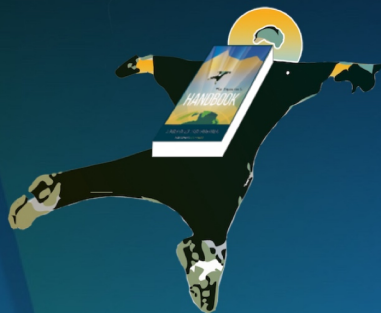


A CS PRIMER FOR SELF-TAUGHT DEVELOPERS



*The Imposter's*  
**HANDBOOK**  
SECOND EDITION

ROB CONERY

# THE IMPOSTER'S HANDBOOK

A CS PRIMER FOR SELF-TAUGHT DEVELOPERS,  
SECOND EDITION

ROB CONERY

**ISBN 978-0-692-93303-9**

*COPYRIGHT BIG MACHINE, INC, 2021*

*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.*

**Published by: Big Machine, Inc**

*Please forward all questions RE publication to [rob@bigmachine.io](mailto:rob@bigmachine.io)*



# FOREWORD: SCOTT HANSELMAN

I never did a formal Computer Science course. My background was in Software Engineering. For a long time, I didn't understand the difference, but later I realized that the practice of software engineering is vastly different from the science of computers.

Software Engineering is about project management and testing and profiling and iterating and SHIPPING. Computer Science is about the theory of data structures and  $O(n)$  notation and mathy things and oh I don't know about Computer Science.

Fast forward 25 years and I often wonder if it's too late for me to go back to school and "learn computer science." I'm a decent programmer and a competent engineer but there are...gaps. Gaps I need to fill. Gaps that tickle my imposter syndrome.

I've written extensively on Imposter Syndrome. I'm even teased about it now, which kind of makes it worse. "How can YOU have imposter syndrome?" Well, because I'm always learning and the more I learn the more I realize I don't know. There's just SO MUCH out there, it's overwhelming.

Even more overwhelming are the missing fundamentals. Like when you're in a meeting and someone throws out "Oh, so like a Markov Chain?" and you're like "Ah, um, ya, it's...ah...totally like that!"

If only there were other people who felt the same way. If only there was a book to help me fill these gaps. Ah! Turns out there is. *You're holding it.*

**Scott Hanselman** @shanselman August 12, 2016 Portland, Oregon

*Scott Hanselman has been computerizing, blogging, and teaching for many years. He works on Open Source .NET at Microsoft and has absolutely no idea what he's doing.*

# FOREWORD: CHAD FOWLER

I've been honored to be asked to write the foreword for several books over the course of my career. Each time, my first reaction is something like "Oh wow, how exciting! What an honor! HEY LOOK SOMEONE WANTS ME TO WRITE THEIR FOREWORD!!!"

My second reaction is almost always, "Oh no! Why me? Why would they ask me of all people? I'm just a saxophone player who stumbled into computer programming. I have no idea what I'm doing!"

No offense to Rob intended, but this may be the first foreword I feel qualified to write. Finally, a book whose very title defines my qualifications not just to write the foreword but to participate in this great industry of ours. A handbook for impostors. It's about time.

You know that friend, classmate, or family member who seems to waste too many precious hours of his or her life sitting in front of a computer screen or television, mouth gaping, eyes dilated, repetitively playing the same video game? In my late teens and early twenties, that was me. I made my living as a musician and had

studied jazz saxophone performance and classical composition in school. I was an extremely dedicated student and serious musician. Then I got seriously addicted to id Software's Doom. I played Doom all the time. I was either at a gig making money or at home playing the game. My fellow musicians thought I was really starting to flake out. I didn't practice during the day or try to write music. Just Doom.

I kind of knew how personal computers worked and was pretty good at debugging problems with them, especially when those problems got in the way of me playing a successful game of Doom death-match. I got so fascinated by the virtual 3D rendered world and gameplay of Doom that my curiosity led me to start learning about programming at around age 20. I remember the day I asked a friend how programs go from text I type into a word processor to something that can actually execute and do something. He gave me an ad hoc five minute explanation of how compilers work, which in different company I'd be ashamed to admit served my understanding of that topic for several years of professional work.

Playing Doom and reading about C programming on the still-small, budding internet taught me all I knew at the beginning about networking, programming, binary file formats (We hacked those executables with hex editors for fun. Don't ask me why!), and generally gave me a mental model for how computer systems hung together. With this hard-won knowledge, I accidentally scored my first job in the industry. The same friend who had explained compilers to me (thanks, Walter!) literally applied for a

computer support job on my behalf. At the “interview”, the hiring manager said “Walter says you’re good. When can you start?”

So, ya, I really stumbled into this industry by accident. From that point on, though, I did a lot of stuff on purpose. I identified the areas of computer technology that were most interesting to me and systematically learned everything I could about them. I treated the process like a game. Like a World of Warcraft skill tree, I worked my way up various paths until I got bored, intimidated, or distracted by something else. I covered a lot of territory over many hours of reading, asking questions of co-workers, and experimentation.

This is how I moved rather quickly from computer support to network administration to network automation. It was at this time that the DotCom bubble was really starting to inflate. I went from simple scripting to object-oriented programming to co-creating a model/view/controller framework in Java for a major corporation and playing the role of “Senior Software Architect” only a few short years after packing the saxophone away and going full time into software development.

Things have gone pretty well since then, but I’ve never gotten over that nagging feeling that I just don’t belong here. You know what I mean? You know what I mean. You’re talking about something you know well like some database-backed Web application and a co-worker whips out Big-O notation and shuts you down. Or you’re talking about which language to use on the next project, and in a heated discussion the word “monad” is invoked.

Oh no. I don't know what to say about this. How do I respond? How can I stop this conversation in a way that doesn't expose me for the fraud I am? WHAT THE HELL IS A MONAD?

*Haha, ya.*

I hope that non-response made sense, you think as you walk toward the restroom, pretending that's why you had to suddenly leave the discussion.

In daily work, I find myself to be at least as effective as the average programmer. I see problems and I come up with solutions. I implement them pretty quickly. They tend to work. When performance is bad, I fix it. When code is hard to understand I find a way to make it easier to understand. I'm pretty good at it I guess.

But I didn't go to college for this stuff. I went to college and studied my ass off, but all I have to show for it is an extremely vast array of esoteric music knowledge that would bore the hell out of anyone who isn't a musician. In college you learn about algorithms. That sounds hard. When I write code, I don't use algorithms I think. Or do I? I'm not sure. I don't invoke them by name most of the time. I just write code. These college programmers must be writing stuff that's unimaginably more sophisticated since their code has algorithms!

And how can my code perform well if I didn't use Big-O notation to describe its performance and complexity characteristics? What the hell does "complexity" even mean in this context? I must be wast-

ing so many processor cycles and so much memory. It's a miracle my code performs OK, but it does.

I think most of my success in the field of computer software development comes from my belief that:

1. A computer is a machine. In some cases it's a machine I own. I could break it into tiny pieces if I wanted.
2. These machines aren't made of magic. They're made of parts that are pretty simple. They're made in ways that tens of thousands of people understand. And they're made to conform to standards in many cases.
3. It's possible to learn about what these components are, how they work, and how they fit together. They're just little bits of metal and plastic with electricity flowing through them.
4. Everything starts with this simple foundation and grows as simple blocks on top.
5. All of the hard sounding stuff that college programmers say is just chunks of knowledge with names I don't know yet.
6. Most of this stuff can be learned by a young adult in four years while they're also trying to figure out who they are, what they want to do with their lives, and how to do as little work as possible while enjoying youth.
7. If someone can learn all this stuff in just a four year degree, it's probably pretty easy to hunt down what they learn and learn it myself one concept at a time.

8. Finally, and most important, somehow I get good work done and it doesn't fall apart. All this stuff I don't know must be just a bonus on top of what I've already learned.

All this is just stuff you can learn! Wow. In fact, the entirety of human knowledge is just a collection of stuff that you can learn if you want to. That's a pretty amazing realization when you fully absorb it. A university doesn't magically anoint you with ability when you graduate. In fact, most people seem to leave university with very little actual ability and a whole lot of knowledge. Ability comes from the combination of knowledge, practice, and aptitude.

So, what separates us impostors from the Real Deal? Knowledge, practice, and aptitude. That's it. Knowledge is attainable, practice is do-able, and we just have to live with aptitude. Oh well.

Here's a big secret I've discovered: I'm not the only impostor in the industry. The first time I met Rob, he interviewed me for a podcast. We ended up talking about Impostor Syndrome. On hearing my interview, several people told me "Wow, I feel like that too!" The more I dig, the more I think we all feel like that at some points in our careers.

So, welcome to Impostor Club! I'm kinda bummed now that I know it's not as exclusive as I had thought, but it's nice to have company I guess.

Anyway, now we have a handbook.



Ironically, reading and using this handbook might cause you to graduate from the club. If that happens, I wish you luck as you enter full scale computer software programmer status. Let me know how it feels. I'll miss you when you're gone.

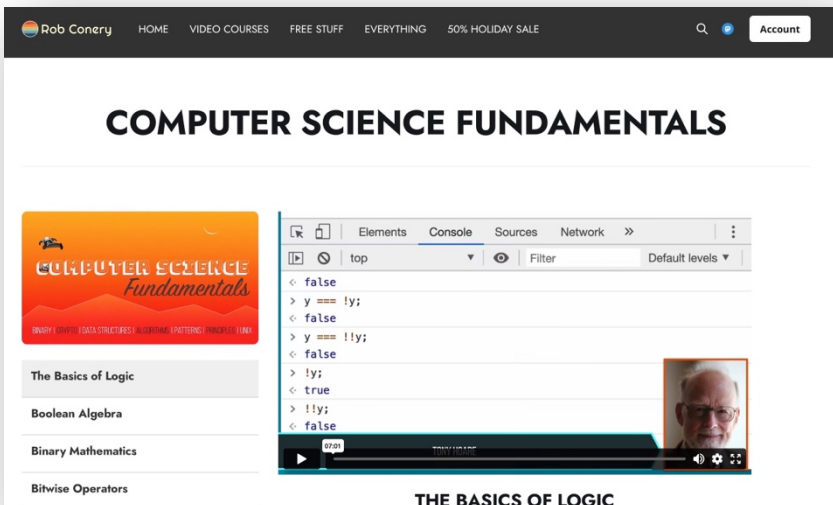
**Chad Fowler** August 12, 2016 Memphis, Tennessee

*Chad Fowler is the CTO of Wunderlist, which is now a part of Microsoft. He is also a Venture Capitalist with BlueYard Capital*

# RESOURCES

There's a lot of code in this book, and I want to make it easy for you to find it so you can follow along. I've pulled everything together in [this GitHub repository](#) so pull it down if you want to play along.

Also: I post [all kinds of things on my blog](#), including CS things I'm investigating. I also have video courses there, many of them based on this book!



# WALKTHROUGH VIDEOS

After I wrote the books, I spent a few months creating video walk-throughs for them. If you're interested, you subscribe to my blog and have access to every one of them *as well as* a bunch of other video courses that I've done. **Because you bought this book, I'm [offering a 50% discount too!](#)**

If you would rather just get the videos, that's fine too! You can access them [directly from here](#).

Finally, I **also have** [a mailing list](#) that I run through my blog. It's completely free and I don't send out shmaltzy marketing spam. Have a look at the posts and you'll see!

# PREFACE

The older versions of this book started off with an exhortation to dig deeper and do your best to inform yourself before arriving at that *strong opinion, weakly held*. I've had some time to think more about the idea, and I've decided to change the opening pages of this book entirely.

**Strong opinions weakly held** is just a euphemism for ***willful ignorance***. That, friends, is a strong opinion and no, I don't hold it weakly.

A strong opinion, by definition, is one you believe firmly in and one that you hold tightly to, for whatever reasons you have. It's *your* opinion, after all. If I can talk you out of this easily, perhaps with a mild comment or slight rebuff, what was the point of you having it in the first place? Were you too lazy to consider other points of view? Too isolated in your thinking? Yes, and yes, and probably yes to every other reason.

Strong opinions are **annoying**. I'll go one step further: they're ***toxic***. Let's sidestep politics, faith, and worldview and just focus on technology for now. Your reasons for having an opinion on anything should be exactly that: *reasons*. A thing you reason using facts, as well as you can assemble them. "I think semicolons aren't

optional” is an opinion and for many a strong one and hopefully we can agree that we aren’t all going to agree.

Taking that idea a step further, however, and claiming that someone else is wrong because of your strong opinion is where it all falls apart. Shielding yourself from criticism by telling someone else they can talk you out of it if only they made sense... that’s toxic, friend.

I think knowing what you’re talking about is much more convincing. Facts are sexy! That’s why I wrote this book and why I’m talking about all of this upfront: ***knowing what you’re talking about is always a good idea***. Education, what a drug.

I prefer to educate myself as much as I can because I loathe the idea that I’m defending my point of view based on what someone else said once or an article I read that told me what to think. If you’re reading this book I think we might both agree on that point!

Informing yourself *before* you reach an opinion is an indispensable skill if you want to advance in the tech industry. Don’t get me wrong, having a strong opinion is just fine (spaces instead of tabs, for instance) as long as you have solid reasons for your strong opinion. The keyword being *reason* instead of *feeling* or *belief*. Those are fine too, but tend to make technical discussions difficult indeed.

As we embark on this journey together, I would encourage you to embrace a different mantra: **informed opinions, happy to discuss**. As strong as your opinion is you should always leave the

door open to have your mind changed. It's a great first step to read up on and investigate your ideas as much as you can, but there really is no such thing as absolute truth; especially in the field of Computer Science.

Your experience will grow throughout your career and your knowledge will grow. **Let it.** Forming strong opinions stops this process from happening and your natural curiosity will ebb... and then you'll park yourself. This could be your happy spot, but eventually it will just be you and your opinions to keep you company.

Keep your mind open and ready. **There's so much to learn, and it's never, ever too late!**

— Rob Conery, *Hanalei, Hawaii 2021*

## CODE SAMPLES

I have published this book with every different kind of editorial trickery possible when it comes to code samples. I've used embedded styling, rich text, and images. If there are other ways I'm sure I'll use those too! Each has their drawback - for instance embedding styled text will invariably be overridden by whatever reader you're using and images are horrible for accessibility.

I think I have something that will work, however. The editing software I'm using will embed alt text in images for accessibility - so

that's what I've done. I want the code to look *precise*, indentations and all, for readability. No word wrapping, no font overrides.

I also want you to be able to play along so each code example (that merits it) will come with a link to either a Gist or a file up on GitHub. If you're wanting to play along, please do - but it will involve you clicking a link in the book.

## COVER PHOTO CREDIT

The image used for the cover of this book (and the inspiration for the splash image on the website) comes from NASA/JPL:



The image is entitled HD 40307g, which is a “super earth”:



*Twice as big in volume as the Earth, HD 40307g straddle the line between “Super Earth” and “mini-Neptune” and scientists aren’t sure if it has a rocky surface or one that’s buried beneath thick layers of gas and ice. One thing is certain, though: at eight times the Earth’s mass, its gravitational pull is much, much stronger.*

David Delgado, the creative director for this series of images describes the inspiration for HD 40307g’s groovy image:

*As we discussed ideas for a poster about super Earths – bigger planets, more massive, with more gravity – we asked, “Why would that be a cool place to visit? We saw an ad for people jumping off mountains in the Alps wearing squirrel suits, and it hit us that this could be a planet for thrill-seekers.*

When I saw this image for the first time I thought this is what it felt like when I learned to program. Just a wild rush of freakishly fun new stuff that was intellectually challenging while at the same time feeling relevant and meaningful. I get to build stuff? And have fun!?!? Sign me up!

This is also what this last year has been like writing this book. I can’t imagine a better image for the cover of this book. Many thanks to NASA/JPL for allowing reuse.

# LET'S TALK ABOUT IMPOSTER SYNDROME

**Y**ou might feel perfectly at home in your job as a programmer, or you may feel that you're fooling everyone and if you don't act quickly you'll be found out, canceled and your entire world blown to hell.

I used to feel this way. I *hated* it. Sometimes I find myself falling into the trap of "what the hell do you know, anyway? Who are you to be telling people about compilers and RSA encryption?"

Both of those are good questions that I think deserve an answer, but it all depends on how my inner voice decides to ask those questions.

My inner voice can be challenging.



robconery@mastodon.social ✓  
@robconery

...

Like everyone, I have an "inner critic" that I struggle with. Sometimes a bully, other times condescendingly surprised when something I do goes well.

Today I realized that inner voice has an Australian accent. I'm 100 serious. I realized it when I was making myself lunch...

5:00 PM · Apr 8, 2022

With apologies to my Aussie friends out there - I really don't think y'all are mean or anything. There's just something... so... *non-bullshitty* about the Aussie twang. Like you know it's telling you the brutal truth with a smile.

Anyway, I'm not a psychologist, believe it or not, but I have read a *ton* about emotional health and wellbeing over the years and I put a lot of what I read to work, and I would like to share that with you now.

## OVERCOMING IMPOSTER SYNDROME

I took some time last year to lay out the steps I took to move beyond my imposter syndrome. At first I thought that it just naturally ebbed as I learned new things but nope, that's not at all what happened.

What I found when I looked over my journal and notes from putting this book together (videos too) is a *self-reinforcing cycle*. I guess you could also call it a *resonant cycle* or just an *upward spiral* - but the idea is generally this: **one good thing led to another**, which pulled me out of the negative cycle of imposter syndrome.

Sounds easy, right? "Just get over it and think positive!" I really hate advice like that but, at its core, this is basically what happened to me although it happened in much smaller steps, which I'll share with you now.

I'll start with a little splash of cold water that's probably going to sting, but stay with me, it's important.

## **Imposter Syndrome is Toxic**

This is the stark truth. If you're convinced that you don't belong somewhere because you're fooling someone, *you* are causing problems for others. It's counterintuitive - how could you be causing problems if you see *yourself* as a problem?

The simple answer is this: **people are generally nice and care about you**. When you're upset, they want to help. If you keep

asking for help, they'll keep helping and this will lead to a lopsided social dynamic that lopsidedly revolves around you. This will eventually cause problems.

Here's a not-so-fun conversation along these lines:

**Me:** *Wow, I am in so far over my head... I don't know how I got hired here in the first place. I feel like studying for my interviews as hard as I did gave the entirely wrong impression of my skills.*

**Friend:** *they do this kind of thing every day Rob, I think they know when someone has talent and when they don't. You're fine, just focus on getting the work done.*

**Me:** *I'm hardly fine - they expect me to X and Y and somehow they think I know what I'm talking about. I really don't!*

**Friend:** *didn't you ship 2 applications last year with X and submit to PRs to the project, which were accepted?*

**Me:** *yeah well we all get lucky sometimes.*

**Friend:** *you sound like a broken record, get over it. You'll do fine.*

**Me.** *But I really think that –*

**Friend:** *ROB. Seriously, you're fine. Shut up.*

One of the worst things in life is to have someone pile on the pain when you're feeling exceptionally crappy. But when it comes to self-obsession (which imposter syndrome is), the only way out is through.

Thankfully, *through*, for us, simply means realizing what you're doing and seeing it for what it is: **a social problem**.

## Using Gratitude to Reverse The Suck

I'm sure you've heard this before: *be grateful! Write down 5 things you're grateful for and you'll feel so much better!* This never made sense to me. Give me a pint and I'll feel better!

The truth is: *it actually works*. But you have to mean it and feel it, and really dig into the deep well of happy feelings that are already sitting inside of you.

I decided to take "grateful breaks" whenever I felt crappy about work, life, or my future. I learned to recognize these foul moods and give myself 7 minutes (don't ask, it's the way my brain works) to ponder the **simple** things that I am truly grateful for:

- Something comfortable I'm wearing, like clean socks or new shoes.
- My kid's smile when she tells a funny joke.
- The fact that I get to write books like this which help people.
- The smell of freshly baked cookies.
- Chicken pot pies.

Yes, pretty mundane things to be grateful for, but that's actually the point, believe it or not. We don't spend enough time recognizing how fortunate we are, which is a powerful exercise.

The entire purpose here is to splash our insides with the golden glow of happy thoughts. *Your* goal should be to cause a *true smile* and if that smile comes from something work-related, even better.

**You know how to code!** That's a pretty amazing skill, I think. Even better, **you know how to type**, which a lot of people can't do! That means **you also know how to read**, which much of the world can't. You were also able to buy this book, which makes you pretty well off...

You get the idea. It's so easy to forget the advantages we have, especially in our industry! The jobs, the creativity, the people we meet and the innovation!

Ride that wave, friendo! It's a wonderful feeling and you can call it up any time you want, which is kind of a super power!

## **Unleashing Your Natural Curiosity**

When your gratitude flows it causes positive feeling and thinking, which can unblock so many things locked up in your brain. That little voice that tells you you're stupid or don't belong tends to grow quieter as you give yourself permission to *seek answers*.

This can cause confusion and disorientation, which will cause our happy wave to collapse, but that's OK! Being curious is so, so powerful and if you let it flow it will take you somewhere fun.

The trick is to *let it be there* and to give yourself the space to ask questions. I like to keep a list of things I'm curious about, along the lines of Richard Feynman's [12 Favorite Problems](#):

*"You have to keep a dozen of your favorite problems constantly present in your mind, although by and large they will lay in a dormant state. Every time you hear a new trick or a new result, test it against each of your twelve problems to see whether it helps. Every once in a while, there will be a hit, and people will say, 'How did he do it? He must be a genius!'"*

I'll talk more about writing things down in just a minute, as it's a critical part of this process. The main thing I want to get across to you is that your curiosity is critical to propelling you forward from this point.

In fact, I hope you're seeing that each step in this process builds on the previous! Stopping the negative spiral by seeing your toxicity is the start, and then reversing the process with gratitude is the second stage.

Using your natural curiosity as a fuel is the third (and very important) next step. Without this, it all falls apart.



## Removing Obstacles with Courage

I'm sure you have a friend who's told you to "just go out there and..." or "you just need to tell them how you feel" or maybe "you'll be fine, you just need to learn to stand up for yourself."

Feh.

This is easy advice and also easily useless. How do you just "become courageous"? After all, that's what our friend is telling us when they say "just stand up for yourself".

The answer, at least to me, is that *you just need a good reason*. We've all had those moments where we say or do something and surprise ourselves (for better or worse) so we just need to find one of those (a good reason).

But where?

I'll give you one: ***you don't want to lose this happy momentum***. If you actually do the process I'm suggesting (toxic realization, gratitude, letting your curiosity out) you'll start to feel a surge of happy creativity. If you stop now, it means descending back into the Pit of Sucky Self-obsession that no one wants to be around.

But here's the thing: *it's doesn't require a ton of courage to begin with, and it's mostly dealing with just yourself*. Your only task is to remove the stupid obstacles in your own mind that are stopping you from *finding out*.

Your natural curiosity wants to know stuff! LET IT. You can trust yourself by courageously removing the blockers in your own head. I told myself that I wanted to learn all the things in the MIT CompSci curriculum and my inner voice instantly thought that was nuts! I had more important things to do, like finish my next video for Pluralsight (whom I was working for at the time).

My inner voice sounded something like this:

*More videos, more money mate. Simple equation dumbass! Get back to work you daft twat!*

Maybe this is why my inner voice sounds Australian to me (though this last was a bit more Birmingham) - I find it easier to laugh at and ignore.

Your instincts are there to guide and help you. You would be surprised at how often they can help you find something wonderful as opposed to fear, which almost always kickstarts the downward spiral into self-obsession and self-loathing.

The only way to know if your instincts are serving you is to actually let them do it. Give yourself the space to *try something new* or to go completely off the rails for a little bit, following your muse.

Fight the tendency to see this as permission to screw off all day - it's not. You have a powerful pattern recognition machine behind your intelligent, reasoning mind that is exponentially more powerful. When you don't give it exercise, it causes you problems.

Like self-loathing and feeling like an imposter. Let the giant wake up and trust that it will serve you well.

## **Journaling and Taking Notes**

This process becomes real when you write it down. Having dreams and ideas is wonderful, but when you put it in ink and describe your goal, it becomes a commitment.

If you've given yourself permission to go where your curiosity takes you, **document the journey**. I cannot stress this enough! I use both a journal and a note-taking app and they have been absolutely critical to this process.

In fact, when I'm feeling doubtful or my inner voice is getting the better of me, I'll write myself little bits of encouragement. The best are my stick figure doodle friends that tell me I'm OK and doing good stuff.

If you haven't journaled before, I would encourage you to have a look on YouTube and watch a few videos on Bullet Journals. You'll be awash in people showing off how beautiful their illustrations are, but that's kind of the point. Creating a beautiful space for your mind to wander and feel settled. I can't draw worth a damn but I keep trying! There's no judgement in there and I like it when something actually *does* come together and look nice.

For notes, I use [Obsidian](#) and I love it. There are loads of apps and also loads of processes for using these apps and I'll leave that to you to figure out what works best.

I filter the notes I take, writing down only the things that “stick” or resonate in my head. I go back and look through them from time to time, and when they relate it gets magical!

Anyway: get the stuff out of your head and on paper somehow so you can see your progress and *feel* the reality of it. It really does fill your soul!

## Sharing With Others

If you've made it this far in your journey, hopefully you're finding each step getting easier. The momentum you build in your upward journey reinforces itself, in the same way imposter syndrome reinforces itself through *negative* resonance.

Good news friendo: *it only gets better from here*. If you've unleashed your curiosity, removed obstacles courageously AND have documented your journey, **you're ready to share!**

This might be hard to read if you're still in the grips of imposter syndrome, but believe me, it's so, so wonderful!

*Sharing what you've learned is a wonderful gift.*

You might think: “who, me? What do I know? I'm no expert on X, why in the world would I tell anyone else about it!” I think these

concerns are valid, especially in the hyper-stupid world of stupid stupid social stupidity where everyone shares everything and it can be extraordinarily ... stupid.

That's not what I'm talking about. Consider this quote from my friend Derek Sivers:

*You should share something you've learned right when you learn it so you remember what not knowing it was like.*

I so, so love that. Look: I'm not a psychologist or spiritual guru. I'm not a consciousness expert or therapist of any kind, yet here I am, sharing a profoundly personal experience with you that changed my life. **I don't need to be an expert** to share this!

That's the key: you've taken the time to discover and learn something, fueled by curiosity and courage. You've written down what you've learned and considered it at length. *You've learned something.*

Boom. Share it with others and change someone's life. How fun is that! The trick, here, is not to tell someone what to do but, instead, share with them how you see the problem, the solutions you considered, the ways in which you failed, and what ultimately worked for you. Invite feedback and discussion - and everyone wins.

Now it goes without saying that there are ego traps here. It's easy to figure something out and have someone refer to you as an

expert, which is never fun. Someone once called me an expert on imposter syndrome and I think I screamed something horrible.

People that *do* fall into this ego trap become *Expert Beginners*, the poster children of Dunning-Kreuger syndrome. It's human, I suppose, but do be aware of these traps. Becoming an expert at anything takes a lot of experience, which means a lot of failures, and that doesn't happen quickly.

Take your time getting to the top of the mountain but, remember, the true joy is in the climb.

## **The Final Step: Recognition**

So here we are. You've reversed that horrible, toxic imposter nonsense and you've pulled yourself up, learning amazing things and setting up plans for a brighter career!

That's the goal, anyway. It does take time, which is where this book comes in.

My goal here is to wedge this book in right above the second stage *gratitude*. I can't stop you from being toxic, you need to do that yourself. I also can't help you with becoming curious, that's something only you can do.

I *can* fire that curiosity, however, and that's my plan.

I want to help push you along the wonderful course your career is about to take. You just need to supply the courage, notes and,

hopefully, the sharing. If you do these things, I just have one request: *stop and notice*.

At some point good things will start happening to you professionally. It's a good practice to recognize these things by writing them down or celebrating with friends and family. It could be as simple as getting a promotion or as dramatic as having the courage to leave that dead-end job.

Maybe you finally understand *P* vs. *NP* (that took me a while!) The point is: **recognize and be grateful for how far you've come.**

Yes, there's that word again: *grateful*. You'll find it's akin to pure magic when it comes to transforming yourself and your work. What's even better is that the people around you will feel it too, and everyone benefits.

Right then! Hopefully you feel mentally prepared because WOW do we have some fun things to learn. Get yourself to stage two, *friendo*, and let the curiosity flow because we're going to hit the ground running.

Here we go...





# THE BASICS OF LOGIC

**E**very 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!

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.

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*.

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 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 a 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 wonderful 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?*

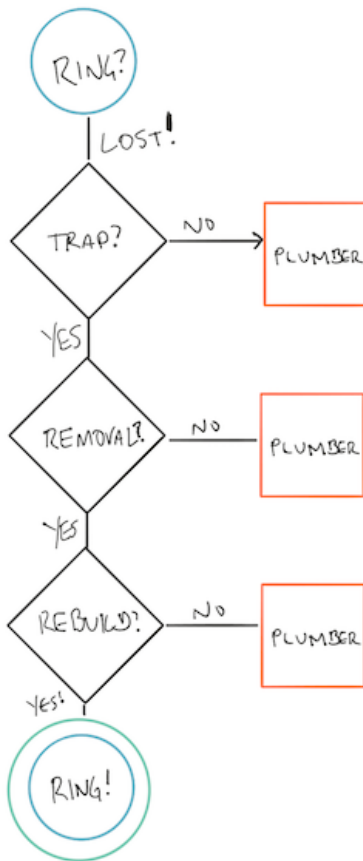
*I am not a plumber and I have no interest in being a plumber... I choose to put a limit on how much I know about plumbing.*

*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 (ca. 20 cm) 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 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 execu-

tive 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*:

1. **Identity:** A statement is its 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 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 identity and their truth values.
2. **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.
3. **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.

# BOOLEAN ALGEBRA

**G**eorge Boole was one of the greatest unknown mathematicians whoever lived, aside from a data type that bears his name. 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.

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.



# AN EVENING WALK

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



*On a walk with the family down the Seattle streets on Christmas*

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 many things on my 1.2-mile walk. The quiet of the night is my audience as I would mutter about the day's events or, more positively, about something I'm trying to figure out for work. This book featured heavily in those walks.

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. 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

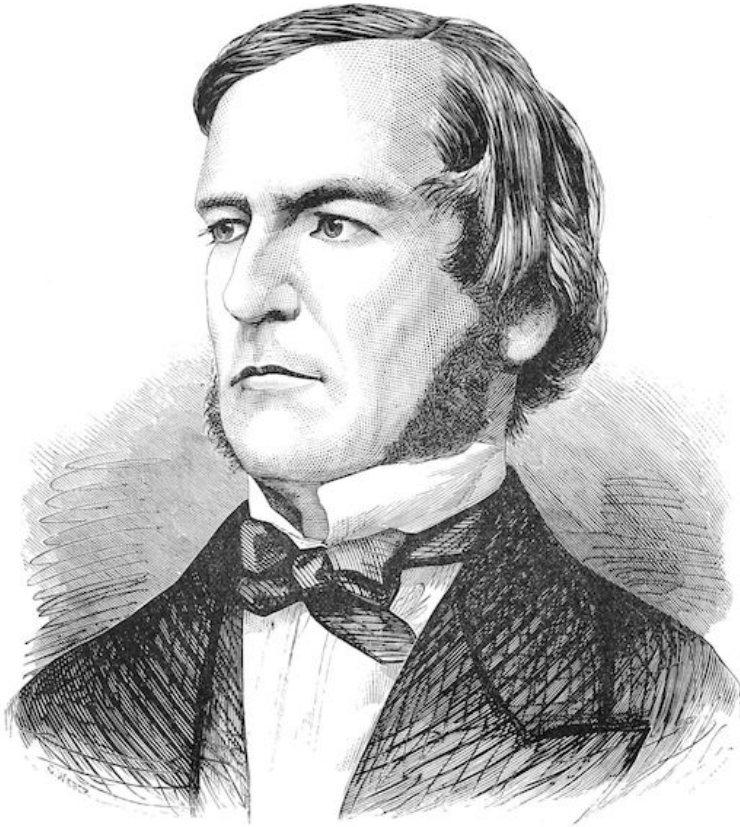
Between 1845 and 1847 quite a few Irish people starved due to a blight on potato crops. This bit of history is well known, but most people just mention it and move on with whatever historical tidbit they're about to share. Have you ever wondered, however, why a lack of potatoes would cause over a million 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.



*George Boole*

George Boole was absolutely obsessed with proving, through purely logical means, the existence of God, a pastime for religious aca-

demics 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 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 "imposter", 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 brain 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");
}
```

*A JavaScript statement using an AND operator*

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 them-

selves may at some levels 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...*

## 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:

1. **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.
2. **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.
3. **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 || B** represents an OR operation
- **A ! B** is a NOT, or inversion operation

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 (v, 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 ( $A \vee B$ ) with the additional rule that A cannot equal B. We can string together Boolean operations to achieve this:

$$A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$$

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 (!A) 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: (!A or B).



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, that satisfies the 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.

Understanding this, we can now do some math. Consider the following:

$$1 + 1 = 1$$

$$1 + 0 = 1$$

$$0 + 0 = 0$$

$$0 + 1 = 1$$

This is Boolean addition, and you can think about this as an OR statement and represents a truth table for OR.

What about subtraction? It might be tempting to think a (-1) would be equivalent to false, but it's not. It doesn't exist in Boolean Algebra! The reason? A negative truth is simply false, or (0).

That means there is no subtraction and, therefore, no division (which is compounded subtraction). There *is*, however, multiplication:

$$1 * 1 = 1$$

$$1 * 0 = 0$$

$$0 * 0 = 0$$

$$0 * 1 = 0$$

This is equivalent to an AND statement.

# 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:

1. To expand on our understanding of logic and to see how it can be treated mathematically.
2. 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
3. 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 in his Master's Thesis, that kicked off the Information Age.

# COMPUTATIONAL MODELS

**C**harles Babbage, Ada Lovelace, Alonzo Church and Alan Turing (all of whom we'll get to know in the coming chapters) laid the foundation for how a machine could be used to compute things, but how did this come about? The abstract notions of machinery and computation led engineers like Jon von Neumann, JP Eckert and John Mauchly to design and build the very first electronic computer. Every computer that exists today is based on these abstractions and designs.

Over the years, the notion of an abstract ability to compute things has taken on many forms. Let's visit that now, starting off with a little history, once again. We'll visit Plato and ponder the true nature of things, drop in on Bernoulli in the 1500s and wind our way to Russia in the early 1900s. We'll visit Bletchley Park and Alan Turing in the early 20th century, eventually ending up back in the United States with John von Neumann, the creator of the modern computer.

# PROBABILITY AND THE THEORY OF FORMS

At some time around 400 BC, Plato mused that the world we see and experience is only a partial representation of its true form. In other words: an abstraction. The real world is either hidden from us, or we're unable to perceive it.

He based this notion on his observations of nature: that there is a symmetry to the world, to our very existence, that lies just beyond our ability to fully perceive it. Phi, the Golden Ratio,  $\pi$ , and  $e$  are examples of some cosmic machinery that is just outside our grasp.

To many, the natural world appears to be a collection of random events, colliding and separating with no guiding purpose. To a mathematician, however, these random events will converge on an apparent truth if we simply study them for a long enough time.

The Italian mathematician Gerolamo Cardano suggested that statistical calculations become more accurate the longer you run them. Jacob Bernoulli proved this in 1713 when he announced The Law of Large Numbers in his publication *Ars Conjectandi* which was published after he died.

This law has two variations (weak and strong) – but you can think of it this way: if you flip a coin long enough, the statistical average will come closer and closer to 50% heads, 50% tails. This might

seem obvious – after all there are only two sides to a coin and why wouldn't it be a fifty-fifty distribution?

The simple answer is that reality tends to do its thing most of the time, with apparent disregard for our mathematical postulations. The fact that we can completely rely on statistical models over a long enough period is astounding.

This is what keeps casinos in business. All they must do is to make sure the mathematical odds of each of their games is in their favor, and over time the money they make on those games will reflect those odds increasingly closely. It doesn't matter if you walk in tomorrow and win all their money – statistics says they will get it back if they wait long enough.

Flipping a coin, however, is just a single event. Playing a game of craps or a hand of blackjack consists of a series of events, one dependent on the next. Does the law of large numbers still hold?

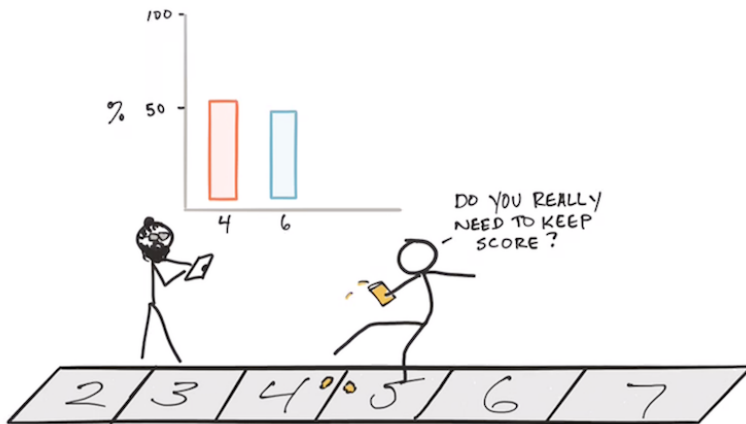
## MARKOV CHAINS

The Russian mathematician Andrey Markov said yes and set about to prove it in the early 1900s with what has become known as the Markov chain.

If you've ever used a flow chart, then you'll recognize a Markov chain. It is usually described graphically as a set of states with rules that allow transition between those states:

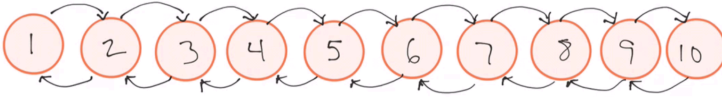


A famous Markov chain is the so-called “drunkard’s walk”, a random walk on the number line where, at each step, the position may change by  $+1$  or  $-1$  with equal probability. From any position there are two possible transitions, to the next or previous integer. The transition probabilities depend only on the current position, not on the manner in which the position was reached. For example, the transition probabilities from 5 to 4 and 5 to 6 are both 0.5, and all other transition probabilities from 5 are 0. These probabilities are independent of whether the system was previously in 4 or 6.



A Markov chain looks a bit like a flow chart, and in many ways that’s exactly what it is: a probability graph of related events. This fits nicely with the notion of an abstract computing machine.

# DRUNK MARKOV CHAIN



In the above diagram, each orange circle is a “state” and the arrows dictate transitions possible between each state. Except for 1 and 10, the only transitions possible at any state along the chain is the next direct state or the one prior. Notice also that none of the states allows for a “loop back”, or a transition back to itself.

This diagram represents a very simple abstract computation, or a “machine”.

## FINITE STATE MACHINE

If we focus only on the notion of state and transitions, we can turn a Markov chain into a computational device called a Finite State Machine (or Finite Automata).

A simple machine does simple work for us. A hammer swung in the hand and striking a nail translates various forces into striking power, which drives a nail into wood. It doesn't get much simpler than that.

The nail, as it is hit, moves through various states, from outside the wood to partially inside the wood to fully inside the wood. The nail transitions from state to state based on actions imparted to it by the hammer.

The action of the hammer transitioning the nail through various states constitute a state machine in the simplest sense.

## **Deterministic**

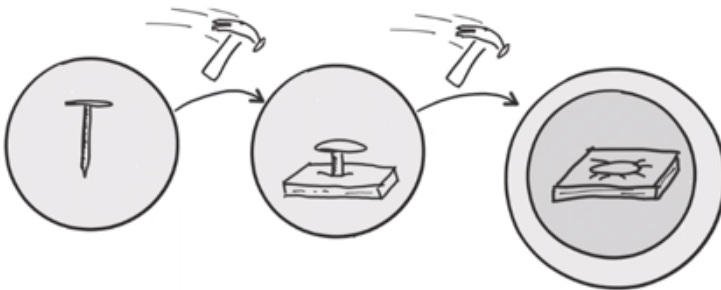
We discussed determinism a few chapters back, and we also discussed Finite State Machines a little. Let's go a bit deeper.

A Finite State Machine (FSM) is an abstract, mathematical model of computation that either accepts or rejects something. We can apply this abstraction to our hammer and nail process above, so we can describe things mathematically. We start by diagramming the state of the nail:

- Outside the wood is the starting state
- Partially driven into the wood is another state
- Fully driven into the wood is a final state, also called acceptance
- A bent nail is also a final state, but not the state we want so it is a rejection state



We can relate these states together through various transitions or actions, which is the striking of the hammer:



This is a deterministic finite state machine, which means that every state has a single transition for every action until we finally reach acceptance or rejection. In other words: when we hit the nail, it will go into the wood until its all the way in, which is our accepted state (denoted by a double circle) – there is no other course of action.

Being a good programmer (which I'm sure you are), you're probably starting to poke holes in this diagram. That's exactly what you should be doing! It's why these things exist!

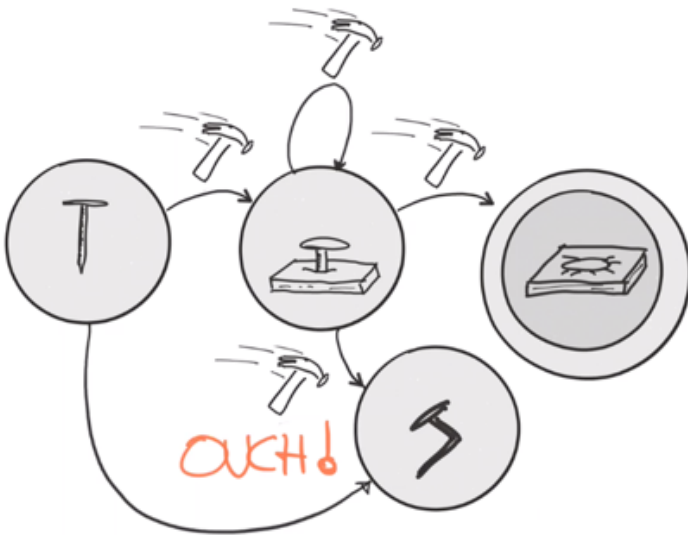
Our FSM only describes moving the nail through three states with only three actions. Furthermore, there is no rejection state for when I partially hit the nail and bend it. How do we account for that?

## **Nondeterministic**

A non-deterministic finite state machine has one or more transitions for a set of actions, and these transitions are essentially "random". A nondeterministic machine is capable of transitioning from one state to the next as a result of some random choice, completely independent of a prior state.

As a programmer, you can think of a nondeterministic machine as a program that will produce different results when run multiple times – even with the same input given.

Let's update our FSM to be nondeterministic by introducing partial hits which can sometimes bend the nail:



Here, I have two possible transitions for the initial state: I hit the nail, or I miss and bend the nail (yelling loudly when I do). I've added an action on the second state as well – the one that loops back to itself. This is the action taken when the nail has not been fully driven into the wood.

The conditional step of hitting the nail multiple times is deterministic, however the random step of bending the nail is not as we don't know if it will happen before we start hammering. Sort of. There are all kinds of variables and probabilities at play here from muscle twitches to fatigue, hammer integrity and wind direction. It will all come together at some point and the probabilities are so out there that we might as well consider this to be a random event.

## Alphabets and Language

Let's move away from hammers and into the world of code and machine processes. You'll often hear people talking about FSMs working over a particular alphabet, string, or language. This is particularly relevant in computer science.

At the lowest level, our alphabet is a bunch of 1s and 0s – bits and bytes on a disk or in memory that we need to understand. This is how computer science people thought about the world decades ago: as holes on a piece of paper fed into a machine.

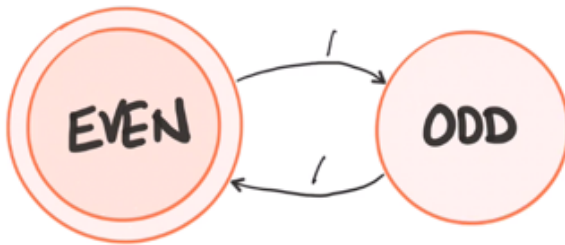
Let's do another FSM, but this time we'll do some real computing. Let's take a string of bits and write a routine that will tell us if there is an even number of 1s in the supplied string.

We'll start out with the two possible states:

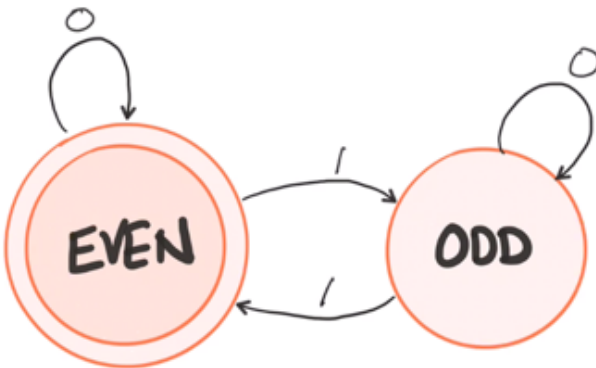


The double-circle here is our accept state, and if that's also our final state we have acceptance. Now we need to account for the action of reading each character in our string, one at a time.

If we read a 1, we'll have an odd number of ones. If we read a 1 while in the odd state, we'll transition to the even state. Another 1 and we're back to the odd state:



What happens if we read in a 0? We do nothing and just keep on reading:





## Limitations

Here is the part where we come to the naming of things. An FSM relies on a simple, known or finite set of conditions. The inputs are 1s or 0s, we have 1Mb of RAM to work with, or maybe 10Mb of hard drive space – these are finite conditions and tend to describe simpler constructs.

Streetlights, alarm clocks, vending machines – these are perfectly good Finite State Machines. An iPhone is not.

To understand this, think about our 1s and 0s example above. What if we wanted to sum all the 1s and then divide that by the sum of all the 0s to get a randomization factor? In short: we can't describe this with an FSM. There is no notion of a summing operation.

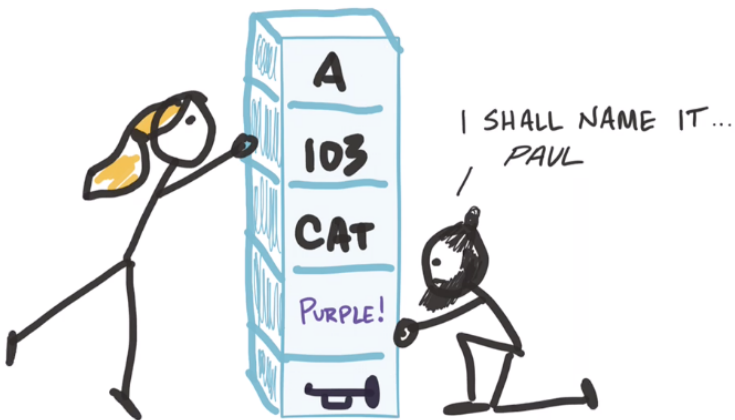
The basic problem is this: *you can't store state outside the current state*. In other words: there is no persistence, or RAM.

We can't calculate the digits of pi, or perform map/reduce operations. We can, however, parse Regular Expressions (ugh).

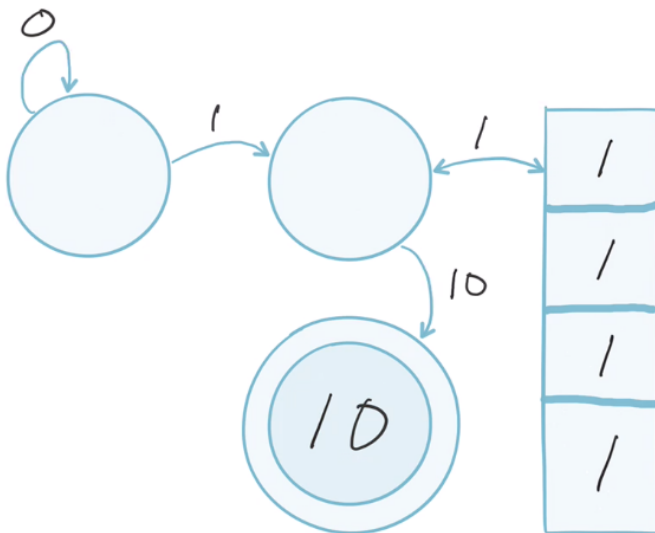
Much of the world runs on FSMs, but for what you and I do to make a living – we need a computational model that allows for a little more complexity.

# PUSHDOWN MACHINE

If we want to sum things with our computation, we need to move away from the concept of an FSM and add computational power to our machine. To sum things, our machine will at the very least need to remember the last state. We can add a facility for this called a stack.



A stack is quite literally that: a stack of data that you can add to (push) or remove from (pop). If we alter our FSM to have a stack, we can now compute a whole new set of problems:



This, however, is no longer a Finite State Machine – it's called a Pushdown Machine (also called Pushdown Automata or PDA). This machine has a lot more computational power because the state transitions are decided by three things working together:

- The input symbol (a 1 or a 0)
- The current state
- The stack symbol

With this new power we can compute running totals of 1s, creating a summing operation, and a lot more.

The notion of a stack is very powerful, but it's also a bit limited. You can only compute something that the space on the stack can han-

dle. In other words, if we had 10 possible states and only 5 slots in our stack and we wanted to run a summing operation as before, we could easily run out of memory and our application would crash.

The Pushdown Machine is bounded by its stack and is, therefore, limited in terms of what it can compute. But what if the stack didn't have a limit?

This problem was solved by Alan Turing.

## TURING'S MACHINE

In the mid-1930s, Alan Turing published what has become one of the cornerstones of computational theory as well as computer science in general: his paper, entitled *On Computable Numbers*.

Written when he was just 24 years old, *On Computable Numbers* described a computational process in which you stripped away every "convenience" and introduced a machine with a read/write head and some tape. The tape has a set of cells, each of which holds a simple symbol of some kind, and you can move that tape under the head so the machine could read from it, or write to it:

*We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions  $q_1$ :  $q_2$ . ....  $q_l$ ; which will be called "m-configurations". The machine is supplied with a "tape*

*"(the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". At any moment there is just one square, say the  $r$ -th, bearing the symbol  $\varepsilon(r)$  which is "in the machine". We may call this square the "scanned square ". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its  $m$ -configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the  $m$ -configuration  $q_n$  and the scanned symbol  $\varepsilon(r)$ . This pair  $q_n, \varepsilon(r)$  will be called the "configuration": thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the  $m$ -configuration may be changed... It is my contention that these operations include all those which are used in the computation of a number.*

Each "machine" that Turing describes is designed to take another machine as its input, which allows for massive computational power. Alonzo Church had the same notion with Lambda Calculus: functions that take other functions as arguments. This flexible structure is the foundation of modern computer science.

We can write a Turing Machine (using mathematical notation) to describe the number 4 (M4). We can write another machine to describe the number 6 (M6) – and yet another to perform multiplication on a set of numbers (MX).

We can then write a final Turing Machine that accepts M4 and M6, and uses MX to run the multiplication. This is central to Turing's idea: small, concise machines orchestrated to derive a final result. 1s and 0s are all we need to describe and compute any problem we can describe to the machine.

This led Turing to claim something rather extraordinary:

*If an algorithm is computable, a Turing Machine can compute it*

This is another way of stating what Alonzo Church asserted two years prior:

*All total functions are computable*

This became known as the **Church-Turing Conjecture**, and was an extraordinary claim for the time. Prior to the idea that machines could calculate something for us, mathematicians would declare that a given problem was effectively calculable if someone could sit down and figure it out with a pencil and some paper.

Church and Turing, however, gave us a different way to run calculations and, therefore, a different way to think about what we can compute, in general. Put in a simpler way: **if you can describe it**

**with a Turing machine or with a set of functions, it can be computed.**

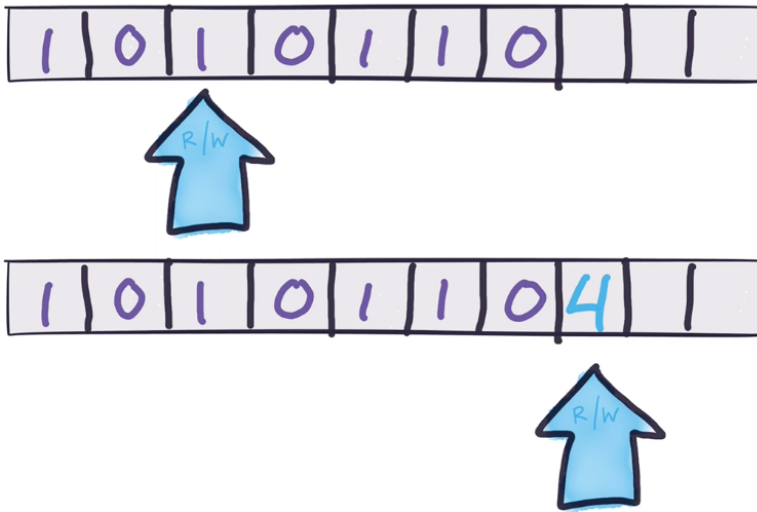
## **Basics**

A Turing machine is an abstract computational device that can compute anything that you can program. It's important to note that this is not supposed to be a real machine although some have made one.

It's an incredibly simple construct that can simulate any computational algorithm, no matter how complex.

A Turing Machine has four main parts:

- A set of symbols, defined in an alphabet or language the machine can understand (usually just 0s and 1s)
- An infinitely long "tape" which contains a single row of symbols that can be anything
- A read/write head that reads from the tape and writes back to it
- Rules for reading and writing from the tape



If you create an instruction set that is capable of running on a Turing machine, it is said to be Turing Complete. All you need for Turing-completeness is:

- Conditional branching
- Loops
- Variables and memory

## Stripping To The Essence

Turing was trying to create a model of computation that could, essentially, scale to any problem thrown at it. A Turing machine has more computational power than the machines previously discussed because a Turing machine has an infinite amount of stor-

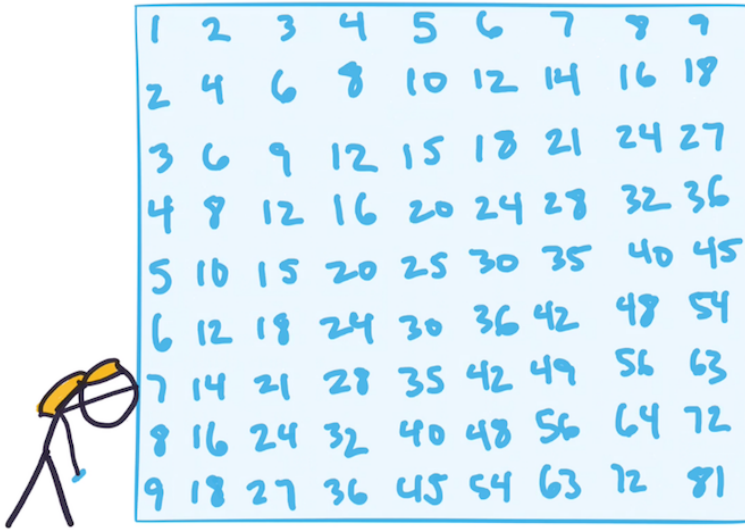


age (in the form of a “tape”) to read from and to store state to. Now, being good computer scientists, we might get stuck on the notion of an “infinite” bit of tape. Such a thing might be a bit hard to come by in the real world.

Believe it or not, it doesn’t matter; this is a conceptual machine after all. The amount of tape use, the speed of the tape through the machine and/or the symbolic alphabet that you choose to use with it – none of these variables effect the overall computational power of a Turing machine. As we learned last chapter: *if you can describe it with a Turing machine, it's computable.*

It’s a wonderful computational model, and it’s derived from the idea of stripping the notion of computation itself to the barest minimum, as Turing further described in his paper:

*Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares.*



## Reading and Writing

Given that a Turing Machine can write to, as well as read from, a tape full of symbols, we now can store an “infinite” amount of information (assuming we have an infinite tape).

The term *infinite* is used often when discussing Turing Machines; don’t get hung up on it. If you need to, translate it as “just enough”. In other words, your computer can store massive amounts of information, but someday it will fill up. At that point you can go get another hard drive, and someday you’ll fill that up. But then you’ll get another drive ... the space your machine has on disk has nothing to do with its computing power – same with the size of RAM.

These things do have a lot to do with how long you'll have to wait, however.

State, in a Turing Machine, is stored on the tape that it is given to read from:

*The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound  $B$  to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations.*

The "tape" is being observed by moving under what Turing called a "head", something that can both read and write. As the tape moves, the machine observes the symbols and the "state of mind" of the machine changes because the information within changes.

By the way: it's quite fun to see Turing's treatment of the machine in human terms.

*We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an "m-configuration" of the machine.*

Multiple machines, multiple configurations – all usable by other machines with different configurations: this is the essence of the Turing machine.

## Universal Turing Machine

One very interesting feature of a Turing Machine is that it can read in and simulate another Turing machine, even a copy of itself. In the 1930s, when *On Computable Numbers* was written, machines were built for a specific purpose. A drill made holes, an elevator lifted things, etc. so it made sense that computing machines would be purpose-built in the same way.

Turing proposed something altogether different. His abstract machine could simulate any other abstract machine – making it a universal computational device.

This idea had quite a profound effect, as you can imagine. The question of what *can we solve* was no longer interesting – with the Church-Turing Conjecture the question evolved to something a bit more profound: what can't we solve?

In a wonderful bit of cheek, Turing provided the answer in the very same paper he used to introduce his machine.

## THE VON NEUMANN MACHINE

We started this book by thinking about complexity and computation in general. We then got a little steampunk and explored the early 1800s and Babbage's Analytical Engine – only to find out that he and Ada Lovelace had designed the first Turing-complete language out of punch cards.

We explored Turing machines in this chapter and got to know both the Church-Turing Conjecture and its counterpart: *The Halting Problem*.

We have a couple of ways of computing anything that is computable – but we’re still stuck in the land of theory. This changed in the early 1940s with John von Neumann, JP Eckert and John Mauchly.

Oh, and Turing too, as it turns out.

## **That Tape Can Hold a Lot More**

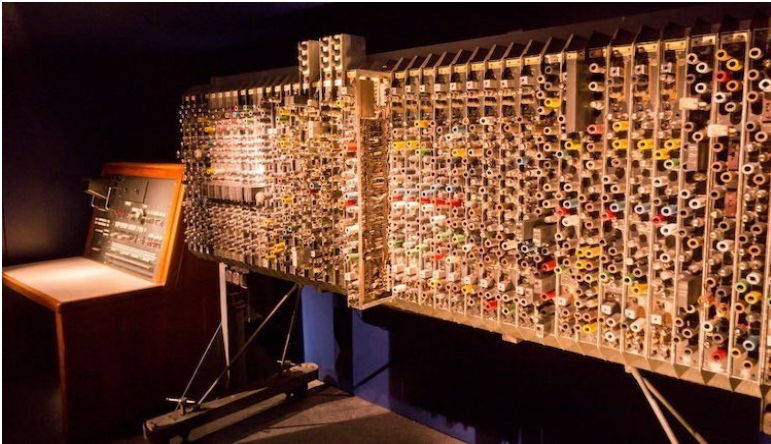
We know that a Turing machine works on a very simple idea: a read/write head and an infinite supply of tape. The machine has some type of instruction set that influences the current state of the machine. It’s simple enough to conceive of a tape and a read/write head – but what about that instruction set? Where does it live?

Initially, the idea of producing a working computer was that you would produce it for a specific purpose – maybe a calculator or prime number generator – a dedicated machine hard-wired to do a dedicated process. This seemed to defeat the notion of a “universal computing device”, but the hardware simply wasn’t there yet.

At almost the same time, Turing and von Neumann were plotting to fix this. They both came up with the same answer: make the instructions themselves part of the tape. This is what we call a program today.

# THE AUTOMATIC COMPUTING ENGINE

In 1946, just after the war, Turing designed and submitted plans to the National Physics Laboratory (NPL) for an electronic “general purpose calculator” called the Automatic Computing Engine – using the term Engine in homage to Charles Babbage. A number of these machines were created, and they stored their processing instructions as part of the machine’s data.



*Turing's ACE Machine. Image credit: Antoine Taveneaux. You can see this on display at the London Science Museum.*

In an interesting historical side note: the NPL wasn't sure that creating such a machine would work. They were unaware of Turing's work during the war, where he worked with an electronic comput-

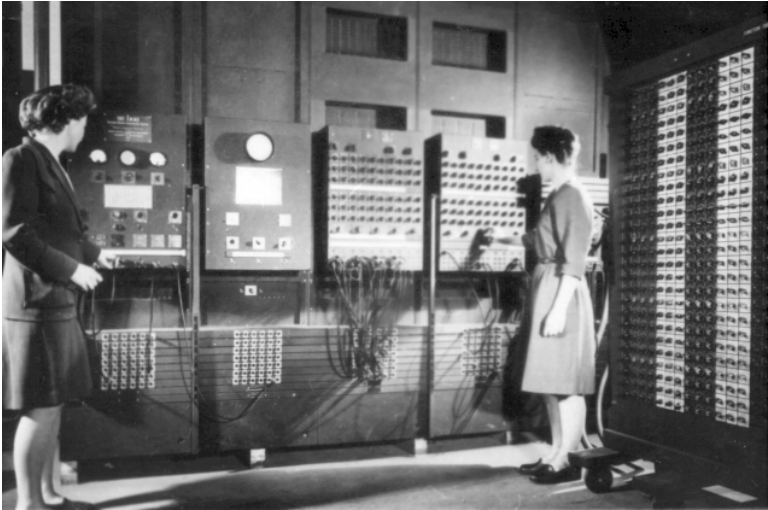
er every day (Colossus) to help break Nazi ciphers. He couldn't tell them this, however, as he was sworn to secrecy.

## ENIAC AND EDVAC

In the early 1940s JP Eckert and John Mauchly were designing ENIAC, the world's first electronic computer. Governments were realizing quickly that computers would give them the edge during wartime, and the race was on.

The existence of Colossus, the machine that helped crack German codes during World War II – is known today. Back then, however, this was a big secret that no one outside of Blechley Park knew about. I bring this up because the initial designs for computers, underway with various science teams around the globe, were done in secret and without much sharing.

ENIAC, for instance, was developed to calculate ballistic trajectories and other top-secret calculations. It was Turing-complete and programmable, but to program it you had to physically move wires around.



### *ENIAC Operators*

In 1943 Eckert and Mauchly decided to improve this design by storing the instruction set as a program that was stored next to the data itself. This design was called the EDVAC.

By this time the Manhattan Project (the US Atomic Bomb project) was rolling, and von Neumann, being a member of the project, needed computing power.

He took interest in ENIAC and later EDVAC which led him to write a report about the project, detailing the idea of a stored-program machine. This is where things get controversial. For one reason or another – maybe because the paper was an initial draft – von Neumann's name was the only name on the report, even though it was mainly Eckert and Mauchly's idea.



As well as Turing's – but he couldn't tell them that because it was still a secret.

This gets further complicated by a colleague of von Neumann's, who decided to circulate the first draft to other scientists. Soft publishing it, if you will. These scientists got very interested in the paper and ... well ... we now have the generally incorrect attribution of stored programs to von Neumann.

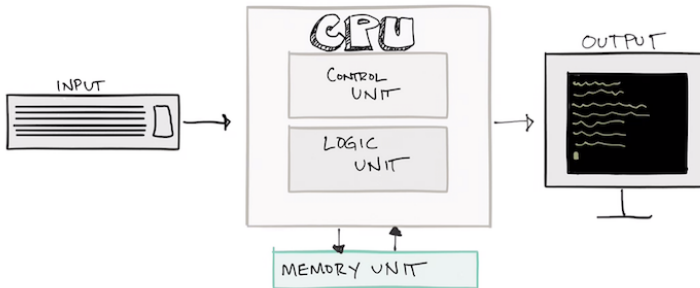
One of von Neumann's colleagues, Stan Frankel is quoted as saying:

*Many people have acclaimed von Neumann as the "father of the computer" (in a modern sense of the term) but I am sure that he would never have made that mistake himself. He might well be called the midwife, perhaps, but he firmly emphasized to me, and to others I am sure, that the fundamental conception is owing to Turing— in so far as not anticipated by Babbage... Both Turing and von Neumann, of course, also made substantial contributions to the "reduction to practice" of these concepts but I would not regard these as comparable in importance with the introduction and explication of the concept of a computer able to store in its memory its program of activities and of modifying that program in the course of these activities.*

As you might imagine, Eckert and Mauchly were not happy about this. Either way, this new idea now has a name.

# VON NEUMANN ARCHITECTURE

Historical silliness aside, we have now arrived at a modern-day computer and the birth of modern computer science. von Neumann's machine should look reasonably familiar:



## *von Neumann Architecture*

This machine architecture allows us to create, store, and deliver functions as a set of symbols. Thinking about this in depth can twist your brain a little.

A Turing machine is an abstract construct and the initial attempts to create one ended up with rather large bits of machinery that needed to be fiddled with by hand to run a given calculation. von Neumann could abstract that fiddling, which meant that Turing Machines could be built with nothing more than code and Turing's original vision of a "Universal Turing Machine" was complete.

To grasp this in a more real sense – think of a pocket calculator.



*It does only one thing:* math. Now open the calculator on your computer.



### *My Mac's Calculator.*

It does exactly the same thing as the calculator above, but there's no hardware involved. I can also change it to do scientific and programming calculations on the fly, which is kind of magical. **The Mac calculator is as much of a machine as the one you hold in your hand**, but it's made of nothing more than pixels.

You can thank Mauchly, Eckert and von Neumann for this. Machines running within machines. When you consider various code execution environments (like the JVM, the CLR, or Google's V8 Engine) that are running on virtualized machines in the cloud this whole idea tends to start spiraling.

How many abstract machines are involved to execute the code you're writing today? When you run a VM or Docker (or some container functionality of your choice), these are machines within machines executing machines made up of smaller machines within other machines...

It's machines all the way down.

# MECHANICAL COMPUTERS

**T**he 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.

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. Before 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 electrical circuits replaced the mechanical bits.

## MATH AND WAR

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 giant war.

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 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 brilliant 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

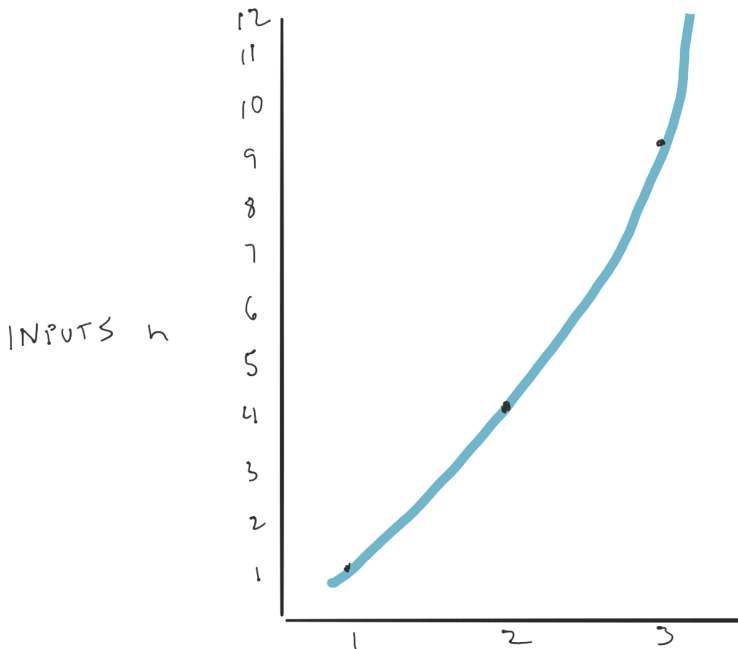
That better way was the focus of universities across the United States and England (not to mention Konrad Zuse's skunkworks projects in Nazi Germany and Sergey Lebedev's experiments with computing differential equations in the Soviet Union) in the late 1930s and early 1940s. This story could be an entire 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.



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 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 . 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, "Con-ery! We're dead unless you can calculate the square root of 834.22!", I would have a few choices:

1. Pull out a big book of precomputed tables and, as quickly as I could, try to find the answer while praying it's correct
2. Scavenge up a pencil, a big sheet of paper and some drafting tools so I could draw a line representing a square root calcu-

lation. 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

3. 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 second self 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.

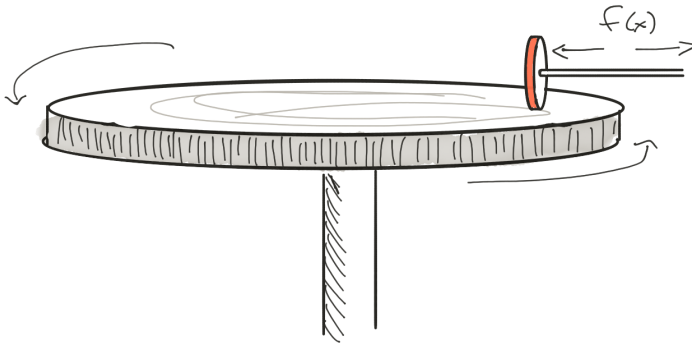
This is what mechanical computers did: compute *curves* that could be used to plot the numbers you need when you need them.

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  $y$ :

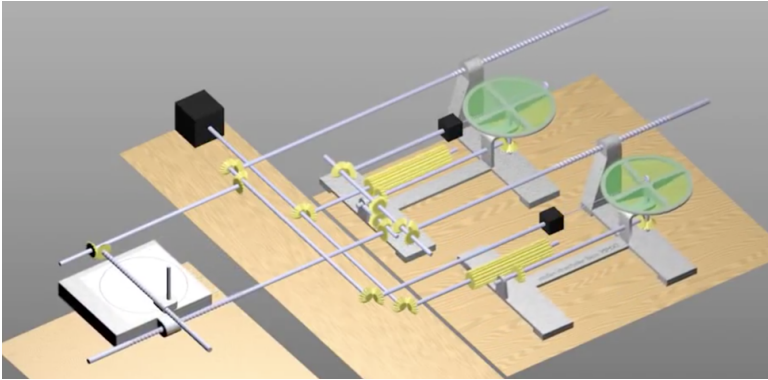
$$(x - h)^2 + (y - h)^2 = r^2$$

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 the 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 precisely 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 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 achieve 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 the mechanical bits and pieces you need to assemble 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 elec-

tric 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.

# COMPLEXITY THEORY

Complexity Theory deals with how difficult a given problem is. There are multiple classifications (P, NP, NP-Complete, NP-Hard, Exp, etc) that identify classes of problems based on the idea of time complexity. Over the years complexity theorists have found that many problems are related in some abstract way, and can be reduced, quite interestingly, from one to another.

As a programmer, you solve problems. You should know if the problem is considered easy, hard, or impossible. In addition, you should be able to identify the type or classification of a problem that you're trying to solve.

Every summer I get together with friends from college for the weekend. We'll rent a cabin in the mountains, go on hikes, fish and try not to hurt ourselves playing American football. I look forward to it every year; it's the only chance I get to see most of these people.

I'm sure you have a group of friends just like this one. People you've known for a very long time. Seeing them is always a treat... deciding what you want to do when you see them... well that's when things get more than a bit complex.

Making decisions as individuals is hard enough. Adding more people to the decision process takes something that was hard and makes it feel almost impossible.

Consider this scenario: my friends and I have just returned from a day fishing for trout in a high mountain lake. We're tired, hungry, and more than a little thirsty. We've cleaned ourselves up and someone asks:

*So, anyone up for going somewhere for a pint?*

There are 6 of us standing in the same room. I'm sure the scene isn't hard to imagine – we look from one to the other, trying to gauge what our answer will be. Someone says “sure”, another says “yeah where?” and a third says “I don't drink but I'm happy to tag along if I can find something to eat”.

Uh oh. Group inertia!

You've had these discussions, I'm sure. They're not very fun because it's difficult to figure out what's going to work out the best for everyone; there are just too many parameters to consider. Joe doesn't like bitter beers, Kim wants a cocktail, Kevin doesn't drink but is craving a pretzel and Olga wants to go to her favorite pub 20 miles away because she knows the bartender and can get us a discount.

These situations are usually resolved by one person (typically the loudest) suggesting that everyone get in the damn car, we'll start out at place X and move on from there. This approach works re-

markably well for for a small group of people. If the group were to grow, however, that's when things get complex.

What do I mean by *complex*, however? In normal conversation we could just assume that this means "it's really hard to figure out". As programmers, however, we should have a more precise way of thinking about complexity. We deal with it every day – it's what we do. When your boss (or client) asks you the Big Question: how hard is it to add feature X?, how do you respond?

If you're like me, you might typically say "I'll have a look and let you know". Maybe you've done feature X before and you might say "it's doable".

Shouldn't we have better words for this conversation? Perhaps a way to more clearly define exactly how complex a given problem might be? Yes. We, as programmers need to be able to do this. The ability to clearly understand the complexity of a problem can save your boss or client tons of time and money. You might even avoid getting fired... like me... which is something I'll explain in more detail in just a few minutes.

Let's head back into the hills and see if we can figure out where my friends and I can go grab a pint and a bite to eat. Our situation is getting complicated, let's see if we can figure out just how complicated it's getting.

# SIMPLE SOLUTIONS AND POLYNOMIAL TIME (P)

My 6 friends and I are standing in the middle of the room, staring at each other, trying to figure out where we're going to go. People are stating their criteria and asking questions at the same time, and it's turning into a bit of a muddle.

Aaron, whose cabin we're staying at, speaks up and says:

*Here's a list of the two pubs in town, and Olga's favorite pub 20 miles away. Put a check next to the one you want to go to. We'll go to the one with the most checks.*

This type of solution is simple to implement and, best of all for the 6 of us, happens in a reasonable time frame. What's a reasonable time frame you ask? Good question! The answer is a bit abstract, but important nonetheless: it's less than the amount of time that would prevent us from doing it in the first place.

If deciding on a pub took us 15 minutes, we might get a bit frazzled, but we'd still do it. 30 minutes and we're grumpy for sure – anything more than that would be anarchy.

Thinking about this in terms of programming, we can think of “a reasonable time” as the time we might consider it acceptable and relevant to execute a routine for a given set of inputs. Sorting 100 things should come back in less than a millisecond, for instance.

Sorting 1,000,000 things we might be willing to wait for a second or two.

Crunching analytics on billions of records, we might wait a few hours. Indexing a huge corpus of free text might take an entire night, or possibly a few days!

Using Aaron's democratic sorting process for selecting a pub is, best of all, fast. It also has another advantage: it scales really well. If our group of 6 turned into a group of 12, the process would take twice as long. A group of 15 would take 2.5 times as long.

Notice that I'm talking about the complexity of our democratic sorting process using time? This is the key to understanding Complexity Theory: you think about complexity in terms of time as you scale the inputs that go into the algorithm that you're using to solve the problem.

We can describe the way our democratic sorting process will scale using a bit of math:

$$T = 2x$$

Where **T** is the overall process time in minutes and **x** represents the number of friends. The equation that describes this scaling is a very simple one, and if you're a mathematician, you would call this type of equation a polynomial equation:



*In mathematics, a polynomial is an expression consisting of variables and coefficients which only employs the operations of addition, subtraction, multiplication, and non-negative integer exponents. An example of a polynomial of a single variable  $x$  is  $x^2 - 4x + 7$ . An example in three variables is  $x^3 + 2xyz^2 - yz + 1$ .*

When the complexity of an algorithm scales according to a polynomial equation, mathematicians say that it scales in “P time”, where P stands for *polynomial*. These algorithms are the simplest and often the most boring to work with. List operations, for example, are P time algorithms. Things like sorting, searching, zipping and enumerating all happen in P time.

Complexity Theory, however, isn’t exactly concerned with the algorithms that are used to solve a problem. It focuses on the problems themselves. We wouldn’t say that our democratic sorting process executes in P time, that only matters to the programmers. A mathematician is much more concerned with the problem itself, and how difficult it is to solve. If one were sitting next to me, she would say that our pub selection problem is “in P”.

In more concrete terms, P is a complexity class that describes the set of all problems solvable in P time. Things like sorting and searching arrays, arranging your sock drawer and finding your cat who’s been wandering the neighborhood.

In other words: *simple problems*. These tend to be less interesting and not typically what programmers want to spend their time solving. Have you ever had to create a sorting algorithm by hand? I

have! I had to do it for this book – and I’m going to make you do it with me in a few chapters. It was an interesting exercise, but I don’t think I’d want to be a programmer if writing sorting algorithms was what I did all day.

99% of the time you and I are working with a class of problems that are not in P. They are significantly more complex (and therefore a bit more interesting) and have a classification all their own.

## HARD PROBLEMS (EXP)

Aaron’s democratic pub selection process will indeed help us find a pub in short order, but it’s a bit too simplistic. It would be nice if we had a chance to consider everyone else’s opinion before making our choice.

Olga speaks up with a suggestion:

*How about each of us takes 30 seconds to explain where we want to go and why. Then we can decide after that.*

This is a much nicer way to do things, and also a bit more complicated. Let’s think about Olga’s idea in terms of math.

Each of the 6 of us will be making a decision. With Aaron’s democratic method, there were only 6 decisions + 1: each of us decided where we wanted to go – that’s 6 decisions, plus one more in deciding the winner.

With Olga's way of doing things, each of us needs to make 6 decisions apiece:

- I need to decide what I want to do
- I need to listen to the other 5 people in the room and then decide if I'm going to change my mind

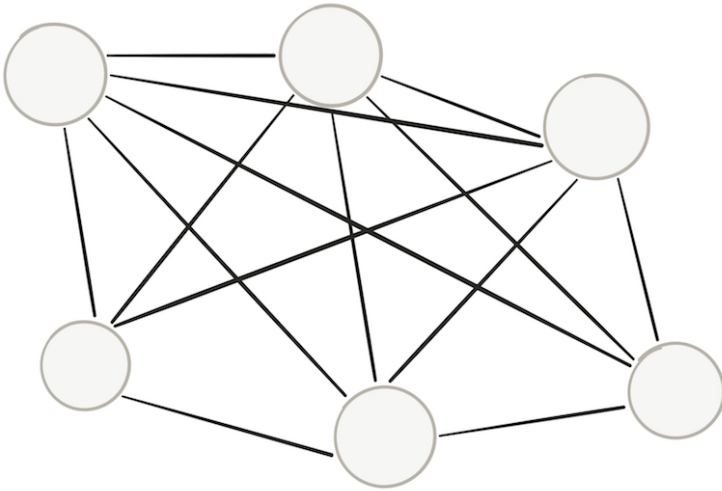
That's 6 people \* 6 decisions, which is 36 total decisions. If this is all it took to make a group decision like ours, we would still be in P time because the equation for calculating the time required is still a polynomial equation:

$$T = n^2$$

This is not the extent of what we need to do, unfortunately. We're trying to *optimize* the selection of where we go – which means that I need to make a decision based on Aaron's, and Aaron's is based on Olga's, and Olga's is based on Todd's which is also based on Kim's...

If your brain is starting to hurt while thinking this through, I don't blame you. It's hard for any human mind to grasp the complexity of this type of process! In mathematical terms, this is a *factorial process*, or  $n!$  where  $n$  is the number of people involved.

You can visualize this with a graph:



Each node on our graph represents one of my friends and each line represents a “weighting” in terms of a decision. If all we cared about was a measurement for each of the nodes, we would have a simple 36-part decision process. We want more than this, however. We want to optimize the combination of decisions, which is a factorial process.

This long-winded explanation is a way of saying that this problem is extremely complex. Rather than use the word “extremely”, however, a mathematician would say it’s “exponentially complex”, or that it is solvable in Exp time.

If we were to add a 7th person to our group – say my friend Kaveh finally shows up – the time to solve this problem would increase exponentially by an order of magnitude.

In pure mathematical terms, exponential time problems are solvable in:

$$T = 2^n$$

Where **T** is time, **n** are the inputs.

Exponential time is a very, very, very long time. It encompasses problems that could take millennia to complete... even millions, or billions of years. The sun can blow up 10 times over and our Exp problem could still be executing... chugging along... trying to figure out the optimal solution for us.

With small cases like ours, it would take us a long time to decide where to go, but it's likely that the sun would still be around when we're finished. Just because a problem is in Exp doesn't mean that it will always take a long time to solve. This is just a classification of complexity. Keep this in the back of your mind – I'll be coming back to it in a bit.

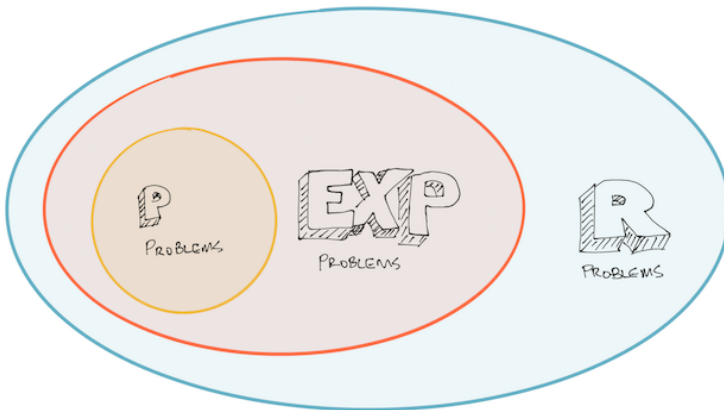
Almost every problem we can think of is in Exp. In fact, all the problems we care about solving are in Exp. I'll get to explain why that's true in just a bit – first, let's see what lies beyond exponential time.

# ALL SOLVABLE PROBLEMS (R)

Simply put, R represents all the problems that are solvable in finite time. In other words: *any problem that we can solve is classified in R*.

You might be confused by some of this as these definitions overlap. Problems in P are “solvable”, aren’t they? As are problems in Exp? Doesn’t this mean they’re also in R?

Yes. These classifications are sets of problems within given timescales. A problem that takes 10 seconds to solve is still solvable within finite time. It’s also solvable in exponential time. These timescales overlap! It might be easier to see this in an illustration:



# INFINITELY COMPLEX PROBLEMS BEYOND R

What problems could possibly take an infinite amount of time to solve? Believe it or not, some deceptively simple ones. There are people who make a living researching these things, if you can believe it.

There's a very famous problem that lies beyond R, and it's known as *The Halting Problem*. You might have heard of it – it was introduced by Alan Turing in the very same paper he used to introduce the Turing Machine. We'll get to Turing, his machine, and the Halting Problem in detail later on – but I'll summarize it very quickly here.

Imagine you've written a program and you want to know if it will end at some point (aka “halt”). Will it return an answer? Will it crash? As it turns out: *it's not possible to write a generalized program that can decide if any other program will halt*. This type of problem is called *undecidable*.

The proof of this is rather elegant and I'll get into it in a later chapter. It has to do with feeding our halt-checking program back to itself, tweaking the internals before we do, which will throw the machine into an infinite loop. The “infinite loop” part is the thing that makes this problem undecidable.

There are quite a few other undecidable problems as well. In fact, if you dive into Complexity Theory a bit more you'll find that most of the problems that exist are undecidable, the proof of which is quite fascinating and a bit beyond my ability to explain. If you want to know more, have a look at MIT Open Courseware. There are hours of material up there.

Let's get back to the Halting Problem, however. I mention that it's solvable beyond R, but the strange thing is that the classification for this problem is something else entirely. It's classified as NP-Hard.

What does that mean? Let's dive into NP.

## DETERMINISM, NONDETERMINISM, AND MAGICAL GUESSES

We have our three main complexity classes defined:

- P, which is the set of problems solvable in polynomial time. In other words: easy
- Exp, which is the set of problems solvable in exponential time. In other words: hard
- R, which is the set of all solvable problems



What do I mean, however, by the term solvable? As programmers we might think of writing a program to solve a given problem, or maybe constructing an algorithm of some kind. For the sake of moving things along, let's assume that these are the same things, and I'll talk about solutions in the context of a computer program.

When you sit down to solve a problem (let's say it's sorting an array), you lay out a set of instructions in code for the processor to execute. Conditional branches, loops, variables, etc. Hopefully you test your code because you need to make sure that the computer can determine the correct answer.

This is how we write computer programs: deterministically. We execute a routine and get a result. Based on that result, we'll execute another one, and then another. Each result links together until we arrive at an answer.

There is another way to arrive at this result, which is nondeterministically. In a nondeterministic system we execute a routine and get a result, from that point we might execute one of a series of next steps. We don't know which next step from that series will be executed – it will be decided at that time.

Once that next step is executed (whatever it is), we might execute one of another series of steps – once again having no idea which step in that series it will be until it's chosen. We keep on doing this until we arrive at the answer. When we do have our answer, there's simply no way of determining how we got there, at least from our deterministic program's point of view.

You can think of nondeterminism as using a series of lucky guesses or chance (each of which is always correct) to figure out the answer to a given problem. If we were able to program a computer this way, things would get very interesting indeed.

Let's revisit the pub selection problem above, where my friends and I are trying to decide where to go have a pint and, possibly, a bit to eat. If I could use a nondeterministic algorithm to help us make this decision, we could leave in very short order! I could phrase the problem in a very particular way, and then off we would go:

- "If we go to the Leaking Moose, would everyone be optimally happy?" Answer: no
- "If we go to the Iron Tongue, would everyone be optimally happy?" Answer: no
- "If we go to the Cougar's Kitchen, would everyone be optimally happy?" Answer: yes

There it is! To arrive at this answer all I needed to do was to iterate over the possible choices and ask a simple yes or no question. My nondeterministic algorithm was able to arrive at the answer directly, because it's capable of correct lucky guesses every single time.

I'll explain more in a second (there is an explanation to this, trust me) – but it's important to understand what just happened. We took an exponentially complex problem and solved it in P time using a lucky nondeterministic process.

This might sound a bit magical, but there are quite a few people studying the notion of nondeterminism, trying to figure out if it's possible to have a nondeterministic algorithm that can solve exponentially complex problems like our pub selection problem. They like this notion so much that they've identified a whole group of exponentially complex problems that can be solved in P time if only we had this amazing nondeterministic algorithm.

These problems are classified as NP: problems solvable in P time given a nondeterministic algorithm. NP can be thought of as a bit of a mythical time frame, because we simply don't know if a nondeterministic algorithm is possible. If it is possible, then we can say that a given problem (like our pub selection problem) is solvable in nondeterministic polynomial time; which is what NP stands for.

More formally: a problem is classifiable in nondeterministic polynomial time (NP) if it is:

- Solvable in exponential time (Exp)
- Verifiable in polynomial time (P)
- Also solvable in polynomial time by nondeterministic methods

NP is where the action is. From the programs we write on a daily basis to the games we play at night with our family and friends: NP problems are what our brains enjoy the most.

# DOES $P=NP$ ?

We know, so far, that problems in  $P$  are pretty simple and can be solved in polynomial time using deterministic methods (aka “computers”). Problems in  $NP$  are quite a bit more complex, but become very simple if we have nondeterministic methods.

But we don’t have those just yet. The question is: **will we ever have the ability to solve problems nondeterministically?** Some people believe it’s just a matter of time before we have a computer chip or algorithm capable of nondeterministic processing. Others think it’s a bunch of silliness. If a nondeterministic algorithm is ever developed that can solve these problems, then all the problems in  $NP$  suddenly become solvable in  $P$  time. Put in mathematical terms:  $P=NP$ . If such an algorithm could never exist, then  $P \neq NP$ . We can’t make either claim immediately because we simply don’t know the answer.

I explained, above, that you can think of nondeterminism as a “lucky guess”, which is sort of true. You can also think of it as a computer’s ability to make the right choice given a set of possibilities. There are languages that are capable of carrying these ideas out. Prolog, for example, allows you to define two functions with the same definition. The programmer does not tell the runtime which to choose (which would be deterministic), the runtime decides. There are various ways that this is carried out, including “backtracking”, where one function is tried, and if it fails, the runtime will back up and try the second function with the same defini-

tion. This is brute force, and won't solve our pub selection in P time... yet. Someday? Maybe?

Who knows...

## REDUCTIONS AND NP

I was able to use nondeterminism to solve the pub selection problem in P time by reducing the problem into a series of yes/no questions. The original problem was something a bit different:

*What are the optimal pubs for optimal group happiness?*

This type of problem is exceedingly difficult in that we're trying to optimize a combination of things: people and pubs. For most people (myself included), this isn't something your brain is equipped to handle – it's just not possible to keep all the permutations together in your mind! These types of problems are called combinatorial optimizations and, as I mention, are very complex. But how complex are they?

Consider the definition of NP as it relates to this problem:

- Solvable in exponential time (check)
- Verifiable in polynomial time (no)
- Solvable in polynomial time by nondeterministic methods (no)

This means that this problem is not classifiable within NP and is, instead, in Exp. Sort of what you might expect as this problem is exceptionally hard.

Our reduction to a decision problem, however, is a different matter:

*If we go to the Leaking Moose, would everyone be optimally happy?*

This problem is solvable in exponential time, so we know it's in Exp. It's verifiable in polynomial time – all I have to do is ask if people are happy once we're at the Leaking Moose. Finally, we already know that our nondeterministic algorithm can solve this problem in P time because we just did it.

That means our decision problem is classifiable in NP. This is where things get weird because, in essence, it's the same problem! I just reduced it from an optimization problem to a decision problem!

This type of thing is very common - where one version of a problem is classified differently than its decision problem reduction. The good news (I think) is that there are two classifications for just such an issue:

- NP-Hard: problems that can be reduced to other problems in NP, but are not within NP themselves
- NP-Complete: decision problems classifiable in NP

I don't blame you if your head is swimming. I know mine is... and I'm the one writing this book! If it helps: decision problems are almost always NP-Complete because of the ability to verify the answer to the problem (make sure it's yes). Combinatorial problems are NP-Hard.

Before we move on to some examples, I want to revisit a statement I made above, about The Halting Problem being classified as NP-Hard. If you've stayed with me through this whole explanation you might be able to reason through the answer.

As we know: solving The Halting Problem is beyond R; solvable only with an infinite amount of time. It is, however, a simple decision problem: will this program halt? Because of that, other problems in NP can be reduced to it, and that's all you need for a classification of NP-Hard.

## NP-COMPLETE AND DECISIONS

One of the main goals of turning a complex problem into a decision problem is the idea of verification. With our initial, combinatorial pub selection problem, we can only verify that we have indeed gone to the optimal pub for our group by going through every iteration of the decision process for each person and each pub. A pub crawl might not sound so bad, but visiting each friend (and each combination of friends) to ask if they like the place would get more than a little tedious.

The decision problem variation, however, is very easy to verify, as we did above.

These types of problems were formalized by Leonid Levin and Stephen Cook in the early 1970s, when they formulated the Cook-Levin Theorem. This theorem states that any problem in NP can be reduced to what's known as The Boolean Satisfiability Problem (or SAT):

*In computer science, the Boolean Satisfiability Problem ... is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.*

This problem sounds intimidating, but it's really not. The best way to think about it is to consider a very long **if** statement (pseudo code):

```
if ((x && y) && (x && !y)) || ((x || y) && (x || !y)){  
    //...  
}
```

Given this, SAT wants to know what values for **x** and **y** will return true?



Imagine a flow chart, with lots of branches, end points and ultimately a satisfiable terminus which returns true. This is a variation of SAT.

Now, imagine the last program you wrote (or the one you're writing now). There are business rules, database queries, and probably some type of UI that a user can use. At its core, this program is a bunch of decisions and can be described by a large decision tree. This is also a variation of SAT.

I suppose if you look hard enough, you could find a software project that exists to solve a problem in P. Most of them don't, however, if they expect to deliver any kind of value. Software exists to make complicated things easier. Given this, most of what we do on a daily basis is to deal with problems in NP.

This is an important thing to recognize, mostly because problems can usually be reduced from one to the other, as we've seen. Put another way: it's highly likely that a mathematician somewhere has tried to solve the very problem that you're working on right now. To understand what I mean, we need to take a small trip back in time.

## CLASSIC NP-COMPLETE PROBLEMS

In the early 1970s, mathematician Richard Karp expanded Cook and Levin's work with his paper *Reducibility Among Combinatorial Problems*. In this paper, Karp showed that you could reduce sever-

al NP problems to SAT in polynomial time, and he came up with a list called “Karp’s 21 NP-Complete Problems”:

### The problems [\[ edit \]](#)

---

Karp’s 21 problems are shown below, many with their original names. The nesting indicates the direction of the reductions used. For example, **Knapsack** was shown to be NP-complete by reducing **Exact cover** to **Knapsack**.

- **Satisfiability**: the boolean satisfiability problem for formulas in **conjunctive normal form** (often referred to as SAT)
  - **0–1 integer programming** (A variation in which only the restrictions must be satisfied, with no optimization)
  - **Clique** (see also **independent set problem**)
    - **Set packing**
    - **Vertex cover**
      - **Set covering**
      - **Feedback node set**
      - **Feedback arc set**
      - **Directed Hamilton circuit** (Karp’s name, now usually called **Directed Hamiltonian cycle**)
        - **Undirected Hamilton circuit** (Karp’s name, now usually called **Undirected Hamiltonian cycle**)
- **Satisfiability with at most 3 literals per clause** (equivalent to 3-SAT)
  - **Chromatic number** (also called the **Graph Coloring Problem**)
    - **Clique cover**
    - **Exact cover**
      - **Hitting set**
      - **Steiner tree**
      - **3-dimensional matching**
      - **Knapsack** (Karp’s definition of Knapsack is closer to **Subset sum**)
        - **Job sequencing**
        - **Partition**
          - **Max cut**

*Figure 1.2: Karp’s 21 NP-Complete problems.*

One of the best things you can do for your career is to get to know these and other NP-Complete problems, at least at a high level. They’re fascinating to understand! Who knows? You might even save your job someday...

Let's take a look at a few.

## **Knapsack**

This problem has been around for over 100 years, and is a combinatorial optimization problem that centers on packing a bag for the weekend:

*Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.*

Once again, we're trying to optimize combinations of values: price and weight for a limited space.

## **Clique**

This problem was first formulated in 1935, and has to do with graphs and graph theory:

*Consider a social network, where the graph's vertices represent people, and the graph's edges represent mutual acquaintance. Then a clique represents a subset of people who all know each other, and algorithms for finding cliques can be used to discover these groups of mutual friends.*

You can see how the solution to this problem could apply to any social aspect of an application, or grouping of like “things” that are self-assembling.

## **Bin Packing**

This problem is a variation of Knapsack, and is once again a combinatorial optimization problem that you encounter quite often in the programming world:

*... objects of different volumes must be packed into a finite number of bins or containers each of volume  $V$  in a way that minimizes the number of bins used.*

If you’ve ever had to pack up your house or apartment and move, you’ve had to deal with the Bin Packing problem. This is a classic NP-Hard problem because of its combinatorial nature, and the fact that verifying that you’ve optimally packed things up means that you have to carry out every possible iteration to prove yours is the best.

We can reduce this to a decision problem, however, by iterating over bin configurations and asking if the current configuration is optimal. This reduction turns the combinatorial problem into a decision problem, which would classify it as NP-Complete.

## Traveling Salesman

The classic NP-Hard problem of trying to figure out the cheapest way to send a salesman on a trip:

*Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*

Once again, a combinatorial optimization problem, which we can recognize now as NP-Hard. You can reduce this in P time to an NP-Complete decision problem by simply enumerating through every valid path and asking “is there a path that’s shorter”?

## APPROXIMATIONS AND LAZINESS

If you follow any of the links for these problems, you’ll see that there are algorithms that exist which “solve” them in some cases in P time. These are called approximations and are very useful if you can tolerate their inexact nature.

For Traveling Salesman, you could start from Los Angeles and head to the next nearest city, which (for our example) might be San Francisco. When you get there, you see that of all the destinations you could get to, Reno is the next nearest, so you go there. This approach is called “nearest neighbor” and is classified as a “greedy algorithm”, which means you do what suits your current position and value on the graph. Nearest neighbor usually returns a path

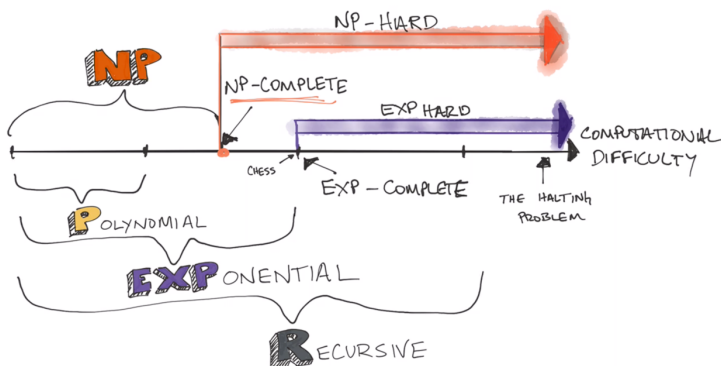
within 25% of the shortest one, on average. Would that work for you? For many companies it just might.

## IN THE REAL WORLD

I'd like to share a story with you about getting fired, and how I could have avoided it. Before I do, let's go over what we know so far:

- Simple, boring problems can be solved in a short amount of time; what a mathematician would call "P" time.
- More difficult problems, such as a group of people trying to decide on the optimal location for a pint and some food, are more complex and solvable in exponential time, or "Exp".
- Some problems are so complex that they can't be solved in all the time we can possibly have (R, or "finite time"), and are simply undecidable. For instance: Turing's Halting Problem.
- There are special subclasses of problems that are of great interest to us, specifically: exponentially complex problems that we can solve in P time with a nondeterministic lucky guess. These problems are in a classification called "NP". We can further divide this complexity class into decision problems that we can quickly verify (NP-Complete) and more complicated problems that are in Exp, but that other problems in NP can be reduced to in P time (NP-Hard).

We can visualize this on a timeline:



It might take a few reads to have this all sink in, but you might be wondering something: why in the world should I care. For that, I'd like to share a story.

## WHERE'S THE PARTY?

In 2010 I was hired to build a Ruby on Rails site for a client that wanted to enrich the lives of college freshmen. The idea was simple: let's help these new students make friends and find interesting things to go do.

Simple enough! I had been building web sites for the previous 12 years, this should be straightforward. Indeed it was! I built the core of the site, integrated the designs, and all was well until a status meeting we had one Tuesday afternoon:

*So, Rob how's the progress on the algorithm?*

Not knowing what algorithm they were talking about, I asked. They responded:

*You know, the one that is going to match students with other students and places to go, etc. We were told you were good with this kind of thing...*

Oh no. It's true: one of the reasons I got the job was because I had a background in analytics, but I didn't think I'd be doing any of it for this site! They hired me to build a Rails app!

This is where my ego took over. I hate saying "no" to clients, and somewhere in my mind was a cowboy looking for adventure... so I said what turned out to be some fatal words: tell me more...

The "algorithm" turned out to be this:

- Gather a basic interest profile from new signups. Things like favorite music, extrovert/introvert, favorite actor, etc.
- Create a tagging system for submitted events. Anyone could submit an event to the system - but they had to tag it so we could match on it.
- Optimally match new students to other students and also to events that they would find interesting.

At this point in the chapter you should know what we're dealing with here. That's right: a combinatorial optimization. I could prob-



ably reduce this problem to any of Karp's NP-Complete problems, but (here, now, in the future) I don't need to do that to know that this problem is going to require some lengthy discussions.

Back then, I simply said "let me take a look".

You probably know where this is going to end up. **I got fired.** I looked into various ways to solve the problem but quickly realized that this isn't something that:

1. A Ruby on Rails site could handle
2. Could happen in real time
3. I would even know how to start implementing

I pushed back as best I could, the client insisted this is why I was hired, I asked them to show me where I agreed to this in the contract... but it didn't matter. I got fired and the project blew up.

If I knew then what I know now about exponential time algorithms, approximations, decision problems and all the other lovely things discussed in this chapter – I might have been able to walk the client through the problem and why, no matter how many developers they hired then fired, they would never get what they wanted.

*Maybe.*

Have you worked on a project like this? I'm sure you have. Or, if not, it's likely you will. People try to tackle NP-Complete and NP-

Hard problems all the time – after all we have machine learning and big data don't we?

The good news for you? Hopefully you will now be able to spot these efforts from the start and, hopefully, help your team approach the problem with care.

# LAMBDA CALCULUS

**B**efore there were computers or programming languages, Alonzo Church came up with a set of rules for working with functions, what he termed lambdas. These rules allow you to compute anything that can be computed. You use Lambda Calculus every day when you write code. Do you know how it works? As a programmer, understanding Lambda Calculus can enhance your skills.

## THE CODE

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video walkthroughs of the algorithms you see in this chapter and others from here. I'll be using screenshots once again for the code samples for formatting reasons – if you want to play along please do... but you'll need the code from GitHub.

In the early 20th century, mathematicians began to ponder what it means to have a machine solve problems for you. The question was simple: how do you compute something?

The steps to solving certain problems (aka: algorithms) had been known for millennia; the trick was to be able to give these algorithms to a machine. But how? More than that: is there a limit to what a machine can calculate? Are there solutions that machines simply cannot compute?

This led to some interesting discoveries in the early 20th century, most notably by two men: Alan Turing and Alonzo Church. We'll talk about Alan Turing in a later chapter.

This section is about Alonzo Church's contribution: the **Lambda Calculus**. I should note here, as I've done in so many chapters, that I could spend volumes diving into the details of Lambda Calculus. What you will read here is a simple summary of the very basics. I do, however, think the missing details are critical and if you care, I would urge you to have a look online – there are quite a few resources.

So, consider this a gentle introduction, the results of my recent headlong dive into the subject. Hopefully you will read enough to ignite your curiosity – which it should! What you're about to read is the foundation of computer programming.

**Note:** *the lambda function syntax you're about to read is included as actual text in the body of the chapter instead of being treated as code. In previous versions of the book I formatted the simple lambda equations as code, but many people found that confusing. The actual code samples that illustrate the lambda functions, however, are left as-is.*

## CREDIT WHERE DUE

I've read quite a few articles and textbooks on Lambda Calculus and I wanted to list them here, as I would never have been able to understand the basics otherwise.

At the top of the list is this detailed explanation of Y Combinator and  $\Omega$  Combinator from Ayaka Nonaka. It is outstanding. I wanted to add some details about combinators and almost gave up, until I found this post.

Next is Jim Weirich's amazing keynote on the Y Combinator. I remember watching it years ago, having my mind blown. I watched it three times over when writing this chapter and most of it still goes over my head. I link to it again below.

Ben Hall has an outstanding GitHub repository that has all kinds of Church encoding magic. If you find yourself lost in any of this, go study the code in his repo. The Church encoding you see below is based directly on his work.

Finally, I'd like to thank James G (last name omitted... but you know who you are) for his patience helping me understand a small but crucial aspect of reduction and substitution. In the original version of this chapter I managed to get a few things wrong; I wouldn't consider them critical, but they were wrong nonetheless. After a number of lengthy, wonderful emails James convinced me of these errors, which I could cross-check and verify easily. Thanks James!

# THE BASICS

Alonzo Church introduced Lambda Calculus in the 1930s as he was studying the foundations of mathematics. As a programmer, you might recognize the description:

*The  $\lambda$ -calculus is, at heart, a simple notation for functions and application. The main ideas are applying a function to an argument and forming functions by abstraction. The syntax of basic  $\lambda$ -calculus is quite sparse, making it an elegant, focused notation for representing functions. Functions and arguments are on a par with one another. The result is an intensional theory of functions as rules of computation, contrasting with an extensional theory of functions as sets of ordered pairs. Despite its sparse syntax, the expressiveness and flexibility of the  $\lambda$ -calculus make it a cornucopia of logic and mathematics.*

A lambda is simply an anonymous function that can be thought of as a value. Most modern programming languages have some notion of an anonymous function, expression, or lambda. Lambda Calculus is where this idea arose. In fact Lambda Calculus is the foundation of what we consider programming today.

Lambda Calculus is an abstract notation that describes formal mathematical logic. There are no numbers or types; only functions. Like modern functional languages, a function in Lambda Calculus can be treated as a value. By arranging these functions carefully, you can build out some very interesting structures.

In this chapter we'll use Lambda Calculus directly, but we'll also jump into ES2016 (JavaScript) to see how some ideas might work with modern code.

## SOME RULES

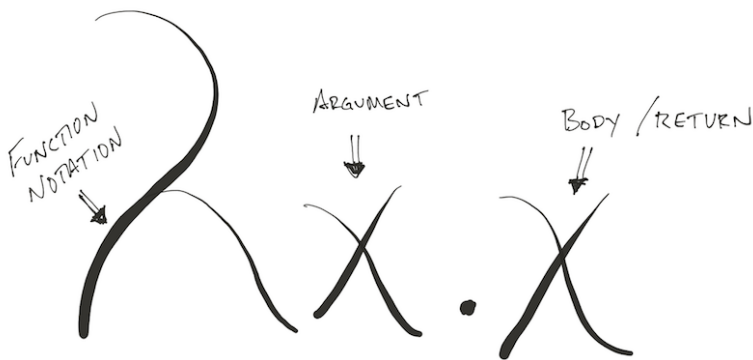
The rules of Lambda Calculus are rather simple:

1. There are only functions, nothing else. No data types (strings, numbers, etc.) of any kind
2. You can substitute functions using variables
3. You can reduce one function to another through application
4. You can combine multiple terms in Lambda Calculus to create a higher-order function called a combinator, which is where the fun begins

There are rules which you can apply to these functions and combinators, and we'll get into a bunch of them using Lambda Calculus and also a bit of JavaScript.

## ANATOMY

Let's see our first function:



This is a Lambda Calculus function, sometimes called a “term” or “expression”. The first thing to notice is the Greek lambda ( $\lambda$ ) on the left, which denotes a lambda function. The next bit is the **x**, which is the argument to the function. The final part of the expression is the body, which is the second **x** and is segregated from the argument by a ..

Using JavaScript, you could think of this function as **function thing(x) {return x}**. The **function** keyword is equivalent to the **λ** symbol. Our function takes an argument **x** and returns **x**. While this is, indeed, an anonymous function in JavaScript, there is a more applicable syntax that can be used with ES6: **(x => x)**

This is a pure lambda expression in the following ways:

- It takes in an argument, **x**, and since we’re using a single line the value **x** is also returned



- It is a functional closure, which means we can set this expression to a variable and invoke it anywhere without worrying about scoping issues
- lambdas are the same as values

In this way, JavaScript follows Lambda Calculus conventions closely.

## FUNCTION APPLICATION

Let's revisit our first function:  $\lambda x.x$  This function takes an argument  $x$  and returns it. We can apply a value to this function like so:  $\lambda x.x(y)$ .

This notation means “substitute all occurrences of  $x$  with  $y$ ” and is called *function application*. It's just the same as if we did this in JavaScript:

```
(x => x)(y)
//or
function thing(x){return x}(y)
//returns y;
```

This function:  $\lambda x.x$  in Lambda Calculus has a special name: the *identity function*. Whatever you pass to it is returned. It's also called the *I Combinator* – which we'll discuss in a bit.

Consider this function and application:  $\lambda x.y (z)$ . What do you think will happen here? It would be the same as doing this in JavaScript:

```
let z = 3;  
function thing(x){return y}(z);
```

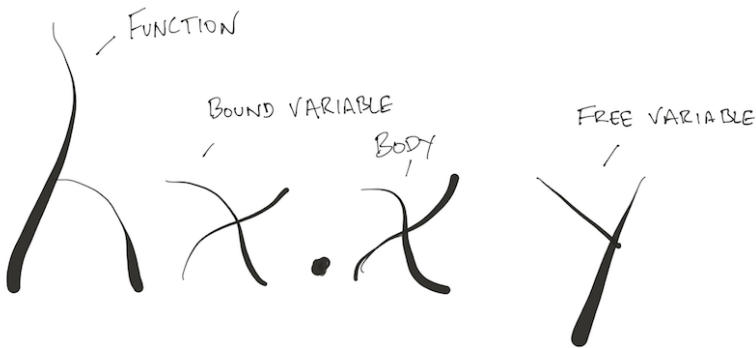
Here I'm using the value 3 just for clarity, though it's worth restating that there are no numbers in Lambda Calculus. There are *representations* of numbers, which we'll get to in a minute, but no strict numbers.

With this function:  $\lambda x.y$ , it doesn't matter what you pass into it, some value **y** will be returned. This is called the constant function as it returns a constant value of **y** no matter what you pass in for **x**.

## BOUND AND FREE VARIABLES

When you apply a value to a function, there are some rules. The first thing you have to know is what *types of variables you're working with*. Note: I'm not referring to data types as there are none in

Lambda Calculus; I'm referring to one of two different designations for variables:



In this expression we're dealing with two variables: **x** and **y**. The variable **x** is in the function head as the argument and thus is bound to this expression. The variable **y** on the other hand is not bound to any lambda function, so its known as "free".

The difference between the two is essential. I can substitute **x** in this function if (and only if) I make sure to substitute every occurrence of the bound variable **x**. So if I wanted to change **x** to **z**, for instance, I could:  **$\lambda z.z y$** .

This wouldn't change the meaning of the function. If, however, I decided to change **y** we could run into trouble. To understand why, we need to dive into substitutions and reductions.

# SUBSTITUTION AND REDUCTION

When you apply a value to a function, you substitute that value in the function itself. Substitution is *left-applicative*, which means you start on the left and move to the right as required.

For instance, let's substitute and then reduce this function:  $\lambda x. x \ 3$

The first thing to do is to state that all bound occurrences (that's important, they have to be bound) of  $x$  will be replaced with 3 in the body,  $x$ :

$$\lambda x. x \ 3$$
$$x[x := 3]$$
$$3$$

This notation:  $x[x:=3]$  can be read as “replace all bound occurrences of  $x$  in body  $x$  with 3”. Now, I know I said that there are no numbers in Lambda Calculus - and this is true! I'm using 3 here simply to show how substitution and reduction work.

Let's apply this to JavaScript. Consider this expression:

```
let y = 3;  
const fn = (x => y)(1);
```

What do you think the value of **fn** will be? The same principles apply here: we're substituting all instances of **x** with 1 in our equation, which doesn't matter as we're returning **y**, which is set to 3. This is, once again, the constant function so it's no surprise that whatever we set **x** to doesn't matter.

Let's try this again with the identity function:

```
let y = 3;  
const fn = (x => x)(1);
```

In this case **fn** would "reduce" to 1.

# APPLYING MULTIPLE VALUES

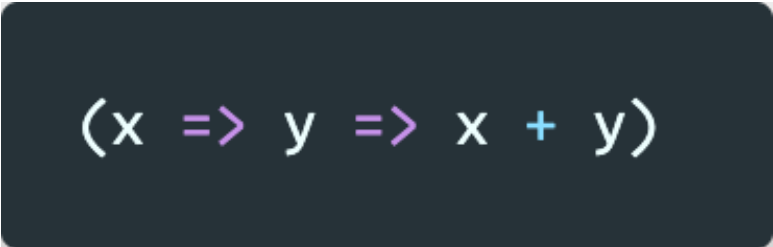
You can create lambda functions that handle multiple values as well. It would be tempting to write them like this:  $\lambda xy.t$

But this would be incorrect. Functions in Lambda Calculus only have an arity of 1 - which means they can only have one argument or one bound variable. To write this expression properly we would need to reorder things:  $\lambda x.\lambda y.t$

This is called currying: breaking a function with multiple arguments into chained functions with an arity of 1. We'll talk about currying more in a later chapter.

In this expression, the lambda function  $x$  has a body of  $\lambda y.t$ . This might look confusing, but if you remember that everything in Lambda Calculus is a function, it makes a bit more sense.

This corresponds to this lambda in JavaScript:



```
(x => y => x + y)
```

Reductions get interesting at this point. Consider this function:  **$\lambda x. \lambda y. y \ x$**  We need to apply the substitution from left to right, so our first substitution will be “replace all bound values of  **$x$**  with  **$x$** ”:

$$\lambda x. \lambda y. y x$$

$$\lambda y. y [x := x]$$

$$\lambda y. y$$

There is no substitution in the body,  **$\lambda y. y$** , which leaves us with  **$\lambda x.$**  ( **$\lambda y. y$** ), which is the constant function, which means we can reduce further to  **$\lambda y. y$**  and we’re set.

We can see this reduction in action with JavaScript as well:

```
const first = (y => y);  
const second = (x => first(x));  
console.log(second(first));  
//[Function: first]
```

In more concrete terms, let's revisit the JavaScript expression above  $(x \Rightarrow y \Rightarrow x + y)$ . This time we'll reduce it by running a substitution for  $x$  and  $y$ ... in other words "invoking" the lambda:

```
(x => y => x + y)(2)(3);
```

In this code,  $x$  is set to 2 and then passed to  $y$ . The resulting function would then be  $y \Rightarrow 2 + y$ , which we apply 3 to, resulting in our answer, 5. This is straight up Lambda Calculus.

But what about this?

```
(x => x)(y => y)(2)(3);
```



This won't work and we'll get an error. Can you reason as to why?  
Let's make it work and then step through it:

```
(x => x)(y => y)(2); //2
```

This substitution and reduction happens in the same way, but the result of one lambda is passed to the next. So, **y => y** is passed to **x => x**, which returns whatever was passed to it as it's the identity function.

This leaves the value 2, which is then passed to **y => y**, which is once again the identity function that returns 2.

To see this in more detail, set **y => y** to **y => 5**. In this case **y => 5** is passed to **x => x** which, once again, just returns **y => 5**. If we pass 2 to that it's ignored completely, so we get an answer of 5.

## ORDER OF OPERATIONS

Just like any mathematical operation, the use of parentheses can affect the order of operations in lambda expressions. Consider this function: **λx.(λy.y x)** How do you think this would reduce? The parentheses dictate the order of operations so we would first reduce the body of the function like so:

$$\lambda x. (\lambda y. yx)$$

$$\lambda x. (y[y := x])$$

$$\lambda x. x$$

If we omit the parentheses, something different happens:

$$\lambda x. \lambda y. yx$$

$$(\lambda y. y)[x := x]$$

$$\lambda y. y$$

An entirely different result.

These are the basics of Lambda Calculus, now let's see what we can do with it!

## CHURCH ENCODING

Lambda Calculus looks a lot like programming, doesn't it? Unfortunately, this can cause confusion for those trying to grasp it. Programming languages have numbers, strings, control structures and conditional branching built in; Lambda Calculus just has functions.

It's important to muse on this for a bit before we push forward. Put yourself in a classroom seat at Princeton, back in the early 1930s.

Alonzo Church is trying to represent constructs that we take for granted today: conditional branches, loops, and higher-order functions that can be used to compute things.

To do this, he set about creating *representations* for various values and operations. This is called *Church encoding*, and allows us to use booleans, numbers, conditional statements, and loops to construct things called *combinators* which are, unsurprisingly, combinations of functions that do a thing. We'll take a look at those later on; for now let's do the simplest of computations with Church encoding.

## Booleans

How would you convey a simple conditional construct, like an if statement, using only symbolic functions? We need to have something like this (pseudocode): **if(true) x else y**.

The first step is to have a notion of what "true" means in Lambda Calculus, and we have that with the following representation:  **$\lambda x.\lambda y.x$** . The first bound variable is returned for a **true** statement in Lambda Calculus. Conversely, a **false** statement returns the second bound variable:  **$\lambda x.\lambda y.y$** .

Formalizing this to JavaScript we might have:

```
let True = (x => y => x);  
let False = (x => y => y);  
True(true)(false) //true  
False(true)(false) //false
```

That's a great start! Now we need to stretch this further to have a conditional statement. We need to evaluate 3 things: returning the first value if **true**, second if **false**. Using Lambda Calculus and leveraging Church encoding, we can use this expression:  $\lambda x.\lambda y.\lambda z.x\ y\ z$ .

Translating this to JavaScript:

```
let If = (x => y => z => x(y)(z));
```

Our first argument, **x**, will be set to the function under evaluation. The arguments **y** and **z** will then be given to **x** to evaluate. The result of that will be the result of our **If** function.

Now, let's apply things. We can reuse our definitions of **True** and **False** to see if our **If** works:

```
let True = (x => y => x);  
let False = (x => y => y);  
let If = (x => y => z => x(y)(z));  
  
If(True)("TRUE")("oops..");//TRUE  
If(False)("oops")("FALSE");//FALSE
```

Yay! We have booleans and conditional statements, now we just need to work with some values.

## Numbers

You can represent numbers in Lambda Calculus by arranging a function to, basically, encapsulate and call itself.

It might be easier to see this rather than to explain:

$$\lambda f. \lambda x. x = 0$$

$$\lambda f. \lambda x. f(x) = 1$$

$$\lambda f. \lambda x. f(f(x)) = 2$$

$$\lambda f. \lambda x. f(f(f(x))) = 3$$

This might look rather arbitrary at first, but there is some logic at work here. Consider this statement: **f(x) = x**. A function of **x** is equal to **x**. This means the function essentially has no value at all. This, therefore, represents 0: **λf.λx.x**. There is no reduction to be done here as nothing is applied. Thus **f** has a representative value of 0.

Now, consider this statement:

$$\lambda f. \lambda x. fx$$

$$(\lambda x. f)x[f := x]$$

$$\lambda x. x$$

The application of this function results in the identity function, which is a bit like multiplying by 1.

Let's try this with JavaScript. The first thing we need to do is to set up a **calculate** function that will figure out how many times a given function is called. This function will accept a Church number and invoke it with a return that is itself a function. Every time that result function is invoked, it will increment itself:



```
let calculate = f => f(x => x + 1)(0);

let zero = f => x => x;
let one = f => x => f(x);
let two = f => x => f(f(x));
let three = f => x => f(f(f(x)));

calculate(zero) // 0
calculate(one)  // 1
calculate(two)  // 2
calculate(three) // 3
```

Very nice! We now have a way of representing numerical values with nothing but a set of functions.

There is a lot more we can do here; things like addition, subtraction, multiplication, simple list operations and so on. In fact, we can encode our way to a Turing complete programming language. Which is really no surprise as the exercise we've just gone through is the foundation of what we think of as *programming itself*.

# COMBINATORS

We have booleans, numbers, and conditional branching; let's see what else we can do with Lambda Calculus. Before we get there, I should note that we've already been working with *combinators*, which are simply combinations of smaller functions that do a thing.

More precisely, a combinator is defined as:

*... a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.*

In other words, we have lambda expressions and bound variables, nothing else. Let's see what we can do.

## Loops and The Omega Combinator

Every good programming language needs to have the ability to loop, so let's see if we can implement that. We've seen that the identity function returns whatever is passed to it, so you might be tempted to do something like this:

$$\lambda x. x(\lambda x. x)$$

$$x[x := \lambda x. x]$$

$$\lambda x. x$$

This works once, but not multiple times. We can see this with JavaScript and Church numbers:

```
const zero = f => x => x;  
const one = f => x => f(x);  
const two = f => x => f(f(x));  
const three = f => x => f(f(f(x)));  
const four = f => x => f(f(f(f(x))));
```

A previously defined, a lambda can call itself, but there's no way to call a lambda from within its definition. We've been working with JavaScript a lot and we're able to work with variables and labels so

it might seem like we can do this, but with pure Lambda Calculus, there are no such conventions as functions don't have a name.

But what if you could create recursion as a function itself? We already have a bit of a start with the identity function. In our example above, however, it only returns what's on the right. How can we get it to repeat? What if we added one more application of **x**?

$$\lambda x. xx(\lambda x. xx)$$

$$x[x := \lambda x. xx]$$

$$\lambda x. xx\lambda x. xx$$

### *The Omega Combinator*

Well now, look at that. A function that not only returns what its given, but also its replica. This is called the **Ω** combinator, or “looping” combinator as it allows for recursively looping over a given function.

What does this look like in JavaScript? Let's see:

```
let Omega = x => x(x);  
console.log(Omega(Omega));  
//RangeError: Maximum call stack size exceeded
```

A recursive loop. This is interesting, but how can we apply some other function to this recursion? In other words, we have a basic looping structure, but it doesn't really do anything.

What would be more fun is to have a recursive function that would execute a given function once per iteration. In other words, what you and I might consider a for/each construct.

## The Y Combinator

The Y combinator is a recursive, fixed-point function. It's sometimes called the "fixed-point combinator" as well. If you're like me you might not know what "fixed-point" means.

It is, essentially, when the result of a given function is the same as the input:

$$yf = f(yf) \text{ for all } f$$

Consider this function:

$$f(x) = x^2 - 3x + 4$$

If I set **x=2**, then the input will equal the output. This is the fixed point of this function. We want to expand on this idea, but instead of passing in a value to the Y combinator, we want to pass in a function, invoke it, and then get that function back so we can invoke it again later.

This is what the Y combinator does: you give it a function as an argument which it invokes. It then returns that function back out to you.

More formally, the Y combinator is defined as:

$$Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$$

It looks just like the **Ω** combinator doesn't it? The main difference, however, is that instead of just replicating **xx**, we're replicating the application **f(xx)**, where **f** is the initial function itself. In addition, the Y combinator allows you to pass in a second expression which represents the number of times you want the loop to execute. We'll see this below.

If you're stuck on this (which I was, for a very, very long time), go through the building up of the  $\Omega$  combinator to see how the recursion/replication works. Ideally you should see the same pattern at work here; all we're doing is adding a function definition (**f**) and then invoking it.

In JavaScript it looks a bit different. You can't reference *f* in the second lambda definition as you'll get an error, so you need to wrap it with an additional lambda:

```
const Y = f => (x => x(x))(x => f(y => x(x)(y)));
```

### *The Y-Combinator in JavaScript*

This is interesting, but what can you do with it? Let's see.

We need a function to iterate over, which you can think of as the code block you might pass to a **foreach** statement. As with all computer science books, Fibonacci must appear at some point, so let's use that:

```
const fib = f => n => n <= 1 ? n : f(n-1) + f(n-2);
```

If you don't recall, Fibonacci defines a set where a given number is the sum of the two previous numbers in the set. The first few numbers are: **1, 1, 2, 3, 5, 8, 13...**

The expression above simply takes in a number, **n**, and checks to see if it's less than or equal to 1. If it is, then it's returned. If it's not, then the function calls itself to figure out previous values.

There is a better, faster way to write this which I'll go into in the chapter on algorithms. It's important to note, however, that it wouldn't be a pure lambda expression if I did that. We want a self-contained function expression. No variables, no conditionals, etc. Just functions - Lambda Calculus style.

Let's use Y Combinator to implement our function:

```
let Y = f => (x => x(x))(x => f(y => x(x)(y)));  
let fib = f => n => n <= 1 ? n : f(n-1) + f(n-2);  
let yFib = Y(fib);  
yFib(10);  
//55
```

Yes! It works! We've just used Y combinator to execute our **fib** function 10 times, which yields the result **55**.

Want to play around with some other combinators and see what you can create? Head over to Ben's GitHub repo and have a play!



# SUMMARY

My head is starting to hurt a bit; it might be time to move on. Hopefully you have a grasp on the basics of what Lambda Calculus is and why it's important. Simple functions that do a simple thing, which you can arrange carefully to do more complicated things in a symbolic way.

A compelling way to compute things, which allowed Alonzo Church to make this claim:

*All total functions are computable.*

Put another way: *if it can be computed, Lambda Calculus can compute it.* Turing agreed with this, and in 1937 he proved that his Turing machine provided the same computational abilities as Lambda Calculus, which led to the Church-Turing conjecture. We'll discuss that in a later chapter.

Today, this type of statement is kind shrug-worthy. We have powerful computers that can compute almost anything – so what? Back then, however, people with pencils and papers were the ones doing the calculations and they were limited in their ability.

This led mathematicians to ponder *just what could be computed* in terms of complexity – the human brain is only capable of so much. What was missing was a method of computation that could guarantee a result if the thing to do the computing had unlimited resources. Like a mechanical computer.

Those came along in just a few short years after Church and Turing's work and guess what we use today for programming these things?

**Lambda Calculus.**

# SYMBOLS AND CIRCUITS

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.

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	<b>00</b>
0	1	<b>01</b>
1	0	<b>01</b>
1	1	<b>10</b>

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:

1. The extra bit is "carried" to the left
2. 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:

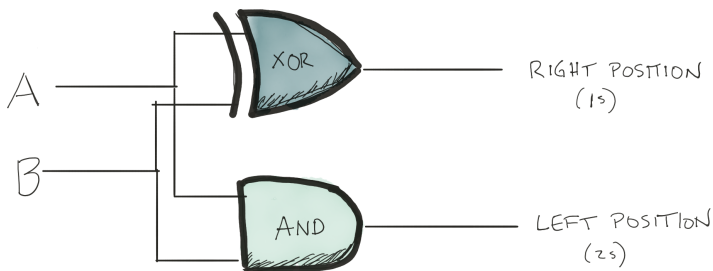
AND

A B		OUT
0	0	0
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":



*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.

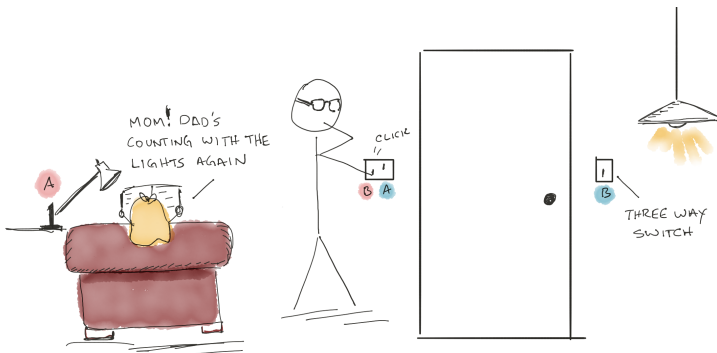
If you're having a hard time thinking about what these things might look like, consider the light switches in your house. If you have 2 separate switches that control the same light, *that's an XOR*. Think about it! If both switches are down, the light is off, same with both up. In any other position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

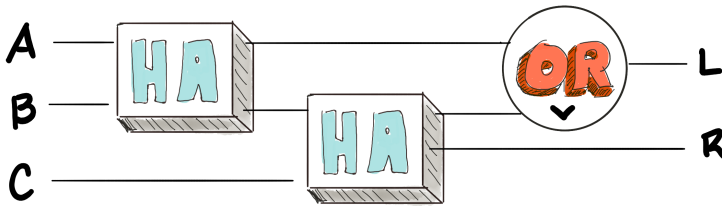
Endless fun for the whole family!



## 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! The full adder circuit consists of two half adders (surprise!) and an OR gate to handle the carry bit:





*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:

16	8	4	2	1	
1	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.

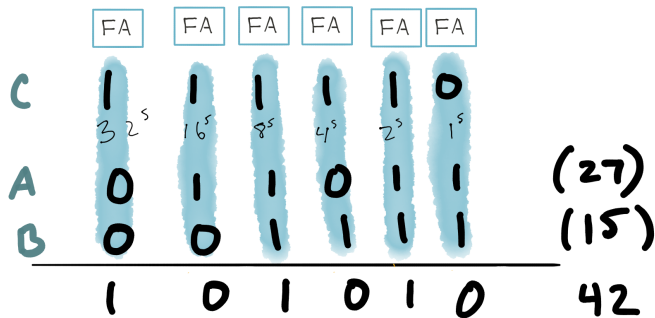
Now we add them together:

C	1	1	1	1	1	0	
	32	16	8	4	2	1	
A	0	1	1	0	1	1	(27)
B	0	0	1	1	1	1	(15)
	1	0	1	0	1	0	42

ONLY 3 NUMBERS.

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.

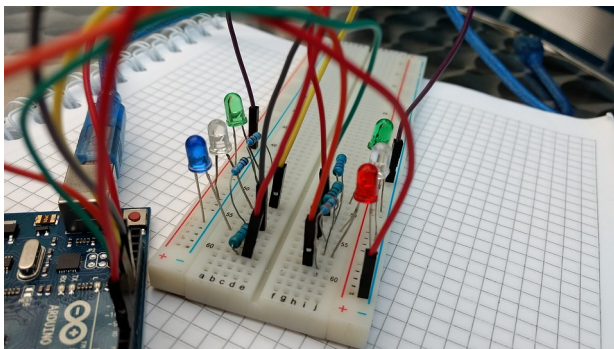
The big thing that we care about is that each of the steps in adding these two numbers can be accomplished using a single full adder:



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 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

**T**heory 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.

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.

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. How exciting!

Let's prove this to ourselves. Crack open your editor and let's go! We're going to code these expressions by hand.

# 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		
0	0	0
0	1	0
1	0	0
1	1	1

OR		
0	0	0
0	1	1
1	0	1
1	1	1

NOT		
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. Here's our truth table:



A terminal window titled "Terminal" shows the command `logic_gates git:(master) node 01-simple.js`. Below the command, three truth tables are displayed, each enclosed in a dashed box. The first table is for XOR, the second for IMP, and the third for EQUIV. In each table, the results column is highlighted with an orange box.

	0	1
0	0	1
1	1	0

	0	1
0	1	1
1	0	1

	0	1
0	1	0
1	0	0

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:

Terminal			
XNOR			
0	0	1	
0	1	0	
1	0	0	
1	1	1	
NEQUIV			
0	0	0	
0	1	1	
1	0	1	
1	1	0	
NIMP			
0	0	0	
0	1	1	
1	0	0	
1	1	0	

← Same as Equiv

← 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

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

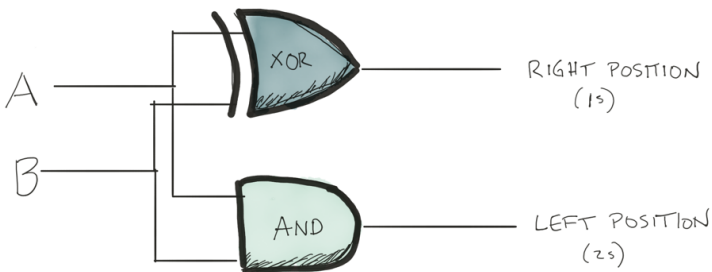
You'll most likely hear other developers talk about these concepts when discussing past interview questions, usually accompanied by a roll of the eyes. It's understandable: we're not hired to write calculators, but we *should* understand how it's possible.

We just went through the process of creating basic logic gates in a medium we understand: code. We could build the 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 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) █
```

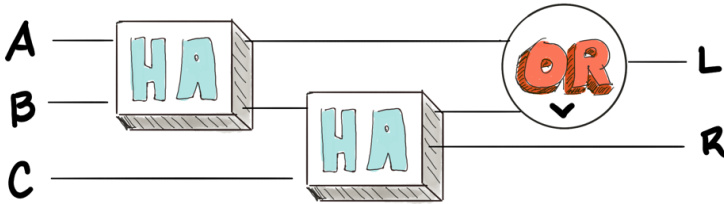
Neato - that's a binary 2! That was pretty simple, don't you think? Now that we have a half adder, let's see if we can use it to create a *full adder*. Before you look at the answer below, look over the diagram I created above. See if you can implement this yourself!

## 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:



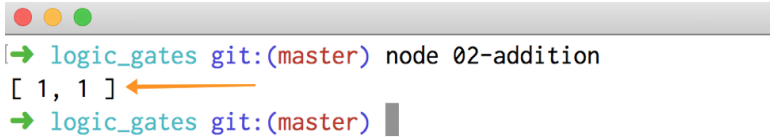
I don't blame you if this looks a little opaque. The thing that helped me to understand this is simply the idea of adding two single bit numbers together and seeing if there's a carry.

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:

**00**

**01**

**01**

**10**

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:

***01***

***10***

***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:

```
→ logic_gates git:(master) node 03-addition.js
101010 ←
→ logic_gates git:(master) █
```

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:

The image shows two handwritten multiplication problems. On the left is a binary multiplication: 11 multiplied by 11. The first row is 11, the second row is 11 shifted one position to the left, and the result is 1001. Red dashed lines connect the 1s in the second row to the 0s in the result. On the right is a decimal multiplication: 3 multiplied by 3. The first row is 3, the second row is 3 shifted one position to the left, and the result is 9.

$$\begin{array}{r} 11 \\ \times 11 \\ \hline 11 \\ 110 \\ \hline 1001 \end{array}$$
$$\begin{array}{r} 3 \\ \times 3 \\ \hline 9 \end{array}$$

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.

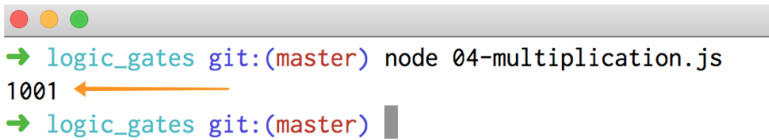
One significant thing to note, however, is that the rule that any number times 0 is 0 still holds! That means I only need to remember sums if the number I'm multiplying by is 1:

```

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):

A terminal window with a grey title bar and three colored window control buttons (red, yellow, green) on the left. The terminal shows a green prompt character followed by the command `logic_gates git:(master) node 04-multiplication.js`. Below the command, the output `1001` is displayed. An orange arrow points from the output `1001` back to the command. Below the output, there is another green prompt character followed by the command `logic_gates git:(master)` and a grey cursor block.

```
→ logic_gates git:(master) node 04-multiplication.js
1001 ←
→ logic_gates git:(master)
```

Yes! That number there is a binary 9! Not the prettiest code, to be sure, but please don't hesitate 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 an elementary (and basic) binary calculator using the 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

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.

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

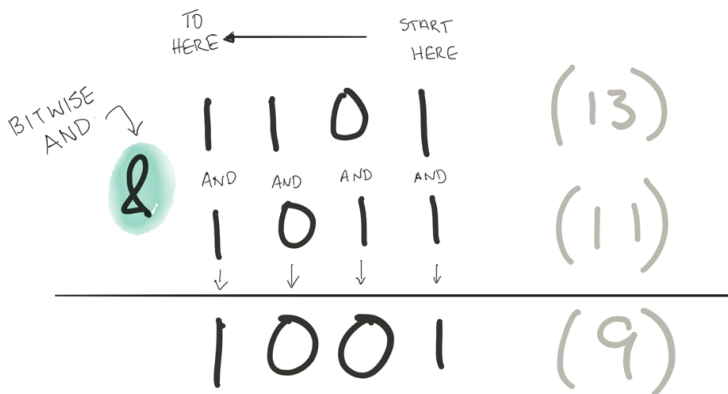
Fair warning: if you're not familiar with binary you might need to go over this section a few times. I'm going to do my best to bring it down to earth, however, because you very well may be asked a question about bitwise operations in an interview someday. You

should, at the very least, be able to recognize the context so you can squeeze the interviewer for some help, rather than shrugging!

# 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:

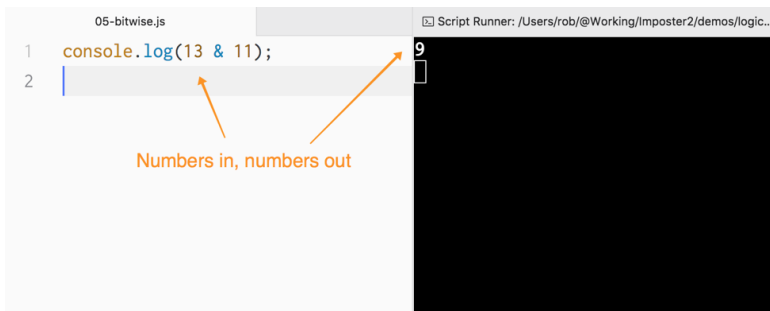


We have two binary numbers: 13 on top and 11 on the bottom. If we wanted to perform an AND operation on the two numbers, we would start at the right, evaluate the digits in the "units" column (1s) and then make our way to the 2s, 4s, 8s, evaluating each pair



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:



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

Numbers in, numbers out

9

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

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:



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

Script Runner: /Users/rob/@Working/imposter2/demos/logic...
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 now!

In JavaScript, I can use the following bitwise operations:

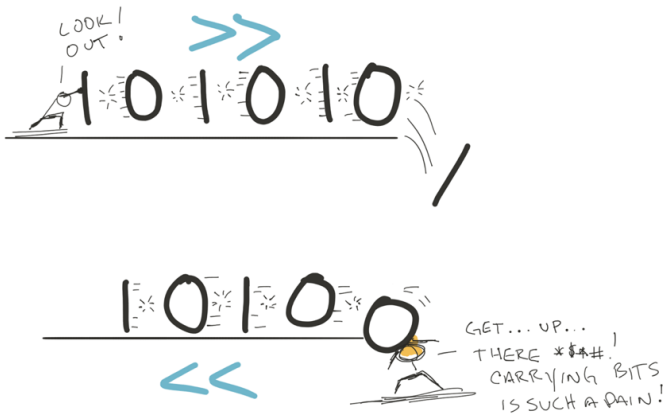
- AND: &
- OR: |
- XOR: ^
- NOT: ~

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 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

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

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 *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

$$\begin{array}{r}
 2 \\
 10 \\
 30 \\
 100
 \end{array}
 = 142$$

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

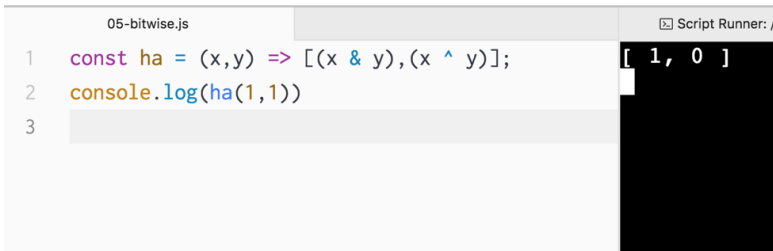
The point of all of this is that this is the *same* process we need to go through with binary addition. We'll get to that in just a second. First, we need to get comfortable using bitwise operators with our half and full adders.

## 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 with a file named '05-bitwise.js'. The code contains three lines:   
1. `const ha = (x,y) => [(x & y),(x ^ y)];`  
2. `console.log(ha(1,1))`  
3. (empty line)  
The 'Script Runner' panel on the right shows the output of the code: `[ 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 cryptic, doesn't it! Let's step through it:

1. I'm using a lambda that takes 3 arguments: `x`, `y` and the carry bit, `c`.
2. 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`.

3. 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 tbl
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())
```

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 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 *the same* process we use in the decimal system above.

Does it work? Let's see:

```
06_bitwise_addition.js | Script Runner: /Users/rob@Work...
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. Before 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 regard 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

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.

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 relevant – 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).

We've blissfully ignored the other half of the world of numbers by focusing on addition and multiplication. Let's fix that now by hav-

ing a look at how negative numbers are handled in the binary world.

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  $\sim 0010 = 1101$ :

$$0101 \ (5)$$

$$\underline{1101 \ (-2)}$$

$$0010 \ (2?)$$

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:



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:

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

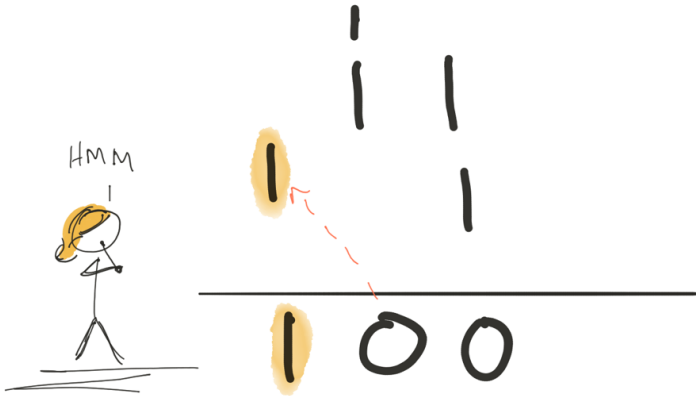
Using Two's Complement also allows us to avoid the negative 0 problem. We can see this by going through the same steps, but with 0:

4. Convert 0 to signed binary, which is 00
5. Flip the bits, yielding 11
6. Add 1 to it, yielding 00

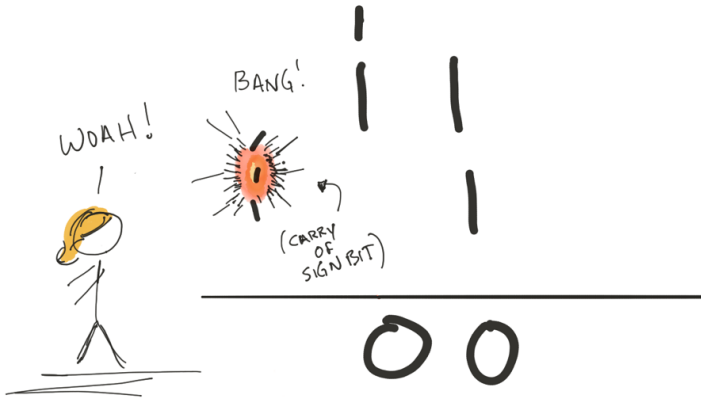
Boom! This might not seem like a big deal to you, but if you were designing computer hardware you'd be quite upset if you had to deal with positive and negative 0. JavaScript programmers occasionally get upset about it too!

## 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 manually until it sinks in. This is precisely 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* these binary operations have become so important to us in the Information Age.

# BIG O NOTATION

**H**ave you ever written some code that you were rather proud of? It's a pretty good feeling to see a test pass so you can move on to solving another problem or, better yet, writing more tests.

Any coder can solve a problem given enough time, *solving it well*, however – that's what we want to do! But what does that even mean?

Simply put it means that your code does what the spec requires, it can scale, and it's written in a way that other developers will understand *your intentions* in the future. This chapter focuses on the second part of that sentence: the "it can scale" part.

How can you demonstrate that your code can scale using something more than just waving your arms? **You can use Big-O.**

# THE CODE

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video walkthroughs of the algorithms you see in this chapter and others from here.

Big-O notation *mathematically describes the complexity of an algorithm in terms of time and space*. It is intimidating and quite a few developers I've encountered (coworkers, at conferences and so on) will instantly withdraw from a conversation at the first mention of "Order N" or the like.

It can come off as elitist if you examine someone else's code and casually drop Big-O into a conversation. It can also mean that you'll pass your next interview when your interviewer asks you about the complexity of some code you've written and whether you can improve it to be **log n vs. n<sup>2</sup>**...

## A WORD ABOUT DATA STRUCTURES

Before we get going, I think it's worth addressing the examples you're about to see. I'll be using a basic array for each one of them, and if you're not a JavaScript developer you might think to yourself *that's ridiculous! I would never write code like that!* And I would understand.



.NET, for example, has a remarkable amount of list types, both generic and non, that allow you to write rather powerful, exact code for the task at hand. Elixir and other functional libraries allow you to choose freely between an enumerable, a list, a dictionary, or a map.

Your choice of data structure depends squarely on the type of data you're working with and then what you're trying to do with that data. *But do you know why you're making these choices?* Do you know why these structures exist in the first place?

Each data structure in .NET (or Java or Elixir) was created for use in a particular type of algorithm. If you weren't formally trained in data structures (as I wasn't), you would just pattern your choice from what you've read in blogs or been told by a more senior developer. This works fine for many, but you're reading this book because you want to go a bit deeper, to learn the concepts that underly so many decisions you've made in the past.

To ask "why this data structure?" is to also ask "how are we using this data", which then naturally leads to "which algorithms are we going to implement?". The answers to these questions are somewhat interdependent, which presents a problem for me in terms of writing this book in that *I have to start somewhere*.

**So I'm going to level the playing field.** All we're going to use in this chapter is an array and some integers - much like any coding interview. I'm doing this because I want to be able to focus on *complexity* as a means for making educated choices about algo-

rithms and, correspondingly, the data structures you choose to work with.

We'll get into data structures and algorithms later in the book, for now I ask you to suspend what you know about various list types and to just go with the flow. Yes, the examples are contrived, but only because I didn't have a choice!

Onward...

## A SUPER SIMPLE FIRST STEP: $O(1)$

Let's just jump right in and see what kind of mess we can make here, shall we? Let's say I have an array of 5 numbers: **[1,2,3,4,5]**.

Now, let's say I ask you to get the first number from this array. Obviously if you look at it can you say "oh - sure that's a 1". A program doesn't have eyes, however, so it needs a way to pull that number out.

Being the smart person you are, you decide that it's a simple matter of using an index – the very first index as a matter of fact. We're using JavaScript here so we can get the value thus:

```
const nums = [1,2,3,4,5];  
const firstNumber = nums[0];
```

Now comes the question: *how complex was that operation?* If you were like me just a few years ago I would have said “not complicated at all – I just took the first element of the array”. This is a correct answer, but we can be more specific by thinking about complexity in terms of *operation per input*.

We have 5 inputs here because there are 5 elements in the array. How many operations did we need to perform on these inputs to derive a result for our algorithm? *Only one* as it turns out. How many operations would we need to perform if there were 100 elements in the array? Or 1000? Maybe 1,000,000,000? Still: *only one*. We just take the very first element at index 0.

We can capture that inelegantly long paragraph in a more scientific way by saying that our algorithm was “on the order of 1 operation for all possible inputs”, or better yet:  **$O(1)$** . This notation is pronounced “order 1” or, more casually, *constant time*. For all inputs to our algorithm there is and will always be **only one operation required**.

As you’ve probably guessed,  **$O(1)$**  algorithms are pretty efficient and also quite desirable!

## ITERATIONS AND ORDER(N)

Now that you can figure out how to pull the first item from our array, let’s try something more complicated: *let’s sum the items in the array*.

Again, let's use some code:

```
const nums = [1,2,3,4,5];
let sum = 0;
for(let num of nums){
  sum += num;
}
```

Now we get to ask ourselves the same question: *how many operations do we have per input to our algorithm?* This time the answer is different: we must add each number to a running sum, so we have to **operate on each one**. This means *one operation per input*.

Using Big-O time notation, we would say this is **O(n)**, or “Order n” where **n** is the number of inputs. This type of algorithm is also referred to as “linear”, or that it has “linear scaling” (think of describing a line on a graph: **y = 2x** or something of the sort).

## Analysis

This type of scaling is common when you calculate a result by iterating over a collection of values like we're doing above. It's a simple way of doing things, but it has implications in terms of complexity which we can see if we ask ourselves a simple question: *how does this scale?*

As opposed to our **O(1)** algorithm, our CPU is doing a bit more work in our summing operation above – in fact it's doing *n* times the amount of work of our **O(1)** algorithm. If we have an array of

10 items, it won't matter; but what happens if our array has 1,000,000 elements? Now we need to worry a bit as we have 1,000,000 operations to perform.

It might seem academic, but it's a good question to ponder whenever you write a loop (or worse: a *nested loop* which we'll address in the next section): "is there a way I can make this algorithm a bit more efficient?". For our summing operation - *no*, there isn't. We must consider every element. For other things, however, sometimes a bit of math might do the trick.

Consider our very same array: **[1,2,3,4,5]**. What if I told you to write a summing function for a sorted, contiguous array of integers that starts with the number 1? Ahhhh this time things are different as we know a bit more about our input.

We can use a bit of interesting math here, specifically something that Carl Friedrich Gauss figured out while in grade school. If you want the full explanation, follow the link above; otherwise I'll just get to it.

We can use this equation to figure out the sum of the series  $[1...n]$ :

$$S = n(n + 1)/2$$

Plugging this into our example array, we would have:

$$5(5 + 1)/2 = 15$$

How do we know what  $n$  is? Simple! It's the very last element of the array. Now we can change our algorithm a bit:

```
const sumContiguousArray = function(ary){  
  //get the last item  
  const lastItem = ary[ary.length - 1];  
  //Gauss's trick  
  return lastItem * (lastItem + 1) / 2;  
}  
const nums = [1,2,3,4,5];  
const sumOfArray = sumContiguousArray(nums);
```

The answer here will be 15!

Notice that we're not running an iteration? That makes our  **$O(n)$**  algorithm *a lot faster* as we're now in constant time thanks to Mr. Gauss. You might be thinking "but wait, if we have to figure out the length of the array *and* pull off the last element *and* run the calculation – isn't that  **$O(3)$**  or something?".

This is something we should get straight about Big-O right up front: yes that would be the literal complexity, but we're not interested in that. All we care about is that it's *constant time*, meaning that the time complexity will not change based on the number of inputs.

Constant time algorithms are always referred to as  **$O(1)$** . Same with linear time. An algorithm might literally be  **$O(n + 5)$**  but in Big-O that's just  **$O(n)$**

## THE NOT-SO-GOOD APPROACH: ORDER $(N^2)$

Let's up the complexity a bit. It was suggested in today's standup that I don't make good integer arrays, and that it's possible that I might have duplicated one of the elements. I insisted that I did not! However, we decided it might be a good idea if you were to create a routine to verify this.

There are some simple solutions to this, and some that are quite a bit more efficient. Let's start with the simple, brute force solution which require a nested **for** loop:

```

const hasDuplicates = function (nums) {
  //loop the list, our O(n) op
  for (let i = 0; i < nums.length; i++) {
    const thisNum = nums[i];
    //loop the list again, the O(n^2) op
    for (let j = 0; j < nums.length; j++) {
      //make sure we're not checking same number
      if (j !== i) {
        94;
        const otherNum = nums[j];
        //if there's an equal value, return
        if (otherNum === thisNum) return true;
      }
    }
  }
  //if we're here, no dups
  return false;
};

const nums = [1, 2, 3, 4, 5, 5];
hasDuplicates(nums); //true

```

It works, but it's not ideal. Here we're iterating over our array, which we already know is **O(n)**, and another iteration inside it, which is another **O(n)**. For every item in our list, we must loop our list again to calculate what we need. This type of complexity is **O(n<sup>2</sup>)**, or "Order n squared" and as you can imagine, it's not very efficient and is considered quite inelegant.

## Analysis

The big problem is this: if we have 1000 numbers in our list, we'll have  $1000 * 1000 = 1,000,000$  operations! That's bad.



If you remember only one thing from this chapter, let it be this: *there is almost always a better way!* In fact, this is one of those things where once you see it, it's hard to *not see it* again. Nested loops working over the same collection - always  **$O(n^2)$** .

These problems often appear in coding interviews, where the simplest answer is to brute force your way through an “ **$n^2$** ” solution, trying to figure out how it's possible to do things better.

Which you might be wondering right now... so let's play Big Job Interview!

Me: *well, this solution works, true, but I wonder if we can do better? Is there a way we might be able to improve on your  $O(n^2)$  **solution?***

You: *possibly –*

Me: *sure, chicken. If it makes you feel better to be told an answer as opposed to figuring it out yourself... then...*

You: *passive-aggressive bullying isn't cool Rob*

Me: *sorry... I don't know the answer either...*

In a few sections we'll revisit this problem and see if we can improve it.

## REFINING TO ORDER( $\log N$ )

We've been relying on a brute force approach to work with our array up to this point which works, but as I keep saying it's 1) inefficient and 2) inelegant. We want to do both of those things, so we don't just look the part of super stellar programmer!

For the next task, let's search through our array for a given number. If you don't have a CS degree (having a few search algorithms at the ready) – how would you go about such a thing. Once again: *yes, we have eyes*, but a program doesn't, so we need to have a systematic (and deterministic) way of finding a given value in a collection.

The simplest way to go about this process is to think about the *time complexity* of the operation, which is what we've been doing. Instead of offhandedly saying “we'll have to search the array top to bottom” or “just iterate until you find the number”, we can now use a simpler, more direct term:  **$O(n)$** .

If we use a brute force approach, we just loop over the entire array *once* to find the number we're looking for. This means that in the worst-case scenario (which is how we think when we use Big-O) we must perform one operation per input.

Can we do better than a linear  **$O(n)$** ? It turns out that yes, we indeed can if we make a few assumptions.

**Assumption 1:** we've noticed that the list is sorted; can we assume that it is sorted for this exercise? For illustration, I'll say *yes, we can*.

**Assumption 2:** I didn't mention anything about *in-place*; can we use more than just the given array? *Sure, why not*.

Given these two assumptions, we can use a clever algorithm called *binary search*. We'll get into the details later, but this algorithm is performed in a set of simple steps where we split the list in half, discard the half we don't need, then go on splitting until we have the value we're looking for. By splitting the list and discarding the half without our target value each time, **we're able to find the number we're looking for in far fewer operations**.

This type of algorithm is called "divide and conquer" and works on the mathematical principle of *logarithms*. If you're like me (just a year ago) you haven't had to think about logarithms in quite a few years and, perhaps, you've forgotten how they work.

Binary search is one of the simplest ways to improve the performance of our applications. I'll explain why (and how) in just a second but before I get there, we need to do some math.

## QUICK LOGARITHM REVIEW

In essence, logarithms help make working with exponents a bit easier. Let's do some quick math and hopefully things will be a bit clearer. What if I asked you to figure out what **x** is here:

$$x^3 = 8$$

To answer this you simply need to take the cube root of 8 to get 2. Simple enough! What if we changed things up now:

$$2^x = 512$$

This makes things a bit harder, doesn't it? How would you solve for **x** here? If you're a math person you already know the answer; but if you're like me and have forgotten most of your math the answer wouldn't exactly pop out at you. What we need here are *logarithms*.

If we used logarithms we could rewrite this equation like this:

$$\log_2(512) = x$$

This equation says "log base 2 of 512 equals some number x". The key here is the "base 2" part. It means that we're *thinking in 2s*, and we want to know *how many times to multiply 2 to arrive at 512*. That's all logarithms do! How do we solve this equation? Duh! With a calculator of course!

The important part here is the recognition that logarithms basically deal with splitting a given number into some type of *base*. The default base is 10, which means you should be able to guess at the answer of this expression:

$$\log(100) = x$$

The question you're asking here is *how many times do you multiply 10 by itself to arrive at 100?* The answer is 2!

OK, so let's get back to the problem at hand. We're splitting a list of numbers in half, continually, until we get down to the single value that we want. The key to this is the name of the algorithm itself: *binary* search... splitting things *in half*.

We're dealing with 2s which means our logarithm will be base 2. So this operation can be expressed like this:

$$\log_2(n) = x$$

Which brings us to an interesting thing about Big-O which I'm repeating now and I'll be repeating later: **the exact math doesn't matter**. What matters is that this is logarithmic - nothing else. We

can even do away with the base and **x** - we just care that the operation is  $O(\log n)$ .

## Analysis

If we were to use an  $O(n)$  scan on our list we would have had to perform  $n = 5$  operations to find our target number 2. By splitting the list and evaluating the number at the split (thus discarding the half without our number), we could bring the number of operations down to 3.

You might be thinking "hey wait a minute! Splitting *and* evaluating is 2 operations!" and this is true - but they are 2  $O(1)$  operations so we just consider the entire thing to  $O(1)$ .

Cutting from 5 operations down to 3 isn't terribly exciting. But what if we had 100 total elements? If you whip out a calculator and run some math, you'll find that:

$$\log_2(100) = 7$$

OK that's not exactly true. The number is something a bit more like 6.643856. But we can reason through this in a simpler way.

We're programmers and have a reasonable facility for base 2 "stuff", hopefully. We know that powers of 2 are 2, 4, 8, 16, 32, 64

and finally 128. The number we want is somewhere between 64 and 128, so we *round up* to be safe. **This gives us 7.**

Why do we care? Because we can see how our search algorithm scales! We upped our inputs by a factor of 20, but our operations only went up by 4.

*That* is why we care about doing this entire exercise! We can prove, with math now, that our approach here is better than scanning the entire list for a given number.

## RETHINKING ORDER( $N^2$ ) WITH ORDER ( $N \log N$ )

A few sections ago I asked you to find duplicate numbers in our list of numbers and we decided to iterate over every number (which is  $O(n)$ ) and then do it once more to see if our current number was repeated.

This put an  $O(n)$  operation inside another  $O(n)$ , which we can simplify to  $O(n * n)$  which is  $O(n^2)$ , which is a Bad Thing.

But wait a minute... didn't we just use a nifty algorithm to find a number within our list that was much more efficient than our brute force linear scan? Yes!

So here's a quiz: *what is the Big-O* of using our binary search inside our linear scan? Let's break it down.

# ITERATING AND SEARCHING

We have an  $O(n)$  as before with our array iteration – we know that much. We now have a single operation within the iteration that is  $O(\log n)$ . Putting the two together we have  $O(n \log n)$  which we can just think of now as  $O(n \log n)$ .

So: in terms of time complexity, we have a better algorithm here! It will provably scale better than our original brute force approach as well!

```
const nums = [1,2,3,4,5];
const searchFor = function(items, num){
  //use binary search!
  //if found, return the number. Otherwise...
  //return null. We'll do this in a later chapter.
}
const hasDuplicates = function(nums){

  for(let num of nums){
    //let's go through the list again and have a
    //at all the other numbers so we can compare
    if(searchFor(nums,num) !== null){
      return true;
    }
  }
  //only arrive here if there are no dups
  return false;
}
```



## Common Uses

I mentioned earlier that binary search is one of the simplest ways that we can, as programmers, increase the performance of our applications. This might have sounded weird, but consider how your database finds information when you run a search:

```
select * from users where email = 'test@test.com'
```

This query will work fine for a while, but as your users table grows, this query will slow down. The simple reason why is that the database is a program like any other and doesn't have eyes. Therefore, it has to loop over every record in your table until the condition is met, which in our case is matching a user's email. That process has  $O(n)$  time complexity.

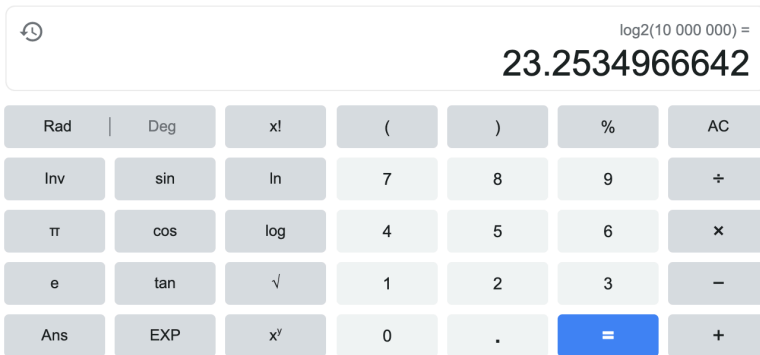
When queries like this slow down the solution is to apply an index. In our case we would pop one on the users table like so (assuming you're using something Postgres, which you should be!):

```
create index idx_user_email on users(email);
```

The index is a sorted list and that's it! Every email in your users table is added there and then sorted. Then, when you run the query above, that index is searched using something very like binary search. This process has  $O(\log n)$  time complexity.

Why do we care about this? Let's pretend we have 10,000,000 users in our database. To match an email on a user, our database would have to perform up to 10,000,000 operations. Not so good.

Using an index, however, our database has to perform  $\log_2(10,000,000)$  operations which we can ask Google to help us solve:



Rounding up, we can say that we improved performance by quite a few orders of magnitude for that one query! 24 operations vs. 10,000,000 – that's pretty good!

## Going Even Further

Our interviewer is digging our style right now and decides to up the game, one more time.

*What if I told you that you could assume there was only one duplicate number?*

Well now... that's an interesting development. Let's consider everything that we just did and see how we can bring this  $O(n \log n)$  algorithm down just a bit more.

The first question we should ask ourselves is *can we do some math* to avoid an iteration? A few sections ago we used Gauss' trick to sum a series of integers, which could help us here. To achieve that, we'll need to perform three operations:

1. Find out what the sum should be for an array without duplicates. For this all we need is the highest number in the array passed to us. Since we know it's sorted, that's the last element and we know this is a constant time thing at  $O(1)$ .
2. Find out the actual sum. This will necessarily be an  $O(n)$  scan.
3. Subtract the non-duplicate sum from the actual sum and that's the number we're looking for! This is also  $O(n)$ .

If we're right, this will bring our algorithm down to  $O(n)$ .

Here's our new algorithm:

```

const findDuplicate = function(ary){
  //sum what we've given
  let actualSum = 0;
  //our O(n) scan
  ary.forEach(x => actualSum += x);
  //get the last item
  const lastItem = ary[ary.length - 1];
  //create a new array
  const shouldBe = lastItem * (lastItem + 1) / 2;
  return actualSum - shouldBe;
}
const nums = [1,2,3,4,4,5];
const duplicate = findDuplicate(nums);

```

Nice work! Our interviewer is impressed!

## THINKING IN BIG-O

By now you should be able to equate certain basic operations in code to a given Big-O. If you practice this for a bit, you'll be able to quickly spot patterns and, ideally, improve them if you know the right algorithm to do so.

Specifically:

- **Random access to a given element in a collection is always  $O(1)$** , depending on how the list is indexed. Arrays, for instance, allow you to access elements randomly if you know their index. HashSets allow you to access if you know what the value is (the hashed value is the index). Dictionaries allow you

random access if you know what the key is, and so on. These types of operations are always  $O(1)$  which means if they are combined with other Big-O, they will remain static or constant time.

- **List iterations are always  $O(n)$ .** If you need to evaluate every item in a list for a given algorithm, this means it will at least be  $O(n)$ . Sometimes you can get around this with some trickery, which I'll discuss later on.
- **Nested Loops on the same collection are always at least  $O(n^2)$ .** Loops within loops ... sometimes necessary, but can usually be improved by thinking about data structures (which we'll do later on).
- **Divide and Conquer is always  $O(\log n)$ .** The very act of dividing a list into smaller sublists is logarithmic. If you have an  $O(1)$  operation once the list is split apart, then the Big-O for the entire operation is  $O(1 * \log n)$  which is just  $O(\log n)$ .
- **Iterations that use divide and conquer are always  $O(n \log n)$ .** Think about looping a list and then executing some algorithm to search for list value or, possibly, to run some kind of sorting.

If you solve a problem by adding another nested loop for every input that you have: that's  $O(n!)$  which is bad and you should probably find another job!

# SPACE COMPLEXITY VS. TIME COMPLEXITY

You may have noticed that I've been using the term *time complexity* a lot in this chapter, as that's what we've been focused on: *how long will it take to do a given operation given  $n$  inputs*. In data analysis speak this is called a *dimension* and is just a way to think about how complex an algorithm is.

There is another dimension that is also important: *space*. In other words: **what are your algorithms resource requirements?**

The same type of Big-O classifications still apply in that we still refer to things such as  $O(n)$  or  $O(1)$  “space”, but the meaning is somewhat different. Space where? The answer is: *it doesn't matter*. “Computer resources somewhere” is all we care about.

However. There is a bit of reasoning we should be able to apply so we can go a bit deeper.

When a program executes it has two ways to “remember” things: the *heap* and the *stack*. I'll get into the details of each later on, but for right now just consider the stack to be the thing we're worried about. It's the thing that remembers the variables *in the scope of the currently executing routine*. When a variable is declared in a block of code, it's stored on the stack. When a block of code goes out of scope, the variables are removed from the stack. Sometimes

the scope is the current block, other times it's the current function or procedure.

Why do we care? The short answer is that **you can easily run out of resources before you run out of time** depending on how you've written your program. The simplest way to think about this is the dreaded "Stack Overflow Exception" which simply means you're executing some kind of loop that has used up every last bit of space on the stack. This can (and often does) happen with a recursive routine as each value remains on the stack until all functions have executed.

Working with strings is another way to cause yourself space complexity problems. For example, if you use a loop to build a string, your space complexity might be as bad as  $O(n * m)$  where  $n$  is the number of iterations and  $m$  is the length of the string. Not so bad if you're just building out memes, but if you're trying to evaluate string patterns in a book... that could be bad.

# DATA STRUCTURES

**N**ow that we understand how to quantify complexity, let's investigate the various ways that we can work with data to increase efficiency. As I mentioned at the beginning of the previous chapter: *algorithms, complexity and data structures work together*.

Arrays are simple to work with but make life hard if you're trying to find a particular value. Dictionaries help with that but are slightly more involved. Hash Sets (or Hash Tables) are lovely because the hash of their value is also their key, so accessing data is constant time and is  $O(1)$  if you know the value you want. From there things get quite a bit more involved.

Some languages offer a plethora of choices when it comes to working with data, but all these choices are variations of (or small improvements upon) a standard, core set of data structures that every programmer should know. That's what we'll look at in this chapter.



# THE CODE

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours video walkthroughs of the structures you see in this chapter and others from here.

## ARRAY

You know about arrays, I'm sure. Unless you work in JavaScript every day, however, it's unlikely you use them much. They're very lightweight and simple to use, but there is also weirdness.

When you declare a non-dynamic array, you must specify its size upfront. This size cannot change. In most modern languages we don't think about this — but in some older languages this is still the case.

Arrays hold values which are referenced by an index. They allow very fast random access, which means you can access any value from an array using a  $O(1)$  routine as long as you know the index.

Arrays are bound to a specific length once they are created. If you need to add an item to an array, the language you're working in will typically copy the original plus whatever value you want to add to approximate dynamically changing the array's size.

**This can be a costly operation**, as you can imagine. Let's see why.

## Resizing

Many languages allow you to dynamically resize an array (Ruby and Python, e.g.) while other, more strict languages, do not (C# and Java e.g.). Ruby and Python allow for dynamic arrays which are basic arrays, but with some magic behind the scenes that help during resizing. C# and Java have different structures (like Lists) built specifically for expanding and shrinking.

Arrays are allocated in adjacent blocks of memory when they are created. There is no guarantee that additional memory can be allocated adjacent to an array if you need to add an element, so it becomes necessary to copy and rebuild the array at a new memory location. This can be costly.

Strings, for instance, are simply arrays of characters. If you append a string with another string value in C#, for instance, an array copy/rebuild needs to happen. This is why working with strings in a loop is typically not a good idea.



## Arrays in JavaScript

JavaScript, however, is an interesting case. Arrays in JavaScript are just objects with integer-based keys that act like indexes. They are instantiated from the Array prototype, which has a few “array-like” functions built into it.

Here is a description from Mozilla:

*Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's length can change at any time, and data can be stored at non-contiguous locations in the array, JavaScript arrays are not guaranteed to be dense; this depends on how the programmer chooses to use them. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays.*

Hey, it's JavaScript.

## Why Choose an Array?

Arrays are the simplest data structure with the least number of rules. Bits of data are stored *in contiguous memory*, so they also have the smallest footprint of any data structure.

If all you need is to store some data and iterate over it, arrays are a fine choice, especially if you know the indices of the items you're

storing. Random,  $O(1)$  access to values in an array is also a great reason to choose an array over other, more “ceremonial” data structures.

The final reason to keep arrays fresh in your mind is The Big Job Search. Coding interviews will almost *always* involve working with an array of values at some level (usually integers), so understanding their restrictions (and advantages) is essential.

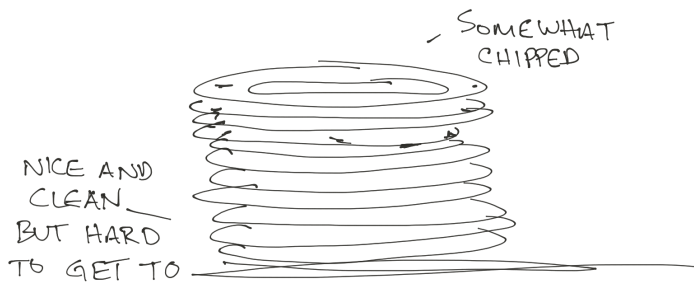
## STACK

I discussed the idea of a stack in the previous chapter when describing a Pushdown Machine. It’s like an array, but it has a few restrictions:

- You can’t access items randomly using an index
- You can only add and retrieve items in the stack from the “top”
- It has three explicit methods: Push, Pop, and Peek.

A term that is often used for the nature of a stack is “Last In, First Out” or LIFO. You *push* an item onto a stack, *peek* to see the value that’s currently on top of the stack and then *pop* that item off the stack.

The simplest way to think about the stack is by visualizing one of my big pet peeves: a stack of plates. The ones on the bottom never get used because we always grab the plates on the top!



## Why Choose a Stack?

Stacks are useful in several surprising ways. They are perfect when you need to know the very last value that was seen in a loop. You might want to know this when you're:

- Reversing a string
- Traversing a graph or a tree
- Checking an opening/closing structure of some kind (such as balanced parentheses in a sentence)

Honestly, the biggest reason you want to know what a stack is (and does) is for interviews. They come up often! You might not use them regularly in your current job, but you never know when they might fit something you're trying to do.

# QUEUE

A queue is also like an array, but with some additional rules:

- You can't access values randomly using an index
- You can only add values in one end and retrieve them from the other
- It has two explicit methods: Enqueue and Dequeue

A queue queues things in a queue, and is described as "First In, First Out" or FIFO. You *enqueue* an object into the "end" of the queue and *dequeue* it from the "front".

It's likely you've used queues often, so I won't labor the point for too long. It's also a word that's quite tough to type, so I'll keep this section short.

An interesting thing about a queue is that you can create one with two stacks. This is one of those interview questions that you'll likely need to know at some point in your career! How would you go about doing this?

The mechanism is obvious: you have an "in" stack and an "out" stack — the tough part is deciding at which point you'll move items from the "in" stack onto the "out" to avoid collisions. If you do it one at a time then things can easily get out of order! Imagine calling *enqueue* 5 times, then *dequeue* once, then *enqueue* again. Your stacks would be a mess!

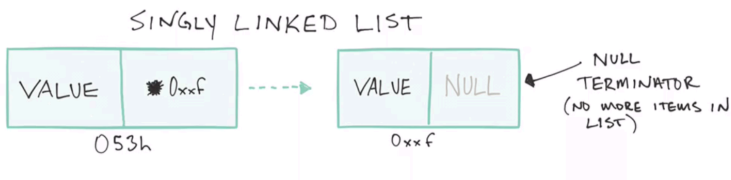
How would you avoid this? I'll leave it to you...

## LINKED LIST

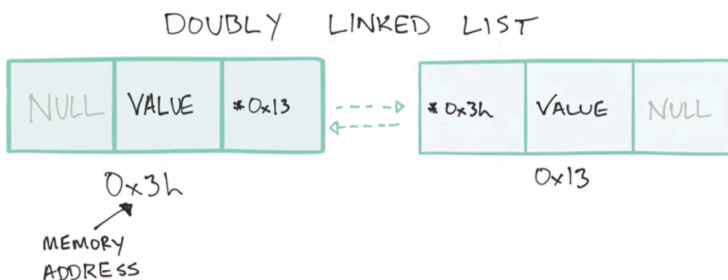
There are two types of linked lists: singly and doubly linked. Both are graphs (which we'll discuss in a minute), which means you can think of them as two-dimensional structures with associations.

A singly linked list consists of a set of "nodes" in memory, that have two elements:

- The value you want to store
- A pointer to the next node in line



A doubly linked list is the same, but contains an additional pointer to the previous node.



## Operations

Nodes in a linked list don't need to reside next to each other in memory, and therefore can grow and shrink as needed.

Their loose structure allows you to insert values into the “middle” of the list by simply resetting a few pointers. Same for deleting a node in a linked list.

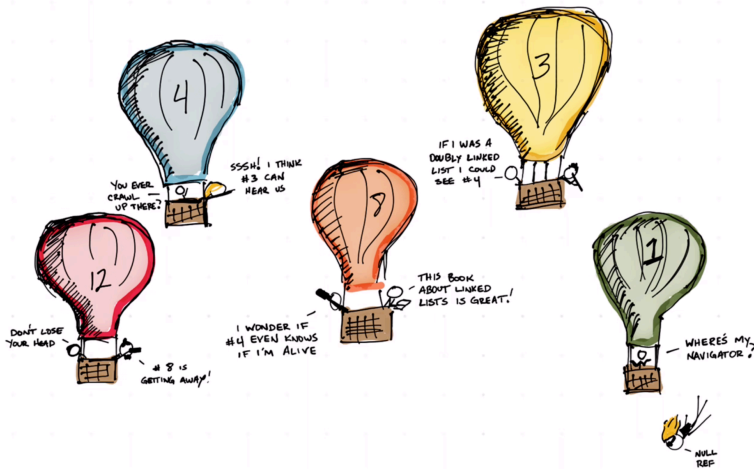
The downside to linked lists is that finding an item means you must traverse the list to get to the item you want, which is an  $O(n)$  operation. Arrays, on the other hand, allow you  $O(1)$  access if you know the index.

Linked lists are null terminated, which simply means if a node's pointer is null, then that signifies the end of the list.

The first item in a linked list is the *head*, the rest of the list is referred to as the *tail*. Some languages refer to the tail being the last item in the list — there is no hard definition so if you hear about



“tailing the list” just think about the end of it and hope for some context.



## Why Choose a Linked List?

The primary reason to choose a linked list over something like an array is *simplicity* and the ability to grow and shrink as needed. Furthermore, because you're in an interview... which might sound horribly snarky but... well, it's true.

Working with them can be a bit weird, but also kind of interesting in that they are self-contained and lightweight:

```

class LinkedListNode {
  constructor(value){
    this.value = value;
    this.next = null; //termination
  }
}
const a = new LinkedListNode(1);
const b = new LinkedListNode(2);
const c = new LinkedListNode(3);

```

Iteration is best thought of as “traversal” because you don’t really know when a linked list will end — so you end up using something like a **while** loop:

```

a.Next = b;
b.Next = c;

let thisNode = a;
while(thisNode != null){
  //do something
  //...
  //traverse
  thisNode = thisNode.Next;
}

```

## HASH TABLE

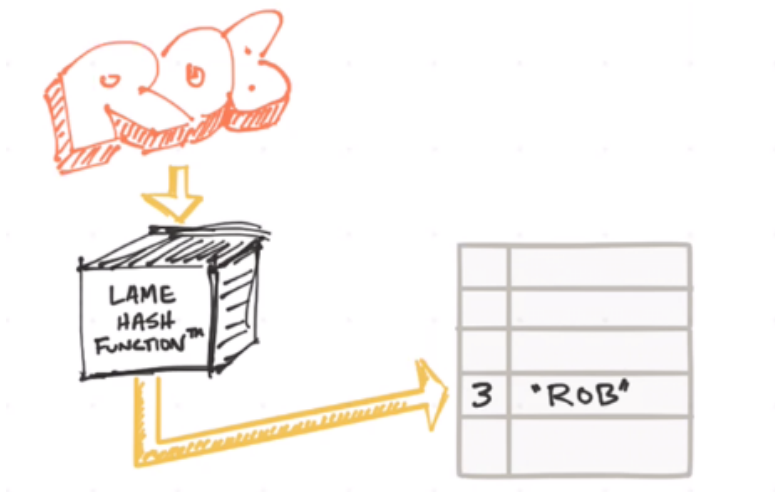
Arrays are fast for reading data, linked lists are good for writing data and having more flexibility. A hash table is a combination of the two.

A hash table stores its data using a computed index, the result of which is always an integer. The computation of this index is called a *hash function*, and it uses the value you want to store to derive the key:

```
const value = "Rob";  
const myLameHashFunction = function(val){  
  return val.length;  
}
```

Most modern programming languages have a hash function you should use; don't create your own. I'll go into why in just a second.

Once you have a key, you can store your value. The key/value pair is referred to as a *bucket* or *slot*.



## Speed

Hash tables are great when you want quick access to certain values. Because their value is also their index, hash table reads are  $O(1)$ . This can be complicated, however, if a hashing function is overly complex. You will typically leave this to whatever framework or language you're using, so assuming  $O(1)$  is fine.

Similarly, adding values to a hash table does not involve (typically) any traversals — you just hash the value and create the key. This leads to excellent performance, however in the real world this doesn't happen that often.

Even the best hashing algorithms will create duplicate keys (called collisions) if the data size is large enough. When this happens, your

reading and writing can be reduced to  $O(n/k)$ , where  $k$  is the size of your hash table, which we can just reduce to  $O(n)$ .

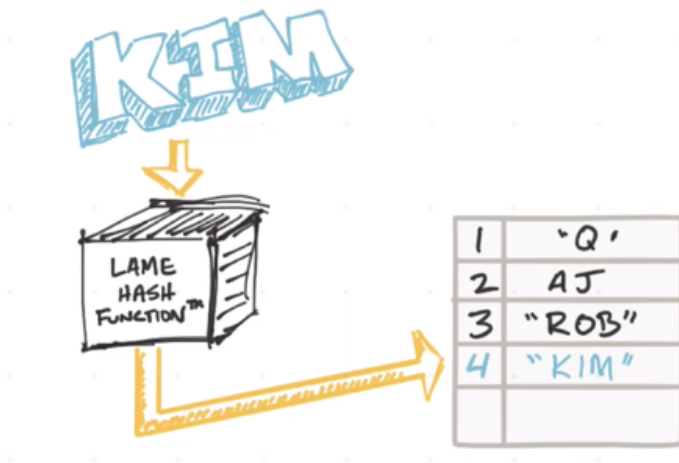
Collisions will likely happen in any hash table implementation, so most implementations have built-in ways of handling these. Let's examine two common ways to deal with collisions: open addressing and separate chaining.

## Open Addressing

Open addressing resolves a collision by finding the next available slot and dropping the value there.

Let's assume I'm using my *Lame Hash Function* with a new name: "Kim". The key produced will be a 3 since my *Lame Hash Function* only uses the length of the value instead of something more intelligent. The key produced (3) will collide with my existing entry for "Rob", which is also a 3. Using open addressing to resolve the collision, "Kim" will be added to the first slot at the end, which is 4.

To find "Kim" in the table, open addressing dictates that we do an  $O(n)$  scan from index 3 (where "Kim" should be) and then work our way down until we find the value.



This presents some interesting problems — what if “Doug” and “Dana” want to join our list? They can, but their indexes will be 5 and 6. It spirals from there.

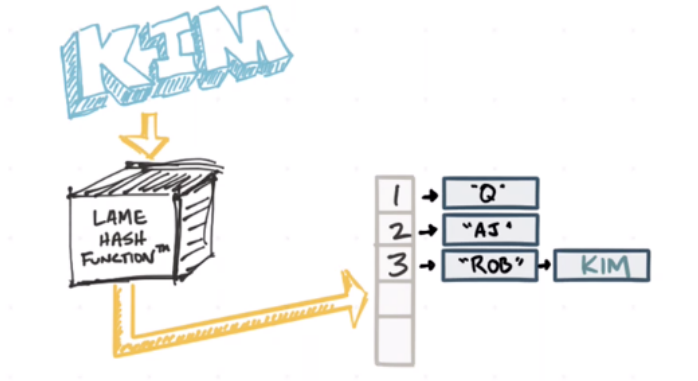
This is called *clustering*, and it can quickly turn a lovely, fast hash table into a simple  $O(n)$ , not fast hash table.

## Separate Chaining

Separate chaining involves combining two data structures that we’ve already looked at: arrays and linked lists.

When a key is hashed, it’s added to an array that points to a linked list. If we have only one value in our hash table for a given key, then we’ll have a linked list with only a single element.

However, in the case of key 3, the linked list can expand easily and accommodate “Kim” as well:



## Why Choose a Hash table?

Hash tables are one of the most common data structures you'll find in modern languages because they are fast. As long as the hashing algorithm is comprehensive and capable, the hash table will be  $O(1)$ .

One of the main drawbacks, however, is again the hashing algorithm. If it's too complex, then it will take longer to run, and you will still have  $O(1)$  read/write, but it could actually be slower than executing an  $O(n)$  over an array, for instance.

# DICTIONARY

A dictionary is exactly like a hash table except it has a unique key for accessing a given value. This has an advantage over hash tables in that you *can't have a dictionary with the same key* — so collisions are not something you need to worry about.

Given that a key is guaranteed to be unique, dictionaries provide  $O(1)$  access to any element, as long as you know what the key is. The downside is that dictionaries can be a bit larger than hash tables and therefore have increased *space complexity*. Most of the time this isn't something you need to worry about.

Dictionaries are ubiquitous in programming, even more so than arrays and hash tables. If you're a JavaScript developer you might be wondering about this assertion — after all, the only type of list you get to work with is an array... which isn't exactly true.

Under the covers, your array is really a dictionary with integer keys, as is a plain old object:

```
var nums = [23,4,42,15,16,8,3];  
//this array is really a dictionary  
//which is also used to represent an object  
nums[0] == 23; //true  
nums[1] == 4; //true  
nums[2] == 42; //true
```



# TREES

Trees are another type of graph (which is a collection of associated nodes that I'll get into in a just a minute) with a hierarchical structure. As opposed to linked lists, which are linear, trees can have 0 or more child nodes.

Every tree has a single root node. Every child of the tree descends from this root node and has *only one parent*.

You work with tree data structures every day: your computer's file system is a tree:









































Folders	Folders	Documents
          	          	 2007-10-26-microsoft-subsonic-and-me.md  2008-01-05-me-gusta-los-angeles.md  2008-02-27-creating...s-with-linq-to-sql.md  2008-11-22-salmin-hawaii-in-a-bowl.md  2009-10-13-thinking...some-tips-for-ya.md  2009-10-24-hello-tekpub.md  2010-11-17-open-id-i...ty-that-happened.md  2010-12-03-video-encoding-rack-attack.md  2011-01-03-marla-singer-didnt-exist.md  2011-02-07-the-sup...assive-freakshow.md  2011-02-16-and-i-shall-call-it-massive.md  2011-11-01-how-to-b...o-amazon-nightly.md  2011-12-10-a-little-bi...ent-into-this-one.md  2012-02-25-nodejs-...ons-and-your-app.md  2012-02-28-someon...save-us-from-rest.md  2012-02-29-alt-tekp...iving-a-restful-api.md  2012-03-03-moving-...y-into-machinery.md  2012-03-08-somethi...d-something-new.md

Figure 1 The Author's Jekyll Blog

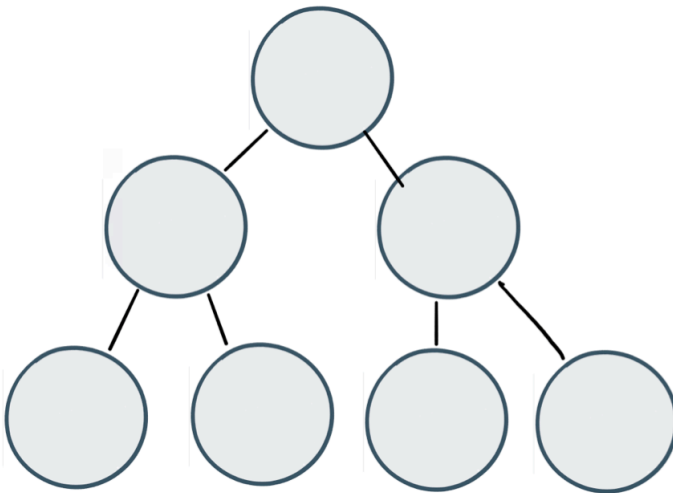
Using a tree data structure usually involves traversing using recursion, which can be expensive in terms of *space complexity*. Every

time you recurse a child node, the stack has to store the values for that scope, and you could run out of space.

There are other ways to traverse a tree by iterating using a Queue and a Stack — I'll get to that in the Algorithms chapter, specifically Breadth-first and Depth-first traversal.

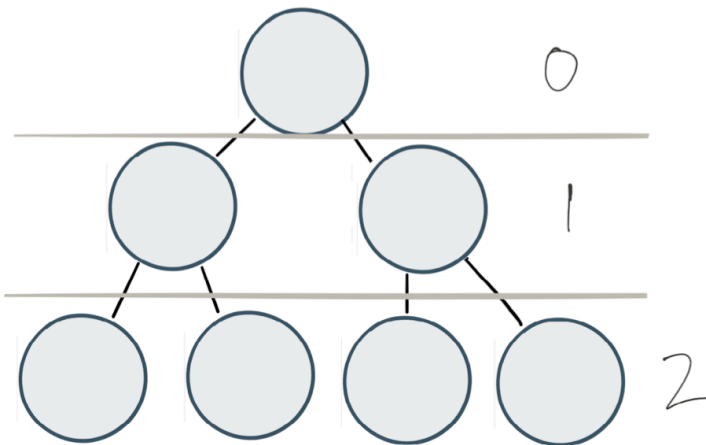
## BINARY TREE

A binary tree is something you've likely seen. It's a type of graph, but with some rules applied, which are straightforward: each node can have 0, 1, or 2 *child* nodes, but only 1 parent:



Binary trees can represent several interesting things, including *decisions*. Each node represents a given state that is the result of a yes or no decision.

Another interesting aspect of a binary tree is that each level of the tree represents a *logarithmic value*:



At level 0 we have  $2^0 = 1$  Node. At level 1 we have  $2^1 = 2$  and finally  $2^2 = 4$  nodes.

You can also visualize the *divide and conquer* search algorithm we used in the last chapter with a binary tree. Each split operation represents another level of a binary tree.

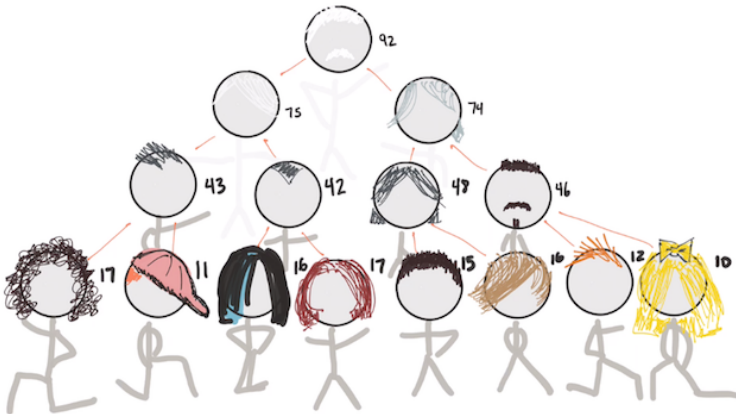
# HEAP

A heap is an inverted tree that is a set of interconnected nodes that store data in a particular way, which is called the *heap property*.

Every child node belongs to a parent node that has a greater priority (or value, whatever) — this is called a max heap. A min heap is the opposite: every parent has a value less than each of its children.

Put another way: if we're dealing with a max heap then every node on a top level has a greater value than every node on the next-level down.

We can see this in a recent family picture of mine, where we all posed based on our ages:

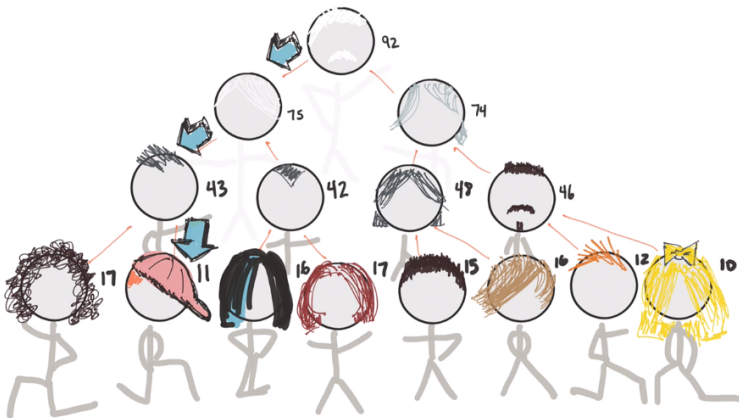


*We couldn't afford a camera, so we had to draw it out*

## Usage

A heap (and its variants, which I'll get to later) can be used in any algorithm where ordering is required. Arrays are random and allow random access to any element within them. Linked lists can change dynamically but finding something within them is  $O(n)$  (linear); heaps are a bit different.

You can't do  $O(1)$  random access and a single node knows nothing about its children. This means you need to do some type of traversal to find what you're looking for. Given the structure of the tree, however, finding things is considerably easier than with a linked list:



We found Tommy, age 11, easily here. However, we could easily have had to traverse over a few times if he was on the end there next to Jujubee, age 10.

So, what are heaps good for then? It turns out they're wonderful if you're doing comparative operations — something like "I need all people in this heap with an age greater than 23". Doing that in a linked list would be quite slow — the same with an array and a hash as no order is implied.

## Why Choose a Heap?

Heaps are used in data storage, graphing algorithms, priority queues and sorting algorithms. In many languages you'll find data structures that will give you the same kind of ordered structure if you choose your keys wisely.

For instance: in C# you could use a **SortedList** or a **SortedDictionary**, storing objects with parent/child associations.

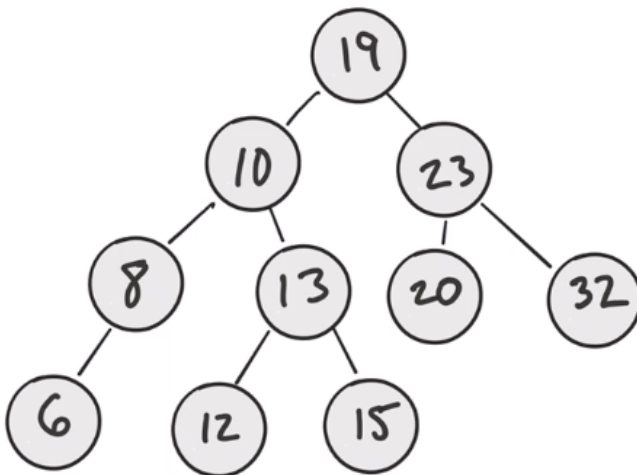
The primary advantage of heaps is performance. They're very fast when it comes to searching as they are pre-sorted and don't involve the extra step of an  $O(\log n)$  split — it's just a traversal.

# BINARY SEARCH TREE

A binary search tree is just like a heap in that it is organized based on the values of the nodes, with one major addition: there is also a left to right value priority.

The rules are:

- All child nodes in the tree to the right of a root node must be greater than the current node
- All child nodes in the tree to the left must be less than the current node
- A node can have only two children



## *Binary Search Tree*

The advantage of a binary search tree is, obviously, searching. It's very easy to find what you're looking for as you'll see down below. The downside is that insertion/deletion can be time-consuming as the size of the tree grows.

For instance, if we remove 13 from the tree above, a decision needs to be made as to which node will ascend and take its place. In this case, I could choose 15 or 12. This operation seems simple on the face of it, but if 15 had children 14 on the left and 16 on the right, some reordering would need to happen.

## **Searching A Binary Search Tree**

If you know the rules, finding a value in a BST can be done recursively, or by using breadth-first or depth-first search (which we'll get to in a minute):

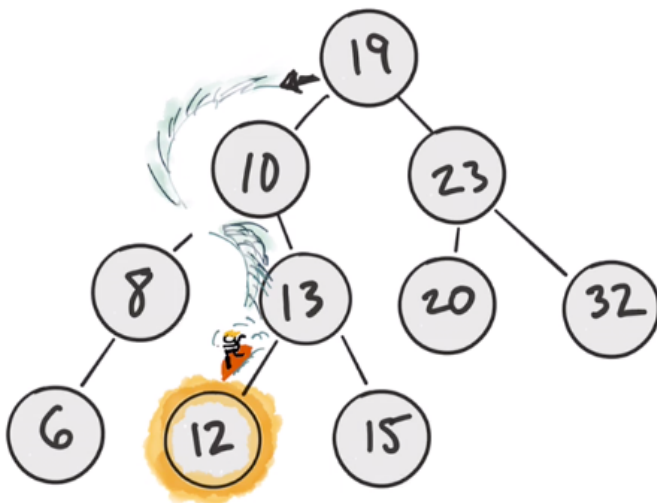
- If the value is less than the current node you're on, go to the left child node.
- If the value is greater, go to the right child node.

Do this until you find what you're looking for.

For example, here we need to find the number 12. Our root node is 19, so we traverse to the left because  $12 < 19$ . When we reach 10, we traverse right because  $12 > 10$ .

Finally, we come to 13, so we go to the left again and arrive at 12.

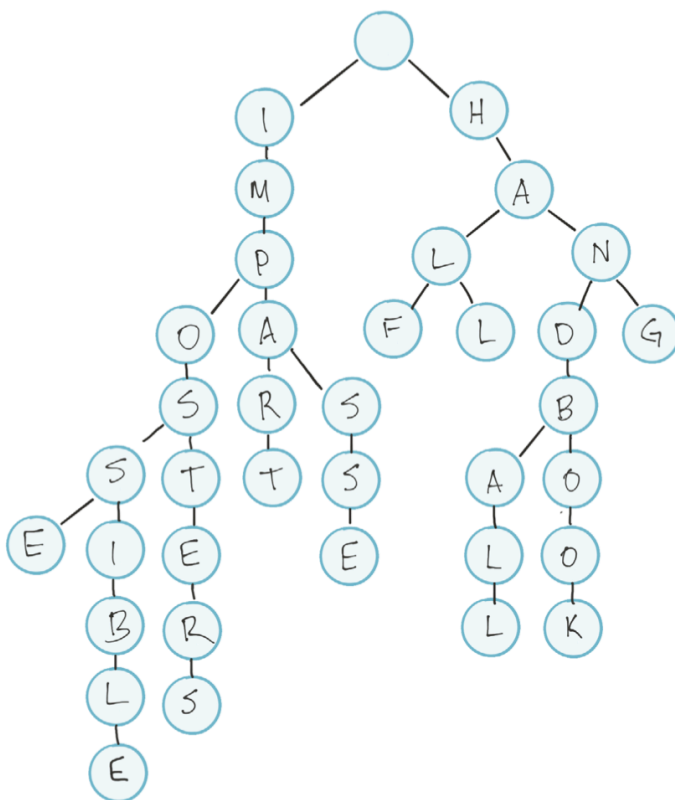




## DIGITAL TREE (OR TRIE)

A digital tree (or “trie”, which most pronounce “try” even though the term comes from **retrieval**) is a specialized tree used in searching, most often with text. In many cases, it can outperform a binary search tree or hash table, depending on the type of search you’re doing.

Tries allow you to know if a word (or part of a word) exists in a body of text. The easiest way to understand a trie is to see one, so here goes:



This trie has an empty root node, and from there letters are added as child nodes. The power of a trie is evident with the prefix “imp” — in this section of the trie there are 6 distinct words represented by 20 total nodes:

- Imp

- Imposter (and Imposters)
- Impossible
- Impart
- Impasse


The advantages of this structure are speed and space. Finding a word in this structure is  $O(m)$ , where  $m$  is the length of the word you're trying to find.

Tries also have a major advantage when it comes to *space complexity*. Common prefixes are reused, so repetition within the structure is kept at a minimum.

Finally, the major advantage of a trie is that you're able to search the structure for *partial matches* based on a prefix. This kind of thing is great for word completion.

Have you ever wondered how code completion works in your favorite editor, or how Google can do the below so quickly?





rob likes **omaha**

rob likes **keybank**

rob likes **to yell**

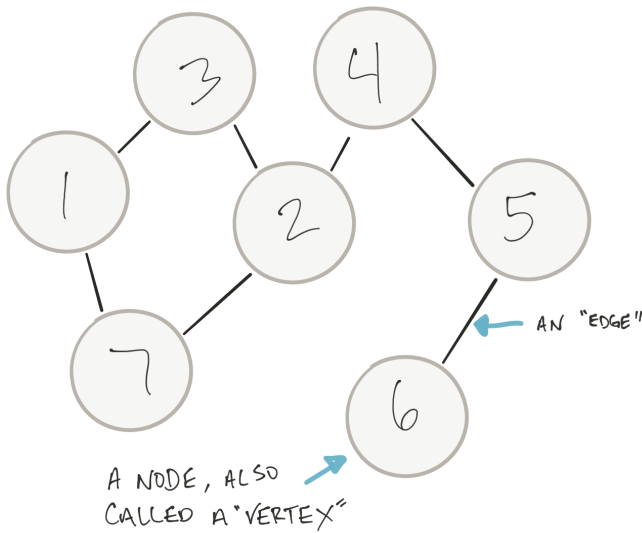
rob likes **to route**

That's right: using tries.

## GRAPHS

I've been mentioning graphs quite a lot in this chapter, and it's time, finally, to dive in and get to know these data structures a bit more.

Graphs are one of the most useful and most *used* data structures in computer science. In short, a graph is a set of values that are related in a pair-wise fashion. Again, the easiest way to understand this (if it's not intuitive) is to see it:



As you're probably figuring — graphs can describe several things. Let's look at a few.

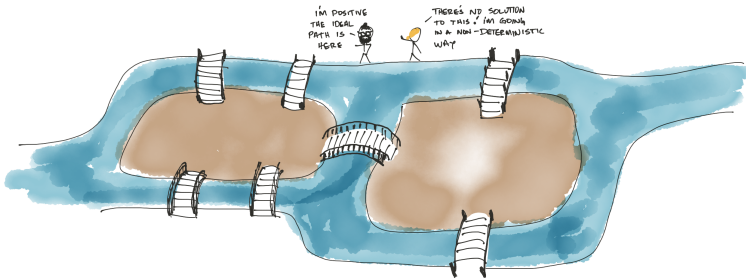
## Graph Theory and the Bridges of Königsberg

If you took calculus in school, you probably learned about the origins of graph theory with Leonhard Euler's Seven Bridges of Königsberg problem:

*The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges. The problem was to devise a walk through the city that would cross each bridge once and only once, with the provisos that: the islands could*

*only be reached by the bridges and every bridge once accessed must be crossed to its other end. The starting and ending points of the walk need not be the same.*

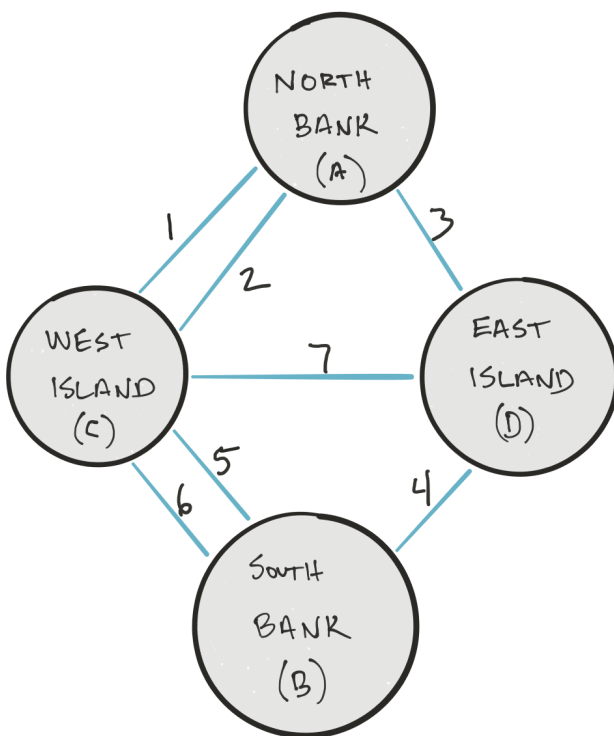
It might be easier to visualize the problem ... which gives me another reason to draw!



### *The Seven Bridges of Königsberg.*

In trying to solve this problem, Euler reasoned that the route on land was not important, only the sequence in which the bridges were crossed. Given this, he could relate the land to the bridges in the abstract, using the idea of nodes that are accessible by an edge.

In other words, *a graph*.



Now that we have our graph, we can restate the problem in terms of a graph:

*Can you visit vertices A, B, C and D using edges 1 through 7 only once?*

Before reading on, take a second and see if you can solve the problem just tracing your finger across the page. Or draw it out on a paper yourself and see if you can trace a line using a pencil or pen, visiting nodes A through D using edges 1 through 7 only once.

In mathematical terms, a *simple path* accesses every vertex. An *Euler path* will access every edge just once — that's the one we want, an Euler path.

## The Solution

There is an algorithm we can use to solve this problem, which is to determine the number of degrees each vertex has and apply some reasoning. A degree is how many edges a given vertex has, by the way.

Euler reasoned that a graph's degree distribution could determine whether a given edge must be reused to determine the path. His proof, in short, is that a graph must have either zero or two vertices with an odd degree to have an Euler path (a path which visits each edge just once).

Let's tabulate the seven bridges graph:



VERTEX	DEGREES
A	3
B	3
C	5
D	3

Here we have four vertices with odd degrees, which tells us that there is no Euler path and, therefore, that the Seven Bridges Problem has no solution.

## So What?

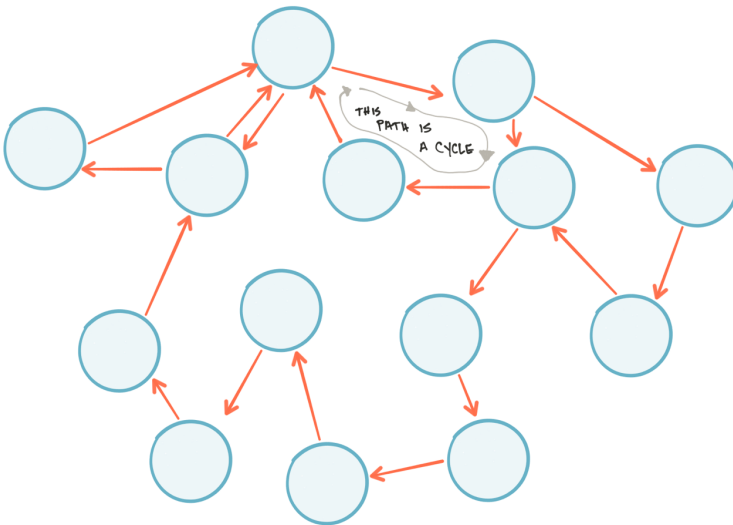
I wouldn't blame you if you're wondering why I've included the Seven Bridges Problem in this book as well as a discussion of Euler. These seem to be more math-related than anything in computer science, don't they?

In your next job interview you may very well be asked how you would solve some algorithmic problem that you (hopefully) will

recognize as graph based. Fibonacci, Traversal, Balancing, or a Shortest Path problem — if you can spot a graph problem, you'll have a leg up on the question.

## Directed And Undirected Graphs

There are different types of graphs, as you can imagine. One which you'll want to be familiar with is a directed graph, which has the notion of direction applied to its edges:

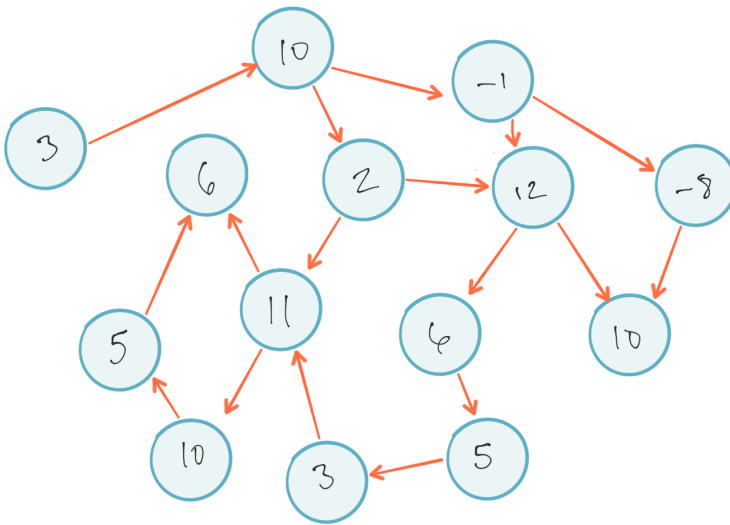


These types of graphs are useful for describing traffic flow of some kind, or any system in which movement is not bi-directional in every case. There is also an undirected graph which you can think of as a series of two-lane highways that connect towns in a coun-

tryside. Travel is bidirectional between each town and not directed along a given path.

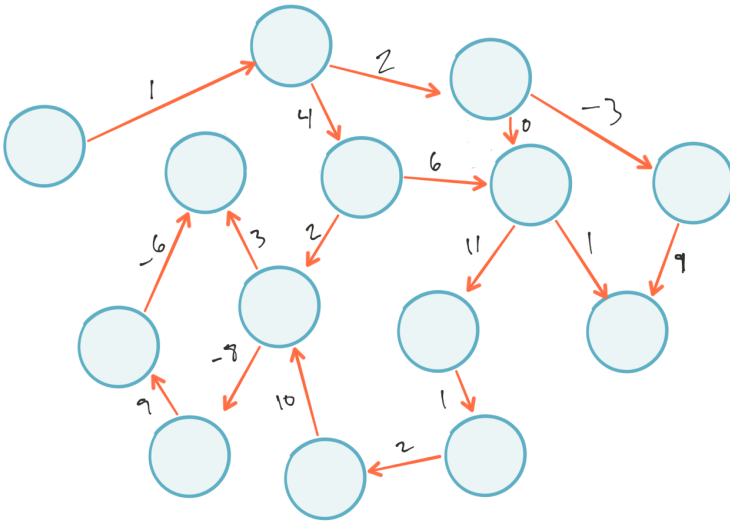
## Weighted and Unweighted Graphs

Values can be applied to various aspects of a graph. Each vertex, for instance, might have a weight applied to it that you'll want to use in a calculation of some kind:



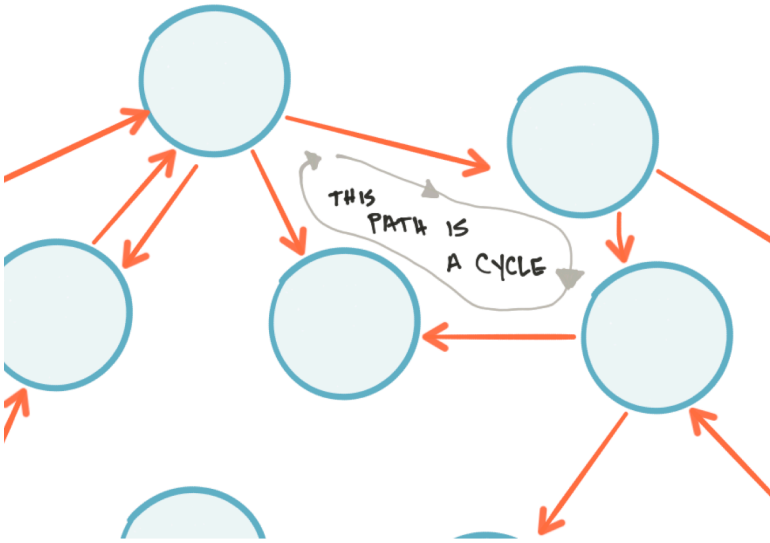
Maybe you're going on a trip, trying to figure out the most efficient way to see the cities you like the most.

There is also an edge-weighted graph, which is useful for calculating optimal paths:



## Cycles

When you have vertices connected circularly, it's called a cycle. We can see one of these as part of our first graph above:



Cycles are common in directed graphs. If a graph doesn't have a cycle, however, it has a special name.

## **Directed Acyclic Graphs (DAGs)**

If we redraw the graph above with edges that don't cause any cycles, we'll have a directed acyclic graph, or DAG:



# SIMPLE ALGORITHMS

**Y**ou don't need to know how to write a sorting or searching algorithm from scratch, frameworks do that for us. You do, however, need to know *how they work* because 1) it's likely you will be asked some details about them during interviews and 2) understanding their complexity could be the difference between keeping and losing your job!

## THE CODE

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video walkthroughs of the algorithms you see in this chapter and others from here. I'll be using screenshots once again for the code samples for formatting reasons – if you want to play along please do... but you'll need the code from GitHub.

# BUBBLE SORT

Let's start with the simplest sorting algorithm there is: bubble sort. The name comes from the idea that you're "bubbling up" the largest values using multiple passes through the set.

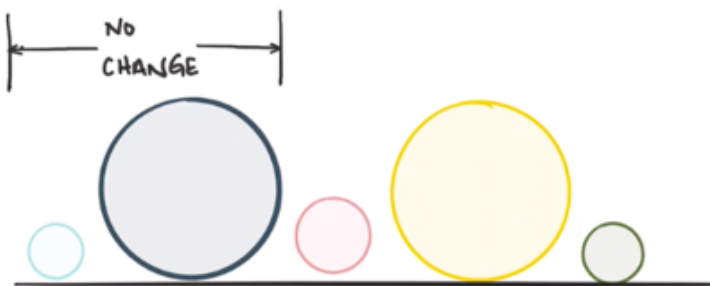
So, let's start with a set of marbles that we need to sort in ascending order of size:



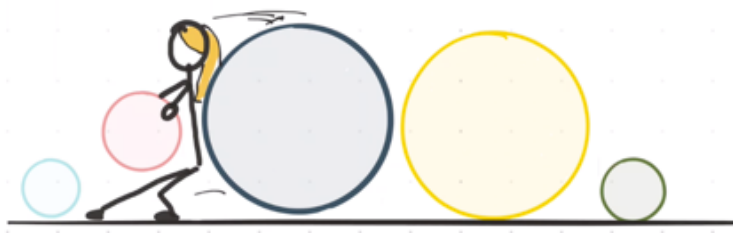
A reasonable, computational approach to sorting these marbles is to start on the left side and compare the first two marbles we see, moving the larger to the right.

As you can see, the smaller marble is already on the left so there's no change needed:

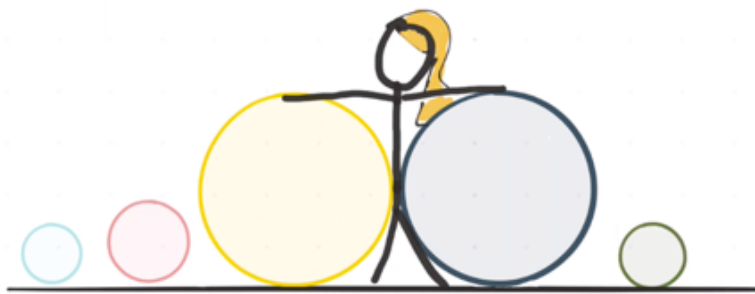




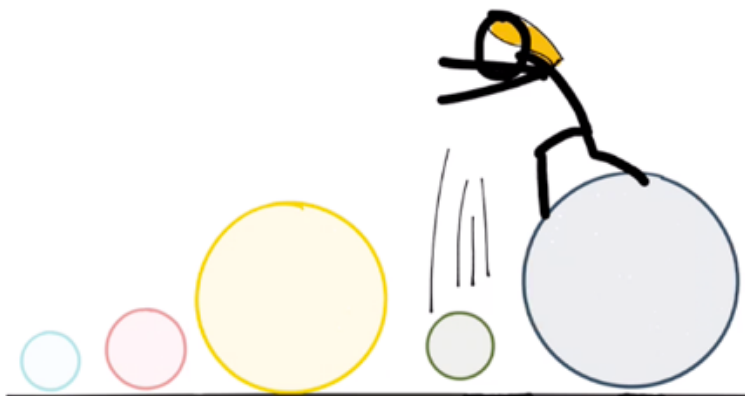
Then we move on to the next two, which are the dark blue and the pink, switching their positions because pink is smaller than dark blue:



The same goes for yellow and dark blue, although an argument could be made that the author's drawing skills don't make it clear that dark blue is slightly larger.



The last two are simple: the green marble is much smaller than the dark blue, so they switch positions as well.



OK, we're at the end of our first pass, but the marbles aren't sorted yet. The green is out of place still.

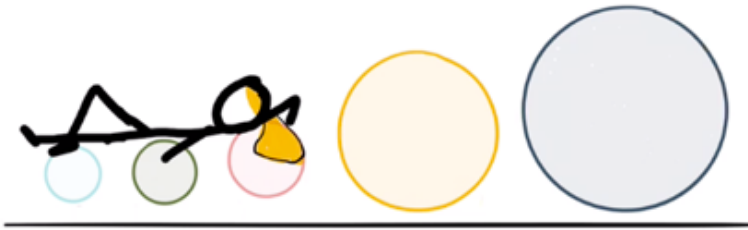


We can fix this by doing another sorting pass. This one will go a bit faster because blue and red are in order, red and yellow are in order, but green and yellow are not – so we make that switch:



Not too hard to figure it out from here. The green ball needs to move 1 more time to the left, which means one more pass to sort the marbles – making 3 passes in total.

Eventually, we get there:



Bubble sorts are not efficient, as you can see.

### JavaScript Implementation

Implementing bubble sort in code can be done with a loop inside a recursive routine. That sentence right there should raise the hairs on the back of your neck! Indeed, bubble sort is not very efficient (as we'll see in a minute).

Here's one way to implement it:

```

//the list we need to sort
const list = [23,4,42,15,16,8];

const bubbleSort = (list) => {
  //a flag to tell us if we need to sort this list again
  var doItAgain = false;
  const limit = list.length;
  const defaultNextVal = Number.POSITIVE_INFINITY;
  //loop over the entries
  for (var i = 0; i < limit; i++) {
    let thisValue = list[i];
    let nextValue = i + 1 < limit ? list[i+1] : defaultNextVal;
    //compare values
    if(nextValue < thisValue){
      list[i] = nextValue;
      list[i+1] = thisValue;
      //since we made a switch we'll set a flag
      //as we'll need to execute the loop again
      doItAgain = true;
    }
  }
  if(doItAgain) bubbleSort(list);
}
bubbleSort(list);
console.log(list);

```

Executing this code with Node we should see this a sorted list: **[ 4, 8, 15, 16, 23, 42 ]**

## Complexity Analysis

As you can see, we're using recursion as well as a for loop, which should set off some alarms. If you recall from the chapter on Big-O, nested loops operating on the same list almost always means

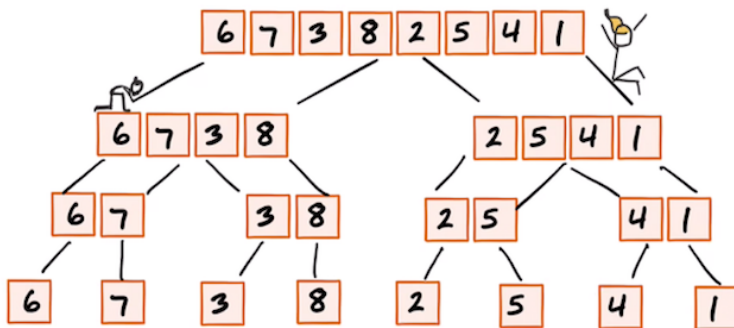
$O(n^2)$  and this algorithm is no exception to that, even if we're using recursion.

The use of recursion also means we're potentially taking up  $O(n)$  space as well and opens the possibility of a stack overflow exception given enough items to sort.

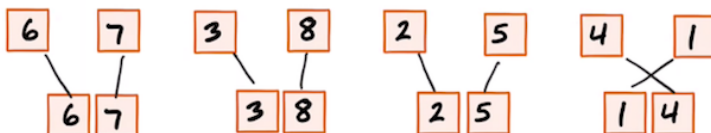
## MERGE SORT

Merge Sort is one of the most efficient ways you can sort a list of things and, typically, will perform better than most other sorting algorithms. In terms of complexity, we're using a divide and conquer approach, which should tip you off that this is going to be at least  $O(\log n)$ . Once we divide the array, we need to sort the items which is going to be an  $O(n)$  operation since we need to address each item. That means this algorithm's complexity is  $O(n \log n)$ .

Merge Sort works by splitting all the elements in a list down to smaller, two-element lists which can then be sorted easily in one pass. The final step is to recursively merge these smaller lists back into a larger list, ordering as you go - this is the  $O(\log n)$  part:

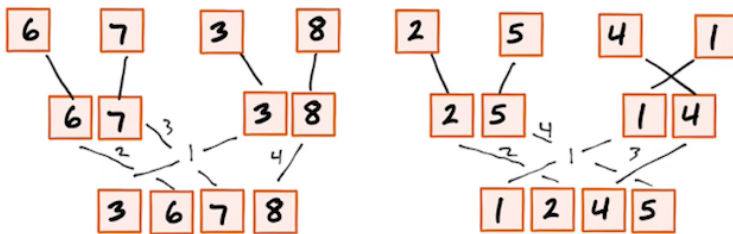


Now we need to merge the lists. The rules for this are simple: compare the first elements of adjacent lists, the lowest one starts the merged list – this is the  $O(n)$  part:



This is straightforward with lists of one element being combined into lists of two elements. But how do we match up lists of two elements?

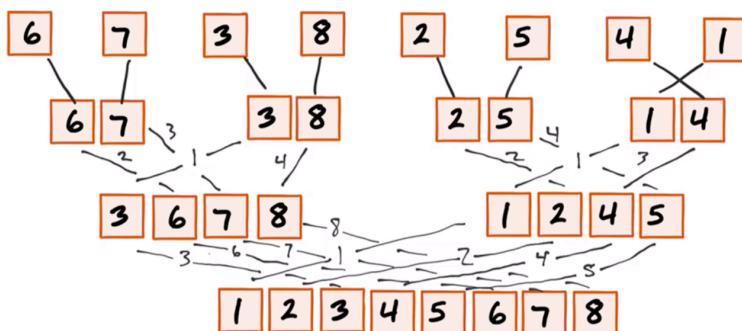
The same way. When combining the **[6,7]** list with the **[3,8]** list, we compare the 3 with the 6 – the 3 is the smallest so it goes first. Then we compare the 6 with the 8 and the 6 is smaller, so it goes next. Finally, we compare 7 and 8 and add them accordingly:



Now, you might be thinking “wait a minute – how do we know that a smaller number isn’t sitting to the right of the 6? Wouldn’t that mess up the sort?” That’s a good question.

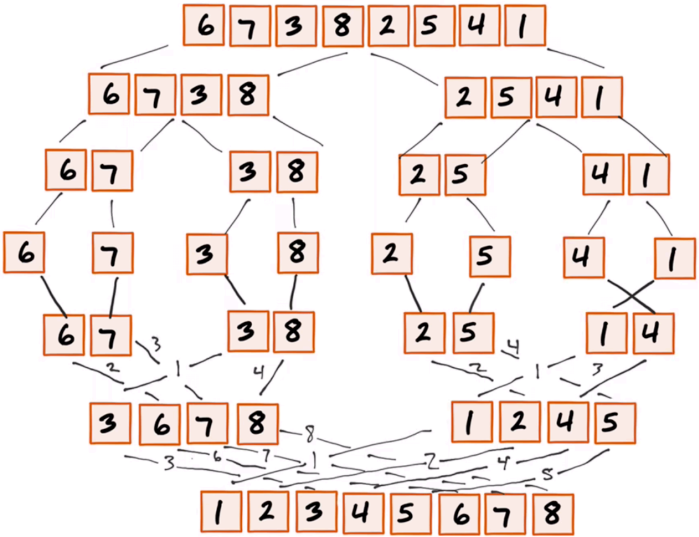
It’s not possible to have a lower number to the right of any element in a merged list – when the [6,7] list was created we sorted it. This is the power of Merge Sort: the leftmost numbers are always smaller which gives us a lot of power.

OK, so now we continue in the same way, merging the final lists of 4. We start on the left-hand side of each list, comparing the values, and adding the lowest to the merged list first:





And we're done! Here's the full operation, in case you'd like to see it top to bottom:



## JavaScript Implementation

Implementing merge sort in code is a bit tricky. You need to have two dedicated routines, one for splitting the list and one for merging.

The first step is to recursively split the list:

```

console.log(mergeSort(list));

const list = [23,4,42,15,16,8,3];
const mergeSort = (list) => {
  //if there's only one item in the list
  //return. This is our recursion check.
  if(list.length <= 1) return list;

  //cut list in half
  const middle = list.length/2;
  const left = list.slice(0,middle);
  const right = list.slice(middle,list.length);

  //recursively run through the splits
  //left and right will be separated down to single elements
  return merge(mergeSort(left), mergeSort(right));
};

```

In this routine we're just splitting whatever list comes in right down the middle. If the list only has one entry, we're returning. This prevents the recursive call on the last line from blowing up.

Next is our **merge** function:

```

const merge = (left, right) => {
  var result = [];
  //if the left and right lists both have elements
  //run a comparison
  while(left.length || right.length){
    //if there are items in both sides...
    if(left.length && right.length){
      //if the first item on left is
      //less than right...
      if(left[0] < right[0]){
        //take the first item on the left
        result.push(left.shift());
      }else{
        //take the first item on the right
        result.push(right.shift());
      }
    }else if(left.length){
      //just take left
      result.push(left.shift());
    }else{
      //just take right
      result.push(right.shift());
    }
  }
  return result;
}

console.log(mergeSort(list));

```

This routine takes two lists and compares their leftmost values. If one of the lists is empty then the left-most value from the other list is appended as the result.

Running this we get should see **[ 3, 4, 8, 15, 16, 23, 42 ]**

# QUICKSORT

Quicksort is a divide and conquer algorithm that uses a pivoting technique to break the main list into smaller lists. These smaller lists use the pivoting technique until they are sorted. The complexity of quicksort, however, is not constant because the pivot (as you're about to see) is determined at random and the partitioning of the list can put the algorithm at a disadvantage.

In the worst case, quicksort is  $O(n^2)$  when the pivot is the smallest or largest element in the list. In the best case it's  $O(n \log n)$ , like merge sort. We'll discuss this a bit more at the end of this section; for now let's get to know this algorithm.

There are two ways to implement quick sort. Let's go over how the algorithm works and then I'll discuss the different implementations.

We'll start with a set of 8 elements (sorry, no cats or marbles this time).

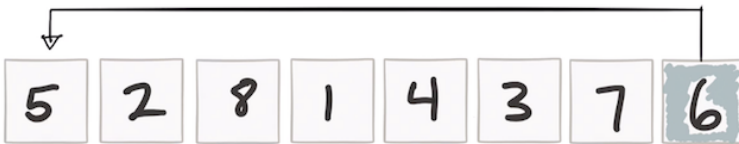


We need a pivot, so we'll choose the very last element of the list (by convention).

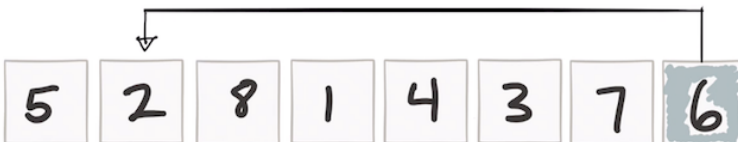


**PIVOT**

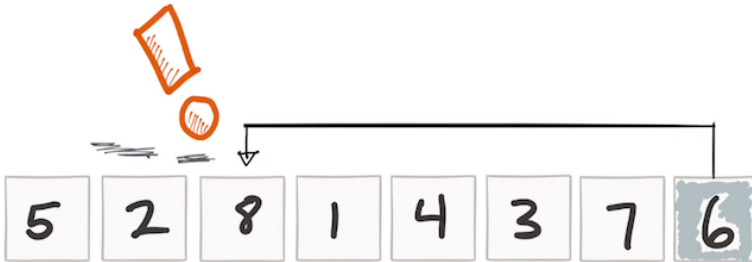
The next step is to partition our list so that all elements in the list that are less than our pivot are in a separate partition to the left, and all the elements greater are in a partition to the right. There are various ways to do this, but the simplest is to start at the beginning of the list – in this case a 5 – and if it's smaller we'll leave it in place.



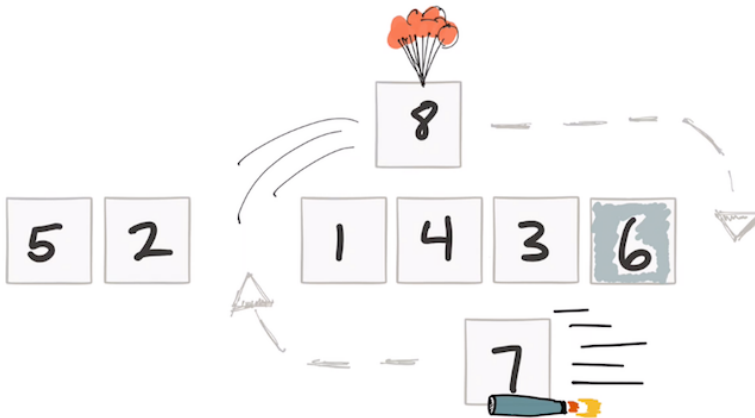
The next element is a 2, so we'll leave that there as well.



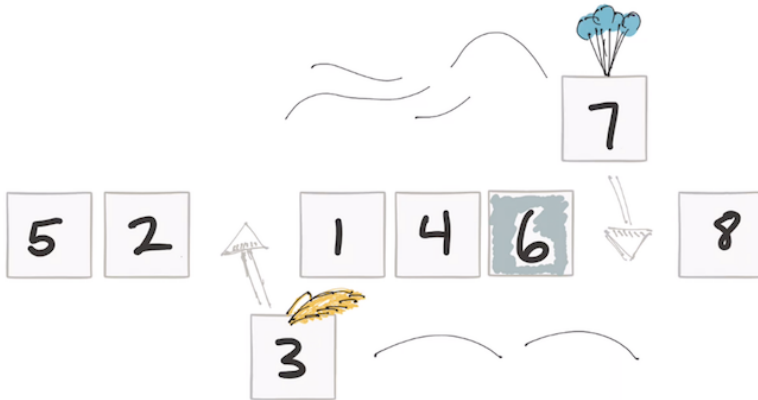
Then we come to an 8, which is greater than our pivot, which means we need to move it.



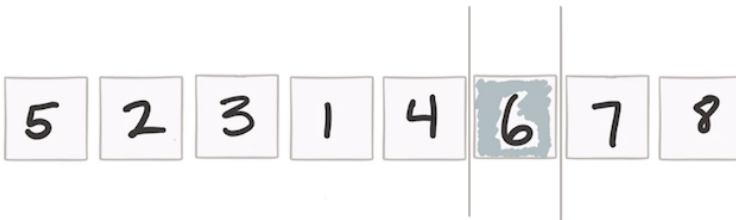
I'll pick it up and move it behind the pivot as it's a larger number, and I'll move the pivot down one position. There's already a number there – a 7 – so I'll move that to where the 8 was.



Now the 7 is the next thing to evaluate. It's greater than 6 so I'll do the same maneuver, switching the 3 and the 7, putting the 7 in the position my pivot was just in.



The next elements are 1 and 4 – which are both smaller than 6. This means we’re done with our first partitions!



We now have two new lists – the numbers less than 6 on the left and the numbers greater than 6, on the right. Six itself in its final position, so we’ll ignore it for now.

Now we pick a pivot for the new lists in exactly the same way – the last element in each list. This means that 8 is the pivot on the right, 4 is the pivot on the left.

The neat thing here is that there are no numbers greater than 8 in our list on the right, so there's nothing we need to do.



The list on the left, however, can be separated in exactly the same way we did before. We'll compare once again from the first position – it's a 5 so we'll stick it behind our 4, and move the number to the left of the 4 (a 1) to 5's old position.

And just like that – we're done!

## Avoiding a Mess

If our list was presorted for whatever reason (and it turns out that yes, this does happen) then our sorting routine here would take a very, very long time. We'd have to split and order the list for every element, turning the complexity to  $O(n^2)$ .

To get around this we can select our pivot intelligently. This requires an initial step where you find the median value of a list and then make that the pivot. From there the sorting operation will usually beat out merge sort because the sorting happens *in-place*,



```

//these are the partition lists we'll need to use
var left=[], right=[];
//default the pivot to the last item in the list
const pivot = list.length -1;
//set the pivot value
const pivotValue = list[pivot];
//remove the pivot from the list as we don't want to compare it
list = list.slice(0,pivot).concat(list.slice(pivot + 1));
//loop the list, comparing the partition values
for (var item of list) {
    item < pivotValue ? left.push(item) : right.push(item);
}
//recursively move through left/right lists
return quickSort(left).concat([pivotValue], quickSort(right));
};

console.log(quickSort(list));

```

over-

parti-

pivot

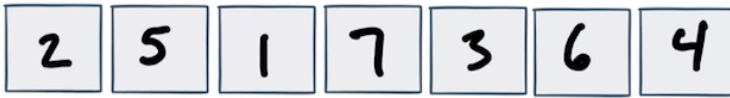
## SELECTION SORT

Selection sort is likely the simplest possible way to sort a list. It's how you and I might think of telling a duck to do it if a duck had hands and needed to get a job as a coder at Amazon.

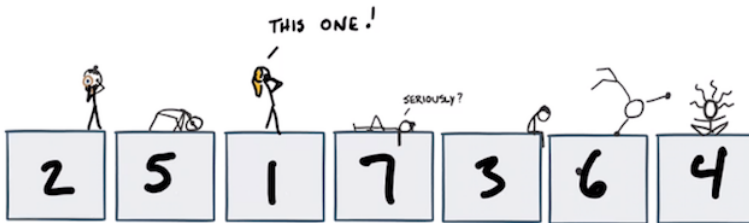
This algorithm works by scanning a list of items for the smallest element, and then swapping that element for the one in first position. This continues with the remaining items until the list is sorted.

The complexity of selection sort ranges from  $O(1)$  when the list is already sorted to  $O(n^2)$  when the list is presorted in the reverse order you want. The  $O(1)$  complexity makes this an interesting choice if there's a chance of sorting a pre-sorted list (or a list of equal objects) – which happens fairly often.

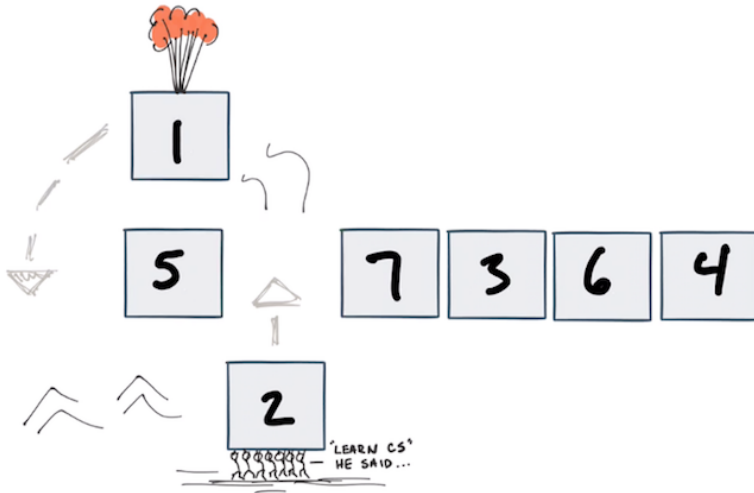
We'll start out with an unsorted list of 7 elements:



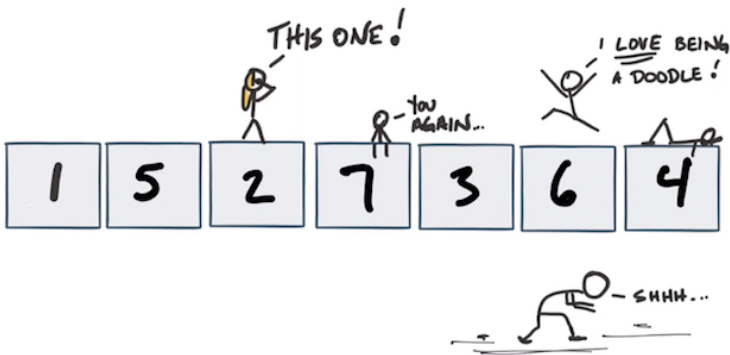
The first task is to find the lowest number, which (in the worst-case scenario) is a linear scan:



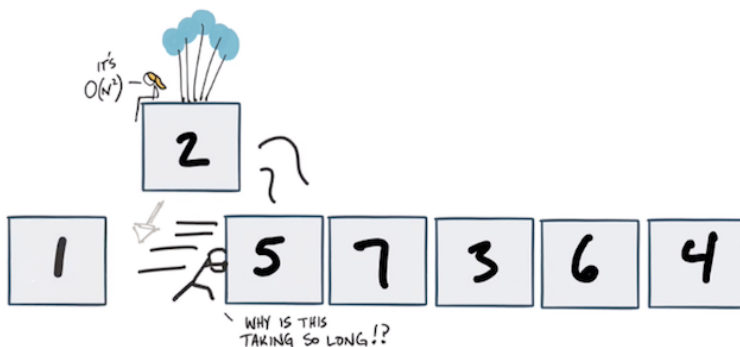
Once we've found the lowest element, we swap it with the first element in the list, as we know this is where the lowest element belongs.



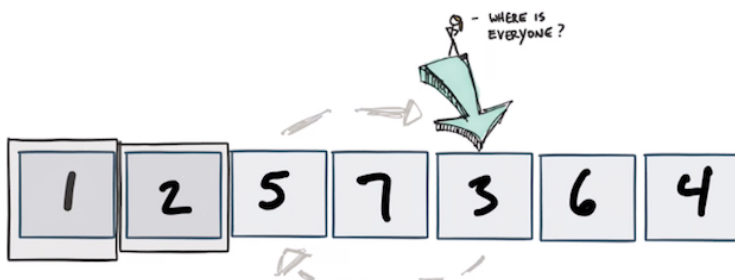
Then we do it all over with the remaining items. In this case the next lowest element is a 2.



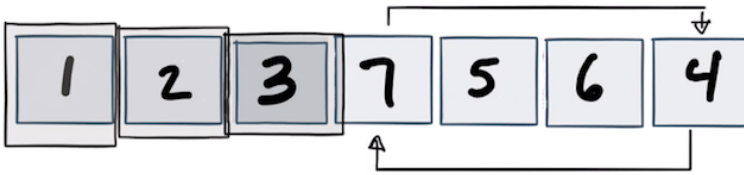
So, we swap the two with the element that was in second position.



Rinse, repeat. We keep scanning the list for the lowest item and swapping it for the items that aren't in their final position (shaded):



Not a very efficient operation. My doodles have rebelled, apparently – but our last operation is swapping the 4 and the 7 and then we're done!



## Implementation in JavaScript

This one is straightforward – no need for recursion. We can use two loops: the first will loop over our list, the second will loop forward for every step:

```

const list = [23,4,42,8,16,15];
const selectionSort = (list) => {
  for (var i = 0; i < list.length; i++) {
    //default the min value to the first item in the list
    //all we need do is track the index for now
    var currentMinIndex = i;
    //loop over the list, skipping the currentMinIndex
    for(var x = currentMinIndex + 1; x < list.length; x++){
      //if the current list item is less than the current min value...
      if(list[x] < list[currentMinIndex]){
        //reset the index
        currentMinIndex = x;
      }
    }
    //has the index changed?
    if(currentMinIndex !== i){
      //if yes, switch the values in the list
      var oldMinValue = list[i];
      list[i] = list[currentMinIndex];
      list[currentMinIndex] = oldMinValue;
    }
  }
  return list;
};
console.log(selectionSort(list));

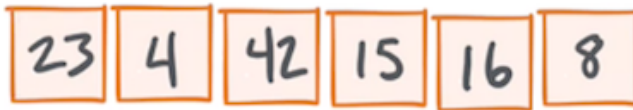
```

## HEAP SORT

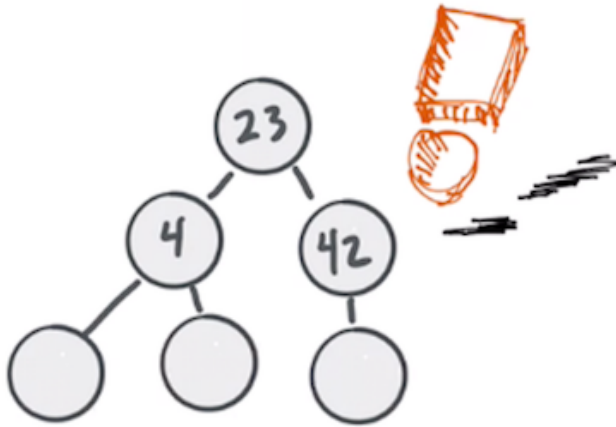
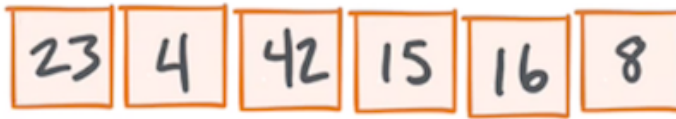
Heap sort is a bit like selection sort in that it moves unsorted data to a sorted “partition” selectively. The difference, however, is that it uses a heap to do so. If you recall, a heap is a tree structure where parent nodes in one level are either greater than (max heap) or less than (min heap) child nodes in descendent levels.

Heap sort is  $O(n \log n)$ , however it has an advantage over quicksort of being  $O(n \log n)$  in the worst-case scenario, whereas quicksort in the worst case is in  $O(n^2)$ .

We have an unsorted list of numbers, as always.

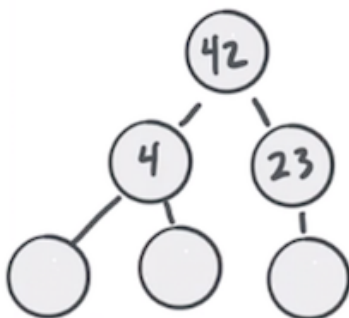
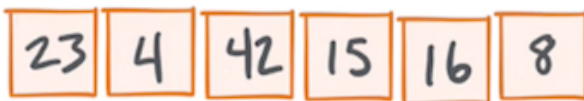


The first step is to move these numbers into a heap:

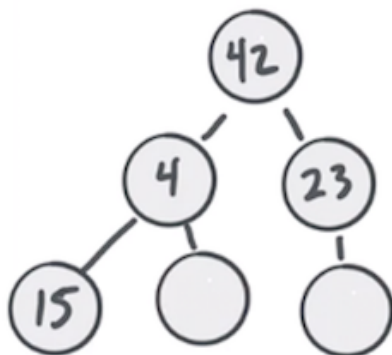
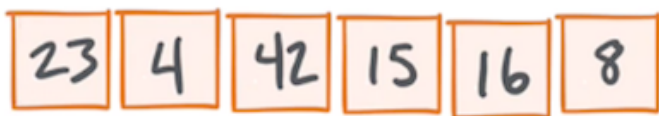


As we go along, however, we realize that we would violate the heap rules if 42 was to be placed before 23, so we swap them.

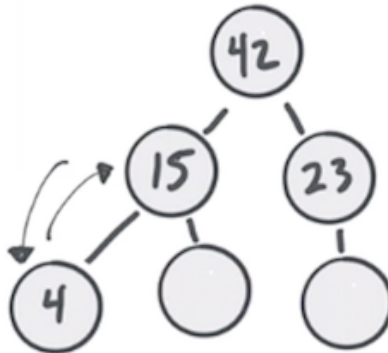
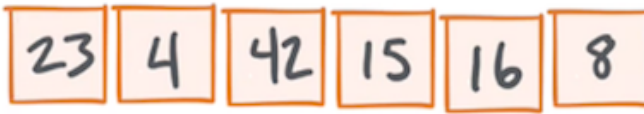




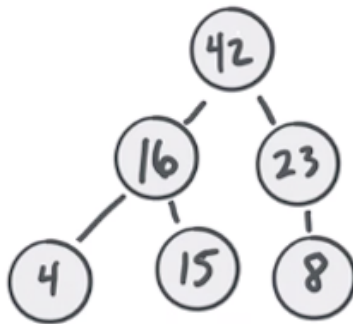
Now we have a valid heap.



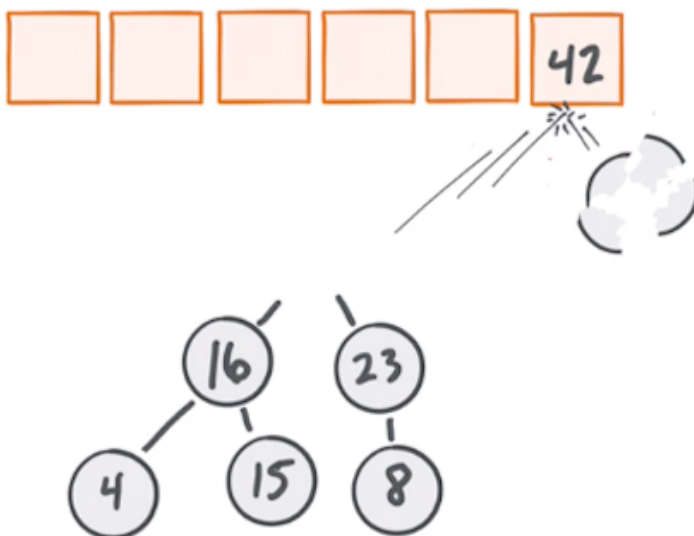
In the same way, 15 and 4 would be in violation – so we swap them.



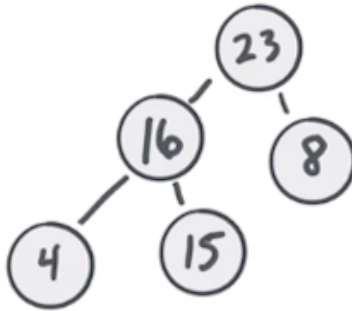
We end by adding 16 and 8, swapping positions for the 15 and 16 so we avoid violations. We now have a completed heap from our original list.



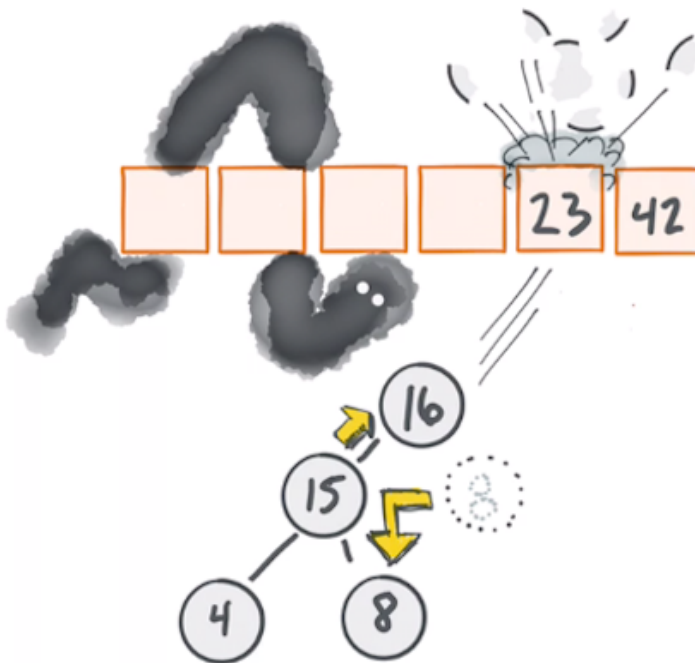
At this point we move on to the final stage, which is systematically removing the root of the heap, putting it back into the list.



As we add items back to our list, the heap is adjusted to make sure it remains valid. Here, we need to move 23 to the root as it's larger than its sibling, 16. 8 tags along for the ride as it's still in a valid position.



Next, we add 23 back to our list, and then elevate 16 to the root, as it's larger than 8. However, this means that 8 doesn't have a parent, so we move it over and place it under the 15.



We keep going in this way until all the nodes in our heap are gone.



## BINARY SEARCH

Binary search is a fascinating thing. At first glance it seems rather ridiculous – the list you’re searching over has to be sorted first! This isn’t so nuts if you store your data in a binary tree (a BTREE) which we’ve discussed already.

Again, this algorithm is *divide and conquer* which should give a clue – I hope it does because we’ve already had a look at it in the Big-O chapter!



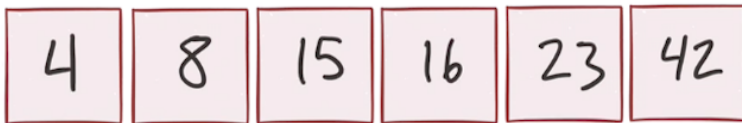
The deal is this: you split a list of sorted items and decide, from there, whether the item you're looking for is in the left or right list. You can decide this accurately because the list is sorted.

You then split that list and check to see if values on the right or left of the split are greater, lesser, or equal to the value you're searching for.

Keep going until you find what you want.

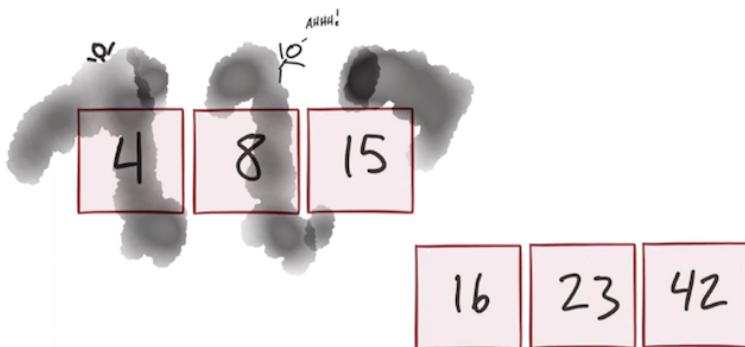
Let's take a look.

We have an ordered set to play with. How it became ordered is a mystery – as are the numbers – which we will figure out later in the book.

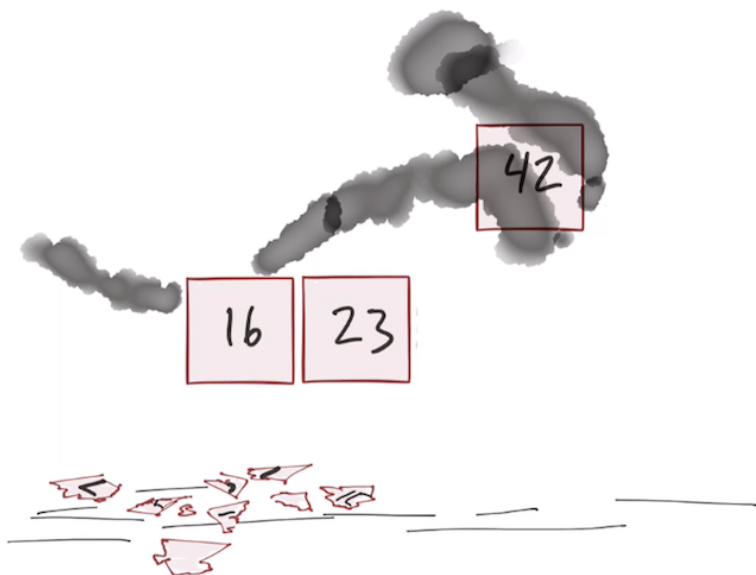


What we need to do is to find the number 23. Now, you and I both have eyes, but a computer doesn't – so we need to give it an intelligent way to find the number 23.

To do this, we'll split the list in two, right down the middle, and remove the entire half that is less than, or greater than, our number. In this case it's the side with 15 and below.



Since our list is sorted, we know the number we're looking for is in the remaining set. So, let's do the same thing. We'll split that list down the middle (or close to it).



The 42 is on the right – which means it and everything after it can be removed.



Now we're left with only two elements, which again we split down the middle. Evaluating each side, we see that the number we're looking for is on the right! We can remove the 16 and we're done!



## JavaScript Implementation

Looking through a list by splitting it in half continually is straightforward. In this routine we'll search for a value and return its index:

```
const list = [4,8,15,16,23,42];

const binarySearch = (list, lookFor) => {
  //define the range
  var min=0, max=list.length;
  var middle;
  //while there is something to search for...
  while(min <= max){
    //define the middle of the range
    middle = Math.floor((min + max) / 2);
    //if we've landed on it...
    if(list[middle] === lookFor){
      return middle;
    }else{
      //if we haven't landed on it, where is it?
      //if the middle is less than the value we're
      //looking for, reset the min
      //otherwise reset the max
      list[middle] < lookFor ? min=middle : max=middle;
    }
  }
  return -1;
};

console.log(binarySearch(list,3));
```

# GRAPH TRAVERSAL

We've learned how to split lists apart using binary search, but how do we search over something a bit more complex, such as a graph? As you've noticed, many of the data structures we've been working with are based on graphs, so traversing them properly is quite important.

Before we go on, it's important to understand that you can carry out these operations in one of two ways: using *recursion* or using *iteration*. One is clever and one is costly – thankfully I'm referring to the same method: *recursion*.

Every time a recursive function is executed, the variables that the function uses remain in scope and are stored on the stack. I know I've brought this up before, but it's worth repeating: *recursion is interesting and clever, but can also be costly*.

I point this out because *you will be asked something like this in an interview someday*. Recursion and *space complexity* are not friends – so come prepared to understand why. Let's go a bit deeper.

## Using Recursion

We've discussed binary trees in a previous chapter:

```

class BinaryTreeNode{
  constructor(val){
    this.value = val;
    this.right = null;
    this.left = null;
  }
  isLeaf(){
    return this.left === null && this.right === null;
  }
  traverse(node){
    if(node.left) return this.traverse(node.left);
    if(node.right) return this.traverse(node.right);
    //otherwise we're a leaf
    return null;
  }
}

```

A node instance has a value as well as two possible descendants, **left** and **right**. The **traverse** method allows you to start from any node in the tree and then traverse through each node, doing something exciting and amazing.

There are several ways to run calculations with this class. We can hard-code what we need done in the **traverse** method (searching for a value, for instance) or we can pass along some kind of call-back. For the sake of keeping things at a pace for this book, I'm going to sidestep implementation details here and simply wave my arms and say "don't worry about this for now".

The reason is that this code, while clever, is also a bit inefficient. I already discussed the stack overflow problems above, but what if you wanted to control what *type* of search you wanted to perform? In other words, what if you wanted to *go deep* into the tree as you knew the thing you're looking for was likely at the lower levels of the tree?

Conversely: what if you wanted to go *wide* instead? Recursion can be tweaked to address these things, but then you're altering your class to accommodate what should really be the responsibility of an entirely different bit of code altogether.

## Using Iteration

Instead of filling up our stack, what if we used a simple loop? Let's strip out the **traverse** method and go with a more cohesive **BinaryNode** class:

```

class BinaryNode {
  constructor(val){
    this.value = val;
    this.right = null;
    this.left = null;
  }
  isLeaf(){
    return this.left === null && this.right === null;
  }
}

//Now let's fill out our nodes:
const rootNode = new BinaryNode(0);
rootNode.left = new BinaryNode(1);
rootNode.right = new BinaryNode(2);
rootNode.left.left = new BinaryNode(3);
rootNode.left.right = new BinaryNode(4);
rootNode.right.left = new BinaryNode(5);
rootNode.right.right = new BinaryNode(6);

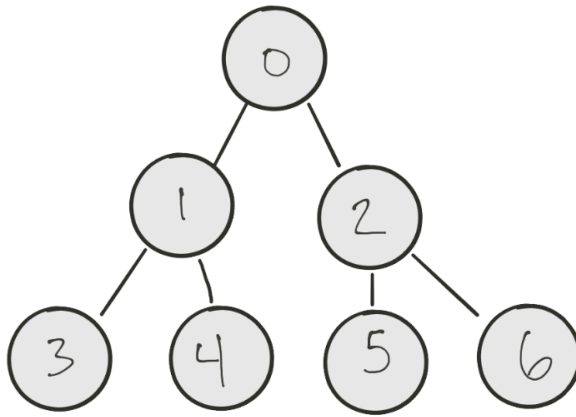
```

This will give us a two-level tree, which will be good enough to traverse. But how? It turns out there are two ways: *Breadth-first* and *Depth-first*. These types of traversals are typically referred to as “breadth-first search” and “depth-first-search”.

## Depth-first Search

Let’s start by visualizing our graph:

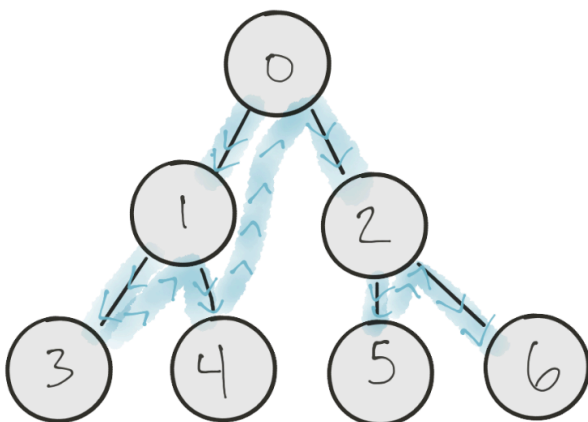




With a depth-first search, we want to go as deep as possible into the graph from either the left or the right. Traditionally it's done from the left.

The idea is to go as deep as you can and then backtrack your way back up and then over to the right until the traversal of the tree is done.

With my crude drawing skills, it might look something like this:



We start at the root, proceed to 1 and then 3. Then backtrack to 4, back up and over to 2, then 5 then 6.

To implement this we need a data structure that will keep track of the very next descendant as well as the current sibling. We will always follow the next descendant, and if it doesn't exist, we'll go with the last saved sibling. If both of those don't exist we're done with our tree.

For this, we can use a stack. Here's some basic code to see how DFS works. Note that I'm wrapping this in a class using a static method for **traverse**:

```

class DFS {
  static traverse(root){
    var stack = []; //arrays in JS act like stacks
    let thisNode = null;
    //push the root onto the stack
    stack.push(root);
    while(stack.length > 0){
      thisNode = stack.pop();
      console.log(thisNode.value);
      //push right then left
      if(thisNode.right !== null){
        stack.push(thisNode.right);
      }
      if(thisNode.left !== null){
        stack.push(thisNode.left);
      }
    }
  }
}

```

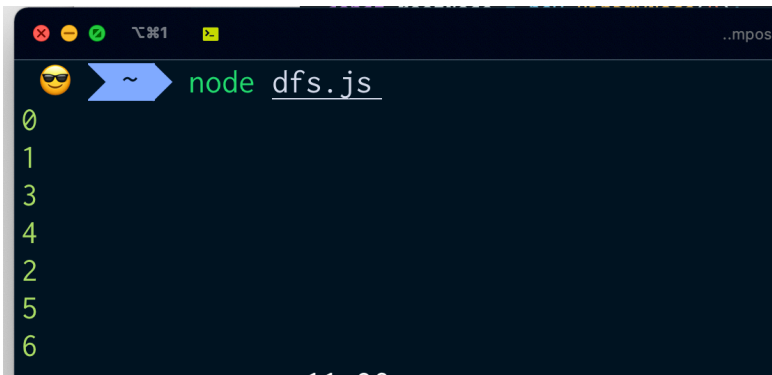
I'm cheating a small bit here by using an array in JavaScript as my stack, but it will work just fine because JavaScript arrays implement the **push/pop** methods in the same way a traditional stack does.

All we need to do is to keep track of how many nodes there are left to traverse. We know that by using a stack and pushing values onto it if they exist down below.

Let's load it up and run it!

```
const rootNode = new BinaryNode(0);
rootNode.left = new BinaryNode(1);
rootNode.right = new BinaryNode(2);
rootNode.left.left = new BinaryNode(3);
rootNode.left.right = new BinaryNode(4);
rootNode.right.left = new BinaryNode(5);
rootNode.right.right = new BinaryNode(6);
DFS.traverse(rootNode);
```

The result, when we execute this, is right on the money:



```
node dfs.js
0
1
3
4
2
5
6
```

“Going deep” on a graph is useful if you’re interested in parent/child relationships.

You could also look through a binary tree and see if it is in fact a *binary search* tree, with all the node values set as required. We know from reading the last chapter that a binary search tree requires that every child node to the right of a given node must have a greater value; every child node to the left must have a lesser one.

Is depth-first the best choice for this? You might check a lot more nodes than you need to! Let's have a look at an alternative: *breadth-first*.

## **Breadth-first**

With breadth-first search we need to track the nodes of a tree in each level before traversing to the next level. This means we need to track every node and its children *in order*.

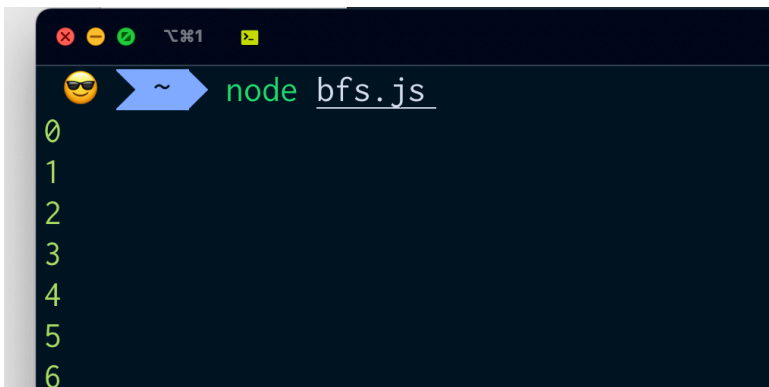
For this we need to use a queue, which in JavaScript once again means using an Array. Unfortunately, there are no **enqueue** and **dequeue** methods, but we do have their equivalent: **push** and **shift**.

```

class BFS {
  static traverse(root){
    const queue = []; //arrays in JS also act like queueus!
    let thisNode = null;
    //push the root onto the queue
    queue.push(root); //equivalent to "enqueue"
    while(queue.length > 0){
      //remove the next item from the queue
      thisNode = queue.shift(); //equivalent to dequeue
      //do whatever calc
      console.log(thisNode.value);
      //add the children into the queue
      if(thisNode.left != null){
        queue.push(thisNode.left);
      }
      if(thisNode.right != null){
        queue.push(thisNode.right);
      }
    }
  }
}

```

If we load it up just like we did in the last section and run it, we will have a completely different result:



We're traversing the graph one level at a time, which makes great sense if we're trying to evaluate all the parent nodes *first*, which we would be doing if we're validating a binary search tree.

# ADVANCED ALGORITHMS

I wasn't sure about using the terms "advanced" and "simple" for these chapters on algorithms. I don't want you to think these are harder to implement or to figure out – they're not (necessarily).

The reason I chose these terms is that it's more likely you'll find yourself getting paid to implement a shortest path algorithm or some type of sieve at some point in your career. It's *unlikely* that you'll be hired to implement bubble sort.

I also wanted to provide a framework for you to create your algorithms for solving the more interesting problems you work on every day. This, to me, is the most important part of this chapter, so we'll start there.

## THE CODE

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video walkthroughs



of the algorithms you see in this chapter and others from here. I'll be using screenshots once again for the code samples for formatting reasons – if you want to play along please do... but you'll need the code from GitHub.

## DYNAMIC PROGRAMMING

No, this section is not about Ruby, Python, JavaScript, etc. Dynamic programming is a way to solve a problem using an algorithm in a prescribed way. It sounds complicated, but it's anything but.

Dynamic programming gives us a way to elegantly create algorithms for various problems and can greatly improve the way you solve problems in your daily work. It can also help you ace an interview.

### **Definition**

Let's start with a quick definition so we know what dynamic programming is and how it works. At its core, dynamic programming is simply solving an optimization problem by guessing systematically. It's almost laughable to think about dynamic programming in terms of this definition, but as you'll see it turns out to be rather powerful.

To use dynamic programming, the problem you're solving must be:

- An **optimization problem**. We saw one of these in Chapter 1 (the Bin Packing Problem) when I tried to optimize storage for my daughter's things.
- **Dividable into subproblems**. With dynamic programming you can recurse over and solve to solve the larger, objective problem.
- Have an **optimal substructure**. That's a mouthful, but what it means is that the subproblems you solve must be complete unto themselves. In other words, if you solve subproblems x, y and z to solve objective problem A, then the solutions to x, y and z should be sufficient on their own to solve A. You don't need to use x plus some other algorithm.
- **Reducible to P time** through memoization. Some problems you can solve with dynamic programming are solvable in exponential time (like Fibonacci), however this can be reduced to P time through memoization. Another fun word, but you can think of this basically as "caching" the answers to the subproblems and then applying.

I wouldn't blame you if you're underwhelmed at this point. The name "dynamic programming" seems bland, and the underlying techniques more than a little vague. You'll understand it well in a few sections as we solve some problems with it, I promise.

Before we get there, it's important to understand where the name came from and why dynamic programming even exists.

## Origins

This is a funny story (emphasis mine):

*An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a*

*pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.*

If you want to read Richard Bellman's original paper on dynamic programming, *you can do so here*.

There you have it: **the name means nothing**. The dynamic programming design process, however, is behind some of the most powerful algorithms we know of. We'll see those in the next section.

The best way to see its power, however, is to just do it. So let's! We'll use dynamic programming to help us get through a job interview.

## **An Example: Fibonacci**

Again with Fibonacci! Ah well it *is* a programming book you know. I'm using it here because it's the simplest way to convey the dynamic programming process. Also: you will be asked how to solve Fibonacci at some point in your career, and you're about to get three different approaches!

Which leads right to a great opening point: *our jobs are about solving problems*. When you go to these interviews, they mostly want to see how you would go about solving something complex. As it turns out, the *Interviewing For Dummies* book says that Fibonacci is a great question for just that case.

## Definition

Let's start with a definition, just in case you don't know or remember what a Fibonacci Sequence is:

*A series of numbers in which each number (Fibonacci number) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc.*

Super. Why do we care about these numbers? These numbers (and the algorithm we're about to discuss) underpin nature's symmetry:

*The Fibonacci numbers are Nature's numbering system. They appear everywhere in Nature, from the leaf arrangement in plants, to the pattern of the florets of a flower, the bracts of a pinecone, or the scales of a pineapple. The Fibonacci numbers are therefore applicable to the growth of every living thing, including a single cell, a grain of wheat, a hive of bees, and even all of mankind.*

If you divide each successive number by itself (so:  $5/3$ ,  $8/5$ ...) you converge on a fascinating number called phi:

*What makes a single number so interesting that ancient Greeks, Renaissance artists, a 17th century astronomer and a 21st century novelist all would write about it? It's a number that goes by many names. This "golden" number, 1.61803399, represented by the Greek letter Phi, is known as the Golden Ratio, Golden Number, Golden Proportion,*

*Golden Mean, Golden Section, Divine Proportion and Divine Section. It was written about by Euclid in "Elements" around 300 B.C., by Luca Pacioli, a contemporary of Leonardo da Vinci, in "De Divina Proportione" in 1509, by Johannes Kepler around 1600 and by Dan Brown in 2003 in his best-selling novel, "The Da Vinci Code."*

Absolutely fascinating stuff. Our interviewer, however, is waiting patiently for us to come up with an algorithm for calculating a Fibonacci Sequence to the  $n$ th position – so let's get to it!

## **The Painful Way**

The interviewer has asked us a standard question:

*How would you derive a Fibonacci sequence up to a given position?*

In other words, if we're given a value of 10, the interviewer will want to see the first 10 Fibonacci numbers. We can solve this (and more!) using dynamic programming.

The first step is to break the problem down into smaller problems (called subproblems) that we can solve. If we're trying to derive a Fibonacci sequence to the 10th position, we can do it with pen and paper like this:

- The first number in the Fibonacci sequence is 0
- The second number is 1

- The third number is  $0+1=1$
- The fourth number is  $1+1=2$

And so on. This would answer the interviewer's question (about the sequence) but it wouldn't show them what they're after: our ability to solve a problem programmatically. We can do this with the next step in dynamic programming: recursively solve the sub-problems until the objective problem is solved.

It's easiest if we see some code at this point. Here's my Fibonacci solver implemented in JavaScript:

```
//the slow way  
let fibCount=1;  
const calculateFibAt = (n) => {  
  fibCount = fibCount+1;  
  if(n < 2){  
    return n;  
  }else{  
    return calculateFibAt(n-2) + calculateFibAt(n-1);  
  }  
}
```

Running this (using Node):



A terminal window with a dark blue background. The title bar shows three colored circles (red, yellow, green) and the text "os/1". The terminal prompt is "~/.os/1" with a yellow cursor. The command "node fib.js" is entered. The output is a list of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. The numbers are displayed in a light green color.

```
..os/1 (zsh)
~/.os/1 master ± node fib.js
0
1
1
2
3
5
8
13
21
34
55
```

Great! By the way I tried four times to write this from memory and *completely failed*. You would think this little routine would be embedded in my mind but ... oh well. If it took you a few times to come up with it don't feel bad! Recursive programming takes some getting used to.

The code in this routine works, is straightforward, and is standard interview fare. We're feeling happy about ourselves at this point, when the interviewer says:

*Talk to me about the complexity of this routine in terms of time and also space...*

Uh-oh... time for some Big-O! Good news for us is the that we started this part of the book off with a discussion of Big-O and our brains are burning now, thinking "right... wasn't there something bad about using recursion?"

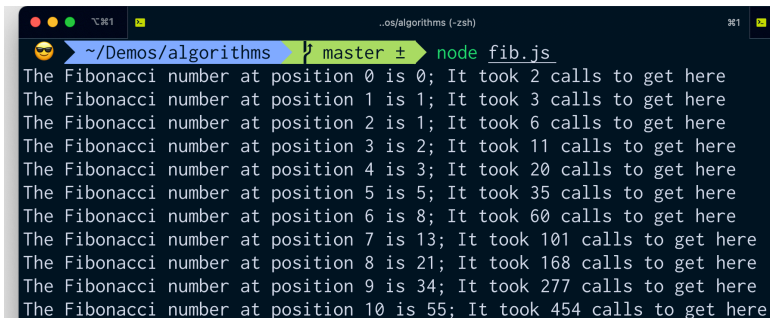


## Complexity Analysis of Our Painful Method

Let's start off by thinking in terms of *time complexity*. How long will it take to run our recursive algorithm? You can see why you might get fired by changing the loop value to 1000. In short: this routine scales horribly. To see this, let's add a counter to the function:

```
for(var i = 0; i<10; i++){  
  let fibbed = calculateFibAt(i);  
  console.log(`The Fibonacci number at position ${i} is ${fibbed}; It took ${fibCount} calls to get here`);  
}
```

When we run this code we should see how many recursive calls have been made. Let's see:



A terminal window titled '..os/algorithms (-zsh)' shows the execution of 'node fib.js'. The output displays the Fibonacci sequence from position 0 to 10, with the number of recursive calls taken for each. The number of calls grows rapidly, reaching 454 for position 10.

Position	Fibonacci Number	Number of Calls
0	0	2
1	1	3
2	1	6
3	2	11
4	3	20
5	5	35
6	8	60
7	13	101
8	21	168
9	34	277
10	55	454

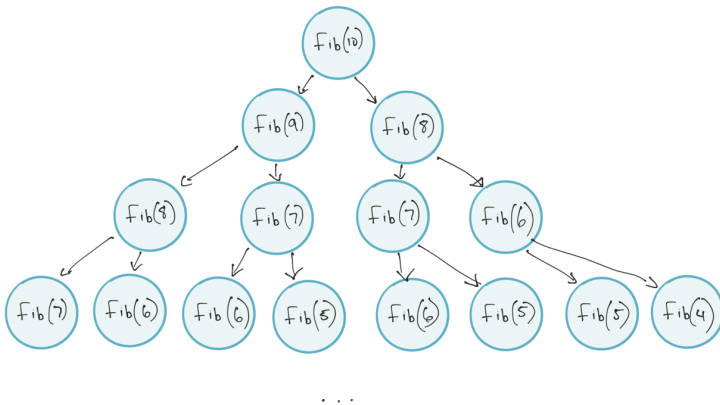
453 calls! Good grief! As you can see, the number of calls to our routine goes up more than exponentially with each additional input:

- Input 0 resulted in 1 call
- Input 1 resulted in 2

- Input 2 resulted in 5 calls
- Input 6 resulted in 59
- Input 10 resulted in 453

If you really want to have some fun, let it run for 15 minutes and see how many calls it takes to calculate **calculateFibAt(32)**... it's 18,454,894!!!

Another way to think about this is from the top down. Here we can visualize the complexity for calculating the Fibonacci number in 10th position using a graph:



This is horribly inefficient. Look how many times **fib(6)** and **fib(7)** are called! The interviewer seems happy with our answer and then asks us:

*Tell me about the space complexity...*

Right. We learned from the last section that a recursive routine will push values onto the stack repeatedly, once for every single call of the current function. If you do this enough, you'll run into a stack overflow exception, which is bad.

As mentioned before: *recursion and space complexity aren't friends*.

*Sounds good. So how would you improve this routine?*

This is where we get to the next step of dynamic programming. We can reduce the time and space complexity of our algorithm by using memoization. We can do this because the solution to each subproblem is optimal, meaning that it can stand alone, and we don't need anything else to use its value.

In Big-O terms, we can use the memoized solution in linear time,  $O(n)$  where  $n$  is the position we're interested in, which will speed things up tremendously. What about *space complexity*? Can you figure out a way to do this in constant space? I'll talk about that in the next section.

## **The Faster Way**

Memoization is simply caching. In more formal terms it's remembering the solution to a subproblem, so you don't have to calculate it again recursively. This only works if the subproblems are in an optimized substructure. You can think of that as a large graph (like

the one above), where you can simply replace **fib(6)** with the number 8. That's darn optimal if you ask me.

To accomplish this, I'll store the results of our loop in some kind of data structure; the question is *which one*? We've learned about a whole mess of them in a previous chapter... which would be the best?

All we need to do is to remember some values in memory and then to iterate over them. If this is all you need, *don't overthink it!*

Since we know that Fibonacci numbers start with 0 and 1, I can use those seeds to calculate the remaining numbers:

```
//the fast way
const calculateFibFaster = (n) =>{
  var memoTable = [0,1];
  for(let i=2; i <= n; i++){
    memoTable.push(memoTable[i-2] + memoTable[i-1]);
  }
  return memoTable;
};

//run it the fast way
console.log(calculateFibFaster(10));
```

The result, when run:

**[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]**

Perfect. But what about *time complexity*? This is simple to reason through, but before we do let's run this faster Fibonacci routine 1000 times. You should see that it returns almost instantly.

## Complexity Analysis of our Faster Method

We want to find the Fibonacci numbers up to a given number  $n$ , which means we'll need to perform some operation for each  $n$ . If you recall from a few chapters back, iterating over a collection is always  $O(n)$ . Accessing a value from an array using its index is always  $O(1)$ , so our total operation here is  $O(n * 1)$  which is  $O(n)$ .

But what about the *space complexity*? We're not using recursion so that means we won't be potentially overloading the stack, which is a good thing. We have a few variables and an array for every  $n$  number that we're evaluating, so our space complexity is also  $O(n)$ .

You might be thinking "hey wait a minute you have a loop variable in there too!" and yes, that's true, but with Big-O you're more concerned about the *nature* of the algorithm. In this case it's simply  $O(n)$ .

Can we do better here? *Yes*. Well... sort of. Now, we're returning an array of Fibonacci numbers up to the *n*th number. We could ask our interviewer at this point if they're interested in the whole sequence or just the *n*th number?

*Just the n*th number will do.

Perfect. This means we can now use a *greedy algorithm*, which is a term you should remember for interviews. Let's take a small diversion (again).

## GREEDY ALGORITHMS

A greedy algorithm does what's best at that moment. Put in math terms:

*A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.*

Say what?

Let's do this with code, then I'll see how well I can explain the idea. We'll start by redoing our **fibFaster** function:

```
const fibConstantSpace = function(n){
  let twoFibsAgo = 0, oneFibAgo=1, currentFib=0;
  //make sure to output the first two
  for(let i=2; i<=n; i++){
    currentFib = twoFibsAgo+oneFibAgo;
    twoFibsAgo = oneFibAgo;
    oneFibAgo = currentFib;
  }
  return currentFib;
}

console.log(fibConstantSpace(10));
```

Running this code we end up with the last Fibonacci number in our sequence, which is **55**.

The best part, however: *no more arrays required!* What we're doing here is simply "remembering" *only what we need to remember*, essentially "locally optimizing" our decision making by setting these values to variables and forgetting the rest.

**That's a greedy algorithm** in practice. Our interviewer likes this, but has a question:

*Can you think of a way to make this a bit more flexible for calling code? Right now you're just returning the nth fibonacci number; what if I wanted to do something else in my calling code?*

If this happens to you in an interview, take it as a good sign. You've nailed the question so far! They're happy with your response and are likely just wanting to see how much better you are than the question allows.

We're using JavaScript, so ideally the answer is jumping out at you. Most languages support the idea of *callbacks*, something that will yield control of the current iteration/operation. We can use that here to *yield* the **currentFib** value back to the calling code:

```
var fibConstantSpace = function(n, fn){
  let twoFibsAgo = 0, oneFibAgo=1, currentFib=0;
  //make sure to output the first two
  if(fn){
    fn(0);
    fn(1);
  }
  for(var i=2; i<=n; i++){
    currentFib = twoFibsAgo+oneFibAgo;
    twoFibsAgo = oneFibAgo;
    oneFibAgo = currentFib;
    if(fn) fn(currentFib);
  }
}

fibConstantSpace(10, console.log)
```



There we go! The best of both worlds: we can report back each number so the calling code can do whatever it wants, or we can just return the final value. Running this code you should see the same list of Fibonacci numbers that we've been working with.

## Use In Heuristics

Greedy algorithms can be useful when solving some very complex problems. A few chapters ago we examined a few very tough, NP-Hard optimization problems, one of which was The Traveling Salesman Problem.

One way to approximately solve this problem is using a heuristic (that's a fancy word for "rule of thumb") called Nearest Neighbor:

*The nearest neighbour algorithm was one of the first algorithms used to determine a solution to the traveling salesman problem. In it, the salesman starts at a random city and repeatedly visits the nearest city until all have been visited. It quickly yields a short tour, but usually not the optimal one.*

In other words: *there's no master plan here*. Nearest Neighbor just looks at the next cheapest city and goes there. This is a classic *greedy algorithm*.

Another greedy solution is finding your way out of a maze. You just put your right hand on the nearest wall and keep walking until you're out. Not the *optimal* solution, but it *will* solve the problem.

For a final example: consider Agile Development. Teams gather quickly in the morning so everyone's aware of what's going on with everyone else. Adjustments are welcomed and deployment rapid - it's all about quick adaptation to the changes in the development process.

Now, if you were to step back and look at a software project as a series of decisions which you could represent on a graph (which you can), then Agile is, itself, a greedy algorithm! It might not be the *optimal* solution to success for a project, but it is a solution! You're simply doing the next, closest thing that makes the most sense to the team and client without a master (waterfall) plan in place.

## BELLMAN-FORD

Right then, done with Fibonacci thank goodness. On to graph traversal! One of the most fascinating uses of graphs is in the optimization of path traversal, which can be used in a vast number of calculations.

As mentioned in the previous chapter, graphs can be used to represent all kinds of information:

- A network of any kind. Social (friends) or digital (computers or the internet), for example
- A decision tree

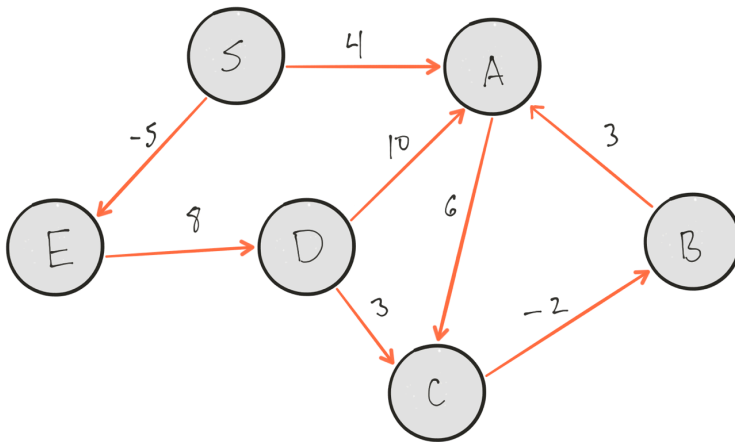
- Contributions from members of any kind to a cause of any kind
- Atomic interactions in physics, chemistry or biology
- Navigation between various endpoints
- If you apply weighting to the edges or vertices, you can run useful calculations for just about anything. One of the most common is finding the shortest path between two vertices.

There are numerous algorithms to touch on at this point, but I must round this chapter out by discussing the two you should be aware of: Bellman-Ford and Dijkstra. In this section we'll discuss Bellman-Ford; Dijkstra comes next.

## **The Problem Graph**

This algorithm is named after Richard Bellman (the same person who wrote about dynamic programming) and Lester Ford Jr. It shares a lot with Dijkstra's algorithm (which we'll see next), but has one major advantage: it can accommodate negative edges.

Let's see how it works. Consider this graph:



This is an edge-weighted, directed graph. We want to calculate the shortest paths between our source vertex  $S$  and the rest of the vertices,  $A$  through  $E$ . We can do this using dynamic programming.

Now, if you were to stare at this and I told you what your task was (to calculate the smallest cost between  $S$  and the rest of the vertices), you would probably get overwhelmed! I know I did.

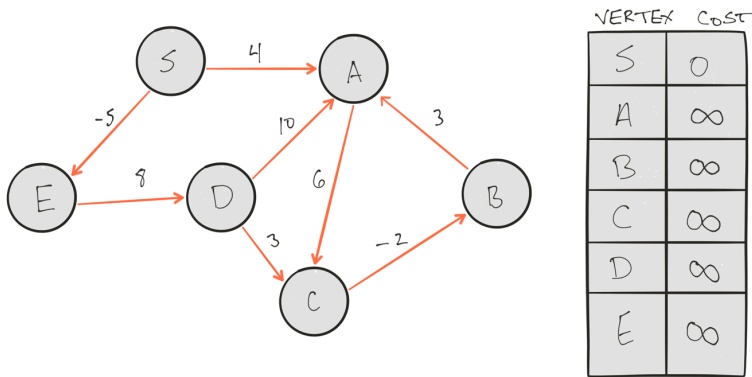
The good news is that we have the Bellman-Ford algorithm, which makes this calculation rather fun. Rather than spend many words on it, let's do it visually, then implement it with some code.

We will be using memoization to keep our calculations in  $P$  time, so let's setup a memoization table that we can update as we run our calculations. We know that there are only 6 vertices, so we only need to track, at most, 6 total costs. The only cost we know is the base cost:  $\delta(S, S) = 0$

**Note:** if you're not familiar with that squiggle,  $\delta$ , it's a "delta", which indicates a difference. This equation states that the difference (or "cost") from S to S is 0.

For the rest of the vertices, we don't know so we'll set them all to infinity. Why infinity? Simply because the calculation we do later will be based on finding the smaller value, and infinity guarantees that the initial state will not remain.

Here's our setup:



Great. The plan is to calculate the cost of each outgoing edge, for each vertex in our graph. This will be 6 total calculations which we'll repeat in iterations. The number of iterations  $i$  you need to perform when using Bellman-Ford is:  $i = |V| - 1$ .

Why is this? You'll see in a second, but the crux of it is that we're calculating the distances between all nodes and remembering the

smallest ones. This is called relaxation: we start with the largest values we can think of (infinity) and then slowly relax the costs through an iterative calculation.

OK, enough words, let's do this.

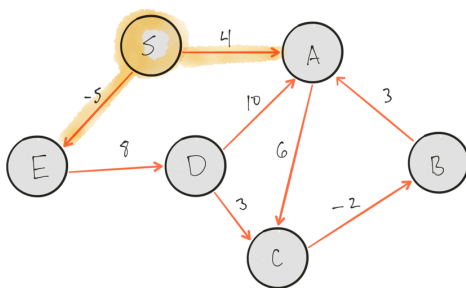
### **Iteration 1**

We'll start with our source,  $S$ , and move clockwise around the graph. From  $S$  we have two outgoing edges with the values 4 and -5. These costs ( $C_{sa}$  and  $C_{se}$ ) associate  $S$  with  $A$  and  $E$  so we'll add them to our table using this calculation:

$$C_{sa} = \delta(S, S) + \delta(S, A)$$

$$C_{se} = \delta(S, S) + \delta(S, E)$$

You can visualize this on the graph itself. We're using the cost of the highlighted edges below and adding them to the initial cost of  $S$  to  $S$  (which is 0):

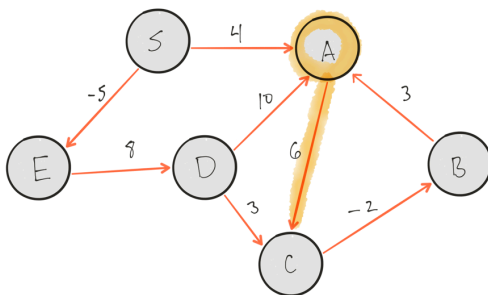


VERTEX COST

S	0	
A	4	0 + 4
B	$\infty$	
C	$\infty$	
D	$\infty$	
E	-5	0 + -5

If this seems complicated, let it wash over you. As we move through this it will make more sense.

OK, we're still in our first iteration of calculating the costs between S and the rest of the vertices. Let's now move to A and calculate the outgoing edges from A:



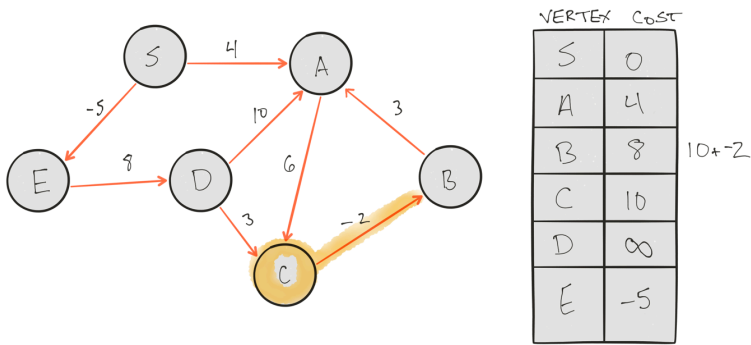
VERTEX COST

S	0	
A	4	
B	$\infty$	
C	10	4 + 6
D	$\infty$	
E	-5	

A only has a single outgoing edge with a cost of 6, so we take that cost and add it to A's current cost (which is 4). This gives us a value of 10, which we add to our memo table.

Now we move on to B, which has a current cost of infinity. This means that we don't have a path from S to B yet, which means we can't calculate it in this iteration. So we skip it.

Next up is C, which does have a current cost of 10. It also has an outgoing edge to B which is good news as we'll be able to use that in the next iteration. For now, let's update our table:

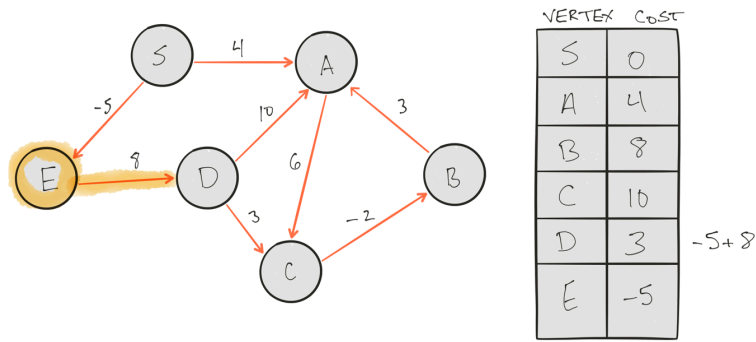


The current cost of the path to C is 10, so adding -2 to that (which is the cost of the edge between C and B) gives B a current cost of 8.

Now we're up to D. What do you think we do here? The current cost is set to infinity, which means we haven't calculated a path to it yet, so we skip it.



Finally, we close off this iteration by calculating the outgoing edges from E:

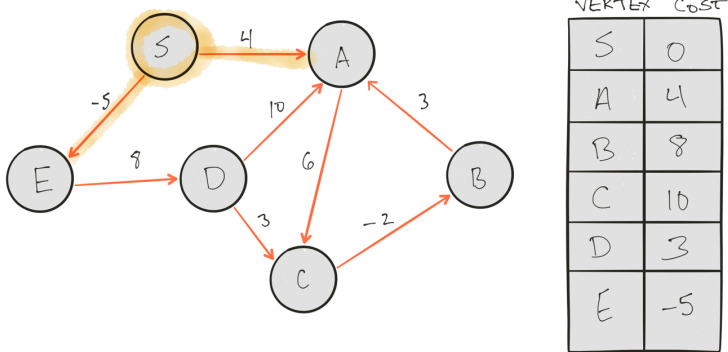


By now you should be able to reason what will happen. We know that E has a current cost of -5, so we add that to the cost of the edge between E and D, which is 8. This gives a cost from S to D equal to 3.

Great! We now have costs for all the vertices! It's time to do another iteration to see if we can relax these values a bit more.

### Iteration 2

Let's step through this a bit quicker this time. We'll start with S again, our source, and note that the cost of the outgoing edges does not improve on the costs we've recorded (4 and -5). This makes sense as these are single edges and there really is no way to improve the costs here:



If we move to A, again the only outgoing edge we can evaluate is A to C, which is 6. The cost from S to C remains the same at 10, so there's no improvement here and we can move on.

Next up is B, which now has a cost associated with it so we can use it in our calculations. The current cost to get to B is 8, so getting from B to A is 11 total, which is not an improvement over A's current cost of 4.

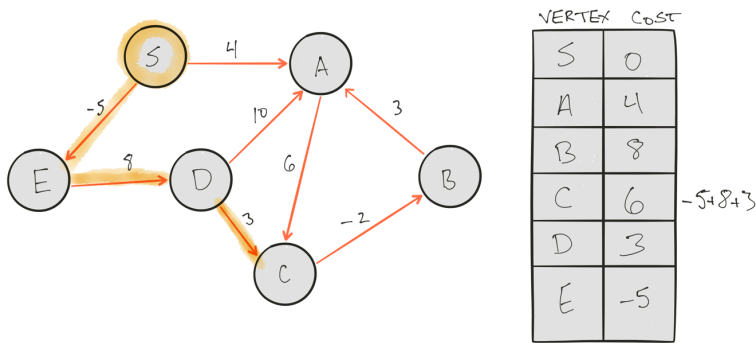
Then we come to C, which is 10. We've already calculated C to B as 8 and there's no improvement here, so we can move on again.

Seems a bit boring, doesn't it? We keep skipping things – but that's OK! It's about to get a bit more exciting when we consider D, our next vertex.

The current value of D is 3 and has two outgoing edges to A and C. The value of the edge from D to A is 10, so adding the current cost

of D (3) to this edge cost of 10 would be 13. This doesn't reduce A's cost so we leave it. But what about edge D to C?

This produces a cost of 6, which means we can lower C's cost in our memo table to 6:



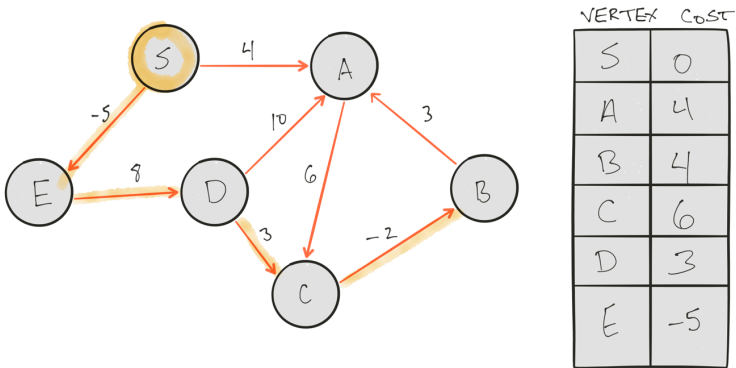
The excitement! Can you feel it! Now, as you might imagine, changing C like this means we'll be able to change more values in our table – but that will have to wait for the next iteration.

Speaking of, we're back to E now and there are no changes we can make here, so we'll skip on to the next iteration.

### Iteration 3

We skipped S and A last time and that's still the case this time: there are no changes we can make to improve our current costs for these vertices. Same for boring old B, again. We can't improve the cost of A, which is its only outgoing edge.

C, however, makes things interesting. The current cost of C is now 6, which means we can reduce the cost of B to 4:



Oh my that’s so exciting isn’t it! From this point we go to D and E just like before, skipping them as there’s no way to improve their costs.

We can also use some more reasoning here. Since B was the only change we made from the last iteration, we can just evaluate its outgoing edge to see if it will improve A. Since B’s current cost is 4 and A’s current cost is 4, then no improvement will happen if we add the cost of the edge from B to A, which is 3.

So we’re done! The shortest paths for each vertex are now calculated.

## Implementing With JavaScript

I'm sure you'll find some ways to improve this code, which is great! I've made it a little more verbose for clarity, which I think will help if you're still having problems getting your head around this algorithm.

```
//define the vertices - these can just be string values  
var vertices = ["S","A", "B", "C", "D", "E"];  
  
//our memoization table, which I'll set to an object  
//defaulting as described  
var memo = {  
  S : 0,  
  A : Number.POSITIVE_INFINITY,  
  B : Number.POSITIVE_INFINITY,  
  C : Number.POSITIVE_INFINITY,  
  D : Number.POSITIVE_INFINITY,  
  E : Number.POSITIVE_INFINITY  
}
```

As you can see we have a simple array and a JavaScript object to serve as our memo table.

Now we need to define the graph itself. For this I simply need to track which vertices are involved and the costs associated with the relationship:

```
//this is our graph, relationships between vertices  
//with costs associated  
var graph = [  
  {from : "S", to : "A", cost: 4},  
  {from : "S", to : "E", cost: 6},  
  {from : "A", to : "C", cost: 6},  
  {from : "B", to : "A", cost: 3},  
  {from : "C", to : "B", cost: 2},  
  {from : "D", to : "C", cost: 3},  
  {from : "D", to : "A", cost: 10},  
  {from : "E", to : "D", cost: 8}  
];
```

Looking good! At this point you should be able to reason how we'll use these data structures to iterate over our graph. In short, we need to:

- Iterate over the vertices array
- Using the current vertex from our vertices array, we select the outgoing edges from the graph array.
- Once we have the outgoing edges, we run a quick calculation using our memo object. The calculation is straightforward: we take the cost of the current vertex and add it to the cost of the current outgoing edge. If that value is less than the cost of the current edge, we update the memo for the current edge.

Translating that word salad to code:

```

//represents a full iteration of Bellman-Ford on our graph
const iterate = () => {
  //do we need another iteration?
  //decided below
  let doItAgain = false;
  //loop all vertices
  for(fromVertex of vertices){
    //get the outgoing edges
    const edges = graph.filter(path => {
      return path.from === fromVertex;
    });
    //loop the outgoing edges
    for(edge of edges){
      const potentialCost = memo[edge.from] + edge.cost;
      //reset the cost as needed
      if(potentialCost < memo[edge.to]){
        memo[edge.to] = potentialCost;
        //if the cost was changed we need to loop again
        doItAgain = true;
      }
    }
  }
  //return the flag
  return doItAgain;
}

```

Great! Now all we need to do is to iterate over our iterate function and we're good to go:

```
for(vertex of vertices){  
  //loop until no changes  
  if(!iterate()) break;  
}  
console.log(memo);
```

You'll notice that I'm only iterating to `vertices.length - 1`. Can you reason why? If you run this code (using Node), you should see:

**{ S: 0, A: 4, B: 4, C: 6, D: 3, E: -5 }**

These are the exact values of our exercise above. Nicely done!

## Analysis and Summary

This is dynamic programming in action. Dividing the objective problem (finding the shortest paths) into smaller problems (calculating the costs between each vertex). We then recursed over the smaller problems (the **iterate** function) and used memoization to derive the answer.

There is room for improvement, however. Think about the process we went through in the first section. I only needed 3 total iterations to derive the shortest paths. The code I wrote, however, required 5. How would you optimize this?

In addition, I'm not using recursion in a code sense. I am calling the same function repeatedly, but there's probably a tighter, cleaner way to do this using a true recursive function. Can you see how?



Finally, here's something to ponder: does the order in which we evaluate the vertices matter? If yes, why? If no ... why not? Rearrange the code a bit and see if you come up with a different answer than you see here.

The Bellman-Ford algorithm is quite effective, as we can see, but it can also take a long time to run. In terms of complexity, the algorithm runs in  $O(V * E)$  time, where  $V$  is the number of vertices and  $E$  is the number of total edges. It can work well for simple graphs, but for more complex (or dense) graphs, it is not the most efficient algorithm.

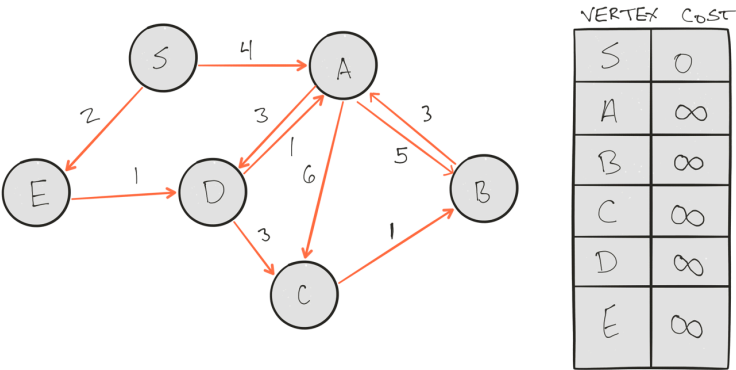
## DIJKSTRA

In the last section we iterated over a simple graph using Bellman-Ford to find the shortest paths from a single vertex (our source) to all other vertices in the graph.

The complexity of Bellman-Ford is  $O(|V|E)$ , which can approximate  $O(n^2)$  if every vertex has at least one outgoing edge. In other words: *it's not terribly efficient*.

Dijkstra's algorithm requires only one iteration, however and has a complexity of  $O(|V| \log V)$ , which is much more efficient. Let's see why.

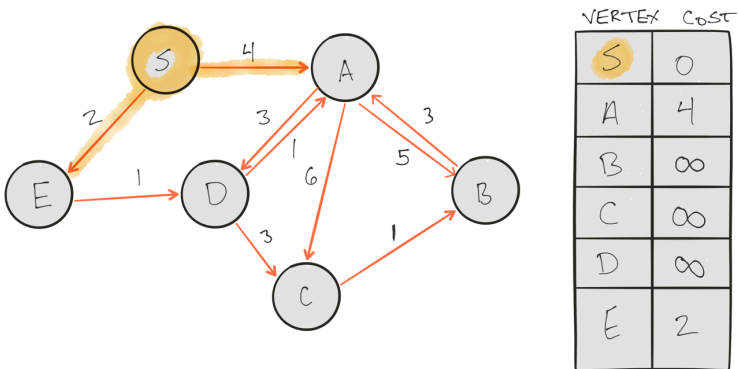
As with Bellman-Ford, we'll use a directed, weighted graph with 6 vertices. In addition, we'll setup a memo table with our source S set to 0 and the rest of the vertices set to infinity:



There is a difference here, however, and it's critical! Dijkstra doesn't work with negative edge weights! I have adjusted this graph so that we don't have any negative weights, as you can see. Specifically, the edges between S and E as well as C to B. In addition, I've added a few edges to show that the algorithm will scale easily regardless of the number of edges involved.

### Starting At The Source

The first step is to evaluate the source, S. We do the same thing as before, with Bellman-Ford, where we tally up the cost of the outgoing edges and store them in the memo table. In addition, we'll mark S as complete:



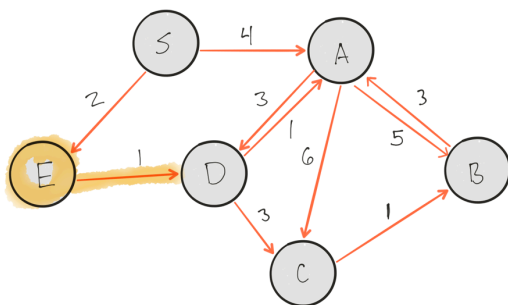
This is another way that Dijkstra differs from Bellman-Ford: each vertex is visited only once.

The next vertex is chosen using these rules:

- It must not have been visited previously
- It has the smallest cost of the remaining unvisited vertices In our case, this would be vertex E.

## Traversing Each Vertex

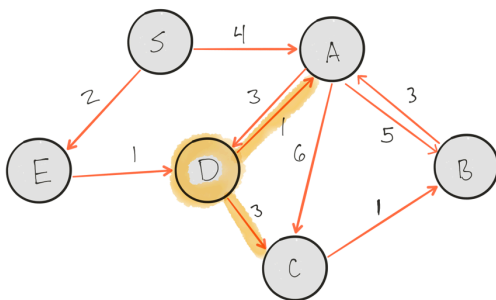
The next vertex that we'll visit is E. The cost to reach E is 2, which is less than 4, and E has not been visited previously:



VERTEX COST	
S	0
A	4
B	$\infty$
C	$\infty$
D	3 <span style="margin-left: 10px;"><math>2+1</math></span>
E	2

We then update the memo table, setting D to 3 (which is the cost of S to E plus the cost of E to D). We then mark E as visited.

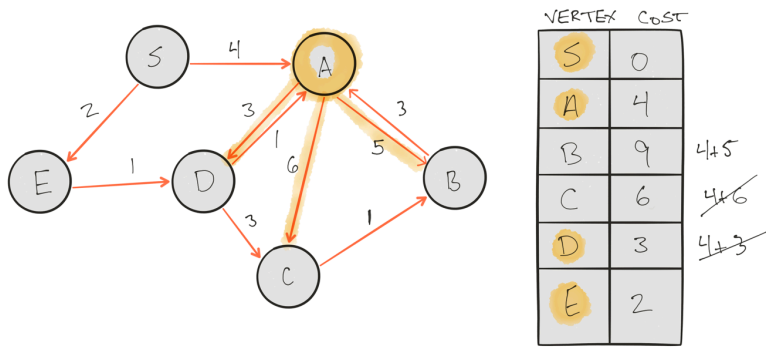
The next vertex in our table that is unvisited with the lowest cost is D, so we calculate that next:



VERTEX COST	
S	0
A	4 <span style="margin-left: 10px;"><math>3+1</math></span>
B	$\infty$
C	6 <span style="margin-left: 10px;"><math>3+3</math></span>
D	3
E	2

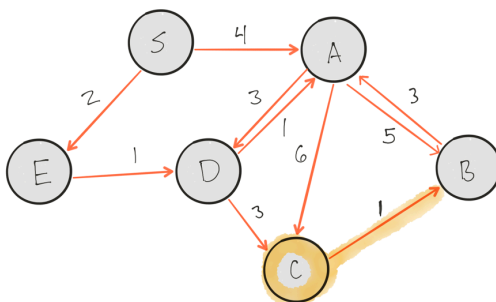
D has two outgoing edges: to A and to C. The cost of A would be  $3+1$  which is 4 and no improvement, so we leave A as it is in our table. C has no cost, so we update it to 6, which is the current value of D (3) + the cost to get to C.

The next vertex we'll visit is A. It has the least cost and has not been visited before:



The current cost of A is 4 and the edges going out of A go to vertices D, C and B. The cost of reaching D through A is 7, which is not an improvement of D's current cost of 3, so we leave it. Same with C: reaching C through A does not reduce C's cost, so we leave it. B, however, is still infinity so we'll set it to 9, which is A's cost plus the cost to reach B, which is 5.

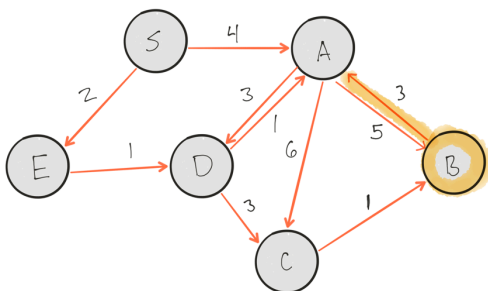
C is the next vertex we'll choose as its current cost is 6 and it hasn't been visited:



VERTEX COST	
S	0
A	4
B	<del>6</del> 7 $6+1$
C	6
D	3
E	2

C has only one outgoing edge: to B. The current cost of C is 6 and adding 1 to it would be less than the current cost of B, so we'll update B's cost to 7.

There is only one vertex left, which is B:



VERTEX COST	
S	0
A	4
B	7 $7+3$
C	6
D	3
E	2

The only candidate for improvement is A, with a current cost of 4. This is less than the tentative cost of B to A, which is 10, so we leave it as it is. We're done!

All vertices are visited, so we're done! We can be sure we calculated the shortest path by using our Bellman-Ford code from the last section to make sure it matches, and it does!

## **Implementing With JavaScript**

The code you're about to see is probably a lot more verbose than you might write it. I like clarity, mostly for my own sake, because you can bet I'll be looking over this page and this code quite a few times in the future! I want to remember what I was thinking.

To start with, let's alter our implementation of Bellman-Ford in the last section to have a memo table with a little more smarts:

```

//The memoization table, which needs to have some smarts
//using an ES6 class here
class MemoTable{
  //build the table using the passed-in vertices
  constructor(vertices){
    //set the root manually
    this.s = {name: "s", cost: 0, visited: false};
    this.table = [this.s];
    //add the vertices, defaulting cost to infinity and visited to false
    for(var vertex of vertices){
      this.table.push({name: vertex, cost: Number.POSITIVE_INFINITY, visited: false});
    }
  };
  //all non-visited vertices
  getCandidateVertices(){
    return this.table.filter(entry => {
      return entry.visited === false;
    });
  };
  //lowest cost, non-visited vertex
  nextVertex(){
    const candidates = this.getCandidateVertices();
    //if there are candidates, find the one
    //with lowest cost
    if(candidates.length > 0){
      return candidates.reduce((prev, curr) => {
        return prev.cost < curr.cost ? prev : curr;
      });
    }else{
      //otherwise return null
      //this will help determine if we need to
      //iterate
      return null;
    }
  };
  //update current cost
  setCurrentCost(vertex, cost){
    this.getEntry(vertex).cost = cost;
  };
  setAsVisited(vertex){
    this.getEntry(vertex).visited = true;
  };
  getEntry(vertex){
    return this.table.find(entry => entry.name == vertex);
  };
  getCost(vertex){
    return this.getEntry(vertex).cost;
  };
  toString(){
    console.log(this.table);
  }
};

```



I've added the logic for retrieving a given entry as well as updating its values. In addition I've added logic for determining the next vertex to traverse to based on the rules of Dijkstra that we saw above.

Next, we have our graph:

```
//the vertices for our memo table
//I could also use a filter or map on the graph below
//to avoid duplication; but this is nice
//and clear
const vertices = ["A", "B", "C", "D", "E"];
//our graph
const graph = [
  {from: "S", to: "A", cost: 4},
  {from: "S", to: "E", cost: 2},
  {from: "A", to: "D", cost: 3},
  {from: "A", to: "C", cost: 6},
  {from: "A", to: "B", cost: 5},
  {from: "B", to: "A", cost: 3},
  {from: "C", to: "B", cost: 1},
  {from: "D", to: "C", cost: 3},
  {from: "D", to: "A", cost: 1},
  {from: "E", to: "D", cost: 1}
];
```

This array represents the visual graph we worked with above. Now we just need to evaluate our graph using our **MemoTable** functionality:

```

//create the table
const memo = new MemoTable(vertices);
//let's do this!
const evaluate = vertex => {
  //get the outgoing edges of the vertex
  const edges = graph.filter(path => {
    return path.from === vertex.name;
  });
  //loop the edges...
  for(edge of edges){
    //calculate the costs
    const currentVertexCost = memo.getCost(edge.from);
    const toVertexCost = memo.getCost(edge.to);
    const tentativeCost = currentVertexCost + edge.cost;
    //if we can improve the cost to the
    //connected vertex...
    if(tentativeCost < toVertexCost){
      //do it!
      memo.setCurrentCost(edge.to, tentativeCost);
    }
  };
  //set this vertex as visited
  memo.setAsVisited(vertex.name);
  //get the next vertex
  const next = memo.nextVertex();
  //if there is a next vertex, let's do this
  //again...
  if(next) evaluate(next);
}
//kick it off from the source vertex
evaluate(memo.S);
memo.toString();

```

The code looks a bit familiar – it follows what I did (for the most part) with Bellman-Ford but this time I've added a few tweaks to accommodate setting a vertex as visited, and I'm also using recursion off the memoization table instead of a loop.

What do you think? Can you improve this without losing its clarity? Let's run it and make sure it works. Again, using Node:

```
~/Demos/algorithms } master ± node shortest_paths/dijkstra.js
[
  { name: 'S', cost: 0, visited: true },
  { name: 'A', cost: 4, visited: true },
  { name: 'B', cost: 7, visited: true },
  { name: 'C', cost: 6, visited: true },
  { name: 'D', cost: 3, visited: true },
  { name: 'E', cost: 2, visited: true }
]
```

Right on! That's the exact answer we got above!

# INDUSTRIAL AGE COMMUNICATION

**W**e'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.

# IN CONVERSATION

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

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 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.

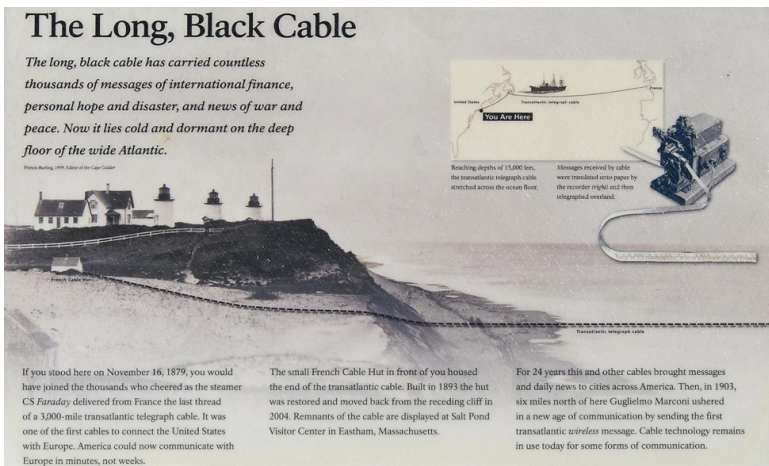
These were tempting enough alternatives to the long, dangerous journey overland through "Indian territory", over the cold Rockies, across rivers and once again over the Sierra Nevada that many people went ahead and made the voyage. You could see why having a railroad would be preferable: the journey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

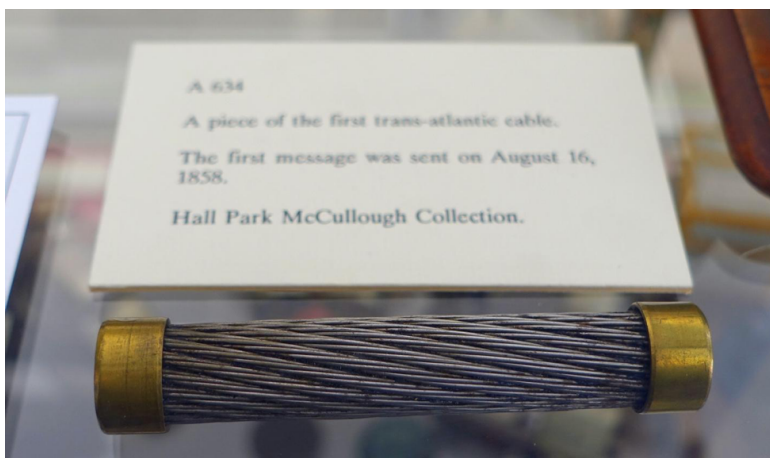
*But that proved insufficient for protecting the conductor, and the cable degraded. By the time the celebratory banquet was held on 1 September, it was almost impossible to receive signals, and by 18 September, the cable was useless.*

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:



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 in 1917 in their research department, which later turned into Bell Laboratories. His focus was to make communication more efficient for both telegraph and telephone.

He wrote some theoretical musings in his 1924 paper *Certain Factors Affecting Telegraph Speed*, which isn't a terribly exciting title, but the synopsis was spot on:

## Certain Factors Affecting Telegraph Speed<sup>1</sup>

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 others<sup>2</sup> and that their arguments are based on erroneous assumptions.

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.

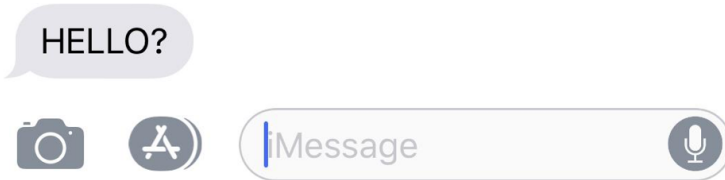
Voice was an even worse choice of code, given the *redundancy* required by formality. Consider the simple phone call, it always opens with the same word:

*A hundred years ago, the word "hello" spoken in Arlington, VA was heard in Paris. The word originated from the vibrations in the vocal cords 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.*

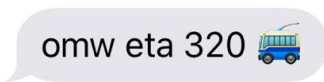
Not to take away from the achievement: it's brilliant. But the first words directly spoken between the continents was "hello"! To humans, a pleasant way to break the ice and begin a conversation. But as far as information theory is concerned, mere social pleasantries are utterly meaningless and can be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



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:



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.

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 using 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 the set of total messages possible. 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-science 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:

1. *gains* when a message is received and therefore implicitly
2. *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

Claude Shannon is largely unknown in the tech world, which is unfortunate. If you start a conversation about his work, you could talk for *hours* and perhaps entertain/enlighten people or, more likely, bore them to death.

The inventor of the digital circuit and the sole creator of the Information Age, Claude Shannon is up there with Einstein in terms of his impact on our modern lives.

## THE MATHEMATICAL THEORY OF COMMUNICATION

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.

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 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:

1. He was a humble, private, playful man who did not enjoy or seek self-promotion at all, and
2. 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 bird-song 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:

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

Number 6 is interesting. Shannon was the first to use the term "bit" as 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 now, conveying a simple message:



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  $p$ :

$$0 \leq p \leq 1$$

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, something like 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  $0.1 \times 0.1$  (.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 could 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 calculation of surprise of a message*

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 mes-

sages 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:

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

*The Shannon Entropy*

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 THERE-  
FORE 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

**L**et's take a break for a minute to contextualize things regarding 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 regarding 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 10, a message with two digits would have an  $M=10*10=100$ . Therefore,  $H$  is:

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

My password would have 6.64 *bits* of entropy according to Ralph Hartley <sup>1</sup>.

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".

---

<sup>1</sup> *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.*

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, it 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 a 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, dig-

its 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:

1. What is the entropy for the set of ASCII characters?
2. 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  $M * M * M * M$ . 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 would execute in under a minute. 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 regarding 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 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 an elementary way to do this *and* to keep your users happy.

## THE ENTROPY OF PASSPHRASES

We need more entropy, but we want to avoid making 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 the upper/lower alphabetical character set, numbers and a few symbols. That leaves us with an  $\aleph$  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 many 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:

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

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.

## IN CONVERSATION

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.

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."

All the core pieces are in place, including:

1. The nomenclature: source, message, channel, bit, etc.
2. The probabilistic nature of successive messages
3. Bits measure the information in a message
4. 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 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.* <sup>2</sup>

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.

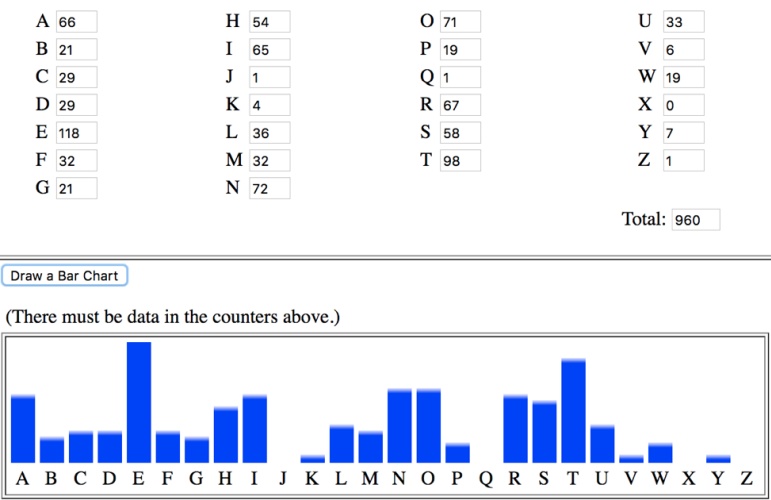
---

<sup>2</sup> *This is 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*

# 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):



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: etaoin-shrdlu-cmfgy-pwbv-kxj-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 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 to 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 with-

out 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:

1. Extreme Excitement
2. Excitement
3. Mild Excitement
4. General Happiness
5. Ambivalence
6. 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:

**1. OMFG!**

**2. !**

**3. OMG!**

**4. OMG**

**5. 🙄**

**6. 💩**

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? Here's one possible decoding:



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:

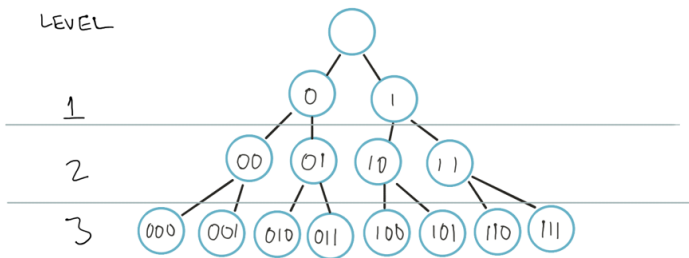
101 000 100 (🙄 M)  
10 100 01 00 (OMG!)

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.

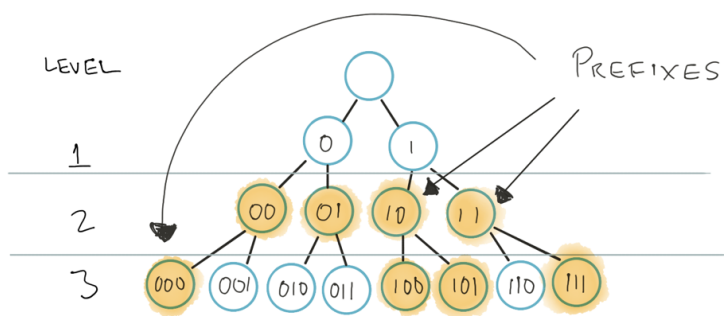
## Using a Binary Tree to Visualize Prefix-free Encoding

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:

**1. OMFG!**

**2. !**

**3. OMG!**

**4. OMG**

**5. 🙄**

**6. 💩**

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.

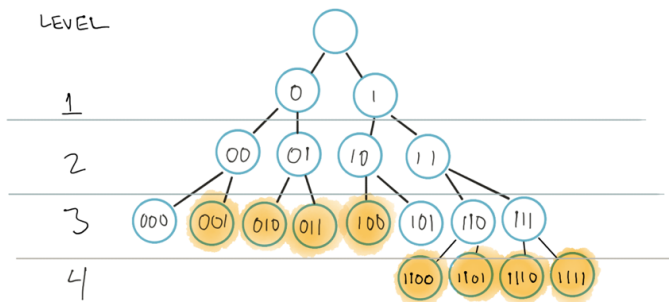
How can we apply this understanding, then, to our Lovebug encoding? If you recall: the amount of information in bits in a given message (according to Shannon) is the *surprise*. All we need to do is to calculate the surprise for each message and count the bits to know how long our code word should be. 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.

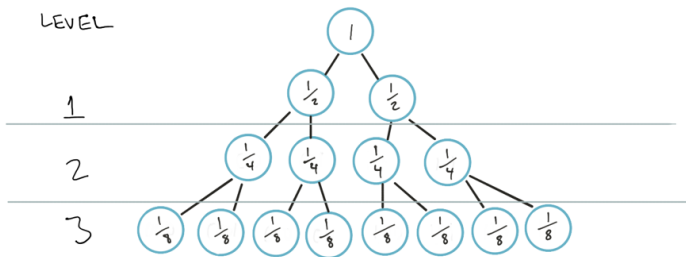
So: according to our Shannon calculation, we should be able to create a prefix free code by creating 4 x 4-bit code words and 4 x 3-bit code words. It's Christmas again! Let's decorate our binary tree:



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 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:

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

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 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

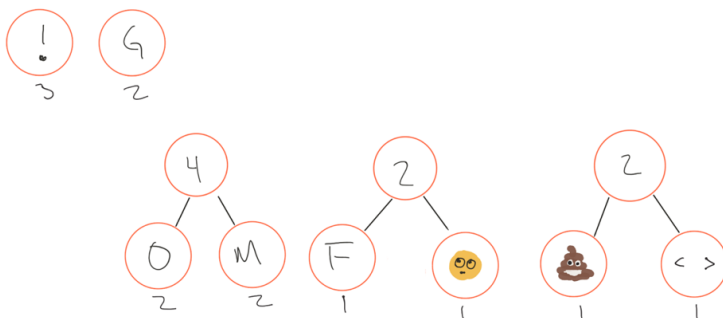
Once we do this, we create a set of nodes, one for each character in our set, sorted by occurrence:



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>:

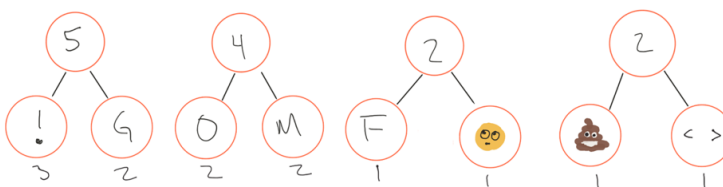


Simple enough! Now we do the same thing with the next nodes in line, again going from least to greatest:



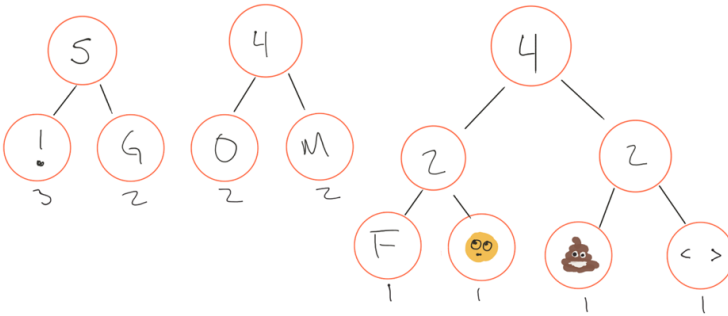
We have two new nodes of 2 and one new node of 4. The remaining nodes are a 2 and a 3, which we can combine in the next step. If, however, these were 3 and 3, our next step would be to combine nodes 2 and 2! For this algorithm to work, we need to be sure all the smaller nodes are combined *before* we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

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

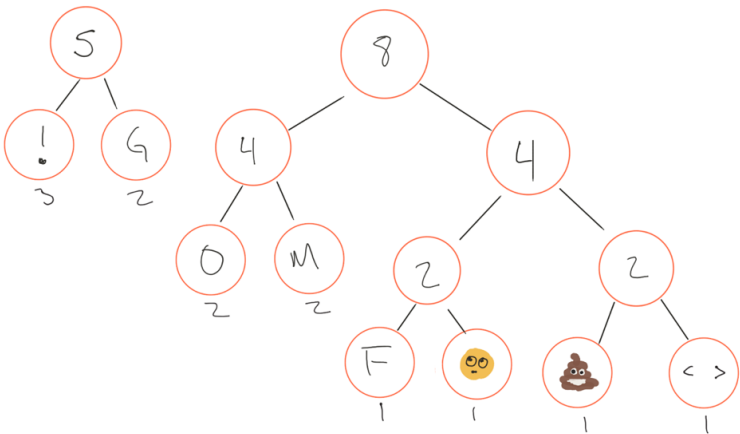


Great. Now we repeat the process, combining from right to left:

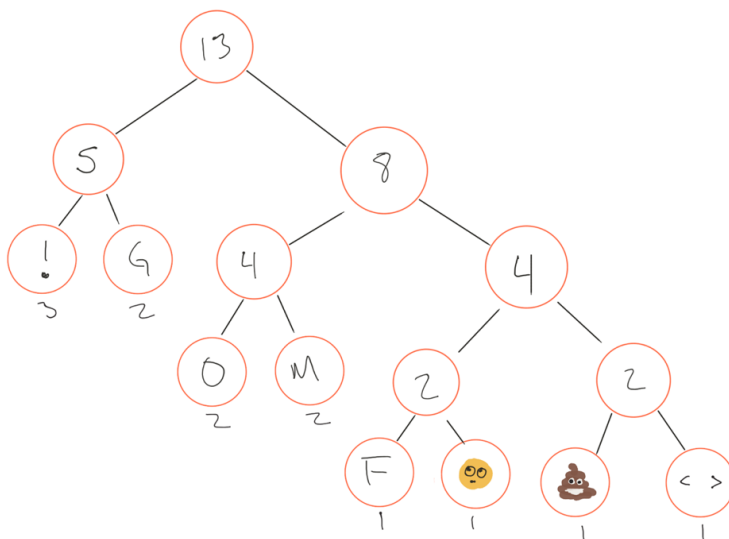




We've combined 2 and 2 in this pass... and that's all we can do for now, since combining 5 and 4 would result in a node with a value of 9, which would take us out of order. 9 is higher than 4 and 4, which is 8. So we start another pass and do that next:

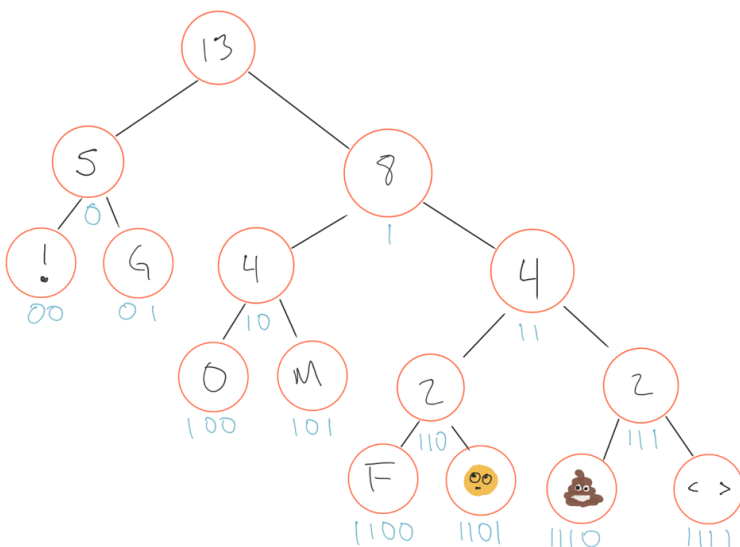


Only one step left now:



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

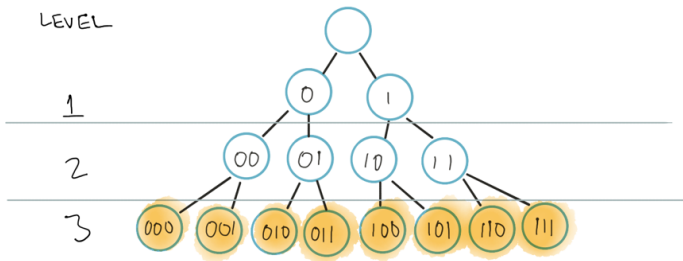
We get *exactly* 1. Beautiful, isn't it?

YES, BUT...

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

$$H = \log_2(8) = 3$$

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



Looks 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

**T**he 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.

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 canonical example for this is the noisy room. You and me 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:

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

This is precisely 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.

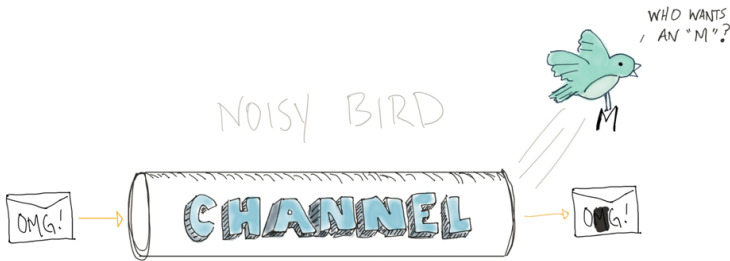
# ILLUSTRATION OF A NOISY CHANNEL

In the perfect world of a noiseless channel, we can consider the message going into the channel to be the 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 from your voice; they're the same, even though the echo is slightly quieter and a bit delayed.

From this we know something:

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

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:

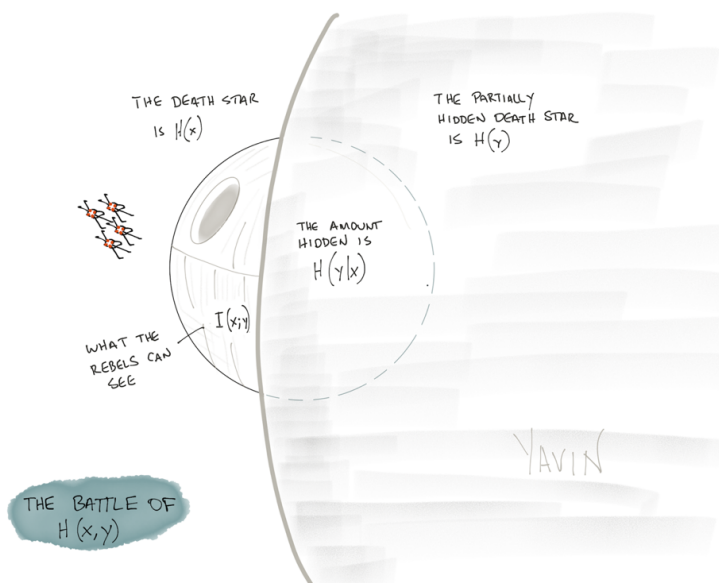
$$H(x) \geq H(y|x)$$

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:

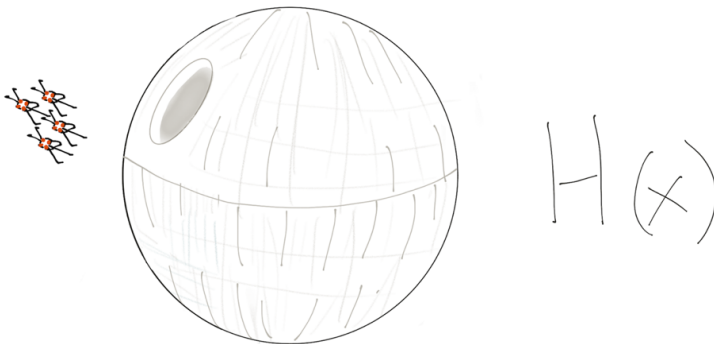


In the final big scene of the original Star Wars (IV), the Rebels were hiding behind Yavin so that the Death Star couldn't blow them up. If you remember the scene, however, they could still track the Death Star, even though they couldn't see it.

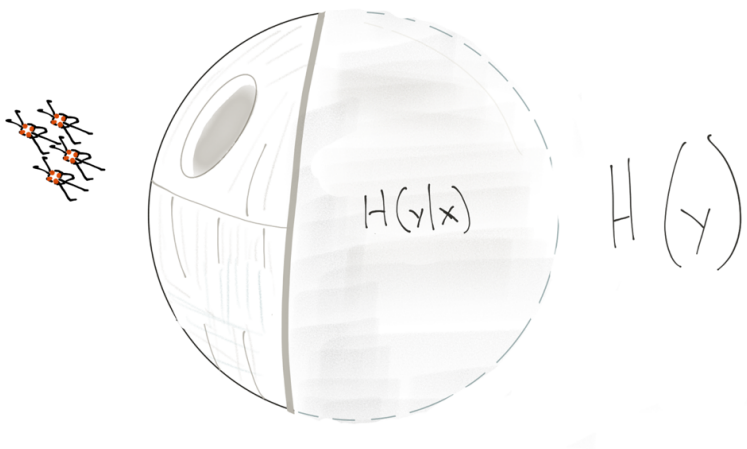
This scene is a good illustration of what we have learned thus far. Yavin is providing some much-needed noise, and all the Rebels and the Empire can see of each other is a ghostly outline.

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 effi-



ciency. 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 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 somewhat 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

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.

The code for this chapter is up on [GitHub](#).

## POSSIBLE INTERVIEW QUESTIONS

- What is a *checksum* and how is it used?
- Calculate the *Hamming Distance* between these two code-words.
- Does this 7-bit binary string have an error?

# IN CONVERSATION

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.

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.

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:

1. Forgot \_\_\_\_\_, please bring \_\_\_\_\_.
2. 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 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. Subsequently, 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" => "O",
  "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. We'll start with our class definition, encoding property and validation bits <sup>3</sup>:

---

<sup>3</sup> *This code is available in the downloads for this book as a single class file*

```

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
end

```

Next we'll add methods to parse the message into code words and finally a decoder:

```

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

```

Great! Now that we've done that we can run it:

```

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 | 1001

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 QUIET 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**

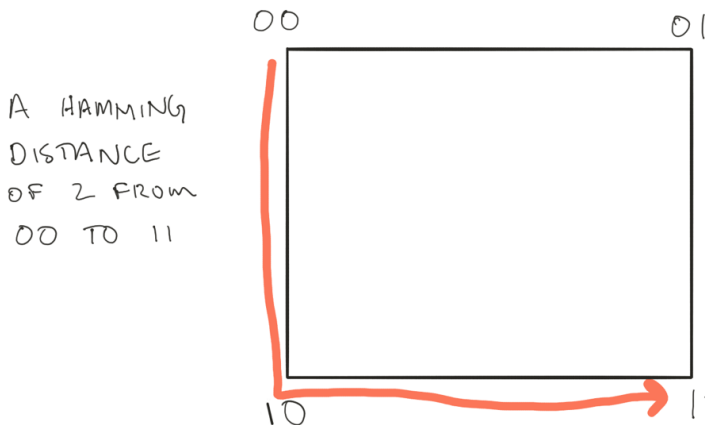
The 1 on the far left differs from the valid code word below by a single bit. If 2 bits had flipped, we would have ( $d=2$ ) between the two:

**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 resolve 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 lunchtime. 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.

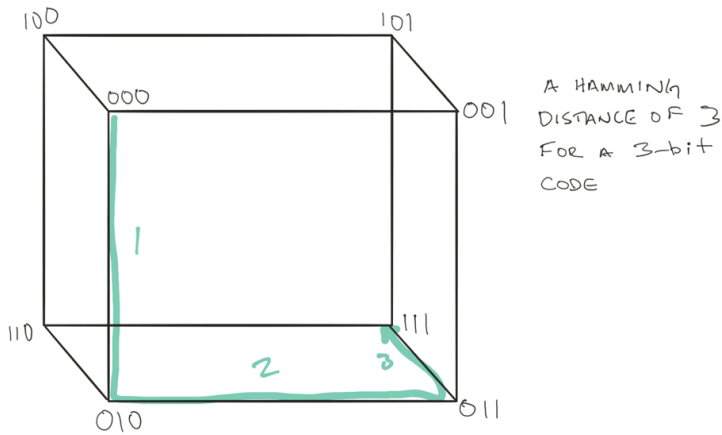
## **Using Codes That Allow for Error Correction**

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 code-word. 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 msg
    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:

1. Isolate the "bad" codeword
2. Calculate the Hamming distance against our encoding scheme using a "best guess" scenario
3. 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
end

```

To replace the code word in error, I simply find the code word in our encryption scheme that has the shortest Hamming distance. I've added this code to my Lovebug class, which you can view in its entirety in the downloads for this chapter.

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 could 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 code-words?

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 | 000100**1**01**

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* mean 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:  $(0 \wedge 0 \wedge 0 = 0)$  and  $(0 \wedge 1 \wedge 0 = 1)$ . 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 ( $(2 \wedge 0 = 1)$ ), the second position ( $(2 \wedge 1 = 2)$ ) and the fourth position ( $(2 \wedge 2 = 4)$ ). 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:

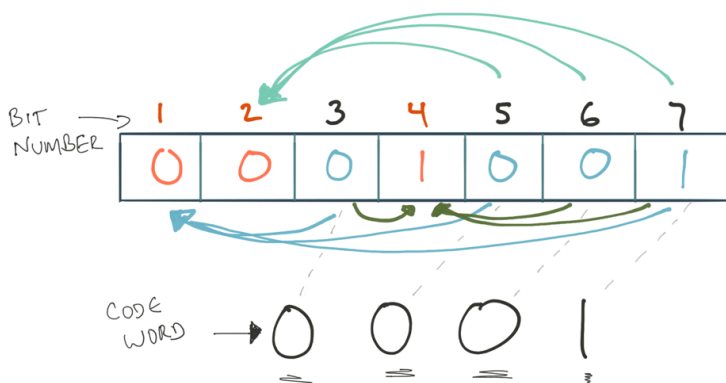
$$D1 \oplus D2 \oplus D4 = P1$$

$$D2 \oplus D3 \oplus D4 = P2$$

$$D1 \oplus D3 \oplus D4 = P3$$

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 occupying the 4th position in our encoding.

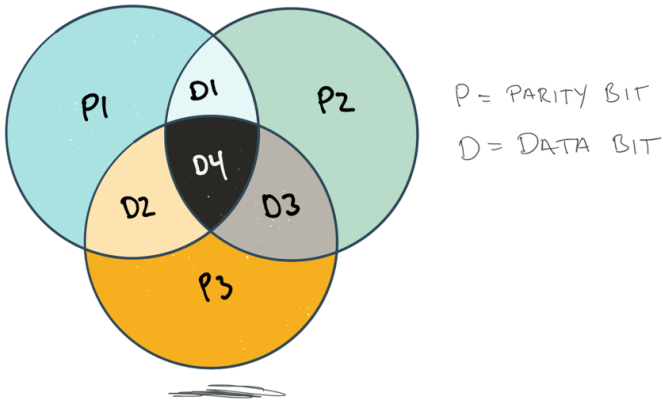
This can all be perplexing. Let's see if we can clear things up with some pictures. Here's a first go:



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:

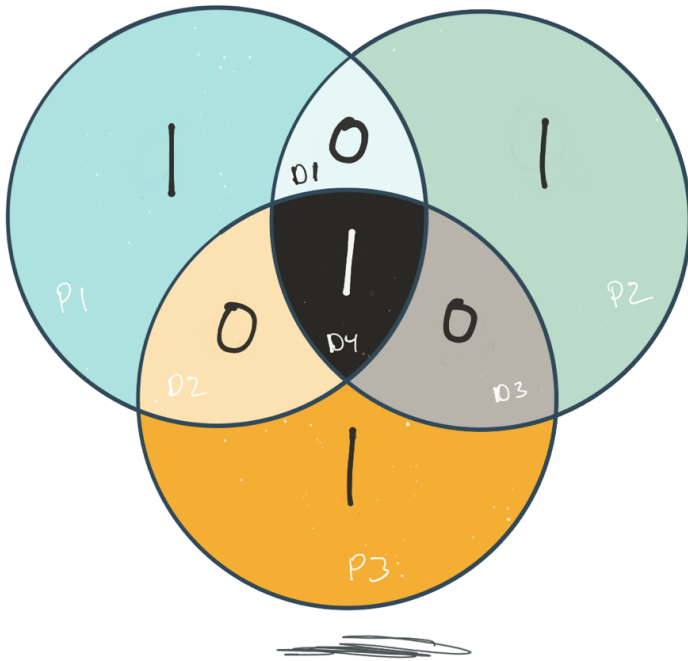
# HAMMING 7,4



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:



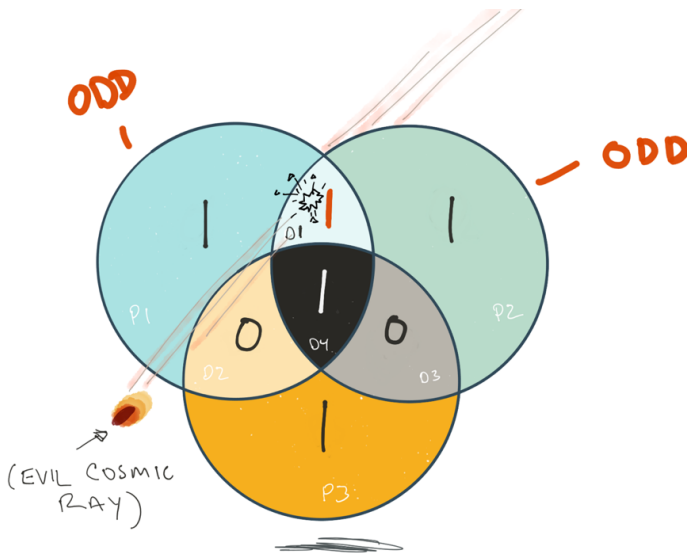
$$D1 \oplus D2 \oplus D4 = 0 \oplus 0 \oplus 1 = 1$$

$$D2 \oplus D3 \oplus D4 = 0 \oplus 0 \oplus 1 = 1$$

$$D1 \oplus D3 \oplus D4 = 0 \oplus 0 \oplus 1 = 1$$

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

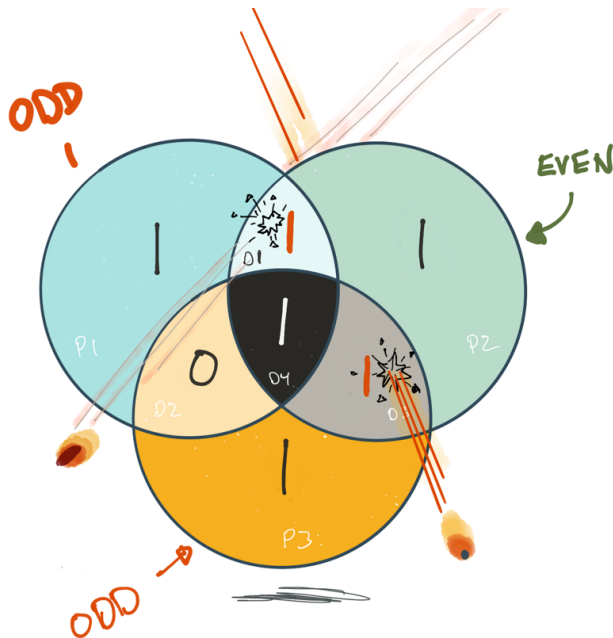
Handling a single error with a Hamming (7,4) code is straightforward, but what happens when there are multiple errors? Transmission errors due to noise aren't typically random, and they tend to follow the same patterns that Shannon described for other bits of information.

Consider your Bluetooth earphones: you're walking around the house, cleaning things on the weekend and listening to an audiobook 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.

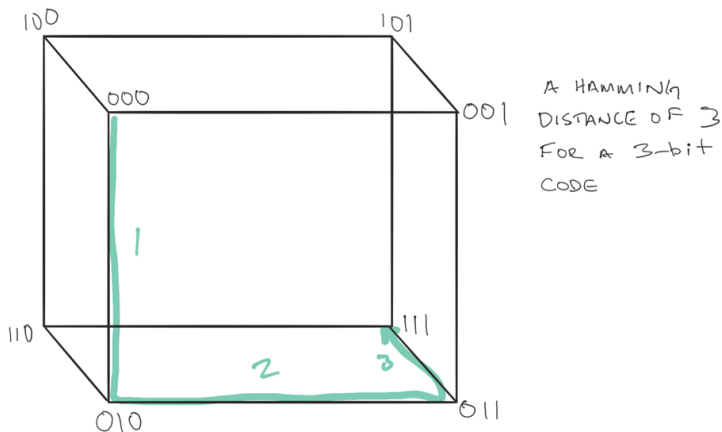
The point is: errors typically come in packs, and this is a problem if we're using a Hamming (7,4) code. To see this, suppose that we have two errors instead of one: D1 and D3 have both flipped:



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 can define 40,000 or so of those words. Again: this is a loaded topic and these numbers can vary by region, education, and a 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 "hmmmpf" are substituted randomly for actual words and it's up to me to decode them properly. The sound "hhhmpf", 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

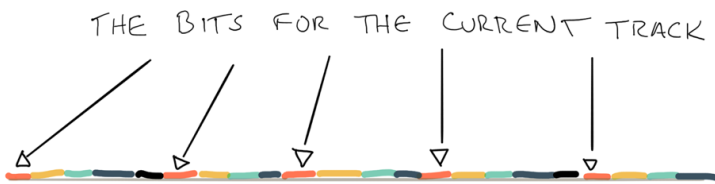
Modern encoding is quite a bit more complex than our Hamming (7,4) code. We transmit bits in the billions, and at very high speeds. To protect against the inevitable noise, we need to bring out a bigger algorithm.

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 uploader 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

**E**ncryption 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?

## IN CONVERSATION

Encryption is one of those subjects that comes up often, usually in conversations surrounding a breach or lack of security on a website. 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 more profound 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.

## INTRODUCTION

Being a programmer, you likely know what encryption is and, perhaps, a little about how it works. As for me, I know just about *nothing* regarding encryption, aside from the names of some algorithms and that passwords should be hashed instead.

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 basic encryption, where you have a single key that both encrypts and then decrypts a message. This is the way it was done for millennia until the 20th century and the discovery of...
- **Asymmetric Encryption.** This is encryption that involves *two keys*, one that's public, which is used to encrypt and one that is private, which is used to decrypt a message which starts life 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 many **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:

1. The raw, unencrypted text or ***plaintext***.
2. The **cipher** algorithm you're going to use.
3. 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 algo-



rithm (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.

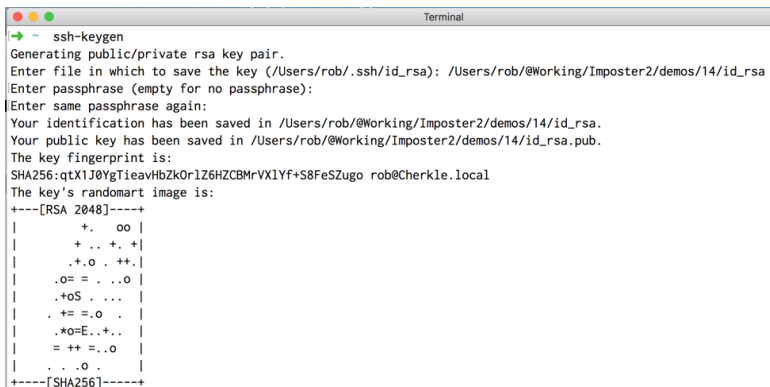
Let's go through each step of generating an RSA key, and then dive into the interesting bits. Since you're not sitting next to me, as much as I am trying to pretend you are, I'll use my ignorance as a guide. Please don't hesitate to skip over whatever you know al-

ready, 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:qtX1J0YgTieavHbZk0r1Z6HZCBMrVX1Yf+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:

```
Terminal
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 kr1_file [-u] [-s ca_public] [-z version_number]
                file ...
    ssh-keygen -Q -f kr1_file file ...
```

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](https://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 MOD-ULI 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.

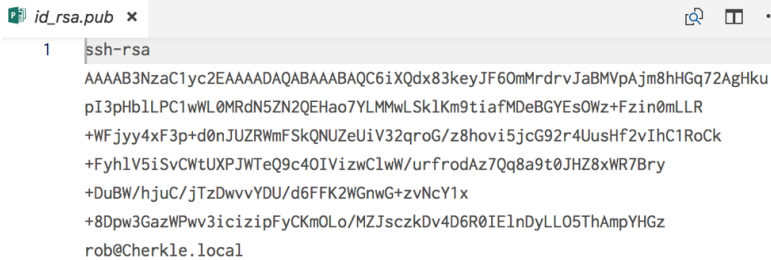
OK – let's have a look at what was generated.

## 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 x
1 -----BEGIN RSA PRIVATE KEY-----
2 MIIEpAIBAAKCAQEAAo10HcfN5HsiRejpjK3a7yWgTFaQI5vIRxqu9gIB5LqSN6R2
3 5SzwtcFi9DEXTEWtdkBB2q02CzDMC0pJSpvbYmnzA3gRmBLD1s/hc4p9Jiy0f1hY
4 8suMRd6fndJyVGUVphUpEDVGX1Ild9qq6Bv8/IaL4uY3Bvdq+FLrB39ryIqtUaAp
5 PhcoZVeYkrwlrVFzyVk3kPXODiFYs8ApcFv7q366HQM+0KvGvbdCR2fMVkewa8vg
6 7gVv4Y7gv408w8L72A1P3ehRSt1hp8Bvs7zXGNcfvA6cnXms1j8L94nIs4qRcgip
7 ji6PzGSbHM5A7+A+kdcBJZw8iyzuU4QJqWBxswIDAQBAoIBABacYr7ZYV04uUbQ
8 HL84s7kZTnzyYTZvw68ax0p9A82978te3Cqbb/IYJu6CpNiEY40J9nGSNpbUq2cS
9 8h9hFQ6w2QwqmbddTugQ+rsOpmziL17CRsXndnFhsudmgvWH23/uQLP+QTFds7z7
10 fIv2GIxjosE2NqVziFXhdyvJfj8n0TW6MHnyw4Yf/0vyJDoWJtGxiTd0zKzy8ebz
11 rZwQgbm8r+TRn45vImq2wx3uxnx7ng0dof31EGVvKxZ3f63Sd6nNLjrgUQAuJ5
12 vAumyjk8i2AJlyEFrbRnUyX9Pi5sEQnjaNuXmwfC0dkYZ5UPNSWCey3bpBvQCQJH
13 cs4F1kECgYEA8f1AoB7Lid9eupHofZKo3JzWok38NuVFAMTGGGq2is6u8cUkVoQw
14 spyJYNzXXIGdI77U85zZia8QfyqvBubNALk2/JQcBxrDBISywmEvf7ZHpbvOnX7+
15 PHnPoxYP8vBeGhmGPU14q42313/QQB54NneznD7mGcqeoDpESaDX070CgYEAxVmN
16 e5VD92/x/VQNJDROt2eJMT0SE+F1oDXzeXD3fo/Fa6VGm2yy7xr0YHefZ0ToJsYf
17 mASMTc5QwCmcJIutX1S51ANmr3ssBTLA/ya9fYoLVPzAaJsUA0wgzNDd/KKiTn+2
18 qzB0Mh9Kvxq6b8F2I8r6LXyItIHv248NDnzJAi8CgYEA00ySR4C8wnfBE/DH8af5
19 NzTqJK8u+Iz7MILsbXP6VXoowM0jbz1d/QrLvRzwH0K8AvPop5cnS5kJMdMHJmKz
20 T9dtEeEQHJAdJwGkFuPgFmfDTVsNpIn1o2UZ21effu5j2vHco7NOhm2GUPuiZ9R
21 4FZ1DKLMcPv+qvJWoWkHqo0CgYAlP9v2oZvsj1KssDqyxe7Sq1Y31TYTTPD3JnfI
22 SkxMtTjC0Rv0pjW37+yEcM/DnDv9ZZP9C463+ONBAhmYxEx7G/DfVa0CHkuWx5yS
23 YGuP0IiHCByBKJREAE3ImvvrF8XJNg3yjfjApStkmtTp1yhHZT6bDKB5AbWWSY/7
24 Vg5SVqKBgQCud7BmVFx+gdsJdTp+1q7r2zVCBDDiaJqDn5FLWB5pJRSmGP1GN+80
25 tVMeh5qbJ40Nn6+ezP5rXELazyRu1Xht9o05gGTqcekBiSUzU07C4oL1qL7vzSNq
26 4d72wjDaP5gxMyxNRD8G8L5THJB0z1m+zRSZ7AbbES+2D180PsZt4A==
27 -----END RSA PRIVATE KEY-----
```

This is the public key:

A screenshot of a terminal window with a single tab titled 'id\_rsa.pub'. The terminal shows the command 'ssh-rsa' followed by a long, multi-line string of alphanumeric characters representing a public key. The string ends with 'rob@Cherkle.local'.

```
1 ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQAC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72AgHku
pI3pHb1LPC1wWL0MRdN5ZN2QEHa07YLMmWLSk1Km9tiafMDeBGYEs0Wz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+Fyh1V5iSvCWtUXPJWTeQ9c40IVizwC1wW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnG+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 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).

## **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 to 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!



*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 many 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 could 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.



*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 256 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 inter-

net. 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.

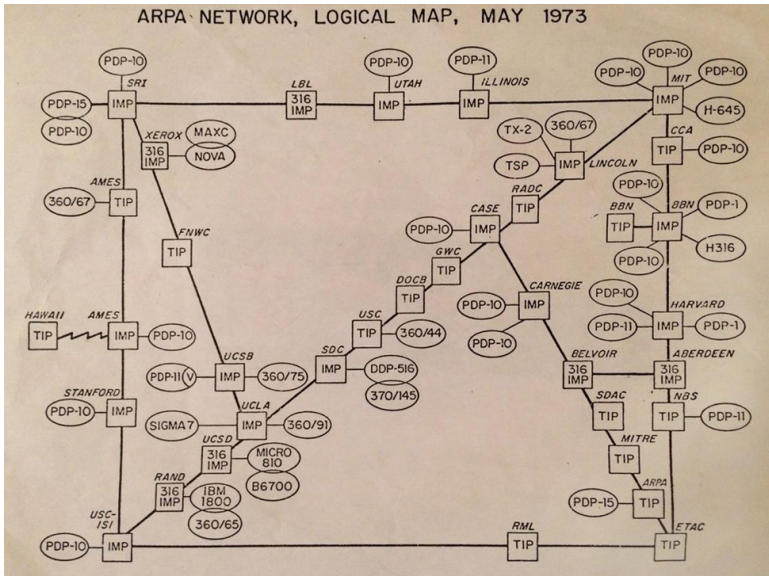
Intelligence services around the world were simultaneously excited and apprehensive about this. Improved encryption was an obvious asset for all manner of things best encrypted. But if your improved encryption were to fall into enemy hands, you were in for a world of hurt.

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:



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 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 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

**W**hitfield 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 achieve this, the three used one-way functions and modular mathematics, a niche scientific field that cryptographers love.

You'll hear Whitfield Diffie and Martin Hellman often name-dropped, 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 calcu-

lates mod (%) differently from the way JavaScript, C# and Go do is a Big Deal!

## INTRODUCTION

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" now might not, in fact, be long at all in 10 or 20 years when machines think for themselves and Skynet can 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 trap-door 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 somewhat 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 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

7.3000000000000001

This works nicely and confirms our calculation <sup>4</sup>. 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 <sup>y</sup>	0	.	=	+

<sup>4</sup> 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 =

7.299999999999

Rad		x!	(	)	%	AC
Inv	sin	ln	7	8	9	÷
π	cos	log	4	5	6	×
e	tan	√	1	2	3	−
Ans	EXP	x <sup>y</sup>	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 a 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:

Search: **19.3 % -12** = 7.3

TOP HIT

**7.3**

DEVELOPER

- adapter.js
- adapter.js

FOLDERS

- 2017-07-03

MOVIES

- IMG\_1663.MOV

OTHER

- LICENSE
- LICENSE

DOCUMENTS

- Ludo\_Support.lua

BOOKMARKS & HISTORY

- 19.3 % -12 - Bing — <https://www.bi...>

MAIL & MESSAGES


19.3 % -12 =

**7.3**

7.3 once again. Let's cross-reference that with Google:

19.3 modulo (-12) =

**-4.7**


Rad  x! ( ) % AC

Inv sin ln 7 8 9 ÷

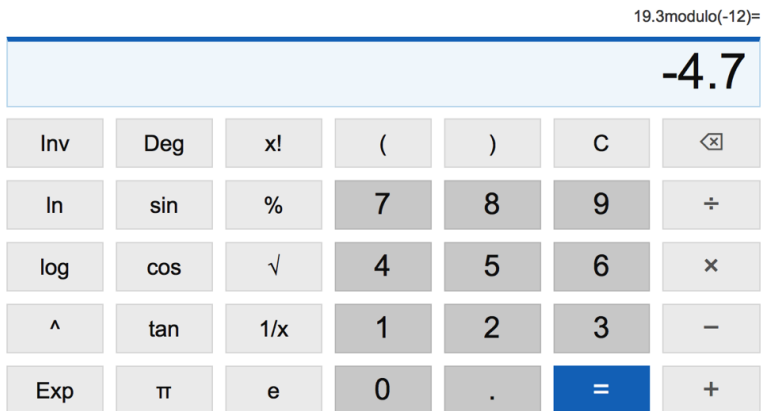
π cos log 4 5 6 ×

e tan √ 1 2 3 −

Ans EXP x<sup>y</sup> 0 . = +



This answer is different! OK, let's get a 3rd opinion and ask Bing:



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 x

```
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 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.

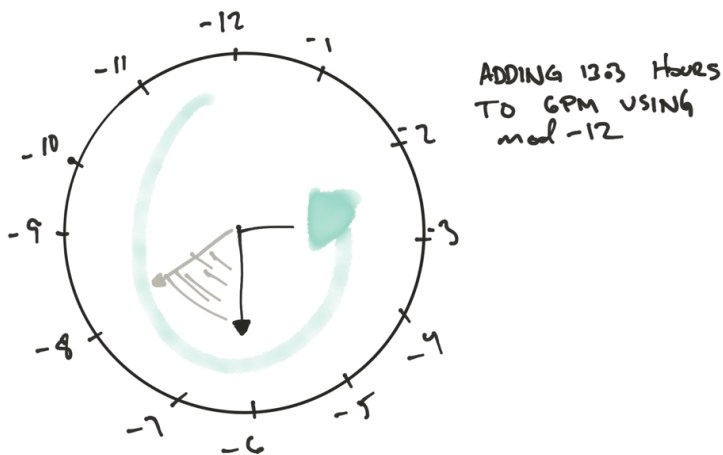
Now let's do the same thing, but with a -12:

$$19.3 = -12 * -1 + 7.3$$

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 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 <sup>5</sup>:

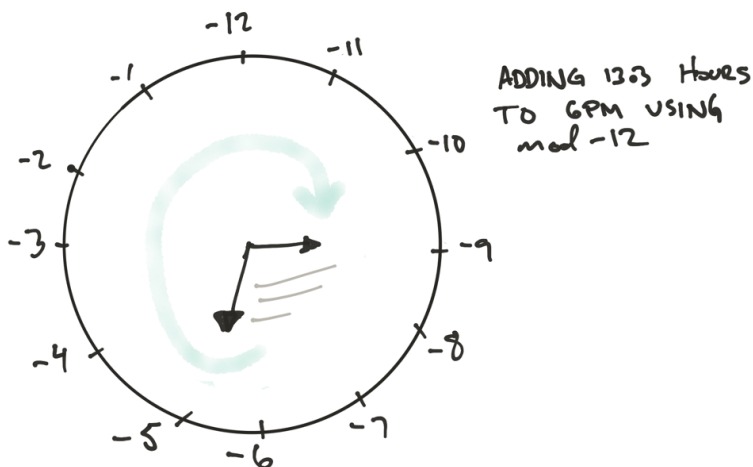
---

<sup>5</sup> *A note to math fans: yes, the 12 on my clock is actually 0 but since it's a clock I'm leaving it as 1. I think this can be confusing and I'm sorry for that. And for clocks in general.*



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 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 ✕

```
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);
16
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
7.300000000000001
-7.300000000000001
7.300000000000001
```

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:

1. Using the `%` operator
2. 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 x

```
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
	7.3000000000000001		
	4.699999999999999		
	-4.699999999999999		
	-7.3000000000000001		

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 x

```
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
----------	--------	---------------	----------

---

	7.3000000000000001		
	-7.3000000000000001		
	-4.6999999999999999		
	4.6999999999999999		

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.

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 }
16
```

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  $ga$ . 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 could 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) => {
  return 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 = 96654654656213684651354954621321654968562132165465654954654654684654546546666546546546546543213332131;
4

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
2.964474462213553e+47
8.788108837116334e+94
6.412747552141642e+106
6.412747552141642e+106
Runner: node
```

Much better. Still brute-forcible 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

**R**SA 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, the British government immediately classified their work.

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?

## IN CONVERSATION

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.

## INTRODUCTION

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.

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.

Before 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 back-solved?

## 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, a 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:

$$C = M^e(modN)$$

Plugging in all the numbers we get:

$$128540^3 mod(1069763)$$

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 x 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)

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 minuscule, 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:

$$C = M^e(\text{mod}N)$$

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  $(M \text{ (pmod } N))$ ! 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:

$$M = C^d(\text{mod}N)$$

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 cryp-

tography 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 equa-

tion 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.

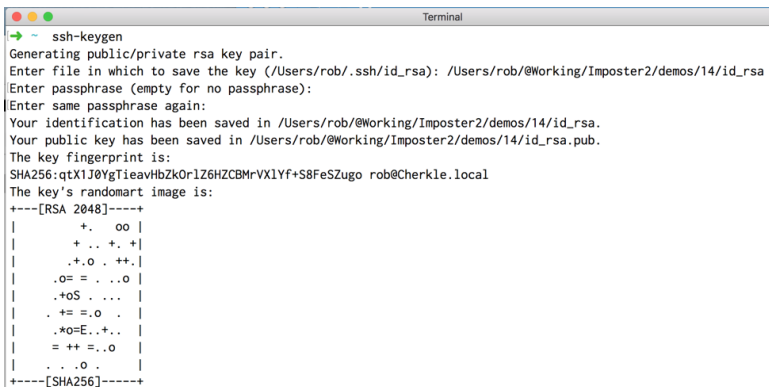
For the purposes of our story, Bob is a bit of a math wizard and knows how to solve this equation, so he plugs in the numbers he used to generate his public key and comes up with the corresponding value for  $d$ .

This allows him to move the *mod N* clock back around to  $M$ , decrypting the message using a simple equation:

$$M = C^d(\text{mod}N)$$

# 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.



```
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:qtX1J0YgTieavHbZkOr1z6HZCBMrVX1Yf+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:

1. 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  $2^{2048}$ .
2. We know why we have **id\_rsa** and **id\_rsa.pub** files: the .pub is our public key, the other file is our private key. We know how they were calculated and why they exist.

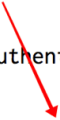
Likewise, we can think a bit more about the key size, 2048. That's a large key and should never be cracked, but 4096 would be much better. Keep in mind that these bits of entropy are logarithmic, so a 4096-bit key isn't just twice as good: it's quite a few orders of magnitude better!

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
    destruction

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:

```
id_rsa.pub x
1 ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQAC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72AgHku
pI3pHb1LPC1wWL0MRdN5ZN2QEHao7YLMmWLSk1Km9tiafMDeBGYEsOWz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRwmFskQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+Fyh1V5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKmOLO/MZJsczkDv4D6R0IElnDyLLO5ThAmpYHGz
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*:



The screenshot shows a terminal window with the file `id_rsa.pub` open. The content of the file is as follows:

```
1  ssh-rsa
   AAAAB3NzaC1yc2EAAAADAQABAAQAC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72Ag
   HkupI3pHb1LPC1wWL0MRdN5ZN2QEHa7YLMmWLSk1Km9tiafMDeBGYEsOWz+Fzin0mLLR
   +WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
   +Fyh1V5iSvCwtUXPJWTeQ9c40IVizwC1wW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
   +DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
   +8Dpw3GazWPwv3icizipFyCKmOLO/MZJsczkDv4D6R0IE1nDyLL05ThAmpYHGz
   rob@Cherkle.local
```

Red arrows and text labels identify the parts of the key:

- Header**: Points to `ssh-rsa`.
- Base64 Encoded Stuff**: Points to the large block of Base64-encoded data.
- Comment**: Points to `rob@Cherkle.local`.

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
```

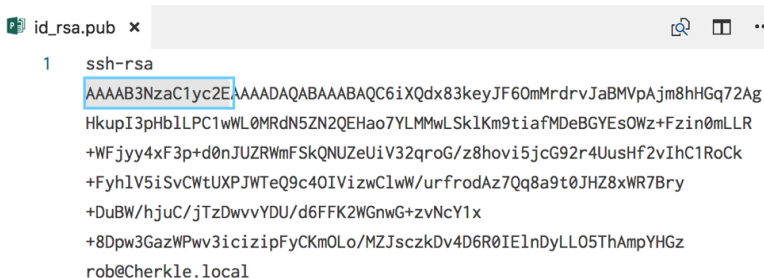
The value for 'rsa\_signature\_blob' is encoded as a string containing s (which is an integer, without lengths or padding, unsigned, and in network byte order).

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:



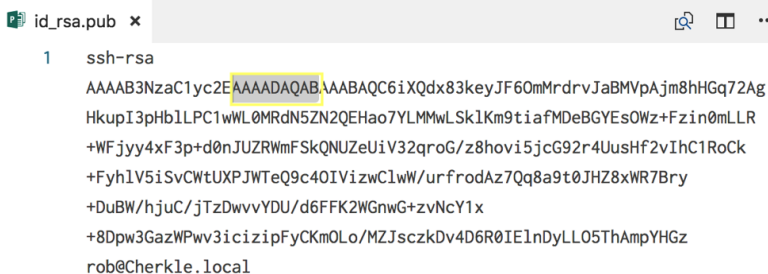
```
id_rsa.pub x
1  ssh-rsa
   AAAAB3NzaC1yc2EAAAADAQABAAQAC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72Ag
   HkupI3pHb1LPC1wWL0MRdN5ZN2QEHao7YLLMMwLSk1Km9tiafMDeBGYEs0Wz+Fzin0mLLR
   +WFjyy4xF3p+d0nJUZRWmFskQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
   +Fyh1V5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
   +DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
   +8Dpw3GazWPwv3icizipFyCKmOLO/MZJsczkDv4D6R0IE1nDyLL05ThAmpYHGz
   rob@Cherkle.local
```

Here's an excellent 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  $\phi n$ . That's an entirely different 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:

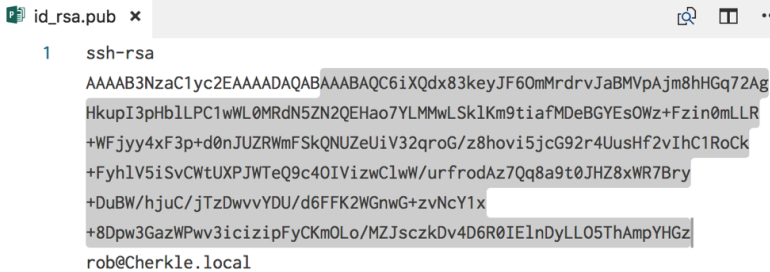


```
id_rsa.pub x
1 ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQAC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72Ag
HkupI3pHb1LPC1wWL0MRdN5ZN2QEHao7YLMmWLSk1Km9tiafMDeBGYEsOWz+Fzin0mLLR
+WFjyy4xF3p+d0nJUZRWmFSkQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
+Fyh1V5iSvCWtUXPJWTeQ9c40IVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
+DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnG+zvNcY1x
+8Dpw3GazWPwv3icizipFyCKmOLO/MZJsczkDv4D6R0IE1nDyLL05ThAmpYHGz
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:



```
id_rsa.pub x
1  ssh-rsa
   AAAAB3NzaC1yc2EAAAADAQABAAQCAQC6iXQdx83keyJF60mMrdrvJaBMVpAjm8hHGq72Ag
   HkupI3pHb1LPC1wWL0MRdN5ZN2QEHa07YLMmWLSk1Km9tiafMDeBGYES0Wz+Fzin0mLLR
   +WFjyy4xF3p+d0nJUZRwmFskQNUZeUiV32qroG/z8hovi5jcG92r4UusHf2vIhC1RoCk
   +Fyh1V5iSvCWtUXPJWTeQ9c4OIVizwClwW/urfrodAz7Qq8a9t0JHZ8xWR7Bry
   +DuBW/hjuC/jTzDwvvYDU/d6FFK2WGnwG+zvNcY1x
   +8Dpw3GazWPwv3icizipFyCKmOLO/MZJsczkDv4D6R0IE1nDyLL05ThAmpYHGz
rob@Cherkle.local
```

There you go! That's what's in your public SSH key.

## RSA, ECDSA, ED25519 – WHAT'S THE DIFF?

One of the **ssh-keygen** options we briefly covered is the choice of cipher used <sup>6</sup>:

---

<sup>6</sup> *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.*

## 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 k  
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, theoretically, 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 websites. They're big proponents of ECDSA, as Nick Sullivan once again explains:

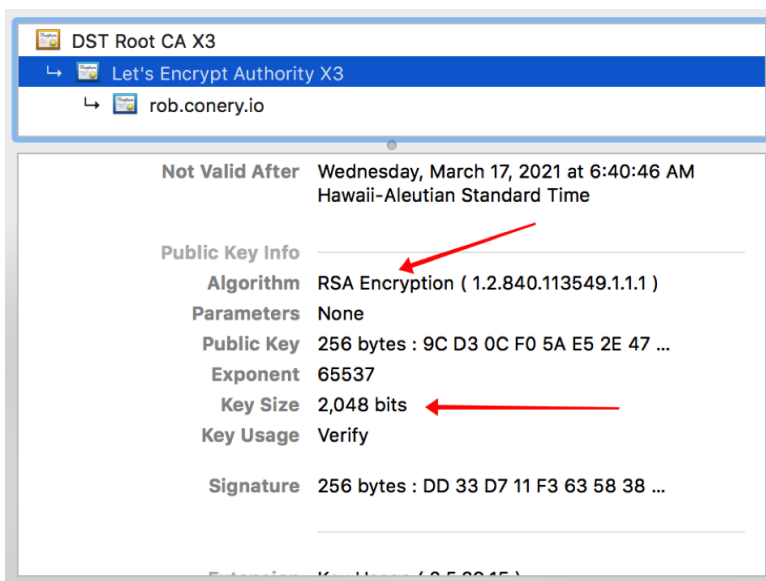
*At CloudFlare we are constantly working on ways to make the Internet better. An important part of this is enabling*



*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:

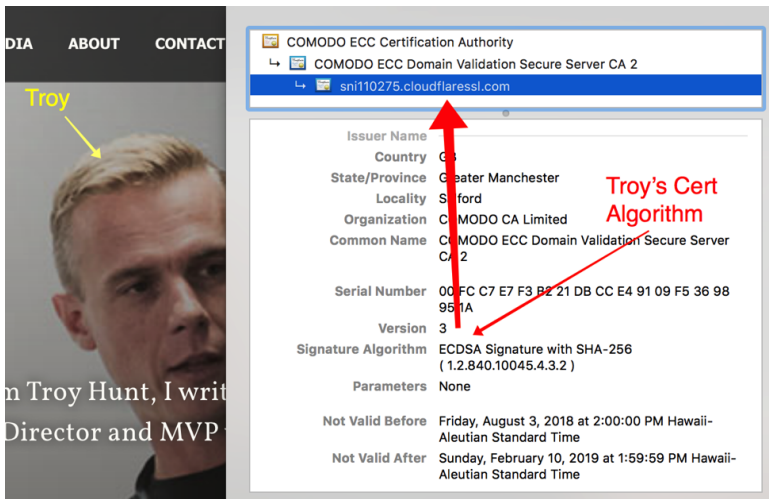


This is the public key used by my server, otherwise known as an *SSL certificate*. You can see that it's been created by Let's Encrypt and that it identifies my blog and some other information about my site. The main thing, however, is the algorithm used and the size of the key: RSA and 2048, respectively.

That means that for your browser to communicate with my server securely, 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 to encrypt information sent to my server.

Ponder that for a second, because I'm going to come back to it in just a bit.

Now, for comparison, here's Troy Hunt's blog, which is also secured with SSL:



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 a wonderful 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.

# MENTAL BREAK – CROWDTESTING CRYPTO WITH RSA 129

Occasionally, 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 I 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-racking 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 to-day, 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 trot out 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 can 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 huge 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 as 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":

**114381625757888867669235779976146612010218296721242362  
562561842935706935245733897830597123563958705058989075  
147599290026879543541**

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 fascinating statement. It didn't take superfast 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 other algorithms and why they are (or *were*, as the case may be) interesting.

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.

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 today.

## 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 it doesn't mean there aren't useful insights for us mere mortals. For instance: some algorithms are *one-way* and create things

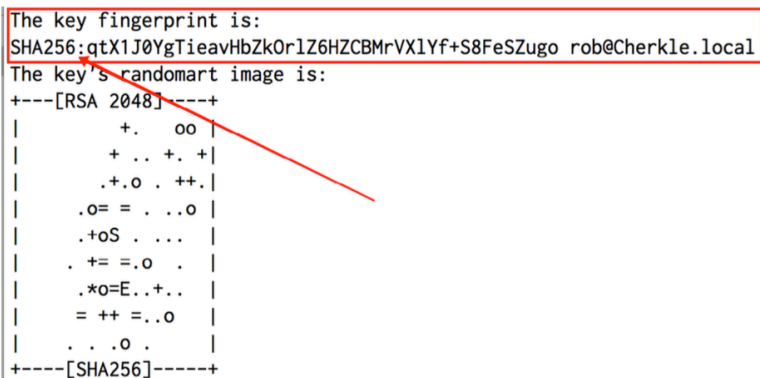
called "hashes." As a programmer, you probably know this already; but if you didn't, now you do.

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:qtX1J0YgTieavHbZkOr1Z6HZCBMrVX1Yf+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 elementary: 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::SHA256.hexdigest("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 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:

1. **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.

2. **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.
3. **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.

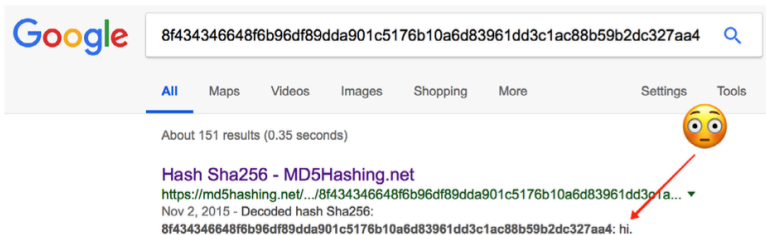
## Trust No One

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:

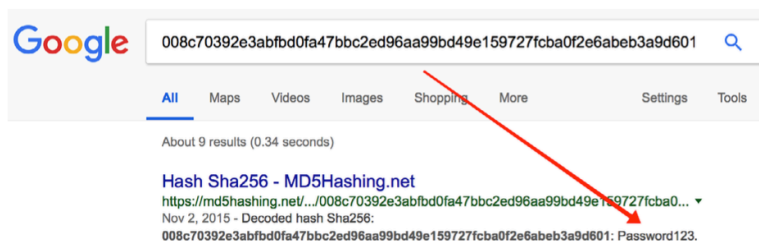
**8f434346648f6b96df89dda901c5176b10a6d83961d-d3c1ac88b59b2dc327aa4**

And plug it into a Google search:



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:



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:

1. **Deterministic**, meaning you get the same output every time for the same input.
2. **One-way** only. No reversing the digest to get at the input!
3. Virtually **collision-free**. It should be essentially impossible for two different strings to create the same digest.
4. **Volatily random**. A small change to the input should result in a drastically different digest.
5. **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 entirely 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 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 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:

1. How does the SHA-2 algorithm work?
2. What's a collision attack?
3. I've heard of rainbow tables before, what are they used for?
4. What's the deal with the randomart thing anyway?
5. 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:

6. Make your boss happy.
7. No extra dependencies.
8. Maybe faster if you can figure out a few tweaks.

Now the cons:

9. You waste hours if not days painstakingly rebuilding something that already exists.
10. You do it wrong after wasting all that time, but only find out weeks later when your first collision happens.
11. You do it right, but slowly, and spend further weeks trying to narrow the performance gap compared to what's already out there.
12. 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...
13. 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 every so often 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 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.

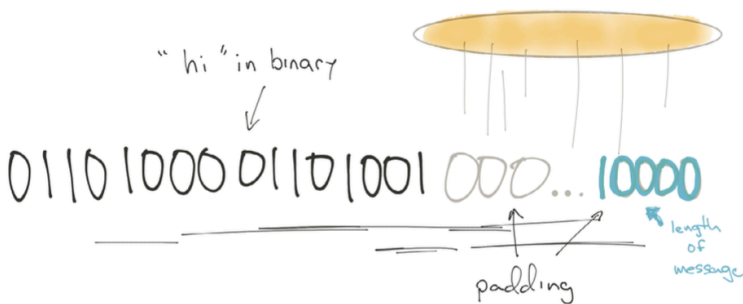
SHA256 creates a 256-bit digest, which is most often seen as a hexadecimal string. Here's the SHA256 for the input "hi":

**8f434346648f6b96df89dda901c5176b10a6d83961d-  
d3c1ac88b59b2dc327aa4**

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

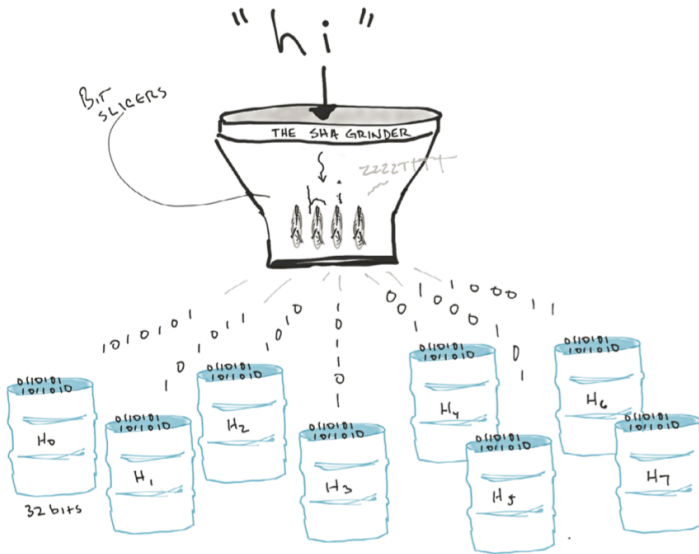
Now that we have a 512-bit string, we need to slice and dice it so that it will fit into the output format, which, remember, is going to be 256 bits long. To create that output string, the SHA-2 algorithm goes through a stepwise scrambling process.

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.

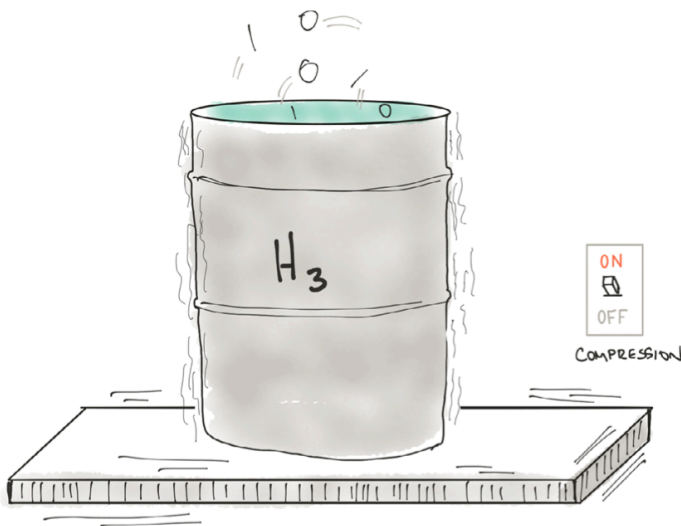
Now comes the fun part: *compression*.

## Step 2: Compression

Our 512-bit chunk of binary-encoded input is now sliced up and fed into the internal state buckets,  $H_0$  through  $H_7$ :



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.

# CRYPTOCURRENCY AND BLOCKCHAIN

**Y**ou 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.

I can't tell you how many times I've been in conversation over the last 2 years when 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.*

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.

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 transac-

tions 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 somewhat devious, and wouldn't mind if some 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 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:

1. We have a distributed set of unknowably devious, power-hungry generals who can only communicate asynchronously via messenger.
2. These generals need to decide on a single course of action: whether to attack or retreat.
3. We can't trust the messages received from the messenger.
4. 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 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. Thirty-two 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 net-*

*work. 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:

1. Memory (RAM)
2. Disk space
3. CPU cycles
4. 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 quite 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, then 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, which is part of the URL:

**0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 3 a 2 a 1 f 8 3-  
fa9b3acd72a17e7cf5583432cde61fa5809e**

... and if you're curious you can watch transactions go through in real time. It's quite 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 min-

ing 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 Block Explorer 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 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 searching for a problem. 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 that 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 huge 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 many 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 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 try-

ing 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 massive 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.

# A BLOCKCHAIN FROM SCRATCH

**L**et's get our hands dirty and see just what this whole thing is all about. I'll do this using JavaScript and Node, but you are welcome 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:

1. Each list element contains data of some sort. Let's pretend we're storing transactions, for now, with "from", "to" and "amount" fields.
2. Each hash key for our list elements must be a SHA256 hash of the element data, the previous hash key, and a nonce.



3. Each hash key must start with some number of 0s, which we'll pass in as an argument called difficulty.
4. Each block must be able to validate itself 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 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.

## 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. We'll start with defining our difficulty using something other than a hardcoded value:

```

//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);
}

```

Now let's add a **mine** method to our Block class:

```

class Block{
  //...
  mine(difficulty=5){
    //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('base64');
      //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}!
          The 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. Please don't hesitate 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) x node step_2.js
Found it: 00YGjGM5Da7C8j1Saun0BNynwB5DpP/AUQZVnSRX6eQ=!
The nonce is 895. It took exactly 0.007 seconds
→ blockchain git:(master) x █
```

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) x node step_2.js
Found it: 00YGjGM5Da7C8j1Saun0BNynwB5DpP/AUQZVnSRX6eQ=!
The nonce is 895. It took exactly 0.007 seconds
→ blockchain git:(master) x node step_2.js
Found it: 00/bwaYm//lW9GsFPcNrunK8L6cV51TXjBQlUNS+1EE=!
The nonce is 9411. It took exactly 0.062 seconds
→ blockchain git:(master) x node step_2.js
Found it: 00Vjbbez//0AIZXhGRiPExmXCXv0pJHsUw4EbjWJIMY=!
The nonce is 2716. It took exactly 0.019 seconds
→ blockchain git:(master) x node step_2.js
Found it: 00EmWoTcavwV15ZYG5krowD2st5lBK8lGYbhzQmtDbU=!
The nonce is 6889. It took exactly 0.046 seconds
→ blockchain git:(master) x node step_2.js
Found it: 00re7lLEzcrik8xxL1KKX8Axq0BJ2CdFfLpUmjxN7Aw=!
The nonce is 4359. It took exactly 0.032 seconds
→ blockchain git:(master) x █
```

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);
```

The result:



```
→ blockchain git:(master) x node step_3.js
Found it: 00kcNWPABTY0Iga2FmwzfT0QreImK58bo4vsylPXVI=!
The nonce is 1785. It took exactly 0.011 seconds
→ blockchain git:(master) x node step_3.js
Found it: 00kcNWPABTY0Iga2FmwzfT0QreImK58bo4vsylPXVI=!
The nonce is 1785. It took exactly 0.01 seconds
→ blockchain git:(master) x node step_3.js
Found it: 00kcNWPABTY0Iga2FmwzfT0QreImK58bo4vsylPXVI=!
The nonce is 1785. It took exactly 0.012 seconds
→ blockchain git:(master) x █
```

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 blockchain 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) x node step_4.js
Found it: 000uWjpSNltonyBuQgz8o/MfMlGt7B1AFj9JaWuqzxs=!
The nonce is 40810. It took exactly 0.137 seconds
→ blockchain git:(master) x node step_4.js
Found it: 000uWjpSNltonyBuQgz8o/MfMlGt7B1AFj9JaWuqzxs=!
The nonce is 40810. It took exactly 0.123 seconds
→ blockchain git:(master) x node step_4.js
Found it: 000uWjpSNltonyBuQgz8o/MfMlGt7B1AFj9JaWuqzxs=!
The nonce is 40810. It took exactly 0.123 seconds
→ blockchain git:(master) x █

```

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) x node step_4.js
Found it: 0000MCnrFr2XjHIAEJSb6JGRe+adT2J63NogDabyFZ4=!
The nonce is 3446324. It took exactly 9.265 seconds
→ blockchain git:(master) x node step_4.js
Found it: 0000MCnrFr2XjHIAEJSb6JGRe+adT2J63NogDabyFZ4=!
The nonce is 3446324. It took exactly 9.627 seconds
→ blockchain git:(master) x █

```

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:

2. Implement the `isValid` method
3. Set the `previousKey`
4. Make sure we can add more blocks!
5. 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 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:

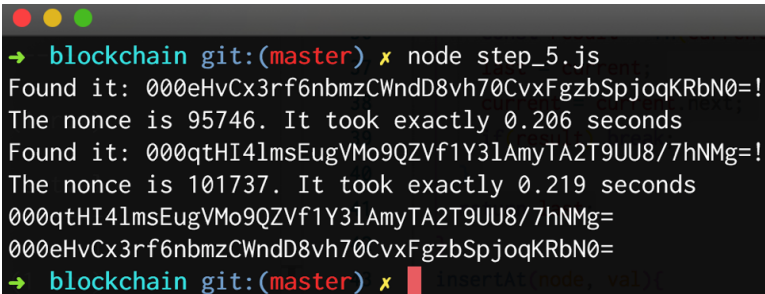
```
const txs1 = [
  {from: "me", to: "you", amount: 12.00},
  {from: "you", to: "me", amount: 5.00}
];

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 terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal shows the command 'blockchain git:(master) x node step\_5.js' being executed. The output consists of two lines of text, each showing a hash, a nonce, and the time taken to find it. The first line shows a hash starting with '000eHvCx3rf6nbmzCWndD8vh70CvxFgzbSpjoqKRbN0=!' and a nonce of 95746, taking 0.206 seconds. The second line shows a hash starting with '000qtHI4lmsEugVMo9QZVf1Y3lAmyTA2T9UU8/7hNMg=!' and a nonce of 101737, taking 0.219 seconds. The terminal ends with the command 'blockchain git:(master) x' followed by a red cursor block.

```
→ blockchain git:(master) x 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) x █ insertAt(node, val)
```

Great! There are many 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.



# COMPILEATION

Compilation is a set of five discrete steps: lexical analysis, parsing, semantic analysis, optimization and code generation. Every compiler does this exact same thing, but some are more capable than others. The main differences between compilers is in the optimization and code generation steps.

A stack is a way to store program data and is a *per thread process*. The stack is very fast and is used for *value types*. The heap is a *per-application process* and is used for *reference types*. You can make your application more efficient by understanding how these two interact.

Garbage collection is the process of freeing up application memory (the heap) automatically. This is not a free process and requires system resources. There are several strategies used for garbage collection (GC), but tracing is the most widely used.

## HOW A COMPILER WORKS

A compiler simply brings different things together and makes them one thing. In a programming sense, a compiler is a program that

reads the code you write and generates something that the run-time engine can execute.

Compilation can happen on command – for instance using a make file. It can happen just in time (JIT) or at runtime with a thing called an interpreter.



They can compile code to different languages (CoffeeScript to JavaScript, e.g) or down to byte code ... or something in between. C#, for instance, compiles to an intermediate language (MSIL), which is then compiled to native code upon execution. This has the advantage of portability – meaning that you could create a compiler for different platforms (Windows 32-bit, 64-bit, Mac, Linux, etc.) without having to change the code.

But how do these things work? Let's look at a high level.

# THE COMPILATION PROCESS

Compilation in a computer is just like compilation in your head when you read these words. You're taking them in through your eyes, separating them using punctuation and white space, and basing the meaning of those words on emphasis.

These words are then turned into meaning in your mind, and at some point sink into your memory ... causing (hopefully) some type of action on your part.

A compiler does the same things, but with slightly larger words. The main compilation steps are:

- Lexical Analysis
- Parsing
- Semantic Analysis
- Optimization
- Code Generation

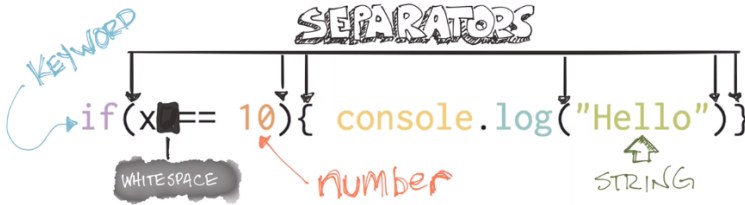
Every compiler goes through these steps.

## **Lexical Analysis**

Lexical Analysis simply analyzes the code that's passed to the compiler and splits it into tokens. When you read this sentence,

you use the whitespace and punctuation between the words to “tokenize” the sentence into words, which you then label.

A compiler will scan a string of code and separate it into tokens as well, labeling the individual elements along the way:



The program within the compiler that does this is called the lexer. So, using our code sample, the lexer will generate these pseudocode tokens (using tuples):

```
{keyword, "if"}
{paren, "("}
{variable, "x"}
{operator, "=="}
{number, "10"}
{left-brace, "{"}
```

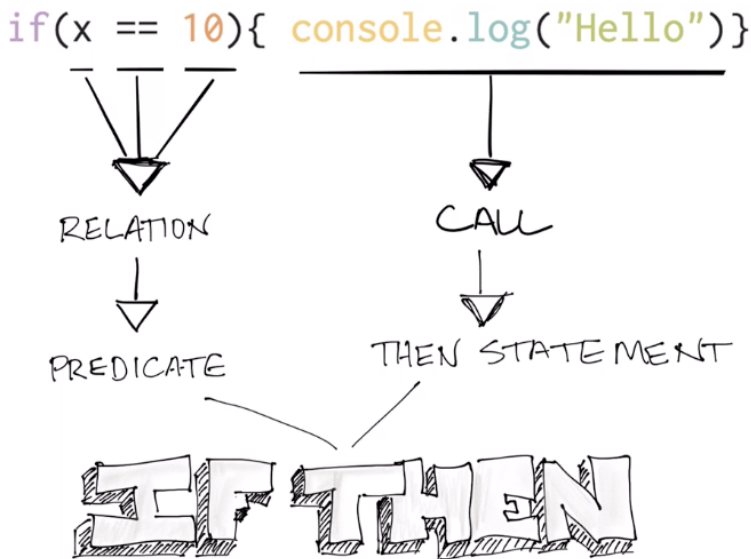
The thing being analyzed by the lexer is the lexeme. A token is what's produced from this analysis. Yay for more random words to know!

## Parsing

After the lexer has tokenized the incoming code string, the parser takes over, applying the tokens to the rules of the language – also known as a *formalized grammar*.

Pushing this back into the realm of written language: the words you're reading now are tokenized using whitespace and punctuation – the next process is to parse their meaning and, essentially, what they're supposed to mean.

A parser analyzes the tokens and figures out the implicit structure:



We know, at this point, that we have an if statement. This is just one line of code, of course, however our entire codebase would be parsed in exactly this way: turning tokens into language structures.

## Semantic Analysis

Now we get to the fun part. Semantic Analysis is where the compiler tries to figure out what you're trying to do. We have an if block for our code, but can the predicates be resolved to truthy/falsey expressions? Do we have more than one else associated with it? Consider this sentence:

*Kim and Jolene want to go to her party*

We can reason that “Kim and Jolene” are the subjects, “to go” is the verb and “party” is the indirect object. But who is her? When a compiler goes through semantic analysis, it must reason through the same thing. For instance:

```
var x = 12;
var squareIt = function(x){
  return x * x;
};
y = squareIt(x);
console.log(y);
```

What will y evaluate to? If you're thinking 144 – you'd be right. Even though we've reused **x** here, the JavaScript interpreter figured out what we meant, even though we tried to confuse it.

```
var x;  
console.log(x);  
x = "Hello!";
```

If you're Brendan Eich (creator of JavaScript) and it's 1995 and you have 10 days to create a language – what choices do you make when it comes to semantic analysis?

I'm sure many of you are thinking “hang on – JavaScript is not a compiled language – it's interpreted” and you'd be correct. They still go through the same steps.

I bring up JavaScript because it's precisely these decisions, made at the semantic analysis level, that have caused developers so much confusion over the years. If you've used the language, you'll know what I mean.

## Lexical Scoping

Most object-oriented languages are lexically scoped, which means the scope of a variable extends to the block of code that contains it. So, in C#, for instance:

```
public class MyClass {  
    public MyClass(){  
        var x = 100;  
    }  
}
```

The scope of **x** in this case is only within the constructor. You cannot access **x** outside it.

Let's change the code so we can rely on lexical scoping to make **x** available to all properties and methods in **MyClass**:

```
public class MyClass {  
    int x;  
    public MyClass(){  
        x = 100;  
    }  
}
```

All we needed to do was to declare the variable within the **MyClass** code block.

JavaScript, however, does things differently. Scopes in JavaScript are defined by function blocks. So, strictly speaking, JavaScript is sort of lexically scoped.

Consider this code:



```
if(1 === 1){  
  var x = 12;  
}  
console.log(x); //12
```

If lexical scoping was enforced here we should see undefined. But what happens in C#? Let's try it:

```
public class ScopeTest  
{  
  [Fact]  
  public void TheScopeOfXIsLexical()  
  {  
    if(1 == 1){  
      var x = 12;  
    }  
    Console.WriteLine(x);  
  }  
}
```

If I run this test, I get the expected response:

**error CS0103: The name 'x' does not exist in the current context**

The reason for the difference? A different semantic analysis for the C# compiler vs. the JavaScript interpreter.

There are more issues that I'm sure you're aware of: hoisting, default global scope for variables with **var**, confusion about **this**. There are many things written about these behaviors and I don't

need to go into them here. I bring all of “this” up simply to note the choices made by semantic analysis.

## Optimization

Once the compiler understands the code structures that are put in place, it’s time to optimize. Of all the steps in the compilation process, this is typically the longest.

There are almost limitless optimizations that can occur; little tweaks to make your code smaller, faster, and use less memory and power (which is important if you’re writing code for phones).

Let’s optimize that previous sentence: compiler optimization produces faster and more efficient code. Reads the same, doesn’t it? The same meaning, anyway – compilers don’t care about creative expression.

Which is a very important point. Modern languages are leaning more on syntactic niceties and shortcuts (aka “*syntactic sugar*”). This is great for developers like me, who enjoy reading code where the intent is clear. It’s not so great for the optimizer.

Ruby and Elixir are prime examples of this. Several constructs in these languages are optional (parentheses for example), and a compiler must work through various syntactic shortcuts to figure out the instructions. **This takes time.**

Does a **meaningful\_variable\_name** need to be 24 characters long. Not to the compiler, which will often rename the variable to

something shorter. List operations as well – often you’ll see arrays substituted for you, behind the scenes.

$$X = Y * 0 :: X = 0$$

B a-  
sically, any time you see a number multiplied by 0, replace it with 0. Seems to make sense ... but ...

```
public class MultiplyingByZero
{
    [Fact]
    public void UsingNaN()
    {
        double x = Double.NaN;
        Console.WriteLine(x * 0); // NaN
    }
}
```

Yeah that won’t work because **NaN** <> 0. It will work if **x** is an integer, however.

Compiler optimizations are at the center of “what makes a language good” – something we’ll get into more, later on.

## Code Gen

Our code has been parsed, analyzed, and optimized – we’re ready for output! This is one of the simpler steps: *we just need to package it up and off we go.*

## Intermediate Language Output

Several platforms out there will compile code text files into an intermediate structure, which will be further compiled later on. Code that runs in a virtual machine (like Java) will do this, and later compile it down to machine-level code.

C# does this as well, and compiles into an actual second language called IL (or MSIL). People don't typically read or write IL directly (unless you're Jon Skeet) – but it is possible to dig down into it to see what's going on.

When a C# program is run for the first time, the IL is recompiled down to native machine code, which runs quite fast. It's also JIT compiled every time it's run, after that. The term JIT stands for *Just In Time* – which usually means “last possible moment”

## Getting The JITters

When discussing multistep compilation, you'll often hear other developers talk about “the JITter” and what it will do to your code.

In some cases it will output code optimized for an executable that will then run it. Other times it will produce native byte code that runs at the processor level.

## Interpreters

Dynamic languages, like Ruby, JavaScript, or Python, typically use a different approach. I say typically because multiple runtime en-

gines have been created that will produce byte code for you from each of these languages. More on that in a later section.

You can think of these languages as “scripting languages” – in other words, they are scripts that are executed by a runtime engine every time they are invoked.

Ruby, for example, has the MRI (Matz’s Ruby Interpreter) which will read in a Ruby file, do all the things discussed above, then execute things as needed. You have to go through the full set of compiler steps *every time a routine needs to be executed*. This is not as fast as native code, obviously, which has already been analyzed, parsed and optimized.

JavaScript is interpreted on the fly in the same way as earlier versions of Ruby, depending on where you use it. The browser will load in and compile any script files referenced on a page, and will hold the compiled code in memory unless/until you reload the browser.

Node works in the same way: it will compile your source files and hold them in memory until you restart the Node runtime.

## LLVM

One of the biggest names in the compilation space (if not the biggest) is LLVM:

*The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them. The name “LLVM” itself is not an acronym; it is the full name of the project.*

So you can use LLVM “stuff” to build your own compiler. One that you’ve probably heard of is CLANG which is an LLVM-based compiler for C.

## **Native Gem/Module Pain**

You know when you go to install a Node module or Ruby gem and you get some nasty message about native bindings or build tools required? The reason for this is that some of these modules use C libraries that need to be compiled to run. You see this a lot with database drivers, for instance, which want to be superfast.

With Node, you’ll usually see a reference to node-gyp rebuild, which is a module specifically created for compiling native modules.

This can cause headaches, especially when you have other developers working on Windows. The workaround, typically, is to install Visual Studio, referencing C++ build tools during the installation. Installing these things on a Windows Server is one of the most frustrating things about using Node on Windows in production.

# GCC

The GCC project is from GNU:

*The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj,...). GCC was originally written as the compiler for the GNU operating system.*

GCC and LLVM do the same things... sort of. They're both compiler toolchains that also provide compilers for C, C++, Java and Go (among others).

The main differences have been speed (with GCC typically being a bit faster) and licensing. GCC uses the GPL and LLVM uses a license based on MIT/BSD which is a little more permissive.

Apple moved from the GCC to LLVM back in 2010/2011 and it caused a lot of code to break, especially for Rails developers. Those darn native gems!

# GARBAGE COLLECTION

Garbage collection (GC) is the process of cleaning up the heap. It's a facility of managed languages such as C# (and other CLR-based languages), Java (and anything that runs on the JVM) as well as dy-

dynamic languages such as Ruby, Python and JavaScript (and many others).

Some languages, like Objective-C, do not have garbage collection directly. This was a choice Apple made to keep their phones as fast as possible. You can, if you want, implement Automatic Reference Counting (ARC) which is a feature of LLVM.

When you write code in Objective-C, you allocate memory as you need and specify pointers vs value types explicitly. When you're done with the variable, you deallocate it yourself.

## **Correction on ARC**

In previous versions of this book, I made the claim that “most developers don't use ARC”, which is false. I made this claim based on a conversation I had 5 years ago with an iOS developer friend. At that time Objective-C developers needed to know how to manually manage memory, and some didn't trust ARC to work perfectly.

This has changed. Use of ARC is standard these days. A comment from a reader (Adrian Kosmaczew) sums it well:

*... after ARC was introduced in 2010 (I was in the room during the WWDC in San Francisco when they announced it) Apple rebuilt pretty much every single app in Mac OS X with it, and ended up removing the garbage collection (which had been added to Objective-C in Mac OS X Leopard, back in 2007) a few years later. iOS never had the GC for the reasons of performance you mention in your book;*



*that is correct. Having a “mark and sweep” GC was too much to handle for the small CPU of the original iPhones, and as such we devs ended up using “retain” and “release” calls to make sure objects didn’t die before we had a chance to use them.*

*The original Objective-C runtime (originally designed for the NeXTSTEP operating system in 1989) used a simpler GC scheme based in resource counting. What ARC does is to perform a static analysis of the source code and to insert the calls to “retain” and “release” automatically where needed. The final binaries are 100% compatible with older versions of OS X and iOS, because there’s no runtime library needed.*

*Now the Objective-C GC is 100% deprecated, and everybody in the “Apple galaxy” uses ARC these days. There are only two known cases of retain cycles one has to take care of when using it (that is, from a lambda to an object holding it, and from an object to another through strong references) but apart from that, it works beautifully well. Actually, the concept was quite revolutionary, it trades a bit of a longer compilation cycle with binaries that literally never leak.*

There are several things to know about GC:

- It’s not free. Determining what objects and memory are subject to collection requires overhead and can slow things down

- It's undecidable. This is as Turing illustrated with the Halting Problem: it's just not possible to know if a process will complete, therefore it's not possible to know if memory will ever not be needed
- It's the focus of a lot of amazing work. Speeding up GC means speeding up the runtime, so language vendors spend much time on it.

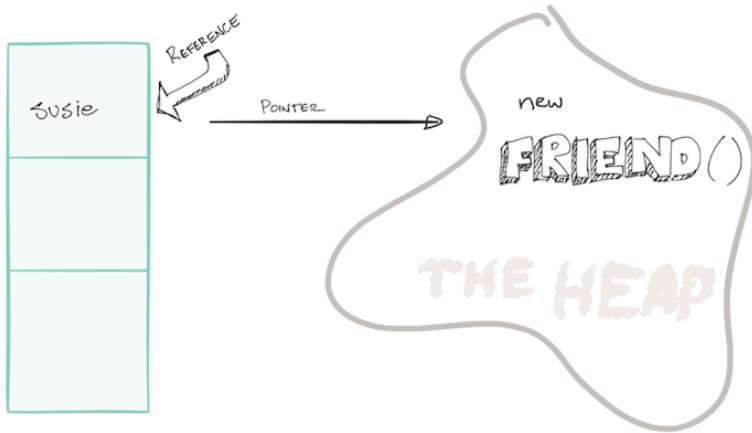
Saying that GC "cleans up the heap" is not going to satisfy the curious, so let's dive into it a bit.

As with all things computer science, there are multiple ways to go about reducing the memory size of the heap and they all center on probability analysis. Some are sophisticated, some are very outdated and buggy.

The most common strategy for GC is tracing, to the point that unless you say otherwise, people will assume this is what you mean.

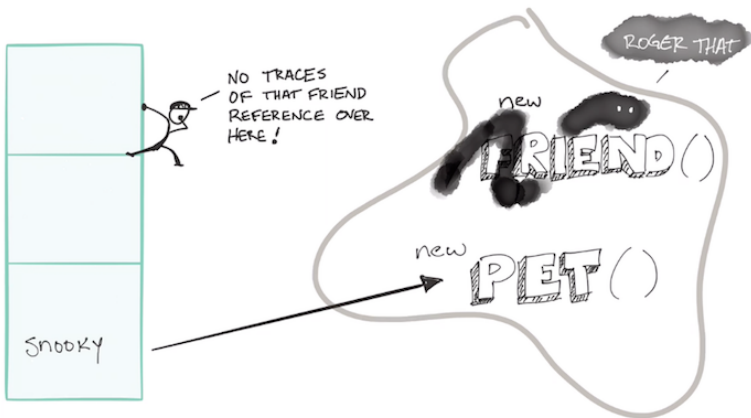
## **Tracing**

Tracing is the most widely used method of garbage collection. When you create an object on the heap, a reference to it exists on the stack, as we've discussed.



As your program runs, the garbage collector will take a look at a set of “root” objects and their references, and trace those references through the stack. Objects on the heap can refer to other objects, so the trace can be much more complex than what is represented here.

As the trace runs, the GC will identify objects that are not traceable – meaning they aren’t reachable by other objects that are traceable. The untraceable objects are then deallocated and the memory freed.



The advantages of tracing are:

- It's accurate. If objects aren't referenced they are targeted for deallocation and that's that.
- It's easy. You just have to find the objects!
- The disadvantages are:
- When will the GC get around to executing?
- What happens when there are multiple threads? The stack for one thread may not have any references, but what about the other threads running?

Obviously, it gets tricky. Within the tracing strategy there are various algorithms for marking and chasing down memory to be cleaned up, with different levels of speed and complexity. Both

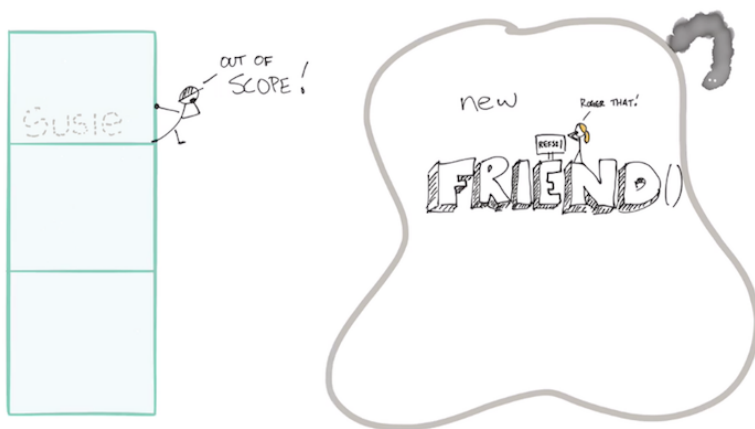
Java and .NET using tracing for GC, both implementing various flavors of generational (or ephemeral) implementations.

This is a deep topic, full of algorithms, probability and statistical analysis ... and I could fill up many pages on the smallest details. Alas I have the rest of the book to write and so I need to clip the discussion here.

If you want to know more, Google away!

## Reference Counting

Reference counting works almost exactly like tracing, but instead of running a trace, an actual count of references is made for each object. When the count goes to 0, the object is ready to be deallocated.



The advantages to reference counting are:

- It's simple. Objects on the heap have a counter which is incremented/decremented based on references to that object (from both the heap and the stack) rather than a trace algorithm.
- There's less guesswork as to "when" GC will happen. When local reference variables fall out of scope, they can be decremented right away. If that count goes to 0 then GC can happen shortly thereafter
- The asymptotic complexity (Big-O) of reference counting is  $O(1)$  for a single object, and  $O(n)$  for an object graph

Sounds simple and obvious, doesn't it? What about this code (pseudocode):

```

public class Customer{
    public int ID {get;set;}
    //...

    public Order CurrentSalesOrder{get;set;}
    public Customer(int id){
        //fetch the current order from the db
        this.CurrentSalesOrder = db.GetCurrentOrder(id);
    }
}

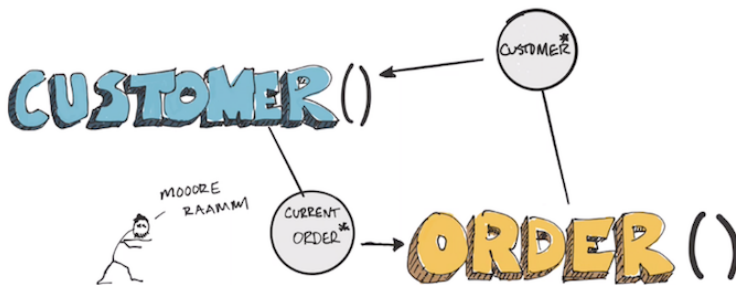
public class Order{
    public int ID {get;set;}
    //...

    public Customer Buyer{get;set;}
    public Order(GUID key){
        this.Customer = db.getCustomerForOrder(key);
    }
}

```

This code is quite common. What we have here is a circular reference on the heap. You could create an instance of the **Customer** and along with it comes a reference to an **Order** ... which has a reference back to the **Customer**.

The reference count for these objects will always be > 1, so they are, in effect, little memory bombs eating up RAM.



## Compile-time

If you have a sophisticated enough compiler, it should be able to analyze what variables need what memory, where, and (possibly) for how long. Without it, Objective-C developers have had to manage memory themselves, allocating and deallocating within the code (much like C).

The XCode compiler analyzes the code written and decides the memory use upfront, freeing the runtime from the overhead of GC. It does this by using the notion of strong, weak, and unowned objects, each with an explicit level of protection that would, again, take me pages to explain properly (along with some goofy drawings).

The essence is this: using strong variables keeps them in memory longer, based on other strong references they're related to. If you have a weak reference to an object it tells the compiler "no need to protect the referenced object for longer than now". Finally with unowned you're making sure that a strong reference won't keep an



object alive for longer than you want – so you explicitly say “make sure this object goes away”.

If you want to know more about this, there’s a great article here that discusses the ideas in terms of the human body:

*A human cannot exist without a heart, and a heart cannot exist without a human. So when I’m creating my Human here, I want to give life to him and give him a Heart. When I initialize this Heart, I have to initialize it with a Human instance. To prevent the retain cycle here that we went over earlier (i.e. so they don’t have strong references to each other), your Human has a strong reference to the Heart and we initialize it with an unowned reference back to Human. This means Heart does not have a strong reference to Human, but Human does have a strong reference to Heart. This will break any future retain cycles that may happen.*

# OBJECT-ORIENTED DESIGN PATTERNS

**P**eople have been writing code in object-oriented languages for a long time and, as you might guess, have figured out common ways to solve common problems. These are called design patterns and there are quite a few of them.

In 1994 a group of programmers got together and started discussing various patterns they had discovered in the code they were writing. In the same way that the Romans created the arch and Brunelleschi created a massive dome – the Gang of Four (or “GoF” as they became known) gave object-oriented programmers a set of blueprints from which to construct their code. The Gang of Four are:

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

# THE CODE

The code for this chapter is, once again, shown in screenshots. If you'd like to play along or have access to the code, you can view it or download it from our GitHub repo. I also have 10 hours of video walking you through the concepts in this book.

# RETHINKING GANG OF FOUR

It's 2021 and I'm revisiting this chapter and redoing a chunk of it. Software has evolved and so have programming languages. A large part of our community no longer conceives of a software program as a singular structure as the result of a group effort. Instead, we're composing elements in smaller pieces and orchestrating them using containers, "pods" and services. The "modern" developer is paying the same price as the classical one when ignoring what's come before.

To that end, *what's come before* must be understood *now*. Only, however, **if it's useful**.

In terms of *usefulness*, I went to the oracle to ask if GoF is still useful. Consider the result:



**Rob Conery**  @robconery · Apr 20

...

Hi friends. Is Gang of Four still important for "modern" applications?  
Apply the term "modern" as you will...

Nope

30.8%

**Always**

**53.3%**

What's Gang of Four?

15.9%

428 votes · Final results



10



3



Tip

Twitter, of course, is Twitter. But if I consider my undertakings over the last 5 years I wonder if there is a reckoning afoot. I think about these patterns a lot, but if I'm honest, I tend to plow ahead with The Way I Do Things (which I'll share with you at the end of the following chapter). This is hard to admit and I feel like many of my experienced friends will be clutching their pearls...

To quote Morrissey: *has the world changed or have I changed?*

In previous incarnations of this book, I opened this chapter with this warning:

***This entire chapter will be argumentative. I hate to say it, but there's just no escaping it: how we build software is still evolving. You might disagree with what you read here, which is fine. I'll do my best to present all sides – but before we get started, please know this: I'm not arguing for or against anything.***

*These concepts exist; you should know they exist. You should know why people like or dislike them and, what is*

*more important, what they argue about when they argue.  
And oh how they do.*

That's not exactly a strong opening. As I read it now, I'm feeling the weight of the hatchet in my hand. The Bridge Pattern, as complex as it is, is simply no longer considered by the bulk of programmers out there. It could occupy a dusty shelf in a programmer's curio shop, I suppose, but that's not what I'm offering with this book. I want to cleave it (and many other patterns) from this book because they're part of a past that I'm not sure is still relevant.

This isn't a good place to be in, as a writer. I *need* to write about history in this book but I also *need* to be sure I take care with what I include, omit and revise. You never know what the future holds in the CS world – I (seriously) just read that Fortran is making a come back. FORTRAN! I don't even know what's real anymore...

Here's the question: I feel like GoF is fading, but should I write about it anyway? Are these things going to be important to you tomorrow, next week, next month?

Who knows, maybe implementing a pattern or two will solve a massive problem you have! Alternatively: you might read this and say "wow Bridge is super cool I'll give this a go" and you write a bunch of code which slows down your app and confuses your team and you get fired. People do weird stuff with Gang of Four.

Let's approach this two ways. The first being, of course, that knowing them is better than not because you might spot one in a code review and be able to ask the person who implemented that pat-

tern why they did why they solved a particular problem that way and then proceed with a thoughtful conversation. The second is that knowing about patterns broadens your mind into how developers in decades past have solved problems (aka “history”) – as long as you know it’s historical. This is always valuable. Knowledge is a good thing, so let’s proceed thus.

I’m simultaneously glad that I know GoF patterns and also sad that I had to live through the Patterning Carousel during the enterprise heyday of Java and .NET. To me, patterns are useful, but if I could impart only a single thing in this brief introduction: **patterns don’t make your code correct.**

You do that. That’s what you’re paid for. No pattern is going to fix your software problems.

– Rob Conery, Hanalei HI, May 2021

## CREATIONAL PATTERNS

When working with an OO language you need to create objects. It’s a simple operation, but sometimes having some rules in place will help create the correct object, with the proper state and context. You’re familiar with this if you’ve used any object-oriented language so I’ll be brisk.

## Constructor

Most OO languages have a built-in way of creating an instance of a class and it typically involves the keyword **new**. I will assume you know how this works if you've written code.

In most languages you can implement a constructor that takes in the parameters it needs to exist:

```
class User {  
  constructor({name, email}){  
    this.name = name;  
    this.email = email;  
  }  
}  
  
//our constructor  
const user = new User({name: "Rob", email: "rob@bigmachine.io"});
```

This is perfectly usable. As with all the GoF patterns, their utility (or lack of) becomes apparent as the program grows. For instance: there's no context here. Our user is being created and returned with a simple parameter assignment. While that's fine for small apps just getting off the ground, more complex apps benefit from something more concrete.

That's where our next pattern comes in. This is also where I will be switching to a more pattern-prone language, C#. If you don't know C#, don't despair! If you know JavaScript or any C-family languages it should be quite familiar.

## Factory

Sometimes instantiating an object can be quite involved, and might require a little more clarity. This is where the Factory Pattern comes in.

For instance: our **Customer** class might have some defaults that we want set:

```
using System;
using System.Collections;
using System.Collections.Generic;

public class Order{}

public class Customer
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public List<Order> Orders { get; set; }

    public static Customer FromDefaults ()
    {
        var customer = new Customer { Status = "unregistered", Name = "Guest" };
        customer.Orders = new List<Order> ();
        return customer;
    }

    public static Customer FromExisting (IDictionary values)
    {
        var customer = new Customer ();
        //populate the values on the class, validating etc.
        return customer;
    }
}
```



Our **Customer** class isn't terribly complex, but you do gain some clarity by calling **Customer.FromDefaults()**. This can become important as your code base grows because it's not clear what's going on if you simply use **new Customer()**.

For complex class construction you could create a dedicated factory class. You see this often in Java. For instance, we could pull the instantiation logic completely out of our **Customer** and into a **CustomerFactory**:

```
public class CustomerFactory
{
    public Customer FromDefaults ()
    {
        var customer = new Customer { Status = "unregistered", Name = "Guest" };
        customer.Orders = new List<Order> ();
        return customer;
    }

    public Customer FromExisting (IDictionary values)
    {
        var customer = new Customer ();
        //populate the values on the class, validating etc.

        return customer;
    }
}

var customerFactory = new CustomerFactory();
var customer = customerFactory.FromDefaults();
```

Again: this is a bit simplistic. You can do a lot more with a factory class, such as deciding which object to create altogether. Your application might have the notion of an **Administrator** that inherits from **Customer**:

```
public class Administrator : Customer {  
    //specific fields/methods for admins  
}
```

You can use a variation of the Factory Pattern (called the Abstract Factory Pattern) to decide whether an **Administrator** should be returned or just a **Customer**:

```

public class AbstractCustomerFactory
{
    public Customer FromDefaults ()
    {
        var customer = new Customer { Status = "unregistered", Name = "Guest" };
        customer.Orders = new List<Order> ();
        return customer;
    }

    public Customer FromExisting (IDictionary values)
    {
        if (values.Contains ("Email")) {
            if (values ["Email"].ToString () == "admin@example.com") {
                var admin = new Administrator ();
                //populate values
                return admin;
            } else {
                var customer = new Customer ();
                //populate the values on the class, validating etc.

                return customer;
            }
        } else {
            return null;
        }
    }
}

var customerFactory = new CustomerFactory();
var customer = customerFactory.FromDefaults();

```

This pattern is useful, but it can spiral on you if you are really into patterns. Consider this question on StackOverflow:

*What is a good name for class which creates factories?  
(FooFactoryFactory sounds silly imo)?*

This happens with C# as well:

*I make extensive use of the interface-based Typed Factory Facility in Windsor, but there are times when I must pass a*

*lot of arguments to a factory around with the factory itself. I'd much prefer to create a factory factory with these arguments so that I don't need to muddy up the constructors of objects more than I need to.*

## Builder

The Factory pattern can only do so much until it becomes too convoluted. This usually happens with very complex objects. Many developers consider this a “code smell” (when you find yourself needing it, it means there’s a simpler way). Overly complex objects are ripe for bugs and, typically, means you’ve probably overthought your solution.

There are times, however, that a Builder makes sense. Consider a class that .NET developers use all the time: **System.Text.StringBuilder**.

Strings are immutable in C#, so if you try to build a string from many string fragments, you can run into the memory problem as seen here:

```

public class NaiveStringBuilder {
    IList<string> _strings;
    public NaiveStringBuilder(){
        _strings = new List<string>();
    }
    public void Append(string val){
        _strings.Add(val);
    }
    public override string ToString(){
        var result = "";
        foreach (var s in _strings) {
            // a new string is built each time
            result = result + s + " ";
        }
        return result;
    }
}

var naiveBuilder = new NaiveStringBuilder();
naiveBuilder.Append("This");
naiveBuilder.Append("could be");
naiveBuilder.Append("very long");
naiveBuilder.Append("and blow up");
naiveBuilder.Append("your program...");
var result = naiveBuilder.ToString(); //BOOM

```

If you ever find yourself writing a string concatenation routine in a loop, stop. It's a memory problem just waiting to happen.

The good news is that the C# team contemplated this and decided to help out, using the Builder pattern with **System.Text.StringBuilder**:

```
var goodBuilder = new System.Text.StringBuilder();
goodBuilder.Append("This ");
goodBuilder.Append("won't ");
goodBuilder.Append("blow up ");
goodBuilder.Append("my program ");
var result = goodBuilder.ToString(); //yay!
```

If you're curious about how the **StringBuilder** works, you can view the source code online. There's a lot going on in there! The thing to take away, however, is that an instance of an object (**System.String**) is being built for us in a very specific way to avoid problems. This is what the Builder Pattern is good for.

There is a more elegant way of doing this, however, which we'll see next.

## Method Chaining

Instead of calling **stringList.Add("...")** or using a **StringBuilder** directly, you can encapsulate what you're doing into a class that uses a fluent interface, otherwise known as Method Chaining:

```

public class Message
{
    System.Text.StringBuilder _stringBuilder;
    public Message (string initialValue)
    {
        _stringBuilder = new System.Text.StringBuilder ();
        _stringBuilder.Append (initialValue);
    }
    public Message Add (string value)
    {
        _stringBuilder.Append (" ");
        _stringBuilder.Append (value);
        return this;
    }
    public override string ToString ()
    {
        return _stringBuilder.ToString ();
    }
}

var message = new Message("Hello")
    .Add("I might be")
    .Add("a really long string")
    .ToString();

//Hello I might be a really long string

```

## Singleton

A Singleton is a class that only allows one instance of itself. It's not an easy thing to do correctly and many blog posts have been written about the perils of Singletons and threading or multiple processes.

You should know the pattern, however. Here's a rather naive one in C#:

```
using System;
namespace Singleton
{
    public class SingleThing
    {
        //single instance holder
        private static SingleThing _instance;
        //disallow calling constructor directly
        protected SingleThing () { }
        //access to the instance
        public static SingleThing Instance ()
        {
            if (_instance == null) {
                _instance = new SingleThing ();
            }
            return _instance;
        }
    }
}
```

The problem with this code is that it will likely work fine *most* of the time. Until it gets used more and the Instance method is called simultaneously, and a nasty collision happens. Or if, more likely, someone decides to use your code in a threaded environment.

Interestingly, in JavaScript land, Node implements Singletons for its moduling system but it's almost by accident. Node is single threaded and when it starts it loads every module into memory, caching it, which creates a Singleton.



Running Node in production, however, usually means spinning up multiple instances of Node to handle the load of your application. I'm sure your Singleton code has considered all the other Singletons running in the other instances, yes? Your database connection, for instance, when executing a transaction would *never* do a row-level lock on data your other Singleton needs!

If you find yourself trying to lean on a Singleton the best advice I can give is, *please stop*. You're going to get it wrong. Everyone does and the realization usually comes in the middle of the night during a frantic phone call at 2am. Yes, I know you *think* you can do it, but I would urge you to consider a different approach. You might be wondering at this point: *what of Node then?*

It's true, you're going to need to deal with multiple instances of your application running in memory, if you've used Node. It's at this point where we discuss database consistency rules, row and table locks and whether you need to think about all of this in your application.

I say you do. We'll get into this in a later chapter but throwing a deadlock on a table is one of the first issues that multi-process (or threaded) apps must deal with. I'm going to ask you to hold tight. We're going to discuss concurrency soon and I'll bring this back up.

# STRUCTURAL PATTERNS

Code needs to have some structure, so we use things like methods, modules, and classes. As code becomes more complex, these modules and classes might also need some structure to reduce confusion and unneeded complexity. That's what we'll look at here.

## **Adapter**

The Adapter Pattern is all about making one interface work with another. You see them used often with data access tools (ORMs) where the abstracted query interface needs to work against different databases, such as PostgreSQL, SQL Server, etc.

For instance, we might create a way of working with a database that we really like, so we'll abstract it into a basic interface of some kind:

```

public abstract class GroovyQuery{
    //groovy interface
    //find
    //fetch
    //save
}

public class GroovyPostgresAdapter: GroovyQuery{
    //implements groovy interface for PostgreSQL
}

public class GroovySQLServerAdapter: GroovyQuery{
    //implements groovy interface for SQL Server
}

```

You just need to pick the correct adapter for the database you're working with. This can be done manually or by way of configuration and some higher-level patterns which we'll see later on.

## Bridge

The Bridge Pattern is quite subtle and tends to look a lot like the Adapter Pattern, but it's one step up the abstraction curve. You use the Bridge Pattern when your abstraction gets complicated enough that you need to split things out.

People really like our **GroovyQuery** tool and we want to add a feature: document queries. It turns out that you can store JSON happily in PostgreSQL and also in SQL Server – so we decide to implement a document API that handles parsing and so on:

```

public abstract class GroovyQuery{
    //groovy interface
    //Find
    //Fetch
    //Save
    public abstract T GetDocument<T>();
    public abstract T SaveDocument<T>();
    public abstract IList<T> FetchDocuments<T>();
    //etc
    //etc
}

```

This is a very interesting idea! The problem is that we now have to go and implement it for every adapter. Unless we abstract the document interface and bridge it to our **GroovyQuery**:

```

public abstract class GroovyQuery{
    //groovy interface
    //Find
    //Fetch
    //Save
    public IDocumentQueryable Documents();
    //etc
}

//a document query interface
public interface IDocumentQueryable{
    T Get<T>();
    T Save<T>();
    IList<T> Fetch<T>();
}

```

Here's how we might implement it:

```

//implementation of the document query interface for
//relational systems.
public class RelationalDocumentQueryable : IDocumentQueryable{
    GroovyQuery _adapter;
    public RelationalDocumentQueryable(GroovyQuery adapter){
        this._adapter = adapter;
    }
    //Implement Get, Save, Fetch
}

//our SQL Server adapter
public class GroovySQLServerAdapter: GroovyQuery{
    public GroovySQLServerAdapter(){
        this.Documents = new RelationalDocumentQueryable(this);
    }
    //implement Get, Save, Fetch
}

```

The neat thing about this new structure is we can change our **IDocumentQueryable** interface and the implementation, without breaking any of our adapters.

## Composite

The Composite Pattern deals with parent-child relationships that are composed to create a whole object. They can grow and shrink dynamically, and child objects can move between parents.

Our **GroovyQuery** tool is really picking up steam! People are really happy with it, mostly because we have a cool document abstraction, they can use next to your typical ORM interface. The problem is we need more speed!

It turns out that some of the drivers we've been using don't implement connection pools – basically a set of 10 or so open connections to the database that we keep alive so we don't need to take the time establishing a connection for each query.

We can create our own incredibly naive implementation using the Composite Pattern. We'll start by defining a **Connection**:

```
public class Connection
{
    public bool CheckedOut { get; set; }
    public Connection (string connectionString)
    {
        //connect
    }
    public void Close ()
    {
        //close the connection
    }
}
```

Now we manage that class with a **ConnectionPool**:

```

public class ConnectionPool
{
    public IList<Connection> Pool;
    public ConnectionPool (string connectionString)
    {
        this.Pool = new List<Connection> ();
        for (var i = 0; i < 10; i++) {
            this.Pool.Add (new Connection (connectionString));
        }
    }
    public void Checkout ()
    {
        //grab a list of connections which aren't checked out
        //return the first
    }
    public void Checkin ()
    {
        //tick the boolean
    }
    public void Drain ()
    {
        foreach (var connection in this.Pool) {
            connection.Close ();
        }
        this.Pool = new List<Connection> ();
    }
}

```

I hesitated to show a **ConnectionPool** example as I'm sure many of you will be poking holes in it (as you should)! Pooling is a hard thing to do, and I don't recommend writing your own. I include it here because it's a real-world example that's easily understood (as opposed to the mind-numbing Foo and Bar nonsense you see everywhere).

If the **ConnectionPool** goes away, so do all the connections. If there are no children (in other words the **ICollection<Connection>** is empty, there is no **ConnectionPool**. The parent and children work together to provide functionality.

If you work in an IDE (such as Visual Studio or Eclipse) – each of the UI elements you see is a component that has a parent. This, again, is the Composite Pattern.

## Decorator

The Decorator Pattern adds behavior to an object at runtime. You can think of it as “dynamic composition”.

We could use the Decorator Pattern as an alternative to the Bridge Pattern above for our **GroovyQuery** engine. We still have the same core bits:



```

public abstract class GroovyQuery
{
    //groovy interface
    public abstract T GetDocument<T> ();
    public abstract T SaveDocument<T> ();
    public abstract IList<T> FetchDocuments<T> ();

    public IDocumentQueryable Documents;
    //etc
}

public interface IDocumentQueryable
{
    T Get<T> ();
    T Save<T> ();
    IList<T> Fetch<T> ();
}

```

But now we get to add Decorators

```

//implementation of the document query interface for
//relational systems.
public class RelationalDocumentDecorator : IDocumentQueryable
{
    GroovyQuery _adapter;

    //Find, Fetch, and Save use the _adapter passed in
    public RelationalDocumentDecorator (GroovyQuery adapter)
    {
        this._adapter = adapter;
    }

    public IList<T> Fetch<T> ()
    {
        throw new NotImplementedException ();
    }

    public T Get<T> ()
    {
        throw new NotImplementedException ();
    }

    public T Save<T> ()
    {
        throw new NotImplementedException ();
    }

    //implement Get, Save, Fetch for Documents below
}

```

With our **RelationalDocument** Decorator we're able to "decorate" the **GroovyQuery** base object with the ability to work with JSON documents, for instance.

## Facade

A Facade hides implementation details so clients don't have to think about it. We can use a Facade for our **GroovyQuery** to pick

an adapter for the calling code, so they don't need to worry about how to wire things together.

We start by defining the classes we'll need:

```
//abstract base class
public abstract class GroovyQuery
{
    public GroovyQuery (string connectionString) { }
}

//implementation for PostgreSQL
public class PostgreSQLQuery : GroovyQuery {
    public PostgreSQLQuery (string connectionString) : base (connectionString){}
}

//implementation for SQL Server
public class SQLServerQuery : GroovyQuery
{
    public SQLServerQuery (string connectionString) : base (connectionString) { }
}
```

We then decide which class to using the Façade pattern, which amounts to a bunch of **if** statements or, sometimes, a **switch**:

```

//a simple class that hides the selection details
public class QueryRunner
{
    string _connectionString;

    //Find, Fetch, and Save use the _adapter passed in
    public QueryRunner (string connectionString)
    {
        _connectionString = connectionString;
    }

    public void Execute ()
    {
        GroovyQuery runner;
        if (_connectionString.StartsWith ("postgresql://", StringComparison.InvariantCultureIgnoreCase)) {
            runner = new PostgreSQLQuery (_connectionString);
        } else if (_connectionString.StartsWith ("sqlserver://", StringComparison.InvariantCultureIgnoreCase)) {
            runner = new SQLServerQuery (_connectionString);
        } else {
            throw new InvalidOperationException ("We don't support that");
        }

        //execute with the runner
    }
}

```

## Flyweight

In the initial versions of **GroovyQuery** we decided it would be very useful to introspect our database whenever a write needed to happen (insert or update query). We did this because knowing more about each table (data types, primary key fields, column names, and default values) would be extremely helpful in crafting up a very usable API.

Unfortunately, this became very slow when the system came under load, so we opted to implement the Flyweight Pattern.

Now, when **GroovyQuery** starts up, it runs a single query that introspects every table in our database, and then loads up a series of small objects that can be used throughout our application:

```

//our Flyweight class
public class Table
{
    public string Name { get; set; }
    public string PrimaryKeyField { get; set; }
    //column and data type information...
}

public abstract class GroovyQuery
{
    //the API as we've come to know it
    List<Table> _tables;
    public void Initialize ()
    {
        _tables = new List<Table> ();
        //query the database for meta information
        //load up the _tables list
    }
}

```

Now, whenever we run an insert or update query we can reuse one of the **Table** instances we have in memory, avoiding the need to make a special query call on each write operation. This pattern can scale to a large number tables, and helps to scale our app to thousands of write operations per second.

## BEHAVIORAL PATTERNS

We've figured out various ways to create our **GroovyQuery** class as well as how to enable functionality by structuring things a certain way. Now let's see how we can use patterns to simplify how clients can use our **GroovyQuery** operations.

Of all the patterns, Behavioral are the most complex. They are also the most useful *when you need them*. When I asked on Twitter that started this chapter (whether GoF patterns are still useful), my friend Jimmy Bogard replied thus:



**Jimmy Bogard** 🍷  
@jbogard

...

Replying to @robconery

**I still use the behavioral patterns but there are a lot more variations and language features as options**

Jimmy creates large, complex applications so he needs these things. I'll get more into large applications in a later chapter, however, if you find yourself on a project creating a stock exchange, you'll need to know these things.

## Chain of Responsibility

We've decided to implement validations for our User and must orchestrate a bit of an approval chain. We can use the Chain of Responsibility Pattern for this, which is focused on moving data through a set of handlers.

### There Is A Better Way

Moving objects through a process chain can be subject to many high-level patterns that are, frankly, much better than this one. I'm showing you this example because you should know the pattern – but when it comes to validations there are better ways to do this.

The first thing to do is to create an abstract handler class:

```
//our handler class
public abstract class UserValidator {
    protected UserValidator Successor = null;
    public void SetSuccessor (UserValidator successor)
    {
        this.Successor = successor;
    }

    public abstract void Validate (User user);
    public void HandleNext (User user)
    {
        if (user.IsValid && this.Successor != null) {
            this.Successor.Validate (user);
        }
    }
}
```

This handler will allow us to define our **Validate** routines as well as any successors that we might have. Now we can create individual validations, starting with a **NameValidator**:

```

public class NameValidator : UserValidator
{
    public override void Validate (User user)
    {
        user.IsValid = !String.IsNullOrEmpty (user.Name);

        if (user.IsValid) {
            user.ValidationMessages.AppendLine ("Name validated");
        } else {
            user.ValidationMessages.AppendLine ("No name given");
        }
        HandleNext (user);
    }
}

```

We can also implement an **AgeValidator**:

```

public class AgeValidator : UserValidator
{
    public override void Validate (User user)
    {
        user.IsValid = user.Age > 18;
        if (user.IsValid) {
            user.ValidationMessages.AppendLine ("Age validated");
        } else {
            user.ValidationMessages.AppendLine ("Age is invalid - must be over 18");
        }
        HandleNext (user);
    }
}

```

One of the nice things about using Chain of Responsibility is that we can formalize our validations into classes that target a single use case, rather than writing a ton of validation code onto our **User**.



Speaking of, let's create our **User** class and orchestrate the validations:

```
public class User
{
    public string Name { get; set; }
    public int Age { get; set; }
    public bool IsValid = false;
    public System.Text.StringBuilder ValidationMessages;

    public User ()
    {
        this.ValidationMessages = new System.Text.StringBuilder ();
        this.ValidationMessages.AppendLine ("Pending save");
    }
    public void Validate ()
    {
        var nameCheck = new NameValidator ();
        var ageCheck = new AgeValidator ();
        nameCheck.SetSuccessor (ageCheck);

        //kick it off
        nameCheck.Validate (this);
    }
}
```

The **IsValid** and **ValidationMessages** properties will let us know if we can save this user, and what's happened during our validation process.

The user's **Validate** routine is where the orchestration is at. We instantiate the name and age validators, and then decide which goes first. We could use Method Chaining here – but I think this is clear enough.

Now we just need to kick it off:

```
var user = new User { Name = "Larry", Age = 22 };
user.Validate();
user.IsValid //true
user.ValidationMessages.ToString() //Pending Save, Name validated, Age validated

var user2 = new User { Name = "Larry", Age = 16 };
user2.Validate();
user2.IsValid //false
user2.ValidationMessages.ToString() //Pending Save, Name validated, Age is invalid - must be over 18
```

As you can see, our validations have gone off and we have errors we need to correct.

## Command

The Command Pattern formalizes requests from one API to the next.

Our data access tool, **GroovyQuery**, is all about writing and reading data from the database. It does this by creating SQL statements that our adapter then executes. We could do this by passing in a SQL string and a list of parameters – or we could formalize it into a command.

We'll start by creating a class for our query parameters:

```

public class QueryParameter
{
    public QueryParameter (string name, string value)
    {
        this.Name = name;
        this.Value = value;
    }
    public string Name { get; private set; }
    public string Value { get; private set; }
}

```

Next we'll create an interface that describes a query command as well as a class that implements it:

```

public interface IQueryCommand
{
    string SQL { get; set; }
    IList<QueryParameter> Parameters { get; set; }
    IDbCommand BuildCommand ();
}
public class QueryCommand : IQueryCommand
{
    public string SQL { get; set; }
    public IList<QueryParameter> Parameters { get; set; }
    public IDbCommand BuildCommand ()
    {
        //return a command that can be executed
        //...
        return null;
    }
}

```

This is our new **GroovyQuery** class, which takes any **IQuery-Command** and executes it:

```

public class GroovyQuery
{
    //the API
    //...
    public IDataReader Execute (IQueryCommand cmd)
    {
        //build the command and execute it
        var dbCommand = cmd.BuildCommand ();
        //...
        return null;
    }
}

```

One thing about formalizing a request like this is that we can scale it to specific needs:

```

public class CreateUserCommand : QueryCommand
{
    public CreateUserCommand (string name, string email, string password)
    {
        this.SQL = @"insert into users(name, email, hashed_password)
            values(@1, @2, @3);";

        this.Parameters = new List<QueryParameter> ();
        this.Parameters.Add (new QueryParameter("@1", name));
        this.Parameters.Add (new QueryParameter("@2", email));
        this.Parameters.Add (new QueryParameter("@3", SomeHashingAlgorithm (password)));
    }
    private string SomeHashingAlgorithm (string val)
    {
        //some solid hashing here...
        return "";
    }
}

```

A little naive, perhaps, but this command encapsulates what it means to add a **User** to our system for SQL and the parameters required.

For what it's worth, I really like the Command pattern. It is a nice alternative to the Repository Pattern which we'll talk about later. In fact, there is a whole "sect" (for lack of better words) in the OO world that uses an architectural pattern called "Command Query Responsibility Separation", aka CQRS. The idea is a simple one: rather than convoluting routines in multiple classes (models, services and so on) you divide things up into simple commands and queries. We'll get into this more in the Architectural Patterns chapter coming up soon.

## Mediator

We want to formalize our document storage capabilities, however adding methods and abstractions to our **GroovyQuery** will make the API more complex, which goes against some programming principles we'll discuss in a later chapter.

In short: simplicity is our goal. We want our class abstractions to do one thing and to do it well.

Let's formalize our document storage idea with the Mediator Pattern. A Mediator is simply a class that sits between two other classes, facilitating communication. It's often used in message-based applications, but we can use a simplified version here:

```

public class DocumentStore {
    GroovyQuery _adapter;
    public DocumentStore (GroovyQuery adapter) {
        _adapter = adapter;
    }
    public T Save<T> (T item) {
        //parse and save the object
        return item;
    }
    public T Get<T> () {
        //pull the record, dehydrate
        return default(T);
    }
    public IList<T> Fetch<T>() {
        //pull the list, dehydrate
        return new List<T>();
    }
    string Dehydrate<T>(T item) {
        //turn the object into JSON
        return "";
    }
    T Hydrate<T> (string json) {
        //resolve
        return default(T);
    }
}

```

Here, we're mediating between our database adapter and any class type of **T**. The adapter doesn't need to know anything at all about **T**, and **T** knows nothing about the adapter that's passed to it other than the fact that it's an abstract **GroovyQuery**.

This is one of those patterns that would cause me to check myself and what I'm doing with my application. I've never had need of a pattern like this but I'm sure it exists somewhere. I have

several .NET friends that use this kind of thing regularly. I prefer simpler and smaller and if that means breaking a single application into smaller ones, so be it.

## Observer

The Observer Pattern facilitates event-based programming. You use this pattern whenever you wire up events in a language like C# or JavaScript (using the **EventEmitter** in Node or listening to DOM events in the browser).

Many frameworks have the mechanics for observation already built in, but let's look at how we can construct an observer by hand by adding methods to our **GroovyQuery** that get fired when certain events occur. These are commonly referred to as callbacks:

```

public interface IListener
{
    void Notify<T> (T result);
    void Notify ();
}
public abstract class GroovyQuery
{
    //API methods etc
    //...
    public IList<IListener> Listeners { get; set; }
    public GroovyQuery ()
    {
        //constructor stuff
        //...
        this.Listeners = new List<IListener> ();
    }
    public virtual IDataReader Execute (IDbCommand cmd)
    {
        //the execution stuff
        //notify all listeners
        foreach (var listener in this.Listeners) {
            listener.Notify (); //optionally send along some data
        }
        return null;
    }
}

```

There are other ways to do this in C# – namely using virtual methods that inheriting classes can implement directly.

A variation on this pattern is the “hook” which you see in Node frameworks like Sequelize or Feathers (a real-time web framework). Ruby on Rails has hooks for **ActiveRecord** as well.



Hooks are explicit observers that you can override on a per-instance basis, such as **onSave**, **onDelete** and so on. The whole idea here is that you're watching a class instance (observing it, if you will) and reacting to things.

Observers are incredibly useful – almost as much as they are annoying when you're trying to track down bugs. The question comes down to this: would you rather put your logic in a single “service” class, or orchestrate a set of events?

Consider a simple case of blog post. When a comment comes in we need to do a few things:

- Run some light moderation, checking for things we'd rather not have in our post comments
- Save the comment to the database
- Notify the blog owner that there's a new comment

There are likely more, but let's go with these for now. We could implement this logic using a simple service class, which we could call **Feedback**. That class could have a method called **newComment** which does the things we need done. We can test that class and verify everything works.

We could also implement these requirements using observers. Our **PostObserver** could see that a new comment has been added with a status of “pending”, which means it needs to be moderated. When the status changes to “moderated” the **onModerated** hook

fires which means we can save the comment in the database. This, in turn, fires the **newComment** hook which executes a notification routine that sends out an email.

I've done both things both ways and you can see a slightly more complex example here, where I handled deep callbacks using Node's **EventEmitter**. This was in the good old days before promises and `async/await`.

I like using events but, as I hope you can see, they can quickly cause confusion in a larger application.

## State

The State Pattern changes an object's behavior based on some internal state. Often this is done by creating formalized state classes. You often hear an implementation of this pattern called a "State Machine", which is a fascinatingly complex way of handling process flow.

For instance, you might have an Order that transitions from pending to paid and then to fulfilled. With each transition, the Order might have specific rules in place such as:

- Refunds are only possible if an order is in the "paid" or fulfilled statuses.
- If an order is in the "refunded" state it can't be changed.

- If an order is “pending” it can only transition to “paid” or “voided”

It's easy to see how the State pattern can help orchestrate these concerns.

To continue with our running example, we want to know what current state our **QueryCommand** is in – if it's new, succeeded, or failed. Let's create some classes that tell us this. We'll start with the **QueryCommand** and a base class for the state:

```
public class QueryCommand
{
    public QueryState State { get; set; }
    //...

    public QueryCommand ()
    {
        this.State = new NotExecutedState (
            "Query has not been run"
        );
    }
    //pass execution off to the state bits
    public T Execute<T> ()
    {
        return this.State.Execute<T> (this);
    }
}
```

Here we've added a class to explicitly handle the idea of a **QueryState** and set it as a property on our **QueryCommand**. The initial state, when the **QueryCommand** is first created is **NotExecutedState**, which is another class we'll need to create.

```

//our base class
public abstract class QueryState {
    protected string Message { get; set; }
    public QueryState (string message) {
        this.Message = message;
    }
    public abstract T Execute<T> (QueryCommand cmd);
}
//an explicit state for not executed
public class NotExecutedState : QueryState {
    public NotExecutedState (string message) : base (message) { }
    public override T Execute<T> (QueryCommand cmd) {
        try {
            //run query execution... and if it works
            var results = cmd.Execute(); //pretend it goes off
            cmd.State = new SuccessState (results, "Query executed successfully");
        } catch (Exception x) {
            //on error
            cmd.State = new FailState (x.Message);
        }
        //return query results
    }
}

```

As you can see, the execution of a query is only available in the `NotExecutedState`, which makes perfect sense. If the execution goes off without a problem, the `QueryState` is transitioned to `SuccessState` which is something we'll need to create. Otherwise we'll set a `FailureState` with an error message.

Something to notice as well is that results are only available in a `SuccessState`, which again makes good sense.

Great. Let's create a **FailState**, for when the query fails and a **SuccessState** for when execution goes off just fine:

```

public class FailState : QueryState {
    public FailState (string message) : base (message) { }
    public override T Execute<T> (QueryCommand cmd) {
        throw new InvalidOperationException (
            "This query already failed execution"
        );
    }
}

public class SuccessState : QueryState {
    public IList<T> Results {get; set;}
    public SuccessState (IList<T> results, string message) : base (message) {
        //assign the results
        this.Results = results;
    }
    public override T Execute<T> (QueryCommand cmd) {
        throw new InvalidOperationException (
            "This query already executed successfully"
        );
    }
}

```

This is obviously simplified but I hope you get the idea. We're handling transitions from one state to the next based on the results of our command behavior and we're being explicit about it. That clarity can be very helpful, but it can also be incredible overkill. We have a lot of code here to convey a simple process: the execution of a query.

With an Order, however, the idea of a *State Machine* becomes a little more understandable. With a State Machine, we transition from one state to the next in a very prescribed way. If you're thinking "hey wait, I read about Finite State Machines a few chapters ago..." and yes! That's exactly what I'm talking about!

## Strategy

The Strategy Pattern is a way to encapsulate “doing a thing” and applying that thing as needed. Code is the easiest way to explain this pattern, as it’s quite simple and useful.

Our document query capability is working well, but it turns out that SQL Server has excellent support for XML, and some users have asked that we support that along with JSON storage.

We can do this using the Strategy Pattern. We’ll start by defining our **GroovyQuery** class and the interfaces we need:

```
public interface IDocumentQueryable
{
    T Get<T> ();
    T Save<T> ();
    IList<T> Fetch<T> ();
}
public abstract class GroovyQuery
{
    //groovy interface
    public abstract T GetDocument<T> ();
    public abstract T SaveDocument<T> ();
    public abstract IList<T> FetchDocuments<T> ();

    public IDocumentQueryable Documents;
    //etc
}
```

We’re going to change things a bit now by using a storage *strategy* which we’ll describe using an interface:

```
public interface IStorageStrategy
{
    T Hydrate<T> (string document);
    string Dehydrate<T> (T item);
}
```

When we store something we either Hydrate it (into an object) or Dehydrate it (into storable XML or JSON). Now we just need to implement them:

```
public class JsonStorageStrategy : IStorageStrategy {
    public string Dehydrate<T> (T item) {
        //turn the object into JSON, return the JSON
    }
    public T Hydrate<T> (string json) {
        //resolve from JSON
    }
}
public class XmlStorageStrategy : IStorageStrategy {
    public string Dehydrate<T> (T item) {
        //turn the object into XML
    }
    public T Hydrate<T> (string xml) {
        //resolve from XML
    }
}
```

Finally we implement the **DocumentStore** with the needed strategies:

```

public class DocumentStore {
    GroovyQuery _adapter;
    IStorageStrategy _parser;
    public DocumentStore (GroovyQuery adapter) {
        _adapter = adapter;
        _parser = new JsonStorageStrategy ();
    }
    public DocumentStore (GroovyQuery adapter, IStorageStrategy parser) {
        _adapter = adapter;
        _parser = parser;
    }
    public T Save<T> (T item) {
        var document = _parser.Dehydrate (item);
        //parse and save the object
    }
    public T Get<T> () {
        //pull the record, dehydrate
        //get the results
        return _parser.Hydrate<T> (result);
    }
    //...
}

```

We’ve “decoupled”, sort of, the storage format from our **DocumentStore** so our users can store XML or JSON. But how and where do you tell the **DocumentStore** which strategy to use? An ENV variable? Configuration file somewhere?

In the .NET world this would likely be done using Inversion of Control as the application starts up and things are configured. We’re going to talk about that and a few other things in the following chapter. What we’re focusing on here is just the strategy part – the implementation is something we’ll deal with later.



## IN THE REAL WORLD...

Many of you will likely notice that I left a few patterns out of the above list – namely the Visitor Pattern, Memento, Template, etc. These are useful patterns to know about, but their use is rare.

For instance, the Visitor Pattern – it's useful if you're parsing tree structures (like Expression Trees in C#) but in everyday code, this is rare. For me, at least.

Also: as you implement patterning as we've done here, the code you write tends to become more generalized and you end up writing a lot more of it just to do a simple operation. This is not what these patterns are for.

A design pattern should make things simpler. If you implement one, have a look at your code before and after, and see if it makes more sense or less. Think about the people inheriting your code in a year, two years and five years. I'll be honest with you and say that if I inherited code that implemented a strategy pattern (or Mediator etc.) I would want to know why and then I would likely see if there was a simpler, more direct way of doing the thing needed.

For instance: our Strategy pattern example is useful because in the future we might want to store documents in an entirely different format – perhaps YAML! I know I run the risk of creating a straw-man argument here, but I do think this perspective is valuable: *at what price do we implement this pattern?*

Let's say we implemented a 20-line switch statement to handle the different storage formats we'd be using. Or maybe we have explicit methods in place like `SaveYaml()` and `GetYaml()`. In terms of pure lines of code written – I would wager we'd be pretty close. Our `DocumentStore` would indeed grow, but we would lose all the Strategy pattern stuff.

The question is: *which one is easier to hold in your head?* What about your teams' heads? My point is this: implementing a pattern doesn't automatically mean your code is correct and maintainable, though they often do help with that.

# OBJECT-ORIENTED DESIGN PRINCIPLES

**A**s you build applications using the patterns we learned in the previous chapter, you begin to see some common side effects or, as they are otherwise known: “code smells”. These monikers are ambiguous and don’t readily communicate what the actual problems might be. Not like the term “carcinogen” or “repellant”.

Understanding what these terms mean as well as why people say them can take years. Often you come to understand that many people who utter these phrases are just repeating what’s been said to them, usually misunderstanding the reasons why and completely missing the point altogether.

There’s a term for this kind of behavior and it’s called “cargo culting”:

*A **cargo cult** is a millenarian belief system in which adherents perform rituals which they believe will cause a more technologically advanced society to deliver goods. These cults were first described in Melanesia in the wake*

*of contact with allied military forces during the Second World War.*

Programmers do this kind of thing all the time, believing if they use a certain pattern, toolset or framework then their application will *just work* where the term “work” means it will be scalable, easily maintainable and solve the client’s problem.

Given this line of thinking, the absence of a pattern or process means that said application will *not* work correctly.

All of that said, a more centered mind can find the process of implementing a design pattern to be very enlightening. The Strategy, Adapter, Mediator and Bridge Patterns, for instance, can lead you to think critically about how much Class A knows about Class B, a term we call “coupling”. Perhaps the Façade pattern will push you to consider how well Class A actually defines the properties and behavior of an A, whatever an “A” might be. This is something we call *cohesion*.

These two ideas: Software Patterns and Design Principles, work together. One does not precipitate the other in any meaningful way. Design Principles help us think about *what* we’re coding and *why*. Software patterns help us assemble that code into a proper structure.

Obviously, this is not a small topic. In this chapter we’ll discover the key principles you should understand, who came up with them, and why. These amount to philosophy dreamed up by people

who've done this sort of thing for a very long time. Some are a bit older, some are newer – all are relevant.

The code for this chapter is, once again, shown in screenshots. If you'd like to play along or have access to the code, you can view it or download it from our GitHub repo.

## COUPLING AND COHESION

You've likely heard these terms before, they're thrown around a lot and have straightforward definitions:

- **Cohesion** applies to how well you've thought out the concepts (and concerns, for that matter) of your application. In other words: how related are the functions of each module or class? You put your **Membership** code into a **Membership** module and your **User** code in a **User** model. The functionality here is cohesive (meaning the ideas bound together logically).
- **Coupling** is the opposite of cohesion. When you couple two or more things, their separate notion becomes one. In our code above we had an example where **Membership** created a **newUser** during the registration process. This coupled **Membership** to **User**. If we moved/renamed/got rid of the notion of a **User** we'd have an error in our **Membership** code. This is tight coupling.

**You want high cohesion, low coupling.** Your classes and modules should make sense for isolating ideas, and not rely on each other to exist. This is the goal, at least.

These ideas were invented in the 60s by Larry Constantine and later formalized in a white paper called Structured Design (Yourdon and Constantine, 1979):

*For most of the computer systems ever developed, the structure was not methodically laid out in advance – it just happened. The total collection of pieces and their interfaces with each other typically have not been planned systematically. Structured design, therefore, answers questions that have never been raised in many data processing organizations.*

It's a fascinating and easy read, and I highly suggest it.

These ideas are *foundational* to writing clear, understandable code and should be in your mind *constantly* as you go about your programming day. A tightly coupled class can be a nightmare to change, especially when the things it relies upon change, which they often will. When one class changes and another one breaks due to that change – that's "tight coupling" and it means you need to fix something pronto.

Cohesion, on the other hand, is more of a "guiding light" if you will. At some point in your career you've probably (or will do so at some point) had a situation where you were coding a class of some kind

thinking something along these lines: “hang on... why is my Post class handling the moderation comments?”

Indeed, a Post is a blob of text assembled in a given way. A User with the role of Moderator is probably a better idea – a more *cohesive* an idea – to handle the task.

Breaking this down even further one could say that this is the act of programming in an object-oriented language distilled to its essence. Ideas resolve into aspects of the real world and we hope that those aspects are what a program is indeed trying to convey. If they are, we’re happy programmers. If they aren’t... we’ve made a big mess. The trick is to keep asking ourselves “is this really what an A (in my Class A) would do?” and, correspondingly, “does Class A really need Class B to exist?”

## SEPARATION OF CONCERNS

Separation of Concerns is about slicing up aspects of your application, so they don’t overlap. These are typically thought of (by developers) as horizontal concerns (they apply to the application as a whole): such as user interface, database, and business logic. The term can equally (and confusingly) be applied to more abstract ideas, such as authentication, logging and cryptography.

Finally, there are vertical concerns, which deal with more business-focused functionality such as Content Management, Reporting, and Membership.

The whole idea of a “concern” in programming has been convoluted, diluted and elocuted to the point of meaninglessness. It reminds me of implementing an API where one of the variables has the name **Context**. It means *nothing* and could represent just about any idea the developer had at the time.

## Some Opinion

I had a discussion once with a developer who suggested I separate SQL from my data access code so I can have a “cleaner separation of concerns”.

Another time it was suggested to me that dividing my .NET code into separate library projects (basically moving files around on disk) was a great way to *separate the concerns* of my application, instead of having all those files together in one place.

Ruby on Rails (which is responsible for the spread of the term) famously suggested that the Model View Controller approach they used was a great “separation of concerns” as it decoupled data access and business logic from HTML. The reality is it did **exactly the opposite**. A Rails view is HTML strewn with artifacts (which are Models) created in a Controller that deal directly with data access.

There is no separation there. Ruby on Rails version 3.0 tried to get there with more generic implementations (so called Railties), but this ended slowing everything down and making a mess in the name of architectural purity. I’m not saying it was *wrong*, I’m saying



it was slow and lost a lot of compelling reasons to use Rails in the first place.

The term “Separation of Concerns” has become a marketing catchphrase that has lost its original meaning. There’s power in the original idea and it’s an expansion on the notion of *coupling*. Instead of decreasing the coupling of your classes, however, Separation of Concerns forces you to consider decoupling your *application*.

The best example of this that I’ve seen is the Django web framework. It pushes the idea of building smaller apps that work together, rather than one large app. Microservices, to use a loaded term, tries to do the same thing but has been cargo-culted into a punchline. We’ll talk more about microservices in a later chapter.

For now, empty your mind and let’s take a trip to the past...

## The Origin Of The Term

The origin of the term comes from this quote from one of my favorite computer science people, Edsger W. Dijkstra:

*Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on an-*

*other day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained — on the contrary! — by tackling these various aspects simultaneously. It is what I sometimes have called “the separation of concerns”, which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts, that I know of. This is what I mean by “focusing one’s attention upon some aspect”: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.*

Every application we build is composed of a vertical subset of processes and rules that try to solve a business need. An eCommerce application will have a sales aspect, a membership aspect, accounting, and fulfillment. These are concerns of the application.

Can we apply this type of thinking to more horizontal ideas? In other words, can we study the notion of logging? Or data access? I’m sure friends of mine would argue that I have, indeed, done the latter many times! Studying these horizontal aspects of our application might make the application better, but I don’t think it will make it more correct.

Now this is where we come up against the weight of history and a little trick that every politician knows: **If you say something long enough it become true.** I think it’s the same with the phrase “separation of concerns”. It’s reminiscent of the phrase “I could care less” or “hone in on”. These phrases make no sense at all, but for

some reason popular American English vernacular has twisted them to mean something and, as time goes on, they get adopted.

I think the same is true with Separation of Concerns. It's a catchall phrase which means "I'm trying to do the right thing", whatever that thing may be. Perhaps it's an effort at file organization or using one of Fowler's enterprise patterns (which we won't be discussing in this book) to abstract away a part of your application ... at this point *it doesn't matter*.

So: when in conversation and someone invokes this trite little phrase, perhaps ask them for some detail. After a few years of doing this, you should have some fun tales to share.

## YAGNI AND DRY

I remember when I started learning Ruby. I loved the simplicity of the language as well as its dynamic design which, I know, many people dislike. You had to have some rigor and much care when building programs with Ruby because you didn't have a compiler and static type checking.

This was freeing, and it was also a little scary.

Part of this rigor was learning a new set of jargon. YAGNI (You Aint Gonna Need It) and DRY (Don't Repeat Yourself) – neither of which came from the Ruby community – started becoming more popular precisely because the practices you needed to adopt to write good

Ruby code leaned squarely on you, as developer, rather than your tooling.

More (and better) testing replaced compiler checks. Test-driven Development (TDD, which we'll get to in a later chapter) helped in this regard as well – forcing you to justify the code you needed to write with a set of tests. In other words: if a test didn't mandate some code's existence, you didn't need that code.

*Don't Repeat Yourself* is something that most developers understand. Duplicated code is difficult to maintain. With large applications, this is almost impossible – but it is an important idea to keep a focus on.

If you have an application with 300 classes and 50,000 lines of code, you're bound to have duplication. The trick is to spot it and, hopefully, to simplify your future by abstracting it in some way.

## **The Wrong Abstraction**

In January 2016 Sandi Metz wrote a great article about the perils of focusing too much on DRY. The punchline being:

*... duplication is far cheaper than the wrong abstraction... prefer duplication over the wrong abstraction...*

The article sprang from a talk she gave at RailsConf in 2014.

The main idea is that you don't want to create abstractions solely for the sake of avoiding repetition; it needs to fit your overall approach. If you can't fit it, just let the duplication be.

## TELL, DON'T ASK

Another Rubyism that I quite like came from Ruby's inspiration: Smalltalk. Whenever you invoke a method on a Ruby class you send it a message. You tell that instance that you need it to do something, or that you need some data back of some kind.

If you ask an object instance a question, then you'll need to know something about that object or its state, which breaks the notion of encapsulation.

If you think back to our **GroovyQuery** from a previous chapter, imagine this as our API:

```

using System;
using System.Data;

public class GroovyQuery
{
    public bool IsCommandValid (IDbCommand cmd)
    {
        //logic
    }
    public bool IsConnectionAvailable ()
    {
        //check connection pool to see if one is ready
    }
    public IDataReader Execute (IDbCommand cmd)
    {
        //execution
    }
}

```

To use this API effectively I would need to ask two questions and finally get around to telling the class what to do (**Execute**). In short: I need to know way more about the API than is needed.

A better way to do this is by moving a few things around:

```

public class GroovyQuery2 //Telling
{
    bool CommandIsValid (IDbCommand cmd)
    {
        //logic
    }
    bool ConnectionIsAvailable ()
    {
        //check connection pool to see if one is ready
    }
    public IDataReader Execute (IDbCommand cmd)
    {
        var commandIsValid = CommandIsValid (cmd);

        if (ConnectionIsAvailable () && commandIsValid) {
            //execution
        } else {
            throw new InvalidOperationException ("Can't run this query");
        }
    }
}

```

The responsibility for deciding whether the query can run is now within **GroovyQuery**, which is where it should be.

## LAW OF DEMETER (OR: PRINCIPLE OF LEAST KNOWLEDGE)

The Law of Demeter (LoD, or “Deep Dotting”) is an offshoot of loose coupling. In short: you shouldn’t have to “reach through” one object to get to another. This can be further nuanced to mean you shouldn’t have to reach deeply into one object to do the thing you need to do.

Let's examine both.

Our **Membership** system is working well, but some users aren't behaving themselves so we need to give them a bit of a timeout. With our first go we decide to drop a Suspend method on User because we're telling them they're suspended:

```
public class DB {
    public User GetUser (int id) {
        //call to the DB, getting record
        //returning an empty user for now
        return new User ();
    }
}

public class User {
    public String Status { get; set; }
    public void Suspend() {
        this.Status = "suspended";
    }
}

public class Membership {
    DB _db;
    public Membership () {
        _db = new DB ();
    }
    public User GetUser (int id) {
        //get the user
        return _db.GetUser (id);
    }
}
```

To suspend a user we need to access them from the Membership module and then suspend them:



```
var membership = new Membership();  
membership.GetUser(1).Suspend();
```

This is a violation of LoD. We had to reach through **Membership** to get to the **User**. You might be wondering ... so what?

It's a subtle point, sure, but the more you think on it the more you realize how you're muddying the principles we've been reading about.

In essence: *we've punched a hole in our membership abstraction* by dividing the responsibility for changing the user between two different classes. Cohesion is breaking down and coupling is going up.

It doesn't make sense to involve **Membership** at all here, except for the fact that we need to get at the **User**. So let's make a choice, and it's a simple one: **Membership** has the responsibility of adding and retrieving users (aka changing them) so let's have it update the user's status as well:

```

public class DB {
    public User GetUser (int id) {
        //call to the DB, getting record
        return new User ();
    }
    public void Save (object item) {
        //save to DB
    }
}

public class User {
    public String Status { get; set; }
}

public class Membership {
    DB _db;
    public Membership () {
        _db = new DB ();
    }
    public User GetUser (int id) {
        //get the user
        return _db.GetUser (id);
    }

    public void SuspendUser (int id) {
        var user = this.GetUser (id);
        user.Status = "suspended";
        _db.Save (user);
    }
}

```

**Note:** *we're being pushed in a particular direction as we consider these design principles. Can you see what it is? We're removing "external", for lack of better words, functionality from our User class. It can now stand on its own with no external dependencies. Our DB, on the other hand, needs to know what a User is in order to work. This is possibly more coupling than we want and we'll discuss it in the Architecture chapter.*

Some developers will focus on “dot counting”, claiming that the use of too many dots is, all by itself, a violation of LoD. Sometimes it is, sometimes not.

Consider this API:

```
var liTag = new Html.Helpers.HtmlTags.Lists.ULTag.LiTag();
```

This API is kind of ridiculous, I must say. However, it’s an organizational choice and not necessarily a violation of LoD. This could be horrible namespacing for all we know! Or it could be a lack of imagination. We don’t know the inner workings of the Html helper library (and trust me, you don’t want to), so it’s not exactly accurate to call for a violation just by looking at dots.

I was reading my friend Phil Haack’s blog while researching this subject, and he had a great quote from Martin Fowler:

*I’d prefer it to be called the Occasionally Useful Suggestion of Demeter.*

I hate to leave you with vagary, but hopefully you can see how “deep dotting” and LoD aren’t always the same thing.

# DEPENDENCY INJECTION

One way to loosen up your code is to send in the dependencies that a class needs through its constructor. The best way to see this is with some code.

Our **Membership** class is using the database to retrieve and save a **User**:

```
public class Membership{
    DB _db;
    public Membership(){
        _db = new DB();
    }
    public User GetUser(int id){
        //get the user
        return _db.GetUser(id);
    }
    public void SuspendUser(int id){
        var user = this.GetUser(id);
        user.Status = "suspended";
        _db.Save(user); //Coupling
    }
}
```

This couples the **Membership** class to the **DB** class which is responsible for data interactions. We've read *The Imposter's Handbook*, so we know that coupling is bad – but how can we change this?

The simple answer is to inject the dependency through the constructor rather than to invoke it in place:

```

public class Membership {
    DB _db;
    public Membership (DB db) {
        _db = db;
    }
    public User GetUser (int id) {
        //get the user
        return _db.GetUser (id);
    }
    public void SuspendUser (int id) {
        var user = this.GetUser (id);
        user.Status = "suspended";
        _db.Save (user); //Coupling
    }
}

```

Now our class doesn't need to know how to instantiate **DB**, which is one step in the right direction. There's still a bit too much coupling, however as our **Membership** class cannot be used unless a **DB** instance is passed in.

Let's see how we can loosen this up a bit more.

## INTERFACE-BASED PROGRAMMING

Many languages support the idea of interfacing with an ability, rather than a type itself. With C#, Java and now TypeScript these are called *Interfaces*. With languages such as Swift and Elixir this is done with Protocols. For our purposes I'll use interfaces, so translate as you need.

The goal of working with interfaces is to describe an ability of your application. In our case all we care about is that we can retrieve

and save a record from a data store. It's important to keep this interface light because doing so will actively increase cohesion and drive down coupling:

```
public interface IDataStore {
    public void Save<T>(T item);
    public T Get<T>(int id);
    public IList<T> Fetch<T>();
}
```

This is a good start. We can now use our new interface:

```
public class Membership {
    IDataStore _db;
    public Membership (IDataStore db) {
        _db = db;
    }
    public User GetUser (int id) {
        //get the user
        return _db.Get<User> (id);
    }
    public void SuspendUser (int id){
        var user = this.GetUser (id);
        user.Status = "suspended";
        _db.Save (user);
    }
}
```

Much better. We still have coupling to the notion of an **IDataStore**, but it's unavoidable at this point (unless we want to work directly

with eventing, but that's probably overkill). We can now implement an **IDataStore** to do all kinds of things for us, such as:

- Store data in a relational system
- Store data in a NoSQL system
- Store data directly in memory for testing purposes

Using interfaces like this is a cornerstone of object-oriented programming. Injecting them, as we're doing here, is a great way to keep your code isolated.

It does come at a price.

## INVERSION OF CONTROL

As you build out your application, paying attention to interfaces and dependency injection, you will start to see the number of dependencies for a given class begin to spiral a bit out of control. The best way to see this is with some code.

In the real world, our **Membership** class will probably need quite a few external dependencies:

- A hashing library for password storage
- An email library for sending a new user a note
- A rules module for accepting new users

- A logger module for logging

This means our constructor is going to grow:

```
public class Membership{
    IDataStore _db;
    IEmailer _email;
    ICrypto _crypto;
    ILogger _logger;
    IRulesEngine _rules;
    public Membership(IDataStore db,
        IEmailer email,
        ICrypto crypto,
        ILogger logger,
        IRulesEngine rules){
        _db = db;
        _email = email;
        _crypto = crypto;
        _logger = logger;
        _rules = rules;
    }
}
```

Yikes. Registering a user isn't much fun either. Here's the **Register** method on the same **Membership** class:



```

public void Register(IRegisterable user){
    //validations etc
    if(_rules.CanRegister(user){
        user.Status="Registered";
        user.HashedPassword = _crypto.HashPassword(user.Password);
        _db.Save(user);
        _email.SendWelcome(user);
        _logger.Info("New user added: " + user.Email);
        return user;
    });
}

```

This is nuts. Every time we want to use **Membership** we'll need to create instances of its dependencies which, themselves, likely have dependencies of their own we'll need to create (and then inject). This is simply sweeping the dependency coupling somewhere else.

This is where Inversion of Control comes in. With Inversion of Control you have a separate mechanism (called a "container") which is responsible for creating and injecting all the dependencies you need and then giving those injected objects to you when you need it.

Here is some pseudocode for an IoC container modeled after my friend Nate Kohari's excellent Ninject Project:

```
//our app start
public void Main(){

    Container container = new Container();
    container.Bind<IMembershipStore>().To<PostgreSQLAdapter>();
    container.Bind<IEmailSender>().To<MailgunSender>();
    container.Bind<ILogger>().To<Log4Net>();
    container.Bind<ICrypto>().To<SuperCryptoThingy>();
    container.Bind<IMembership>().To<Membership>();

    //get an instance of Membership
    var membership = container.Get<IMembership>().InSingletonScope();

}
```

We have our interfaces mapped to concrete implementations in a single place. If we ever need to change anything, we just change it here.

When we need an instance, we simply need to access our container, which needs to be global to our application<sup>7</sup>. The container then orchestrates the instantiation of the classes we want. A bonus to this is that we can set a “scope” on the object. For instance, in the example above I’m setting the lifecycle to a Singleton.

You can do many other things with Inversion of Control containers – and they are quite useful.

---

<sup>7</sup> I know the word “global” is a bad one when it comes to programming and variables. Don’t get stuck on this! We simply need to ensure that we can access the container when we need it, and often tools such as Ninject have ways of ensuring that you can access the container without using global variables.

## Is This Really Useful?

You might be getting the sense that we're creating a bit of a "meta" programming system here, where object instantiation is removed from the language constructs themselves and into this separate... mechanism of our own creation.

This is where we start getting subjective. There are quite a few developers out there who see patterns like the above as flaws in the language or, more broadly, as flaws in object-oriented programming itself. This is a great quote from Lawrence Krubner in his essay *Object-oriented Programming is a Disaster and Must End*:

*I have seen hyper-intelligent people waste countless hours discussing how to wire together a system of Dependency Injection that will allow us to instantiate our objects correctly. This, to me, is the great sadness of OOP: so many brilliant minds have been wasted on a useless dogma that inflicts much pain, for no benefit.*

Now that we understand a bit more about dependency injection and inversion of control containers – do you think you'll waste "countless hours"? To be honest: *yes, I have*. But it was my fault. That last bit doesn't make me feel any better and is the chorus sung by most framework adherents when you criticize their framework of choice: "PEBCAK" or *Problem Exists Between Chair and Keyboard*. It's always the user's fault innit?

Keeping your containers happy and working properly in an IoC sense is not as simple as it seems. As your application grows and

becomes more complex, it becomes easier to find yourself creating circular dependencies. For instance: we might decide to create an **ILogger** implementation that saves logs to a database. We decide to reuse our **IDataStore**, which requires an instance of **ILogger** which requires an instance of **IDataStore**...

These problems, as you might be sensing, typically have to do with application design rather than object-oriented programming. Which seems to be a recurring problem in our industry.

If a language, platform or framework leads you down a snarled path of bad design, it's usually your fault. Or is it?

I'd like to leave this chapter with a great quote from my friend Gary Bernhardt, which he offered during his amazing talk *The Birth and Death of JavaScript*:

*The behavior that you see a tool being used for is a behavior that tool encourages.*

It's easy to dismiss recurring structural problems as ignorance on the programmer's part. If the same problem occurs throughout the development community, however, is it really a problem of ignorance?

I don't have an answer.

# SOLID

In object-oriented programming circles it's almost impossible to escape Martin/Feathers/Meyer's SOLID principles, which are:

- **Single Responsibility Principle** (SRP): a class should do one thing and have only one reason to change.
- **Open/Closed Principle**: a class should be open for extension but closed for modification.
- **Liskov Substitution Principle**: a class should be able to be substituted with its subclass.
- **Interface Segregation Principle**: small, role-based interfaces are better than a massive object.
- **Dependency Inversion Principle**: higher level classes shouldn't depend on lower-level ones; abstractions shouldn't depend on details - rather details should depend on abstractions..
- SOLID has been around for many years and is considered to be something you just don't argue with. Which is why I'm going to start this chapter doing just that.

## Some Opinion On Solid

I've struggled with SOLID. On the face of it, SOLID makes sense doesn't it? Smaller classes and clear thinking when creating your domain objects - what's wrong with that? Nothing, really, especially

if you want to build a career as a coding guru - writing books and booking speaking engagements based on an acronym which is otherwise *common sense*.

That's a strong opinion, but it's an important one. Let me explain.

I used to work in rather large development groups. Nothing was more infuriating than a condescending grunt from a code reviewer who would point at some code saying "that violates SOLID". As if that makes what you've written *wrong*.

This is my main critique of SOLID: *it gives people a tower to climb and a robe to hide behind*. Your code is wrong because it violates a doctrine that was created to propel a career, not your project.

This is where the "science" part of "computer science" needs to come into play: *How you've built something needs to be considered separately from how it works*. It's easy to read that sentence and think "oh wow Rob is suggesting developers become sloppy" - but I'd like for you to consider the opposite of that opinion.

How many times have you looked over a nicely SOLIDified code-base and thought "this is pure crap". An explosion of classes, interfaces and abstractions all thrown together to create a todo list. Ceremony for the sake of ceremony and escaping the dreaded "this class violates Liskov..."

There's a good idea behind SOLID, but the idea should be the focus. The inspiration to keep things as simple as you can so your future self (or another developer who has to support what you've

created) doesn't get overwhelmed and confused. This has a practical aspect as well as egotistical: *if they can't figure out what you've done, they'll probably throw it away*. Just like you've done many times with other people's code.

I like how Dan North puts it: *just write simpler code*. This is a powerful idea because "simple" doesn't mean "wrong" - in fact it means the opposite. Think about world-class athletes - how easy do they make their sport look? The best interface designs and hardware form factors all strive for the same thing: *blissful, wonderful simplicity*.

I invite you to follow the link above and have a look at Dan North's presentation on "Why every element of SOLID is wrong". It's good stuff, if only to make sure we don't fall into the trap of adhering to doctrine over good design.

Dan, by the way, is the creator of Behavior-driven Design and is a rather well-known figure in the development world. This talk (which I was at) was a lightning talk and was intended to be for laughs as much as for thought - but to me it was the best talk I had seen that entire week.

## **Single Responsibility**

A class should have a single responsibility to your application or, as Martin puts it, *a single reason to change*. This sounds simple, but it's a little tricky to grasp.

Most applications have a class for a **User**:

```
public class User
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User ()
    {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}
```

This class changes when the data about the **User** changes. But how does that information actually change?

Let's say we want to register the user into our system. We could do something like this:



```
public class User {  
    public string Name {get;set;}  
    public string Email {get;set;}  
    public string Status {get;set;}  
  
    public User() {  
        this.Name = "Guest";  
        this.Status = "Anonymous";  
    }  
  
    public void Register(string name, string email) {  
        this.Name = name;  
        this.Email = email;  
        this.Status = "Registered";  
        //save to the DB or something else  
    }  
}
```

This class is now doing two things: describing a **User** based on some data and registering a user into the system. We can keep with SRP if we move the registration responsibility off to another class:

```

public class User {
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User ()
    {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}

public class Membership {
    public User Register (string name, string email)
    {
        //validations etc
        var newUser = new User { Name = name,
                                Email = email,
                                Status = "Registered" };

        //save to the DB or something else
        return newUser;
    }
}

```

## OPEN/CLOSED

This principle may seem obvious to you, but at the time it was created and refined (late 80s, early 90s) it addressed a real problem.

Open/Closed says that a class or module should be open for extension, closed for modification. Or, put another way: let people override/extend your code without needing to modify it.

Let's say we're working in Node, and we need to install a module from NPM. We run our `npm install` command and a **node\_modules** directory appears with our module.

It turns out that we find a bug in one of the methods! We could go and fix the bug directly inside of **node\_modules**, but that would violate Open/Closed! The good news is that the developer left the module extensible, so we can go in and override the bug with a fix. This module was open for our extension but closed for our modification.

With object-oriented languages, this kind of thing is solvable if you allow users of your classes and modules to inherit and extend key bits of functionality that you're providing.

## LISKOV SUBSTITUTION

Liskov is a subtle principle but can catch some very serious bugs that creep into your application. The principle has to do with inheritance and how inheriting objects behave. It says:

*if  $S$  is a subtype of  $T$ , then objects of type  $T$  may be replaced with objects of type  $S$*

Let's say we have a User and Administrator class:

```

public class User{
    //...

}
public class Administrator : User{
    //...

}

```

My program should work if I pass **Administrator** to any routine that expects a User.

The classic example of breaking LSP is the **Square** and **Rectangle** analogy:

```

public class Rectangle {
    int _height;
    int _width;

    public virtual void SetHeight (int height) {
        _height = height;
    }
    public virtual void SetWidth (int width) {
        _width = width;
    }
}
public class Square : Rectangle {

    public override void SetHeight (int height) {
        this.SetHeight (height);
        this.SetWidth (height);
    }
    public override void SetWidth (int width){
        this.SetHeight (width);
        this.SetWidth (width);
    }
}

```

This code is obviously confusing in that we're coding our way around a problem in our implementation. While a square, in reality, is a rectangle (if we're talking math) – implementing it as a **Rectangle** in our code causes us to do some weird things.

Moreover, passing a **Square** around as if it were a **Rectangle** could cause some very strange things to happen in our video game code later on.

## INTERFACE SEGREGATION

The Interface Segregation Principle is all about targeted, simple API creation so code is easy to use and implement. Rather than create a small set of large interfaces, favor a larger set of smaller, more generic interfaces.

Let's think about registering our **User** again:

```

public class User {
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User () {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}

public class Membership {
    public User Register (string name, string email) {
        //validations etc
        var newUser = new User { Name = name, Email = email, Status = "Registered" };
        //save to the DB or something else
        return newUser;
    }
}

```

This works fine, but the notion of a **User** is bound to our **Membership** class. We could solve this with an interface (something like **IUser**), or we could lean on ISP and focus on what's really needed. Here we can use **IRegisterable** with our **User** class:

```

public interface IRegisterable
{
    string Name { get; set; }
    string Email { get; set; }
    string Status { get; set; }
}
public class User : IRegisterable
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User ()
    {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}

```

Now our **Membership** class is a bit cleaner:

```

public class Membership {
    public IRegisterable Register (IRegisterable user) {
        //validations etc
        user.Status = "Registered";
        //save to the DB or something else
        return user;
    }
}

```

We could extend this notion further with **IAuthenticatable** as well, if we want. Abstracting your code like this helps hide implementation details – which means making changes in the future becomes a lot easier.

# DEPENDENCY INVERSION

Dependency Inversion is all about loosening up the relationship between classes and modules in your code. The definition is a bit wonky:

- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions

Our **Membership** module is a high-level module, and we need to depend on some low-level modules in order for things to work.

For instance: we need to save our **IRegisterable** object to the database. We could go the route of creating a new database connection right in our **Register** code, if we want:



```

public interface IRegisterable {
    string Name { get; set; }
    string Email { get; set; }
    string Status { get; set; }
}
public class User : IRegisterable {
    //...implement interface
}
public class Membership {
    public IRegisterable Register (IRegisterable user) {
        //validations etc
        user.Status = "Registered";
        //save to the DB or something else
        var db = new PostgreSQLAdapter ();
        db.Save (user);
        return user;
    }
}

```

But this is coupling or binding our **Membership** class directly to our **PostgreSQLAdapter**. This will solve our problem now, but in the future, we will likely come across some problems:

- We might want to switch data access tools later on, maybe moving from our homespun routines to an ORM
- We might move to a document database, or something hosted (like Amazon's Dynamo DB)
- Our adapter might change the way it's constructed – perhaps moving to a Factory pattern or the like. Or maybe we'll have a Mediator...

In short: our **Membership** class knows way too much about the construction and execution of our adapter – it's too tightly *coupled*. Let's invert this dependency, shall we?

```
public interface IMembershipStore {
    void Save (object item);
}

public class PostgreSQLAdapter : IMembershipStore {
    //implement interface
}

public class Membership {
    IMembershipStore _store;
    public Membership (IMembershipStore store) {
        _store = store;
    }
    public IRegisterable Register (IRegisterable user) {
        //validations etc
        return _store.Save (user);
    }
}
```

Now we're free to change our storage approach however we like in the future. Our high-level module doesn't depend on a lower-level one (our database bits) and the abstractions we're using don't depend on the details of the implementation.

## Dependency Inversion vs. Dependency Injection

These are not the same thing, though they sound alike and, in many cases, look alike. Dependency Inversion is simply structuring our code to work with interfaces in a particular way (as described

above). Dependency Injection is how these interfaces are provided to the classes that need them.

## SOMETHING I FIND USEFUL

Here we arrive at the end of my research into patterns and principles and the beginning of my opinion on the matter. It's important to me that you recognize that what's to follow is like every other thing you'll read about architectural strategies in software: *just my opinion, not necessarily factual*.

I do think it's important to share our experience, but I *don't* like positioning that experience as "The Way". Of course it's not The Way. How could it be? Yet, there is value in learning about how others have built things (so you can come up with your own "Way") so I decided to change course in this book, for this one section, and share with you my *opinion*.

So here you go. All of what you're about to read is, in effect, how I've been assembling programs for the last decade <sup>8</sup>. Some things felt right, some didn't and I have been able to tweak the things that didn't feel right as the result of writing this book. Now they feel very good indeed.

---

<sup>8</sup> Sort of. I've used Rails and Django a few times, which have their own ways of doing things. I've also used ORMs for smaller projects which also have their own opinions. When I'm free to choose what I want, this is what I do.

## Navel Gazing

I remember buying book after book in the late 90s and early 2000s as I was trying to improve my skills as a developer. I read about domain-driven design (DDD), TDD, BDD and was constantly reading Martin Fowler's blog, digging for guidance on how to put things together. I was convinced that knowing my patterns and principles would help me write better software. I was wrong.

It's easy to get vapor-locked on doctrine, focusing on that instead of solving the problems you're getting paid to solve. This is a common thing but I'm here to suggest a way to avoid this entirely: come up with your own personal, architectural "baseline". The thing you do when you're not sure what else to do.

Here's mine.

## Simple Models Used in Services, Injected with the Needful

My goal is to keep models as light as possible and the dependencies between *everything* as light and simple as I can make them. Here are the rules:

- Models are simple classes, inheriting *nothing* and can only change their own state. As you'll read more in the next chapter, this is a functional approach with the goal of "code purity".

- Decisions are made in “service” classes, which model a process. These classes are the most susceptible to coupling, which is something I try to avoid at all costs.
- Service classes are handed everything they need to execute, taking the smallest dependency footprint possible. This is achieved by making sure I inject interfaces or lightweight structs whenever and wherever I can.

That’s the core of it really. To be more concrete about it, let’s use the shopping cart analogy one more time:

- A **ShoppingCart** class is concerned with one thing only: holding items for a buyer and representing that data.
- A **Checkout** class might represent the idea of closing a sale, accepting a cart, payment processor and repository as arguments and using these to execute business logic.
- The cart, payment processor and repositories would be represented by interfaces, not the actual classes. If I’m using something like Ruby or JavaScript, these would be represented by lightweight structs or hashes.

I feel like I’m waiving my arms around some and I really want to show you what I mean with code, but I think it’s more valuable if you could *feel* this approach and visualize it mentally.

When checking out, who cares about the cart? What you really care about are the items in the cart so in that sense all I really need to do is to have the SKUs and quantity of each item. The total will be

determined by the checkout routine which will go and look up the products using the repository interface.

A lot of frameworks (Rails, Django and others) provide models that are database aware which means that in order to test them, you have to boot up the entire application to have the context for that model. That's some serious coupling, and also very slow. It's great for smaller applications, absolutely blows for bigger ones.

I favor models that inherit nothing, have the fewest number of dependencies possible and fit on a single page (aka "no-scroll models").

## **Using a More Structured Language**

Using C#, for instance, we might be tempted to create a `ShoppingCart` class that holds `ShoppingCartItems` and has a `checkout` method:

```

class ShoppingCartItem{
    string sku {get;set;}
    int quantity {get;set;}
    string name {get;set;}
    int price {get;set;} //pennies
}
class ShoppingCart{
    IEnumerable<ShoppingCartItem> items;
    constructor(){
        this.items = new List<ShoppingCartItem>();
    }
    addItem(Product product){
        //adding logic
    }
    checkout(){
    }
}

```

This will work just fine in most cases, but I've come to *loathe* dependencies of any kind. My `ShoppingCart` depends on `Product` and `ShoppingCartItem` and the `checkout` method will surely depend on a payment processor of some kind as well as a database.

This is going to get yucky, fast. For instance - our `Checkout` process might be used in the future to process a `Subscription` ... which means writing code to get around the type restriction in our `addItem` method. This means that we'll likely be breaking things.

What if, instead, we used an interface to define this relationship:

```

interface IBuyable{
    string sku {get;set;}
    int price {get;set;} //pennies
    int quantity {get;set;}
}

class Product: IBuyable {
    //interface bits
}

class ShoppingCart{
    IEnumerable<IBuyable> items;
    constructor(){
        this.items = new List<IBuyable>();
    }
    addItem(IBuyable item){
        //adding logic
    }
}

```

The only thing my cart cares about, really, is a SKU, price and a quantity. We're now free to add other things that our users might want to buy - like gift purchases, gift cards or support time.

Also notice that I removed the `checkout` method. That's a business process and, to me, belongs in a service class:



```

class Checkout {
    ICatalog catalogRepo {get;set;}
    IPaymentProcessor processor {get;set;}
    constructor(ICatalog catalog, IPaymentProcessor processor){
        this.catalogRepo = catalog;
        this.processor = processor;
    }
    process(IEnumerable<IBuyable> items){
        //pull the products from the catalog
        //validate the order
        //process the sale
        //save it
    }
}

```

This is where the decisions happen. Creating a checkout requires the tools to execute the checkout - a payment gateway and some type of repository of products (which I'm calling a catalog).

The `process` method executes the transaction. This is where the heavy lifting happens! Note that since we're using interfaces here, we can mock them easily for testing, which is a huge bonus.

## Creating the Actual Instances

You might be wondering: *all these interfaces! Where do you instantiate your actual types?* I tend to push that as far out as I can. A lot of people will orchestrate these things using Inversion of Control containers (see above) so that types and instances are defined and dealt with in one place.

For me, I like to keep things exceedingly simple. In this case, I might have the Controller (if I'm creating a web application) or a config file somewhere create the actual classes for me:

```
//pseudo code  
class CheckoutController {  
  constructor(){  
    this.stripe = new StripeGateway();  
    this.catalog = new ProductRepository();  
  }  
  checkout(post){  
    var checkout = new Checkout(this.catalog, this.stripe);  
    checkout.run(post.items)  
  }  
}
```

This is pseudocode, but hopefully you get the idea. When the controller is created, it creates the classes that I will then pass to an instance of my Checkout class. This, to me, is about as complex as I like to keep things.

If you don't have interfaces, let's say you're using JavaScript, you could implement something like this:

```
class Checkout {  
  constructor(repo, {charge}){  
    this.repo = repo;  
    this.chargeMethod = charge;  
  }  
  process(items= []){  
    //...  
  }  
}
```

We're doing the same thing but, fortunately or not, we're leaning on the dynamic nature of JavaScript to help us out. All we care about for our `Checkout` class is that we have a repository passed in (`repo`) and something with a `charge` method, which would be our Stripe gateway.

The bare minimum needed to run, that's what we want to focus on.

## Why This is Useful

I find this way of doing things useful for two main reasons:

1. It keeps my models extremely lightweight and easy to test.
2. It allows for easier testing with my service classes given that I can mock, or stub, the things my service classes need.
3. It keeps my tests fast since I don't need to boot up an entire application context, with a database connection, in order for them to run.

Of course things could get more complex as time goes on but this chapter isn't about architecture nor how to scale the complexity of your application. I did think it would be nice, however, to show you the patterns I use on a daily basis.

# FUNCTIONAL PROGRAMMING

**S**ome people love it and claim it's the only way to write software. Others see it as a fad and roll their eyes.

Let's come away from those extremes. Functional Programming is based in a foundational concept which we learned about a few chapters ago: *Lambda Calculus*. It's not magical, nor is it something you should ignore because you're amazing. It is simply something you need to understand.

Like most things in Computer Science, functional programming is full of jargon, idioms and practices that, at first, might be a bit opaque. If you take the time, however, to let it soak in... functional programming can change the way you write software if you come from an object-oriented background.

## A CHANGE IN THINKING

I learned a functional language two years ago; one that I love using: *Elixir*. It has changed the way I think about programming. It's

the only functional language I know, so I'll be using it to do some of the demos you're about to see.

For others, I'll be using JavaScript/ES6. I chose this language because 1) most programmers know it at least a little and 2) it has some functional characteristics - at least to the level where you can get your point across.

We'll cover four main topics in this chapter:

- Immutability (things that can't/don't change)
- Purity (functions rely only on what they're given)
- Side Effects (functions only operate on data they're given)
- Currying (breaking big functions down to little ones)

I also threw in a very brief discussion about functors and monads at the very end, something I barely understand but I think is critical to have a think on.

Off we go...

## IMMUTABILITY

The first word that comes to mind whenever you hear "functional programming" is usually "immutable". As I am sure you know, it means "not changeable" in plain English, but how does that translate to programming? Moreover: *who cares?*

If you're an object-oriented (OO) programmer you're used to creating classes and then instantiating them. You set their properties and send them messages which change their properties or tell you something about themselves. This is not possible with functional programming.

There are no objects, no classes, no "state" as you might think of with OO. There are only functions which transform things. You give a function what it needs, it hands you back what you want (hopefully). This might sound incredibly confining, but there are practices that go along with this idea that make it rather compelling. Let's have a look.

If you want to follow along, you should have Elixir (version 1.3 or so) installed, as well as Node (version 6+).

## Simple Immutability with Elixir

Consider this assignment:

```
friend = %{name: "Clara"}  
IO.inspect friend  
#%{name: "Clara"}
```

This is a **map** in Elixir and it looks a lot like an object in JavaScript or a hash in Ruby. The difference is you can't do this:

```
friend = %{name: "Clara"}
friend.name = "Mike"

*** (CompileError): cannot invoke remote function friend.name/0 inside match
[Finished in 0.159s]
```

The error message is a bit odd as it has to do with the way Elixir tries to match values (pattern matching), but essentially it means “you can’t change Clara’s name to Mike”. So, what do we do?

We ask the **Map** library to update the map. This is going to look a bit weird, but once I explain it more you’ll hopefully understand:

```
friend = %{name: "Clara"}
friend = Map.put(friend, :name, "Mike")
IO.inspect friend
#%{name: "Mike"}
```

I do not blame you if you’re completely underwhelmed at this point... and more than a little confused. There’s just no graceful way to go about this, so I’m throwing you in the deep end straight away.

Here’s what just happened:

- We created a map with a name key set to “Clara”
- We asked the Map library to put the value “Mike” in Clara’s place
- The Map library gave us back a completely new map ... sort of



- The friend variable was rebound to the new Map

Elixir provides some helpful features which you don't find in more "strict" functional languages. For instance: in Erlang (a functional language that Elixir is based on) you can't do what we just did (re-binding **friend**). You would need to create a whole new variable - something like **renamed\_friend** to hold the result of the **Map.put/2** operation. Elixir is less strict, so it *looks* like we're changing the **friend** variable.

A natural thought that you might be having at this point is *wait a minute, if I need to update a map, I need a whole new one? Isn't this horrible for memory?* This is a very good question! The short answer is: *no, it's not*. This is because Elixir simply uses a pointer under the covers. The initial **friend** map is still there and the updated map points back to it. This keeps things rather light and you don't fall into the immutability traps that you have in other languages (strings with .NET, for example). Furthermore: this kind of thing is only possible with a functional language.

That's the technical teardown, let's discuss something a bit more intangible: *this code is just ugly*. I would agree with you there too. Fortunately what I wrote is *not* idiomatic. I said at the beginning that functional programming is all about *transforming* data through a set of functions. Let's rewrite our code to support that idea:

```
%{name: "Clara"} |> Map.put(:name, "Mike") |> IO.inspect
```

Much better. That toothy thing is the “pipe” operator and works in much the same way as its Unix counterpart. The result of a given function is piped into a following one as its first argument, and so on.

Before we end our introduction to immutability, I want to underscore the notion of rebinding. Let’s change our code a bit, and I’ll reintroduce our **friend** variable:

```
friend = %{name: "Clara"}  
friend |> Map.put(friend, :name, "Mike") |> IO.inspect  
IO.inspect friend
```

Running this you’ll see that our **friend** variable wasn’t changed at all. **Map.put/2** simply passed back a new map with the updated key, which we piped into **IO.inspect** directly. Thus, we have two different outputs:

## FORMALIZING DATA WITH STRUCTS

I don’t want to leave you with the impression that data in a functional language like Elixir is just tossed around without any formalized rules. There are no classes in Elixir, but their close relatives, “Structs”, do provide some of their utility.

Let’s formalize our code, making our intentions clearer and giving our friends some structure:

```

defmodule Friend do
  defstruct name: "Clara", age: 0
end

defmodule Immutability do
  def change_name(friend, new_name) do
    Map.put(friend, :name, new_name)
  end
  def get_friend do
    %Friend{}
  end
end

Immutability.get_friend |> IO.inspect

```

This looks better don't you think? Structs allow you to set defaults for your data and give it a prescribed structure. Elixir also gives you some shorthand for updating a struct:

```

#...

friend = Immutability.get_friend
%{friend | name: "Mike"} |> IO.inspect
IO.inspect friend

```

Once again, we get the same result as we did with **Map.put/2** above as the rebinding returns an immutable new **Friend** struct:

```

%{name: "Mike"}
%{name: "Clara"}
[Finished in 0.22s]

```

# TRANSFORMING DATA

If you're new to functional programming ideas than none of this is probably convincing. It takes a while to get into the functional groove, so let's write some more code.

Three things to focus on for this section are:

- We are transforming data by passing it through a set of functions
- We like smaller functions
- We can treat functions the same as values

The latter comes straight from Lambda Calculus. In fact all of this does - that's where functional programming has its roots in case that wasn't obvious. In addition, things work better overall if we focus on smaller, clearer functions.

To see this in action, let's do some math, shall we?

```
defmodule Ops do
  def square_it(num), do: num * num
  def double_it(num), do: num + num
  def root_it(num), do: :math.sqrt(num)
  def print_it(num), do: IO.inspect num
end

4 |> Ops.square_it
|> Ops.double_it
|> Ops.root_it
|> Ops.print_it

# 5.656854249492381
# [Finished in 0.274s]
```

What we have done here is fairly typical in the functional world. Small, concisely-named functions are arranged as needed, and we can simply pass some data along. This type of approach can work with anything, you just have to change your thinking a bit.

## Discussion: Why Would You Do Any of This?

Now, you might be thinking about how certain things might be solved with functional programming, maybe translating problems you work on every day? It's an interesting thing to ponder, but from my experience it takes a good two weeks to really hit that AHA! moment... at least for me.

What you need is a *reason to care about it all* to pull you through the process of changing your thinking. Let's see if I can help with that.

First: *no, functional programming is not perfect nor the answer to everything*, of course. But it does greatly expand your ability to think through a solution!

The first advantage of functional programming is its immutability. Quite a few bugs in OO are caused by the state of something not being correct, or what you wanted. You might have quite a few tests that show that yes, indeed, this code *should work* provided X, Y and Z conditions are met, but then condition  $\Sigma$  comes along and screws the whole thing up!

If you write code that depends on anything going on outside it, you're prone to these types of errors. It's the same reason most developers loathe global variables - one change and it ripples throughout your program, causing things to break.

Functional programming is different in that, ideally, a function should receive everything it needs to do its job, when asked. There are all kinds of safeguards you can attach to these functions to prevent them from being called if the data isn't correct. As opposed to being a burden, it's quite freeing! To get it right, however, you have to change your thinking, which takes time and effort.

Speaking of effort, let's get back to it. We'll pick up this conversation again later.

# A REAL EXAMPLE: A SHOPPING CART

Let's compare and contrast a functional style vs. an OO style, and I'll do that by creating a **ShoppingCart** in Elixir and also in JavaScript.

Let's start with JavaScript:

```
class Cart{
  constructor(){
    this.items = [];
  }
  addItem(item){
    //make sure it's a proper item
    //and then...
    this.items.push(item);
  }
}
const cart = new Cart();
cart.addItem({sku: "SOCKS", price: 12.00})
cart.addItem({sku: "MORESOCKS", price: 18.00})
```

Lovely. A very, very simple cart but it captures the idea: we have a class, an instance and we change the state of our instance by adding items to it.

Elixir is a bit different. As a first pass you might be tempted to do something like this:

```
defmodule Cart do
  items = []
  def add_item(item) do
    %Cart{items: item}
  end
end
```

Which wouldn't work. Functional languages don't have the notion of state, so holding on to an **items** array would cause an error. We need to pass everything to the **Cart** that it needs to operate, *including the items*:

```
defmodule Cart do
  def add_item(items, item) do
    items ++ item
  end
end
```

This works and, once again, underscores the notion of rebinding. Let's tweak this to look more like functional programming:



```

defmodule Cart do
  def add_item(items, item) do
    items ++ item
  end
end

items = []

Cart.add_item(items, %{sku: "SOCKS", price: 12.00})
|> IO.inspect

# %{sku: "SOCKS", price: 12.00}
# []

```

Much cleaner. Still a bit... *wonky* however. We're passing around arrays and maps without knowing what's going on. Let's clean this up a bit.

When you're forced to do without state and objects, you begin to think in terms of data and things that happen to that data. In our example we're using OO thinking by trying to represent a **Cart** as an object, which isn't very functional of us. Instead, we should think about it in terms of data moving through a *process*.

The **Cart** is our data, and our process is, more accurately, described as **Shopping**:

```

defmodule Cart do
  defstruct items: [], total: 0, count: 0
end

defmodule Shopping do
  #use pattern matching to guarantee data we need
  #the _ ignores the data, we just want the pattern
  def add_item(%Cart{} = cart, %{sku: _, price: price} = item) do
    %{cart | items: cart.items ++ [item], total: cart.total + price, count: cart.count + 1}
  end

  #get from the DB?
  def get_cart, do: %Cart{}

  #save to DB?
  def save_cart(cart), do: cart
end

```

Now, to add an item, we *transform* the **Cart** using the **Shopping** process:

```

# add an item to the cart
Shopping.get_cart
|> Shopping.add_item(%{sku: "SOCKS", price: 12.22})
|> Shopping.save_cart
|> IO.inspect

```

I threw a bit more Elixir at you in this go, hope you don't mind. Of note is how I'm able to use pattern matching to guarantee the data this function needs, which is a **Cart** and a map with a **sku** and a **price**.

From end to end, I'm able to retrieve a **Cart** from somewhere (assume it's a database for now), add an item to it and then put it

back. It's clear what this process is by reading the code and the only change of state is that of our database, which is acceptable in functional programming realms. Barely.

## SIDE EFFECTS AND PURITY

There are two terms that you often hear in discussions about functional programming: *purity* and *side effects*. Both terms stem from the idea of immutability.

The whole notion of interacting with a system outside the scope of the function you're in is called a "side effect" - something that happens as a result of your function being invoked. Working with a database, for instance, is referred to as a "necessary side effect" because you're changing the state of something outside the scope of your function.

The more you do this, the less "pure" your code is. *Purity* is not a term dedicated to functional programming - it refers to the level of interaction any code has with the outside world. You can write "pure" code in OO programming just as you can with functional, it just happens to have a little more focus in the functional world.

Functional programmers like purity. It's easier to test a function that doesn't change behavior based on some external setting or function call. It's also easier to debug. The downside is that you end up with more functions that are (typically) a lot smaller than what you might write in OO land.

There's a way to work with those, too, as we're about to see.

## CURRYING

Functions, functions, functions. They're everywhere! Organizing a program full of them can feel overwhelming, especially when you're just starting out with functional programming.

Let's look at one of the very first practices you'll want to take advantage of: *currying*. Currying is the act of using smaller, single arity functions in a chain rather than a larger function with multiple/complex arguments. It's easier to understand using code.

Consider date night with your partner:

```
const dateNight = (who, what, where) => {  
  return `Out with ${who} having fun ${what} at ${where}`;  
};
```

This function takes 3 arguments (or, in more programmy speak, has an arity of 3). If we split these functions into a smaller set of chained, single arity functions, we're currying:

```
const nightOut = who => what => where => {  
  return `Out with ${who} having fun ${what} at ${where}`;  
};
```

Looks a weird, doesn't it? Especially if you're not used to lambdas. But how might we use this function? Like this:

```
const funTime = nightOut("Dancing")("wife")("Club 9");
console.log(funTime);
//Out with wife having fun Dancing at Club 9
```

Does this look familiar to you? It should, this is the *exact* thing we did when applying values to functions in Lambda Calculus a few chapters back.

I know what you're thinking: *calling functions like this is a bit goofy*, and I would agree. This is where the idea of *partial application* comes in:

```
const dancing = nightOut("Dancing");
const dancingWithWife = dancing("wife");
const funTime = dancingWithWife("Club 9");
console.log(funTime);
//Out with wife having fun Dancing at Club 9
```

This is the power of currying. Let's see something a bit more concrete, however.

## A Curried SELECT Query

I worked with a very nice, very opinionated developer recently who loves her object-oriented programming. I was trying to explain how I thought it would be fun to add some more functional ideas to the

data access code we were writing, and she was rather resistant. Finally, she said “just show me what you mean”.

So I did. I started out with a curried function set for building a basic select query:

```
const selectQuery = table => (where, params) => order => limit => {  
  
};
```

At this point I simply needed to evaluate what was sent in to each function call:

```
const selectQuery = table => (where, params) => order => limit => {  
  const whereClause = where ? ` where ${where}` : "";  
  const orderClause = order ? ` order by ${order}` : "";  
  const limitClause = limit ? ` limit ${limit}` : "";  
  if(params.length > 0) params = [params];  
  const sql = `select * from ${table}  
    ${whereClause}${orderClause}  
    ${limitClause}`; //wrapped this for readability  
  return {sql: sql, params: params};  
};
```

I've written a lot of code for building SQL statements, and this is probably the smallest I've ever created. Using this is even more fun:

```
const usersQuery = selectQuery("users");  
console.log(usersQuery())();  
//{sql: "select * from users", params: []}
```

You probably wouldn't want to create a query in that way, however, and this is where partial application comes in. You could create a module (let's call it a repository for fun) which built up the following:

```
const usersQuery = selectQuery("users");
const allUsersQuery = usersQuery()();
const usersByEmail = email => usersQuery("email = $1", email)();
console.log(usersByEmail("test@test.com"));
//{sql: "select * from users where email = $1", params: ["test@test.com"]}
```

I think this is interesting, and a nice way to reuse functionality. Let's round this out by plugging in pg-promise and having it actually do something. I'll be using the Chinook test database for this:

```
const pgp = require('pg-promise')();
const db = pgp("postgres://localhost/chinook");
//own selectQuery as before
const selectQuery = table => (where, params=[]) => order => limit => exec => {
  const whereClause = where ? ` where ${where}` : "";
  const orderClause = order ? ` order by ${order}` : "";
  const limitClause = limit ? ` limit ${limit}` : "";
  if(params.length > 0) params = [params];
  const sql = `select * from ${table}${whereClause}${orderClause}${limitClause}`;
  const query = {sql: sql, params: params};
  //should we execute?
  if(exec){
    db.many(sql, params).then(res => exec(res)).catch(err => console.log(err))
  }
  //if not, just return the query
  else return query;
};
const albumQuery = selectQuery("album");
const albumSearch = title => albumQuery("title LIKE $1", `${title}%`)();
const rockAlbums = albumSearch("Rock");
//line it up!
rockAlbums(res => {
  console.log(res);
})
```

If you're playing along and run this code, you should see all the albums in the Chinook database with the term "Rock" in the title.

When I showed this code to my friend, her response was wonderful:

*...oh my god, that's equal measures ingenious and horrifying :D Being able to reuse selectQuery, albumQuery, etc. almost seems like a twisted take on models at an arbitrary scope...*

Twisted indeed. Personally, I dig it!

## A VERY BRIEF DISCUSSION ABOUT FUNCTORS AND MONADS

There's a great line from Douglas Crockford about monads:

*In addition to it being useful, it is also cursed and the curse of the monad is that once you get the epiphany, once you understand - "oh that's what it is" - you lose the ability to explain it to anybody.*

*From "Monads & Gonads", December 2012*

Very, very true. That's why I want to keep this discussion as brief as possible and invite you to do your own investigation. You should have a little bit of an itch in your brain right now, wondering *just*



*how am I supposed to orchestrate things beyond simply currying my way to insanity?*

This is where structures like *functors* and *monads* (among others) come in.

When you're slinging functions around as we have, you might want to have a higher level of abstraction for working with them. Something that might *wrap* those functions and handle certain situations for you the way you want.

One thing that you might want to do is to interrogate/iterate over some values, such as an array or a struct like a **user**. You could write a loop or do some type of recursion, or you could orchestrate the effort using a set of functions.

A popular way of doing this is to create a function which allows you to map values from a given object:

```
class Monkey{
  constructor(val){
    this.__value = val;
  }
  map(fn){
    return fn(val);
  }
}
```

This is simplistic, but hopefully you get the idea. This simple construct has a lofty name: it's a *functor*. It's only purpose in life is to run **map** over things that are *mappable*.

Functors have big brothers that will implement some logic on top of that mapping, namely returning some alternate values based on the data contained in your mappable object. For instance, you might want to access a **name** field on your **user** object, but don't want bad things to happen if there is no **name** field.

This is where monads come in, and where I start gracefully backing my way to the door. They're easy to understand in concept, and when you first encounter them, it can be a little anticlimactic. I don't know why this is. Monads really aren't terribly scary, but it's like some beautiful gateway to programming hell: if you understand it, you're doomed. I've found Crockford's rule above to be spot on.

Let's see if you agree with me.

Assume that we're allergic to **if** statements. This is one lovely oddity that happens when you start working with functional languages. There are so many interesting constructs and ways of doing things (like pattern matching) that you find yourself avoiding complex conditional trees.

Our **selectQuery** is pretty good and I feel happy about it, but we wouldn't be allowed into Functional Club with this code:

```

const selectQuery = table => (where, params=[]) => order => limit => exec => {
  const whereClause = where ? ` where ${where}` : "";
  const orderClause = order ? ` order by ${order}` : "";
  const limitClause = limit ? ` limit ${limit}` : "";
  if(params.length > 0) params = [params];
  const sql = `select * from ${table}${whereClause}${orderClause}${limitClause}`;
  const query = {sql: sql, params: params};
  //...
};

```

Monads exist to handle just this case! Let's create the simplest one: the **Maybe** monad:

```

class Maybe{
  constructor(val){
    this.__value = val;
  }
  isNothing(){
    return (this.__value === null || this.__value === undefined);
  };
  map(fn){
    return this.isNothing() ? Maybe.of(null) : Maybe.of(fn(this.__value));
  };
  val(){
    return this.isNothing() ? "" : this.__value;
  };
}
Maybe.of = (val) => new Maybe(val);

```

That's not so scary is it! See what I mean? Ah the temptation... let's see if we can confuse ourselves a bit.

This class exists to do one thing: *handle conditional values* in an orchestrated, functional way. When you call **map()** you say "here's a function, use it if there's a value". The trick, however, is that if there is a value for your monad then **map** hands you back *another*

**Maybe** *monad* with its value set as the result of your mapped function call.

This means we can move data through a chained set of functions without worrying about **null** or **undefined** values. How fun.

We can now use our **Maybe** monad to evaluate our **whereClause**:

```
const whereClause = Maybe.of(where).map(w => `where ${w}`).val();
```

If **where** has a value then our lambda will be called and we'll get back another **Maybe** with its current value set to a where clause. We can then call **val()** if we just want this value directly, which we do. If where isn't set, we'll get back an empty string.

We can apply this to the other arguments as well:

```
const whereClause = Maybe.of(where).map(w => `where ${w}`).val();
const orderClause = Maybe.of(order).map(o => `order by ${order}`).val();
const limitClause = Maybe.of(limit).map(l => `limit ${limit}`).val();
```

We could, at this point, build our SQL statement as before by building a string with template literals, but where's the fun in that! Let's use our **Maybe** monad one more time:

```
const sql = Maybe.of(`select * from ${table}`)
  .map(sql => `${sql} ${whereClause}`)
  .map(sql => `${sql} ${orderClause}`)
  .map(sql => `${sql} ${limitClause}`)
  .val();
return {sql: sql, params: params};
```

Let's step through this. We start out with an initial value that we want to work with, in this case it will be our **select \* from \${table}** statement. We move this value through a successive chain of functions, where it is applied to a function that is passed in via map. That value is then passed on to the next in the chain. If you're thinking that this looks a lot like Elixir's `|>` (pipe) operator, you'd be correct.

*Note: if you're a functional programming person and you see a better way to do this, please do let me know!*

Here's the final code:

```
const selectQuery = table => (where, params=[]) => order => limit => {

  const whereClause = Maybe.of(where).map(w => `where ${w}`).val();
  const orderClause = Maybe.of(order).map(o => `order by ${o}`).val();
  const limitClause = Maybe.of(limit).map(l => `limit ${l}`).val();

  const sql = Maybe.of(`select * from ${table}`)
    .map(sql => `${sql} ${whereClause}`)
    .map(sql => `${sql} ${orderClause}`)
    .map(sql => `${sql} ${limitClause}`)
    .val();

  return {sql: sql, params: params};
};

const albumQuery = selectQuery("albums");
const albumSearch = title => albumQuery(`title LIKE $1`, `%${title}%`);
const rockAlbums = albumSearch("Rock");
console.log(rockAlbums());

//{
//  sql: 'select * from albums where title LIKE $1 ',
//  params: '%Rock%'
// }
```

Wahoo!

# DATABASES

There is an established way to design a transactional database: following the rules of normalization. This is the process of essentially turning a single, large spreadsheet into a set of related tables.

Transactional systems can back our applications, but analytical systems power decision-making. Data warehouses and data marts store the data and OLAP systems provide the analysis infrastructure.

Distributed database systems use the notion of *horizontal* scaling (adding more machines) vs *vertical* scaling (increasing machine size and power). Distributed systems are much different from traditional “big machine” databases in that they have to balance trade-offs with data consistency, availability, and network problems.

Social media is driving the need to store gigantic, petabyte-sized data stores. This movement is called Big Data and it’s often difficult to grasp the sheer size of it all.

# NORMALIZATION

A relational database consists mainly of a bunch of tables. These tables contain rows of data, organized by columns. There must be a method to this madness, and it's called normalization.

Database normalization is all about controlling the size of the data as well as preserving its validity. Back in the 70s and 80s you simply did not have disk space, but companies did have a ton of data that would fill it up. Database people found that they could reduce the size of their database and avoid data corruption if they simply followed a few rules.

Before we get to these rules, let me just add upfront that it's truly not that complicated once you grasp the main ideas. As with so many things computer science related, the jargon can be rather intense and off-putting, making simple concepts sound hard. We'll slowly work our way up to it.

Finally: rules are meant to be broken. In fact a DBA will break normalization quite often in the name of performance. I'll discuss this at the end.

## **First Normal Form (1NF): Atomic Values**

We've decided to buy a food truck and make tacos and burritos for a living. We have our ingredients and drive up to our favorite street corner, opening our doors for business.

When the orders come in we put them into a spreadsheet – knowing we’ll need to deal with it later on:

email	name	order_id	items	price
joe@example.com	Joe Tonks	1	Pollo Burrito, Diet Coke	\$12,50
jill@example.com	Jill Jones	2	Carne Asada, Sprite	\$14.50

Let’s move this spreadsheet into the database, altering it as we go.

First normal form (1NF) says that values in a record need to be atomic and not composed of embedded arrays or some such. Our items are not atomic – they are bunched together. Let’s fix that.



EMAIL	NAME	ORDER_ID	ITEMS	TOTAL
JOE@ EXAMPLE.COM	JOE TONKS	1	POLO BURRITO DIET COKE	\$10.50
JILL@ EXAMPLE.COM	JILL JONES	2	CARNE ASADA SPRITE	\$14.50



EMAIL	NAME	order_id	items	total
Joe@ EXAMPLE.COM	Joe Tonks	1	Polo Burrito	\$ 8.00
Joe@ EXAMPLE.COM	Joe Tonks	1	DIET COKE	\$ 2.50
Jill@ EXAMPLE.COM	Jill Jones	2	Carne Asada	\$12.00
Jill@ EXAMPLE.COM	Jill Jones	2	Sprite	\$2.50

Lovely. Our new orders table is 1NF because we have atomic records. You'll notice that things are repeated in there, which we'll fix in a bit. Our next task is to isolate the data a bit.

## Second Normal Form (2NF): Columns Depend On a Single Primary Key

Now that we're in 1NF, we can move on to 2NF because part of being in 2NF is complying with 1NF. Our task to comply with 2NF means that we need to identify columns that uniquely define the data in our table.

An order is a customer buying something. In our case the email field uniquely identifies a customer, and the **order\_id** field uniquely identifies what they've ordered. You put that together and you have a sale – which could uniquely identify each row in our table.

The problem we have is that name does not depend on the **order\_id** and items and price have nothing to do with email – this means we're not in 2NF. To get to 2NF we need to split things into two tables:

email	name	order_id	item	total
JOE@EXAMPLE.COM	JOE TONES	1	Pollo Burrito	\$ 8.00
JOE@EXAMPLE.COM	JOE TONES	1	Diet Coke	\$ 2.50
JILL@EXAMPLE.COM	JILL JONES	2	Carne Asada	\$ 12.00
JILL@EXAMPLE.COM	JILL JONES	2	Sprite	\$ 2.50



email	name
JOE@EXAMPLE.COM	JOE TONES
JILL@EXAMPLE.COM	JILL JONES

order_id	customer	item	price
1	JOE@EXAMPLE.COM	Pollo Burrito	\$ 8.00
1	JOE@EXAMPLE.COM	Diet Coke	\$ 2.50
2	JILL@EXAMPLE.COM	Carne Asada	\$ 12.00
2	JILL@EXAMPLE.COM	Sprite	\$ 2.50

Much better, but not quite there. Our customers table looks good, but our orders table is a bit of a mess.

## **Third Normal Form (3NF): Non-keys Describe The Key And Nothing Else**

3NF says that every non-primary field in our table must describe the primary key field, and no field should exist in our table that does not describe the primary key field.

Our orders table has repeated values for the key (which is a no-no) and the price of each item has nothing to do with the order itself. To correct this, we need to split the tables out again:

email	name
JOE@EXAMPLE.COM	JOE TONKS
JILL@EXAMPLE.COM	JILL JONES

order_id	customer	item	price
1	JOE@EXAMPLE.COM	PIZZA BURRITO	\$ 8.00
1	JOE@EXAMPLE.COM	DIET COKE	\$ 2.50
2	JILL@EXAMPLE.COM	CARNAL ASADA	\$ 12.00
2	JILL@EXAMPLE.COM	SPRITE	\$ 2.50



3NF

PRODUCTS	SKU	PRICE	NAME
	PB	\$ 8.00	PIZZA BURRITO
	DC	\$ 2.50	DIET COKE
	CA	\$ 12.00	CARNAL ASADA
	S	\$ 2.50	SPRITE

ORDER_ITEMS	ID	ORDER_ID	SKU
	1	1	PB
	2	1	DC
	3	2	CA
	4	2	S

CUSTOMERS	ID	EMAIL	NAME
	1	JOE@EXAMPLE.COM	JOE TONKS
	2	JILL@EXAMPLE.COM	JILL JONES

ORDERS	ID	CID	DATE
	1	1	04/03/2016
	2	2	04/03/2016

We now have 4 tables to support our notion of an order:

- **customers** holds all customer data
- **orders** holds metadata related to a sale (who, when, where, etc.)
- **order\_items** holds the items bought
- **products** holds the items to be bought

You'll notice, too, that I replaced email with an integer value, rather than using the email address. I'll explain why in just a second.

## **In The Real World: The Normalization Process**

Normalizing a database requires some practice. As programmers, hopefully you understand how to model classes and objects. It's almost the same process that we just went through: *what attributes belong to which concept?*

It's at this point that I get to tell you (with a sinister giggle) that the rules of normalization are more of a guideline, not necessarily law. A well-normalized database may be theoretically sound, but it will also be somewhat hard to work with.


We managed to move a fairly simple spreadsheet with 2 rows and 5 columns into a 4-table structure with 3 joins and multiple columns! We only captured a very, very small fraction of the information available to us and our Taco Truck.

It's very easy to build a massively complex database with intricate lookups, foreign keys, and constraints to support what appear to be simple concepts. This complexity presents two problems to us, right away:

- Writing queries to read and write data is cumbersome and often error-prone
- The more joins you have, the slower the query is

The bigger the system gets, the more DBAs tend to cut corners and denormalize. In our structure here it would be very common to see a total field as well as **item\_count** and embedded customer information.

To show you what I mean – have a look at the structure for the StackOverflow posts table:

Name	Type	Length	Decimals	Dimen...	Not Null	Key
id	int4	0	0	0	<input checked="" type="checkbox"/>	
post_type_id	int4	0	0	0	<input type="checkbox"/>	
accepted_answer_id	varchar	255	0	0	<input type="checkbox"/>	
parent_id	int4	0	0	0	<input type="checkbox"/>	
created_at	varchar	255	0	0	<input type="checkbox"/>	
deleted_at	varchar	255	0	0	<input type="checkbox"/>	
score	int4	0	0	0	<input type="checkbox"/>	
view_count	varchar	255	0	0	<input type="checkbox"/>	
body	text	0	0	0	<input type="checkbox"/>	
owner_user_id	int4	0	0	0	<input type="checkbox"/>	
owner_display_name	varchar	255	0	0	<input type="checkbox"/>	
last_editor_id	varchar	255	0	0	<input type="checkbox"/>	
last_editor_display_name	varchar	255	0	0	<input type="checkbox"/>	
last_edit_date	varchar	255	0	0	<input type="checkbox"/>	
last_activity_date	varchar	255	0	0	<input type="checkbox"/>	
title	varchar	255	0	0	<input type="checkbox"/>	
tags	varchar	255	0	0	<input type="checkbox"/>	
answer_count	varchar	255	0	0	<input type="checkbox"/>	
comment_count	int4	0	0	0	<input type="checkbox"/>	
favorite_count	varchar	255	0	0	<input type="checkbox"/>	
closed_date	varchar	255	0	0	<input type="checkbox"/>	
community_owned_date	varchar	255	0	0	<input type="checkbox"/>	

Notice the **view\_count**, **owner\_display\_name**, and **\_count** fields? Also the **last\_editor\_** field?

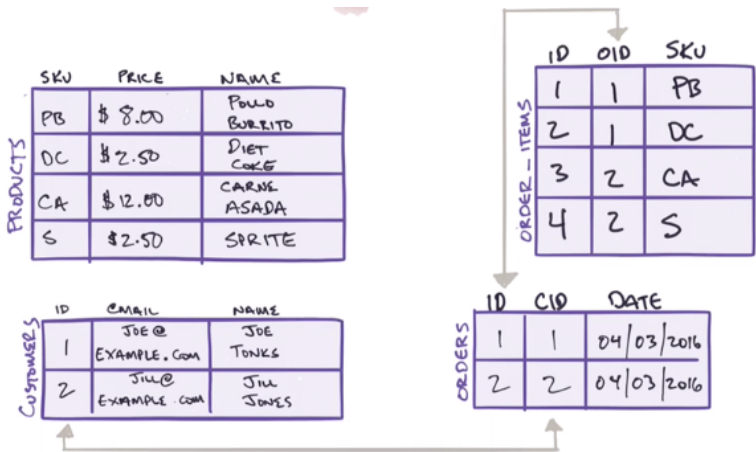
The count fields are calculated and don't belong in a table, theoretically speaking. The **owner\_display\_name** and **last\_editor\_** fields

should be foreign keys that link to an **authors** or **users** table of some kind – and they do with **last\_editor\_id** and **owner\_id**. Querying this massive table using the required joins, however, would be way too slow for what they need.

So, they denormalized it. Many businesses do – it just makes things faster and simpler.

### More Real World: Is This Schema Correct?

Let's take a look at the final schema we came up with:



While it is theoretically correct, there is a problem with being historically correct. For instance, if you come to my Taco Truck and buy some Carne Asada, I'll have a record of it stored happily in my orders table.



When I run my sales queries at the end of the month, your sale will be in there, adding \$12.00 to the total. In July of this year, I have \$6800 in total sales! Wahoo!

Sales have gone well and being a good capitalist I decide I'm going to charge \$15.00 for Carne Asada from now on. I'm proud of myself and so I run July's sales reports one more time so I can print them out – I want to see that money rolling in!

Hmmm. The numbers are off for some reason. It used to say \$6800 for July, but now it says \$7300! What happened?

We've made a rather critical mistake with our design, here. One that you see constantly. The deal is that **order\_items** is what's known as a "slowly changing historical table". The data in this table is not transactional, it's a matter of record.

So, what do you do if you want to avoid changing the past? We'll discuss that in the next section.

## OLAP AND OLTP

99% of the databases you and I work in are considered "OLTP": On-line Transaction Processing. This type of system is based on performance – many reads, writes and deletes. For most applications this is appropriate.

At some point, however, you're going to want to analyze your data, which is where "OLAP" comes in: *Online Analytical Processing*. These

systems are low-transaction systems that change little, if at all, over time apart from nightly/weekly loads. These systems power data warehouses and support data mining.

The structure of each system varies quite a lot. OLTP systems are relational in nature and are structured using the rules of normalization discussed in the last chapter.

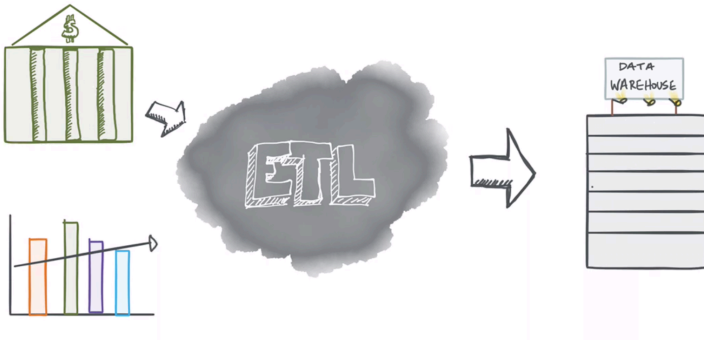
OLAP systems are heavily denormalized and are structured with dimensional analysis in mind. Building these systems can take hours and usually happens on a nightly basis, depending on the need.

Let's start where OLTP ends and OLAP begins...

## EXTRACTION, TRANSFORMATION, AND LOADING (ETL)

My accountant, who's also my best friend, has the same thing to say to me every year when preparing my taxes: "trash in, trash out". That's his warning to me as I prepare my account statements to bring to him.

This is a form of ETL that people do every year (especially in the US): pull all their financial data from their banks, savings, investments etc. and compile it in a single place – maybe Excel. They go through it at that point, making sure it all adds up. Each bank statement reconciles and there are no errors.



You do the same with analytical systems. The first step is to extract the information you want from your OLTP system (and/or other sources) and comb through it for any errors. Maybe you don't want null sales totals, or anything tagged "test".

You then transform as required. Reconciling customer information with your CRM system so you can add history data, account numbers, location information, etc.

Finally, you load the data into your system, which is usually another database that has a special layout. The system I'm most familiar with (and spent years supporting) is Microsoft's SQL Server Analytical Services (SSAS) so I would usually extract the data from one SQL Server database to another.

They also had a built-in transformer that worked with VBScript, of all things! I used it sometimes but often it would fail. We later moved to a system called Cognos that was a gigantic pile of XML pain.

Today, you can perform quite complicated ETL tasks efficiently and simply by using a set of simple scripts. These can be as simple as shell scripts or, more commonly, quite complex using a programming language like Python or Ruby. Python's speed and popularity make it a very common choice for ETL.

## DATA MARTS AND WAREHOUSES

You'll often hear these terms used interchangeably, but they're two very different things. A data warehouse is like a filing cabinet in your office or at home where you keep all your financial information: statements, tax documents, receipts, etc. Hopefully you keep this organized so it's easy to sift through and put in a form that your accountant can understand – such as an Excel spreadsheet.

There are other things we can do with this data; maybe we want to know how much we spent on groceries so we can budget properly next year. We might want to calculate how much our life savings could be if we had any ... stuff like that. For now, however, we need to get some data to our accountant because it's tax season here in the US, so we load the data into Excel with a targeted use case: Accounting.

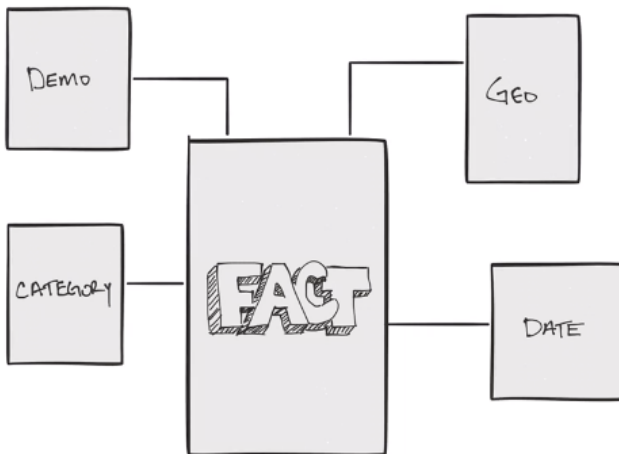
This is a data mart. A place (typically focused/targeted) that can answer questions. We could use the Data Warehouse for this (sending the accountant our shoebox full of receipts and state-

ments) but that would take her a long time and she wouldn't be happy.

## Data Mart Schemas

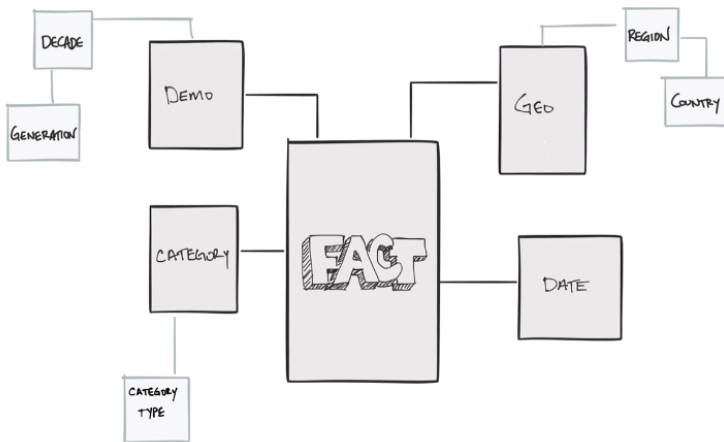
The Excel spreadsheet is an apt way to think about how data is stored in a data mart: flattened. You might have a few joins in there, but the fewer the better because processing the data mart, which is typically millions and millions of records, will slow down with more joins.

What does this look like, however? The one you'll see most often is the star schema:



In the center is a table called “the fact table” which represents a single fact you want to report on. For my accountant this would be a singular transaction of some kind: deposit, debit, adjustment, etc.

A *snowflake* schema is the same, but the dimension tables themselves have more dimensions.



## Dimensions

The fact table has keys which link off to dimensional look up tables, which you can use to roll up the data for specific queries. They’re called dimensions because they present a different way of rolling up the data. For sales (or anything relating to people) these are typically:

- Categories of some kind
- Time (week, month, quarter, year)
- Geography (city, state, country)
- Demographic (gender, age, race)

Selecting dimensions is much harder than it seems. They need to be fundamentally different from each other, or you'll be rolling up on data that has crossover meaning.

You see this sometimes with schemas that confuse demographic data with geographic data – typically with the region that a person is from. I had an hour-long discussion with a client once about the meaning of “Southerner” in their data.

It might not seem like a big deal but making sure the data can be cross-checked is absolutely critical.

With a data mart it's possible (and common) to query on multiple dimensions at once. If we had left “Southerner” as a bit of demographic information, we would have had conflicting questions and answers:

- Show all sales for both men and women located in the Southern United States
- Show all sales for both men and women who are “Southerners”

I have friends in Hawaii who call themselves “Southerners”. I have Hawaiian friends who live in Louisiana. What are we learning with these questions? *Analytics is difficult*. The point is: pick your dimensions with care and make sure you involve the people who are using the reports you’ll generate.

## Bad Dimensions

The “Southerner” problem (as it became known) is intangible, and it takes some experience with data to be able to spot reporting issues like that.

Others are far easier to spot – such as “double-labeling”, which happens all the time and is infuriating.

As a programmer I hope you have a blog where you share your ideas. If you do, it’s likely you have a way of tagging your posts with small, contextual keywords (tags).

Let’s do a counting query to find out how many comments your blog has for the tag **opinion** vs the tag **humor** (if you have such things... if not let’s pretend). It’s a simple enough query because, as it turns out, you only have 3 posts with 5 comments apiece:

- “Data Analysis is Silly” tagged opinion; 5 comments
- “Hadoop Honeybadger” tagged opinion and humor; 5 comments
- “A DBA Walks Into a Bar...” tagged humor; 5 comments



So, you run these queries:

```
select count(1) as comment_count from posts
inner join comments on post_id = comments.id
inner join posts_tags on posts_tags.post_id = posts.id
inner join tags on posts_tags.tag_id = tags.id
where tags.tag = 'humor'

--comment_count
---
--10

select count(1) from posts
inner join comments on post_id = comments.id
inner join posts_tags on posts_tags.post_id = posts.id
inner join tags on posts_tags.tag_id = tags.id
where tags.tag = 'opinion'

--comment_count
---
--10
```

Simple enough. But then you remember reading *The Imposter's Handbook* which mentioned cross-checking your rollup queries for accuracy, so you do:

```
select count(1) from comments;

--comment_count
---
--15
```

*Uh oh.* 10 humor posts + 10 opinion posts does not equal 15!

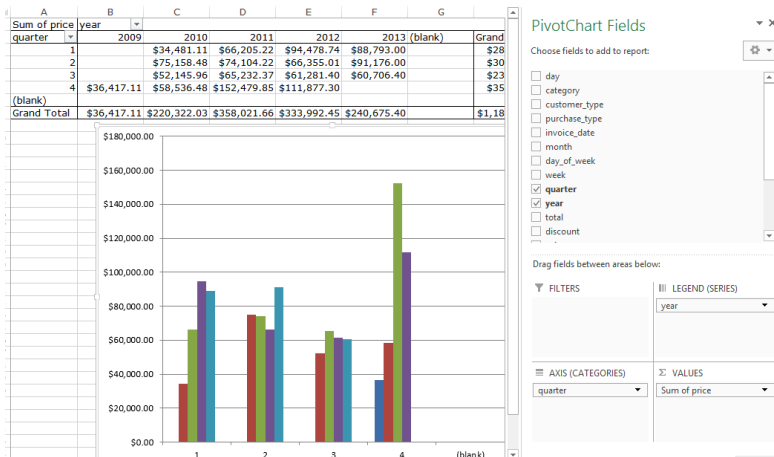
Now you might be thinking “of course it doesn’t” and that cross-checking like this is not accurate! My answer to that is “tell it to the product specialist who wants a sales rollup on various product tags”.

Right now, across the world, sales reports are in this “semi error” state. You cannot do rollup queries that involve many to many categorizations and expect to keep your job. Even if you add warnings! The numbers will suggest a reality that’s not there.

By the way: cross-checking like this is all part of ETL. Bad data should never make it into your data warehouse/data mart.

## ANALYZING A DATA MART

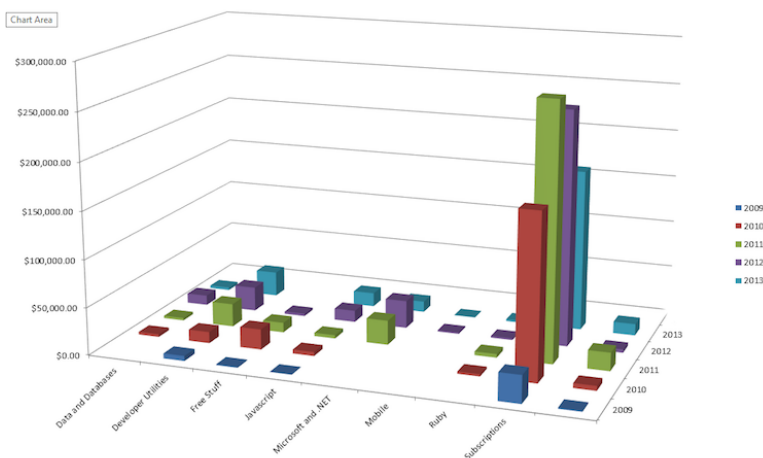
OK, you’ve gone through your data and have decided it’s clean of weirdnesses (good job!) and imported it into your data mart. How do we analyze it? The simplest way is with a Pivot Table and/or a Pivot Chart. It’s likely you’ve seen these in action – here’s some sample data in Excel:



The idea with a pivot table is that you can move dimensions around, rolling your facts up in various interesting ways.

The axes of the graph is a perfect example of why a dimension is called a dimension: the x dimension is often time and the y dimension is often the sum of sales.

What if you wanted to visualize sales by category over time? You just need to add another dimension to the graph – and thank goodness most people in the world can understand things in three dimensions:



Your boss likes this report a lot! It's interesting to give your data a "surface" as we're doing here because, in a way, you can feel what's going on. Now your boss wants to see this data with some demographic information – for instance the buying patterns between men and women.

That requires a fourth dimension. How in the world would you do that! Well, without getting into a physics discussion – you can treat time as a fourth dimension – which can work really well. Unfortunately for me, this book only works in two dimensions (with the illusion of a third), so I can't show you a moving time-graph ... but close your eyes and see if you can imagine a three-dimensional graph slowly changing over time...

The neat thing is that time is that one of your dimensions so you can lift that to the fourth axis and watch sales by category and gender change slowly.

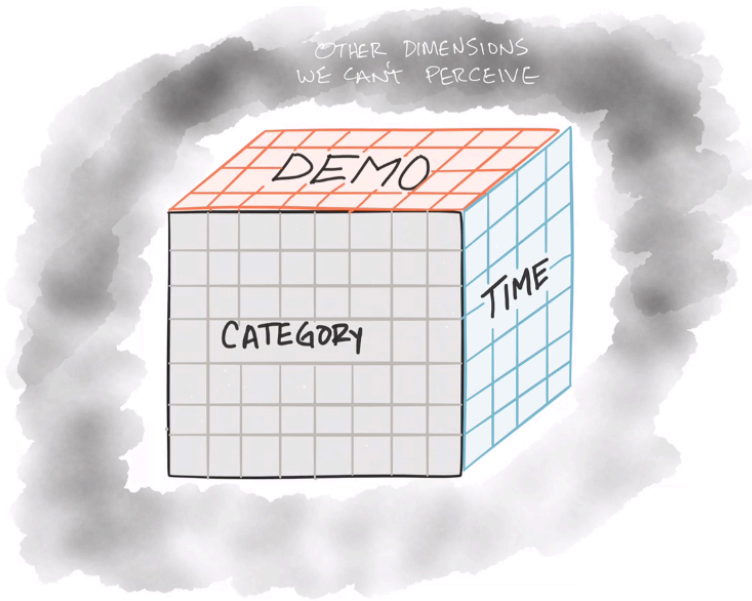
If your boss asks for more axes on this report you need to ask for a raise.

## USING AN OLAP CUBE

Pivot tables work well over data structured in a flat way. If you have more than a few thousand rows, however, things can get mighty slow.

This is where a structure called an OLAP Cube comes in. You tell it about your fact table, the dimensions you're using and their hierarchy, and then begin processing.

An OLAP cube is simply a bunch of pre-calculated data. It's called a "cube" because the data is described typically in three dimensions, and as I mention above people can't really conceive more than four dimensions anyway. Time is usually one dimension, some type of categorization is another, and customer demographic is usually the third. Any more than that and things just get weird.



When you view data along a dimension, you're viewing a slice of the cube and you usually do this with a pivot table of some kind. Excel, for example, will hook up to an OLAP cube.

Pre-calculating data like this makes OLAP cubes very fast at preparing ad-hoc reports over millions of rows of historical data, but that comes at the cost of preprocessing the data. For this reason, data marts that act as the source of an OLAP cube should be structured in a very specific way ... and this will sound counterintuitive.

## **Fact Tables and Indexes**

Your fact table should not have a primary key or indexes of any kind, for that matter. Inserting data into a table with an index

means the database needs to update the index whenever data is inserted, which takes time.

Ideally, you've already vetted and cleaned your data during ETL and you trust it – so no keys or indexes. A fact table can grow to billions of records – can you imagine the index sizes on that!

## **Favor a Star Schema**

Joins are slow, so denormalize your dimensional look up tables for speed of processing. Building an OLAP cube with millions of facts can take hours depending on the number of dimensions you're rolling up on.

Date rollups are the easiest thing to contain. For instance, your boss might think she wants a weekly sales report – but that's adding 52 additional slices to the OLAP structure – and every other dimension will need to be pre-calculated based on those 52 weeks times however many years.

This will move a three-hour processing run to an overnight run easily. So push back, if possible, or consider building an additional cube.

# DISTRIBUTED DATABASE SYSTEMS

It's 2008 (or thereabout) and you're realizing that processors aren't really getting any faster. Buying more RAM used to solve all kinds

of problems but lately you're finding that 12G is really all you need; the processor has become the bottleneck for your database.

In 2010 you had two processors. In 2012 you have four – all for the same price. Today if you pay enough money, you can have 32 ... THIRTY TWO CPU cores on that lovely machine in the sky.

Can your database take advantage of each of those cores? THAT is the question for this chapter.

Multiple CPUs means that many things can be processed at once. This means the software has to be able to *do things in parallel* without freaking out. Parallel processing is not a simple topic – especially concerning data.

Imagine 3 things happening in parallel, each on a different core:

- user 3002 changed their password
- user 3002 updated their profile picture
- user 3002 canceled their account

What happens first? Is there a priority here ... and if so, what is it based on? What happens if core #1 goes offline for 30 milliseconds because of network trouble?

Very Smart People have focused specifically on these problems over the last 10 or so years and come up with some interesting ways of doing things. **Parallel processing is where things are going** because that's where the hardware is taking us.



# A SHIFT IN THINKING

When computer science people tried to figure out data storage back in the 70s and 80s (aka: databases), they did so with two primary constraints in mind:

- Storage capacity: hard drives were not cheap so they had to focus on ways of storing data that would be extremely efficient with hard drive space. This led to column names like "UUN1" and hard-core adherence to normalization.
- Memory and processing speed: computers were simply slower, so storage needed to be optimized for read efficiency as well as overall size.

This is what most developers (including myself) "grew up" with. You used a relational engine to store data and you created your tables, keys, etc. in a very particular way.

NoSQL systems have been around since the 60s, but it was only in the late 90s that the development community really started to pay attention. Then, right around 2010, big software (Amazon, Facebook, Google, etc.) began to see the advantages of using NoSQL systems.

There was one advantage, however, that stood out above the rest: distribution. Simply put: **it's easier (technically and economically) to build distributed databases with a NoSQL system than it is to run a few, gigantic servers with oceans of RAM and disk**

**space.** Smaller servers are cheaper, and you spread your risk, mitigating data loss and disaster recovery.

You can scale horizontally with relational systems such as PostgreSQL and SQL Server – the problem, however, is that these systems need to remain ACID Compliant, which is a problem in distributed systems. They don't like to work in a parallelized way.

ACID-compliance means that you have certain guarantees when writing data in a transaction. In summary form, each transaction will be:

- **Atomic.** This means that a single transaction happens, or it doesn't. There is no concept of a "partial" transaction
- **Consistent.** The entire database will be aware of a change in the data whenever a transaction is completed. In other words: the state of the database will change completely, not partially.
- **Isolated.** One transaction will not affect another if they happen at the same time. This has the basic appearance of transactions being queued in a single process.
- **Durable.** When a transaction concludes it concludes. Nothing can change the data back unless another transaction changes the data back to the way it was. In essence: there is no undo.

Distributed systems are much different from this and rely on a different rule set entirely.

# CAP THEOREM

In 1998 Eric Brewer theorized that distributed processing of any kind can only provide two of the following three guarantees at any given time:

- **Consistency.** The same meaning as with ACID above; the state of the database will change with each transaction.
- **Availability.** The distributed system will respond in some way to a request.
- **Partition tolerance.** A distributed system relies on a network of some sort to function. If part of that network goes offline (thus “partitioning” the system), the system will continue to operate.

So far, this has proven to be true. Sort of. In 2012 Brewer wrote a followup which suggested “picking two of three” can be misleading:

*...the “2 of 3” view is misleading on several fronts. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of*

*consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.*

Modern distributed database systems are addressing exactly this. RethinkDB is a prime example (full disclosure: it's one of my favorite distributed databases and I love it. Further disclosure: during the writing of this book the company behind RethinkDB closed its doors. The project, however, is open source and will continue).

You can choose the level of consistency you want on a per table or per query basis. Meaning that you choose whether you want an ack (acknowledgment of write to the entire system) or you can just trust the system to do it when it gets around to it.

In addition, you can architect your database on a table-by-table basis to enhance which of the three you want.

This can be really confusing, so let's dive into each of these ideas (as well as the jargon for each) using RethinkDB as an example system (because it's what I know).

## **Enhancing A and P with Eventual C**

You've heard of eventual consistency, it's a buzzword that makes many ACID-loving, relational DB people freak out. I was one of them.

The idea is a simple one: a write operation (think of it as an insert into query) is handed to the system and you receive an ack imme-

diately, meaning that the system has received it and will write it to disk when it can.

At this point (before the write hits the disk), the database is not consistent. You can visualize this by thinking of a database with three nodes. One node receives the write request, which means the other two nodes are not consistent.

This inconsistency is risky. If the power goes out for some reason, the write will be lost, which could be a bad thing depending on the data.

The benefit, however, rests squarely on the benefits of distributed systems in the first place: parallel processing. The system is available to handle more operations, so many things can happen at the same time, making the system extremely fast.

If the node handling the write goes offline (due to a netsplit, or network partition) it doesn't matter (assuming it still has power) because the write is queued, and will remain queued, until the rest of the system is brought back online and consistency is achieved.

These systems are called "AP systems" (generally) and are built to handle gigantic loads with guaranteed uptime. Facebook's Cassandra and Riak from Basho are prime examples of these systems.

# AN ALTERNATIVE TO ACID: BASE

Systems that focus on availability and partition tolerance comply with BASE:

- Basically Available
- Soft state
- Eventually consistent

If you've grown up with ACID systems, as I have, this idea might sound nightmarish. ACID is all about data paranoia: *protect it at all costs*. Make sure it's written to disk in isolation.

BASE, on the other hand, is the opposite of paranoid – which kind of makes sense when you consider the idea of strength in numbers: the more machines, the wider the risk is spread.

ACID systems, on the other hand, are typically “big machine” systems. You scale these systems up (bigger hardware) as opposed to out (more hardware).

The problem for ACID, however, is there is only so big you can get. BASE systems can scale massively simply by adding more nodes. I'm not saying this is easy by any stretch, running these systems is very complicated – but it's doable and is being done in very large companies. We'll discuss this more in the chapter on Big Data.

## Assessing and Mitigating The AP Risk

You're in a meeting where the CTO has just announced that your fictional company, Red:4 Aerospace, is moving its orbiter telemetry data over to a distributed system. She's heard about CAP and needs a risk analysis – and she's looking at you.

What do we tell her in our analysis?

The first thing is that we'll gain a ton of processing power, which is good. Customers might have to wait a few extra seconds (depending on load), but if things get too slow we can just add a few more nodes!

This has the dual advantage of mitigating a netsplit. If we're strategic about our nodes and data centers we should be able to survive just about any outage.

But... what about the data? *There is the possibility of data loss*, always, when you make the AP tradeoff. Losing data can mean losing business (or worse) and to people (like me) who live and breathe the idea of Good Data, this is a hard subject to muse on clearly.

This is where we begin staring out the window as we remember various Hacker News threads on the unreliability of MongoDB which, in many developers' minds, means all distributed NoSQL systems. We remember the many blog posts we've read that ultimately resolve to "we lost data because we did NoSQL wrong"...

## **Giving Into Paranoia By Leaning On C**

You've turned in your report on AP systems and the CTO now wants to know what other options are. Most (if not all) of the distributed database systems out there support partition tolerance well – it's just a matter of choosing availability or data consistency.

Do you want your system to stay up? Or the data to be correct? RethinkDB and MongoDB lean towards the latter – they are CP systems. By default, RethinkDB will only acknowledge a transaction when the data is persisted to disk.

Both MongoDB and RethinkDB are configurable so that you can tweak consistency settings the way you want and need. You can make some tables more AP if you want as the data allows.

The more you get into the nuances of CAP and how modern NoSQL databases handle it, the more confusing things get. Rapidly. As with every topic in this book I could fill chapters and chapters with detail. Instead, I'll leave the CAP discussion here and suggest you read more about how RethinkDB and MongoDB are put together.

It's time to turn our attention to the mechanisms for tweaking availability and consistency.



## **Best Of Both Worlds: Applying A and C Where Needed**

Distributed databases specialize in handling very large volumes of data. They do this by “spreading the load” between individual database nodes.

### **Not Only NoSQL**

I’ve been focused on NoSQL systems like Cassandra, MongoDB and RethinkDB but relational systems can be clustered into a distributed system as well. Namely my favorite database engine: PostgreSQL. This is how Instagram does it and they’ve written extensively about maintaining a PostgreSQL distributed cluster.

The rest of this section is about distributed systems in general, unless otherwise specified.

Your CTO is quite happy that the distributed system chosen by the company can handle both AP and CP, and she wants you to come up with strategies for data coming from the orbital probe that’s due to arrive in orbit around Jupiter 8 months from now.

Let’s shape our distributed system using sharding and replication.

## Increasing A With Sharding

We will have telemetry data coming in at a very high rate, and we need to process this data continually to be sure our orbital calculations are correct and that our probe is where it's supposed to be.

A fast database can hold most *current data* in RAM. By current data I mean the stuff that we care about. With a demo database, this might come to a few megabytes.

The data generated by most consumer-focused websites remains in the < 1Gb realm, which means that sharding/replication will have little effect.

Another good friend of mine, Rob Sullivan is a PostgreSQL DBA who fields some very interesting questions from developers he runs into at conferences and cafes. Recently he was asked how he would suggest sharding a database system that just hit 5Gb total data.

His answer (paraphrased):

*... they were trying to do all of this on the cheapest Heroku database possible, and having issues because they didn't want to pay for the appropriate tier... talking themselves into a complete rearchitecture because of a price list...*

There is, and will always be, confusion about when and mostly if you should shard your database.

# Knowing When To Shard

The outcome: when you run out of RAM and it's cheaper to add a machine vs. scaling up to a bigger VM or server.

Let's take a closer look at this.

Here's a price breakdown for Digital Ocean as of summer, 2016:

Choose a size					
<b>\$5/mo</b> \$0.007/hour	<b>\$10/mo</b> \$0.015/hour	<b>\$20/mo</b> \$0.030/hour	<b>\$40/mo</b> \$0.060/hour	<b>\$80/mo</b> \$0.119/hour	<b>\$160/mo</b> \$0.238/hour
512 MB / 1 CPU 20 GB SSD Disk 1000 GB Transfer	1 GB / 1 CPU 30 GB SSD Disk 2 TB Transfer	2 GB / 2 CPUs 40 GB SSD Disk 3 TB Transfer	4 GB / 2 CPUs 60 GB SSD Disk 4 TB Transfer	8 GB / 4 CPUs 80 GB SSD Disk 5 TB Transfer	16 GB / 8 CPUs 160 GB SSD Disk 6 TB Transfer
<b>\$320/mo</b> \$0.476/hour	<b>\$480/mo</b> \$0.714/hour	<b>\$640/mo</b> \$0.952/hour			
32 GB / 12 CPUs 320 GB SSD Disk 7 TB Transfer	48 GB / 16 CPUs 480 GB SSD Disk 8 TB Transfer	64 GB / 20 CPUs 640 GB SSD Disk 9 TB Transfer			

The price of each machine goes up according to the RAM. Double the RAM, double the price. So here's the question: *would bumping to the top instance they have (\$640/month) be the same as 4 of the \$160/month?*

In short: **no**. A single machine is simply easier to maintain, especially when it comes to databases. You're much better off just upgrading until you can't – let's see why.

# SHARDING

The top-of-the-line DigitalOcean VM has 64G of RAM, most of which you can probably use for your database cache. You also have 20 CPU cores at your disposal: this is a fast machine.

Unfortunately, the telemetry data is projected to have a current data size of about 250Gb – this is data we'll need to actively write and query extremely quickly.

We don't have access to a machine with this much RAM at our provider, so we'll need to distribute the load across multiple machines. In other words: shard.

There are two ways we can do this with modern databases:

- **Let the machine decide how.** Modern databases can use the primary keys to divide the data into equal chunks. In our case, we might decide to go with 6 total shards – so our database system will divide the primary keys into groups of 6 based on some algorithm
- **We decide how.** Our telemetry data might lend itself to a natural segregation in the same way a CRM system might divide customers by region or country – our telemetry data could be divided by solar distance, attitude, approach vector, etc. If the majority of our calculations only need a subset of this data, sharding it logically might make more sense.

The less we mess with a sharding situation, the better – so I would suggest letting the machine decide unless you’re utterly positive your sharding strategy won’t change.

Once you’ve decided the strategy, it’s a matter of implementing it. Some systems make this very easy for you (RethinkDB, for instance) and others require a bit of work (PostgreSQL requires an extension and some configuration, MongoDB is a bit more manual as well).

If everything works well, you should be able to up A – your throughput and processing capacity – dramatically, at the price of C, consistency.

Other data, however, needs some additional guarantees in a distributed system.

## REPLICATION

We’re receiving and processing telemetry data constantly, and sometimes it might cause us to alter the current mission plan just a little.

Given that we’re using a distributed system, we need to be sure that the mission plan the probe is following is up-to-the-minute and guaranteed to be as accurate as possible – even if there is a massive power outage at our data center.

So, we replicate the data across the nodes in our cluster.

We have data centers in the US, Europe, Asia and the Middle East – all of which can communicate with the probe throughout the daily rotation of the Earth. This data doesn't change all that often – perhaps a few hundred times per day – so we can implement replication at the database level.

Whenever data changes in our replicated tables, we can guarantee that:

- The data will, at some point, propagate if a netsplit occurs. In other words: if we write to a node in the US and our US data-center goes down, the write will happen eventually when the US datacenter comes back online.
- The system will compensate and rebalance for the loss of a node, ensuring the data will be as consistent (and available) as possible.

## BIG DATA

Most applications generate low to moderate amounts of data, depending on what needs to be tracked. For instance: with Tekpub (my former company), I sold video tutorials for a 5 year period. The total size of my database (a SQL dump file) was just over 4Mb.

Developers often over estimate how much data their application will generate. My friend Karl Seguin has a great quote on this:

*I do feel that some developers have lost touch with how little space data can take. The Complete Works of William Shakespeare takes roughly 5.5MB of storage.*

## **Millions vs. Billions vs. Trillions**

A megabyte (1 million bytes) seems so small, doesn't it? A gigabyte (a billion bytes) seems kind of skimpy for RAM and we all want a few terabytes on our hard drives. What do these numbers really mean outside of computers? Consider this:

- A million seconds is almost 12 days
- A billion seconds is just over 31 years
- A trillion seconds is 317 centuries Many people simply do not grasp just how much bigger a terabyte is vs. a gigabyte. Let's do this again with inches:
- A million inches is almost 16 miles
- A billion inches is almost 16,000 miles, or a little over half way around the earth
- A trillion inches is 16 million miles, or 631 trips around the planet When considering the data generated by your application, it's important to keep these scales in the back of your mind.

# A Truly Large Data Store: Ancestry.com

In 2006, Ancestry.com added every census record for the United States, from 1790 to 1930.

DEPARTMENT OF COMMERCE-BUREAU OF THE CENSUS  
FOURTEENTH CENSUS OF THE UNITED STATES: 1920-POPULATION

STATE: Illinois COUNTY: Peoria TOWNSHIP OR OTHER DESIGN OF COUNTY: Peoria NAME OF INCORPORATED PLACE: Peoria NAME OF INSTITUTION: Peoria DAY OF INCORPORATION: 1837 SUPERVISOR'S DISTRICT NO.: 13 SHEET NO.: 5901 ENUMERATION DISTRICT NO.: 68 WARD OR CITY: Peoria CENSUS YEAR: 1920

PLACE OF BIRTH	NAME	RELATIVE	SEX	AGE	COLOR	EDUCATION	OCCUPATION	MARRIAGE AND MOTHER TONGUE		OCCUPATION
								Place of birth	Native tongue	
1	James, James J.	Head	M	34	W	High school	Farmer	Illinois	English	Farmer
2	Elizabeth, Elizabeth	Wife	F	32	W	High school	Farmer	Illinois	English	Farmer
3	William, William	Son	M	12	W	High school	Farmer	Illinois	English	Farmer
4	John, John	Son	M	10	W	High school	Farmer	Illinois	English	Farmer
5	Mary, Mary	Daughter	F	8	W	High school	Farmer	Illinois	English	Farmer
6	James, James	Son	M	6	W	High school	Farmer	Illinois	English	Farmer
7	Elizabeth, Elizabeth	Daughter	F	4	W	High school	Farmer	Illinois	English	Farmer
8	William, William	Son	M	2	W	High school	Farmer	Illinois	English	Farmer
9	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
10	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
11	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
12	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
13	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
14	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
15	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
16	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
17	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
18	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
19	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
20	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
21	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
22	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
23	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
24	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
25	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
26	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
27	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
28	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
29	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
30	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
31	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
32	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
33	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
34	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
35	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
36	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
37	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
38	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
39	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
40	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
41	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
42	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
43	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
44	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
45	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
46	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
47	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
48	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
49	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
50	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
51	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
52	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
53	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
54	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
55	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
56	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
57	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
58	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
59	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
60	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
61	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
62	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
63	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
64	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
65	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
66	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
67	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
68	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
69	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
70	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
71	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
72	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
73	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
74	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
75	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
76	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
77	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
78	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
79	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
80	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
81	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
82	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
83	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
84	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
85	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
86	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
87	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
88	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
89	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
90	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
91	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
92	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
93	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
94	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
95	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
96	James, James	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
97	Elizabeth, Elizabeth	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer
98	William, William	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
99	John, John	Son	M	1	W	High school	Farmer	Illinois	English	Farmer
100	Mary, Mary	Daughter	F	1	W	High school	Farmer	Illinois	English	Farmer

That's a lot of data. Ancestry.com tracks many people, as you can imagine. How big was their archive back then?

The project added 540 million names, increasing the company's genealogical database to 600 terabytes of data.

## Storing This Data In Your Closet

If you had an extra \$32,000 lying around after your seed round of funding, you could buy 15 x 40Tb drives and store all of this infor-



mation in your closet today. In 2006 this amount of data was quite impressive and would have cost a fortune. Today ... not so much. Don't get me wrong: \$32,000 is a lot of money but it's pocket change for big companies needing to store tons of data.

In 2013 Information Week wrote an article about Ancestry.com and how it stores its data. This, friends, is a massive growth in data:

*A little over a year ago [2012], Ancestry was managing about 4 petabytes of data, including more than 40,000 record collections with birth, census, death, immigration, and military documents, as well as photos, DNA test results, and other info. Today the collection has quintupled to more than 200,000 records, and Ancestry's data stock-pile has soared from 4 petabytes to 10 petabytes.*

A petabyte is 1000 terabytes – or 1000 trillion bytes of data. If we translate that into seconds it would be almost 32 million years.

Computer Weekly wrote a fascinating article on visualizing the petabyte, with some amazing quotes from industry experts:

*... Michael Chui, principal at McKinsey says that the US Library of Congress "had collected 235 terabytes of data by April 2011 and a petabyte is more than four times that."*

*Wes Biggs, chief technology officer at Adfonic, ventures the following more grounded measures... One petabyte is enough to store the DNA of the entire population of the US – and then clone them, twice.*

*Data analysts at Deloitte Analytics also put on their thinking caps to come up with the following... Estimates of the number of cells in a human body vary, but most put the number at approaching 100 trillion, so if one bit is equivalent to a cell, then you'd get enough cells in a petabyte for 90 people – the rugby teams of the Six Nations.*

A petabyte is **huge**. You might be wondering why I'm throwing these statistics at you? It has to do with the title of this chapter.

## WHAT IS BIG DATA?

Social media is driving the idea of Big Data:

*Big data is a term for data sets that are so large or complex that traditional data processing applications are inadequate. Challenges include analysis, capture, data curation, search, sharing, storage, transfer, visualization, querying, updating and information privacy. The term often refers simply to the use of predictive analytics, user behavior analytics, or certain other advanced data analytics methods that extract value from data, and seldom to a particular size of data set.*

It's a buzzword, sure, but there is meaning behind it. Companies like Google, Facebook and Twitter are generating gigantic amounts of data daily. Back in 2008 Google was processing over 20 petabytes of data per day:

*Google currently processes over 20 petabytes of data per day through an average of 100,000 MapReduce jobs spread across its massive computing clusters. The average MapReduce job ran across approximately 400 machines in September 2007, crunching approximately 11,000 machine years in a single month.*

While researching this chapter I stumbled on an interesting post from FollowTheData.com, which outlined how much data was processed by certain organizations daily back in 2014:

- The US National Security Administration (NSA) collects 29 petabytes per day
- Google collects 100 petabytes per day
- Facebook collects 600 petabytes per day
- Twitter collects 100 petabytes per day
- For data storage, the same article states:
  - Google stores 15,000 petabytes of data, or 15 exabytes
  - The NSA stores 10,000 petabytes
  - Facebook stores 300 petabytes

These numbers are rough estimates, of course. No one knows about Google's storage capabilities outside of Google, but Randall Munroe (of xkcd fame) decided to try to deduce how much data

Google could store for one of his What If? articles using metrics like data center size, money spent, and power usage:

*Google almost certainly has more data storage capacity than any other organization on Earth... Google is very secretive about its operations, so it's hard to say for sure. There are only a handful of organizations who might plausibly have more storage capacity or a larger server infrastructure.*

A fascinating read. Please take a second and have a look – but be warned! You will likely get lost in all the *What If?* posts.

## **Processing Petabytes of Information**

Simply put relational systems are just not up to the task, for the most part. The reason for this is a simple one: processing this data needs to be done in parallel, with multiple machines churning over the vast amounts of data. This is the most reliable way to scale a system like Google's that generates gigantic quantities of data every day: just add another data center.

How do you process this kind of information in parallel, however? This is where systems like Hadoop come in:

*Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.*

Hadoop was born from efforts at Yahoo!, and then turned into an open-source project that any organization can download and install.

Hadoop partitions your data using its dedicated file system, HDFS, which is based on Java. When you query data, you use Map/Reduce.

## IN THE REAL WORLD

I worked at a business analytics company for about 4 years, working with companies who wanted us to sift through their warranty claims information, looking for patterns. This involved natural language processing (NLP), where we split claim information into sentence structures and then ran various algorithms over it.

It was fun, but 90% or more of my work was trying to figure out which data were good. Emails, phone calls, sifting through and correcting millions upon millions of records ... it's a lot of work.

Even then, I wouldn't call that Big Data. That was just basic analysis over a large set of data. As Sean Parker's character said in *The Social Network*:

*A million dollars isn't cool, you know what's cool? ... A billion dollars.*

These days a billion records of anything doesn't even mean much. Terabytes and ... yeah you're getting there. You know what's cool? A *petabyte* is cool.

# TEST DRIVEN DESIGN

I am, strictly speaking, not a practitioner of Test-driven Design (TDD) or Behavior-driven Design (BDD). I know what these things are and I *feel* why they are useful. I also know that people like to argue about what they think it is and what they think it is not.

This leaves me with a problem. How do I explain what these things are without being over-opinionated in one direction or another?

So here's my plan: *just show the essence of the idea*. I think we can all agree on that, can't we? For this chapter I approached several friends and asked them about what they considered the essence of testing to be, and how they think of using it. By the way – each of them hedged their opinions with a variation of “this isn't strictly TDD (or BDD)... but...”.

**Testing requires discipline** and you're not alone if you sort of do it. As long as you're testing your code!

As with many of the chapters in this book, the code for everything you're about to read is at GitHub. Clone or download if you want to play along.

# SOME OPINION ABOUT TESTING IN GENERAL

Before we begin, let's have a think about testing your code, aside from TDD. No matter what: **test your code**. There really is *no excuse* to not test what you create, to make sure it's correct.

I will say this, directly: if you're not testing the code that people are paying you money to write *you deserve to be fired*. I won't justify that remark - if you don't believe me you should move on. Better yet: close this book and rethink your career. The fact that we should test is a given. It's *how* (and in some cases when) we test that is the subject of this chapter.

In that spirit, let's engage with Test-driven Design (TDD) and Behavior-driven Design (BDD) focusing on the fun parts. It's my hope you can see how creating your tests with a design pattern in mind can be extremely useful.

We'll start with TDD and then move on to BDD. After that, we'll discuss some of the filigree that goes into testing code in general.

## THE NUTS AND BOLTS OF TDD

Unless you've been living under a rather large rock over the last 10 years, you've heard of TDD. Maybe in good ways, possibly in bad ones. At its core it's a simple practice:



- You think about what you need to create
- You write a simple test to get yourself started
- You run that test and watch it fail
- You write some code to make the test pass
- You write another test for the next step, and repeat the process

As you go along, you're constantly refactoring what you've written – and this is the somewhat goofy part: you write the barest minimum to make a test pass.

## AS CLOSE TO A REAL EXAMPLE AS I CAN GET

A few years back I recorded a video with my friend Brad Wilson and the idea was to capture him doing “real” TDD. No to-do list, no fake blog demo example ... *real stuff*. Brad is the creator of XUnit (along with Jim Newkirk) and is an everyday practitioner of TDD. I couldn't imagine a better person for that video.

The video was for my former company, Tekpub, and it was entitled “Full Throttle: TDD With Brad Wilson”. Brad didn't know what I was going to ask him to do – so we sat together, I recorded his desktop, and then asked him to create a subscription billing system for me.

What Brad did next changed the way I thought about TDD. Unfortunately, the video is no longer available (it belongs to Pluralsight who has retired it) – but I will recount the highlights for you here.

## Just Start

Brad used Visual Studio 2012 and C# 4.5, creating a library project (BillingSystem) and a test project (BillingSystem.Tests) in a matter of seconds. He added XUnit to his test project and then paused to think about a few things:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Xunit;

namespace BillingSystem.Tests
{
    public class BillingDoohickeyTests
    {
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }
}
```

Two things about this code struck me. The first is Brad's use of comments (lines 11 through 14) to get the concepts in his head out onto the screen and, more importantly, he's not getting in his own way thinking about names and structure.

Brad had no idea what to call his test suite just yet, so he called it **BillingDoohickeyTests** in part for fun, but also to remind himself to rename it once things started rolling.

What comes next? What classes should we create right off the bat? This plagues many developers who can't even get past this point.

## **Patience, Discipline**

Here's the thing with TDD that causes anxiety almost immediately: it takes rigor, and it feels pretty silly, if I'm honest. So far nothing we've done is overly goofy (apart from the naming thing) – but in a second, you'll see what I mean.

You can do a little thinking upfront, but TDD tries to discourage over-engineering by pushing you to let your tests tell you what to write. That's where we're going to start.

## **The Customer**

We're building this system so we can charge customers, so why not start there? This is exactly what Brad does:

```
namespace BillingSystem.Tests2
{
    public class BillingDoohickeyTests
    {
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }

    public class Customer
    {
    }
}
```

He just put the class to test right there, next to his test code. Why not? TDD is a rigorous process, but it doesn't need to be slow.

OK, so we have a **Customer**, now we need to charge the customer on a monthly basis. At this point: *stop thinking*. Let's put this idea in motion with a test:

```

namespace BillingSystem.Tests3 {
    public interface ICustomerRepository { }
    public interface ICreditCardCharger { }
    public class BillingDoohickeyTests3 {
        [Fact]
        public void Monkey () {
            var repo = new Moq.Mock<ICustomerRepository> ();
            var charger = new Moq.Mock<ICreditCardCharger> ();
            BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
            thing.ProcessMonth (2016, 8);
        }
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }
    public class BillingDoohickey {
        public BillingDoohickey (ICustomerRepository repo, ICreditCardCharger charger){}
        public int ProcessMonth (int year, int month) {return 0;}
    }
    public class Customer{}
}

```

A lot just happened here. Let's step through it.

You'll notice that Brad isn't concerning himself, again, with names. We have **Monkey** and **thing**, which might be making you cringe – but for Brad, he's removing obstacles to his design process.

Which is what TDD is supposed to be: **a design process**.

Next, he's using mocks as you can see on lines 12 and 13, provided by the Moq project (fake classes for testing) upfront so he doesn't need to think about implementation just yet – he's leaving that for later. We'll discuss mocks and stubs (also known as "test doubles") more in just a minute.

## The Happy Path

At this point we have a little machinery to play with, but we still don't know what we're doing completely. When I write tests, the very first thing I do is to create what some people call the happy path: one or more tests that pass when everything works as we expect it to work.

In other words, if we're building a registration system then our happy path would be something like **User\_is\_registered\_with\_a\_valid\_login\_and\_password**. This test should always pass.

Once our happy path is set, we go about trying to break it. We'll do that later. Right now, let's create a "happy path" for ourselves to get us off the ground, renaming Monkey to focus ourselves:

```

namespace BillingSystem.Tests4 {
    public interface ICustomerRepository { }
    public interface ICreditCardCharger { }
    public class MonthlyChargeTests {

        [Fact]
        public void Customers_With_Subscriptions_Due_Are_Charged () {
            var repo = new Moq.Mock<ICustomerRepository> ();
            var charger = new Moq.Mock<ICreditCardCharger> ();
            BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
            thing.ProcessMonth (2016, 8);
        }
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }
    public class BillingDoohickey {
        public BillingDoohickey (ICustomerRepository repo,
                                ICreditCardCharger charger) {}
        public int ProcessMonth (int year, int month) {
            return 0;
        }
    }
    public class Customer {}
}

```

In this code I've renamed a few things because I now have an idea what I'm doing. The test suite is called **MonthlyChargeTests** and my test name makes it clear what it's going to test for. This name is far too broad, but it will change at some point.

The simple renaming, however, has forced me to consider a few more bits of functionality:

- What is a Subscription?
- What does it mean for a Subscription to be Due?

- A Customer, apparently, needs to have a Subscriptions property

I stop here – thinking about YAGNI <sup>9</sup>(You Aint Gonna Need It). It's tempting to plow ahead and add a **Subscription** class and a **Subscriptions** property to my **Customer**... but do I need to just yet? It does seem obvious, but this is the rigor part.

Let's focus on the test, add an assertion, and move on from there:

```
[Fact]
public void Customers_With_Subscriptions_Due_Are_Charged () {
    var repo = new Mock<ICustomerRepository> ();
    var charger = new Mock<ICreditCardCharger> ();
    BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
    var processed = thing.ProcessMonth(2016,8);
    Assert.True(processed > 0);
}
```

This is where we venture into silly land. Our first goal is to make sure this code can compile – so I've added the interfaces and classes that we need. I also added a **ProcessMonth** method to **BillingDoohickey** and, for now, I'm returning 0 because I don't know what else to return.

If we run this test, it will fail. We'll also probably feel a bit badly about ourselves because we might not have any customers with subscriptions due ... so what then? I'll get to that in a minute – for now we can compile our code and run this test: **watching it fail**.

---

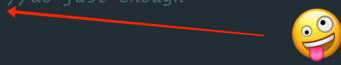
<sup>9</sup> YAGNI is covered in the *Software Design Principles* chapter  
863



That's a critical aspect here – our initial test needs to fail because we don't want to accidentally write a test that passes! Which does happen.

Now, let's get our test to pass:

```
public class BillingDoohickey {  
    public BillingDoohickey (ICustomerRepository repo,  
                             ICreditCardCharger charger){}  
    public int ProcessMonth (int year, int month){  
        return 1; //do just enough  
    }  
}  
  
public class Customer{}
```



Ugh. Our tests pass and, as dumb as it seems, *this is TDD*. We will fix this code and, in fact, the dumber it feels the better it is because it forces you to write more tests just to get this kind of thing out of your code!

For now, our happy path is set. Let's blow it up.

## The Sad Path

The sad path is all about trying to blow up the happy path. It's "what happens when I do this!" The obvious first thing is to write a test that is in complete opposition to our happy path:

```

[Fact]
public void Customers_With_Subscriptions_Due_Are_Charged () {
    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed > 0);
}

[Fact]
public void Customers_With_No_Subscriptions_Due_Are_Not_Charged () {
    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed == 0);
}

```

I moved some declarations around in my test because I want to keep things DRY <sup>10</sup>(Don't Repeat Yourself) – a messy test suite is one you'll want to stay away from – so I moved everything up top and into the constructor.

Next, I created the opposite test, asserting that customers without subscriptions would not be charged – which fails because I've hard-coded the result into the **BillingDoohickey**.

Now I need to think about a few things. Specifically: *what does it mean to have no subscriptions?* **Who cares!** For now, let's get this test to pass.

I'll start by setting the mock for **ICustomerRepository** available for orchestration using the **repoMock** variable:

---

<sup>10</sup> DRY is covered in the Software Design Principles chapter

865

```

ICustomerRepository repo;
ICreditCardCharger charger;
BillingDoohickey thing;
Mock<ICustomerRepository> repoMock;

public MonthlyChargeTests () {
    repoMock = new Mock<ICustomerRepository> ();
    repo = repoMock.Object;
    charger = new Mock<ICreditCardCharger> ().Object;
    thing = new BillingDoohickey (repo, charger);
}

```

Next I'll add a method called **Customers** to the **ICustomerRepository** interface because my test told me I needed to:

```


[Fact]
public void Customers_With_Subscriptions_Due_Are_Charged(){
    repoMock.Setup (r => r.Customers())
        .Returns (new Customer [] { new Customer () });

    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed > 0);
}

[Fact]
public void Customers_With_No_Subscriptions_Due_Are_Not_Charged (){
    repoMock.Setup (r => r.Customers())
        .Returns (new Customer [] { });

    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed == 0);
}

```



Now that I've done this, I can refactor the **BillingDoohickey.ProcessMonth** method to return a count of the records in the repo.

This is a prime example of “just doing enough to get the tests to pass”:

```
public class BillingDoohickey{
    ICustomerRepository _repo;
    public BillingDoohickey (ICustomerRepository repo,
                           ICreditCardCharger charger) {
        _repo = repo;
    }
    public int ProcessMonth (int year, int month){
        return _repo.Customers().Length;
    }
}
```

This kind of thing is fun when pair coding. I remember pairing with a friend once and laughing so hard at just how creative she was at writing the dumbest code possible to get my tests to pass:

*You keep writing tests like that, I'll keep writing code like this.*

It's fascinating to split your personality when testing like this. See if you can outsmart yourself with a more interesting test – something to break the happy path for once and for all! Then stave off the attack with some goofy way around it, like we just did here, returning the count of users.

It's tempting to scrap this test and try to write something more concise, which you're welcome to do. I typically just write another test specifically so I can get this crappy code out!

How about this:

```
[Fact]
public void A_Customer_With_Two_Subscriptions_Due_Is_Charged_Twice ()
{
    var customer = new Customer ();
    customer.Subscriptions.Add (new Subscription ());
    customer.Subscriptions.Add (new Subscription ());

    repoMock.Setup (r => r.Customers())
        .Returns (new Customer [] { customer });

    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed == 2);
}
```

What do you think about this? You can play my pair who's writing code to get the tests to pass ... think you can make me sad?

Here's a way to do it:

```

public class BillingDoohickey
{
    ICustomerRepository _repo;
    public BillingDoohickey (ICustomerRepository repo, ICreditCardCharger charger) {
        _repo = repo;
    }
    public int ProcessMonth (int year, int month) {
        var customer = _repo.Customers().FirstOrDefault();
        if (customer == null) {
            return 0;
        } else {
            return customer.Subscriptions.Count ();
        }
    }
}
}

public class Customer {
    public IList<Subscription> Subscriptions { get; set; }
    public Customer () {
        this.Subscriptions = new List<Subscription>();
    }
}
}

```

Ha! With that test I was able to push the code so that two new concepts could be added: a **Subscription** and a property on **Customer** called **Subscriptions**. You were still able to write silly code to get the tests to pass – so you win too!

Notice the ongoing *failure, pass, refactor ... failure, pass, refactor* that we're doing here. That's exactly what TDD is all about and despite the way many people make it seem – it can be extremely fun. Especially if you have a pair to code with you – real or imaginary.

A final thing to notice, *I did one step at a time*. I didn't write out a set of tests upfront, which would defeat the idea of challenging myself as I go with YAGNI.

# TEST DOUBLES AND YOUR SYSTEM DESIGN

As your test suite grows, you're going to find yourself up against the "Integration Wall", which can present problems. The most typical problem you'll face is *persisting data* and how to treat things that need to be saved to a database as part of a test run.

Using our subscription payment scenario, let's say that a payment comes in as a webhook ping from Stripe, the popular payment processor which handles recurring billing. A typical process might be:

- The ping is received and saved with some kind of initial status.
- A Payment is created from this ping and the ping's status is updated
- A Subscription is found based in information on the Payment, some form of customer ID for instance. The ping is once again updated with the subscriber info
- The Customer for that Subscription has their status reset with a payment recorded.
- An email is sent to the customer thanking them for their continued support
- The ping status is set to "resolved" so we know it was handled OK

A process like this needs to be thoroughly tested, of course, but how do you do it without hitting the database repeatedly?

This is where mocking and stubs can greatly help, but they can also hurt tremendously if we do them wrong.

## DON'T MOCK WHAT YOU DON'T OWN

I can't tell you how many times I've fallen back on this rule when my testing strategy was falling apart. It's tempting to think "well I'll just mock the X driver/service so I don't hit the database", but that's disastrous for a simple reason: *you'll never be able to match what someone else has done*. It almost always comes back to bite you!

The code you "own" is the code you've written for your application. Feel free to mock it however you want, but wrap the code you don't own (like an ORM, for instance) in a service class that you can mock or stub later on<sup>11</sup>.

---

<sup>11</sup> A "mock" is typically an object, or a "what" that we set explicit expectations on. Like when we mocked the subscription repository above to return two subscriptions. Stubs are dummy objects, typically, that contain the answers we're looking for. For instance, you might stub a Customer to see if the status changed during a test. These terms are often confused.



I tried to do this once with Nodemailer and I'll never do it again. It seemed like such a small thing but once I went down this path I found I had to mock out some very deep internal method calls and it made me want to cry.

Databases are also a problem in this way. Mocking an ORM, for instance, is something that at least every developer must try at least once. There is just no way you'll ever be able to do this properly and I feel like you're going to try anyway, which is how we learn isn't it? When you're mocking out your 20th method response (incorrectly) perhaps you'll remember the phrase *don't mock what you don't own* and try something different.

What's the answer then? That's a big question, but the short answer is to rethink your system design and what each object is doing. That's why they call testing in these ways a "design process". It's a happy coincidence that a testable application is typically one with an architecture that lends itself to change. We like this as software architects, but at this point I'm going to stop because this book is *not* about architecture!

## AVOIDING COMMON PROBLEMS

It's always simple to talk about TDD and BDD in the abstract, but when your test suite tips 300 different specifications and tests... things can be a bit difficult to manage. You find this out quickly

when one change in your codebase causes you to update 50 different tests.

These are called “brittle” tests and can really be a pain and I hate to say it, but there really is no avoiding it: *it’s going to happen at some point*. Some IDEs excel at refactoring (such as Visual Studio) so if you change a property name, for instance, the change ripples throughout the code. If your tests break due to a property name change, however, you might want to think about why that is.

Obviously tests that focus on the property you changed *should* break. This is a natural part of refactoring. When unrelated tests break it’s often because you’re doing too much or involving too many external factors for a given test.

Testing our MonthlyBilling service would be one such candidate for brittle tests. Knowing this from the start, however, gives you an advantage! How much do you need to involve your Customer and Subscription classes in the MonthlyBilling service? If you change Customer.firstName to Customer.first – will that break your MonthlyBilling tests? I sure hope not! In that sense you might want to work with a lighter interface for the customer rather than the entire thing. Perhaps an IBillable or something?

This is a great discussion to have with your colleagues and is precisely why testing can guide your architecture to good places. The simple reason is this: **if your tests are hard to refactor, your code will be 10 times harder!**

## In The Real World

Like I said above: *I get lazy sometimes*. OK *many* times. I'll get into this in more detail in the chapter on Behavior-driven Design (BDD - coming up next) but I tend to use tests as more of a checklist. Things I expect to work, etc.

I often get carried away and I just keep coding – which I know is bad. I pay for this choice often when I'm deleting code that took a while to get to work, but that I ultimately don't need.

So, I take a deep breath, maybe I'll go for a walk or get some tea. When I come back, I clean up my tests and refocus myself. This is TDD to me – more of a battle with myself than anything.

# BEHAVIOR DRIVEN DESIGN

**B**ehavior-driven Development (BDD) is basically the same process as TDD, but you have a specific focus: *behavior of the application*. It's a subtle shift, but an important one.

## GETTING STARTED

In the last section on TDD we began to build out a **BillingSystem** using TDD which works, but it's slightly mechanical. In other words: Widget X will return Y when I pass in Z. This defines what we expect to happen as developers, not what we expect as humans.

Let's shift this to focus on a story instead, with some scenarios. We'll start with what our application will do when a payment is received for a monthly subscription:

```

using System;
using Xunit;

namespace BillingSystem.Specs {
    [Trait ("Monthly Payment Is Due", "Payment Is Received")]
    public class PaymentReceived
    {
        [Fact]
        public void An_Invoice_Is_Created(){}

        [Fact]
        public void Subscription_Status_Is_Updated(){}

        [Fact]
        public void Next_Billing_Is_Set_1_Month_From_Now(){}

        [Fact]
        public void A_Notification_Is_Sent_To_Subscriber(){}
    }
}

```

As you can see, I'm thinking in terms of how the system will react when an event happens. Another way to put this is "this is the specified behavior of a feature". The mental approach to this idea is outside in, meaning that we typically come at this type of testing from a user's perspective. Therefore you'll sometimes hear this type of testing as "acceptance testing".

BDD, like so many things, has been jargoned to death and is also subject to "cargo-culting", which means if you call your file "some-

thing\_spec” and the thing you’re testing a “feature” given a “context” then you’re doing BDD.

This is not the case. BDD is about behavior of your system when a thing happens, that’s all.

For instance, what happens when a payment fails? Let’s spec it out<sup>12</sup>:

```
[Trait ("Monthly Billing", "Payment Fails")]
public class SubscriptionPaymentIsDue
{
    [Fact]
    public void An_Invoice_Is_Not_Created(){}

    [Fact]
    public void Next_Billing_Is_1_Day_From_Now(){}

    [Fact]
    public void A_Notification_Is_Sent_To_Subscriber(){}
}
```

---

<sup>12</sup> The Xunit way of doing BDD is a little verbose and there are other ways. I typically don’t like a ton of testing dependencies so I’ll stick with the most basic tooling I can. In Node, for instance, you can write “specs” easily by telling Mocha (a Node testing framework) what style you’ll be doing. This amounts to syntactic sugar, but it is very helpful when reading results.

This code is doing the same basic thing as TDD: *testing our code*. This time, however, we're doing it in the form of "how does our application respond when this thing happens". In other words: **behavior**.

## PHILOSOPHY

This approach is different from strict unit testing, which tends to be more clinical. In other words, you might put a certain class under test, vary the input data to see where it fails and then refactor until it succeeds. This is fine, but has some disadvantages, which are:

- The tests are bound to the design of your class by definition. That is the point of TDD. If you change your design, you have to change your tests and your code. This can be quite frustrating.
- The focus is on engineering, not application experience. When you're focused on code, the code wins. When you're focused on behavior, however, you're focused on the user's experience and the business wins.
- Testing proliferates. The tendency with TDD and unit testing is to have as much code coverage as possible. When you use BDD, you typically write few tests which are more targeted to application experience.

With BDD you tend to write your tests detailing what the application will do under certain circumstances. Given this, BDD fans will call their tests “specifications”, as they tend to read as if dictated directly by the client. In the example above, I’m using XUnit’s **Trait** attribute to decorate my scenarios so they’re a little more readable in the test runner.

I’ve also made sure that my test names rely completely on the test class itself (called the scenario). When you run this test, you see this (or something like it):

```
[Monthly Billing]: Payment Received
```

- An\_Invoice\_Is\_Created
- Subscription\_Status\_Is\_Updated
- Next\_Billing\_Is\_Set\_1\_Month\_From\_Now
- A\_Notification\_Is\_Sent\_To\_Subscriber

```
[Monthly Billing]: Payment Fails
```

- An\_Invoice\_Is\_Not\_Created
- Next\_Billing\_Is\_1\_Day\_From\_Now
- A\_Notification\_Is\_Sent\_To\_Subscriber

This is the XUnit runner output, which you can jiggle in Visual Studio if you want. If you’re using a framework in another language (like Mocha for Node or RSpec for Ruby) you can have a more readable output.



# FEATURES, SCENARIOS, EXPECTATIONS

I try to focus on the notion of *Feature*, *Scenario*, *Expectations*. I know this might seem like a syntax dance to you, but it's incredibly easy to lapse back into unit testing mode – not that there's a problem with that! Unit tests are indeed needed in some cases (testing utility code, parsers, etc.).

Let's revisit our example code:

```
//...
namespace BillingSystem.Specs {
    [Trait("Monthly Billing", "Payment Is Received")]
    public class PaymentReceived{
        //... the constructor prepares the test data
        [Fact]
        public void An_Invoice_Is_Created(){
            //...
```

In this code you can see the features directly by examining the **Trait** attribute on the test class. The first element is the feature ("Monthly Billing"), the second is the scenario ("Payment Is Received"). Your testing library might have a different way for specifying these things.

# SO WHAT?

You might be wondering, at this point, why all of this even matters? BDD does have several advantages:

- **Readability.** It sounds idealistic but being able to print out a test run and read (in common language) what's going on is powerful.
- **Ubiquity.** I hate that word, but it's applicable here: you know what these tests describe, and your client/boss will know as well. In this, you're speaking the same language.
- **Focus.** If you're focused on how your application behaves, you're aligning yourself with the business goals. This is important for programmers! You can watch your application evolve into something exciting and understand why it's doing what it's doing. You might even have some questions about this, which means you can contribute your genius to the application's design.

There's obviously some jargon to muddy up what is, otherwise, an elegant development practice. I brought this up before so let's find out what I mean.

## ON JARGON AND CARGO CULTING

BDD tests are typically called specifications or “specs”. In the theoretical world you could sit with your client, create a list of specifica-

tions for various aspects of your application, and then translate that directly into your test suite.

In the real world I've found that my clients have never cared about my tests. Maybe it's just my clients – not sure – but even when I worked at Microsoft and tried to show progression using my test suite I was laughed out of the room.

Clients like to see results, not test runs. It is good, however, to use the same language in your tests as you do with them in email or on the phone. Trying to align your thinking is important.

To that end, you could have this conversation and you could, using BDD, translate it directly into some tests:

*So let's recap the conversation: given a successful monthly billing – the billing system should generate an invoice, set the next billing date to exactly one month from now, set the user's subscription to active and then email the user. Correct?*

Say this out loud to yourself, as if in conversation. Feel free to use your own words.

Do you hear any problems? Your client probably will. Here's a reply I received once, when I said almost exactly that sentence to a startup client back in 1999:

*Rob - yes for the most part this is correct but accounting handles the invoices so I think just notifying them will work. I don't think we should be creating our own invoic-*

*es. As far as email goes, I'd want to loop in our marketing team as they own client communications...*

Do you see what just happened there? Not only did I save myself some work, but I also opened a great conversation about the role of our application within the new company.

The feature we were discussing is the monthly billing run. The scenarios we created were payment received and payment failure; these are also called contexts. We describe the behavior of the application in response to a scenario with specifications:

```
Monthly Billing Run //feature
- Payment is successful //scenario, or context
- an invoice is created //specification
```

Is this jargon important? *Yes, and no.* It is important in the sense that you should be thinking in these ways when doing BDD. It's also important to know that just because you call a test class **MonthlyBillingFeature** and a method on that test class **SuccessfulPaymentScenario** does not mean you're doing BDD. BDD is a process of discovery for both you and your client.

That said, you can call a feature a **pancake**, a scenario a **lovely-butterfly** and each specification **larry**. The naming doesn't matter – as long as your team understands what it is you're doing and why.

# GIVEN, WHEN, THEN

Cucumber is a popular Ruby test tool that helps you focus on BDD. It popularized a certain syntax, called Gherkin:

```
Feature: monthly billing run
  Scenario: payment received
    Given a charge of 20 USD
    And today is the 1st of the month
    When the charge is applied to Subscription x
    Then that subscription is considered active
    And an invoice is created
    And the customer gets an email
```

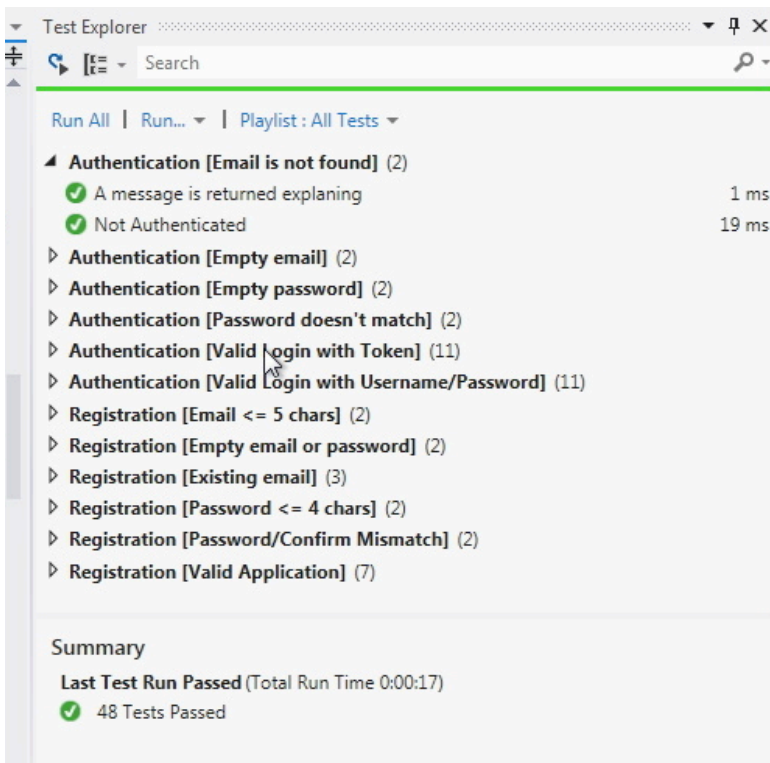
Gherkin is a specialized syntax you can use to get your head into the Given, When, Then syntax in a formal way. These rules might sound goofy but when you just start getting into BDD it can help get your mind in the right place.

It's effective but it does takes time to break things down in this way. Think about an application you're writing now ... how would you detail the behavior of it? Give it a go!

# IN THE REAL WORLD

You might be wondering what this really looks like once a project has been up and running for a few months. Is it possible to scale this idea? To keep it readable and focused?

This is a project I did five years ago, using .NET, Visual Studio and XUnit - no additional tooling:



The feature under test is the first item in the list (**Authentication**, for instance) and the scenario is the second. Underneath each feature/scenario is a set of specifications. This approach worked well for me.

If you want to check out the code, I put it up at GitHub. The code is a few years old now, but it should still work.

If you're feeling a little vague on the idea still, I don't blame you. It took me a while to get my tests (ahem, sorry: specs) to flow with behavior vs simple unit tests. The best thing I can offer you is to look at the difference between the billing system code at the start of this chapter compared with the code from the last chapter on TDD.

This is important: *both are valid, and both are lovely*. Some people favor TDD, others BDD. Experiment!

# SHELL SCRIPTS

**U**nix and Unix-like systems (Linux, BSD, Solaris, RedHat, etc.) have been around forever. You simply can't expect to grow much in your career if you don't have a basic competency with Unix and its commands. While this kind of thing isn't (typically) a part of a CS curriculum, it *is* something you pick up along the way in university.

If you don't believe me, skip right over this chapter. It'll be here when you come back, after you've realized just how true this is.

This is an exciting thing! Crawling under the hood of your computer can increase your efficiency dramatically. Shell scripts, Makefiles, server setup routines, quick little commands to update your system, configuring your web/database server remotely over SSH ... these are skills you must know.

So let's wander through the shell. I won't go into Unix history as I'm just not qualified to do so. I'll also sidestep the basics of the Unix commands – that'll be up to you.

Instead, let's get right to the thing that will help you the most in your job: basic shell scripting skills.



*You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy 34 video walkthroughs of the code you see in this chapter and others from here.*

## WHAT IS A SHELL?

A computer needs a way to receive data, and we're going to do that through the command line using a thing called a shell. The first computer ever conceived used punch cards to receive data, when I was in high school, I used a combination of a keyboard as well as a cassette tape player to boot my computer!

Today we have visual interfaces that look quite juicy and convey information in a friendly way. We use mice to issue commands (most of the time) and, occasionally, our fingers or a stylus.

During the 1960s through 1980s, computer users entered their commands as text from a keyboard. This practice has continued today and is what you're about to do, using the command line interface.

All these things are shells. A shell is simply a generalized way in which you give commands to a computer and receive the output. A visual shell uses a graphical interface, or a GUI, and is what I'm using now to type this sentence on my Mac, using a visual editor.

A text-based shell has no visuals except for things you can do with ASCII symbols. To work with a text-based shell (like Bash, for instance), you use a command line interface, or CLI.

There are several shells that you can work with, so far we've discussed two: Powershell and Bash. You can install other ones, if you like, including:

- Z shell (or zsh). I like this one a lot and it's what I use every day together with Oh-My-Zsh from Robby Russell
- Fish. They win for the best tag line: "Finally, a command line shell for the 90s"
- Tcsh (or "tc shell"). This is a common one you see on many Unix machines

## WHY THE NAME "SHELL"?

At this point you might be wondering why these things are called "shells". It has to do with the way Unix is constructed. There is a kernel that does all the processing, which is protected by several "protection rings". Each ring provides certain services, with the most sensitive being closer to the kernel and the least being on the very edge, or "shell" of the system. I won't go into Unix design at this point (mostly because I'm not qualified to); but I find that an interesting way to think about Unix.

If you look around, you'll find many shells that look interesting. Bash works well for most things, but if you're looking for something a bit more friendly than I might recommend having a look at Z Shell. I've been using it for years and love it. One main reason is that it has helpful completions, spelling corrections, and you can program the prompt to be colorful and pretty.

The biggest reason, however, is the Oh-My-Zsh project, mentioned above. You get a sane way to organize scripts, aliases and other things. Here's their project description:

*A delightful community-driven (with 1,000+ contributors) framework for managing your zsh configuration. Includes 200+ optional plugins (rails, git, OSX, hub, capistrano, brew, ant, php, python, etc.), over 140 themes to spice up your morning, and an auto-update tool so that makes it easy to keep up with the latest updates from the community. <http://ohmyz.sh/>*

It's been very useful for me.

## KEEPING SHELL STUFF ORGANIZED

This is kind of a big deal. As you learn to work with the shell more and more, you find good organization becomes important. Oh-My-Zsh can help with most things, but not everything.

For example: settings. If you ever work with Vim, Git, Atom, AWS, etc. – you know they have various startup settings in an rc file somewhere. These files (typically ending in “rc”) <sup>13</sup>need to live somewhere. This is where strong organizational skills will help a lot.

## DOT FILES

Most Unix-adept developers treat their scripts and settings with the same respect as any other code: organizing it carefully and versioning it with git. For example, one of my very favorite people is Gary Bernhardt and he keeps his dot files on GitHub.

You’ll hear the term “dot files” a lot – and you’ll see them a lot. It’s a convention to begin a file name with a period (or “dot”) to hide it from the finder and from the standard listing command, ls. These

---

<sup>13</sup> *The more you work with command line tools, the more you’ll come across “rc files”. You don’t have to name startup scripts with an “rc” ending, but if you do, people will know what it’s supposed to do by convention.*

---

*Like so many things in Unix land, the origin of the term “rc” is a bit cloudy. Google it if you like, but the actual meaning of it doesn’t matter. Just know that .thingrc will be the startup script for the thingcommand/binary/whatever. Some people like to think it means “runtime configuration” – that sounds like a good explanation to me.*

files are shell scripts that are read in by various programs and they contain settings of all kinds.

One that a lot of people obsess on (including yours truly) is `.vimrc`. In this file you will find settings for Vim. It's a shell script that's executed every time Vim starts.

Think about developers you follow on Twitter and see if they have a dot files repository on GitHub. I already mentioned Gary Bernhardt - here's another one of my favorite people: Ryan Bates. Have a look at how they organize these files and what's in there. You could lose hours on this!

In case you haven't figured it yet, "rc files" are simply shell scripts that are executed by a program when it starts. They're simply text files that are, typically, well-commented and allow you to change how a program behaves.

OK, so now we understand how programs use shell scripts to configure themselves. How can you use them to help with what you do every day?

## WHY SCRIPT A SHELL?

Think about the project you're working on right now and the tasks you need to perform on a routine basis. Here are some that I do when building web sites with Node, Ruby, or Elixir:

- Navigate to a project

- Open up the project in an editor of some kind
- Work with a source control system, something like Git perhaps
- Work with a database, something like PostgreSQL, MongoDB, etc.
- Write a blog post, perhaps
- Lint/concatenate/minify/compile your code files (CSS, JavaScript, whatever)

Sure, there are plenty of tools that can do these tasks graphically for you. Code editors have dialogs for opening certain directories, database GUI tools can help you write queries, and there are plenty of tools out there for graphically working with Git. These tools look nice, but they're horribly slow when compared to their command line (CLI) counterparts.

Every task you and I do daily can be done faster with a CLI tool. You can type much faster than you can visualize/click. We could argue that point, I suppose. I have some good friends who work in Visual Studio (Microsoft's .NET IDE) and with the purchase of a few plugins they can write code rather quickly and have gained a decent level of efficiency.

It still doesn't come close to what you can do using shell scripts.

Let's say your boss comes in and asks you to make sure you have a database backup setup on a nightly basis that zips and loads the

file to your Amazon S3 bucket. How would you do this with your favorite database GUI tools? This is a 20-line shell script.

You need to lint, concatenate and minify your JS files in a very particular way, using the rules your development lead has set for the team. In addition, you need to create a warning when the build size exceeds 100K. This is a 15-line Makefile.

You have a new marketing manager who has decreed that your company site needs to load faster – your current YSlow Grade is a D. It's decided that the 1200 JPEG files your site serves need to be optimized and reduced in size to no more than 600 pixels wide. This can be done in a 30-line shell script.

I know what you're thinking: *I can do all of this from the shell using Ruby/Python/JavaScript – why do I care about your shell scripts?* It's a good question, and a fair point. My response to that would be ... how many packages and supporting files are you going to need? How long will it take, as they say, to “shave that Yak”?

This is the great thing about shell scripts: once you know them, you know them. It's one of those skills that you can use to do ... just about anything.

## A SIMPLE SHELL SCRIPT

Your company website has quite a few images; some of them rather large. Much larger than they should be. A new marketing

director was just hired and found out the site is ridiculously slow to load, and has decided that these images are to blame. In short: *you have an image problem.*

Your boss has tasked you with auditing the images and then resizing them. What fun! Isn't this why you became a programmer? The very first thing she's asked for is a list of all the images in our site's directory. That will be our first task.

In the downloads for this section, you'll find a directory called "images". You can use that directory to work on.

## The First Step

Let's crack open our terminal. On a Mac, this is (most likely) going to be Terminal.app, which you can find in /Applications/Utility. Or you can get your keyboard skills on by typing CMD-Space to bring up Spotlight, then type "Terminal".

It will open in your home directory, or **\$HOME** in Unix land. To navigate around you can use **cd** to change directories – just use the name of the directory you want to go to. If you want to go back one, you can use **cd ..**; if you want go all the way back to **\$HOME** you can just type **cd** followed by **<Enter>**.

Let's assume you downloaded the image files to your Desktop. For simplicity, let's create a directory in our **\$HOME** called "imposter", and then another inside that one called "demos". In your terminal, type **mkdir -p imposter/demos**.



This command will create a directory set in your **\$HOME**. The **-p** flag tells **mkdir** to create the entire structure if it's not already there.

Nice work, now let's move our demo files in there, and then change into that directory:

```
mv ~/Desktop/task-images ~/imposter/demos
cd imposter/demos/task-images
```

The command **mv** will move files and directories around on your machine and **cd** will change directory, which I'm sure you could reason out.

Is all this typing getting you down? Bash and many other shells support command completion using **TAB**. Try it! It really helps when navigating around your machine.

Now that we're here, let's list out the images. You can list files with the **ls** command, but you can also restrict it with what's known as a glob. You can think of this as a series of wildcards:

```
ls **/*.jpg **/*.png
```

This line right here says "list out the jpg and png files, any name, any directory". Your output should look something like this:

```
rob@Petew: ~/imposter/demos/task-images — .s/task-images — 2
→ task-images ls **/*.jpg **/*.png
images/doodles/3nf2.png          images/screenshots/calc_mac.png
images/doodles/gc-1.png          images/screenshots/calculator.jpg
images/doodles/gc-2.png          images/screenshots/difference_engine.jpg
images/doodles/lex-1.png         images/space/17071818163_66adaafda2_k.0.jpg
images/doodles/snowflake.png     images/space/17504334828_6d727a0ecf_k.0.jpg
images/screenshots/ace.jpg       images/space/17504602910_a939b425ba_k.0.jpg
images/screenshots/ae_plan.jpg
→ task-images
```

If this isn't what you're seeing, make sure you're in the correct directory. Also, be sure you entered the glob correctly as well.

OK, we're almost done. What we need to do now is to create a list that we can show our boss. To do that, we'll redirect the output of the command into a text file:

```
ls **/*.jpg **/*.png > images.txt
```

And we're done! If you want to see this file, you can use the command `open images.txt`, and you'll see them in your default text editor.

That wasn't so bad, was it? That one line saved us quite a bit of work, don't you think? How would you have done this using visual tools?

I just threw a lot at you, but I'm sure it wasn't that difficult. There are two things I want to highlight, however.

## Environmental Variables

I was using the term **\$HOME** a lot. This is a special place on a Unix machine – it's where you get to do whatever you want. Visually speaking, you can think of **\$HOME** as the place the Finder opens up to when you first open it. It's usually a place like /Users/rob (in my case). You don't ever work on the root of the machine – that's only for special users which we'll discuss later.

Look at your **\$HOME**. You can do this using the command **echo \$HOME**. The **echo** command simply outputs a value to the screen, in this case it will be whatever the **\$HOME** variable is set to. That's right – **\$HOME** is a variable, and a special one at that. It's called an "environmental variable" and there are many them. You can tell you're working with a variable in Unix because they have a **\$** prepended to them (this is parameter expansion, which I'll get into below). Other variables include **\$PATH** and **\$USER**.

We'll be working with variables of our own making later on.

## STDOUT and STDIN

The next thing I mentioned (but kind of glossed over) was that I redirected the output to a text file. I did this using the **>** operator. This is a crucial thing to understand when working with the shell:

there is a standard input and standard output. The standard input is the keyboard, the standard output is the terminal.

In the same way you can refer to **\$HOME**, you can refer to standard output as **STDOUT** and standard input as **STDIN**. This might seem a bit academic at this point, but if you think about working with a computer, in general, you give it information and it gives you something back. It does this with **STDOUT** and **STDIN**.

You wouldn't want to have to specify where you want the output sent every time you executed a command, would you? This is where **STDOUT** comes in. If you did want the output of a program to go somewhere, it's easy to specify. Which is what we did using **>**.

## Creating An Executable Script

When you're working in a shell you're working in a REPL (Read, Eval, Print Loop). If you don't know what that is – it's a way of working directly with a language. If you have Node installed on your machine you can type in "node" and you'll be in the Node REPL. From here you can enter all kinds of JavaScript code.

If you have Ruby installed on your machine you can enter "irb" in the terminal and you'll be in the Ruby REPL. The Unix command line is the same thing. Bash (or Z shell or whatever shell you're using) will expand and then execute the commands you give it, executing them directly. We'll get into command expansion more later on; for now let's keep rolling.

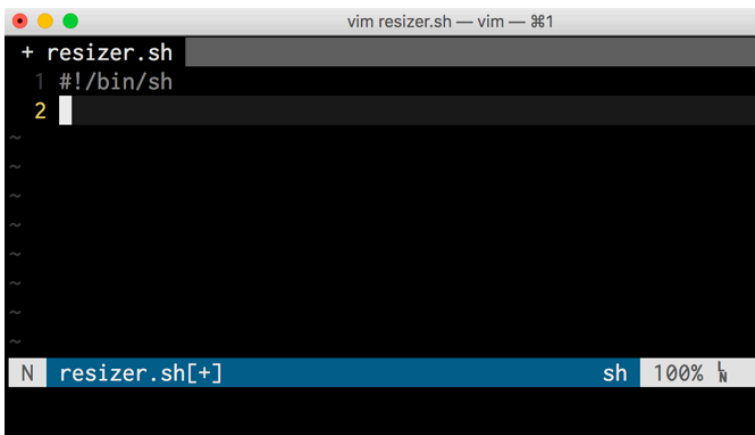
Create a new file called “resizer.sh” and open it up in Vim. If you’re not a Vim fan, use whatever editor you like – but I would challenge you to just give it a try, at least for this walkthrough.

You’ll want to be sure you’re in the same directory as before – the one with the “images” subdirectory. Then, open up Vim, passing it the file name you want:

```
vim resizer.sh
```

This will create a buffer in Vim, not an actual file. That won’t be created until we save the file.

The first thing we need to enter is a *shebang*, which is a great word don’t you think? It’s also called a *hashbang* by some. To do this in Vim, enter “i” to go into “insert mode” (so you can type some text) and then enter this at the top:

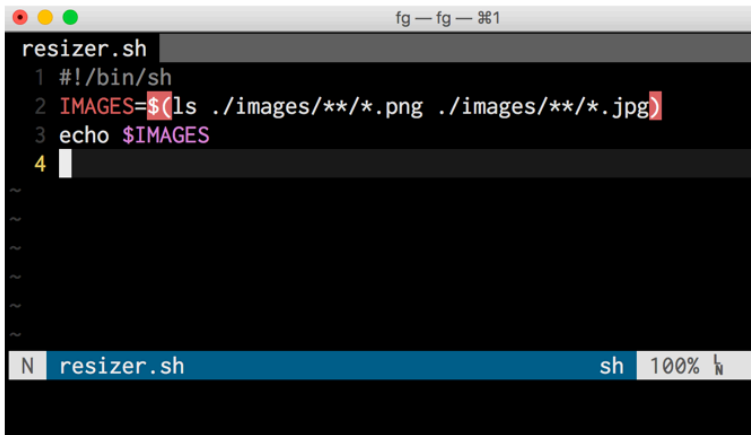
A screenshot of a Vim editor window. The title bar at the top reads "vim resizer.sh — vim — %1". The editor shows a new file named "resizer.sh" with a tab icon and the name. Line 1 contains the text "#!/bin/sh". Line 2 is currently empty, with the cursor at the start. The status bar at the bottom shows "N resizer.sh[+]" on the left, "sh" in the center, and "100% ↵" on the right.

When you're done typing, hit the ESCAPE key (<ESC>). This will return you to "normal mode".

Line #1 above is our shebang, it tells the shell what interpreter we want to use to run this script in the form of an absolute path. In this case, the interpreter is the shell itself, which uses the "sh" program to read in commands from the keyboard, a file, or **STDIN**.

The next thing we'll do is assign our image files to a variable. To make sure we've done this right, I'll output the result to the screen in the same way I might use **console.log** in JavaScript or **puts** with Ruby.

Your cursor should still be on line 1. If it is, enter "o" to create a new line and enter insert mode. You should be on line #2, where you can enter the following:

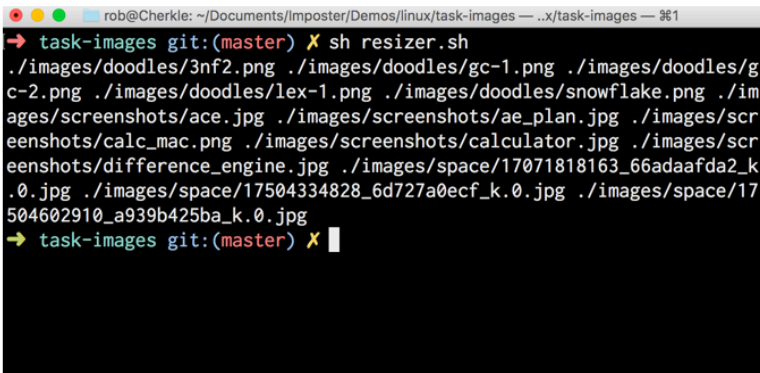


```
resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3 echo $IMAGES
4
```

As with last time, make sure you hit **<ESC>** to go back to normal mode. The next thing we need to do is save the file, and you can do that by entering **“:w”**, which means “write this buffer to disk”.

Let’s run it to be sure everything works, and then we’ll get to the explanation. Enter **CTRL-Z** to suspend the Vim session, flipping back over to the terminal. If you can’t get this to work for some reason, just open up a second terminal window and navigate to the same directory you’ve been working in – make sure you’re in the shell, not Vim. Our goal is to have the shell execute what we’ve just written.

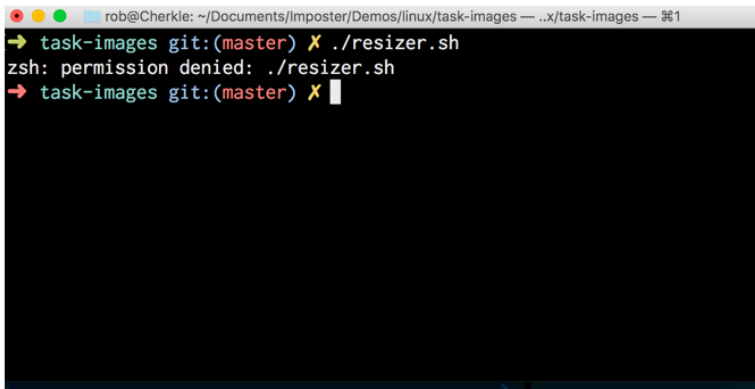
Enter **“sh resizer.sh”** into the terminal and you should see an amazing splash of text:

A terminal window with a dark background and light-colored text. The window title bar shows 'rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — .x/task-images — %1'. The prompt is 'task-images git:(master) X'. The command 'sh resizer.sh' has been entered and executed. The output is a long list of image file paths, including './images/doodles/3nf2.png', './images/doodles/gc-1.png', './images/doodles/gc-2.png', './images/doodles/lex-1.png', './images/doodles/snowflake.png', './images/screenshots/ace.jpg', './images/screenshots/ae\_plan.jpg', './images/screenshots/calc\_mac.png', './images/screenshots/calculator.jpg', './images/screenshots/difference\_engine.jpg', './images/space/17071818163\_66adaafda2\_k.0.jpg', './images/space/17504334828\_6d727a0ecf\_k.0.jpg', and './images/space/17504602910\_a939b425ba\_k.0.jpg'. The prompt is now 'task-images git:(master) X' followed by a cursor.

It worked! Or did it? We used the **sh** command, which feeds a file to the shell, the contents of which are expanded and executed. That’s kind of like executing it, but not really. It’s a bit like using

**eval** in Ruby or JavaScript to run a string of code, rather than executing it directly.

To properly execute a shell script, you just invoke it. Try entering “./resizer.sh” directly. This command simply gives the exact location of the shell script file, with the leading “./” indicating “this directory”. Doing this should lead to some problems:

A terminal window with a dark background and light-colored text. The window title bar shows 'rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — .x/task-images — %1'. The terminal content shows a prompt 'task-images git:(master) %' followed by the command './resizer.sh'. The next line shows the error message 'zsh: permission denied: ./resizer.sh'. The prompt is then shown again as 'task-images git:(master) %' with a cursor.

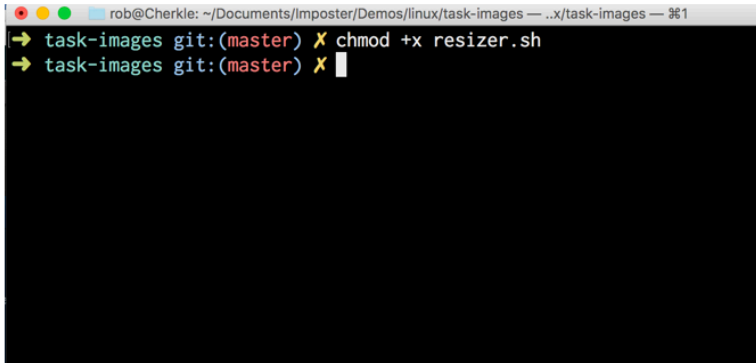
```
rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — .x/task-images — %1
task-images git:(master) % ./resizer.sh
zsh: permission denied: ./resizer.sh
task-images git:(master) %
```

By default, you can’t execute a file directly in the shell unless you specifically say it’s OK to do so. This is a security feature of Unix, as you might imagine.

To grant execution permissions you need to tell the operating system, and you do that by using the **chmod** command (change file mode). By the way, if you ever want to know more about any of the commands you see here, you can use **man** in the terminal itself. This shows the “manual” for each command. Try it now – enter “man chmod” and it will tell you all about it.



For our needs, I need to **chmod +x** our resizer, which means “add execute privileges to this file:



```
rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — .x/task-images — %1
→ task-images git:(master) X chmod +x resizer.sh
→ task-images git:(master) X
```

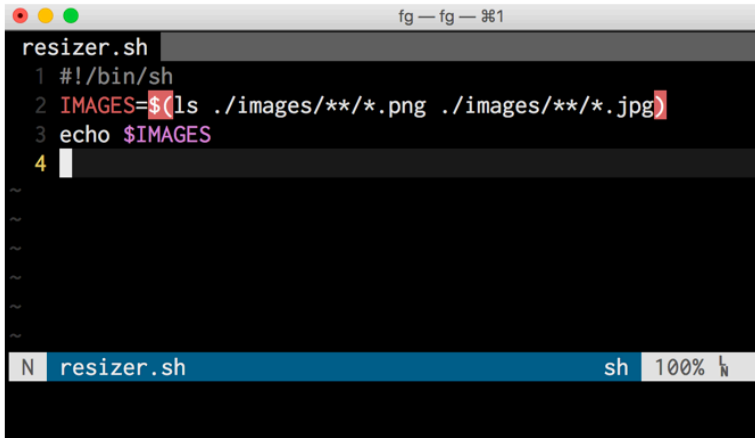
One thing to get used to with Unix: commands will typically not return any kind of result if they are successful. Silence, in this case, means all went well. Now we should be able to execute our script:



```
rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — .x/task-images — %1
→ task-images git:(master) X chmod +x resizer.sh
→ task-images git:(master) X ./resizer.sh
./images/doodles/3nf2.png ./images/doodles/gc-1.png ./images/doodles/gc-2.png ./images/doodles/lex-1.png ./images/doodles/snowflake.png ./images/screenshots/ace.jpg ./images/screenshots/ae_plan.jpg ./images/screenshots/calc_mac.png ./images/screenshots/calculator.jpg ./images/screenshots/difference_engine.jpg ./images/space/17071818163_66adaafda2_k.0.jpg ./images/space/17504334828_6d727a0ecf_k.0.jpg ./images/space/17504602910_a939b425ba_k.0.jpg
→ task-images git:(master) X
```

Nice work! Now let’s dive into the code some. To get back over to Vim, enter “**fg**” in the terminal to bring up the suspended app (**fg**)

means “foreground”). If you get a warning about the file being modified, just enter “l” to load it anyway. You should see this now:



```
fg - fg - %1
resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3 echo $IMAGES
4
```

We understand the first line, which is our shebang, but the second line looks a tad cryptic. Here, we’re setting a variable called **IMAGES** to the result of our image listing ... but what’s that syntax?

## Command and Parameter Expansion

When you surround a command with a **\$(..)** it’s called a *subshell*. As you can probably reason, I need to set the **IMAGES** variable to the result of the list command, which means I need to invoke it in place. I can do that by wrapping it in a subshell. This subshell will be expanded, and the results returned to the **IMAGES** variable.

We'll do more with subshells in a later section; for now, you can think of it as invoking a command in place and using its results directly.

The next line uses the `echo` command to output the value of the **IMAGES** variable to the screen. To use a variable (which should always be upper-cased), you must expand it as well using the `$`. This is called *parameter expansion* and might seem a little weird until you have a play with it.

Open a new terminal window and type in **THING=1**. This will set the variable **THING** to the value 1. Now let's use **echo** one more time to have a look at this value. Try entering **echo THING**.

What happened? The **echo** command doesn't know if you're giving a literal value or a variable – it's up to you to expand the value before **echo** gets ahold of it. You do this in the same way you expand a subshell: using `$`.

There are different ways to run subshells and expansions, and we'll get into that in a later section.

## For Loops

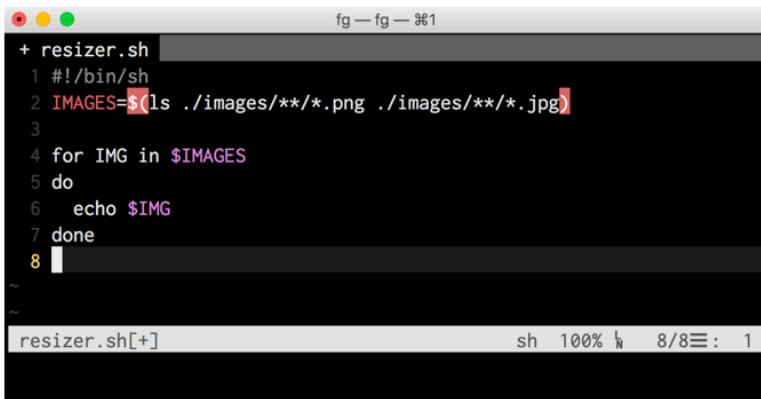
We have our list of images and, at some point, we'll need to operate on those images individually. This means we'll need some kind of loop – a **for** loop specifically.

We can do this with our shell. You should be in normal mode with Vim; if you're not, just hit **<ESC>** until you are. You can move

around the screen using the h, j, k and l keys. Give it a try, see what happens.

It helps me to think of the “j” as an anchor, pulling down and the “k” as a rock climber, clinging to the face of a vertical wall. Once you’re on line #3, enter “dd”, which will delete the line.

Now, enter “i” to get back to insert mode. We need to type some more code:



```
+ resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3
4 for IMG in $IMAGES
5 do
6   echo $IMG
7 done
8
```

This is our for loop. We declare a variable inline (**IMG**) and then create a **do** block, ending it with **done**. Inside this block we’ll report the value of the **IMG** variable. Hit **<ESC>** when you’re done entering this code, then type “:w” to save it. Now CTRL-Z to suspend and flip back to the terminal.

Execute again, invoking the file directly (or hit the up arrow until you see it):

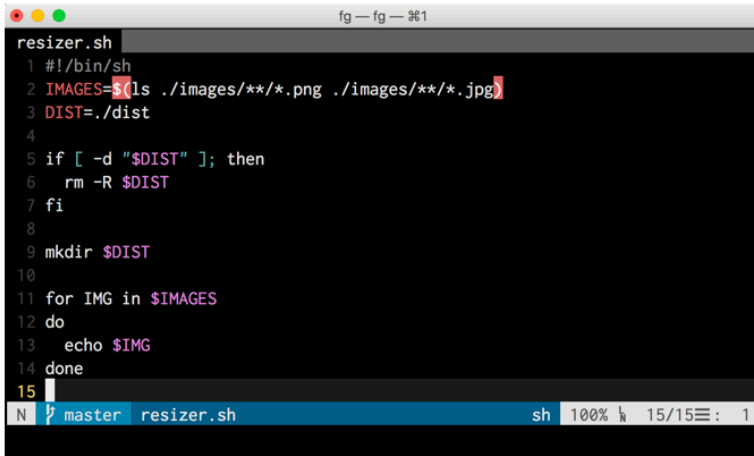
```
rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — %1
➔ task-images git:(master) X ./resizer.sh
./images/doodles/3nf2.png
./images/doodles/gc-1.png
./images/doodles/gc-2.png
./images/doodles/lex-1.png
./images/doodles/snowflake.png
./images/screenshots/ace.jpg
./images/screenshots/ae_plan.jpg
./images/screenshots/calc_mac.png
./images/screenshots/calculator.jpg
./images/screenshots/difference_engine.jpg
./images/space/17071818163_66adaafda2_k.0.jpg
./images/space/17504334828_6d727a0ecf_k.0.jpg
./images/space/17504602910_a939b425ba_k.0.jpg
➔ task-images git:(master) X
```

Nice! OK, let's keep rolling and pick up some speed.

## If Statements

We don't want to blow away our original files, so we'll create a destination directory, where all the modified files will be placed. We need to check if this directory exists before we do anything. If it does, we'll delete it, otherwise we'll just create it.

Use **"fg"** again to get back into Vim and then navigate to line #2, where the **IMAGES** variable is declared. Now enter **"o"** to open a new line below it. Enter the following:



```
resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3 DIST=./dist
4
5 if [ -d "$DIST" ]; then
6   rm -R $DIST
7 fi
8
9 mkdir $DIST
10
11 for IMG in $IMAGES
12 do
13   echo $IMG
14 done
15
```

The first thing is the most important: use variables for everything. It's bad form to hardcode values in a shell script! Here, I'm simply specifying where the output directory is going to be.

On line #5 I'm testing to see if this directory exists. Notice the spacing here? It's important! Make sure there's a single space between the brackets and the conditional statement. Also: notice the semicolon? That's optional – it signifies a code line termination. If I put them on a new line the semicolon isn't needed. I use it here because my eyes are used to reading **if** statements in this way.

Next: notice that I wrapped **\$DIST** in quotes? This is a subtle point and not one I expect you to remember entirely. If you write many scripts, however, you'll have a very interesting problem to overcome: *how do I control this expansion thing?*

## Quoting

We're using the command line, which is text-based, so Bash will take you literally when you type in anything. We need a way to work with text in this environment.

For instance, let's say I create a file with a silly file name:

```
touch super-*.txt
ls super-*. *
ls: super-*. *: No such file or directory
```

Now this file name and, indeed, this entire example is ridiculous ... sort of. I've seen some amazingly weird file names! Anyway: this command set will cause an error. We're confusing our shell because it can't tell the difference between our literal text **super-\*** and our wildcard placeholder **.\***. How do we get around this problem?

To "unconfuse" the shell we can use quoting:

```
ls 'super-*. *'
super-*.txt
```

I'm using a single quote here, which is referred to as "strong quoting", which means absolutely nothing within the string will have any special meaning to Bash. For our needs this is actually a bit too much (or I should say too literal). We want to have some expansion in there!

For instance, we might want to set the **\$DIST** directory to be something like **\$HOME/fixed** or something. We want **\$HOME** (and other variables) to be expanded in this context.

We can do this by using “weak quoting”, or double-quotes, which will allow us to access variables using **\$** (we can do other things too, like use a **\( \sim \)** for our home directory and **\** to escape things).

It’s good practice to use quoting when referencing variables as we’re doing here. If any of our directories or files contain characters that will confuse our shell, their expansion will be ignored.

OK, we’re almost done – let’s rock this out!

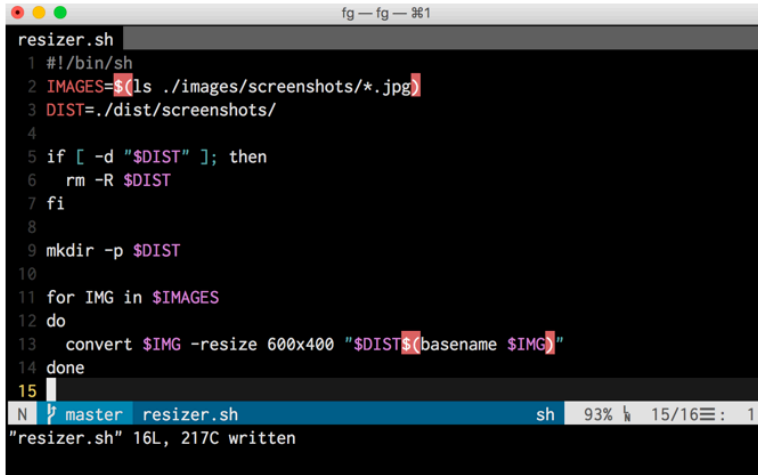
## Using ImageMagick

To run the actual image conversion, I’ll use ImageMagick, a popular open-source image manipulation tool. You can install it on your Mac using Homebrew (“brew install imagemagick”) or with MacPorts.

Once ImageMagick is installed, I simply need to use the **resize** command with some dimensions and an output.

We only need to resize our screenshots, so I’ll reset the **\$DIST** variable to only include them (I’ll change this later). Finally, I just call **convert** within our loop, and off we go:





```
resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/screenshots/*.jpg)
3 DIST=./dist/screenshots/
4
5 if [ -d "$DIST" ]; then
6   rm -R $DIST
7 fi
8
9 mkdir -p $DIST
10
11 for IMG in $IMAGES
12 do
13   convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
14 done
15
N master resizer.sh sh 93% 15/16: 1
"resizer.sh" 16L, 217C written
```

If we execute this script as before, we'll have a lovely new set of resized images waiting for us in the `./dist/screenshots` directory. Not bad for 15 lines of code!

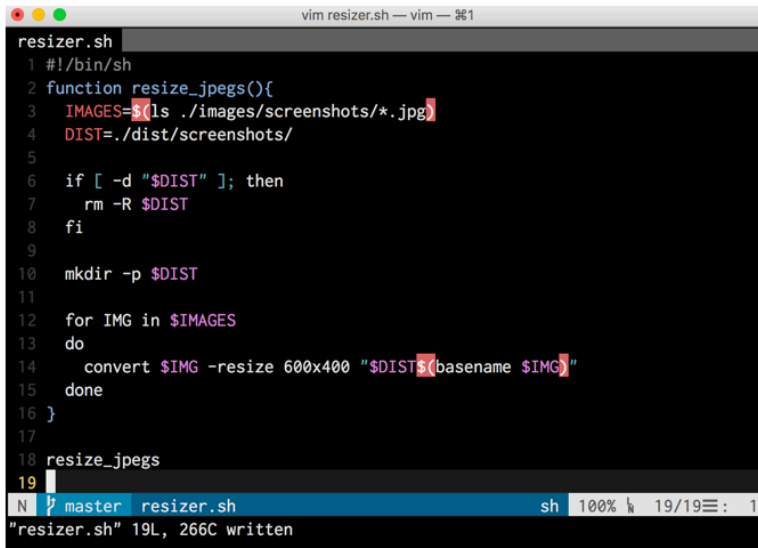
But we can do better.

## Using Functions

Being the good coder that you are, you're probably wondering why I hardcoded everything for the "screenshots" directory? Good for you!

What we have here is a very workable shell script, but it could be better. Just like any code that you write, think about modularity and reuse. In our case we have a lovely bit of functionality that I'll

probably want to use later. The good news is that I can do this by turning it into a function:

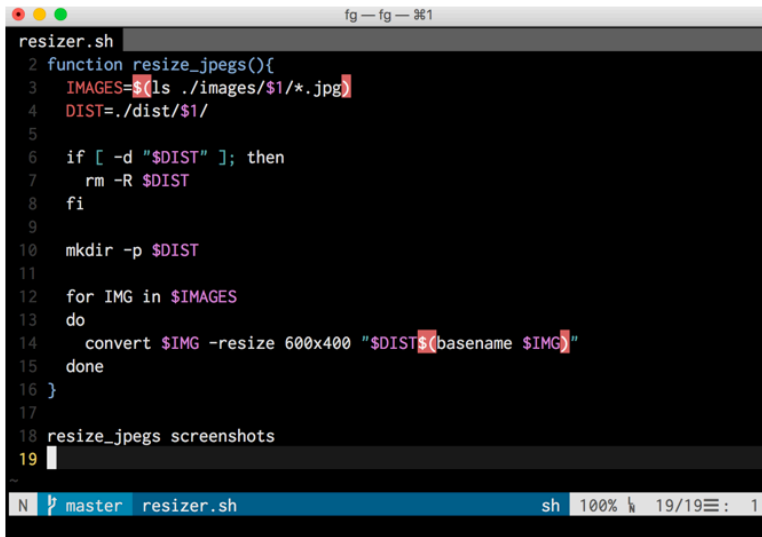


```
vim resizer.sh -- vim -- 961
resizer.sh
1 #!/bin/sh
2 function resize_jpegs(){
3     IMAGES=$(ls ./images/screenshots/*.jpg)
4     DIST=./dist/screenshots/
5
6     if [ -d "$DIST" ]; then
7         rm -R $DIST
8     fi
9
10    mkdir -p $DIST
11
12    for IMG in $IMAGES
13    do
14        convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
15    done
16 }
17
18 resize_jpegs
19
N ? master resizer.sh sh 100% 19/19: 1
"resizer.sh" 19L, 266C written
```

On line #2 I just declare the function, which looks a bit like JavaScript doesn't it? On line #16 I close it off with a brace and then I can call it directly on line #18.

This is neat, but it's not modular! For that I'll need to send in some arguments. You work with arguments positionally in a shell script, using parameter expansion like we did before with variables.

In this case I'll change "screenshots" to reference the first parameter passed to our function (**\$1**):



```
fg - fg - %1
resizer.sh
2 function resize_jpegs(){
3   IMAGES=$(ls ./images/$1/*.jpg)
4   DIST=./dist/$1/
5
6   if [ -d "$DIST" ]; then
7     rm -R $DIST
8   fi
9
10  mkdir -p $DIST
11
12  for IMG in $IMAGES
13  do
14    convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
15  done
16 }
17
18 resize_jpegs screenshots
19
```

Nice! To pass a value into the function you just tack it on to the function call, which you can see on line #18. I've replaced the hard-coded value with **\$1**, and we're good to go.

## Execution vs. Loading

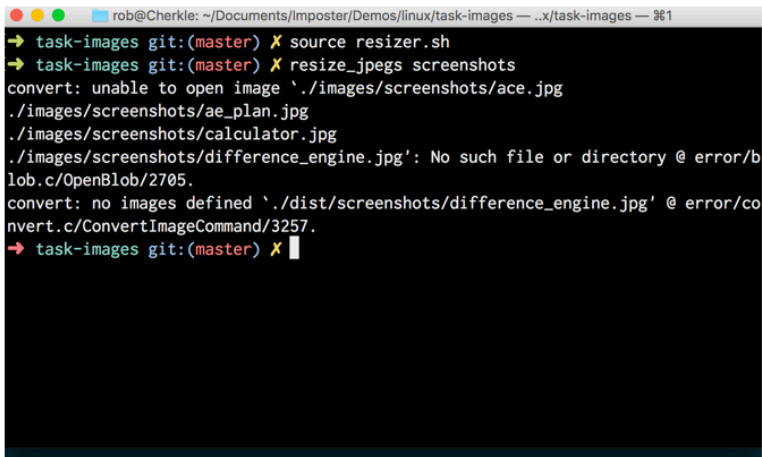
Right now, our script executes every time we call it, which is fine, but we might want this command available to us from anywhere. We can do this if we just tweak a few things.

First, let's get rid of line #18. This will prevent our function from executing each time. Then, instead of invoking our script file, we can just source it:

```
source resizer.sh
```

When you source a file, as we've done here, you load its contents into the current shell. This has the same effect as execution, essentially, but it does a bit more.

For us, we can now call our **resize\_jpegs** function from anywhere:



```
rob@Cherke: ~/Documents/imposter/Demos/linux/task-images — ..x/task-images — %1
→ task-images git:(master) X source resizer.sh
→ task-images git:(master) X resize_jpegs screenshots
convert: unable to open image `./images/screenshots/ace.jpg'
./images/screenshots/ae_plan.jpg
./images/screenshots/calculator.jpg
./images/screenshots/difference_engine.jpg': No such file or directory @ error/b
lob.c/OpenBlob/2705.
convert: no images defined `./dist/screenshots/difference_engine.jpg' @ error/co
nvert.c/ConvertImageCommand/3257.
→ task-images git:(master) X
```

Uh oh. Looks like I was wrong about that. What happened?

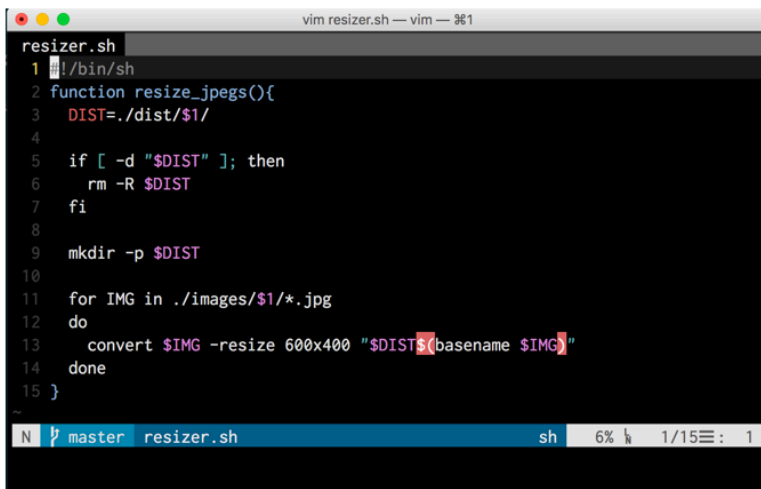
Remember above when I said that executing a shell script is sort of like loading it directly into the shell? That sort of just came back to bite me.

When you execute a shell script, as we did above, it's given its own process. Our **\$IMAGES** variable executes properly in a subshell and

returns a list of files, as it should. We can then loop over that and all is well.

When we source a script file, we load it into the current terminal session and the code we write in our shell script acts just as if we typed it in there ourselves. What this means is that the expansion we're using to load the **IMAGES** variable is viewed as string value. If you read the error output above, you'll see that ImageMagick is trying to convert a file identified by a huge string value!

We don't want that. The good news is that we can fix this easily by just looping over the glob directly:



```
vim resizer.sh — vim — 261

resizer.sh
1 ./bin/sh
2 function resize_jpegs(){
3     DIST=./dist/$1/
4
5     if [ -d "$DIST" ]; then
6         rm -R $DIST
7     fi
8
9     mkdir -p $DIST
10
11     for IMG in ./images/$1/*.jpg
12     do
13         convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
14     done
15 }
~
N master resizer.sh sh 6% 1/15: 1
```

Confusing, isn't it? You'll run into these problems, and the only way to solve them is to think about which process is executing them: your current shell or a spawned shell?

If you're still confused, hang tight – we're going to do more in the next section. For now you have a usable script! Can you see improvements to be made? What about the sizing? Or the input directory?

Have at it. Playing with shell scripts can be fun, and if you get stuck or need to try something new, there are plentiful resources online.

# MAKE

**M**ake is a build utility that works with a file called a “Makefile” and basic shell scripts. It can be used to orchestrate the output of any project that requires a build phase. It's part of Linux and it's easy to use.

It's important to understand that the utility, make, is not a compiler as many people believe. It's a build tool just like MSBuild or Ant.

Make is an extraordinarily simple tool which, combined with the power of the shell, can greatly reduce the complexity of your application's build needs. Even if you're a Gulp/Grunt/Whatever fan, you should understand the power of make, as well as its shortcomings.

# THE BASICS

Make will turn one (or more) files into another file. That's the whole purpose of the tool. If you run `make` and your source hasn't changed, make won't build your output.

Make runs on shell commands, orchestrated using the concept of "targets". Let's have a look.

First, create a directory where we can work and then create a Makefile:

```
mkdir make_demo && cd make_demo  
touch Makefile
```

Like many build utilities, a single file with a particular name drives the process. With Grunt, it's a Gruntfile; with Rake it's a Rakefile. Ever wonder where that convention came from? Yep: *it's Make*.

## ANATOMY OF A MAKEFILE

All of our build commands will go into the Makefile. Let's create the basic skeleton:

```
all:

clean:

install:
```

What you see here are targets: the things that Make will try to build for you. By convention they are named based on the file they are building – in some cases, like ours here, they are directives and don't build anything.

Every Makefile should have these directives:

- **all**: builds everything
- **clean**: cleans up all the existing build artifacts; usually deleting the files built

It's typical, as well, to have these directives as well:

- **install**: installs whatever built files are generated
- **.PHONY**: lists out the directives that don't create a file

Have you ever downloaded software from source onto a Linux box and had to run **make && make install**? This is why. The downloaded source (usually C or C++) is compiled and then installed wherever binaries go in the system, which can be /opt or somewhere else such as /usr/local or /usr/bin.



With our example we won't be installing anything so we don't need the install target. In its place we'll add another directive that tells Make which of the targets don't create a file. For this we'll use .PHONY, for "phony" targets:

```
all:  
  
clean:  
  
.PHONY: all clean
```

## UNDERSTANDING TARGETS

Make builds output files from input files. It was originally designed for C programs, which utilize both code and header files which are built into object files. These object files are then compiled to binary. This is a multistep build that requires some orchestration. That's what Make is all about.

Sometimes, however, you'll want a build step that might transform some input – it might not create a file. Letting Make know about these special (phony) targets will increase performance greatly. We'll get more into this in the next chapter.

This is a silly demo, but it's critical you see the way things work before we dive into the good stuff. Let's create a file in the root of our project directory that we want to transform and distribute. We'll

leave the file empty for now – let's also create a Makefile while we're at it:

```
echo "//some code" > app.js && touch Makefile
```

The simplest first step would be to copy our app.js code file to a /dist directory. So let's do it:

```
all:
    mkdir -p dist
    cp app.js dist/app.js

clean:
    rm -rf dist

.PHONY: all clean
```

The indents in a Makefile must use tabs. If you don't use tabs you'll get an error about an invalid separator. If you copy/paste the code here, be sure you indent with a tab that doesn't get translated to spaces.

OK, save the Makefile and run make:

```
rob@Cherkle: ~/Documents/Imposter/Demos/linux/make — ..os/linux/make — 1
→ make git:(master) X make
mkdir -p dist
cp app.js dist/app.js
→ make git:(master) X
```

Yes! Make outputs each command as it's executed – so here it's reporting that it ran `mkdir` and `cp` successfully. This can be good ... it can also be annoying.

Let's fix the chatter and provide a clean target while we're at it:

```
all:
    @ mkdir -p dist
    @ cp app.js dist/app.js

clean:
    @ rm -rf dist

.PHONY: all clean
```

By prepending an `@` sigil to a command line in our Makefile, we're silencing the output. Also - our clean target will delete the entire build directory. We can test that by running **make clean**.

Try it out!

# ORCHESTRATING THE BUILD

Our code files are a bit messy - just stored right in the project root. So, let's clean things up with some organization:

```
mkdir src mv app.js src
```

Great. Let's assume (for now) that all our project code goes into the `/src` directory.

Our current Makefile is not really doing any build orchestrations of any kind - it's just copying a single JavaScript file to the `/dist` directory. Let's change that by adding a build timestamp to the output, so we know when the last build was run.

Let's clean things up a bit so we have a build target that produces a file and another that produces the destination:

```
all: dist app.js

dist:
    @ mkdir -p dist

app.js:
    @ cp src/app.js dist/app.js

clean:
    @ rm -rf dist

.PHONY: all clean
```

A lot just went on there – and with it we get to learn some more jargon.

First, I created two new targets called `app.js` and `dist`. Targets in a Makefile are just the labels; what comes after the target is the recipe. So, for the `dist` target, **`mkdir -p dist`** is the recipe.

The **`all`** target has prerequisites that appear on the first line, but it has no recipe. Prerequisites are targets that must be built before creating the current target. So, for **`all`** to work, **`dist`** and **`app.js`** need to have run first.

If you put a target, recipe and prerequisites together, you have a *rule*. Which is precisely what a Makefile is: *a set of rules for building your software*.

This is the power of Make. Orchestrating what happens when, and in what order. You could say that this is all that Make does, which

isn't surprising if you understand the Linux philosophy of "do one thing well".

## USING VARIABLES

We're dealing with shell scripts here, and just like any programming effort, repeating ourselves is typically frowned upon. Let's take the hard-coded stuff out first so we can change as we need to.

By convention, variables should be declared at the top of your Makefile:

```
JS_FILES=src/app.js
DIST=dist

all: dist app.js

dist:
    @ mkdir -p $(DIST)

app.js:
    @ cp $(JS_FILES) $(DIST)/app.js

clean:
    @ rm -rf $(DIST)

.PHONY: all clean
```

Now that's starting to look like a proper Makefile! Variables in any shell script should typically be upper-cased.

Notice how I must use **\$(DIST)** to reference the variable? Do you remember what that is? It's a *command-replacement subshell*. Make runs all the commands in a subshell, outside its process.

At first glance it might not look like we've done much here – but if we change our source files, which we will, it's a simple change to the **JS\_FILES** variable.

OK, one last thing. We're still repeating ourselves in a couple of places – specifically with the target and the destination file names of **app.js**. If we're leaning on convention, we shouldn't have to specify things twice.

To get around this, we can use the **\$@** shorthand – which means “this target name”:

```
JS_FILES=src/app.js
DIST=dist

all: dist app.js

dist:
    @ mkdir -p $@

app.js:
    @ cp $(JS_FILES) $(DIST)/app.js

clean:
    @ rm -rf $(DIST)

.PHONY: all clean
```

We're getting there. Now, let's create our timestamp. For this I'll use a variable straight away, and put it right at the top:

```
JS_FILES=src/app.js
DIST=dist
TODAY=$(shell date +%Y-%B-%d)
TIMESTAMP="//Created at $(TODAY) \n\n"
#...
```

Here I'm using the shell command, which, if you recall from previous sections, executes shell commands for you. We need to get the literal value of the date and the date's formatting instruction, and store it in the **TODAY** variable.

We then create our timestamp. Using variables is a great way to keep your code clean and understandable, and that's critical in Makefiles as the syntax can quickly become overwhelming.

The next task is to read the source file and timestamp it:

```
#...
app.js:
    @ echo $(TIMESTAMP) > $(DIST)/$@
    @ cat $(JS_FILES) >> $(DIST)/$@
#...
```

This is a bit roundabout, but it works. As I keep mentioning, there's almost always a better way when it comes to shell scripting – and we should most definitely extend that idea to Makefiles. This is



nice and clear to me, however, and hopefully it makes sense to you as well.

I'm redirecting the output of **echo** (which is the **TIMESTAMP** variable) into **dist/app.js**, which is our output file. On the next line I'm appending **STDOUT** using **>>** to the same file – this time with the contents of the JavaScript files in our **src** directory.

Here's another go at this:

```
JS_FILES=src/app.js
DIST=dist
TODAY=$(shell date +%Y-%B-%d)
TIMESTAMP="//Created at $(TODAY) \n\n"

all: dist app.js

dist:
    @ mkdir -p $@

app.js:
    @ echo $(TIMESTAMP) > $(DIST)/$@
    @ cat $(JS_FILES) >> $(DIST)/$@

clean:
    @ rm -rf $(DIST)

.PHONY: all clean
```

This is a solid Makefile, but it's not quite there yet. Notice that I have a **dist** target and a **DIST** variable? This is redundant! You can

use variables as target names, so let's do that and arrive at our final Makefile:

```
JS_FILES=src/app.js
DIST=dist
TODAY=$(shell date +%Y-%B-%d)
TIMESTAMP="//Created at $(TODAY) \n\n"

all: dist app.js

dist:
    @ mkdir -p $@

app.js:
    @ echo $(TIMESTAMP) > $(DIST)/$@
    @ cat $(JS_FILES) >> $(DIST)/$@

clean:
    @ rm -rf $(DIST)

.PHONY: all clean
```

Let's give it a run using **make clean** && **make**. You should see a **dist/app.js** file with this in it:

```
//Created at 2016-October-19

//some code
```

Nice work! Once again, this is a bit of goofy demo, but your brain should be racing a little now... thinking about all the ways you might put Make to use with your project.

In the next section we'll do even more work with JavaScript files, kicking Grunt right out of our project.

## USING MAKE TO BUILD YOUR WEB APP

Every language/platform seems to have its own build and automation toolset. Microsoft has MSBuild, Java has Ant, Maven, Gradle, and perhaps a few others. JavaScript has Grunt, Gulp, Jake and a few more. Ruby has Rake. Why is this?

The short answer is that Make can be a little opaque and, predictably, language users like to build things using their own language and believe it will impart some type of benefit over the decades-old Makefile.

This subject, like others in this book, is a tad volatile and I want to recognize that right up front. An easy way to start an argument between two developers is to bring up the subject of build automation! A good friend of mine, Rob Ashton, thought it would be fun to submit a pull request to the MSBuild GitHub Repo, suggesting that it be replaced with Make. If you've ever had to wrestle with MSBuild before (which I have, numerous times), then this might

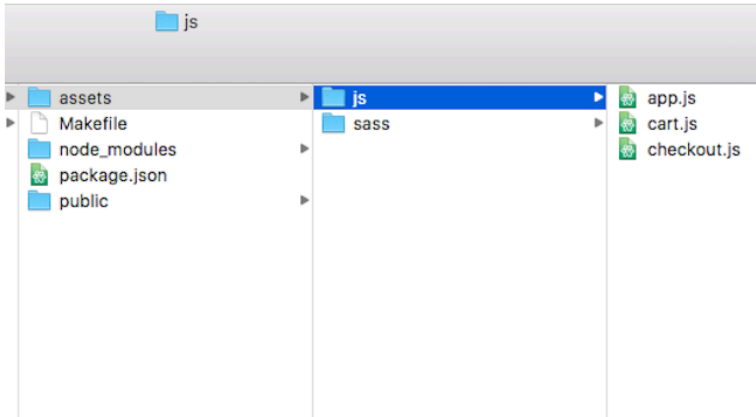
make you laugh. I thought it was funny, but there are many who did not.

No matter which build tool you use and love, knowing Make will give you *a perspective that you need*. It's built right into Unix-based systems and you can use it to automate just about anything. Whenever you crack open that Gruntfile, Gulpfile, or Rakefile, ask yourself whether the layers of complication you're adding to your project by using these tools are really warranted. The answer, typically, is no.

The problem, typically, is that most people don't know how Make works. Let's change that now, using a fairly typical use case: compiling assets for a web project.

## **The Web Project**

I have a typical web application that uses SASS and a bit of JavaScript:



As you can see, it's a simple app. I have a set of JavaScript files and a single SASS file that I need to build and then output to a directory somewhere.

This is my goal for this task: *concatenate, compress and build my SASS files and then concatenate and uglify my JavaScript files.*

## Working With SASS

Some people dig SASS, others LESS. Other people (like myself) tend to knuckle under and just write CSS directly. Hopefully what you're about to do will translate to how you develop things.

Also: when working with SASS and LESS it's common to import all of your files into a main.scss file, which you then build with the interpreter. That's what I'll be doing today.

## Step 1: The Node Modules We Need

I need to install a few Node modules to help me out:

- **node-sass**: the module I need to compile and build my SASS file
- **uglifyjs**: removes whitespace, concatenates and then minifies my JavaScript files

That's it. To install these we use NPM:

```
npm install uglifyjs node-sass --save-dev
```

## Step 2: Our Makefile Skeleton

The next step is to create the Makefile in the project root and then give it a default structure:

```
touch Makefile
```

Open the Makefile and then add the default structure – the targets we know we're going to be working with:

```
all:

clean:

.PHONY: all
```

Hey we're getting good at this!

### Step 3: Define The Variables

We don't want any hardcoded values in our rules, so let's define as much as we can at the top:

```
SASS=node_modules/.bin/node-sass
SASS_FILES=assets/sass/main.scss
JS_FILES=assets/js/*.js
UGLIFY=node_modules/.bin/uglifyjs
DIST_CSS=public/css
DIST_JS=public/js

all:

clean:

.PHONY: all
```

I'm defining the binaries, the source files, and the destinations.

### Binaries

I've installed the **node-sass** and **uglifyjs** binaries locally as development dependencies. This will increase the time it takes to run npm install, but it also guarantees that the binaries will be present. It's really annoying to have to install global dependencies (my opinion).

In case this is all new to you or if you've never used Node: you can install packages from NPM and have them run locally or globally. You've probably seen a node\_modules directory at some point in your career – this is the local package installation location. If I was to install node-sass locally I could run it by using node ./node\_modules/.bin/node-sass. Or I could make life easy on myself and install it globally in the global NPM cache, which would allow me to run the binary anywhere on my machine.

## Step 4: Define The Targets

Given that I want to smash all the JavaScript files into a single app.js file, I can create that target. Same with app.css. The destination for these files will be the public directory, which should exist already in my project. However we can't guarantee that so let's make sure that we have a **dist** target as well:



```

SASS=node_modules/.bin/node-sass
SASS_FILES=assets/sass/main.scss
JS_FILES=assets/js/*.js assets/js/**/*.js
UGLIFY=node_modules/.bin/uglifyjs
DIST_CSS=public/css
DIST_JS=public/js

all: dist app.css app.js

dist:
    mkdir $(DIST_CSS) $(DIST_JS)

app.css:

app.js:

clean:
    @rm -rf $(DIST_CSS) $(DIST_JS)

.PHONY: all

```

Simple enough. Creating the directories, we need with the **dist** rule, removing them with **clean**. Now all we need to do is to fill in the commands to create the files<sup>14</sup>.

---

<sup>14</sup> *Loading up your source files like this is simple when using a glob pattern, but you might have different needs. For instance: you might have an `app.js` file in `assets/js` that you want loaded first, and then all the other files loaded after that. You can do this easily by specifying **`assets/app.js`** as the first file in the list, and then the glob **`assets/js/**/*.js`** after that.*

## Step 5: Building The Files

The first step is simple: we only need to compile the main.scss file using node-sass, which comes with a binary in the node\_modules/.bin directory.

This is what the command would look like normally:

```
node_modules/.bin/node-sass --output-style compressed assets/sass/main.scss > public/css/
```

We just need to replace this command with our variables and we're good to go:

```
app.css:  
@ $(SASS) --output-style compressed $(SASS_FILES) > $(DIST_CSS)/$@
```

Bam. That couldn't be easier!

OK, I lied: *it could be easier*. You'll find that most of your time is spent wrangling the commands together – reading the docs, trying to understand how to use the binaries properly.

When I first put this file together for a project I was working on, I had to read through the source code of some binary files to understand all the options and how they worked. I then ran those commands from the command line to make sure things happened the way I wanted.

The JavaScript files are a bit of a different story. I need to concatenate them together, and then uglify/compress them. Concatenat-

ing is simple – we can use `cat` directly to pull the code from each file – which we can then pipe directly to **uglifyjs**:

```
app.js:
@ cat $(JS_FILES) | $(UGLIFY) > $(DIST_JS)/$@
```

We haven't discussed the `|` notation you see here. That's called the pipe operator and is a Unix operator that redirects the output of one command to the input of the following command. In the same way I redirected **STDOUT** to a text file using `>` in a previous section, the `|` operator redirects **STDOUT** and **STDIN** for two given functions.

In this example, I'm using **cat** to concatenate all the JavaScript files specified by my **JS\_FILES** variable. I'm then piping that text into the **uglifyjs** binary, and then redirecting the output of the **uglifyjs** call to my destination.

I wish I could tell you it was harder than this ... but it's not. This is the power of Make. Run `make` and look at the built output in the public directories. If you've already run it, be sure to run **make clean && make** before you run `make` again to avoid errors.

## A Little Cryptic?

If this seems a bit cryptic or if your reaction to this code is something like "yeah great for you – you know shell scripts!" you might be interested to know that I'm a complete n00b to this stuff. I

learned it better over the last year but creating Makefiles like this was absolutely something that was beyond me just 8 months ago.

It just takes a little time, some Googling, and some practice and you will get it too.

## Ordering Of Files

Ideally your JavaScript files are not dependent on load order, but in the Real World this is typically not the case. There are several ways to get around the problem and they involve knowledge of some basic commands.

Personally, if I need files loaded in a particular order (as with my SQL files – tables need to be built before views and so on) – I just name them in a particular way:

- **01-init.sql**
- **02-tables.sql**
- **03-views.sql**
- **04-funtions.sql**

This works fine for SQL files, but it might not work for your JavaScript build. If you need files loaded in a particular order you can pipe the result to **sed**, as we did with the Jekyll task, and then transform it from there.

You can also separate the build steps, so you build a directory first, then another, outputting interim files to `assets/tmp` directory, which you then concatenate later.

## **Summary**

In this section, we've touched on some very fundamental ways we can use Make to automate and simplify our lives. But we have only scratched the surface. Make has a long, long legacy, and is the spiritual great-granddaddy of newer tools like Grunt, Gulp, and other automation tools we take for granted. I can't overstate the value that even a basic fluency with this tool will bring to your understanding of how source files are assembled into working software.

# FINAL THOUGHTS

**T**his book began as a series of tasks in Wunderlist: *things I need to know someday*. I didn't intend to write a book. I thought maybe I would write some blog posts, perhaps do a video or two on a few of the topics I found.

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.

There is comfort in not-knowing something. More than that: there is magic. Even more than that I think there is also a **bit of dumb courage**. You just don't know if it will hurt – so you try it out.

I'm reminded of being 15 and jumping off the roof of my house with an umbrella. Along with wondering if the umbrella would slow me down, *I also wondered how badly it would hurt*. Turns out the umbrella did nothing and jumping off the roof of my mom's house did, in fact, hurt. I didn't break anything or seriously injure myself, which is a good thing ... but I only tried the umbrella jump once. The next time I did it without the umbrella ...

I picked the cover of this book because that's what it felt like when I learned to program on my own: *jumping off something rather high*,

*hoping the landing wouldn't break me somehow.* I had a job and a budding career as a geologist – but it was boring. A good friend of mine was a trainer – the very first one certified by Microsoft to teach Active Server Pages. So he taught me (thanks Dave!) and I **jumped**.

I've hit the ground hard a few times, but the joy of jumping into new things is my drug. Which is kind of a problem because new things don't stay new for very long, and soon you start to feel the magic of it all start to fade.

It's the curse of learning: remove the mystery, remove the magic. Move on to the next mystery. Pretty soon you wonder what the point is.

However.

The process of writing this book has been *transformative*. I'm going to push the whole jumping metaphor a little more – and I'm also going to reflect on the cover image a bit more too. You may not have noticed, but on the back of the “jumper” on the cover where a parachute should be is ... this book. If you look closer, you can see the same cover. I felt it appropriate as recursion seems to ... recur throughout this book.



More than that, though, is the idea of a parachute. This book has become my parachute, in a sense – the things I've learned over the last year are allowing me to jump off higher rooftops – but this time I have a bit of knowledge to soften the fall.

Climb higher, jump farther. The mystery/magic/dumb courage seems to be increasing as I learn these things.

## ONE FINAL THOUGHT

I've gone deeper into these topics than I ever thought I would. I feel like I have a solid grasp on complexity theory, Big-O, graphing



algorithms and Bernoulli's Weak Law of Large Numbers and ... just writing this sentence is giving me a rush! This shit is BREATHTAKING!



Sorry. I promised myself I wouldn't swear in this book – but if any of the paragraphs written deserve some caps and a four-letter word ... well it's that one.

### **There. Is. So. Much. Magic.**

Go find it. Don't stop here. It's so easy to fall into snark and uncertainty; to feel like *you just don't know what other people are talking about*. It can really be isolating and lead one to look upon others with scorn for trying new things. As I write this sentence, Facebook just pushed a new package manager for JavaScript. As expected, quite a few people ridiculed it and laughed without even trying it!

*Let's not be those people.* We're the explorers who keep trying (and hopefully failing). We push the edges to find out what's possible. Enjoy this!

I urge you to explore and dig deeper, ask more questions, take a friend out and bore them to tears with all you know! Draw a picture of Traveling Salesman for your kids and ask how they would solve it. Make a Markov Chain out of popcorn for the holidays!

With that: *I leave you.* Thank you for going on this journey with me. Now let's go make a difference.

