Cryptography, Math and Programming with Cryptol

(and maybe some Python)

Dylan McNamee

Chapter 1

Getting started

This book is a guide to learning about cryptography, the math that cryptography is built on, and how to write programs that implement cryptographic algorithms. Don't worry if any of that sounds too complicated: it's all explained inside. Also don't worry if it sounds too boring: these topics are surprisingly deep and interesting – there's a lot of cool stuff even jaded students (of all ages!) can learn.

At the beginning of each chapter, we'll describe what you'll know by the end of the chapter. If you feel like you already know that stuff, try skimming the chapter (don't skip it!) to make sure that you do, and slow down and read the new stuff.

At the end of this chapter you'll know what resources you'll need to complete the activities in the book, and you'll have encrypted and decrypted a message using the Caesar cipher. Let's get going!

What you'll need

You'll need a sense of curiosity. You'll also need a dedication to actually doing the exercises: if you just read this material you'll only get a small fraction of the benefit of doing. You'll need access to a computer. It can be a Windows, MacOS or Linux computer - they're all fine.

Having a group of people to work with is a good idea. A lot of the activities require serious thought, and it's totally normal to get stuck (often). If you work with a group of people, when one of you gets stuck, the others can help out. Not by giving answers, but by nudging in productive directions. Often all it takes to do this is to ask “What are you working on? What have you tried? Is there anything you haven’t tried yet?” Explaining the answers to these questions to another person is often enough to get unstuck.

Why cryptography?

Cryptography is the mathematics of secret messages. The popularity and pervasiveness of social media has caused some people to comment “nothing is secret.” But is that really true? People share photos taken in restaurants all the time, but is it a good idea to share a photo of the credit card you used to pay for your food? What about sharing your social media passwords? Needless to say, privacy and cryptography are both interesting, and related to each other in subtle ways.

Let’s get started

Print out and assemble the Reverse Caesar Cipher kit that comes with the book. The Caesar Cipher is one of the earlest known ciphers, used by Caesar to communicate orders to distant generals. The idea is that if the messenger was intercepted by foes of Caesar, that they wouldn’t learn any secrets from the message they carried.

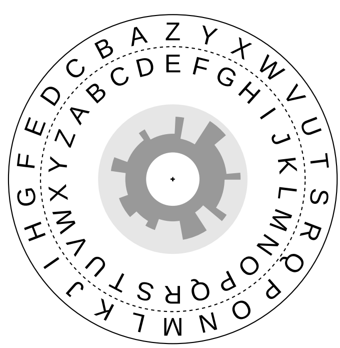

Figure 1.1: Example cipher wheel setting - you could call this A↔D, Z↔E, among other things

Follow the instructions on the kit to decode the following punchlines:

I wondered why the ball was getting bigger … (Use the key: J↔M)

CORI NC ONC JR

What do you call a counterfeit noodle? (Key: F↔O)

TG LHETBAT

A backward poet … (Key: H↔H)

SXGVKW GBTKXWK

If you managed to decrypt all three of these, congratulations - that's a lot of work! When I use a Caesar's Cipher wheel, it takes about 3 seconds per letter to encrypt or decrypt a message. At that rate, it would take about an hour to encrypt a whole page of text, which is way too long.

Given how tedious it is to decrypt, even when you know the key, it's not too hard of a stretch to imagine Caesar thinking this cipher was good enough. Now that we have computers, it's a lot easier to encrypt and decrypt messages, and the Caesar Cipher is not close to good enough. We'll learn about how "real world" cryptography works later in this book.

Things to ponder

1. if a computer can decrypt a message in 1 millisecond[1], how long would it take, on average, to decrypt a Caesar Cipher message whose key you don't know[2]?

2. how much more secure would it be to have weird symbols (Greek letters or Egyption heiroglyphics), instead of letters, for the cypher-text in a Caesar's Cipher? Explain your answer.

3. *key distribution* is the challenge of getting the secret key to your friend. One way to distribute a key would be to include it in some hidden way in the message. Come up with a few ways you could do this with the Ceasar Cipher. Another way would be to agree on a shared key when in the same room as your friend. What are some of the advantages and disadvantages of these two approaches?

[1] that's 1/1000-th of a second.

[2] Hint: first, how many different possible keys are there? It's safe to guess that on average, you'll have to try half of them before guessing the right one.

Take-aways

You've thought about why cryptography is important. You know how to encrypt and decrypt messages using the Caesar Cipher. You have thought about how secure it is, including the aspects of key distribution.

Chapter 2

Encoding data into bits

"There are 10 kinds of people in the world: those who know binary and those who don't."

Now that you've seen encryption and decryption at work, it's time to learn how computers do it. Our Caesar's Cipher wheel is a paper computer which has an alphabet of 26 elements. You've heard (most likely) that computers work with ones and zeros. One's and zeros are not very helpful by themselves, so people figured out how to represent integers, floating point numbers and all of the letters in all of the languages around the world using only ones and zeros

The process of representing one set of things (integers, for example) using another set of things (sequences of ones and zeros) is called *encoding*. *Decoding* is reversing the process; getting back the original information from the new representation. In this chapter, we'll learn how to encode and decode unsigned and signed integers, simple Latin alphabets, as well as the rest of the alphabets in the world

Encoding integers

If the joke at the beginning of this chapter makes sense, and you know about *number bases*, encoding integers using ones and zeros is simply converting to base 2, and you can skip to the next section. To learn what this means, and why that joke isn't leaving out eight kinds of people, read on[1].

We're so used to seeing a number like 533 and understanding it to mean "five hundred and thirty-three" that we forget that it's an encoding of a numeric value into the symbols 0, 1, 2 … 8, 9. Reading from right to left, the n^th^ digit is the 10^{n-1}-place[2]. So deconstructing our example number we get:

Exponent	10^3	10^2	10^1	10^0
Value	1000	100	10	1
Digit	0	5	3	3
Digit value	0	500	30	3

So finally we get $0 + 500 + 30 + 3 = 533$.

Binary representations of numbers

Encoding numbers in binary is the same recipe, but with 2 as the base of the exponent instead of 10. Each place (digit) can either have a 1 or a zero in it. As a result, you need more digits to represent the same values, but it works out.

Exponent	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	512	256	128	64	32	16	8	4	2	1
Digit	1	0	0	0	0	1	0	1	0	1
Digit value	512	0	0	0	0	16	0	4	0	1

In this case we get $512 + 16 + 4 + 1$ = 533.

To convert a number into binary, take the largest digit that isn't bigger than your value, and set it to 1, then subtract that digit's value from your value and repeat down the line, not forgetting to put in 0's for the values you don't want to add.

Adding binary numbers is super-easy. It's a lot like the addition you're used to, with carrying and everything, except simpler. If both places are 0, the sum is 0. If either is 1, the sum is 1. If both are 1, the sum is 0, and you carry 1. When you include the carry, the rule is the same, except

[1]Note that we didn't promise we'd convince you this joke is funny, only that you'll understand what it's getting at.

[2]Remember that $10^0 = 1$

sometimes you have both 1 along with the carry, in which case the sum is 1 and the carry is 1.

Here's a four-bit addition of 2 and 3:

```
   1    <- carry
  0010
+ 0011
------
  0101
```

If you're talking about numbers in different bases, and it's not clear which one you're refering to, it's common to include the base in the subscript after the number. So the joke at the beginning of the chapter would be "There are 10_2 types of people…"[3].

Working from the right, $0 + 1 = 1$ with no carry, then $1 + 1 = 0$ with carry, and finally we have a carry with $0+0$, so that's 1. 101_2 is 5, which is what we're hoping for.

Representing negative numbers:

> This section is a deep-dive. You can skip it the first time through - but if you get bored or ever get curious, come back here for some cool stuff.

The encodings we've discussed so far are only for positive numbers. If you want to represent negative numbers, there are a few options, but one fairly universally agreed-on best way to go.

The obvious (but not best) way is to reserve the top-most bit to represent negation. If the bit is 0, the rest of the bits are a positive number, if it's 1, the rest of the bits are interpreted as a negative number. This encoding makes sense, but it makes arithmetic difficult. For example if you had 4-bit signed numbers, and wanted to add -1 and 3, you'd get

```
   11  <- carry
  1001
+ 0011
------
  1100
```

[3] But that way kind of ruins the joke, doesn't it?

This shows that if we apply our naive addition to $-1 + 3$, we get the unfortunate answer -4. It turns out that if you represent negative numbers by flipping the bits and adding one, you can do arithmetic using simple unsigned operations and have the answers work out right.

To get a four-bit -1 in two's complement, here's the process:

```
Step 1: 0001   <- +1
Step 2: 1110   <- flipped
Step 3: 1111   <- add 1 is -1 in two's complement
```

Here's $-1 + 3$ again, in two's complement:

```
  111  <- carry
  1111 <- -1 (from above)
+ 0011 <- 3
------
  0010
```

In the one's place, $1 + 1 = 0$ carry 1, then we have $1 + 1 +$ carry = 1 carry 1, then we have 1 + carry = 0 with carry, and the last digit is also 1 + carry = 0 (and the carry goes away). You'll see the answer, $0010_2 = 2$, which is what we're hoping for.

Things to think about

1. What's the largest value you can represent with one base-ten digit? Two digits? n-digits?
2. What's the largest value you can represent with one binary digit? Eight digits? n-digits?
3. When we did $-1 + 3$, the carry bit got carried off the end of the addition. This is called overflow. In some cases (like this one), it's not a problem, but in other cases, it means that you get the wrong answer. Think about whether you can check whether overflow has occured either before or after the addition has happened.
4. Two's complement is a slight change from *one's complement*, in which negative numbers just have their bits flipped, but you don't add a 1 afterwards. A big advantage of two's complement is that there are two ways to write 0 in one's complement: 10000... and 0000.... Essentially you have a positive and a negative zero. Think about what problems this might cause.

5. What's the largest value you can represent in a two's complement 8-bit number? What's the smallest?

Encoding text into ones and zeros

Now that you understand how numbers can be represented as ones and zeros, we can explain how text can be represented as sequences of numbers, and you can convert those numbers into bits.

It turns out that how to assign numbers to letters can be arbitrary. Until the early 1960's, there were a number of competing text → bits encoding systems. People realized early on that deciding on one system would let them communicate more easily between different machines. The most common text encoding, called ASCII, was agreed on in 1963, and was in wide use through the mid 1990's.

Here's how ASCII represents the basic letters, numbers and punctuation:

sp	32	!	33	"	34	#	35	$	36	%	37	&	38	'	39
(	40	)	41	*	42	+	43	,	44	-	45	.	46	/	47
0	48	1	49	2	50	3	51	4	52	5	53	6	54	7	55
8	56	9	57	:	58	;	59	<	60	=	61	>	62	?	63
@	64	A	65	B	66	C	67	D	68	E	69	F	70	G	71
H	72	I	73	J	74	K	75	L	76	M	77	N	78	O	79
P	80	Q	81	R	82	S	83	T	84	U	85	V	86	W	87
X	88	Y	89	Z	90	[	91	\	92	]	93	^	94	_	95
`	96	a	97	b	98	c	99	d	100	e	101	f	102	g	103
h	104	i	105	j	106	k	107	l	108	m	109	n	110	o	111
p	112	q	113	r	114	s	115	t	116	u	117	v	118	w	119
x	120	y	121	z	122	{	123	|	124	}	125	~	126	del	127

So the string "Hi there" in ASCII is: 72, 105, 32, 116, 104, 101, 114, 101.

Some exercises

1. Encode your name in ASCII.

ASCII has some clever design features. Here are some questions that may uncover some of that cleverness:

2. Is there an easy way to convert between upper and lower-case in ASCII? Think about the binary representations.

3. Is there an easy way to convert between a digit and its ASCII representation? Does the binary representation help here? What aspects of the ASCII encoding make this easy/difficult?

Encoding *all* languages: Unicode

> This section is a deep-dive: you can do the rest of the book knowing only ASCII. On the other hand, if you like to know how things work under the hood, you'll enjoy learning how non-Latin web pages are encoded and transmitted.

Up until the mid 1990's, computer systems that needed to process languages whose characters are not in the ASCII tables each used their own encodings. When the Internet and Word Wide Web started to gain adoption, people realized that they would have to standardize how these other languages encoded their alphabets into bits. The Unicode Consortium was the group founded to make those standards. They took the sensible approach of splitting the problem into two stages:

1. Enumerating all of the symbols that can be represented. This includes accents, special glyphs, and now also includes emoji. As of 2016, there are over 1.1 million different "code points" in the master Unicode table.

2. Devising efficient ways of representing sequences of those symbols as bits.

The hard work of the first stage is to come to agreement on which symbols go in (and which to leave out), what to call them, and how to organize them. The folks working on stage two have come up with a number of encodings, but the one that is most common on the Internet is UTF-8. The genius of UTF-8[4] is that it's *backwards compatible* with ASCII. What that means is that if your text *does* fit in the ASCII table, the ASCII representation of it is also the UTF-8 representation of it. The key to making that work is that while ASCII is an 8-bit representation, the topmost bit of the ASCII table is always 0.

[4] UTF-8 was invented at Bell Labs by Ken Thompson, who co-invented Unix, and Rob Pike, who subsequently invented the Go programming language.

If you're decoding a UTF-8 stream of bytes, and you encounter any byte with its top bit off (i.e., its decimal value is <= 127), decode it as ASCII. If the top bit is on (the number is > 127), follow this procedure:

1. The first byte tells you how many bytes are in this character. Count the number of bits set before the first "0"-bit. That number is the number of bytes in this character. The remaining bits after the 0 are data. UTF-8 supports up to 4 bytes, so the longest (4-byte) UTF-8 character will start `11110...`

2. The remaining bytes are tagged with a leading "10" (so you can tell they aren't beginnings of characters), and the remaining 6-bits are data.

3. Concatenate the data bits into one binary number.

4. Look up that number in the Big Unicode Table.

Pretty cool!

An aside: Hexadecimal

Writing numbers in binary is tedious. It takes eight digits to count up to 128, after all! Writing them in decimal is convenient for us humans, but a downside is that there's no easy way to tell how many bits a number has. Computer scientists have settled on *hexadecimal*, or base 16, to write numbers when the number of bits matters. How does one write a hexadecimal number? After all, we've only got ten digits, 0 -> 9, right? Well, as a convention we use the first six letters of the alphabet to represent the digits past 9. So counting to 16 in hexadecimal (or "hex" for short), looks like this:

1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10

Hex, just like decimal and binary, has a *one's place*, but the next bigger digit in hex is the *sixteen's place*[5], so 10 in hex is 16 in decimal (also written as $10_{16} == 16_{10}$). A in hex is 10 in decimal. This means that one hex digit holds exactly four bits, and it takes two hex digits to hold a byte. This is important right now, because Unicode tables are all written in hex, as you're about to see:

[5] and the next digit is the 256'ths place!

Back to Unicode

Below is a table with three sample Unicode symbols. Each symbol has a long, boring unambiguous name, its graphical symbol (which can vary from font to font), its global numeric code in the master Unicode table, and finally how that number is encoded in UTF-8.

Unicode name	Anticlockwise Gapped Circle Arrow	Bicycle	Pile of Poo
Symbol	⟲	🚲	💩
Numeric code	`U+27F2`	`U+1F6B2`	`U+1F4A9`
UTF-8	`E2 9F B2`	`F0 9F 9A B2`	`F0 9F 92 A9`

Figure 2.1: Some example Unicode glyphs, their official Unicode name, number and UTF-8 encodings

In the table above, the "U+" lets you know that the hex number that follows is the location in the Unicode table, and you see that the UTF-8 encoding is also written in hex. There's a cool webpage at http://unicode-table.com/en/ that has the whole table in one page. On the right of the page there is a live map with dots in the parts of the world where the characters visible on the current screen are used.

Independent study questions

If you're interested into learning more about how information can be digitally encoded, here are some questions you can research the answers to.

1. Two common ways of **encoding images** are pixel-based (or bitmap) and vector-based:

    a. The main aspects of **pixel-based** encoding are resolution (how many pixels there are in the image), how to encode colors (the value at each pixel), and compression (e.g., to reduce the storage for simple scenes like a plain blue sky). Common pixel-based formats are PNG, JPEG, and GIF.

b. The main aspects of **vector graphics** are what *primitives* to provide, which are the shapes that are supported built-in (lines, curves, circles, rectangles) vs. which ones need to be assembled from sequences of primitives, what the *coordinate system* for describing shapes is, and what the *syntax* is. Vector graphics formats tend to more-resemble programming languages, and are often in human- readable ASCII. Common vector-based formats are PDF, SVG, and PostScript.

What's an image encoding method you know about? Use Google to find a specification for that format, and Write down how files in that format are structured. Most formats have a *header* which provides *metatada* about the file[6].

2. **File archives** are encodings that combine a bunch of files and folders into one file that can be sent by email, or downloaded from a website, etc., and then *unpacked* at the other end. Archive formats often include the ability to compress the files as well. It's often surprising which file formats are archives. For example, most word processing document formats are file archives, to allow you to include graphics. Installers for most systems are also archives, such as Windows MSI files, MacOS DMG files, and Linux RPM files. Early archive formats include TAR and ZIP, which were invented more than 30 years ago, but are still used every day.

   If you know a particular file archive format, look it up on the Internet and write it up in a page or so.

Take-aways

You've learned about how to encode data of different types (numbers, characters) into binary representations. You've learned some binary arithmetic, and why 10_2 is 2_{10}. Finally you've learned that nerds (the author included) can have a terrible sense of humor.

[6] The word metadata literally means "data about data", which particularly makes sense in this context