in this new visualization. The +1 and -1 jumps (i.e. the firings in the previous model) only occur at the end of the columns, just before jumping back to the top of another column. Likewise, the +2 and -2 jumps occurred close to the end of columns, though almost never at the very end. I also noticed that the places where my old model was still mispredicting a +1 or -1 firing were, in the new model, where the column jump caused a wraparound from the end of the buffer to the front. Of course, these patterns were still pretty vague, but the new model seemed to have promise.

At some point I was looking at the initial position. As mentioned earlier, the initial position tended to decrease by two as the last digit in the key increased by one, with exceptions such as when it wrapped around from the front to the back.

6	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14
7	127	119	111	103	95	87	79	71	63	55	47	39	31	23	15
72	0	120	112	104	96	88	80	8	64	56	48	40	32	24	16
73	1	121	113	105	97	89	81	9	65	57	49	41	33	25	17
74	2	122	114	106	98	90	82	10	66	58	50	42	34	26	18
75	3	123	115	107	99	91	83	11	67	59	51	43	35	27	19
76	4	124	116	108	100	92	84		68	60	52		36	28	20
5	125	117	93	101	77	85	61	69	45	53	29	37	13	21	
118	110	102	94	86	78	70	62	54	46	38	30	22	14	7	127
119	111	103	95	87	79	71	63	55	47	39	31	23	15	72	0
120	112	104	96	88	80	8	64	56	48	40	32	24	16	73	1
121	113	105	97	89	81	9	65	57	49	41	33	25	17	74	2
122	114	106	98	90	82	10	66	58	50	42	34	26	18	75	3
123	115	107	99	91	83	11	67	59	51	43	35	27	19	76	4
124	116	108	100	92	84		68	60	52		36	28	20		5
125	117	93	101	77	85	61	69	45	53	29	37	13	21	126	6
102	94	86	78	70	62	54	46	38	30	22	14	7	127	119	111
103	95	87	79	71	63	55	47	39	31	23	15	72	0	120	112
104	96	88	80	8	64	56	48	40	32	24	16	73	1	121	113
105	97	89	81	9	65	57	49	41	33	25	17	74	2	122	114
106	98	90	82	10	66	58	50	42	34	26	18	75	3	123	115
107	99	91	83	11	67	59	51	43	35	27	19	76	4	124	116
108	100	92	84		68	60	52		36	28	20		5	125	117
93	101	77	85	61	69	45	53	29	37	13	21	126	6	110	118
86	78	70	62	54	46	38	30	22	14	7	127	119	111	103	95
87	79	71	63	55	47	39	31	23	15	72	0	120	112	104	96
88	80	8	64	56	48	40	32	24	16	73	1	121	113	105	97
89	81	9	65	57	49	41	33	25	17	74	2	122	114	106	98
90	82	10	66	58	50	42	34	26	18	75	3	123	115	107	99
91	83	11	67	59	51	43	35	27	19	76	4	124	116	108	100
92	84		68	60	52		36	28	20		5	125	117	93	101
77	85	61	69	45	53	29	37	13	21	126	6	110	118	94	102

As I mentioned, it was harder to see patterns in consecutive grids in the new display, but eventually I realized: It wasn't just the initial position that was moving back two places as the key increased. Actually, the *entire grid* was moving back two places, as a solid block. The only exception were the two positions that fell off the very front and wrapped around to the end: they had swapped positions with each other. I looked more closely and saw that, yes, this was a consistent pattern. As positions wrapped around from the front to the back in pairs, their order would be reversed. Spot-checking throughout my collected data sets showed that this behavior was consistent regardless of the grid size. Consistent among even grids, that is: odd grids didn't reverse the pair order when wrapping around, but otherwise showed the same shifting pattern.

This may sound too obvious to be worth pointing out explicitly, but I'm going to anyway: This pattern of left-shifting constituted the full contribution of the key's fourth digit to the reordering process. It's not used any further. (Because it if was, it would be visible.) So we can factor it out, just like we factored out the various encryption layers, and hopefully what's left will be simplified. In this case, there's an obvious way to factor the fourth digit out, and that is to extrapolate backwards to a zero digit. Real keys never contains a zero digit, but as we right-shift to go from a digit of three to two to one, one more shift will give us the grid for a zero digit.

So I wrote a script to generate a full set of 729 grids that had a zero for the fourth digit, for a handful of grid sizes, and examined them.

36	28	20	12	5	125	117	109	101	93	85	77	69	61	53	45
37	29	21	13	6	126	118	110	102	94	86	78	70	62	54	46
38	30	22	14	7	127	119	111	103	95	87	79	71	63	55	47
39	31	23	15	72	0	120	112	104	96	88	80	8	64	56	48
40	32	24	16	73	1	121	113	105	97	89	81	9	65	57	49
41	33	25	17	74	2	122	114	106	98	90	82	10	66	58	50
42	34	26	18	75	3	123	115	107	99	91	83	11	67	59	51
43	35	27	19	76	4	124	116	108	100	92	84		68	60	52
5	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13
6	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14
7	127	119	111	103	95	87	79	71	63	55	47	39	31	23	15
72	0	120	112	104	96	88	80	8	64	56	48	40	32	24	16
73	1	121	113	105	97	89	81	9	65	57	49	41	33	25	17
74	2	122	114	106	98	90	82	10	66	58	50	42	34	26	18
75	3	123	115	107	99	91	83	11	67	59	51	43	35	27	19
76	4	124	116	108	100	92	84		68	60	52		36	28	20
101	93	85	77	69	61	53	45	37	29	21	13	6	126	118	110
102	94	86	78	70	62	54	46	38	30	22	14	7	127	119	111
103	95	87	79	71	63	55	47	39	31	23	15	72	0	120	112
104	96	88	80	8	64	56	48	40	32	24	16	73	1	121	113
105	97	89	81	9	65	57	49	41	33	25	17	74	2	122	114
106	98	90	82	10	66	58	50	42	34	26	18	75	3	123	115
107	99	91	83	11	67	59	51	43	35	27	19	76	4	124	116
108	100	92	84		68	60	52		36	28	20		5	125	117
69	61	53	45	37	29	21	13	6	126	118	110	102	94	86	78
70	62	54	46	38	30	22	14	7	127	119	111	103	95	87	79
71	63	55	47	39	31	23	15	72	0	120	112	104	96	88	80
8	64	56	48	40	32	24	16	73	1	121	113	105	97	89	81
9	65	57	49	41	33	25	17	74	2	122	114	106	98	90	82
10	66	58	50	42	34	26	18	75	3	123	115	107	99	91	83
11	67	59	51	43	35	27	19	76	4	124	116	108	100	92	84
	68	60	52		36	28	20		5	125	117		101	93	85

It's now easy to see the influence of the third digit on the reordering process. It proves to be similar in nature, but now four positions are getting shifted left instead of two. And we can see that the four positions that wrap around aren't getting completely reversed, but rather than the last position is getting shifted to the first. (Sort of a right-shift inside of a left-shift: *mobilis in mobili.*) Again, the odd-sized grids show the same shifting-by-four but without the back-to-front shuffle.

The next step, of course, is to extrapolate to grids with a zero digit in the third position, and generate a set of 81 grids for keys ending in 00:

3	123	115	107	99	91	83	75	67	59	51	43	35	27	19	11
4	124	116	108	100	92	84	76	68	60	52	44	36	28	20	12
5	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13
6	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14
7	127	119	111	103	95	87	79	71	63	55	47	39	31	23	15
72	0	120	112	104	96	88	80	8	64	56	48	40	32	24	16
73	1	121	113	105	97	89	81	9	65	57	49	41	33	25	17
74	2	122	114	106	98	90	82	10	66	58	50	42	34	26	18
67	59	51	43	35	27	19	11	4	124	116	108	100	92	84	76
68	60	52	44	36	28	20	12	5	125	117	109	101	93	85	77
69	61	53	45	37	29	21	13	6	126	118	110	102	94	86	78
70	62	54	46	38	30	22	14	7	127	119	111	103	95	87	79
71	63	55	47	39	31	23	15	72	0	120	112	104	96	88	80
8	64	56	48	40	32	24	16	73	1	121	113	105	97	89	81
9	65	57	49	41	33	25	17	74	2	122	114	106	98	90	82
10	66	58	50	42	34	26	18	75	3	123	115	107	99	91	83
4	124	116	108	100	92	84	76	68	60	52	44	36	28	20	12
5	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13
6	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14
7	127	119	111	103	95	87	79	71	63	55	47	39	31	23	15
72	0	120	112	104	96	88	80	8	64	56	48	40	32	24	16
73	1	121	113	105	97	89	81	9	65	57	49	41	33	25	17
74	2	122	114	106	98	90	82	10	66	58	50	42	34	26	18
75	3	123	115	107	99	91	83	11	67	59	51	43	35	27	19
68	60	52	44	36	28	20	12	5	125	117	109	101	93	85	77
69	61	53	45	37	29	21	13	6	126	118	110	102	94	86	78
70	62	54	46	38	30	22	14	7	127	119	111	103	95	87	79
71	63	55	47	39	31	23	15	72	0	120	112	104	96	88	80
8	64	56	48	40	32	24	16	73	1	121	113	105	97	89	81
9	65	57	49	41	33	25	17	74	2	122	114	106	98	90	82
10	66	58	50	42	34	26	18	75	3	123	115	107	99	91	83
11	67	59	51	43	35	27	19	76	4	124	116	108	100	92	84

As I'm sure you've already guessed, they show a left-shift of eight positions as the second digit increases, and with the same back-to-front shuffle on wraparound for the even grids. There are only 9 grids for keys that end in 000, giving this:

126	118	110	102	94	86	78	70	62	54	46	38	30	22	14	6
127	119	111	103	95	87	79	71	63	55	47	39	31	23	15	7
0	120	112	104	96	88	80	72	64	56	48	40	32	24	16	8
1	121	113	105	97	89	81	73	65	57	49	41	33	25	17	9
2	122	114	106	98	90	82	74	66	58	50	42	34	26	18	10
3	123	115	107	99	91	83	75	67	59	51	43	35	27	19	11
4	124	116	108	100	92	84	76	68	60	52	44	36	28	20	12
5	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13
127	119	111	103	95	87	79	71	63	55	47	39	31	23	15	7
0	120	112	104	96	88	80	72	64	56	48	40	32	24	16	8
1	121	113	105	97	89	81	73	65	57	49	41	33	25	17	9
2	122	114	106	98	90	82	74	66	58	50	42	34	26	18	10
3	123	115	107	99	91	83	75	67	59	51	43	35	27	19	11
4	124	116	108	100	92	84	76	68	60	52	44	36	28	20	12
5	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13
6	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14
0	120	112	104	96	88	80	72	64	56	48	40	32	24	16	8
1	121	113	105	97	89	81	73	65	57	49	41	33	25	17	9
2	122	114	106	98	90	82	74	66	58	50	42	34	26	18	10
3	123	115	107	99	91	83	75	67	59	51	43	35	27	19	11
4	124	116	108	100	92	84	76	68	60	52	44	36	28	20	12
5	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13
6	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14
7	127	119	111	103	95	87	79	71	63	55	47	39	31	23	15
1	121	113	105	97	89	81	73	65	57	49	41	33	25	17	9
2	122	114	106	98	90	82	74	66	58	50	42	34	26	18	10
3	123	115	107	99	91	83	75	67	59	51	43	35	27	19	11
4	124	116	108	100	92	84	76	68	60	52	44	36	28	20	12
5	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13
6	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14
7	127	119	111	103	95	87	79	71	63	55	47	39	31	23	15
8	0	120	112	104	96	88	80	72	64	56	48	40	32	24	16

The pattern continues: a left-shift of sixteen positions as the first digit increases. So we extrapolate backwards from these:

120	112	104	96	88	80	72	64	56	48	40	32	24	16	8	0
121	113	105	97	89	81	73	65	57	49	41	33	25	17	9	1
122	114	106	98	90	82	74	66	58	50	42	34	26	18	10	2
123	115	107	99	91	83	75	67	59	51	43	35	27	19	11	3
124	116	108	100	92	84	76	68	60	52	44	36	28	20	12	4
125	117	109	101	93	85	77	69	61	53	45	37	29	21	13	5
126	118	110	102	94	86	78	70	62	54	46	38	30	22	14	6
127	119	111	103	95	87	79	71	63	55	47	39	31	23	15	7

And here, at last, is the original Ur-sequence: The ordering of positions before the key-based scrambling is done. With this, I had finally cracked the actual scrambling algorithm once and for all. My pseudocode looked like this:

```

tmp[0..N] = solution[0..N]
j = -1
for 0 ≤ i < N
    j += 16
    j -= N | 1 while j ≥ N
    solution[j] = tmp[i]
for 0 ≤ k < 4
    n = 24-k
    n -= N | 1 if n > N
    for 0 ≤ i < key[k]
        rotate solution[0..n], +1 if N % 2 == 0
        rotate solution[0..N], -n

```

The unusual adjustment of line 9 is to handle the special case of a grid that has less than 16 positions. You may recall that the

scrambling algorithm permits grid sizes as small as 12 positions. That special case also the reason why line 5 needs to use `while` instead of `if`. (If you're unfamiliar with the C bitwise operators, the expression  $N \mid 1$  simply gives  $N$  when  $N$  is odd, and  $N + 1$  when  $N$  is even.)

It was a little humbling that this short little algorithm had had me running around in circles for weeks. And really, it was mainly due to a single line — namely, line 11, since that's the line that introduces the irregularities that are unique to even-sized grids.

With this major milestone checked off of my to-do list, I could then turn my full attention back to the encryption process.

## The Encryption Process: Layers Between Layers

My current set of layers worked for grid sizes down to about 50 or so. By adding more layers, layers that had no effect on larger grids, I managed to improve my encryption model so that it worked correctly for all grids of size 20 and above. For all practical purposes, I had a working encryption algorithm — who cares about crossword puzzles with less than 20 squares? — but it was ridiculously complicated and obviously not representative of the real algorithm.

Instinctively, I guessed that I had incorrectly identified several of the layers, and that certain layers, when combined, would cancel out a lot of unnecessary complexity and leave me with fewer layers that were more well-behaved. Unfortunately, this theory failed to be borne out in practice: The layers kept failing to line up with each other consistently.

But: I was still better off than I was. The nice thing about having a overly-complicated working model, is that I had a working model that expressed the change to each letter in terms of the digits from the key. Instead of just saying e.g. "This position had thirteen added to it," I could say, "This position had been increased by **c** plus three times **d**." This would represent the sum total of all of the encryption layers onto a single position. I therefore started working on a tool that would display the encryption for each position by how many times each key digit was added (or subtracted) from the original letter. My hope, of course, was that displaying the four coefficients for each position might make new patterns visible, and point the way to reorganizing the layers.

I never came up with a good way to represent this information visually, but as it turned out it didn't matter. The first rough version of my script quickly showed me two new facts. First, I found that the coefficients were never negative. This was surprising, since nearly all of the layers subtracted some values. But all of the subtractions were balanced out by other additions. That argued strongly that all of the subtractions were just manifestations of incorrect layer separation — which worried me, since that included pretty much all of them. The second fact, however, rendered that nearly irrelevant. I realized that the total number of factors (i.e. the sum of all four coefficients) always added up to four. For every position in every grid, without fail.

This is a pretty big example of an invariant that one would expect to be more prominently visible in the actual algorithm.

My initial gut reaction was to throw out the all of my layer work and start over. Once I calmed down, however, and started looking at it with a clearer head, I began to look for new ways forward.

First I made a handful of attempts to construct a model that enforced the four-coefficients requirement, but I quickly discarded them all as being unlikely. Eventually I connected the requirement with the fact that there were four stages to the reordering process. For most of this time, I had tentatively assumed that the encryption and reordering processes were separate stages. But now that I had a clear understanding of the reordering process — i.e. I knew that it occurred in four phases, and I had a sense what the grid looked like in between each phase — it seemed entirely possible that some or all of the irregularities in the positioning of the encryption layers could be because they were done in between the reordering steps, on a partly rearranged grid.

## The Scrambling Process

Investigating this called for some new tools. Up until then I had been doing colorized output separately in each script, hardcoding the colors for whatever feature I was trying to look at. Now that I wanted to display a lot more data at once, I decided to try a different approach. My new scripts would "mark up" each character in the output with arbitrary attributes, by appending extra data to each character of output. I then had a separate script that would read in all of the output, and dynamically select colors at runtime for each unique combination of attributes. This put all the colorizing in one place, which made it easier to write scripts that could read the output of another script (since it was easier to subject my markup to a read-modify-write process than ANSI escape sequences).

I had already written a nice little script that could do selective decryption. As I had separated out the various layers, I had given each one a vaguely descriptive name (stripes, bands, spike, and so on), and I had written a script that removed layers selectively, allowing me to specify on the command line which ones to undo and which ones to leave in the output. Now that I understood how the reordering was done, I wrote another script that allowed me to selectively undo the various reordering phases. With this arsenal, I could now do some serious exploration of how the reordering and the encryption steps interacted with each other.

The image shows a large grid of text, likely a crossword puzzle or a similar structured text. Each character in the grid is color-coded. The colors represent different layers of encryption or reordering. The grid is 20 rows by 100 columns. The colors include various shades of green, yellow, orange, red, and blue, indicating different stages of the cryptographic process.

In this example the colors indicate how the letter changes (or doesn't change) along with **d**, the key's fourth digit. Since the colors are now being assigned dynamically, you have to study the output to see what each one indicates. (In this display, for example, green indicates a letter that doesn't change as the digit changes, red indicates a letter that goes up by one as the digit increments, and so forth.) By displaying each grid as a square, that also allowed me to do rough visual comparisons of the third digit, **c**, as well. (Technically, they're rectangles, since there are twice as many columns as rows.)

The image shows a large grid of text, likely a crossword puzzle or a similar structured text. Each character in the grid is color-coded. The colors represent different layers of encryption or reordering. The grid is 20 rows by 100 columns. The colors include various shades of green, yellow, orange, red, and blue, indicating different stages of the cryptographic process.

This example shows the display of grids as the previous display, but now I've used my scripts to remove all but one of the layers (the one I had named the "blocks" layer). Visually, the layer appears to consist of four sections. Two of the sections are just unencrypted **As**. In the bottom three rows, where the key begins with 89, the top colored section is unaffected by changes to **d**,



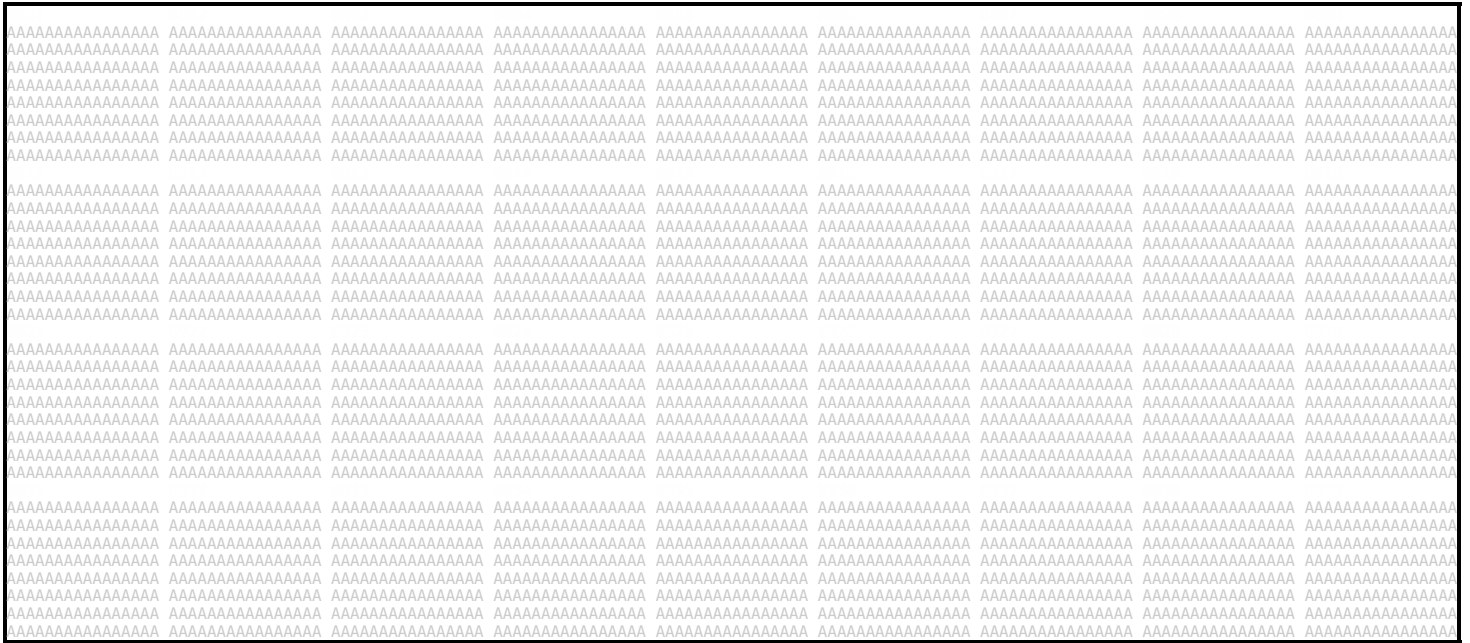


But now, this is what the underlying quartet layer looks like after the first phase of the reordering.

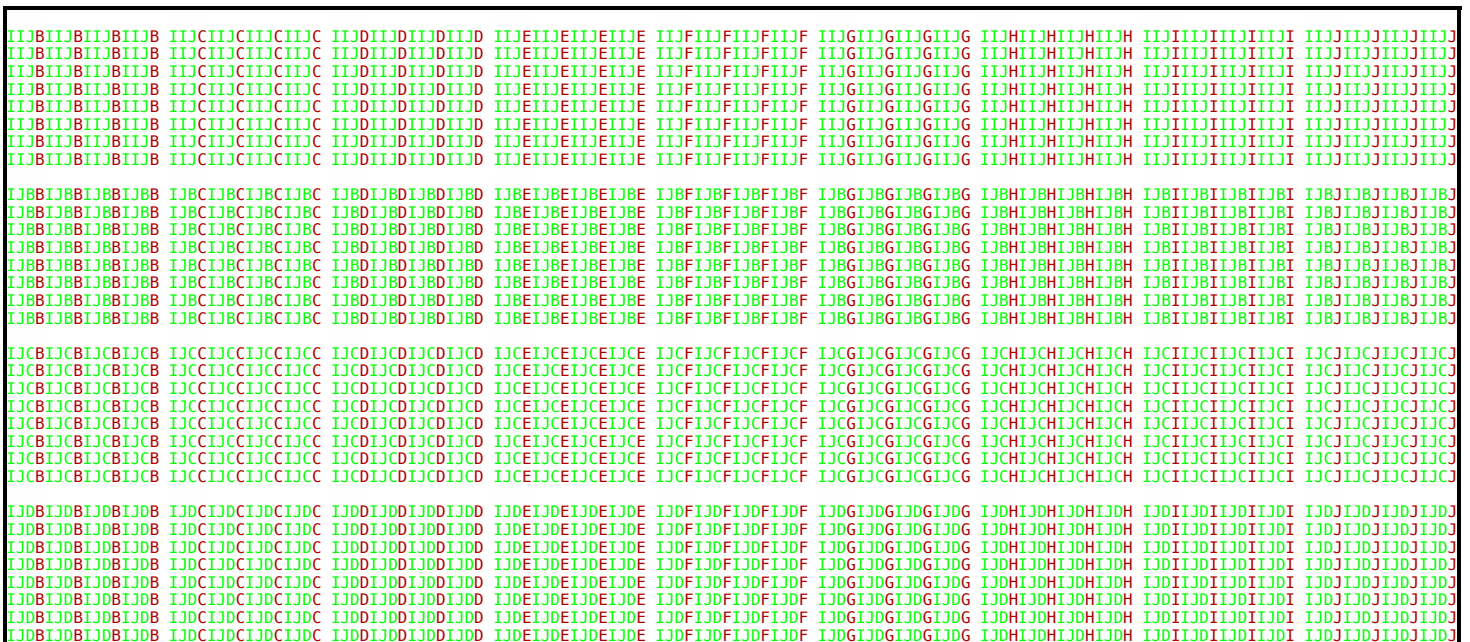
[illegible]

## The Scrambling Process: For Real This Time

No, really. That's the whole thing. Start out with a collection of grids of all As:



The first phase of the encryption applies each of the digits of the key in succession, repeating until the grid is filled. After reordering, the grid regions of width sixteen have been completely intermixed, so that they look like regions of width one.



The second phase of encryption does the same thing with regions only half as wide. But since it gets applied at a later phase of the reordering process, the regions don't line up with the regions from the first encryption phase, and thus a bit of chaos is introduced.

Adding in another phase of the encryption process, again applied at yet another point in the reordering process, continues to obscure the patterns that were visible.

And finally, with all four phases of the encryption process applied, we are back to the display that we were examining at the beginning of this section.

All that was left was a little hiccup of how to apply the regions of size sixteen when the entire grid had less than sixteen positions. That threw me for a loop until I realized that instead of filling out regions consecutively, the algorithm was jumping around from one region to the next, each time filling in one position in each region and wrapping around until the whole thing was completed. (All that time I spent looking at "strides" in the reordering process, that wound up not being used at all in the actual algorithm? It turns out that something very much like the strides were in the encryption process all along.)

So here's the complete algorithm, encryption and reordering:

```

tmp[0..N] = solution[0..N]
j = -1
for 0 ≤ i < N
    j += 16
    j -= N | 1 while j ≥ N
    solution[j] = tmp[i]
for 0 ≤ k < 4
    n = 24-k
    j = -1
    for 0 ≤ i < N
        j += n
        j -= N | 1 while j ≥ N
        solution[j] = (solution[j] + key[i % 4]) % 26
    n -= N | 1 if n > N
    for 0 ≤ i < key[k]
        rotate solution[0..n], +1 if N % 2 == 0
        rotate solution[0..N], -n

```

Yes. That's the whole thing. If you compare it to the pseudocode I supplied for the reordering process, you'll see that the encryption process is just five lines added in the middle.

I sat down and wrote my scrambling and unscrambling algorithms as [simple, portable C code](#), and delivered it to my friend. They were surprised and very happy to get it, which was gratifying, and soon afterwards their fully-featured app was available for purchase in the Apple store.

## And So What Have You Learned?

At the end of this extended venture, it seems only natural to try to extract some lessons from the experience. Here are the ones that I pulled out:

- Presentation matters

It's a bit comical to look back to how much I initially resisted using color in my exploratory scripts, knowing now how much it made various patterns stand out. Likewise, I wish that I had created the script to do coloring on-the-fly much earlier in the process, instead of waiting until nearly the end.

- for n in 1 2 3 ; do echo iterate ; done

Reverse engineering is full of chicken-and-egg problems. You need data in order to form a sensible hypothesis, but you need a hypothesis to know what data to gather. The solution, as always, is to start with a first approximation and iterate your way into the solution space. Favor exploratory prototypes and quick turnaround. Write high-level scripts using easy-to-debug text interfaces. Test, refine, test, refine.

- Perspective can make all the difference

Think about how the scrambling process looked so complicated when viewed as series of "strides", filling in the grid from a source, instead of as a simple series of shifts and rotations applied to the initial grid. Examining something from the wrong point of view can just import extra complexity. The fact is, all of my work on dissecting the behavior of the strides wound up being a complete waste of time, causing me to spin my wheels on a bad model for weeks. And when I finally hit upon the better model, all of that earlier work became nothing more than a distraction. On the other hand, my original ideas about the encryption process were just as wrong, but I was able to leverage them to dig deeper, and thus I was able uncover new patterns that pointed me towards a better model. So, time invested in a bad model isn't always time wasted.

- Don't get attached to your jury-rigging

After I had figured out how to influence the key selection, and rewritten my data collection tools to start collecting grids in sequence, I wondered why I hadn't done this sooner. Some of it was laziness, no doubt, but some of it also was that I was perversely proud of my Rube-Goldberg OCR-based data collection process. Make no mistake: I'm still proud of it, and justifiably so. But I should have realized sooner the necessity of examining grids with contiguous key values. Given how easy it was to figure out the method for choosing the key value, there was really no reason why I couldn't have done that much, much sooner.

- Remember Occam's Razor

Assume things are simple at first. Explore the complicated explanations after the simple one has failed. I once heard it put this way: "Everyone knows that hindsight is 20/20. So, take advantage of that: Try to write as much of your program as possible with the benefit of hindsight." In other words, don't over-engineer complicated solutions before you know why the simple one isn't good enough. In a way this is just a restatement of Occam's Razor, but I like it because it clarifies *why* Occam's Razor is a good idea. It's not because simpler solutions are actually more likely to be true; they frequently aren't. It's because it's almost always easier to improve a simple solution by adding complexity, than it is to improve a complicated solution by digging out a simple solution buried within it.

## Postscript

The whole reverse-engineering process, from start to finish, took six weeks. I was working a full-time job during that time, of course, so this was six weeks of evenings and weekends. (And other idle moments: I made the final connection that explained the encryption process while sitting on the bus.) Of course, the real process was more haphazard than this essay might have made it seem. But a more faithful narrative would be far more confusing to read. To a certain degree, I've favored pedagogy over precision.

Unfortunately, the app containing my code didn't remain in the store for too long. The people involved moved on to other projects (and/or were hired for other jobs), and when the original iPhone was superseded by later versions, the app wasn't updated.

I had made a mental note to publicly document the scrambling algorithm at some point in the future, but like most people my to-do list is longer than my available free time. A little over a year later, somebody else had beat me to it, and published the scrambling algorithm as part of their own software project.

So instead I took the notes that I had made during the process, and used them as the basis for a presentation on the subject of reverse engineering. (The presentation was delivered at [SeaGL](#) 2013.) I realized that it made for a good story, something that we don't have enough of in this field. As programmers we usually present the end result, the *fait accompli*. But sometimes, the story that includes the fumbblings in the dark and the chasing down dead ends will be more informative, more memorable, or just plain more interesting. By studying the epicycles of Ptolemy and Copernicus, we hope to get better at recognizing them in our own investigations.

---

[Texts](#)

[Brian Raiter](#)