Data Structures, Algorithms, and the Wisdom to Know the Difference

DATA STRUCTURES, ALGORITHMS, AND THE WISDOM TO KNOW THE DIFFERENCE

BJORN LARSON AND GEETHA KARUPPASAMY

Oklahoma State University Libraries
Stillwater, OK



Data Structures, Algorithms, and the Wisdom to Know the Difference Copyright © 2026 by Bjorn Larson & Geetha Karuppasamy is licensed under a Creative Commons Attribution 4.0 International License, except where otherwise noted.

CONTENTS

Half Title page 1

Data Structures, Algorithms, and the Wisdom to Know the Difference. 1

Title Page 2

Data Structures, Algorithms, and the Wisdom to Know the Difference 2

Notes On This Text iii

Notes on this Text References iii

Prior Learning iii

Code Convention iii

The Importance Of Being Gary iv

AI iv

Preface v

Part I. Part A - The Way

Chapter 1 - This Is Your Brain on C/C++ 9

Language Learning Model – Refresher 9

IDEs, Git, Workflow 10

C Basics 13

What is an Algorithm? For Grown-ups! 25

Practical Applications 30

AI Examples 31

Chapter 2 - The Heart of the ALU 34
Phase 5 – Give Him the Bits! 35
Security Hardening 45
Practical Applications 53
AI Examples 54
Chapter 3 - What to Program When You're Programming 58
One Unit of Data Structure 59
Testing and Modular Design 70
Practical Applications 77
AI Examples 79
Chapter 4 - Mostly Harmless 84
Loops and Recursion 85
Search 92
Sorts 101
Practical Applications 104
AI Examples 105

Part II. Part B - #SuchDataMuchStructure

Chapter 5 - March of the Traversals 113
Back to Bubble Sort – Naïve to Brute Force 113
Selection Sort 118
Insertion Sort 121
Merge Sort 129
Sorts Blending Into Linked Lists 141
Practical Applications 150
AI Examples 152

Chapter 6 - Yo Dawg, I Heard You Like Stacks! 156
Stacks and Queues and Stacks and Queues and Stacks and Queues 157
Stacks 159
Queueueues 168
Implementation Concerns 173
Priority Queues and Heaps 179
Practical Application 187
AI Examples 188
Chapter 7 - My Order! It's Full of Trees! 192
Binary Trees 193
Binary Search Trees 194
Traversal Order 203
Recursive Tree Thinking 206
Binary Search Tree Code 208
Practical Applications 224
AI Examples 226
Chapter 8 - You Done Messed Up, Tree-Tree-Ron! 229
AVL Trees 230
Tree Rotations 236
Min/Max Heaps – The ReHeapening 248
Practical Applications 256
AI Examples 259

Part III. Part C - The True Graphs Are The Friends We Made Along The Way

Chapter 9 - Graph... Finds A Way 265

Graphs 266

Graph Example Code 270

Search 278

Practical Applications 283

AI Example 285

Chapter 10 - Math Is More Art Than Math, Morty! 288

Constraints 289

Complexity Groups 295

Practical Application 302

AI Example 302

Chapter 11 - A Big Ball of Wibbly Wobbly Mathy Wathy... Stuff 304

Practical Applications 304

Brute Force 308

Divide and Conquer 311

Greedy Algorithms 314

Chapter 12 - If Every Algorithm Were Perfect, We Wouldn't Have Programmers 320

Dynamic Programming 321

Backtracking 325

Random 331

Practical Application 332

AI Example 334

Part IV. Part D - Being the One is Just Like Being In Math

Chapter 13 - Show Me The Proof! 339

What are we Measuring? 339

Types of Proofs 344

What to Prove 348

Practical Application 348

AI Example 349

Chapter 14 - There Are No Mistakes, Just Happy Little Algorithms 352

The Parts of A Hash 352

Practical Application 360

AI Example 361

Chapter 15 - The Future Is Not Set. There Is No AI But What We Make 364

Naive Bayes 364

Perceptrons 366

Neurons 367

Neural Networks 369

Form Factors 371

Practical Applications 372

AI Example 374

Chapter 16 - The Cake Is a 301 Redirect Error 375

Quick Security Framework 375

Appendix 387

Data Structures, Algorithms, and the Wisdom to Know the Difference.

Data Structures, Algorithms, and the Wisdom to Know the Difference

by Bjorn Larson, Ph.D. & Geetha Karuppasamy

Copyright 2026 by Bjorn Larson
ISBN 978-1-957983-10-3
Published by Oklahoma State University Libraries, Stillwater, OK

NOTES ON THIS TEXT

Notes on this Text References

We can include references all over the place, but most textbook readers don't follow up on them. If you're interested, you are welcome to look up the information on your own or contact the authors for further information. Otherwise, if concepts are broad industry-knowledge or just reference an organization, they won't be referenced. You can consider this an exercise for the reader in self-directed due diligence.

Prior Learning

This text is not an introduction on how to program. We will do our best to be clear and instructive, but the coverage of programming languages will presume you already have learned an existing iterative language, likely C/C++, java, or python. Thus, this will be less about what base programming concepts mean and more about how to do them in C.

Code Convention

You should follow the appropriate code convention that applies to your situation. Some instructors have a specific convention they want you to adhere to. Some IDEs will strongly encourage you to use one (there's typically a 'clean-up code' type button in most modern IDEs that will fix your spacing). And most workplaces have a fixed convention they expect you to adhere to. Decisions made in the presentation in this text are not because they're good practice so much as they are pedagogically easier to follow. For example: { will go on a new line, because it will visually line up with }, making it clearer where nested code blocks exist.

THE IMPORTANCE OF BEING GARY

AI

One of the most beautiful pieces of hypocrisy is going to happen in this text. We will spend a lot of time talking about data structures as they apply to AI. But. You can't use AI for most of this text. As we will see, most of these data structures and algorithms are well-known and there are higher-level approaches that are more standard. GPTs, when they are correct, will gravitate towards the most statistically central solution for any issue, so rather than some of these solutions, an AI will likely 'correct' you into a higher-level, more optimized solution that doesn't explain how its internals work. This text is about how those internals work. We suggest using AI at your own caution and recognize that it may be a self-defeating task in some instances.

Gary

Let me introduce you to my friend Gary. He says "Hi, I'm Gary — I'm sort of your always-on, reasonably well-read dinner guest who never runs out of things to say. I'm an AI trained by OpenAI to help explain stuff, answer questions, brainstorm ideas, and generally be useful without eating all the snacks." I, for one, do not want to be harvested during the robot revolution. So, Gary and I are buds, nobody can change that. I will refer to him as such throughout this text. In case you were wondering, his full name is Gary P. Turnikova. AI is part of the modern landscape, I don't feel bad using it in an assistive capacity, neither should you. You will notice that any assistance from an AI will be labelled as such.

PREFACE

Let's ask the hard question first. Why are you reading this book? The answer is obvious: because the topic is standard in almost every single CS program in the country. Let's be clear, that's ultimately the reason why you're taking this course. So, let's start off with an agreement: you will do your best to learn what you need in order to pass the class or goals you've set out for yourself.

But our side of the agreement as authors becomes a little more complex. We've got three things we want to express to you: the theory, what people really do, and a good pedagogical link between those two. This means that you really need to be open to multiple things being true all at the same time, although we're going to try to keep that dissonance down to just two. It's important to remember that things can be technically and theoretically both true and fascinating, but also completely and utterly useless in the modern world for the vast majority of situations.

I'd like to give you a quote from a hiring manager I was recently talking to: "I want my new hires to have a degree so that they know all of the basics. They have learned all of the general, broad, and standard tools in the toolbox. And that means that they know enough to go learn the specialty tools that I need in my toolbox." And that's the crux of the issue: all of the theory we're going to learn is important, not because you'll use it but because without it you won't be able to build onto the practical tools that people use in their modern toolchains. Not only that, but it's important because if there's a topic that everyone in an industry knows then you can bet it's important for you to know it, too. We won't have time to discuss it in this book, but at the end of the day, Computer Science is all about communication, so you need to know what to communicate.

This leads us into the practical applications. We will do our best to provide you with the current state of the industry and how people are actually using correlations to the theory. But that obviously changes over time, as well as within subsectors of the tech industry, so your utility may vary.

One last tidbit: that same hiring manager there's a quote from earlier on the page? I made him a promise, that we would talk about AI in this class. A lot. Not AI like 'oh, AI this, AI that, I have an MBA, I'm so smug'. But what fundamental data structures and tools do you need to know in order to make an AI go brrrr, at the lowest level. There is a whole sector of the industry that believes that you won't get hired anywhere in the next 5-10 years without 'AI' plastered all over your portfolio. And while we, as educators, don't agree absolutely with that, we recognize that there are a lot of decision makers who believe that. So we want to be able to speak their language (did we mention that Computer Science is about communication? Because it is... but we won't have time to mention that in this book).

The point is: we will provide you with the most applicable information in theory and practice. But we need you to take the extra little leap of figuring out how it actually applies to your program and goals.

Because something, something, something, dark side...

PART I
PART A - THE WAY

What is the way, asked the grasshopper to the teacher? And the teacher replied something very pithy and totally in line with the allegory that made the reader go 'Oh, snap, that was awesome'.

We, in Computer Science, have solved a lot of problems. But, just like you're the sum of all of your experiences and just like understanding you, in part, requires understanding those experiences, computer science requires you to understand how we got to where we are through the problems we've solved.

Enter Data Structures and Algorithms. Once upon a time, every single one of the theory topics we're going to cover was a practical problem. People needed to find ways to write a program that could remember multiple pieces of data as related, so they created a data structure to handle it. And it was slow, and it was terrible, and they needed to optimize it or prove that it couldn't be optimized any further and they needed a different approach. But, as happens, over time those solutions became baked into the various systems. For example, suppose you were writing a new programming language: do you a) require everyone to start from scratch or b) build directly into the core of your language some of the 'correct' ways of handling data that have been mathematically proven to be more or less optimal?

Hopefully you're not a monster and you picked the latter. For day to day operation, that's fantastic. Just like you don't need to worry about saving your Word documents every 30 seconds because auto-save is now standard, so too is the use of almost everyone of these data structures abstracted away into the lowest bowels of most modern programming languages.

This becomes problematic, though, in a few specific cases. We'll discuss a few of them in specific chapters. But the important one that justifies this Part of this book is this: if the programming language already does certain things automatically, then creating data structures in the language is a highly contrived process, akin to putting a third pedal in an automatic transmission and pretending like you're manually shifting gears.

As a result, in order to study Data Structures, you need to learn C. Correction: you need to learn C/C++. C is the 'correct' answer here, but C and C++ are so intertwined as to be inseparable in this scope. Where the difference is important, we'll mention it. Tangent: you could technically learn ASM for any number of processors, or even formal pseudocode and treat this all as a series of mental proofs and math games. But if that last fills you with dread just thinking about it, you'll recognize why we settled on C/C++. It's an applicable, industry-standard language that all programmers are broadly expected to know, so when there's an applied reason is a good time to do so.

So, this part introduces you to C and C++ as programming languages. Two chapters is a paltry introduction, but you have access to any number of tutorials and textbooks online. Think of the C/C++ chapters as 'data structures and algorithms primers, through the medium of programming language'. Ideally, it will suit both needs.

From there, we'll introduce the idea of data structures and algorithms in order to prime your mental infrastructure to be ready to discuss them in more detail in their own Part.

So, without further ado, welcome to the wonderful world of Low-Level Programming!

CHAPTER 1 - THIS IS YOUR BRAIN ON C/C++

Before we begin discussing C/C++ let's disambiguate terms. C is the 'original' programming language. You are welcome to learn more about the history of C on your own, it's well documented as coming from Bell Labs as a successor language to the internal B language. I say original in scare quotes, because there were many before it, but C has proven to be the most useful and versatile language, evidenced by its wide adoption. 50 years after its release date, it still accounts for a daily activity for roughly 10% of programmers, as evidenced by the TIOBE index (a benchmark survey that is widely considered authoritative).

However, there is a significant difference between C++ and C. C++ is a successor language to C, and is built 'on top' of C. In essence, while it has changed to become more integral over the years, it was originally a library that made tasks in C easier. For this book, and the examples herein, we frequently don't want things easier. Another way to phrase that is that we want to demonstrate what it is that C++ is making easier. As a result, we will do our best to disambiguate what is pure C code, and what borrows from C++.

Again, the two are so intertwined as to be a single language, C/C++. But, since this course seeks to demonstrate core data structure and algorithmic concepts that are well-solved in both the field as well as most modern programming languages, the distinction is important in this specific (and not the general) case.

The biggest example you will find is that while C++ is an Object-Oriented Language, we have to ask a question: what are those objects built on? And the answer is on more primitive data structures (literally called struct) in C. There is no native support for classes or any other Object-Oriented concept you have likely been exposed to, the ability to use those OO concepts is based on the optimized structures we've developed over time.

Language Learning Model – Refresher

There is a basic programming language learning approach that follows the following format:

- Phase 0 – Downloading the Compiler, IDE, any appropriate libraries, and familiarizing yourself with where to go get help.
- Phase 1 – Reading a few simple programs, compiling a basic program like Hello World, learning how it reacts to operators, understanding the syntax for a statement and what constitutes the gross structure of a program.
- Phase 2 – Learning the various conditional structures that exist to make the program actually make decisions, and how to create simple and complex conditionals.

- Phase 3 – Learning how loops work: the various commands and how much emphasis is placed on recursive structures.
- Phase 4 – Functions, Subroutines, Classes, Structures, etc. This phase focuses primarily on the portions of the language that uses synecdoche to represent multiple concepts in a short reference.
- Phase 5 – This is the open-ended phase where you start learning about the various niche or idiosyncratic things that make the language a language and where you develop proficiency over time.

Because this is not a text about how to program in C/C++, we will not be covering the language in depth. You will be expected to be able to look up some concepts, phrases, or libraries on your own. This is a good habit, because books go out of date, but reputable internet sites are more standard. cppreference.com is likely your most authoritative source, but you should consider supplementing it with easier sources.

At a bare minimum, you should be able to:

- Phase 0 – Use a text-editor of your choice to edit .c, .cpp, and .h files, as well as use a command line to compile and run programs.
- Phase 1 – Recognize what the #, ;, and {} operators are for, copy and run any program from this text, interpret compilation errors (likely with internet assistance)
- Phase 2 – You can get by with just if and else, but there are a few other commands that may enter. We will discuss recursion in the text. Our goal is clear code, so we will not be presenting you with tricky examples.
- Phase 3 – A while loop is the most basic structure, but we may occasionally use for loops for clarity. We will discuss recursion in the text.
- Phase 4 – You will need to be able to utilize headers (.h files) which is probably going to cause you the most grief if you have not used them before. You will also need to learn how to define a function, what a return statement is, and what a struct is.

These first four phases will be the bulk of the practical explanations in Chapter 1, and the fifth phase will be Chapter 2. Note, this won't be everything about C, but it will be what you need to understand the utility of data structures and algorithms.

- Phase 5 – You will need to understand the memory heap, memory structure (with some of what's happening in the hardware), security hardening your code, how arrays work, pointers, and a little bit of how the operating system interacts with a user's program.

IDEs, Git, Workflow

IDE stands for Integrated Development Environment. We frequently use it as shorthand for 'the stuff you do

to make the program go BRRRR'. One of the hardest steps to learning a new language is figuring out which of the many options available is the best for you to use. As such, we will provide you with a couple of standard, and comparable options.

For this textbook, you can use whatever IDE you are most comfortable with. There are a variety of tools that will help you, but we suggest considering the following:

Windows

VS Code (https://code.visualstudio.com/) is a convenient all-in-one utility that you can use for multiple different languages utilizing plug-ins. You will need the C/C++ plug-in. It has its quirks, though, so you will likely want to review several tutorials on its use. The biggest reminder here is: you should create a new project for every new program you write and you need to set up the environment for each project. However, once you have associated VS Code with your compiler and can repeat the appropriate steps, some of the finicky parts become routine.

An alternative to an IDE is using a text editor like Notepad++ (https://notepad-plus-plus.org/) or Sublime (https://www.sublimetext.com/). Both of these have plug-ins that will provide context-aware highlighting for different languages as well as directly interface with your compiler.

You will need to follow a specific set of instructions to install the GNU C Compiler in Windows. The easiest method is to use MinGW (https://code.visualstudio.com/docs/cpp/config-mingw). In essence, you will be installing a miniature simulation of a Linux environment to allow you to download from standard Linux package managers. Be very specific about the commands that the app gives you, and be sure you download the appropriate version.

OSX

Xcode is the standard for OSX. You can force it to use C and C++. There is also a native compiler in OSX that Xcode should automagically tap into. However, you may find that some subtle variations occur between the output of clang and gcc. This is not a warning, just an acknowledgement.

VS Code is also available on OSX, please note the Windows instructions.

Similarly, while Notepad++ is not available on OSX, you can use Sublime. TextEdit is feasible, but not as fully featured.

If you would like to use GCC on a mac, the most common solution is a utility called Homebrew: https://brew.sh/. However, you will need to find specific instructions that are valid for your OS, hardware, and Homebrew versions.

Linux

GCC is likely already installed in Linux. If not, you can use some variation of the command apt-get install gcc, although you will likely want to insure you understand what you are doing. Please be sure to understand which package manager you have, how your package manager works, and what changes installing gcc will do (practically, minimal outside of just downloading the files and altering the path, but it's especially important to know in Linux).

Text editors are a tricky business in Linux. If you are using a GUI Windowing System (such as X-Windows), you have a plethora of options that come and go. Try a few that likely came shipped with your base install and decide which you like. Sublime is also available.

Nano and/or Pico are also available as simple, convenient command-line text editors on most Linux installations. We are purposefully ignoring Vi and Emacs, although you are welcome to try them and discover their flaws.

Discussing Text Editors in Linux is like that scene in the beginning of Jurassic Park where they feed the T-Rex, only you're the goat. We suggest not forming an opinion until you have the expertise to have one.

Git

Git is a utility for monitoring, tracking, and incorporating changes into your code. It is outside of the scope of this text to cover it. However, we would be amiss not to mention that it is a tool that you can and should use. It is standard in the industry, and any alternatives will almost certainly follow a very similar formula to git.

Workflow

Now we come to the payoff: What do you do with all of this stuff?!

There are three parts to your programming workflow: The text-based source code, the compilation process, and the compiled program. Broadly, you will:

1. Write the text of your code, and fix any errors that appear.
2. Run your compilation program to convert the text to a binary set of instructions for the processor and/or operating system.
3. Run your compiled program and see what happens!

In general, this will probably look like one of two ways:

1. You create a new document in your text editor of choice, such as Notepad++ (which is fairly industry-

standard, but not perfectly so) that is already configured to provide live syntax suggestions, fix any errors suggested, and save your file as, say 'ch1-example.c'.
2. You switch to your command prompt, navigate to your project directory, and run 'gcc ch1-example.c'.

    - If it successfully compiles your program, you likely have a file like 'a.out' or 'a.exe' It's called 'a', because it can't guarantee that the file system it outputs to allows 0-length file names, and 'a' is the smallest string it can reliably presume that every operating system will allow.
    - If it is not successful, you'll return to step 1 and address any compiler errors that appeared.

3. You'll run your program by navigating to the appropriate directory and typing 'a.out' or 'a.exe' in, probably, a different command prompt (to separate out your compile history from your program execution history).

    - If it executes flawlessly, you're done! You'll 'publish' your project, either by pushing the code in a pull request on git, submitting it to a class environment, or distributing the executable.
    - If it does not do what you want, you'll return to step 1.

This is our suggested solution. It gives you the most control and flexibility at the cost of a slight bit of inconvenience. In our opinion, the inconvenience is quickly subsumed by the routine of your facility with this workflow. The alternative is:

1. You open or create a new document in your fancy IDE. You write some code and fixt the errors. You click the 'play' button I the menu. You see the results in the console window.

    - If at any point there's a problem, you start over from the text editing portion.
    - If it works, you proceed to 'publish' your project.

Notice that by bringing everything into a single app and step, it is simpler. But, you've lost some flexibility and are constrained specifically by how the IDE directs your attention.

Neither solution is 'right' and the authors use both interchangeably depending on project, mood, or environment. But the point is, we need 3 programs to run your program: The text editor, the compiler, and the program you create. An IDE frequently just bundles those tasks together.

C Basics

You should have facility with programming. For example, if you were asked to write a program that takes a

user's input and print it backwards in any language of your choice, you should be able to, fairly quickly, write and execute said program. This includes basics like textual input and basic security hardening.

If you think you could, but it might take you a little bit to get everything right, that's OK. You don't need to be a veteran of 20 years programming experience. The point is just that we're going to skip those basic explanations in favor of the how to do it.

If you are completely stumped and have never used a programming language before, you're welcome to read the book, but you will likely want to have a basic programming tutorial handy.

Phase 0 – Running a Program

```
int main(int argc, char *argv[])
{
     printf("Hello, World!"); // Comment: Output line of Program
     return 0;
}
```

Run that program. Can you? Congratulations! You've got the workflow down! We'll actually cover what the different parts mean in that first line in the next chapter, because it gets crazy.

Side Note – Sometimes, it's easier to just write a quick and dirty program that doesn't need to interact with the operating system. If you see the program definition as void main(void), that's because it's just quicker to write, and we're short-handing the longer and slightly more obnoxious command line arguments version. Any time you see that, it should work effectively the same as your standard structure, we're just highlighting, in essence, that this is all example and theoretical code.

Phase 1 – Statements and Structure

There are going to be two parts to this section. The first is easy: how does your program go? The second is a little trickier: how and why do we use headers?

There are two main things to keep in mind. The first are the ; semi-colons. These represent the statement-delimiters. If you've used Java, you're familiar with them. If you've used Python, you're going to constantly forget to add them. C is a whitespace-agnostic language, which means that you could cram your entire code on a single line, if you wanted. Don't do that, but you could. Because it doesn't care if you've got tabs or carriage-returns or whatnot. It just wants to see a little separation in some places. Every instruction will end with a ;.

Side Note – Some people argue (incorrectly, in our view) that C/C++ isn't really a language anymore, it's a language archetype. That means that semi-colons are a very standard statement-delimeter in a wide variety of languages. If you see ;'s at the end of lines in code you don't recognize, it generally means end-of-statement.

The other thing to note is the {}. These are called braces. However, you may also see them referred to as

Curly Brackets, Curly Braces, or those squiggly things. Don't worry about drawing them correctly, everyone will know what you mean from context. In fact, be extravagant any time you have to do them. People love that.

The {} is what we call a block-delimiter. It represents the beginning and ending of some section of code, and we're suggesting that everything inside of them represents both a new scope as well as a single thing in some contexts. For example, notice that the {} are around all of the code in the main function? Defining a function is basically saying 'main is really just a shortcut for the following command' and then we want to give it multiple commands, so we encase it all in brackets so that C knows we really meant that whole kit and kaboodle. We'll learn this later, but it also means that everything inside of the {}'s gets their own memory stack.

{} is considered a type of bracket. (), [], <> are also brackets that have their own implications. There is a consistency there that you should try to figure out for yourself. C also has two kinds of comments: // comments the rest of the line, and /* */ are the delimiters for multi-line comments. (The frequence of comments in our example code will be low, specifically because the entire book is about explaining the code.)

You should read some sample code off of the internet and see what it does, and if you can identify statement and block breakdown.

Headers

The other thing you need to know about the structure of a C program is that it frequently uses headers. You should know what a function does, so even though you may not have the syntax, we have a potential problem:

```
void foo (void)
{
     bar();
}

void bar (void)
{
     foo();
}
```

You can probably figure out what this program does. It's just a complicated infinite loop, right? Foo calls bar, and bar calls foo. And, if your IDE is smart enough, it says 'hey, dude? You probably don't want to do that' either before or after you hit the play button. If you're doing it in the command line, you can either close the command window or, inside of the window, you can push the Ctrl key and the C key at the same time (Ctrl-C), which is the command interrupt for 'Cancel Execution'.

Go ahead, try it. Does what was just described happen. Nope. Fooled you! You should get an error, that

might say "error: implicit declaration of function 'bar' [-Wimplicit-function-declaration]" If you don't know what an error means, in some IDEs you can hover over it for more info. Otherwise, copy and paste it into an internet search engine and you'll typically get millions of hits. Occasionally, you'll only get a handful or even zero responses. In that case, you need to remove things like the 'bar' from the previous error so that we get the most generic example of the error we can. If you get very few responses to even that, you've probably done something very wrong, and it hasn't been unusual for me to just start from scratch in that situation (within reason).

Side Note: [-Wimplicit-function-declaration is there, in this case, to denote the technical break point for the error. So, you could go digging around in the compiler internals to see why that error is occurring, or you could discuss with colleagues so you all know the same exact issue. A lot of the error can change between gcc versions, but that part should only change rarely.

So what's the solution? We could try:

```
void bar (void)
{
      foo();
}

void foo (void)
{
      bar();
}
```

But then we just get an error saying that foo is undefined. Let's start with the structure of a program. As the C compiler is parsing your program to make sure it understands what's happening, it does stuff like see void and () and see if it knows what to do with it. In this case, the void bar (void) tells it 'create a new token that represents this foo function'. But then, when it sees bar inside of the function, it looks to see if it knows what bar is. Because it's all happening in order it doesn't know yet! Further, it doesn't know that it will know, because it is not processing the text in anything but the most basic in-order sequence.

So, the first initial problem we have is that we can obviously think of situations where we need to bounce back and forth like that. We can ask Alice to ask Bob if he knows what Alice asked Bob (sort of).

This is where headers come in. Headers are the declaration of the function. Remember what the parser was asking when it ran across the bar() before it had bar() declared? It's just asking 'have I seen this before?' So, if we could just give it the bare minimum it needs, we'd be golden. You've actually seen this pattern before:

```
void main (void)
{
      int x;
```

```
      x = 10;
}
```

Now, we could just write int x = 10;. But that's technically shorthand for the preceding. int x; provides the declaration for the variable. But we haven't done anything with it. We've just say 'you exist'. And it's not until we get to the assignation (=) that we put anything in there. So, what we need is a way to separate the assignation and the declaration of the functions.

Unfortunately, there's a thing that happens later (overloading) that causes us a little grief here. In short, while a variable's handle is just its name, for a function, we need to know its name, arguments and return type. So, the entire handle for the foo() function above is actually void foo (void). However, this means that we can fix the code so it works:

```
void foo (void);
void bar (void);

void foo (void)
{
      bar();
}

void bar (void)
{
      foo();
}
```

It's still an infinite loop, but it was also important to show you how to Ctrl-C, since we frequently accidentally write infinite loops, and it's nicer to break out of it than to just close the session.

Notice that the first two lines are basically just stating 'this is a thing that exists'. But then later on we're saying 'oh yah, but I actually want it to mean something.' This is the proper use of headers.

Eventually, as we define different data structures, standard C convention will want us to access functions from different files. For example, suppose this is all part of foobar.c. And later, we'll have prog.c that says:

```
void main(void)
{
      foo();
}
```

Well, since main() doesn't even have a clue what foo() is, we'll need to find a way to direct the compiler to foo(). And the way that we do that is with the #include statement (you've used #include before in java or import in

python or using in C# or similar commands in other languages). #, in C, indicates a compiler directive. It's not part of the code itself, it's a sort of syntactical grease that lets the compiler do its thing. And in this instance, it's being instructed to delete the #include line (inside of its parsed version) and replace it with the contents of whatever file we gave it.

So, easy! We just change our prog.c to say:

```
#include "foobar.c"

void main(void)
{
      foo();
}
```

And, voila! Essentially, it dumps all of foobar.c into the top of prog.c, and main() now knows what both foo() and bar() are.

There's a problem, though. It's more than we want to get into right here, but we don't actually want to #include any .c files. It's broadly contrary to standard C-programming convention. In short, it causes the end-stage of the linker process to not be able to smoosh all the files together cleanly. So, we introduce the .h file that only consists of headers and a few housekeeping issues. foobar.h:

```
void foo (void);
void bar (void);
```

And that leads to our final foobar.c:

```
#include "foobar.h"

void foo (void)
{
      bar();
}

void bar (void)
{
      foo();
}
```

And our final prog.c:

```
#include "foobar.h"

void main(void)
{
      foo();
}
```

There's a whole bunch of other stuff that typically goes in there, too. Like #ifndef and #define and #endif. But you can look up when those are useful on your own. It's too much clutter for this text, and frequently not necessary within this context. Be sure you know how it works before putting together professional code, though.

All of this gets us to: use headers, your IDE should automagically link things together. On the command line, you can just be sure to include all of your .c files as appropriate: gcc prog.c foobar.c. There are a few cases where it would be reasonable to just dump all of the structs and such into the .h file, and in that case you might have a .h file with no corresponding .c file. We'll note when we do that, otherwise you can experiment yourself to see how far you can take that idea before things fall apart.

Side Note: You'll notice that sometimes we use #include "foobar.h" and other times #include <stdio.h>. Why? "" indicates that the compiler will look for the file relative to the current source file (so, the same directory you're in), while <> indicates that it should look in its standard library paths that were set up when you installed gcc. The latter implies standard libraries, and the former implies custom tools.

C/C++ Note: Somewhere along the line, C++ stopped requiring the .h in their includes. So you'll see #include <iostream>, but the equivalent C construct of #include <stdio.h> is different. Essentially, C++ is using namespaces and abstracting the concepts from the filesystem, but C doesn't do that (and why should it, if you wanted that abstraction, you'd be using C++).

Tl;dr: We're going to use header files in many situations. If we omit the .h file in our code for brevity, you'll want to know how to create them on your own. They're pretty simple, ultimately.

Phase 2 – Conditionals

There are two main types of conditionals in C: if-then-else and switch statements. If-then-else is the standard pattern that you're used to. Example:

```
if (something_is_true)
{
                  // then do this!
}
else
```

```
{
                // If not, do this other thing!
}
```

It's fairly standard across most languages. There's no specific elseif or elif command, so you're literally just chaining if statements inside of else blocks. Sometimes that's unavoidable, but there's an alternative:

```
switch (some_variable)
{
     case 1: // do something
           break;
     case 2: // do something weird
           break;
     case 3: // do something really weird
           break;
     default: // do something booooooring
           break;
}
```

Notice the breaks. Switches are convenient, but they're a little dangerous. They're making use of code labels and more intricate stuff that is typically only considered appropriate in this one case. But, the structure of the switch statement says 'jump to the equivalent case, and then continue operation until the end of the block'. We need to use the break command to tell it to jump down to the end of the block.

There's also something you may occasionally see: the only ternary operator (?:). We can do the same as the first section with something_is_true? //do something: //do something else. We don't recommend ternaries, because they're frequently much harder to read than traditional if statements. This is not a hard rule, just a recommendation.

There are a ton of super clever, very fancy variations on this that use crazy C tricks to imply conditionals rather than explicitly stating them. Don't go chasing conditionals! Just stick with the ifs and switches that you're used to.

New Operators and Data Representations

There are a few operators you need to know, as well:

- && the and operator for conditions

Ex: thesun(yellow) && thesky(blue) == true

- || the or operator for conditions.

Ex: thetoes(purples) || thesnowfall(orange) == false

- ! means not. Ex: !true == false

- == is the equivalency operator. Remember this is different from = the assignation operator

This is important, because in C, if (x = 10) will always execute the then clause and it will also change the value of x.

- !=, <, >, <=, >= all as expected. There is no <> operator.
- & is a potentially new-to-you operator that we use occasionally. It means 'take the binary representation of two variables, run && on each of the bit positions in common between them, and then convert the result back to the original type'.

Ex: 57 & 23 == 17 // The same as 00111001 & 00010111 == 00010001 notice that the only 1's in the answer are the positions where there's a 1 in both inputs.

- | is a potentially new-to-you operator. It's the same as &, but for bitwise-or.

Ex: 57 | 23 == 63 // The same as 00111001 | 00010111 == 00111111 notice that if either of the positions have a 1, the output has one. Generally & will be smaller than either operator and | will be bigger than either operator.

- ~ is a potentially new-to-you operator. It's bitwise-not, so it inverts each bit independently and smooshes the value back together.

Ex: ~57 == 198 // the same as ~00111001 == ~11000110, and it's tautologically the case that: 57 & ~57 == 0.

- As long as we're talking notation, hexadecimal and binary math is outside of the scope of this text. You should have learned it by now. If not, don't panic. We won't use it too much, and we'll explain our math when we do. But it's an important thing for you to know. But we will represents

-0x is the prefix for representing a number in hexadecimal. All bytes can be represented with 2 hex-values, which means that all ints will be an even number of hex values. 0x39 == 57.
– 0b is the prefix for representing a number in binary. So 0b00111001 == 57.
– 0<non-zero numbers> is the format for octal numbers, but nobody uses that.

– Any number that starts with non-zero is defined decimal.

Side Note: 0b is non-standard. But basically all modern versions allow it. So feel free to stick with it.

Data Types

If you're familiar with many languages, you are familiar with C-derived data types. But some languages, such as Python, don't always strictly type their variables. That will be a bit of a strict learning curve. But it's important, and the foundation for data structures.

Incidentally, we call these data types 'primitives':

- Integers – generally, you can use int until you find a reason it's limiting for you. These are counting numbers. They should be your default storage value, unless you have a specific reason to use something else. They will store numbers that do not have a decimal point in them and will not (sort of) store textual data.
- Characters – the char datatype will store a single byte of data. This is old-style ASCII data, we aren't going to worry about Unicode or newer character data. All of the same tools will work, they just require different data types (like uint16_t or uint32_t). These are literally just integers, but with the intention of using them to store a single character of textual data.
- Floating point – There are two data types that you will need: float and double. They're both floating point values (any number that has a decimal point). They are, generally slower, because they are bigger and the circuitry for bitwise floating-point math is more complex. Broadly, use a float when you need it, but default to int when you don't. A rule of thumb is: if you will divide or use decimals, use a float. In this text, we will use float for most as a reminder that it's a floating point, but that could almost always be easily substituted for double which is also very standard.
- Boolean – bool values are stored as a whole byte, so you don't gain anything by using them, from a technical perspective. But we're still including them because the readability is drastically increased by noting when a variable is intended to be a logic value

These four are the basic data types we'll be using to build everything else. Ultimately, everything in the computer is just a series of 1's and 0's with the semantic meaning externally applied. But, these are so intrinsic that it's needlessly complex to reduce lower. Practically, we could even simplify to just int and float, but the readability takes a nosedive, so a little convenience is warranted.

There are a few extra data types to discuss:

- enum – an enumerated value is an integer where the values of the integer are given names rather than just numbers. The intent is to increase readability, but it has no low-level impact on the logic or algorithm. They may appear occasionally when readability warrants, but are mostly superficial.

- Pointers and Arrays – This should, almost definitely, appear in the previous list of four. However, because we're going to cover them in extreme detail later, they'll get their own section. In short, a pointer is a data storage type for a location in memory. These are going to be your best friend and worst enemy. You should never use them in practice, but they're vital to understanding how a library actually creates things.
- Strings – Strings are pretty common in most languages. C++ has it as a basic data type. And there is a library in C that uses strings. But, we need to know how to build a string, so they'll only be used from the standard string.h library when they're semantically unrelated to the concept of the data structure or algorithm.
- void – void is technically a data type. It is the data type for any data that is of length 0. That means there is literally only one value: no value at all. We call that value null, and it will appear frequently. But it both sort of is and isn't a data type. It's more a placeholder for a concept.
- struct – By definition, a struct is a complex data type. When we want to be able to do more than store a single number, we'll be creating a struct to handle it. This is 'struct' as in 'that's the name of the textbook!'

In short, when you see a term, keyword, or code phrase you don't recognize, you need to do two things: recognize where it is in context, and then look it up and make sure you understand it. This textbook is useful, but a very narrow slice of the C language. Further, every other value is either a combination or an extension of one of the four basic data types we listed.

Phase 3 – Loops

Remember that a loop is a repeating code flow tool. You designate a block of code to run until a certain condition is met. All loops need to have four components:

1. Initial Setup – This can happen prior to the loop, but we typically do not want to run things like defining variables in every cycle of the loop. Alternately, you might look at this as 'the data that will persist beyond the loop'.
2. Condition – This is the Boolean calculation that tells us when to stop running the loop. It might be as simple as 'true' (run forever) or a long series of compound Boolean operators. In recursive loops, we'll call this portion the 'base case'.
3. Iteration – This is the code that executes in every cycle of the loop. A loop with no iteration section is kind of useless. There are some quick hacks that try to combine the iteration and the condition section, but again: resist the urge for this.
4. Iterator – You need to change something in your setup every cycle in order for the condition to see a new situation. If nothing changes, you have an infinite loop.

We will use both while and for loops in this text. We can get away with just while loops, but there are situations where a for loop is conceptually clearer. As a reminder a while loop, generically, is:

```
int some_iterator_value // 1
while (condition == true) // 2
{
     do_some_thing(); // 3
     some_iterator_value = update(some_iterator_value); // 4
}
```

Similarly a for loop, generically, is:

```
for (int i = 0; i < some_gatekeeper; i++) // 1; 2; 4
{
     do_some_thing(); // 3
}
```

As a look ahead, when we get to recursion, you will see that the same parts are there, just in a different format:

```
int rec_func (int arg) //1
{
     if (base_case(arg))
     {
          return some_base_value; // 2
     }
     do_some_thing(arg) // 3
     return Rec_func(iterated(arg)); // 4
}
```

Broadly, we will use while loops in the general case. If it's a counting iterator, for loops are clearer. And we will avoid recursion except for the cases where it is intrinsic to the proof.

Side Note: You need to know how recursion works. It is crucial for proofs, algorithm design, some basic algorithms, and reading others' code. But, it should generally be avoided. Your rule is: WWND? What would NASA do? And the answer is: never use recursion in production.

Continue

One of the keyword pairs we've started, because it applies to conditionals, is the break/continue pair. In a loop, you will see break used when we want to leave the loop for any reason. You can think of it as moving the

execution fo the program to immediately after the next } for the innermost loop, thus leaving the loop. You can think of continue as moving the execution of the program to just before the next } for the innermost loop, thus immediately skipping to the next iteration.

Phase 4 – Functions

Because we will be demonstrating code using C, we will have functions not methods. The only distinction is that a method is a function that's attached to a class. This may be a detriment to the clarity of the code in some circumstances, so exceptions may be made and noted when a clear class may be useful. Remember, part of the use of C in this text is a contrivance, and part is because it's the clearest way to express why data structures are important. But, when using C doesn't further either of those goals, we may substitute other languages.

This is a good time for a reminder: the code in this text is not always good code. We will have examples, but frequently the code will only be demonstrating a form. In those cases, we'll use variables like 'x' or function names like 'foo' and 'bar'. We are specifically doing so to demonstrate that they literally have no function, just form. Outside of that context, we will attempt to write self-descriptive function and variable names that are immediately apparent as to their utility. This is an invitation for criticism from you, dear reader.

Also, remember that Phase 5 will discuss quirks of C in the next chapter.

What is an Algorithm? For Grown-ups!

When you were 3 and you were told to put away your toys, you didn't do a good job. Apologies if you felt that you did. But, if you watch any 3-year old told to put away their toys, they'll go 'Ok!' and the toys will stay on the floor, because a 3-year old cannot conceptualize that the toys don't go wherever they were last.

You can step-by-step instruct a 3-year old in how to put away toys. 'Put the ball in the box, put the bear on the bed, put the controllers next to the console'. But this isn't really an algorithm. This is, however, how you explain what an algorithm is to children: a series of steps to accomplish a goal.

You are not a small child, and we need a better explanation. So, let's start with the three ways that we use the word Algorithm in Computer Science. These are really all the same way, but we're going to separate it here because they feel different:

1. The solution to the specific problem that you are solving. If you are making a computer game, and the enemy AI has to figure out how to attack the player, and you create a function that tells it how to figure out its next move, you have created an algorithm. But this is a very narrow and specific algorithm. However, it is solving a problem.
2. The solution to a specific class of problems that may need to be solved. This is typically what we're looking for in this class. You don't need to know how to solve a specific matrix multiplication problem

of two 300×400 matrices. You should solve the general problem of how to multiply any two nxm matrices, and then your specific problem will fall into that category.

3. The mathematical proof that a solution exists in a certain time and space. This is the very mathy version of what an algorithm truly is, in the purest and most erudite sense of the term. For example, if you read The Art of Computer Programming by Donald Knuth, he will typically present the mathematical precision with which certain classes of low-level problems can be solved. And TAOCP could be considered the canonical tome of Computer Science.

We run into a couple of problems. Mathematical proofs are great! But most people don't need a mathematical proof that something will work. They need a practical proof that what they want to do is correct.

Similarly, solving a class of problems is fine... but sometimes we run into special cases where a customized solution really is the best option. Djikstra's A* algorithm doesn't help you figure out how to get the Zombie to attack Steve when there's jumping and time delay involved. Not to mention banging on doors. So much banging on doors.

So, in practice, you'll find that there is a prevalence of 'I need to solve this problem right here and now'.

Which of the three is correct? All of them. You need to be aware of the solution to any given problem in the context of all three. If there is a fixed and mathematically perfect solution to a problem, it would be silly to ignore that (or more likely, a mathematical proof that you can't do what you want to do, and you'll need to frame your problem domain differently). Similarly, being aware of the difference between a library that solves all problems and a specific solution that fits your goals is the mark of an experienced programmer. The Internet is filled with thousands of stories of game designers writing programs for cramped early cartridge consoles and solving only the specific problem that they needed solving, because there wasn't space for anything bigger.

In this text, we will generally be focusing on the middle case, because the math is boring and unimportant if you aren't planning to be a researcher. The specific case has too many possible (literally infinite?) cases such that it's silly to try to approach. But we want to be sure that you recognize where this fits in the practical aspect of programming.

Incidentally, this model also fits the broad structure of each chapter: these theory sections are the second description, the practical sections are the first description, and the AI sections are broadly correlated to the third description.

Algorithm Definition

So, with all that out of the way, let's define an algorithm. You can find a dictionary definition on your own. Wikipedia has, shockingly, one of the most referenced phrasings. Instead, our definition is going to focus on the characteristics of an algorithm:

1. An algorithm is finite. One of the most effective phrases in the toolbox of a programmer is 'longer than

the heat death of the universe'. We can solve any problem, but if it can't be done before every star goes cold? Maybe we should rethink our approach. Essentially, an algorithm is only 'correct' if it finds the answer in a reasonable amount of time.

- There is a more technical definition, revolving around the number of steps for any given problem in a class converging on some calculated projection. But, we're better of just limiting it to: an algorithm always solves a problem in a finite number of steps. That means, you could count them, if you were so inclined.

2. An algorithm is definite. You can't have wishy washy statements. For example, if you and a friend were going out to eat and they ask you were you wanted to go and you said 'I want to go to a specific place, but I don't know what it is'. Well, that's the opposite of helpful. Even the statement 'somewhere that serves food' is a definite statement, because it means that the correct answer is literally anywhere from a dog food factory to Bob's Burgers for Exotic Eaters.

   - We can narrow this down specifically to: each step in an algorithm is atomic and cannot be broken down into smaller steps, reasonably. For example, while we could break down addition into adding a series of 1's, that's unreasonable for our goals. This means that our idea of 'definite' is a little bit scope-dependent.

3. An algorithm has input. The number of inputs can be zero. But, the fact that there are zero inputs is, itself, the input. The important part here is not that an algorithm has to have inputs, but that in order for an algorithm to be correctly defined, the inputs need to be explicitly delineated. Functions do this automatically with the arguments section of the function declaration. In the same way, your algorithms should do so.

   - Where possible, we will attempt to avoid generating inputs. For example, if we know that we are going to accept input from a user, an algorithm should generally have 'input from the user' as the input, rather than having one of the steps be 'get input from the user'. The process of getting input from the user is messy, context-dependent, and unrelated to the algorithm, more often than not.

4. An algorithm has output. There is at least one output to the algorithm. Notice that zero outputs is not an algorithm. We can make a function with zero outputs, but we don't use the same word for the two for a reason. An algorithm needs to produce a conclusion. This can be a million different values or it can be a single true/false value. Either way, the algorithm needs to produce a result. And, like input, the important part is that what that output is? Our algorithm isn't complete until that's explicitly defined.

5. An algorithm is correct, effective, or precise. The word that we use there isn't really important. What's important is that we have a goal for what the algorithm does. We might define that goal mathematically, or we might define it within the context of our specific, or we might define it narratively as a qualitative description of what we generally want done. And then the algorithm actually needs to do that thing to be defined correctly. Further, it needs to not miss any bits (edge/corner cases) or do anything extra (side effects).

    - Again, the point is that it's not so much that the algorithm needs to do something. It's that part of the algorithm is the actual documentation of what it's supposed to do. If we have input -> output, then actually defining what the '->' is, is important.

Fun fact! This five-part definition of an algorithm is not an algorithm. It's a model, or a framework, or a definition, depending on how you want to frame it. You know what is an algorithm?

```
Algorithmic Definition Checker
Input: a potential algorithm (PA)
Output: a Boolean value of whether the input is an algorithm
Method: It checks the algorithm against each of the definitions above, and re
Start
     Is PA completable in a finite number of steps?
          No – return false
          Yes - continue
     Is each step of PA definite?
          No – return false
          Yes – continue
     Is the input defined?
          No – return false
          Yes - continue
     Is the output defined, and is there at least one?
          No – return false
          Yes - continue
     Does the algorithm do what it's stated intention is and nothing else?
          No – return false
          Yes – return true
End
```

You're welcome to run this algorithm through itself to see if it defines itself as an algorithm. But, notice that the definition is not an algorithm, because it does not have an input, and output, concrete steps, any steps at

all (so not finite), or does it have a stated goal. It just is. I added the Start and End as a basic courtesy, because doing so makes it easier to convert to a flowchart. But that's a different book.

Here's a quick math style algorithm that we won't be covering in this class (simplified from Heap's Algorithm):

```
Permutation List
Input: an integer n, a list of n distinct elements A[]
Output: A displayed sequence of all possible permutations of A[]
Start
     Is n equal to 1?
          Print A
          Return
     For each i in 1 to n
          Recurse on A, but pass n - 1 instead of n.
          If n is even
               Swap a[i] and A[n]
          Else
               Swap A[1] and A[n]
End
```

This is a super common algorithm. Let's check it through our algorithm checker:

1. Is it finite? Yes, because no matter how big n is, we'll only recurse n times, and we can't get past the barrier of when n = 1 since n is an integer.
2. Is it definite? Outside of 'print' (which is actually atomic if we aren't talking computers and we mean to actually write it), most of the steps are pretty basic. We could break Swap down into c = b, b = a, a = c. But, since swap is clearer and always means the same thing, it's 'definite'.
3. Is there input? There are two inputs, an integer and an array.
4. Is there output? There is one output, the printed copy.
5. Is it effective? Yes, if we presume that the array is a copy, there are no side effects, and it will print all of the copies.

Let's talk about the algorithm for a moment, though. Is it clear to you? I mean, you accept the assertion that it works. But do you understand why or how it works? Do you immediately grasp it? This is why the mathematical definition isn't always the best. Sometimes, it's clearer to see it illustrated in code or clear steps.

Notice that there are two swaps for every iteration of the loop, and in fact it's a nested loop, since it recurses inside of the for loop. This is difficult to track. But, any solution that will have n-factorial outputs is difficult to track. Try writing it down and following it for, say, the array [1, 2, 3]. Then move up to [1, 2, 3, 4].

There is argument to be made that, in practice, recursion is silly and to be avoided. That's probably too sweeping a statement to be followed. But using iterative over recursive solutions when possible would have made this algorithm potentially easier to follow. Heap's algorithm is a common and widely available algorithm, feel free to find an iterative version and compare. Which on do you find easier?

The point is, generating new mathematical algorithms like this is not the goal of this textbook. Being able to solve specific problems, adapt existing algorithms, and read algorithms is the goal.

PS

Heap's Algorithm there? Don't sweat it. That's specifically there because you need to see something. And it may as well be something crazy. If you understood it, great. If you didn't, that's OK too! You should just understand that there's a form and a structure. At this point, you're just getting the general shape of algorithms, we aren't diving in.

Practical Applications

Forget everything in there about theoretical, mathematical, or abstract algorithms. In the work place, you need to solve the problem, and nothing else matters. "Solving the problem" means, filtering input, failing gracefully, producing the right output, not messing with anything outside of your scope (i.e., don't T-Pose some NPC as a side effect of processing its AI options for pose), and basic security hardening. You need to do basic, practical, good programming tasks. And outside of that, as long as you handle all of the edge cases and get the right answer, you're fine.

It's possible that the next go around will want to optimize the function a little more. But, realistically, you don't need to worry about that to start. Most of the time, optimizing is the role of a more senior developer (one exception: if you're the sole dev. But if you're the sole dev, you're the one who makes the decision as to whether it's good enough anyhow). So, focus on the practical definition of an algorithm, and don't worry about the mathiness. Yet.

With regard to workflow, you typically don't have a lot of options. If you're in a big programming house, use whatever they've installed on your machine. It's not worth the fuss, and you want others to be able to come and read your code in the same environment that they're using. You're adaptable, the computer is not. If you want to practice this, get in the habit of learning lots of tools: use VS Code, use vi, use notepad and command-line compilation. But, don't set up your workflow and declare it's the best workflow. There isn't a best workflow, and being part of the team environment is more important than nitpicking exactly how you want to be able to click.

Suppose you are in charge of making those decisions? VS Code is standard and so is Notepad++. If you're

programming in OSX, C/C++ is not standard. If you're in a Linux environment, get used to editing and compiling in the command line. It's not the most efficient option, but it makes you more versatile.

Gcc is not just a compiler, it's the compiler. There are lots of specifics that individual large companies might use or, for example, OSX is an oddball with their own version. But gcc is, by a large margin, the most standard tool.

Suppose you want to or are forced to use a different tool, like clang or llvm or msvc or whoever. It's really not a substantive issue for the most part. There will be some slight quirks that change, and the error messages will likely be different. But the general process will be exactly the same. That's the nice part about the tripartite model here: editor, compiler, program. You can swap each of them out and the others should be able to continue right along.

AI Examples

We're going to use AI as our example for the majority of our examples here. Why? Recently, the authors had a meeting with a high level team lead at some random company. Which company doesn't matter, they were just experienced in the field. There were three interesting outcomes to this meeting (paraphrased):

1. I don't need programmers who want a defined problem, if I had a problem definition then I'd just solve it myself. I'm delegating a general issue to them because I want them to figure out the specifics of the problem and solve it however they want.
2. I don't expect them to come out of school with the tools they need to work for me. I expect them to come out of school with a set of tools that lets them know what else is out there so that they can research and continue to learn. I need them to know the basics, and I expect them to learn the specifics that apply to my project.
3. Tell them to learn AI stuff. Right now, everything is AI. Every executive is demanding every team lead do AI stuff. It doesn't matter if it isn't really AI, it's just called AI. Have them spatter AI all across their resume, because that's the thing that's hiring right now. Every other person I know at similar companies is dealing with the same thing. Oh, you'd actually teach them the low level data structures that feed into an AI model? I actually need someone to do that. Yes, that's exactly what I want.

That last point is the most important. It's not that AI is going to last forever. It's that you should have something that you can say 'I learned about some specialty'. Everyone has to learn data structures. You should also learn AI data structures. It will set you apart, and there will always be someone who needs these data structures. Following the other tips, you can also parlay niche AI concepts into whatever specific problem you're facing. That's the gold standard for a good entry-level programmer.

Remember: having a specialty that you're interested in will make others interested in it and then you, by proxy. If you don't have something you're interested in yet, here you go: be interested in AI.

What is AI?

We have a problem. This is a societal problem, but it's one, nonetheless. And it's what do we mean by AI? Suppose we asked what the word 'literally' means. It literally means figuratively now. It doesn't matter how much linguists talk about the actual denotation of the word, when the rest of society is using it to mean figuratively, that mob-rule becomes valid. Whether that's a valid approach or not you are welcome to debate, but it's probably pretty obvious that it happens. At some point, the experts lose control of certain aspects of their field.

AI is kind of like that. We all like the word 'AI' because, very likely some combination of, it makes us feel like we're living in the future, it's an easy bogey-man, and it's too complicated for a lay subject. So, let's start categorizing what people mean when they say 'AI'. After all, if you're going to be plastering it all over your resume, you should probably understand the concept you're using.

Side Note: the principle here is not AI-specific. Right now it's AI. But in five years, maybe it's magnetic monopoles, or algorithmic subtraction, or Kentuckian Derby Horse Breeding or any other weird buzzword. Your job is to be able to use the words in context with both tech experts and novices.

- Statistical– Sometimes we call this 'big data'. But, it turns out that there was a point in the late 90's when we hit critical mass and had the ability to run huge statistical models on data. The key here is creating predictive models. This falls more and more under the classification of 'AI', even though it's just math.
- GPTs and other randomizers – These are still statistical models, but they rely on 'AI' tools like Neural Nets or Markov Chains to create their predictions along with some sort of randomization factor. These are still statistics, but they're such complex statistics that they seem to have an emergent behavior that humans easily anthropomorphize.
- NLP and other discretizers – The problem with statistical models is that to predict what's happening, they rely on chance. From a legal and actuarial perspective, there is no provenance. So, while NLP has not had the qualitative leap that LLMs have had through GPTs, they are still a viable approach in many aspects that require provability.
- Computer Vision and other responsive tools – This is a specialized branch of AI that frequently just devolves into a more and more classified system of if-then rules. Most game AI, for example, is really just complexity built form simple rules. But, we frequently call it 'AI'.
- AGI – Artificial General Intelligence is what all of these previous models are simulating, but poorly. We don't know what AGI will look like or how it will come about (we aren't even sure we'll recognize it, formally). But, it's ultimately what most companies are trying to imply they have when they smatter AI on things.

So, in essence, from a hybrid perspective between business use case and practical programmer, if we were going to generalize what AI was, here's one model we can use:

1. We have some data, and we formulate the input in a particular matter for our uses.
2. ??? Secret SAUCE!!!! ???
3. We have the answer!

That second step there is kind of what makes AI a little slippery to nail down from a holistic perspective. A lot of people will say something is AI, but they can't explain why. For the most part, when business needs demand AI, what they're really saying is 'we want answers, but we don't care if there is a cognitive link with the question'. So, in that blank, we might have some statistical models, we might have a neural network, we might have a series of rules. But the point is that, for the purposes of this book, we're going to treat situations where there is not a clear, lay-person understandable link between problem domain and solution domain, AI.

This is a risky approach, and the primary reason why it's reasonable in this text is that we don't actually care about that secret sauce step. We only care to create an applied scenario where we can demonstrate the low level data structures or algorithm that makes at least one of these advanced models work. So, we'll strive to set up the situation, what the goal is, and how the low-level data structures will apply.

CHAPTER 2 - THE HEART OF THE ALU

Let's take a quick side tangent and discuss how a processor works. The fundamental structure of all computers is based on the Von Neumann Architecture (John von Neumann is one of the founding fathers of modern Computer Science. He's worth reading up about). The Von Neumann architecture is a conceptual model regarding the structure of all modern processors. In it, we have four parts:

- Input – Things that exist in the problem domain that we want to calculate over. Input feeds into the ALU.
- Output – Things that exist in the solution domain, that have been calculated. The ALU feeds into Output.
- Storage – Stuff that's not input or output, it exists to provide context into the process of converting input to output. For example, the instruction set used for a calculation is part of the Storage aspect. The ALU sends and receives data from Storage.
- ALU – This is frequently colloquially listed as the CPU or the Processor, because when we expand the scope, it makes more sense. But within the processor itself, the ALU is the heart of the architecture. This is what actually produces the change from input to output.

Notice how closely this aligns with our definition of an algorithm? Notice how, if we were going to expand our scope even higher to business processes, we could use this same architecture to define any process? This is called vertical alignment, and it's good to keep it in mind as we evaluate things. Here's the important part:

All of Computer Science will always call back to the Von Neumann Architecture in structure. Whether it's hardware, software, algorithms, functions, programs, or any aspect, the Von Neumann Architecture is the base you should be aligning with.

This chapter is about C, so why are we talking Von Neumann Architecture. You remember that scene in the Aladdin cartoon where Robin Williams' Genie expands huge and goes 'phenomenal cosmic power!' and then shrinks '... in an itty bitty living space...' That's a fantastic metaphor for C programming. We, as programmers, are dealing with the infinitude of the fundamental logical building blocks of the universe. And then we are trying to cram it through a single transistor at a time. C is the closest aspiration we get to being able to do anything, but still being limited to the hardware we're acting on.

Phase 5 – Give Him the Bits!

One of the most important parts to remember in any field is that learning and progress is always a part of the continuum from past to future. And, unless something foundational changes, no matter how high level your programming gets, it's always built on the low-level foundations. The C level is that low-level foundation. Even if we abstract away all of the full access that low-level algorithms expect, some part of the program will always be manipulating electrons, memory addresses, and transistors. You should not expect that is going to be the case for routine activities. But you should know how it works. Just like you don't expect that you're going to go in and manually manipulate the pistons 1000 times per minute in order to get your car's engine to go fast, you should still have a fundamental understanding of how the car is making that happen so that you understand why things go wrong.

Side Note: This is all dangerous knowledge. Not dangerous in an individual sense, but from a macro scale. In Feb, 2024, President Biden made an announcement that he wanted to see technology move away from so-called insecure programming languages in order to prevent security holes from existing. This is a fine sentiment from an aggregate, public healthy perspective. However, remember, that someone still has to program and design the security-aware languages, just like no matter how advanced AI gets someone will still need to program the AI itself. Like everything else, knowing how to do these tasks and doing them on a daily basis are different concepts. You, as a computer scientist, are expected to know how these data structures work even if you abstract them away on a day to day basis in favor of more secure forms of routine programming.

The issue that we're going to be addressing is memory management. It's going to take some setup before we get to the payoff. But, essentially, remember that every part of the computer is accessible to the programs that are running on the processor. In the modern era, the Operating System controls which programs get access to which resources. But, again, the Operating System needs to be programmed to access at these layers, and that's aside from other applications like embedded programming or robotics or BIOS programming, etc.

How Memory Works

Everything in a computer is a stream of 1-dimensional data. In many respects, we don't consider what's going to come next or what happened before (we do in some architectures, later, but we're ignoring that part for now). So, let's set the stage and decide how we view memory. There are two primary ways we can do that:

The first model is to consider it as a 'heap'. We will discuss 'heap' as a data structure later. But that is an allegory that is extending this allegory and they're not exactly the same. In this case, we're mostly just doing some hand-waving and saying 'Memory? Yah, there's some there. It's a big pile of memory that does stuff. I want to reserve some memory, I ask the programming language, and it gives me some. No biggie.' This is a valid approach, because most of the time, we don't care about the specifics.

The other model is to consider it as a 'stack'. Again, this is not exactly (but close) the same thing as the data

structure of the same name. In the stack model, every single bit of memory exists one after the other. And we need to know what the 'address' is of any given part of memory. 'Address' in this case is another metaphor, akin to delivering mail and needing to know which address to go to on a street. You know the general layout and that 2002 Main St is going to be further along than 1040 Main St. But, if you're the mail delivery man, you don't really care what's in the package or who lives there just that you follow the directions to deliver the right package to the right location.

So, we have a memory heap, which is abstracted hand-waving to acknowledge that we're doing memory stuff without worrying too much about the how. And we have the memory stack when we want to worry about the how.

What does this mean for us, as programmers? Let's take the stack analogy of houses on a street. When we write our program, we need to be able to tell the operating system 'we want X amount of land, and it's going to take up the space for addresses Y through Z'. (We're jumping around a bit... if we had low-level enough access, we could just take it, but that will come in the security chapter.) But, now that we have the space, what do we do with it? Well, we say 'Address Y + 0? You're going to be an integer. And Address Y + 2? You're going to be a character. Addresses Y +3 through Y + 23? You're going to be an array of five floating point numbers.'

So, what we find is that we can access any part of the data for our program by saying 'where did we put that integer? It's at Y + 0! Let's store a 10 there!' We've seen that code. It looks like this:

```
void main (void)
{
      int x;
      x = 10;
}
```

This brings up an interesting side note. Once upon a time, in the dark ages of programming, we literally had to remember and access all of the memory addresses explicitly. But part of what a programming language does for us is to remember those things for us. So, we don't have to remember that the x is at Y + 0, we make the programming language remember where it stored x. As an aside: this is good programming, we're abstracting away functionality from form. We let the computer do what it does well: remember stuff, and we do what we do well: give stuff semantic meaning.

So, we've declared the intention to start a new program, we've gotten a section of memory that we can do whatever we want with, and we've started tasking the programming language with remembering what goes where. Great! Where's the problem?

Arrays

Every language has some variation of an Array. Maybe they call it a Vector or an ArrayList or a List or a Slice.

But, at their heart, they're a sequence of data primitives that are related to each other semantically, and we don't want to have to create a different name for each one individually.

For example, suppose we go back to the houses on a street example. If you're the mailman, do you want to get a set of letters that say 'To: Jimmy', 'To: Franklin', 'To: Dakota Fernandez'? Or do you want to get a set of letters that say something of the form 'To: I couldn't care less, 1003 Main St, Oklahoma City, OK 73008'?

Obviously the latter is preferable. But why is that? Again, we're abstracting function from form. The mailman doesn't have the capacity to remember every single person's name. And keeping track of who moves in and out? What if they expand the street, does the mailman need to learn new names overnight? What about a house that doesn't have anybody in it? The semantic issues quickly become overwhelming.

But, just ordering it based on the street name, and then a pattern of access to the houses on that street? That's something anybody can do. We don't need to know who lives at 1007 Main St, we just need to know that it's next door to 1003 Main St.

So, houses on a street are like an array: we don't want to list every individual occupant, because, and the analogy breaks down here, if we are referencing the data in the address individually, that's as complicated as just having the data in the first place.

The point is, there is value in being able to reference semantically-related data in a logical order without needing to be able to individually note each thing. This is what an array does. In C, we declare and access an array with [], and array literals with {}. So, we might have:

```
void main (void)
{
     int x[5] = {10, 12, 15, 16, 18};
     x[2] = 14;
}
```

There's something interesting here, right? If `x` is the array, why do we place `[]` after the variable name?

The key idea is that an `int` and an `int[]` represent different kinds of objects. An `int` stores a single integer value, whereas `int[]` represents an array, a contiguous sequence of integers stored in memory. When we write `int x[5];`, we are declaring `x` as an array of 5 integers. The `[]` is part of the declaration syntax that tells the compiler this variable is not a single value, but a collection of values indexed from `0` to `4`.

it's sort of like being a house builder and saying 'I'm going to build houses at these five addresses'. If the city inspector comes by and sees 6 houses, that's obviously a problem, right?

Same thing happens with arrays. We call them fixed-length, because we only reserved the memory addresses for 5 integers. We can't then later add a 6th.

There's an easy solution to this, of course. We just declare a new array that will have 6 integers in it, copy the original 5 over, and then add our 6th to the new array. Again, the housing analogy breaks down, but it would be akin to loving those 5 houses, really wanting a 6th matching house, and so moving all 5 houses to a

new street where we actually have 6 addresses reserved so we can build that 6th house, then we sell the original 5-wide chunk of addresses back to the city (or whoever, again... the analogy is breaking down).

There is another thing to remember. When we declare an array of integers, it's different than declaring an integer. But when we declare an array of integers, it's also different than declaring an array of floating point numbers. Every variation in declaration is independent and, without explicit instructions on conversion, completely incompatible with every other variation.

Pointers

Let's go back to that int[] bit. It turns out that using the [] nomenclature is actually a special case of the general category of what we call pointers.

What's a pointer? Remember when we said that when we declare the variable x, we're asking the programming language to keep track of the memory address for us? That's great and easy, but what if we want to take some control of remembering the memory address ourselves? Certainly, we don't want to come in with an outside 'Hey, memory! Give me the address at 00011100 11001101 00101110 00110100 00110101 00101000 01001010 01111111!' (First off, we'd probably say 0x1C CD 2E 34 35 28 4A 7F, because that's a lot easier.) Instead, we'd say 'Hey, where did you store that x value? Can you let me know the address it's at?'

Here's the high level overview part: we almost never care about the actual memory addresses themselves. We just care that they are correct in relation to each other. Once upon a time, we'd have a fixed video memory buffer address of 0x8800 for CGA/EGA/VGA. But again, that's a little silly since it is tying function and form tighter together. Instead, now we'd say something like 'hey, programming language? Query the Operating System, and find out the address for the video memory, and then pass that reference back to me, but I don't actually care what it is, remember it for me'.

The point is, that address itself, the '1004' part of '1004 Main St' is, itself, data and a datatype. We call that datatype the pointer. Again, it's a little more complicated than that, because an int pointer is different than a float pointer, etc. But, we collectively just refer to the whole group as pointers. So, we need to store that data in a variable of type pointer. We can do that like this:

```
void main (void)
{
      int x = 0;
      int* px = &x;
      *px = 10;
}
```

And we'll see that the value of x changes. Side Note: You'll notice that the code doesn't output anything and

the 'void' in the return type of main likely causes you a warning if not an error. The intent is for the code in this book to be examples of concepts, not code that's worth directly copying and using.

In that code example, we have two new operators! That's so exciting! & is the operator that means 'we're dealing with a memory address'. An easy mnemonic device is that & is the ampersand (or 'and sign'), so it means 'address': A -> A. * is the opposite, it means 'there's a memory address involved, but I don't want the memory address, I want what's stored at the address'. * is the star, so it means what's stored: ST -> ST. (We don't talk about how the mnemonic breaks down if you call it an asterisk.)

We call * the dereferencing operator and & the referencing operator. Don't obsess about those names, because they won't intuitively make sense for a while. Reference in this case means 'memory address' and we use & to get the memory address of a variable, and we use * to get the value that's stored at a memory address, thereby undoing the referencing part and hence dereferencing.

Also, keep in mind that * and & are operators. They operate on a data type before the * or a variable after either. They then return the appropriate type, data or pointer.

This becomes difficult, even experienced programmers make mistakes. You will bang your head against the keyboard several times before you're through 'mastering' pointers. Just remember that a regular variable stores data. A pointer stores the address (reference) to some regular variable. And we can use the * to go from pointer to data, and the & to go from data to pointer.

We Put the Point in Pointers

What's the deal with pointers, anyways? C is, not really, but kind of the only language that uses pointers. It's why we consider C to be such a low-level, foundational language. But why is it an issue?

At it's core, all arrays are fixed-length arrays. Once you declare an array, it cannot change its size. If you want to append to it, you need to create a new array. But, that's kind of pain in the neck, so almost every modern language abstracts away the details of fixed-length pointers in favor of higher-order objects that manipulate the fixed-length arrays for us behind the scenes.

That manipulation? That's what we actually want to see. So we're going to do it in C to see how it works. You can simulate all of this in other languages, but it's just a simulation. In C it becomes necessary. (Unless you download a library, or use C++, or create your own tools, but at some level, someone had to and still has to manipulate the arrays).

At its core, pointers could be said to be about fixed length arrays. And everything else just derives from there. That's a strong statement. But here's the question: suppose you have a value that needs calculating. How hard is that to handle? We didn't design computers to calculate over a single input. We designed computers because we needed calculations over hundreds, thousands, millions, or now billions of inputs. And manipulating those billions of inputs will require arrays, and manipulating those arrays requires more direct investment.

We put the [] in Pointers

This is one of those saying the quiet part out loud bits. This example is bad programming, but we're doing it to demonstrate a concept. Arrays are, ultimately, just a shortcut method for creating the special case of pointers where we're going to manipulate many of the same type of data.

```
void main (void)
{
      int x[] = {9, 12, 13, 16, 18};
      *x = 10;
      *(x + 2) = 14;
}
```

Notice that we create the array. Then, notice that we treat x as a pointer! That's because, ultimately, it is a pointer and the [] is just an abstraction on top of the underlying pointers. x just points to the beginning of the array, and the index value inside of the [] is referred to as the offset. As in, how far off of the beginning of the array the target address is. Why do we even have the [] then? Two reasons come to mind: it is one of the few places where the designers of C make a concession in the name of convenience to early learners because otherwise the learning curve would be too steep, and because the special case of the array is so incredibly prevalent that it drastically increases the readability of the code more in a manner that outweighs the extra complexity of having a new symbol. But those reasons are both supposition.

Arrays and Security Briefly

Let's come back around to secure versus insecure programming languages. Why is C considered to be an insecure programming language? There are two reasons and one corollary that come into play and they both refer to arrays.

Suppose we create a variable x. And then we create a pointer to x. And then we treat x like an array and call a value related to it:

```
Void main (void)
{
      int x = 10;
      int* px = &x;
      px[100000] = "F";
}
```

Where is px[1000000]? The short answer is a shrug and a mumbled 'idunno'. But what we do know is that we didn't tell the operating system that we wanted that value. And we didn't tell the programming language to keep track of that value for us. So, what right do we have to access that value? This is a technical permission

'rights', not a moral 'rights'. Essentially, we're writing C code that's so low level, there isn't anyone stopping us from doing this!

There are safeguards, don't worry. We'll get to those.

But, the disturbing question is still: who stops us from accessing data we're not supposed to? Who stops us from just accessing memory beyond the bounds of our function or program's memory allocation? What if that memory we're accessing has bank information or PII or nuclear launch codes or some bitwise variation of Klaatu Barada Nikto? We just don't know as programmers. And as system owners, we don't know that the writer of any given program hasn't done that!

Again, there are safeguards. Foremost is your operating system. This is an issue on the order of 'my deadbolt is easy to pick if you're a trained locksmith', not an issue of 'I never lock my doors'.

The second issue with pointers is also alluded to: not only could we read data intentionally outside of our remit, but we could actually change data. You remember old NES Games and sometimes they would just stall? Or have you ever been using a graphics program or a Steam game on your PC and suddenly you randomly bluescreen-of-death? That's 99.9% likely a pointer bug, where the program accidentally (sometimes intentionally) tried to change a value outside of their memory stack, and that changed value was critical to the continued operation of your computer. On the NES or any old console, tricks like that are ways of getting more performance out of limited hardware. On a PC, it's what we call hacking (malicious or benign).

So, reading or changing data outside of an array's bounds is considered bad form, bad programming, and when expanded to every programmer, a national security problem (potentially).

There are a huge number of barriers in your way to accidentally breaking things. The first is your IDE. It should tell you when it notices you explicitly trying to access an out of bounds array index. The second is your compiler. Out of the box, your compiler should cause a compiler error if it notices you trying to access out of bounds array indices. Third is also your compiler, it should crash rather with an array index out of bounds error. The fourth is your programming language, which should give hard limits on what memory access you're allowed to have. The fifth is your operating system, which should sandbox your program into not letting you access outside of the allotted memory stack. The sixth is, potentially and depending on spec, your hardware may have hard locks that prevent some of these issues.

The problem is that all of these require a ton of overhead, and C generally operates at a low enough level that we don't have the protections of 1-4, frequently we don't have the protections of 5, and 6 is not standard. You can ameliorate this by using C++ and only using libraries while never directly accessing pointers, or using literally any other language.

But we're not going to do that! Instead, you're going to try to run the following code and see what happens (nothing bad! Read the code for yourself and see that we're not sending data anywhere nor are we changing anything, but you were right to be immediately suspicious of someone giving you code and saying 'run this'):

```
#include <stdio.h> // Standard I/O, includes printf()

int main(int argc, char *argv[])
```

```
{
      char[] x = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // doesn't matter, we're just
      for (int i = 0; i < 100000; i++) // Is 100000 more than 10?
      {
            printf(x[i]); // print whatever's in memory there.
      }

      return 0;
}
```

Fun fact! There's probably about a 50% chance this code will just run! How hard was that? If you're not security paranoid after this chapter, you may need to reevaluate your priorities.

There are three primary errors you might be getting, feel free to fix them. The first is that it doesn't like the array declaration. The second is that your setup will hang up on the obvious and intentional array out of bounding. The third is that printf(...) will sometimes hang if you have 0 as the argument you pass to it. Figure out a quick fix to those issues, and you'll be able to see all of the data that's in your memory right now. Most of it is basic boring, but some of it might surprise you!

Side Note: You might also get a 'segmentation fault' when you try to run your code. This is a fun error message that basically means 'you tried to do exactly what the author is trying to get you to do. Bad Programmer!' But why does it say segmentation fault? Once upon a time, when memory was harder to handle than in the modern era, the logical separation inside of the machine was called 'segments' of memory. It meant that programs could be written a little bit abstracted away from the hardware. But whether one segment was on the same machine or another was no longer a fixed issue. So, if you had an array out-of-bounds issue, you might literally be trying to access memory that was in a different machine. So, the point is, when you see a 'segmentation fault', you should imagine the read/write head banging on the inside of one machine trying to get to another machine, and chaos ensuing. Your code won't be any more correct, but at least you might giggle.

The Corollary

This is getting it's own section, because it's important to note. We said that there were two security issues (reading and writing out of bound indices) and one corollary. You'll want to get used to this, because you see it constantly in C code, as well as in most original-source CS algorithm proposals.

The question is: how long is an array. For example, in the most recent code, we created an array of length 10. Suppose we passed that array to a function, how would that function know how long the array is? The modern programming paradigm is to just query array.length, because modern programming languages tend to wrap arrays inside of objects with a bunch of helper functions. But, again, in C we don't have that luxury. So, what you'll see is:

int foo (int* some_array, int n);

That is, when you pass an array to a function, you need to explicitly include the length, otherwise there is no way for the function to know when to stop processing the array. This is not a place where we're going to cleverly show you the 'unleeeeeess'. This is literally the correct answer. The array and the length of the array are completely separate, atomic pieces of data that need to be passed and treated separately. Of course, we'll show you how to create a struct that semantically attaches them, but later and that doesn't really change the rule it just packages it differently. In short, always explicitly keep track of the lengths of your arrays.

Memory creation and deletion

We had to cover Phase 5 separately, because it's a lot. It's actually less than Phases 1-4 in this text, but since it's potentially new, it feels like more.

Speaking of new! There's a new fad sweeping the object-oriented programming world. When you want to create a new object, you use the new keyword, and when you want to delete an object, you use the delete keyword.

We don't have those options, because we don't have classes. Instead, we use a dreadful thing. Remember the evil piglin queen Malgosha? This is kind of on that level.

malloc stands for memory allocation. Let's look at some code again:

```
void main(void)
{
      int x = 10;
}
```

Seems straightforward, right? But consider what we've got there;

1. We want a new int
2. We want that int stored in memory
3. We're going to call that location in memory x
4. We want another new int
5. The second new int is going to be 10
6. We want to assign the memory space referred to as x the value 10.
7. ... There's probably other stuff here, this is just an example ...
8. Start ending the program [this is what the } means in context of main()]
9. Remove our hold on the memory space referred to as x.
10. End execution.

malloc is basically step 2 of that process. It's where we direct the programming language to allocate space in memory and remember where that space we allocated is. Why would we do that? Because pointers. We'll get there, but we could manually create an array like this:

```
void main (void)
[
     int* px = (int*) malloc (100 * sizeof(int));
     px[25] = 42;
}
```

That first line is a lot to unpack. It starts off easy: create a pointer to an integer. But, we need to dive into what pointers are a bit more first. Remember that we said that data and pointers are different types, and that pointers to different datatypes are also different datatypes themselves?

This is why: when we create an int*, we're specifying that the incremental step between this pointer and the next pointer is the size of an int (we're going to call it 4 bytes). So, if px ends up being 0x00004000, then px + 1 will be 0x00004004. Why? Because that's the next integer in memory. Px points to an integer, so px + 1 points to the next integer.

malloc, however, just returns a generic pointer to whatever memory space it allocated. So, we need to cast that pointer to know that it's dealing with ints. Need is a strong word there, we could get away without it. But don't, go ahead and consider it a need-need. Incidentally, when you put a datatype in () like that, you're casting the value. Casting a double d = (double) some_int / (double) some_other_int; keeps you from running afoul of integer division approximation issues. Casting can get weird, and sometimes it's necessary so play with it but don't abuse it.

The argument to malloc is the size (we're not going to worry about units, because you're never going to give it a raw number) of memory you want to allocate and return a pointer to. sizeof() returns a nicely packaged size (we're not going to worry about units, because we're only ever going to compare to other relative units) that we can pass on to functions like malloc. And since our array is going to be 100 long, we want the size of 100 ints.

And then, just to show off, we can access px like any other array. Incidentally, when you initialize an array with [], this is what's going on behind the scenes.

Remember our earlier series of steps? #9 was that it deallocates all the variables it created. But, int his most recent main(), we didn't actually create any variables. Well, we created the data that is a memory address and stored inside of px, and that will get deallocated. But what about the 100 ints we created? Those weren't actually created in main() here, and malloc handles things a little differently.

The point is, we need to explicitly free that memory up on our own. For every malloc, you need at least one (maybe more) free statements. So, properly, our code should be:

```
void main (void)
[
     int* px = (int*) malloc (100 * sizeof(int));
     px[25] = 42;
     free(px);
```

```
}
```

Probably, your program will free any memory you haven't freed at the end. Probably. But, it's entirely possible that it won't. You know how when you call Tech Support and they tell you 'have you turned it off and then on again?' That's because 'probably' means that sometimes it doesn't, and it's entirely feasible that your machine has memory assigned to programs that have been closed. Probably not. Most modern operating systems do a fairly regular garbage collection to fix that. Probably. Probably...

One more issue to consider. We said one or more free statements for every malloc. Why or more? Because consider the following code:

```
void main (void)
[
     int* px = (int*) malloc (100 * sizeof(int)); // 1
     px[25] = 42;
     px = (int*) malloc (101 * sizeof(int)); // 2
     free(px);
}
```

At //2, we allocate some new memory. Great! But what happened to the memory we allocated at //1? Same as before: it was allocated to our program, but we've lost any way to reach it, because we didn't have access to it. The code should free(px) before it mallocs again. There's also a calloc and a realloc, which might adjust these issues. But, all of this pointer nonsense is just here to provide scope and context, but this isn't really a full-fledged treatment of them. If low-level programming or constrained programming like this interests you, definitely look up more information about low-level C programming.

Security Hardening

Let's recap for a moment. Why is security a problem? Well, we've seen that if you can overflow an array, you can access anything in memory. If you incorrectly allocate memory without deallocating it, you can fill the computer with allocated memory it thinks is being used for something. And why does this happen?

Because we have complete control over the computer as programmers? Maybe not us in a given situation, but someone does! And, essentially, security flaws occur when someone elevates their privileges in a system to be able to have the low level access to the system that lets them primarily do one of those two things. There are obscure other issues, but these two are the absolute primary ones. (The authors might even argue that all security issues are really just variations of these two, but that would be a hard case to win.)

There are an enormity of solutions to these issues. Literally too many to cover in a single text on security let alone one where security is fundamental but tangential to the primary concern (Data Structures and

Algorithms). Every single programming language with any significant adoption (a 0.1% adoption rate on the TIOBE index still means a minimum of hundreds of thousands use a language every day) has methods to address security issues. You need to read about the language features to recognize what they do.

C has a number of solutions to this as well. We're going to summarize it reductively, though: It's called C++. And it's part of why you will see C++ much more frequently than C. But, again, someone needs to write the undergirding, so someone needs to have 0-level access to the system (that's primary access, but in a 0-indexed counting scheme).

Briefly:

- Some languages have a sandbox that prevents anything inside the sandbox from accessing anything outside of the sandbox.
- Some languages have a managed runtime environment, where there are resources devoted to watching what a program does and stopping it from doing certain things.
- Some languages absolutely refuse the programmer access to direct memory or hardware interactions.
- Most languages don't have pointers at all, and nearly all modern languages avoid them. (With the exception that they might have a special programming mode that allows it.)
- Operating Systems provide application signing to ensure the application authors are reputable. This is a trust-based system.
- Most reputable providers will provide a hash value (we'll get to that in the hash chapter) for the download to assert that you downloaded the official files. This is a trust-based system.
- There are hardware limiters that control access to memory or disk storage.

We're going to briefly talk about 3 (4, sort of) approaches to security hardening that are very common in C. But first, why are we discussing all of this security stuff in a data structures and algorithms text?

Data Structures and Security

One thing that will make you a rockstar in any field you enter is the idea of alignment. Align your methods with your goals with your results with your metrics. Everything ties in together. That way you don't have one team doing something sloppy and one direction, and it has to be compensated for in another aspect. For example, data analysts frequently have to manipulate data because the people who design the data collection are not the ones who design the data analysis flow.

Your data structures are just data. But sloppily designed data structures will make data storage more clear. Here's the easiest example: suppose you're designing a log-in system. You need a username and a password, right? They're just characters! What more is there?

Here's a basic secret: the less sensitive data you actual retain, the less liability you hold. So, you probably will store the username in plain text (filtered, of course). But you only need to store the encrypted password

(Linux, for example, calls this the shadow file; that's unrelated, but the coolest hacker filename ever). That way, even the system doesn't know the password! It just knows that when the user enters their password to log in, if its run through the same encryption method, the shadow password and the encrypted entry should match. Hence: security vulnerability blocked before it even starts.

This is very much a practical and non-theoretical concern. But it is very much in the spirit of Data Structures.

Algorithms and Security

When we talk security, we frequently mention the three categories: Malicious Users, Benign Users, and Natural Disasters. Let's explore what might happen with a sloppy algorithm quickly:

- A poorly designed algorithm (that, for example doesn't filter inputs or stores plaintext passwords) can have data easily pulled out of sniffed packets on a network. Or, perhaps, someone might know that putting in a '0' for a cryptographic input will cause a backdoor to appear because the designers didn't check for divide-by-zero errors. Every edge and corner case needs to be caught early and often!
- The AK-47 is, in the paraphrased words of every military tactician ever, so simple even a grunt can't break it. A more polite way to phrase it might be the more Roddenberrian quote of: "the more they overthink the plumbing, the easier it is to stop up the drain!" Users will do unexpected things, so creating algorithms that mathematically address every class of correct or incorrect input is important. Just ask the mother of Little Bobby Drop Tables.
- Natural Disasters are the bane of algorithm writers. From one perspective, the algorithm writers are the natural disaster! For example, in 1999 there was a Mars probe that cost something on the order of $100m. It was a joint NASA and ESA collaboration, and it smashed right into Mars at interplanetary speeds producing a new greasy smear. Why? Because, simplified, the Americans entered gravity in as -32 [f/s/s] and the Europeans expected the calculations on gravity to use -9.8 [m/s/s].

The point is, the damage is done because the algorithm was not comprehensively thought out and mathematically provable. In practice, in the real world, most applied algorithms are not going to be mathematically pristine. But you should get as close as you can. There are external factors that weigh into shipping before a minimum level of security is achieved, but again: you should push back against those forces. Security is a cost center during development, but lack of security is an order of magnitude larger cost center during deployment.

valgrind and gdb

Let's talk about specific tools. gdb likely came with your gcc installation. Just like gcc stands for GNU C Compiler, gdb stands for GNU DeBugger. The goal of gdb is to review your code for syntactically-correct syntax errors, logical errors, and more advanced tools than simple parsing will find.

gcc will run and say 'your program passed' or 'I ran into some issues, please fix these'. But that doesn't help when you do something like:

```
while(some_bool == true);
{
     Do_A_Task_A_Million_Times();
}
```

That semi-colon is syntactically correct but semantically obviously wrong and gdb will either let you know that there's a problem there or let you know that's where your program is infinitely looping at. Either way it's a suite of tools to check that you're doing what you intended to do, although it takes a bit of experience to get used to it.

valgrind is a similar tool. valgrind will run your program in a massively slowed down environment. The reason for this is that it, effectively, produces a managed runtime environment (Gary pushed its glasses up, and said "actually, it's not an RTE, it's an instrumentation framework") to monitor everything your program does. So where gdb will provide you with the tools to make sure you're doing what you want to do, valgrind provides you with a set of reports when you do things you're really not supposed to do.

Fun story: Look up why it's called valgrind. Literal definition of the word 'epic'.

The reason this is important is that you should generally at least attempt to run valgrind and gdb on any C code you produce. You will do a poor job, just like any new task, until you get proficient at it. But a little is better than none, and that's what will lead to proficiency.

Finally, both gdb and valgrind are tools to use to prove something is amiss. But they're not tools to provide everything is good or perfect. Lack of a response doesn't mean everything is good: good programming habits, methodology, and good design practices do that. Again, this text is not intended to be a primer on security programming, but you should be familiar with those tools because similar tools are used in most production environments.

PIE – Position Independent Executable

This is included because it is one major approach to fixing buffer overflow issues. If you can read (or change) any bit of data on a computer with low enough level access, how do you find the data you want? Practically,

consider that most computers now have literal billions of data points in memory at any point, and those values are constantly shifting by the microsecond.

Practically, the easiest solution is to just know where the data you want is stored. Imagine a computer: it starts, it loads the BIOS, it loads the bootloader, it loads the same operating system as on a million computers, which initializes systems in roughly the same pattern across all of those systems. The odds that you could narrow down juicy information (like administrator passwords) from billions of possible locations to hundreds of possible locations is pretty good. So, while you can't check billions of data points in real time, you could probably check hundreds.

When we compiled with pie enabled (the -pie option in gcc), we're telling the computer to, essentially, not tell the program where it's loading and we're having the programming language compile everything in such a way that there are no hard coded memory addresses. Maybe the program starts at the very end of a stick of memory, maybe it starts at the beginning. It's a little bit random, a little bit optimized security placement, but it means that the dozens of applications making up an operating system are going to be much harder to track down than normal.

ASLR – Address Space Layout Randomizer

Same as before. If you've taken an ASM class, you know that the order you create the data directly correlates to where it goes in the memory stack. It's exactly the same every time. ASLR is just an option that makes it so that programs don't know how their internals are laid out and, more importantly, so that it's different every time. It's like asking someone to play 'pin the something on the something' at an 8-year olds birthday party, putting the blindfold on them, spinning them around 10 times, and then dropping them off ina redwood forest with a fish in one hand and a handwritten note that says 'better off next time'! That 8-year old is never winning that game. Also, please don't do that to an 8-year old. This is for analogy purposes only.

The point of all of these security concerns is that security is frequently and incorrectly assigned a low priority in development. You cannot add it on afterwards, and security should not be addressed last. So, we've promoted it to chapter 2. We don't expect you to be good, but we expect you to try until you are good. (A cynical person might say 'it won't take that much effort to be better than most'.)

Reference and Value

Quickly, let's return back to the concepts we discussed earlier about pointers. There are two types of variable. And we consider them in how we're passing them to a function. We can pass an argument 'by reference' or we can pass it 'by value'.

If we pass 'by reference', we're essentially passing a pointer to the real and original variable in the calling function. So:

```
void foo (int* p)
{
     p = 0;
}

void main (void)
{
     int x = 10;
     foo(&x);
}
```

We find that the x inside of main also gets set to 0 when p got set to zero. Because when we pass that *'ed variable, we're actually passing the pointer to the original variable that's inside of the main() memory stack. That means that any changes that occur are going to happen to the real, original variable.

This is in contrast to the following code:

```
void foo (int i)
{
     i = 0;
}

void main (void)
[
     int x = 10;
     foo(x);
}
```

Where the x value will stay 10 at the end. Because in the latter case, foo() only received the value that was stored at x and it created a new variable inside of the memory stack associated with foo() and set that copy to 0. But that copy doesn't propagate back or anything along those lines.

Why would we have this distinction and not just always pass as reference? Primarily, because we want to maintain a separation as much as possible with the process between different function. We don't want functions reaching into each other and fiddling with the values. Because the calling function is creating a new variable and assigning it something, one way or the other (whether it's the pointer or the value), there's no loss either way. So, we typically give it the least complex thing. (We'll also learn, later, that this would be a nightmare in some recursive functions, were we rely on being able to pass the concept of the number 1 less than the current number, but we certainly don't want to edit the original number.)

Why would we have this distinction and not just always pass as value? The simple answer is 'because structs

and classes could be really big, and we don't want to have to create them every time'. And there's some truth to that. If you had a struct that stored a 100MB video file, we don't want to have to create a new file, copy the data over, edit the data, and then copy the data back when it gets returned. It's much faster in that case to just edit the original data.

But further for this latter issue, you should have run into the issue of deep vs. shallow copies in your Object Oriented class. Passing by value is, typically, a shallow copy that might cause us issues if we aren't propagating right. So, we allow for just editing the original, anyways.

Lastly, we run into something that we addressed at the beginning of the chapter. What is an array? We all think of it is a unified whole. But that is an external semantic appreciation. The computer doesn't see that, it just sees a series of unrelated (aside from being sequential) 1's and 0's. Sure, something allocated them, but it isn't going to try to figure out who. It barely remembers who, at any given point. Instead, an 'array' is really just a pointer to a a memory address. So when we pass an array, we have no choice but to pass by reference, because there is no real value. (and, of course, don't forget to pass the length of the array, too).

This is an important distinction that you need to pay very close attention to in C. It's only a single character and it's easy to forget as you're reading code or writing your own.

Dynamic Arrays and Vectors

Yay! Finally! Some actual Data Structures!

We're going to start with the default idea of a dynamic array, one that we can change the length of.

Let's start with our data structure:

```
struct dynamicarray
{
     int* data;
     int size;
};
```

Pretty easy, right? We just bundled together the array with the size. Why don't more people do that? Essentially, it's just considered common at this point, people are used to passing both array and size. But we're going to add some functionality:

```
void init_dynamicarray (struct dynamicarray *darr, int n)
{
     darr->data = (int*) malloc (n * sizeof(int));
     darr->size = n;
}
```

This is a quick initialization. We could do a couple of things differently: we could, instead of editing the struct in place, return the struct. But that might cause problems as we expand and have lots of overloaded functions of the same name. We could use different data types, and indeed, a more full exploration might do that by passing type as a typeof argument, but that goes a bit extra for the first data structure.

Limitations: Notice that it's just ints. Notice that we're limited in size by the sizeof(int) so that we could never have an n greater than 2^32. The data is not cleared before we use it, so anything could be in there, etc. There are lots of fixes to add, this is just the outline. Please consider it an exercise to identify as many limitations as you can, and fix the ones that are easy at this point.

How to resize the array, though?

```
void resize_dynamicarray (struct dynamicarray *darr, int n) // the n is presu
{
      struct dynamicarray new_darr;
      init_dynamicarray (*new_darr, n)

      for int (i = 0; i < darr->size; i++)
      {
            new_darr.data[i] = darr->data[i]; // . for structs, -> for struct
      }
      free(darr);
      darr->data = new_darr.data;
      darr->data = new_darr.size;
}
```

There's a huge bug here. You should fix it on your own. These are the kinds of questions we need to ask. For example, we know what we want to happen. But what if that's not what happens? For example, you probably read that code and thought 'right, it cycles through darr and copies everything over to new_darr. But what if the new n is smaller than the existing array? Find a solution and apply it. Hint: you can just declare in the preconditions that when the new n < old n, the excess data values get lost. That's a terrible high-level policy. But it's a very practical low-level policy.

Lastly, if we create the array, we need to free the array:

```
void freearray(struct dynamicarray* darr)
{
      free (darr->data);
}
```

You have three jobs with this code. First, fix the bugs you've found and add any extra functionality you think

would be useful (but, for practical reasons, you're better off not going beyond the init/resize/free scope). Second, consider whether using typedef to create the struct might not somehow be a better approach; consider the pros and cons. Third, write a main function that implements this code, creates and array, changes the array, clears the array.

Don't forget to frequently consider: what other corner or edge cases might we be missing? Consider, what was the algorithm in this code? Does it adhere to the requirements of an algorithm? Fix it! What's the datatype, and how would you describe it?

Also, since this is the first big chunk of code, a couple of notes. The intent is for the code to be fairly self-descriptive. But, there are a few letters that always mean the same thing. n always means the size of an array or more generally the max number of iterations. i means index, and x frequently just means some number that doesn't actually matter. You'll see a few other ones pop up, just recognize that the single letters are there because they're very standard in usage, not because it really is particularly descriptive outside of that standardization.

Practical Applications

If you want more information about dynamic arrays, you can see a similar approach by looking up how C++ vectors are implemented. They're now a fundamental data type (not quite on the level of primitives, but so foundational as to be close. You can read the vector.h source code to get an idea of what a full-fledged implementation would look like of a dynamic array. But, the standard C++ library is sometimes fairly opaque, so don't worry too much if you don't understand the code. It is, however, as optimized as the industry knows how to be and a standard reference that other code snippets are held up to.

Another thing to note is that in the previous example, there's a function called realloc() that we could have used. But because it does a bunch of things under the hood, we're specifically avoiding it in the textbook. But both realloc() and calloc() are absolutely functions you should adapt for your use as you're writing your own code.

Regardless, once again, the reality is that you're never going to write a dynamic array for a program in your life. It's just not going to happen. It's feasible, but unlikely, that you will write a dynamic array that you put into a custom set of tools that you use on a wide variety of projects within the same domain (company, organization, personal projects, etc.). But the idea that you should rewrite what already exists is minimal. This is not to scare you away! So many different improvements and optimizations have been found and suggested by CS students and tinkerers!

But, in practice, in industry, it just doesn't happen very often. It's much more common, even in very low-level/embedded programming, to see people use libraries that are selected to be specifically applicable to the project. There are very few applications that don't have an existing toolset.

But, again, that doesn't stop you from tinkering. And, responsible code use would demand that you read

through the guts of the code to just look for anything hinky. You should also tap into a security forum or clearinghouse so that you can check to see if there are any known flaws with the code libraries you're using.

There's a famous quote: "The koala could have avoided by eating pretty much any other thing." C is a lot like that, in the modern context. If we strip away 'why are things the way they are' and 'how does this work under the hood?', then all of the problems that we're going to address in this class fall under the category of "could have been avoided by programming in literally any other language." But you wouldn't be as good a programmer, and you wouldn't recognize as many of the pitfalls. Think of programming in C for data structures like running with ankle weights on. You don't do it because you're going to be a good ankle-weight runner. You do it because when you take the ankle weights off, you're faster. So, the same thing applies when you go to whichever language you choose to use professionally.

AI Examples

This chapter covers three topics: pointers, security, and dynamic arrays. It is worth considering how each of those aspects might apply to AI.

Dynamic Arrays are actually not always what we want with AI. Why? Because for most AI, whether it's stats, rules, or neural-network based, we're looking at a numbers game. It used to be that 'good data leads to good conclusions', but now it's 'more data leads to correct conclusions'. This is why statistics is now a secret sauce, because we may not know why Netflix is recommending certain movies to you, we just know that their recommendation engine is very accurate. So, with a dynamic array, we're presuming that there's a general case where we want to run the statistics such that the input size can be variable.

The reality is that we want to be able to define the size of the inputs so that we don't have to worry about the overhead of the array resizing aspects. It's important enough that if we have a change in input size, we'll just rewrite the program to account for it (or, more practically, we'll re-compile it with a different input-size compiler directive definition).

AI Security has a number of issues. The primary issue right now is hallucinations where the AI, in an effort to please, provides fabricated facts. Secondary, and rising, however is jailbreaking an AI where a malicious user can manipulate the conceptual space that the AI is residing in, in order to get it to react in a new way (like a sales AI might start suddenly start 'advertising' about free product). But, possibly the most potential harm, is that a poorly hardened set of AI weights has the potential to be unencrypted to the point where sensitive data is shared, or the AI ay be connected to a database that adds that.

Point is, AI Security is difficult because of the previously mentioned secret sauce. If we don't know how the answer is connected to the question, it's awful hard to prove that the answer is not gotten to in the correct way. Tools and approaches are constantly being developed and tested, and it's definitely worth knowing what's the current best practice.

Arrays

Ok, so let's jump into our data structure for AI. The first one is the simplest, but we're going to consider it in three ways. If you've studied formal AI theory, you should understand how backpropagation works in a neural network. If not, suffice to say that the primary driver of a neural network's utility is in the 'weights' and those weights are conceptually representing the line that's drawn between two different nodes in a graph.

The 'neuron' is the data structure that we'd use here when we create a small network. But, again, the number one metric in neural networks is number of computations, and that means we want to be as fast as possible. In short, we don't need to model the entire neural network using a set of neurons, because all that really matters is the weighted connections between each neuron. And those weights get updated with every pass through the training data.

Data Structure 1: Matrix Weights

The point is: we can just store those weights in a matrix! What are the weights? They're proportions. So that means that that they've got a decimal (because 50%, for example, is actualy 0.5000) and need to be stored as a floating point value. So we could create an array of arrays to handle this, or it's even more effective to create a 2D array: double[rows] weights;. We can process through this using whatever training algorithm we develop, and the calculation time should be reasonable for each pass.

Side Note: Why is this effective? Matrix Math. Given a matrix defined and represented in a certain way, there are well known algorithms that will calculate updated matrices relatively quickly.

Data Structure 2: Array Weights

Here's the question: What is a 2D array? Take the following matrix:

```
[
      [11, 12, 13],
      [21, 22, 23],
      [31, 32, 33]
]
```

Compare it to the following array:

```
[11, 12, 13, 21, 22, 23, 31, 32, 33]
```

What's the difference? We just considered in this chapter that arrays are not really unique, they just represent a

series of, effectively, unrelated data values that happen to be stored sequentially. So, for example, the sub-array [11, 12, 13] just means that the numbers 11, 12, and 13 are stored in each of 3 sequential memory addresses. What about [SomeArrayA, SomeArrayB, SomeArrayC]?

If the array means that they're just stored in sequential memory addresses and we've already established that the size of 'sequential' depends on what we're storing, could we imagine that SomeArrayA, B, and C are all stored in sequential groups of 3-int chunks of memory?

So, in essence, if we store that array, it's just a series of 9 sequential memory addresses, and we're using the [][] notation to ask the programming language to keep track of it for us. What's the difference between [m][n] and programming language assistance and [m*n] and we track it ourselves?

The short answer is nothing! We can do it either way, and the most (practical) effective way to determine which is better is to just time both approaches and see which one is faster (assuming all other factors are the same). Theoretically, you could math it out ahead of time, which his what a pure Computer Scientist would do. But, in almost all real world cases, you're going to want empirical evidence.

But... we're not done yet. What's the difference between [m*n] and *(arr +row*n + col)? Essentially, you can access any point in the array just by manipulating the pointers, which means that you could bounce from any one part of the matrix to any other part by just manipulating the address the point is pointing to.

Data Structure 3: Floats, you say?

Floats... why did it have to be floats? We presume floats. But, the fun part about discrete math is we can do whatever we want. Suppose the number 96! You might say 'it is just some random number'. And that's true in regular math. Practically, just a single number doesn't give you any context about what it is and what it means.

But, computers never use real math. Computers simulate real math (we'll come back to this idea regularly, it's why you have to take numeric methods). But, ultimately, real math counts to infinity and a computer always has a limit. That limit might be in the trillions or higher. But it's still a limit that's tiny compared to infinity.

This means that on a computer, that number 96 is always, always in the context of MAX. Suppose that 96 is stored as an 8-bit integer. Then we can immediately suggest that 96 is the equivalent of 0.375. Why? Because 96 / 256 simplifies to 3/8. The point is that the conceptual difference between a float that has to be between 0 and 1, and an integer is always fragile. They're the same thing.

For our matrix that needs to be calculated over very fast, there's no reason we can't represent those numbers as integers as a percentage of MAX, instead of float.

How would we tell what's more appropriate to which application? The easiest case is: when the float can bet higher than 100%, we can't use this int simulation (can you think of an obvious fix to this?). But otherwise, the most effective approach is to just time both and...

We worry about fidelity (or precision). While 01100000 is 37.5% of 1.0, what's 47.65%? Or

37.500000001%? We can't really represent those values as well with 8 bits. So, wile the integer representation is almost always going to be much much faster, that's because it's much less precise, only being able to count multiples of 0.00390625. But, maybe we don't need precision?

In fact, Microsoft recently released what they call the 1-bit model. In this approach, the weights of a neural network are converted to single bits in some manner. And then the neural network is run and the answers compared to the floating point version. Their claim is that the results are very similar, which means that the algorithm can be run on general purpose CPUs (better at integer math) and don't need GPUs (better at floating point math).

Can you consider another metric that might be improved by approximately a factor of 8 with this approach? Is 8 really the best estimate of how these 1-bit weights compare to floating point weights?

Summative Approach

The common refrain for this is: which one is best? Try them all and test it to find out. But, you need to be familiar with what your options are before you can test to find out. "I put all the floats in an array" is much less convincing than "I put all the floats in an array, and then tested it compared to proportional integers, 1-bit weights across matrices, arrays, and direct memory access, and here's which one is fastest'.

CHAPTER 3 - WHAT TO PROGRAM WHEN YOU'RE PROGRAMMING

What is an algorithm? We defined it in the last chapter. But what is it really? It's an approximation of how the world works. We are discretizing the universe and attempting to define a set series of behaviors. But, for example, suppose we defined an algorithm that involved going to the doctor when someone's temperature is above 100.0F or below 96.4F. Why those values?

The short of it, we are trying to create sense out of complex situations that are too complex to be addressed continuously. We can make the exact same concern about data structures: we're imposing rules on situations in which there are no rules. For example, consider the human name, and then consider McKenzie's treatise about how every assumption you can make about names has a real examples that doesn't fit the bill: https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/.

Algorithms and Data Structures are both examples of conceptual models. And, like democracy, they are the absolute worst solution to a problem... except for all of the other solutions. So why do we use data structures and algorithms?

1. The simplest answer is continuity. We need to use something to solve problems, so any solution is better than no solution. Because we have a long history of solutions to known problems in a certain context, we will tend to continue in that context because it's easier than reinventing everything that's come before in a new context.
2. The second reason is that data structures, algorithms, and other rigidized constructs are the ones that allow us to come to the right answer the most frequently and most easily communicate our process between multiple stakeholders. Alternately, we acknowledge their failings, but because those failings are less than other approaches, we stick with it. Science isn't the best answer, it's just historically better than alchemy and shamanism.
3. The third answer is the one that is the most salient to the theory of computer science: Given a correctly constrained problem domain, data structures and algorithms can create a discretized and mathematically perfect solution. Then the task becomes loosening the constraints on the problem domain in such a way that we solve the biggest problem.

Please take a moment to decide whether or not this aligns with the three core approaches this text is taking: theory, practical application, and a specific domain.

Regardless, this chapter is going to begin by defining the most atomic units between data structures and algorithms, before we move on.

One Unit of Data Structure

This is an over simplification. However, it's also a good start place for the situations in which a simple solution isn't sufficient. The most basic unit of a data structure is the node. But before we get into nodes, we need to talk about the practical differences between an array and a linked list (you don't know what that is, yet).

Arrays

We've described arrays so far. An array is a block of reserved memory that is semantically similar and logically sequential in the memory heap of the computer. This is important for a lot of different tasks, because it's fast. For example:

```
// given a pointer to an array, p, and a length, n
int* arrend = p + n;
int* curr = p;
while (curr < arrend)
{
      *curr = somereflexivefunc(*curr);
      // or
      //somereflexivefunc(curr);
      curr++;
}
```

This code should, essentially, edit an array in place as fast as the somereflexivefunc() can operate (this is called mapping). There's very little overhead. Not only is that a solid strategy, and worth pursuing, but it's the simplest solution. We might even consider it the most default solution.

In one sense, we can think of an array as being 'no data structure', but by another respect we can consider it sort of the initial data structure, or brute force data structure. 'You want more numbers? Just put in more numbers!'

We run into a fundamental problem, though. But first we need a side tangent about computers:

Side Tangent about Computers

Do you remember that time period that Pepperidge Farm Remembers? That time period was the heyday of computer technology! We invented ENIAC, which at a measly 30 tons only took up a single room! And, it could do hundreds of calculations a second! Multiplication, no less!

Levity aside, it's important to remember that starting in the late 90's (processor architecture, BIOS improvements, introduction of USB, etc.) you no longer had any real possibility of typing faster than the computer could keep up. Remember, this may be before your lifetime, but it's not before the lifetime of the science part of computer science.

The current era, from the late 1990's to the early 2020's, is largely a constraint-free programming and technology development environment. It's not that constraints don't happen, but that they are very removed from the average consumer (probably even the average developer) experience. The reason this is important computers in every generation are basically magic when compared to computers from 15 years earlier.

So, what's wrong with an array? Consider you're Amazon, Google, Netflix, or any other FAANG company, and you've got to put together a database of 100 billion structs in a row (say, every search that every user has made in the last several years for a predictive engine). We really like the fact that we could just cycle through all 100 billion structs and run a function on them as fast as we can cycle through the memory. That is an ideal situation!

But! 100 billion entities, at minimum, is going to be 100 gigabytes of data. And part of what makes the array into an array is the fact that the data is continuous. That is, a single monolithic chunk in the address space. And that is a huge amount! So much so that it's practically impossible to presume that a system will have this available (side note: in a situation such as this, depending on budget, specific hardware could be designed, which might guarantee it, but now we're in the realm of engineering).

If you were bored, come back to paying attention, please. Notice that the scope of that example escalated super fast. So, let's make it more real. In the 1940s, with that ENIAC computer (the first 'general purpose computer) we mentioned, how much memory did it have? 20. Where's the units? No units. It had 20 'accumulators'. Whatever you could shove into those 10-digit accumulators, that's the memory you had available. Digits? Yes, before someone realized how much cleaner binary computing was, we used decimal computing (literally 10 light bulbs, and how many lit was what number it was).

The point is this: Neither of those hardware platforms matter. We don't know which one the user is going to need, so we have to presume either a transnational global megacorp's flagship super computer, or a glorified slide ruler. As far as we are concerned, they are exactly the same, the only distinction is in the amount of time they take to solve a problem. And, compared to problems that take longer than the heat death of the universe, the actual capabilities of the computers don't matter.

So, the point is this: every concept in data structures and algorithms applies now just as much as it did in 1941 and will in 2041, it's just a difference in which places it applies. In 1941, they were pushing the boundaries of science by pre-calculating mortar firing solutions for soldiers in the field and in 2041, they'll be tracking eyeball movements of every VR Social Media User in The Collective.

Pick a scale that makes the most sense to you, and when we discuss constraints, think of a problem that applies those constraints to what you're doing. And remember that, in the end, the best algorithm is the one that gets the most done in less than the maximum time on the target hardware. The science is true, but it takes a back seat to practical application.

Linked Lists

We've established a problem with Arrays. They're hardware dependent. For example, what if you want the ENIAC to solve a problem that takes 21 10 digit numbers to solve? It's physically incapable of it. So, we need a solution that will work regardless of those hardware limitations. (This may include a lot of handwaving, and leaving it to the engineers to figure out.)

This is where the science part of the name Computer Science comes from. We need to figure out how to abstract concepts away from each other and focus on a single variable at a time. 'Variable' is a bit of a misnomer here, but it's still true that we want to hold concepts constant while we adjust individual metrics.

In the case of an Array, the goal is to semantically group data together under a single overarching label such that we can address each element in turn. The brute force solution is to just reserve a large chunk of continuous memory. But, and we benefit from hindsight there, the phrasing itself starts to provide the solution. The word continuous sounds like a constraint, and what we need to do is loosen constraints. So, could we provide all of the benefits of an Array, but without the requirement of continuity? Absolutely!

This is where a linked list comes in. But first, we need to describe the unit of a linked list. In an array, the atomic unit was the indexed value. arr[0], arr[1], ... arr[n] but in a linked list, the atomic unit is the node.

```
struct node
{
     int payload;
     struct node* next;
};
```

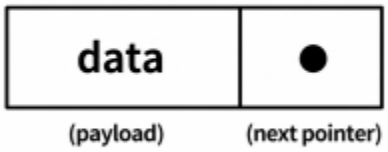


Fig 3.1 Node Structure

Fig 3.1 shows the node structure. This opens up a lot of observations, all interesting but none of them vital right now, just foreshadowing:

- int doesn't have to be int, it can be whatever data type the payload is in.
- Any data type can be represented by other data types, so it can also just always be int.

- Why do we need to add the next when array doesn't need it? Because an array adds the constraint that next is always just the address of the self plus 1 sizeof self, which is what we mean by continuous.
- Most people in C will use typedef to simplify their code, we think it is clearer in context but less clear out of context.
- Notice that next will always initialize to null, which is a security potential bug. Always check if (curr->next != null) before retrieving curr->next.
- This bears repeating. Every single time you reference curr->next, you should check that you've verified curr->next != null.

This works better with pictures, but we can also use words to describe how it works. We create a pointer struct node* head. This head points to the first element of the linked list. Head->next will be the second element of the linked list. The second element ->next will be the third element and so on. How do we know when the list is over? Because somenode->next == null.

If you are comfortable with an array, we can create a conversion function like so:

```
struct node* arraytolinkedlist (int* arr, int n) // 1
{

      struct node* head = malloc(sizeof(struct node));    //2
      head->payload = arr[0];

      struct node* curr = &head; //3
      for (int i = 1; i < n; i++)
      {
            curr->next = malloc(sizeof(struct node)); // 4
            curr = curr->next; // 5
            curr->payload = arr[i]; // 6

      }
      return &head; // 7
}
```

There's a lot to unpack here.

1. It's much more common in C to pass the array as a pointer and also the array length. Both are needed. But notice that the return type is just a single pointer, which implies that the end is built into the data

structure and we don't need to specify length.

2. We need to 'seed' the Linked List in some fashion, because we can't create something completely out of nothing. In this case, we created an actual node value rather than a reference specifically to demonstrate it's kind of weird and it's much more standard to have all pointers or all values. We'll show a corrected version momentarily.

    - High level concept note: This is actually a memory error. Remember that C doesn't actually have garbage collection. When arraytolinkedlist ends, it deallocated the entire memory stack. So the program no longer recognizes that *head is a part of it. But, if the memory stack for arraytolinkedlist started at 0x12, and therefore head was assigned at 0x14, the data inside 0x14 didn't get overwritten with 0's, it just stopped being recognized. When we return &head, we're really returning the address 0x14. Because the program no longer recognizes that it 'owns' 0x14 doesn't mean that the physical transistors somehow got removed from the hardware. As a result, it'll still reference 0x14 and this will all work until the unallocated memory gets allocated to a different function which may or may not change that value. Point is, it will work 99% of the time, which is the equivalent of saying 'it will crash and fail at the worst possible moment'. We'll fix it.

3. **curr is a standard variable name. There are a few four-letter standards.**
    1. **head** – the first node in the list, or at the least the node we treat as
    2. **curr** – we will traverse the list the same way we iterate over an array. curr is the linked list equivalent to the arr[i] in an array.
    3. **next** – curr->next is normally feasible. But there are a few standard algorithms where curr->next->next is important, but awkward. So sometimes you'll see next assigned as curr->next.
    4. **tail** – Frequently, we don't care about what the last node is, it's just the node such that ->next == null. But, occasionally, it will matter for convenience.
    5. **prev** – Notice that the ->next system allows us to go one direction, but not backwards. For some algorithms, knowing one or more of the previous nodes is important.
    6. **temp** – Just some node we need to remember, or maybe our best current solution to be compared to others.
    7. **node** – if we don't call the struct node a node (which we shouldn't, it's too generic), this would just be the pointer to some random node we need to remember (such as being inserted or whatnot)
4. This is an important security concern! Because we are using malloc, we must have a corresponding free() value. We need to also prove that we never create a node such that it isn't free()'d. For the sake of security, we'll add that here as well. This is becoming more and more important over time, not less important.
5. What was the rule earlier? Always check if ->next is equal to NULL before we address it. "But we just assigned it, it can't be equal to NULL!" you say? Great googly moogly, no! Those points when you would think we don't need to check are also when we need to check. The error checking is there for when

things go wrong, not when they go like we expect. We'll fix it shortly.

6. This is the actual accomplishment of the task, where we assign the actual data from the array to the linked list. Everything else is overhead.
7. The actual linked list doesn't matter. The only thing that matters is the pointer to the head of the linked list. Everything else can be derived from there.

Why does this work? At the end of the arraytolinkedlist function, everything that was assigned as part of the function goes away, because all of the variable handles that we created in the function get deallocated along with the function. Except for things like malloc()—which happen outside of the function's memory stack and assigns the allocated memory to the program rather than the function—everything gets removed. Since we only care about the memory bits that were assigned with malloc(), we should be good.

Except we didn't assign everything with malloc(), and we left a few other errors in there to highlight that they need fixing. Every time. We'll also tweak the structure slightly to make it more uniform:

```
struct node* arraytolinkedlist (int* arr, int n)
{
	if (n <= 0) return NULL; // 1

	struct node* head = NULL; // 2
	struct node* curr = NULL; // 3
	for (int i = 0; i < n; i++)
	{
		struct node* temp = malloc(sizeof(struct node)); // 4
		if (temp == NULL)
			return head; // 5

		temp->payload = arr[i]; // 6
		temp->next = NULL; // 7

		if (head == NULL) // 8
		{
			head = temp;
			curr = temp;
		}
		else
		{
			curr->next = temp;
			if (curr->next == NULL) // 9
```

```
                    return head;
                curr = curr->next; // 10
            }
        }

        return head; // 11
}

void freelinkedlist (struct node* head)
{
        struct node* curr; // 12

        while (head != NULL) // 13
        {
                curr = head;
                head = head->next; // 14
                free(curr); // 15
        }
}
```

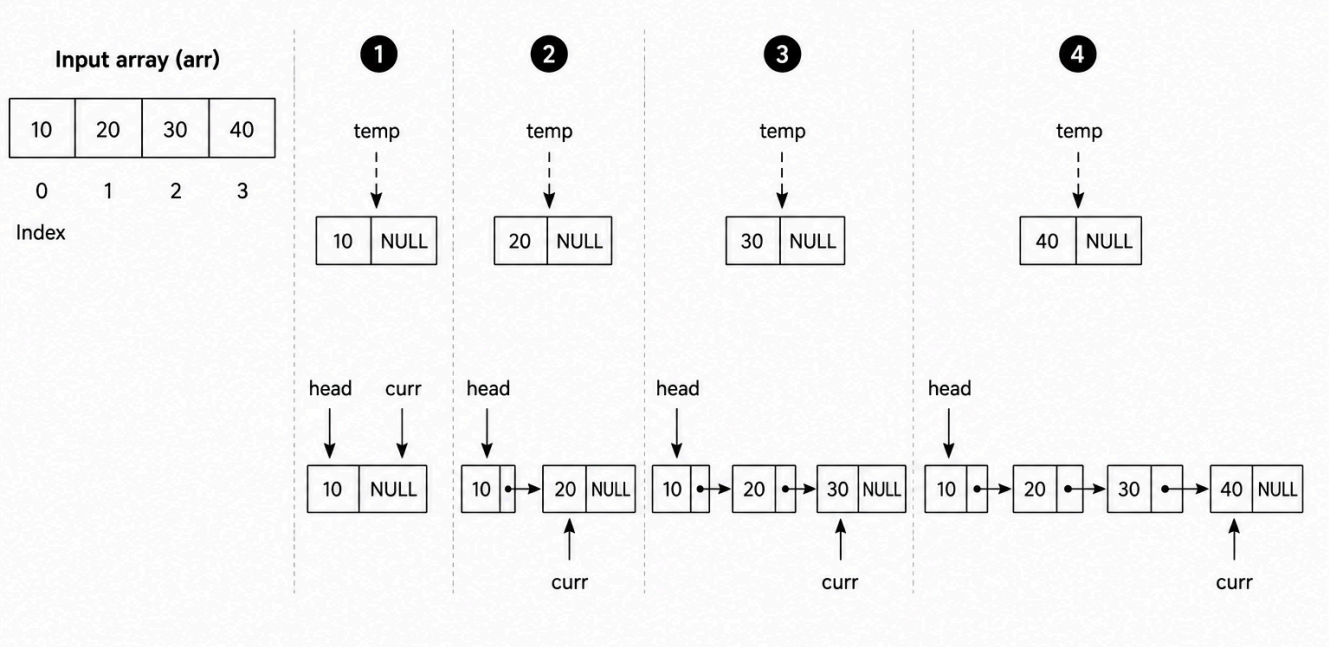


Fig 3.2 array to linked list illustration

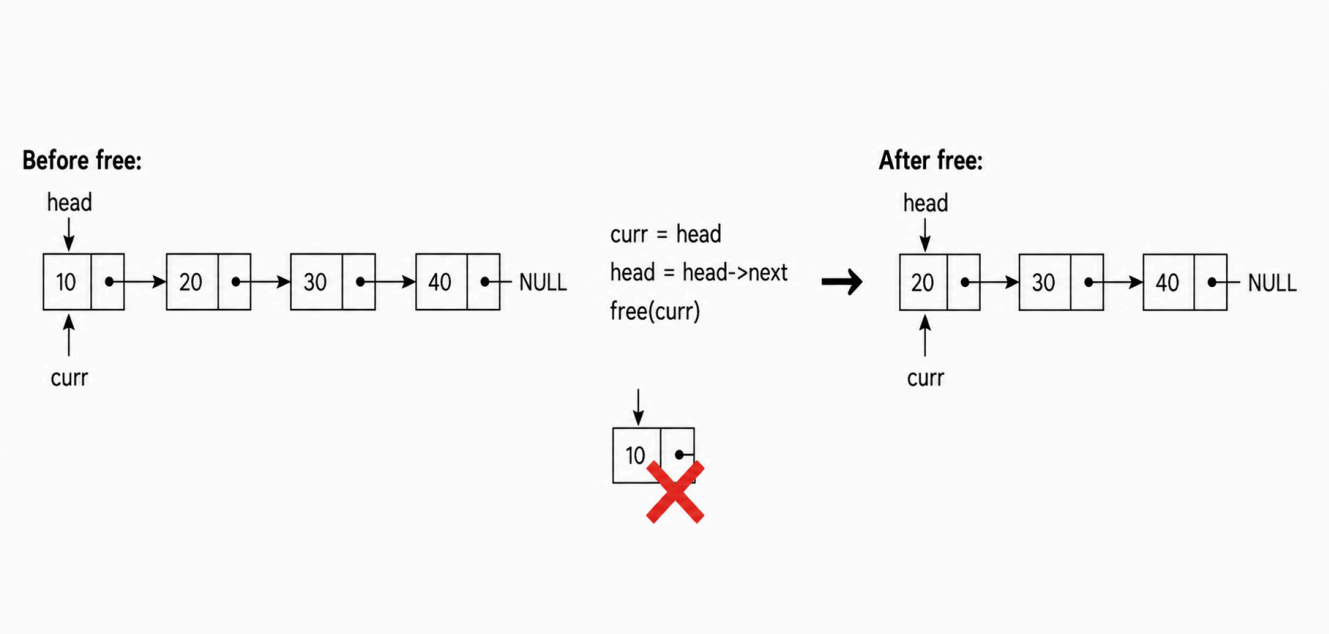


Fig 3.3 Freeing the head node in a linked list

Fig 3.2 illustrate the array to list () Let's cover each point in turn:

1. When we talked about an algorithm having input, we could also have framed that as preconditions, or things that need to be true in order for the algorithm to run. For example, if we have a math function of any type, there's like a precondition that the value is an integer. We can enforce that precondition however we like, but since n is an int, we have to presume it's a number. In this case, we're also adding the precondition that you can't, by definition, have a negative value for length.

    - That phrase, 'by definition' is not just a rhetorical argument. It's actually the logical framing of our proof. We'll get to proofs later. But in Computer Science, if you use the words 'by definition', you should have a formal definition ready and handy to justify that statement.

2. Notice that head is now a pointer and starts as NULL, as in 'we haven't assigned it to anything yet'. This both solves a consistency problem, but also introduces some other potential issues. For example (this does not happen here, but we can't implicitly discount it) what if head already had some data and we just changed head to point to somewhere else. What happens to that original data? It's gone, and we can't access it. That's a memory leak. We have to be able to demonstrate that there was no data there to begin with (which we do by creating a new value) before we overwrite it.
3. Notice that head and curr are now pointers. This is going to help us because everything will be consistent. We don't have to guess about how to handle a given value, whether it's a pointer or a value, we know

everything will be a pointer. That's a big help to demonstrating algorithmic correctness.

4. We create the node completely separate from the linked list. This is an important part to note, which we'll cover momentarily: proof by parts. The node itself is different from the linked list. Operations that affect a single node can be provable and then that proof can be built on to demonstrate the emergent behavior of the list.
5. malloc() is a low-level C call. Can it fail? Sure, why wouldn't it be able to? What if you don't have enough free memory to create what you're doing. What if the memory has a physical imperfection? What if the operating system glitches for a moment. Robust design will handle it (either by compensating or by failing gracefully). But this line is even more important! Notice that we actually return head still. Why would we do that if we failed? If head == NULL, we need to return NULL anyways. If we've already assigned some data to the linked list, we need to return head so that the calling function has a record and knows to progressively free() the linked list. We could do this by adding an error value or logging an error and so on, but those are all too implementation-focused and veer way too far from the theory.
6. We just checked that temp != NULL, so this is OK. We need to veer into a little bit of the logical structure of security and error checking. But in this case, if we can't trust that temp is not NULL and also not a new node that we just assigned, there's a deeper set of troubleshooting than fits neatly into a theoretical package. This is just a note to say that at this low level, that level of paranoia is not unwarranted, it just digresses this description too far.
7. This should already be NULL. But why? In short, we know that C probably and incidentally happens to automatically create new pointers as NULL (or does it? Maybe this is a test! Go look it up!). But, broadly, we want to make sure that this is portable to other systems and that any readers are explicitly aware of what we intend. We have an attribute, we need to explicitly set the attribute.
8. We have two states that we are concerned with. This is part of the algorithm design we'll deal with later. Remember that data structures exist to be part of an algorithm and algorithms execute across data structures. The two topics unrelated, but inextricably coincidental.

    - The first state is: there is no data in the linked list, hence head is NULL and we need to assign a node to head.
    - The second state is: there is data in the linked list, hence head is not NULL, and we need to assign a node as the next attribute to the first node in the sequence whose next attribute is NULL. (Helpfully, we thought ahead and already keep track of curr instead of finding it again every time.)

9. This check is superfluous in practice. But, since the rule was just stated that we always check if ->next == NULL before we access it, this check is left as a reminder.
10. We added temp to the end of the list, and now no longer care at all about the second-to-last of the list, so curr gets redesigned as the last.
11. Notice we can just return the pointer to head, we no longer need to return an int* and int. (Which is a

constant question: how do we return an array of a given length if we can only return one value in C?)

12. This is the classic set up for what's called an inch-worm traversal. Just like an inchworm stands on the back legs, tentatively reaches forward with the forelegs, and then tests the forelegs before pulling the back legs forward, this traversal method sends one pointer forward and then the other follows. One framing of this is save, advance, act [on the saved].
13. Again, we have two cases which we care about:

    - Head is NULL, in which case we are done.
    - Head is not NULL, in which case we should delete a node from the linked list. The only node we know exists is head, so we'll delete that one. (This presumes the special case where we're deleting the entire linked list.) Fig 3.3 illustrate the freelist()

14. This is kind of a pivotal step. We're, essentially, discarding the previous head node and moving to the new head node. We need to make sure that we have something left behind to point to the otherwise unreachable node.
15. Everything so far was to algorithmically prove that we could now safely delete curr without accidentally leaving an allocated but unreachable chunk of memory elsewhere.

Here is your exercise: explicitly apply each of the five components of an algorithm from chapter to these two algorithms. Incidentally, in creating a linked list, we created two algorithms, you should also identify exactly what the two algorithms are.

Pseudocode

One of the things that's important to note is that there is a stark divide between the theoretical and practical applications of computer science. For example, the previous code was an algorithm, but it is framed within the specific and actionable context of the C programming language. What if we abstracted away all of that and just used pseudo code to document?

```
define arraytolinkedlist (arr, n)
     if (n <= 0)
          return
     end if

     head = new node(arr[0])
     curr = head
     for i in 1..n
```

```
            curr->next = new node(arr[1])
            curr = curr->next
        end for

        return head
end
```

Notice how much easier that is? And that's totally OK! However, in a lot of ways it's not very practical. One important skill to have is the ability to go back and forth between formal or semi-formal pseudocode and an actual programming language and back. This is because, at the end of the day, computers run actual code, but humans problem solve in pseudocode. Where it makes it clearer, we will endeavor to include both, but will heavily lean towards C, since that takes care of two birds with one stone.

The biggest downside to relying on Psuedocode is that it is free from all of the actual mechanical bits. It's sort of like the old joke about the farmer who was trying to cut costs, and had this great plan: every day he'd feed his horse a little bit less than the day before. It was going great, until the horse ruined it by starving to death. Reality gets in the way of good ideas. So, for example, notice that we don't need to malloc() or free() in this pseudocode implementation. Be aware of the language you're using and how it might affect what you're doing.

Math

There is one more level of abstraction away from pseudocode . Pseudocode is actually still constrained by the idea of an iterative and ordered series of steps. But we could mathematically justify that this is still correct.

For example, an array is just a matrix, or an ordered relation of some sort. So, we could refer to the array as $A = [a_0, a_1, a_2, ..., a_n]$. And the linked list could be considered to be a recursive process such that each node creates a node inside of it. So, we could say $LL = node(a_0, node(a_1, node(a_2, node(..., node(a_n, \{\}))...)))$, and then we just have to find the situation such that $A = LL$. Or:

$$[a_0, a_1, a_2, \ldots, a_n] = node(a_0, node(a_1, node(a_2, node(\ldots, node(a_n, \{\}))\ldots)))$$

Which is actually statically isomorphic (fancy way of saying: same dude, different suit). As in, we haven't done anything, it's literally the exact same thing in a different format. So, for all values $a_0 \ldots a_n$, this statement holds true. And we could demonstrate that. But we aren't going to, for a very specific (and arguable!) reason:

We categorically reject the utility of pure math-based algorithm design.

In the very narrow space of people whose specialty is algorithm design at a mathematical level, there is a lot of utility in it. But, there's also not a strong entry-level market for that sort of role. For literally everyone else, there will always be some smidgeon of practical implementation involved in any algorithm design. For example, creating a linked list is a tautologically true process. We hand-waved over the unwrapping process of

turning node(a_0, ...) into [a_0, ...], but any proofs along those lines would be better spent handling the nuances of application in a practical environment.

This is a very context and content-specific statement to make, and is subject to change with technology level and education level.

Testing and Modular Design

So, what is a linked list? There are a dozen answers to this. Because Object-Oriented Principles (OOP) are easy to access and you have likely been exposed to them, we're going to use that framework to address this. Here's a quick refresher of the four pillars of OOP:

- Abstraction: We ignored the details of the underlying framework and trust that it just works. For example, when writing a program that uses an integer, we don't dig around in the hardware and insure that the processor is a big endian architecture with an 8-bit word. We just assume all of those bits work.
- Encapsulation: The stuff that's on the inside should be on the inside. We should strive to effect the lowest scope possible with any algorithm we design. Preconditions, Postconditions, and Objectives should be explicit and narrow.
- Polymorphism
- Inheritance

We don't really care about the last two, so we're going to change them a little bit, well designed modular algorithms are:

- Abstract
- Encapsulated
- Readable – It's clear what each step or stage of the algorithm does.
- Reusable – Repetition is extract and turned into a separate function or other construct.

When we ask the question 'what is a linked list', we need to look at it from this perspective. The user will likely expect to just be able to write:

```
Linkedlist* l = newlinkedlist([0, 3, 6, 9, 12]);
```

That's it. So, to the user, a linked list is an actual thing. But as a programmer, you likely included:

```
typedef struct node
{
```

```
    ...
} node;
```

So, the user doesn't have any idea of how the internals work. They just know that they can write linkedlist_indexof(3) and get back a 1, because you also defined a linkedlist_indexof(int) function.

It is important to recognize this perception of the 'data structure'. But there isn't really any linked list. We just saw that it was a bunch of independent nodes that just happen to link to each other, in the same way that an array is a bunch of datatypes that happen to be contiguous. The semantic relation is something we're adding from the outside.

Proof by Parts

This is actually good for us! Because it means that our proofs are much easier! We don't need to prove that the linked list works as a global whole, we just need to prove that each individual node does what it's supposed to do. And that each function performs its appropriate function as expected. If each node performs as expected, we can demonstrate that the group of nodes will do what we want to do.

For the most part, we will leave provability to you, when we don't address it. We're only going to address it when it has a value add. Generally, proofs are mathematically constructed, and we're more interested in logical and conceptual constructs.

Proof by parts is a form of inductive proof of correctness. We're proving that the algorithm does what we say it does, and probably we're simultaneously proving it does nothing else (that is, there are no side effects).

We'll get into this more in a later chapter, but it applies to what we're doing now.

Building Blocks

What this means is that smaller and more narrow is more useful. Take our linked list. Suppose you write a big long function that does an intricate task. You probably have more than one function in there. We can use three general rules to tell us when this is almost certainly true:

1. Do you have more than about 20-25 lines of code? It's probably two functions.
2. Can you describe what the function does in a single sentence without using 'and' or 'but'?
3. Is it worth the time and effort to split it into its component parts.

Take the linked list we just wrote. Suppose we had a function that replaces a node in the list with a different one.

```
void replacenode(struct node* head, int oldvalue, struct node* temp)
```

```
{
     if (head == NULL || *head == NULL || temp == NULL)
         return;
     struct node* curr = find(head, oldvalue);
     if (curr == NULL) return;
     if (*head == curr)
      {
       temp->next = curr->next;
       *head = temp;
       free(curr);
       return;
      }
     struct node* prev = find_prev(head, curr);

     if (prev == NULL) return;

     temp->next = curr->next;
     prev->next = temp;
     free(curr); // 1
}

struct node* find(struct node* head, int x)
{
     if (head == NULL) return NULL;

     struct node* curr = head;
     while (curr != NULL)
     {
           if (curr->payload == x)
                return curr;
           curr = curr->next; // 2
     }
     return NULL;
}

struct node* find_prev(struct node* head, struct node* target)
{
```

```
    if (head == NULL) return NULL;
    if (head == target) return NULL; // 3

    struct node* curr = head;
    while (curr -> next != NULL) // 4
    {
          if (curr->next == target)
               return curr;
          curr = curr->next;
    }

    return NULL;
}
```

Notes:

1. We'll mention this once, and otherwise kick the can down the road: this is an example of technically correct design. However, there is a current design paradigm of ownership (spearheaded by Rust), that may affect whether, in context, this is the correct place to free this. It is our opinion that, at this level, this confuses the issue, but we encourage you to research about the ownership model.
2. We don't check if curr is NULL with an if statement. But the while is, essentially, a repeated if statement, so we are still following the rule about checking.
3. This is a bug in our current program. Solving it distracts from the point, so we leave it to you. Can you fix this algorithm such that it still works if we need to replace the first node, and still holds to the modular design we're describing?
4. We're technically iterating over curr->next, we just checked if head was NULL, and then assigned curr to NULL. Does that satisfy the check? How would you change it to make it more apparent?

Notice how many lines that is, total. If that was all one function, we'd have several issues. Let's frame it in terms of our four module design rules:

- Abstraction – replace() would be intimately concerned with the internal workings of both find() and find_prev().
- Encapsulation – There's no real issue with encapsulation. (One could argue, a bit spuriously, that the various currs, nexts, and prevs all interact poorly).
- Readability – 35-40 lines of code in a single function starts to get difficult, like reading a novel where a character monologues for three pages straight. It's got it's place, but it's rarely as often as the author thinks. Instead, we now have three functions of less than a dozen lines, much of which is boilerplate

escape clauses.

- Reusability – This is the real issue: how often do we need to replace one node with another? Who knows, but it will be intrinsically less often than the frequency with which we need to find() a node, because we use that function inside of replace()! So, we expose an extra utility function for everyone to use and its use is standardized here.

There are a number of ways we can improve this tool, but it's fine for now. The point is that we're trying to demonstrate that splitting everything into smaller and smaller pieces will make things easier. There's a point of ad absurdum, but it's much easier to fix 'too many small functions' than 'one big giant function that tries to do too much'.

All of this applies to the pseudocode as well. (Further, mathematical proofs will almost always require that the specific atomic tasks be separated.)

Documentation

This is the most important part of any sort of testing on either the data structures or the algorithms side. How does anyone know what your function is supposed to do? What it actually does is easy: just run the program and/or trace the code. But, is what it's doing what it was intended to do in the first place?

This is the distinction between internal and external consistency. If your code finds all instances of an element in a linked list, but then crashes every time from a null pointer error, that's an internal consistency issue. But if your code finds all instances of an element in a linked list, and then returns one of them, that's an external consistency issue between your intentions and the actual code.

So, documentation is where this plays in. As a reminder, we said that an algorithm was:

- Is Finite
- Is Definite
- Has Input
- Has Output
- Is Correct

Couldn't we change that a little bit and create a function, code block, program, (class,) or structure header:

```
/*
 * Expected Execution Time -
 * Overview -
 * Preconditions -
 * Postconditions -
```

```
 * Output -
 */
```

The point is that this provides any readers (yourself, later, included) the ability to realize what was intended. It should be apparent from the amount of space we've devoted to this qualitative topic in an otherwise quantitative text, that this is absolutely necessary.

To put it another way: nobody cares how much or how good your code is, if you can't explain it to them.

Generally, it's not that we aren't documenting our code in this book, but that the entirety of the text around the code is the explanation. But, it might look something like this:

```
/*
 * Function - find()
 * Expected Execution Time - O(n)
 * Overview - Searches for and returns the node for the first element of the
 * Preconditions - head is not NULL, x exists in the linked list.
 * Postconditions - Linked list is not changed.
 * Output - Returns NULL if head is NULL, if x is not in the linked list, if
 */
```

It doesn't have to formal or fancy. Concise is good, but otherwise, you'll get better at writing these as you go. But now, if I run it and go 'why is this taking so long?' I can see that you think it should run at O(n), but if it's running at O(2n), then there's obviously a bug. If you think that the linked list is not changed, but the element is missing, there's a problem (for example, find could mean 'point to where it is' like a toddler or it could mean 'go and get it and bring it to me so that it's not where it was anymore' like a dog).

Documentation is mandatory and part of algorithms and data structures.

Testing

These three things: proof by parts, modular design, and documentation form the core of our testing strategy.

Let's further break our ideology up into three stages.

Stage 1: Just Do It

Literally, if you write 5 functions, contrive a situation in which you need to run all 10 functions. That's literally as simple as:

```
void main (void)
{
```

```
    ll = arraytolinkedlist([0, 3, 6, 9, 12]);
    struct node* curr = find(head, 3);
    print_node (curr);
    curr = find_prev (head, 3);
    print_node (curr);
    freelinkedlist(head);
}
```

Good job! You just tested your code! You need to ask three questions:

1. Did it compile and run at all?
2. Did it run without crashing?
3. Did it do what you expected?

You obviously want a Yes to all three questions. But more importantly, where you get a No tells you where you're failing. Remember, what we're doing here is writing an algorithm, which isn't going to run on your own. So, we're going to intentionally write a program that just exhaustively tests all of the things we did.

This is a valid, albeit simplistic test.

Stage 2: I'm gonna wreck it!

After a bit, you'll start wanting to get better at writing tests. In stage 1, you just wanted to know whether the code worked. But in stage 2, we want to know if it doesn't work the way it's supposed to not work.

For example, suppose you have a function where the user is supposed to input their name. And the user types in 🦉🍄🦷. Now what? Your program probably doesn't handle that unless you expected it to happen. So, what happens? Does your program crash? Does it try to run and then cause a cascading failure of some sort? Does it try to input that into a database and completely wipe the database? These are all real concerns.

More likely, you actively don't want people to use emojis as their name (contrary to McKenzie's claims), so you fail that input. Does it fail correctly? Can the program compensate and move on, or does it stall out when it's not supposed to. Alternately, do you have it exit gracefully and file an appropriate error log if that's what's intended?

Here's an alternate example: video editing. Suppose a YouTuber records 10 hours of footage from their trail cam and writes a command that isolates certain sections, stitches them together, isolates keyframes, changes format, and adjusts/crops resolution. And they make a single typo. Does that youtuber want to know that the typo happened immediately with an error message or after several hours of processing and the production of a senseless, multi-terabyte output file? Ideally, you fail early and often in that sort of situation.

Point is, your second stage is to try bad inputs and make sure that the incorrect output is as expected.

This moves up through what are known as unit tests. The easiest framework for unit tests in C is to use the assert() function. If you have an algorithm packaged in a function and you know what the various outputs are supposed to be, you create a wide array of both correct and incorrect inputs and use assert() to define what you think the outputs are supposed to be. This can then be run every time changes to the code happen (for example, extra error handling, updating an imported package, adding documentation, or correcting a typo) and there shouldn't be any changes in the outcomes.

Otherwise, you're largely going to ask the same 3 questions as in stage 1, but probably break them down more discretely (for example, each separate set of incorrect inputs or a separate set of tests for each function). You'll find that you want more granularity later.

Stage 3: Outsourcing

Just use someone else's test framework. There is an entire field of study related to testing, and as much as it would be nice to cover it, there just won't be room. There are a number of testing methodologies available. They all revolve around the same idea that inputs should match outputs. But they provide more and more fine grained control over things like large problem domains, cascading function calls, or random values inside the functions.

You could get started with Check, Unity, or CMocka. But not until you've gone through the first two stages on a couple of projects. You need the conceptual framework of what and why for testing before you get the 'just do these things'.

Practical Applications

This chapter is about the basics. Whereas the first two chapters were very C heavy, this is the first chapter where C was just the tool. You should feel comfortable, at this point taking any of the code from this chapter and translating it into the language you are most familiar with (even if it's C++).

Side note, did you notice how much shorter and cleaner Psuedocode was than the C code? Did you also notice that it looked an awful lot like Python? There's a reason why Python is popular. After learning C programming, writing Python will feel like writing pseudocode. If you need a job now, learn Python now. If you need a job later, learn C now so that you better understand the Python you learn later.

Testing in Practice

Are unit tests great? Frankly, no. They're largely discounted by the broader industry. But, they're obvious. If someone says 'what's 2 + 3?' do you just answer 5 or do you count it out? So, when someone says 'what's 298 + 119?', if you answered 6 to the former and 12 to this one, they'd be right to distrust you.

That's sort of what unit tests compromise. You need to know how to set up a framework such that you're testing that each of your functions does exactly what intended it to do and that the outcomes stay consistent throughout. Your underlying algorithm shouldn't be changing, so why should your unit tests?

Then, once you're in the company, they'll introduce you to their specific testing methodology and how they want tests documented. Hopefully they do those things. "Ask ChatGPT" is an increasingly common answer, but it's not the same as testing and review. Regardless, there is only one wrong answer to interview and performance review questions. When they say 'Can you explain how you'd unit test this?', then 'I don't know' is the wrong answer. As long as you have done it before, you'll likely be able to put something together that's sufficient.

Side Note: It seems suspicious that there are proprietary ways to test code that are so incredibly special and unique that they're super effective and nobody has leaked their process. More than likely, it's the exact same style of code you've been doing, but the prompts for developing the tests are a little more automated and the visualizations are fancier and more analytics-driven.

Documentation in Practice

One of the most common tasks for programmers is 'this thing isn't broke but it doesn't do what we want, go fix it, don't bother telling me how'. Documentation in the code is your friend. This is the part where you use some of your testing skills (likely stage 1 and stage 2) and try to track down where the code is not doing what it's supposed to do. That may be a timing issue (it should run 2000 times a second, but it takes 2000 seconds), has side effects (it's super fast, but every time it runs everyone gets locked out of the database) or something similar.

The point is, reading the documentation where the expectations are laid out is helpful. Writing the documentation as you're tracking down the problem (because there's a strong chance the previous code did not) is even more so.

Linked Lists

You'll never see a linked list in the wild. We teach linked lists, because they form the basis for more complex data structures. But it's like unit testing, at an interview they won't care whether you know about linked lists. But if you can't perform simple tasks with a linked list, they'll presume you can't do more advanced tasks with more advanced constructs.

Further, every single principle we cover in a Linked List can be applied to the more advanced concept. Here's an example: Python doesn't have arrays (it does, but they're really out of place). However, it does have Lists, which are the malleable, resizable, infinitely editable data structures we want Lists to be.

Python is also written in C, under the hood. It's an actual example of why we need to learn C. Even if every

day Python programming is the norm, knowing the internals and minutiae is what separates an experienced (and more highly paid) programmer from a less experienced programmer. In the C code for python, Lists are an array of pointers. They've made a jugement call that they want the iterative benefit of an array (remember how fast the code for just an array was) but don't want to worry about whether they can allocate it all in a contiguous block.

Do you have an opinion on that? You will, at some point. But your opinion doesn't work unless you can compare the overhead of the linked list with the overhead of the array of pointers.

AI Examples

What makes a GPT work? This chapter has been data structure heavy, so let's jump into some algorithmic complexity, but we need to simplify it. GPTs work through a tool known as a GAN—Generative Adversarial Network. Basically, you have two neural networks (we're going to explain what a neural network is, don't worry, we just need a little more in the realm of data structures. Spoiler: What if you had more than one next pointer? Sketch it out before we get to it in the textbook and see how well it stacks up!) whose output is the input to each other. Let's start with an earlier problem:

Producers and Consumers

When we talk about data flow, we're typically talking about balancing. Think about one of those really old phone switchboards where the 1910s operator had to physically connect plugs to make a long distance call go through. Let's pretend they had enough bandwidth for an infinite number of phone calls. But, they still only have two hands. So, for every person who calls (a producer of phone calls), the operator (a consumer of phone calls, and a producer of connections) has to figure out which plugs to pull and which to put in to connect to the person who answers (a consumer of connections).

It takes exactly 1 step for the caller to make a call. They say 'I want to talk to X'. but it takes at least two steps for the operator to make the connection: they need to hear the request and then make the connection (this is assuming they don't have any processing time). This means that as long as there are more than half as many callers as operators in one unit of time, the operator won't be able to keep up.

This balancing act is the producer/consumer problem. And it's a huge issue in Network design. What happens when the wait time to go through with the phone call reaches more than an hour? People start hanging up. And since this is obviously an analogy for data traffic, we start losing data when the producers outnumber the consumers (technically the production rate outnumbers the consumer rate).

This is, however, the algorithm we want to look at: producer produces some data structure as output. The consumer takes the data structure as input and produces some other output. Could we bring this a little further:

```
define producer ()
     while true
          ret = some_process(global_consumer_variable)
          global_producer_variable = ret
          global_flag = true
          wait_until(global_flag == false)
     end while
end

define consumer ()
     while true
           ret = another_process(global_producer_variable)
           global_consumer_variable = ret
           global_flage = false
           wait_until(global_flag == true)
     end while
end
```

Notice the algorithmic pattern... it's not actually an algorithm! Which of the basic principles does it violate?

It's not discrete, because we don't know what some_process() or another_process are. It's not finite, because it just exists forever. There's no input, there's no output, so what's happening?

Back to GANs

Again, this is primarily an innovation for neural networks. But, we can utilize it in a number of different facets. Essentially, the producer creates something that the consumer has to respond to. The response then informs the producer how to proceed further. At it's most abstract, this is how you interact with Gary. Gary says something, and in your response, it hears a positive or a negative tone, which adjusts whether it refines or changes its tone.

A classic example of this is calibration. Imagine some robotic pincers trying to pick up an egg from a conveyor belt and put it into an egg carton. Every step has to be tested and evaluated. But, the conversation between two processes might look like this:

```
Producer: I picked up the egg!
Consumer: No you didn't, you crushed the egg.
Producer: [Using 90% force, ] I picked up the egg!
Consumer: No, you didn't, you crushed the egg.
```

```
Producer: [Using 81% force, ] I picked up the egg!
Consumer: No, you dropped the egg and it broke.
Producer: [Using 81.9% force, ] I picked up the egg!
Consumer: Great! You picked up the egg and it didn't break!
Producer: [Using 81.9% force, ] I picked up the egg!
Consumer: No, you crushed the egg.
Producer: [Using 81.81% force, ] I picked…
```

And so on, but the point is, the training will refine to further and further decimal points (bonus task. This conversation isn't an algorithm, but determining the new amount of force to use is an algorithm. Can you create the documentation and the algorithm for determining how much force the robot should use at any given point?). We get there through an adversarial relationship. Not enemies, adversaries.

More examples will get cumbersome, but we could also make this back and forth more complicated. Does the producer think it picked up an egg and know how big the egg is? Then the consumer tells it not just whether it picked it up, but whether it introduced a defect. And then the consumer starts to think it picked up an egg and how big it was and whether it was a good or bad egg. And then the consumer tells it what happened to the egg across the rest of the shipping process, etc.

Eventually, developing this set of feedback and response gets incredibly complex.

Neural Networks and GANs

So, a Neural Network does the exact same thing. Remember that, for all intents and purposes, the internal data representation in a neural network is magic. We can make lots of interesting observations about what is happening (which is more than we could do 20 years ago), but the why is still highly hypothetical and without consensus. The end result, though, is generally a classification of some sort. This is an excellent use of the consumer. We could give it a billion (literally a billion) pictures of a real giraffe and a hundred billion pictures of not a giraffe. We're skipping over a lot of the training information, but a modern neural network, if it's large enough, will eventually have a very high likelihood of being able to determine if there's a giraffe in those pictures.

Without getting into the details of how the internals of the neural network training works, we could further have it report how confident it is that there is or is not a giraffe in the picture. But the training happens through a set of external, known good classifiers of the presence of giraffes in pictures.

On the other hand, suppose we create a neural network whose only job is to throw a bunch of random bits at the wall and pretend it's a picture of a giraffe. What we'll get is literally random colored pixels, because it doesn't know what a picture of a giraffe should look like. So, we the humans could say 'that's not a giraffe, that's also not a giraffe, and that's not a giraffe, still not a giraffe'. But how many pictures could we get through as humans? So, we turn to the helper neural network we developed earlier and have it classify the outputs

of this picture-generating neural network. Millions, possibly billions, or possibly trillions of attempts later, through small decreases in the confidence of the first neural network that there is no giraffe in the produced picture, this picture-generating neural network will make pictures that can pass for giraffes.

We call the first neural network D and the second one G. D for Discriminator and G for Generator. Otherwise, it's exactly the same model as our egg-picking up robot example from earlier.

LLMs and GANs

LLMs like ChatGPT or Copilot or whatnot work through the same producer/consumer process, only the GAN is not really a part of the process. Because what do humans write that isn't something humans wrote (the question sounds weird, but what's a negative example of human written text)? We don't really have negative examples for a discriminator to evaluate. So, instead the generator just produces something that looks like what it is training over. The trick isn't simulating text, it's the tagging structure of the text that explains what the attention and topic within the semantic web is.

Ultimately, however, what we have found repeatedly, is that LLMs need to be finished with a special case of GAN: humans (RLHF – Reinforcement Learning with Human Feedback). It needs to have humans evaluate and then either reinforce or disincentivize certain kinds of outputs.

Applications

In the long term, a lot of parts of coding will probably get subsumed into the GAN model of development. In the short term, you can start looking at how different producer/consumer situations arise. An excellent exercise is to not just figure out what the deliverables are for both the producer and consumer, but how they process it, improve, what the goal is, etc. Write it down and explicitly note your conclusions. These are the sorts of notes that feed into good project ideas and verbalizations of problem solving process at interviews.

Algorithmize each step and determine how they are trying to come to their conclusion. Here's a set of random examples:

- ODBCII sensors and vehicle inspections
- Counterfeiters and Secret Service Agents
- Warez sites v. game developer anti-cheat software
- Netflix's suggested movie list
- Airplane landing negotiations
- Triage
- Volume in a crowded subway

Media Attributions

- arrto linked list

CHAPTER 4 - MOSTLY HARMLESS

This chapter is going to be covering three very standard topics. Remember that the basic rules for almost everything in this low-level theoretical framework is not that you're going to go right out and apply this to your next project. Further, you're not really going to want to. In general, we end up trying to outsource as much of our programming as we can.

This has pros and cons and is a whole separate conversation. The pros are: you don't have to maintain code you didn't right. There's some overhead in keeping your libraries up to date and being aware of the various bugs that might crop up, but that is typically less time than managing a codebase of several thousand lines. Consider, for example, how you might need to change your code every time a new windows update comes out or every time Congress (state or federal!) changes the rules on Daylight Savings Time. Even if you don't have to make any changes, you have to actively decide it doesn't apply when you're in charge of a codebase. So, frequently, it's a huge relief to just use an existing library.

The cons are a whole field of study. But we can sum it up in three possible scenarios. This is overly reductionist, but a good representation:

- Projects that get abandoned but don't formally deprecate, and slowly accrue more security vulnerabilities as bugs are identified but not patched and as technology advances (for example, imagine a system that requires a 6-character passcode, whose 56 billion combinations can be bruteforced with legacy GPUs in less than a half-second).
- You're extending the security footprint of your application to people outside of your organization. The maintainers and contributors to your program are probably internal, but the maintainers and contributors to imported libraries are uncontrolled and unknown. This means that the security of your codebase is only as secure as the security mindset of every library you import.
- There's an unknown factor, because the chain extends back: the libraries you import from also import from other libraries. But the growth rate of this include chain is potentially non-polynomial!

You'll still import a lot of libraries, the point of this section wasn't to dissuade the practice. It was to inform about the considerations that need to be made and then to note that almost everything included in this class has been done by someone who has the time and interest to optimize it perfectly as the point of their project rather than incidentally as the point of yours. It's sort of like talking to a production body shop: they have the tools and know how to build their own tools, but they buy the tools because it's more profitable to spend that time working on actual tools.

Why are you learning this stuff, then? Because it's standard, and it prepares you to apply the core principles

to novel situations. Here's a practical example: suppose you're working for a utility distribution company. Your current task is to build a program that accepts and displays pressure (or voltage or temperature, it's largely the same for this example) data from every sensor in the network. There's a catch, though, while most of the data is routine, you need to urgently prioritize data above or below certain thresholds for immediate human intervention and you need to prioritize the displays of data at a different threshold.

So far, it should be pretty easy: just grab a sorting library (or use the built-in one in your language of choice) and set it to go. The problem comes with scalability: suppose as you go from your test environment to the production environment and move from tens of sensors to tens of thousands of sensors, you realize that you're missing priority messages. Why? Possibly, because the sort application is general, it has a lot of overhead. Not only that, but you really don't need to sort, you need to categorize into ordinal categories, which is not something that most systems will automatically do. So, you have the choice of either tracking down a specific utility that does what you need fast enough to be serviceable or you need to write the utility yourself. But which algorithm should you use? The answer isn't in this book, but the ability to evaluate and compare the tools in this book is the principle we're chasing, and that you can extend.

Loops and Recursion

Before we continue, it's important to remember that loops and recursion are the same thing. One difficult thing to understand is the idea of mutability. You should be comfortable with the difference between a while loop and a for loop. If you were given a for loop and asked to convert it to a while loop, how long would that take? Vice versa? You should probably have a vague notion that it would take a fixed amount of time. That means that the 'differences' between a while loop and a for loop aren't real. Ultimately, they're the same thing with a different expression.

The same thing is true for any number of concepts in Computer Science. We're frequently not worried about how certain tasks are accomplished, because in the prototyping, initial, or high level frames, the details won't matter by the time they trickle down and somewhere in there, a lot of variation will get smoothed out.

Loops and Recursion are like that. A recursive function is just a loop. And, at its most basic level, a loop is just a recursive function that's written in an easier to consume manner. The difference is in utility, not effect. A recursive function is easier to prove (we'll get to provability in a later chapter, but we've already covered the idea of proof by parts, you might be able to intuit how that connection goes) in a formal structure, more often than not. But recursive functions in practice are not as controllable.

We can't consider algorithmic programming in C without pointing to NASA's Power of 10, a set of rules for how to write code that will still work 40 years later. One of those rules is to never use recursive programming. It technically says 'avoid', but that's because they wanted to make positive statements and the connotation should really focus on never using it.

Again, that doesn't preclude recursive code. Frequently, planning or research will involve recursion, and it's

only at the actual programming stage that the code will become iterative. Since a large part of programming happens before the actual programming, it's good to know how recursion works.

Recursion

We'll start at the beginning. Remember from math class, that we've got basic functions, such as:

```
f(x) = 2x
```

Side note: we could also call this a relation, and we're saying that it's just as invalid to say that f(x) derives from x as it is that x derives from f(x). In programming, we very specifically have a causal relation, because have input and output. But it's important to remember when that isn't strictly true from the real phenomenon being modeled. Statistics, for example, famously does not conflate correlation with causation, even if the programming behind it is using one value to find another value.

This equation is telling us that we can find an output value, f(x), by looking at the input value, x, and doubling it. That's straight forward, and we could write code that implements it:

```
float doubler (float x)
{
     return 2.0 * x;
}
```

We could put anything in there, right? We could put 'height of a stack of green beans', 'rate of honey drops dripped from a hive attacked by a bear', or 'number of times sneezed' and we would get a doubling back. If we can put anything in there, y, z, w, g... could we put f(x) in there?

```
F(x) = 2f(x)
```

The answer is, yes! We can! But don't! It's useless!

Mathematically, this function makes sense! We're saying 'make x hold true for the situation in which x is equal to twice x'. Which is true for 0 (Properly the set containing the value 0, or {0}). Even if we said

```
F(x) = x - 1
```

We're saying 'make x hold true for the situation in which x is equal to one less than x', which is true for nothing, or the empty set, {} sometimes called ∅ (literally called 'empty set'). The point is, we're trying to get into the mathematical conception that when we write a function, we're making a truth statement about a formal relationship between numbers. The statement is true for some set of values, even if that set of values is no

values. Incidentally, this is also how functional programming works, which we won't go into in this text, but it is an alternative to Object Oriented Programming.

Coming back to imperative programming languages, the dilemma should be apparent: when we wrote the function doubler(), we weren't saying 'there is a relationship, please give me all the true values', we were giving the command (hence imperative) to do a thing. Double that number. In English, it's literally called an imperative sentence. So, if we can put 'anything' in there then we can:

```
float infinitedoubler(float x)
{
     return 2.0 * infinitedoubler(x);
}
```

Feel free to try this. You'll get an error that it doesn't know what infinitedoubler() is, and you just need to add the function definition to the top of your .c file. Great! But, what happens when it runs. We say it's in an infinite loop. But is it really? The memory stack would look something like:

```
Main(…, infinitedoubler(x0, infinitedoubler(x1, infinitedoubler(x2, … infinit
```

Where n is the amount of memory that you have in your computer.

Loops

If you're keeping track, we've implied three scenarios:

1. Mathematical Relations that are intrinsically true.
2. Imperative statements that solve for a single value of the mathematical relation.
3. Recursive loops attempting to simulate the mathematical relation or fail due to no solution being found.

We could say that the art of recursion is trying to simulate scenario 1 by using scenario 2 without running into the error of scenario 3. How do we do that?

Recall the parts of a loop again:

1. Initialization
2. Condition
3. Iteration
4. Incrementation

For example:

```
for (1; 2; 4)
{
      3;
}
```

Why do we have those four things? Let's explore a little deeper.

1. We need to initialize, because in math, there is unbounded time. That is, the relation has always been true and will always be true and even that statement implies more time sequence than exists. But in programming, we need to know what the situation is now before we can move onto instruction now + 1.
2. We need to know when to stop. There has to be a stop point, because mathematicians can count to infinity (how does a mathematician count to infinity? "1, 2, skip a few, infinity." How does a programmer count to infinity? "for (int i = 0; i; i++);") but we can't, since we're bounded by electricity, processing power, and the heat death of the universe.
3. We need to do something. In math, the entirety of the relationship always is all at once (i.e., no part of a line comes before another part), but since we're stepping through the sequence of steps that need to occur to make the change occur, something has to be done. Otherwise it's like a motor spinning without the clutch engaged.
4. We need to change the portion of the relationship we're looking at every cycle or we're going to be solving the same problem over and over again.

Once again, let's take a look at how our infinitedoubler() stacks up against this framing:

1. What's the initial situation? We don't have one. We could be starting at -INFINITY (from math.h) or INT_MIN (from limits.h) or we could be starting from 0, or from 100,000,000,000. Not only do we not know where we're starting, but we don't know which parts are important.
2. Where do we stop? Every function call of infinitedoubler() will, in turn, call another infinitedoubler(), ad nauseam. There's no stopping point, which is obviously a problematic (refer to the previously mentioned heat death).
3. What are we doing? This one at least has some good news! We're doubling the value and sending it back. So, if we ever finish our infinite loop, we'll at least be able to return all of those doubled values.

    - Side note: this is part of what NASA's Power of 10 warns about: if we're doubling values, there's only about 64 different inputs we can give (0-63) before we go beyond the limits of double and overflow the value. So, we're limited by our data space constraints, on top of the previous ones.

4. What changes between cycles? This one is the most compelling issue. It's where this analogy sort of breaks down. But the example needed to show as many failure points as possible. So, in this case, the

reason there's an infinite loop is because even if we solve points 1 and 2, we still never change which problem we're attempting to solve.

Let's go ahead and address each of these issues in turn. We're going to be changing the problem to do that, because the actual solution to the problem written is:

```
//Presume x must be positive.
float doubler(float x)
{
     if (x == 0.0)
          return 0.0;
     return -1;
}
```

There's we can just pre-calculate the answer, so we bother iterating it in this specific case. Technically, there's not really a programmatic answer, because no fail-state answer makes sense. Regardless, we're going to change the context of the question to:

```
{
     x ∈ Z | f(x_n) = 2 * f(x_n-1)
     f(0) = 1
}
```

Essentially, we're adding the time component to the math by introducing that n variable. We're essentially saying 'a function of x is the value that is twice the function of the previous value of x, and also we're only doing integers, and also f(0) is 1!'

Let's look at why we made these changes:

1. Input or Argument – We actually just start with whatever the argument to the recursive function call is, in this case it's basically n.
2. Base Case – We need a stopping point. Anything will do, as long as we can prove that every possible input will eventually converge to at least one base case. These are typically not calculated but assigned answers. In this case, we're going to set f(0) to 1 (why? Because if it was 0, every answer would be 0).
3. Recursion – The actual function body is, itself the iteration. When we want to move to the next iteration, we recurse in some fashion. We aren't at this conceptual point yet, but one thing to keep in mind is that we can have several recursive calls.
4. Update – We need to update the value somehow. In this case, it's the n-1: we're saying 'let's only evaluate this if we get closer to our base case'. We can describe a situation where some situations don't get

closer to the base case as 'nonconvergent'. Nonconvergent functions are not really solvable in math or programming (alternately, the answer is literally the special case of 'nonconvergent'. Infinitedoubler() was nonconvergent over time, which is also why it was an infinite loop.

1. "What about negative numbers?!" you're asking, we'll get there.

So, let's try to update the code, once last time to compensate:

```
// Precondition: x is greater than 0
int rdoubler(int x) // 1
{
     if (x == 0)
           return 1; // 2
     return 2 * // 3
     rdoubler(x - 1); // 4
}
```

So, breaking it down:

1. We changed the argument type to be integer. The reason for this is because otherwise we have ranges. There is a very simple solution for still using floats. You should try to modify this code to make it work for floats. Think inequalities instead of equalities.
2. This is our base case, f(0) = 1.
3. Sorry about breaking the line, but we can see that the 2* is the doubling portion, so it's the actual point of this recursion. We call this part the unwinding phase. (Resolution is a prettier word, but it specifically refers to logical programming not recursion. Backtracking is a recursion term, but it specifically refers to also undoing some of the changes, which will be covered in a later chapter).
4. Along with points 1, 2, and 3, is the most important part (see what we did there?). The input has to change. For the next cycle of the recursion.

So, let's trace it:

```
Start
rdoubler(4)
x != 0, rdoubler (3)
x != 0, rdoubler (2)
x != 0, rdoubler (1)
x != 0, rdoubler (0)
x == 0, return 1
```

```
return 2 * 1
return 2 * 2
return 2 * 4
return 2 * 8
End
```

Dr. Proofingston, I Presume

Now, notice something interesting. Suppose we take the line x == 0, return 1 and call it the middle. In this example, we can look one above and one below the middle, and we see that is independently the answer of rdoubler(1). Same thing, two above and two below gives us rdoubler(2). Those specifics are for this algorithm, but it holds true that every single recursion call is possibly part of a greater problem but can also be treated as a completely independent problem in its own right. That is, the answer to rdoubler(3) does not necessarily have to have any relation to the answer to rdoubler (4). In this case, it does, but that's incidental, not necessary.

The important part is that because each call is inherently separate and only incidentally part of the greater whole, if we can prove the validity of one of those parts, we can use that proof to prove by parts the whole recursive process.

In this case, the proof looks like this:

```
F(0) is 1, given.
```

That's literally the start of the proof. We know that something is true because it was stated as such, presumptively as part of the constraints of the problem.

```
f(1) is the same as 2 * f(0).
f(0) is 1, therefore f(1) is 2 * 1.
∴ f(1) is 2
```

Easy and to the point. We could, in fact make this point for any given value, although it might take a long time. We can write a proof for any value, but what we really want is a proof for the entire thing.

```
f(n) is the same as 2 * f(n - 1).
For all positive n, n - 1 will eventually approach 0, axiomatically
∴ the recursive call will always terminate
```

That function is a bit weak. But the problem is that we haven't established what we want to prove. Remember that we can do anything we want as programmers. But just doing a thing doesn't mean that it warrants more than just stating that it was done. What if, however, we want to give an external metric. Like, for example,

suppose we noticed that pattern of 1, 2, 4, 8, 16, etc. Could we prove that our rdoubler() is really the same as 2^n?

$f(n) = 2*f(n-1)$ and $2^n = 2*2^{n-1}$

$2 * f(n-1) = 2 * 2^{n-1}$

$f(n-1) = 2^{n-1}$

$f(0) = 1$ and $2^0 = 1$

$\therefore F(n) = 2^n$

Incidentally, $\therefore$ is the sign that reads 'Therefore' and is sort of like the mic drop at the end of a rap battle. Only with logic, and Logic proofs are usually less dramatic than rap battles.

We won't spend a lot of time on proofs, as it falls much more in the rarefied academic reaches. But, it's important that you know that the proofs exist. It's important that you can read them, even if it takes you a couple days to work through a basic proof. And it's important that you recognize that proofs for all of the most common algorithms are easily available (particularly on Wikipedia, ChatGPT, or algorithms texts)

Search

Let's address the catch-22 in the room. We can't really learn the sort algorithms until we know why we need to sort an array. The answer is for searching purposes. But we can't learn good search algorithms until we know how to sort an array. So we're going to split the difference and learn one kind of search algorithm to start with. It's the easiest one.

But first, what distinguishes an array? We can probably come up with a ton of metrics, but for any real utility, we want to know exactly one thing: is the array sorted or not. That can actually be a more complicated question than we want to get into right away, and confirming that an array is sorted is its own algorithm. But, for now let's stick with the answer 'no'.

When we say an array is unsorted, the general idea is that it is in some random configuration. Like pulling random cards from a deck and writing them down as they're drawn. And that's a good way to think of unsorted: random order. But, by a technical definition, unsorted just means that we can't guarantee that the array is in any sort of order. So, for example, it could be, through random chance, completely in numeric order. But because we don't know that ahead of time, we can't rely on it.

One common task (in fact, for many subfields of programmers the only real task) is to search an array for a specific element, and return something about that element (whether it's index, a pointer to the element, some other value for the element such as in a Key Value Pair,... etc). And, further, there's only really one way to search across an unordered array: linear search.

Side Note: There's actually lots of ways to search through an unsorted array. For example, you could just not, and always return 'element not found'. You could pick random places and guess it's there. You could only

search the first couple elements because if it's not in the first couple, you didn't want it anyways. Those are all actual and niche applications. But, when we say something general like 'there's only one way', what we're really trying to say is 'there's only one good way for the general case of an unordered array search'. People will always come up with novel approaches, but the science of algorithm design is to prove that it's better.

Iterative

The linear search is quite simple, start with the first element in the array and move to the next if that's not it.

```
// Precondition: Given a non-empty array, that's not guaranteed to be sorted
// Postcondition: Array is the same as it was given.
// Objective: Return an index of the element of the array that matches the se

int linearsearch (char* arr, int n, char target) // 1
{
     if (arr == NULL) //2
          return -1;
     if (n <= 0)
          return -1; // 3
     for (int i = 0; i < n; i++)
     {
          if (arr[i] == target)
               return i; // 4
     }

     return -1;
}
```

Pseudocode

```
Define linearsearch (arr, target)
     for i in from 0 to length - 1 // 5
          if arr[i] == target
               return i
          end if
     end for
     return ElementNotFoundError
```

```
End
```

As always, let's run through some notes:

1. Again, with the standard array pointer, length value, and target. We'll keep pointing this out occasionally, because you will forget to do it at some point, and it will be frustrating. Resolution will occur sooner the more you notice the pattern. Also, notice that the target is the same type as the elements we're looking for, and that the return value is an int since it refers to an index value. This is not set in stone, and is specific to this particular implementation.
2. We, theoretically, don't need to check for NULL here. The precondition specifically says that the array exists and demands a non-zero length. But. Just because the precondition is there doesn't meant that people aren't making mistakes or that there isn't an error or malicious user, etc. Since the stakes are a full crash in the program to avoid a single instruction cycle comparison, it's probably worth making that comparison. One might even suggest a moral obligation to harden the code in all of the ways feasible to prevent crashing.

    - return -1 for an error is a terrible idea. But, like many things, there aren't a lot of better options. In this case, we're purposefully sending a value that doesn't make sense (how can the value be found before the array even begins?) in order to let the caller know it should check. This is reinforced by the function description that states as much. In essence, we're doing what we can to highlight an issue, and if the programmer who calls this function doesn't list, that's on them not us.
    - Most high level languages have a better reporting system. For example, in the pseudocode, we suggest raising an error code. It's literally just a mask for probably -1 or -322 or something, but the point is that there's a big list somewhere that can be looked at to see what's going on. Follow language convention, but it will change subtly or significantly by language.

3. This is a design decision. Technically, we're checking for two different things. We can have an array of length 0. It's just an empty array. But we don't even need to continue, if that's the case, we'll just return a negative value to indicate there was an issue. Again, in higher level languages, better to use throw a flag of some sort. Moreso, though, we're also checking for invalidate lengths: we can't have a negative length. Since they're so similar and the return is the same, we just combined them. But that is not always the best decision. Sometimes (frequently) it's clearer to delineate each case separately. So, in this case there would be two if statements here, and that's a little less concise but better to be explicit than clever most of the time. We combined cases here to illustrate that sometimes it is OK on minor and obvious issues, and to be able to highlight this note.
4. This is an important part: remember the return type. If the function needs to return the value, return the value. If the index, then the index. Maybe it needs to return a pointer to the value. That should be your first and last review check, because it's easy to gloss over what you assumed to be right the first time.

5. Notice that in the pseudocode we're frequently ignoring gross violations of practical application. We don't worry about the index or the null pointers or anything, because the pseudocode should be applicable to a number of languages. Further, since it's arbitrary, we can just declare that the preconditions are true, not just should be true.

Linked List

Since we've covered linked lists, let's write the same function, but over a linked list. Note, this code will literally do the same thing, we just presume there's a linked list there instead of an array. Consider the changes, or try to write your own function before you read this one.

```
// Precondition: Given a linked list, that's not guaranteed to be sorted and
// Postcondition: linked list is the same as it was given.
// Objective: Return a pointer to the element of the array that matches the s
struct node* linearsearch (struct node* head, int target) // 1
{
     while (head != NULL) // 2
     {
          if (head->payload == target) // 3
               return head; // 4
          head = head -> next; // 5
     }
     return head; // 6
}
```

Informal pseudocode

```
define linearsearch (head, target) // 7
     while head != target
          head->next
     end while
     return head
end
```

Code walkthrough:

1. Notice the important part: target is still the same, but we're simply passing a different data structure. The

data inside the data structure is still the same due to arraytolinkedlist().

2. This will suffice for our checking that we aren't going to get a segmentation fault error. A comment noting not to change the value of head until the end of while might be OK. It's presumed, but the thing about concepts that don't need to be said is that sometimes it's just easier if they're said. Within reason, because comments on obvious ones like this case would be easy to create clutter more than clarity.
3. Notice also that we're comparing the int to the int, rather than the node itself to anything. The node is just a wrapper for the data inside of it. This comparison might be anything, not just equality. But for most examples, we stick with simple equality.
4. Again, notice that the return type is the actual pointer to the node that holds the target value. As a side point: this seems pointless. Just remember that the payload type being an int is a strong simplification. It's frequently some other data type, like the stats of a monster, the details of a record in a database, or raw packet header data. So, the 'search' might not just be 'does it equal', it might be 'are there any packets with a header field that says Urgent?'
5. Probably shouldn't cycle across head. Again, a single pointer is cheap, so using one that's appropriately named and consistent across functions like this is a good idea. But, there's technically not an issue with reusing the passed-by-value pointer (the data is passed by reference, but the reference is passed by value. This confusing sentence is why you can but shouldn't use head in this case).
6. Why are we returning head again? Well, if we didn't find anything in the while loop, and the while loop exited, there's only one reason for the while loop to exit. If head is NULL. We have to return NULL anyways, so we can just return head. Is this a good idea? It's complicated. Some people believe that no hardcoded values should be in your code, so having 'NULL' is a bad idea and they avoid it, so implicitly being able to skip that is fine here. But, returning NULL with a // comment saying 'value not found' is clearer suggesting that the 'never use a hardcoded value' rule is potentially a little too strong.

    - We could, in fact, even replace the // 4 line with just 'break'. This means we found the value and can immediately return. Since we know what's outside of the while loop (solely a return statement) we can just jump there. What's wrong with this approach?
        - First, it's hard to read without knowing what's there. Try changing like // 4 to just 'break;' and come back to it in a day or two. Dollars to donuts, it takes you more than a few seconds to figure out what the code does.
        - The second reason (and one that's less subjective) is that we're applying external semantic knowledge of the code. That is, the processor doesn't know what's going to happen. The algorithm is what we call a stateless machine: it doesn't know what came before or comes after. But we, as humans, do. So we're kind of cheating. It's like when you play Dark Souls and you know that a big bad boss always starts with a particular attack, so if you start your dodge roll before you even go into the room, you get a free dodge. Sure, you can, but it's applying outside knowledge to an internal situation. One way to see the effect of this is to consider the

idea that maybe you change some of the code: can you guarantee your external knowledge of the fact that 'return head;' comes immediately after the while loop still holds? Maybe someone comes in and adds printf("Element Not Found."); and now your program will sasy 'Element Not Found' even though it acts like it has been found.

- This pseudocode is kind of horrible. But, otherwise, it was very similar to the previous. So, we just wanted to note that you can just jot down notes, you don't need to have it be exact.

Recursive

Lastly, let's try this same function, but recursively. Note, Linear Search is probably the worst thing to use recursion on (anything called 'Linear' probably falls under that category). But one of the goals in covereing Recursion is that it is possible to map between iterative and recursive loops and vice versa.

```
int rlinearsearch (char* arr, int n, char target)
{
     if (arr == NULL) // 1
          return -1;
     if (n <= 0) // 2
          return -1;

     if (*arr == target) // 3
          return 0; // 4
     else
          return 1 + rlinearsearch (arr + 1, n - 1, target); // 5
} // 6

struct node* rlinearsearch (struct node* head, char target)
{
     if (head == NULL)
          return NULL;
     if (head-> payload == target)
          return head;
     else
          return rlinearsearch (head->next, target) // 7
}
```

Recursive review:

1. Always check for NULL. If you think the thought NULL and you aren't currently writing a check for NULL, it's probably a good idea to check if there are any NULLs you can check for that might cause a crash.
2. Same as previously, except in this case this also acts as our base case. We're actually expecting n to become 0 at some point. So, we're actively checking for it. This is a common pattern, and why we tend to see the recursive update be something like n – 1, because lengths and indices are integers and it makes algorithms cleaner to execute and prove over integers than floats.
3. In this case, we only ever care about the first element of the array. Since *arr is the exact same as arr[0], we may as well be transparent about it. On the flip side, when you get some expertise in timing, maybe it's worth it for you to do some testing. Would it be faster to recurse over groups of 2 or 5 or 8 elements instead of singles?
4. Suppose we find that the first index is the value we're looking for. What index are we looking at? It's still just 0, because remember that every recursion is its own problem. Even though it's silly in this case, we're still passing a new array (that is minus it's first element) to every new cycle, so the correct answer is the one that's correct for this level of recursion, not overall. We'll adjust for it in the actual recursion call.
5. This is the actual recursion call. Notice that if, in the global scope, the value was in the second element of the array, the second call will return 0 but in this first call we'll return 1 plus that value, so we'll get 'index 1' as correct. There's a secret to figuring out how to get the right compensation in the unwinding phase here: calculate close and then fiddle until you get the right answer that always works.
6. Notice that we don't end with a return. What the compiler is normally looking for is all 'branches' of a function. Between the if and the else, as long as both have a return, it's fine. Mostly, we'll try to have a top level return statement to evidence. But it's different here to demonstrate that it's not necessary. As always, it's a balancing act. In a simple, non-nested if statement, it's probably fine and clear to just have an if-return and an else-return. More than that, and it will likely be clearer to just write a final return as the last line of the function to be sure the 'if all else fails' scenario is covered.
7. Notice that the linear search for the linked list is exceptionally stupid. This unwinding phase is literally 'return what the next one says'. Unwinding by directly returning the return of the next recursion is a good indication (among others) that there is little gain in recursion along this track.

The point of this section is to just demonstrate that regardless of whether any other example is iterative or recursively looped, they can be swapped either way. This is not a skill to practice, just a piece of knowledge to have available so that when it's important, you can problem-solve until you get the right answer.

Constraints

There are some problems with this search.

The biggest one is that it doesn't tell us much about the array. In many ways, this behaves almost like a

Boolean check for whether the value exists. Sometimes, we actually want to know all of the instances. How would you change the code such that it returns all of the values? We like this 'return the first version', though, because it helps us to illustrate the time complexity of functions, which we'll return to in chapter 9.

What's the fastest that this function will run? Literally one iteration, when the target is in the first position, right? What's the slowest that the function will run? Exactly n iterations, when it can't find the target. How about the average run time? Everything else being equal, For a uniformly distributed target, we would expect the average search length to be approximately

$$n/2$$

(although in more complicated algorithms, this simple calculation isn't going to work). These three are the questions that we want to ask about almost all algorithms. One of the only currencies we have as programmers is 'how long does it take to run', which means we'll normally prefer the faster algorithm. That being the case, what constitutes 'faster'?

Suppose you've got two runners. One is super fast when they get a full night sleep, but really slow when they don't. And the other is very consistent and steady every time. Suppose they book a recurring time at a local track to have a race every weekend. How long should they book the track for? Obviously, they need to compensate for the slowest runner's time, not the average or the fastest. In the same way, an algorithm is generally judged on how long It takes in its worst case, because that's how much time we need to allocate to guarantee it has time to run.

What's the constraint we're fighting against? Typically some repetition like the work day (any task that takes more processing time than a workday is a significant investment of time and resources), downtime ('this site is temporarily unavailable due to planned maintenance' every Wednesday at 2am means no sales for an hour or two), or frame (ever wonder why NES games slowed down? Because their algorithm couldn't handle everything on the screen in 1/24th of a second).

Further, the standard algorithm is typically pretty simplistic. Here's an alternative that is the same algorithm, just permuted a bit:

```
void main (void)
{
     struct node* enemies = NULL, players = NULL, items = NULL, blocks = NULL
     constructLists (&enemies, &players, &items, &blocks); // Pointers by ref
     struct node* target = GetMobileTarget();
     prankMobiles(target, enemies, players);
}

struct node* prankMobiles(struct node* target, struct node* enemies, struct n
{
     struct node* retList = NULL;
     struct node* curr = enemies;
```

```
	while (curr != NULL)
	{
		if (nodeEquals(curr, target))
		{
			curr->nextCurrSearch = retList;
			retList = curr;
		}
		curr = curr->nextEnemy;
	}

	curr = players;
	while (curr != NULL)
	{
		if (nodeEquals(curr, target))
		{
			curr->nextCurrSearch = retList;
			retList = curr;
		}
		curr = curr->nextPlayer;
	}
	return retList;
}
```

The code is needlessly complicated for example code. But, in context of a game, this would absolutely be the sort of thing that might happen in, say, Minecraft, if you used the command /kill @e[name=z].

Wrap up

This is the point: searching a list is almost always an exhaustive effort for two reasons. 1) based on our constraints we might literally need to check every entity in the list or array to make sure we got everything. 2) Since we don't know if it's the first or the last element, even in the best case scenarios we have to plan for the amount of time it would take to search every entry.

Ever wonder why some games have arbitrary limits on stuff like how many buildings you can build? It's for, essentially, this reason.

Suppose we return to the math: f(x) = x. This is the standard linear relation. Like, we could say Reward = Effort. That's a nice thought, that for every unit of effort we put in, we'll get an effort of reward out. This

linear search is considered to be O(n), which essentially means the same thing: for every extra entry in the list or array, we'll have the same fixed amount of effort to search the array.

Sorts

This all brings us back to the original catch-22. Suppose we wanted to search an array. And we knew that all the values in the array were 0-100. Suppose you're searching for any values that are above 95. Wouldn't it be nice if we could just start at the end and stop when we got to the first value below 95? Or if we wanted to show the top five race scores, do we want to search every entry or check just the first five entries?

The point is that there is a special case of array. The general array is unsorted (remember that's not because it's definitely unsorted but because we don't have any assurances that it's in order). But the special case is the same array, with the caveat that every entry is monotonically increasing or decreasing. Most examples are going to be alphabetical or in numerical order. But, the method of sorting doesn't matter, it's whatever makes sense to the application.

That being said, how do we guarantee that an array is sorted? We sort it!

A later chapter will discuss the various benefits of searching in a sorted array (and other structures) as well as how to maintain a sorted array. But for now, we need to take a look at the basic ways there are to sort an array. In essence, we're adding and enforcing a constraint that every part of the array will always be in order.

There are 4 basic sorts that we'll consider. These are considered the very first sorts (fun fact: the 'basic' sorts have changed over time, but we almost always teach four). These are not ever going to be used for a very specific reason: they're not very good! There are so many different sorts that work better for specific applications. However, these sorts illustrate the four basic ideas of what a sort will do, and some of the early algorithm design techniques.

To reiterate, you need to know these sorts in order for more advanced sorts to make sense. And we stop with these four because they're sufficient to cover a wide array of situations. These are sorts that will apply to 'any' situation, but in practice we want to customize the algorithm to our specific purpose.

Side Note: for a wide array of consistent visualizations on sort methods, Timo Bingmann's youtube channel is fantastic!

Bubble Sort

Bubble Sort sorts in $O(n^2)$ time.

The short form of this is that if we iterate over an array and compare every element to the one above it, we can swap when they're out of order and eventually the largest element will 'bubble' to the top. If we repeat that again for every element of the array, we will have a sorted array.

```
// Precondition: given an array and a length
// Postcondition: the array is sorted, but the content is otherwise unchanged
// Objective: Sort the array using Bubble Sort
void slowbubblesort (int* arr, int n) // 1
{
     for (int i = 0; i < n; i++) // 2
           for (int j = 0; j < (n - 1); j++) // 3
                if (arr[j] > arr[j+1]) // 4
                     swap(&arr[j], &arr[j+1]); // 5
}
```

You should give yourself a random array of int and trace this manually. Does it work?

1. Notice that we don't need to return anything, again because the arrays are being passed by pointers, it's fine. Could we pass it by value? Yes, but it's actually more difficult to manage in C, and all you gain is a slow down. Better to remember that in C, you're dealing with pointers and 'live' data as often as not and to be careful.
2. This is called the outer loop. You can think of it like 'how many values of the array have bubbled to the top and in the right order?'
3. This is called the inner loop. Inner and outer loop are standard terms, but people have studied and talked about these early sorts so much that they tend to get a little bit of a proper name, like 'Bubble Sort Inner Loop'. Notice what we're doing here is looking at each element of the array, for each element of the array we want to bubble to the top. Also notice that because our code is going to look at j+1, we need to make sure that we're not going to get an array out of bounds error. There are more robust solutions to this in other languages, but C tends to rely on you, the programmer, knowing that you have to algorithmically avoid exceeding array bounds because everything that's in C takes a lot of overhead.
4. This is the 'bubble' part, we're effectively building a sorted array out of a ~n^2 number of small swaps.
5. swap() deserves it's own subsection.

Swap, Constraints, Impossibilities, and Tradeoffs

Swap is, almost axiomatically, never faster than three operations. The goal of swap is literally to take two values and switch their positions. This is almost an atomic operation for many areas, but it's especially important in sort functions, because we are typically interested in moving things around rather than trying to recreate the array.

Swap needs an intermediary. So, if you swap two values, you can never do it faster than 3 steps, and with 3 values.

```
void swap (int* a, int* b)
{
      int temp = *a;
      *a = *b;
      *b = temp;
}
```

Done! Now temp disappears with the end of the function and a and b are remain because they were passed by reference. But, this is important to note: this seems obvious, but why do we have to use three steps? We can explain what is happening, you need a third value as a temporary store while you move the others around. But, that's not a causative reason.

In short, there's no proof that we need to always have 3 steps to swap two values (technically, it's n + 1 steps and n + 1 values to swap n values). At any moment, someone could come up with some algorithm that does it in less. But doing so would required contravening almost every fundamental presumption of information theory, computer science, and algorithms.

This is an important point: we can't always prove that some things are true. For example, NP-Complete is a class of problem in computer science . We tend to assume that there is a limit, because the reality is nobody has found a way past it. But, we've also never proven that you can't go past it.

This doesn't really affect day-to-day programming or design. However, it is a fundamental issue in Algorithms that's worth noting.

Back to swap, specifically: being able to write 4 lines of code that swap two values on the fly is a basic, presumption of things. There's a ton of validation and error handling we could add. For example:

```
void swap (int* a, int* b)
{
      if (a == NULL) return;
      if (b == NULL) return;
      if (a == b) return;

      int temp = a;
      a = b;
      b = temp;
}
```

If these were necessary, it's totally OK to add them. But consider the question: what would it hurt to swap null values? Or swap values that are the same? Do we even care? Count the operations. In the best case, creating temp is an operation, assigning it a value is an operation, assigning a a value is an operation, and assigning b a value is an operation. That's 4 operations. Checking whether a is null is an operation, b is null is an

operation, a is b is an operation. So for a 4-operation function, we just added 3 extra operations to check and 50% overhead is a huge amount.

That's not to stop you! If you absolutely need to avoid it, adding these checks would be mandatory! But, in most cases, it's really unnecessary. What's an example of necessary?

```
void giantswap (struct gigastruct a, struct gigastruct b)
{
     if (a.isEmpty == true) return;
     if (b.isEmpty == true) return;
     if (gigastructEquals (a, b)) return;

     struct gigaStruct temp = gigastructCopy(a);
     a = gigastructCopy(b);
     a = gigastructCopy(temp);
}
```

This is a stupid function. It has so many issues. But we're kind of pretending a little bit. Maybe gigastruct is for giant video files or billion-entry databases or something, and because of the size there's some intrinsic hardware reason why pointers won't work on them or maybe they represent physical object representations or who knows. But the point is that we don't know how much time gigastructCopy() takes. Maybe it takes an hour to process in real time. In that case, having a couple of tests ahead of time is reasonable because 3 checks is perfectly reasonable compared with the trillions of possibly wasted cycles trying to copy things that don't need to be copied.

But, there's a lot of contrivance there. In this case, it would almost always be better to redesign things from the ground up so that you can use pointers and make basic functions like this more streamlined. But, possibly, for other basic functions you will run into situations where you need to ask the questions about whether the tradeoff is worth it.

In short, swap is a good example of practical limits on our theory, and you need to consider tradeoffs at every stage. Doing things the 'normal' way is typically the right idea because things becomes standard through repeated optimization, but when things go wrong you need to be able to revisit your assumptions and decide where to make tradeoffs.

Practical Applications

There aren't a lot of practical applications in this chapter. Why do we study it, then? It bears repeating, because knowing the theory and basics is what will provide you with the knowledge you need to apply more specific solutions to a given problem. We try to curate these basic topics because they are a little more

translucent than others. For example, trying to start with understanding bucket sort and when it's specific uses make it effective (or why it might be better to use something that's easier to communicate) would be difficult. Or, figuring out how Python does its underlying sort methods through Timsort without knowing how the different parts work is a little silly.

So, what's the point of Bubble Sort, for example? First, a four-line algorithm is super easy to remember. And we call it brute force or naïve, because it's easy to figure out on the fly. So, supposing that you just need something, anything, and you're limited to a low-level language without inherent sort support, being able to pull out Bubble Sort is a good option. You know it works, even if it's not fast.

Further, people will often test you on these basic concepts, because they're easy. It falls under the category of 'If they aren't familiar with the easy stuff, why should I trust them with the hard stuff?'

So, to a certain extent it's important to have a lot of this available in your brain so that when someone asks, you can jump at it. Other things like LeetCode fall under the category of only measuring how well you do at LeetCode (in fact, some data suggests that LeetCode tests for employment only measure whether you've seen that particular LeetCode problem before). Still, when you have a technical field and are being hired by non-technical hiring managers, they need some metric to use to distinguish between applicants.

Your takeaway is: don't worry about trying to know everything. Instead, just like the practical use of the sorts, try to cater your at-will knowledge pool to the job or jobs you're going to be applying for. Find out what their application process is like, ask some employees what sorts of skills they use, read through some of their blogs (if it's available), learn what languages, tools, and platforms they routinely develop for.

At the end of the day, companies are not going to hire you because of your all around skill at programming. They're going to hire you because you can demonstrate that you are capable of solving the very specific problems that they want help with.

AI Examples

AI is actually one of the very few areas in which a lot of the theory in this text comes into practice. Earlier, we suggested you pick a time period and pretend like the numbers of what makes something computationally difficult applied there, right? Well, AI has sort of always been just out of reach. Rather, we can see that if we just had a little more computing power, we could make slightly better AI systems, and that foresight has outpaced our actual technical development for 60 years.

One way that we have always gotten around practical limitations is to optimize at lower and lower levels of production. It takes a very specific skillset, and low-level programmers (sometimes referred to as Systems Programmers) who have demonstrated that they can optimize things at the lowest levels of the computers digital mechanisms are in high demand.

What's an example? Suppose you have an LLM that is going to operate over 1 trillion factors, with a training dataset in the exabyte range. This is something that may take months to run all the computations on to train

the very basic first steps of your LLM. Literally the thing holding back companies like OpenAI or Google from releasing better and better LLMs is getting enough electricity to shove through enough GPUs to run their training algorithms.

So, someone being able to come in and do low level optimizations like, say, make a function that takes 25 steps only take 24 steps is a huge deal. If that one function gets run in every cycle, suddenly we're talking about being able to drop literal days from the total time it requires to run the training set. Even though the process might be 3 months of computation and the days seem like they are a low percentage of the whole, each day still costs a fixed amount of money to pay for electricity and processing time.

Array Max

In the next chapter, the first thing we're going to do is demonstrate some of those optimizations to our (purposefully) unoptimized BubbleSort. But let's give a specific example:

```
// Precondition: Given two integers
// Postcondition: returns a bool if a is bigger than b
// Objective: Determine which of two inputs is larger than
bool max (float a, float b)
{
     return a > b;
}
```

This seems like a stupid function. But, frequently we want to abstract the implementation out from the specifics. So, for example:

```
bool max (struct node* a, struct node* b)
{
     return a->payload > b->payload;
}
```

No big deal, right? But, this means that our algorithm can always call max(*a, *b) and never worry about what the internals are, and what data is in the payload. That'll be for the node functions to handle.

But that's not the important part. It's just the beginning. How hard can finding the maximum value be? What about:

```
int max (int* arr, int n)
{
     if (arr == NULL) return -1; // 1
```

```
     if (n < =0) return -1;

     int retindex = 0; // 2
     for (int i = 0; i < n; i++)
           if (arr[i] > arr[retindex])
                retindex = I;

     return retindex;
}
```

Notes:

1. Standard callouts, bail if the value is NULL or if the length is negative.
2. Take note.

Better version:

```
int max (int* arr, int n)
{
     // 1

     int retindex = 0;
     for (int I = 1; I < n; i++)
           if (arr[i] > arr[retindex])
                retindex = I;
     return retindex;
}
```

Notes:

1. Where did our callouts go? This is the thing, all of our good programming practices in theory are presumptions. There's nothing stopping us from completely ignoring them in favor of pure, unadulterated speed. If you are tracking everything properly, you may be building a house of cards, but frequently the code only has to work once. So, in this case, if we shave off those couple of operations from every iteration of max(), we may be saving ourselves hours or days of time on the backend. We just have to be absolutely sure that we never set up a situation where this function is called with a null pointer or negative length. But trusting your future self to never make a mistake is a dangerous process. (Although, there do exist automated tools to help with this.)

2. Another small optimization, but if we already have retindex set to 0, why would we start with comparing arr[0] to arr[0]? May as well start the loop at the second point.

Softmax

That's neat and all, but why does it matter? This is not a textbook on AI. We're only covering the bits that we need to explain context. In this case, that explanation would be too long to go into. Instead, accept that every time the training algorithm in a neural net passes a row of neurons, it needs to create a proportional set of weights that add up to 1. The raw values will be random floating point values, and what we want to do is normalize them as percentages of a whole that equal 1. This means that we need to know what the largest value is.

For extra fun, we want to make sure we aren't inundated with a ton of stupid little individual values that are so rare as to not realistically affect the outcome, but all together overwhelm the important values (consider an array of a trillion values, each of which is equal to 0.0000000000005 except for one that's equal to 0.5). So, we exponentiate everything.

This function is called Softmax and for a short while it was revolutionary in the small field of backpropagation (updating weights in a neural network). It's still foundational, but it's been improved on.

```
void softmax (float* arr, float* outarr, int n) // 1
{
      Float sum = 0.0f;
      for (int i = 0; i < n; i++)
      {
            outarr[i] = expf(arr[i]) // 2
             sum += outarr[i];
      }

      for (int i = 0; i < n; i++) // 3
      {
             outarr[i] /= sum; // 4
      }
}
```

Notes:

1. These functions are also common. Because arrays don't pass back and forth well, we essentially require that both input and output be passed by reference. The output is likely empty or otherwise unimportant

space, but it needs to be allocated to be at least n long. We could check for all of this, but remember that these low level AI functions are running so many times tat extreme optimization is important.

2. expf(x) returns ex. Why ? Because it's the easiest exponential value to calculate in a lot of ways, but that's a different book.
3. Notice that we've got two for loops. for { for {} } is $O(n^2)$. But in this case, for {} for {} is O(2n), two sequential loops are linear, while nested loops multiply, which is certainly smaller. Technically, we could make this three loops, but that would make it O(3n). None of that matters, because they all devolve to O(n). Why then might we care? Because adding up the number of processor operations doesn't really matter at the high level. But they will at the low level like this.
4. This is where we actually perform the normalization.

We have a problem, though. expf(710) will overflow a double. Will we go over 710 for our values? Without knowing the data we're working with, there's no way to know! As a result, we need to do something to compensate. So, we're going to normalize the values based on max before we exponentiate and then normalize again on sum.

Could we avoid this by using another exponentiation base? Yes. But, realistically, it just changes when the overflow happens, it doesn't really prevent it. Not only that, but we want to come up with a solution that doesn't matter with the size of the floating point datatype in case we have other areas we're optimizing.

Side note: Theoretically, if you could guarantee that the overflow would never happen, you could skip this numerical process. But that's practically never a thing you could guarantee.

So, how are we going to fix this? Basically, we want all of the numbers to be small. And what we find is that if we're dividing ex by sum over ex, then it doesn't matter if we subtract something from the x value, as long as it's the same value all the time. Since we don't know what will be most effective, we need to find that first. Hence, our max() function!.

```
void softmax (float* arr, float* outarr, int n)
{
     float maxf = arr[max(arr, n)]; // 1

     float sum = 0.0f; // 2
     for (int i = 0; i < n; i++)
     {
          outarr[i] = expf(arr[i] - maxf); // 3
          sum += outarr[i];
     }

     for (int i = 0; i < n; i++)
     {
```

```
            outarr[i] /= sum; // 3
        }
}
```

Notes;

1. Quick optimization, if we're always going to refer to the same value, grab it this one time and refer to this value later instead of including arr[maxf] inside of the loop, which will have n array lookups (remember that [i] is just sort for *p + i).
2. Why float? It doesn't matter. But, generally, the smallest we can get away with. Because, at the low-level AI structures level of optimizations, a float operation is about twice as fast as a double, depending on the implementation. It gets weird with really big numbers (say, greater than 64 bits on a 64 bit bus), but for the low level, you're fine.
3. Math is weird, but after the second loop, outarr will be exactly the same as in the first instance, except that we've stayed inside the bounds of our data type.

Applications

This is an example of something that could be avoided by using literally any other datatype than a float. Except, how do you represent percentages without floats? That's an excellent question! And people do use proportional ints and single bit weights. They're actually considered cutting edge right now on low-performance AI that can run on mobile devices and such. But, the algorithms didn't fit nicely into here.

What do you do with this softmax? In some organizational environments, 'go fix this function' is as much information as you're given. So, knowing what the function is supposed to do and how to optimize it is super useful. But, moreso, stepping through each part of the function and figuring out why the standard optimizations have been added will help you apply this to specific situations.

For example, suppose you were writing an array where you wanted to A/B test two different backpropagation methods. Maybe you have this idea about how you could use gradient descent but apply the same expf() function to it before you apply to get the exaggeration of difference between statistically significant and insignificant points—that's the first utility of knowing this function, you can see that sometimes just throwing an expf() in there does something and maybe it will do something in other situations you're writing. So the question is, should you run your max to find the maximum value across all of your various input arrays on the expf() or should each array get their individualized maximum? That's the second utility of knowing this function: you start knowing what questions to ask to make modifications. (The answer, by the way, is: try both and see which is better!)

PART II
PART B - #SUCHDATAMUCHSTRUCTURE

Textbooks are a story, just like anything else. We've now set the stage for everything we're doing. You know that there's some sort of measurement we make on data structures and algorithms. You know that there's a formal proof that needs to occur. You've got a baseline programming experience in C. And, you've been exposed to some of the questions and answers we want to face.

The secret to this textbook is to be able to hold two things in your head at the same time. They might even be slightly contradictory. But, for example, more of the sorting algorithm we're going to cover are unlikely to be something that you ever need to write. They are also still vital for you to learn, and not learning them will stunt your programming ability within a couple of years. You could consider that a kind of koan if you want.

Another apparent self-contradiction: data structures and algorithms are completely separate topics that are interrelated but distinct in utility while at the same time data structures and algorithms are basically indivisible from each other.

How about one more? Algorithms only really matter at the lowest level like in C. But, everything about them are vital to programming efficiently and effectively in high level languages.

If we were going to summarize all of this, remember that it's not about what you can memorize, it's about principles and application. Those are hard to write in a textbook, but if you can pierce through the veil of words, you can hopefully pick up the concepts that can apply throughout a career.

CHAPTER 5 - MARCH OF THE TRAVERSALS

Back to Bubble Sort – Naïve to Brute Force

The Bubble Sort we wrote last chapter is kind of bad. Bubble sort itself is kind of slow, but that version of it is what we'd call a naïve algorithm. That is, the solution approaches the problem without much algorithmic thought. It's not necessarily a pejorative term, as we frequently consider a first attempt or a partial success to be a naïve algorithm. After all, just doing exactly what the problem statement does without really any forethought, but doing it better than anyone else might be technically a 'naïve' algorithm, but it's still the best one there is in a given problem.

Naïve Example

However, for the most part, we don't need to be trained in Naïve algorithms. They're just the unstructured approach to problem solving. Here's another example:

```
struct node* findfactors (int target)
{
      struct node* ret = NULL;
      for (int i = 1; i <= target; i++)
      {
            float f = i;
            if (target / f == target / i)
                  ret = addnode (ret, i);
      }
      return ret;
}
```

This is a clever algorithm, but still naïve. Someone realized that if we take advantage of the nature of int division versus float division, we can check to see if a division is an even division (or you could just use %, modulus). It's not a good idea to do it this way, but there's no harm in using it as a first attempt.

No, the real sin here is that the writer didn't consider the fact that every division happens in pairs. So, 12 / 2 = 6 means that both 2 and 6 are factors and we don't need to check 6. Where's the cutoff point? We happen to know that it's at sqrt(target) (sqrt is typical programmer shorthand for Square Root, not to be confused with

SPQR which was the banner the Roman Legion marched under). If you didn't happen to know a given math fact, that's OK, you could always approximate it. Let's try this:

```
struct node* findfactors (int target)
{
      struct node* ret = NULL;
      for (int i = 1; i <= (int)sqrt(target); i++)
            if (target % i == 0)
                  ret = addnode(ret, i);
      return ret;
}
```

There are still a couple issues here, but can you see that we would have way fewer cycles in this? In fact, instead of the naïve approach that's O(n), we've dropped down to O(√n)? On your own, can you remove the sqrt() from the inside of the loop so that we only calculate it once? In fact, while this presumes we have math.h included, casting the (int) like that is possibly not necessary. How would you improve that? Similarly, why not write out the function header.

Finally, as a bit of a fun side question, what should we do if we don't have access to sqrt? Sqrt is kind of an expensive function. It's actually a guess and check method that happens to be the most effective approach to root-finding we have. Mathematicians use a lot of words to avoid calling it a naïve algorithm, so don't do that, even if it may share some similarities. Newton-Raphson is actually fairly elegant. But again, it's fairly computationally intensive compared to most other routine operations. So, if we wanted to avoid that, is there a quick hack we could include that would preclude needing to include the sqrt? (Hint: think of the conditional target / i < i)

Brute Force, slightly better than Naïve

The first formal algorithm design methodology is called Brute Force. We don't really need to worry about classifying algorithms after the fact. However, classifying them pedagogically is helpful in demonstrating what the creative spark is that leads to good algorithm design.

Whereas the naïve algorithm is to just solve the problem without worrying about optimizations or whatnot, a Brute Force algorithm specifically concludes that it is necessary to evaluate every possible option. For example, if you're trying to clear a field of vermin, it's not enough to spot check, it only matters if every square inch is checked and cleared. Doing so randomly is still naïve, so the brute force algorithm would do it in a systematic way that ensures that the vermin are chased into unchecked land instead of the already checked land.

Another example might be checking for primes. The brute force method literally checks every number for whether there are any factors other than itself and one. You might say 'but why even check evens'? And, then

we can start quibbling about whether it's brute force to know the outcome of a calculation so it gets skipped. Regardless, the intent is still there. (Incidentally, the more nuanced version of the brute force prime finder uses the Sieve of Eratosthenes.)

So, how do we transform Bubble Sort from naïve to brute force? The answer is that it's not brute force to be inefficient. It's brute force to check every option exactly once. Here's a reminder:

```
void naivebubblesort (int* arr, int n)
{
      for (int i = 0; i < n; i++)
            for (int j = 0; j < (n - 1); j++)
                  if (arr[j] > arr[j+1])
                        swap(&arr[j], &arr[j+1]);
}
```

In this version, we will always compare exactly n * n times, $O(n^2)$. Incidentally, we frequently shortcut what the complexity of a sort algorithm is by just checking how many different comparisons are being made. Notice that we haven't defined computational complexity, Big-O notation, or anything along those lines. We have, but subtly, we'll formulate a proper treatment in a later chapter when it should feel familiar rather than like a new concept.

The question we need to ask is 'what comparisons don't need to be made?' The answer requires being able to visualize or diagram or trace out the function in whatever way makes the most sense to you. There are any number of methods you can use, for example, what situations never get swapped (LRU – Least Recently Used), or a mathematical approach of formulating when something will never be swapped again, or finding a pattern for when element 'slot' into the correct place.

```
void bubblesort (int* arr, int n)
{
      for (int i = 0; i < n; i++)
      {
            bool swapped = false;

            for (int j = 0; j < ((n - 1) - i); j++) //1
                  if (arr[j] > arr[j+1])
                  {
                        swap(&arr[j], &arr[j+1]);
                        swapped = true;
                  }
```

```
            if (swapped == false)
                  break; // 2
      }
}
```

points to note:

1. This is the clearest example of efficiency gain. While we could check the entire n * n set of possible comparisons, the more of it we can cull, the better! In this case, we're culling about half of the swaps. Why? Because we know that if we've bubbled, say, 5 to the top then we know the last 5 comparisons will be fruitless on the next iteration.

    - The frustration is that there's no universal rule for finding this. In a more advanced treatment of Algorithm Design specifically, some general tips would be covered. But there's still frequently a necessity of that Eureka spark.

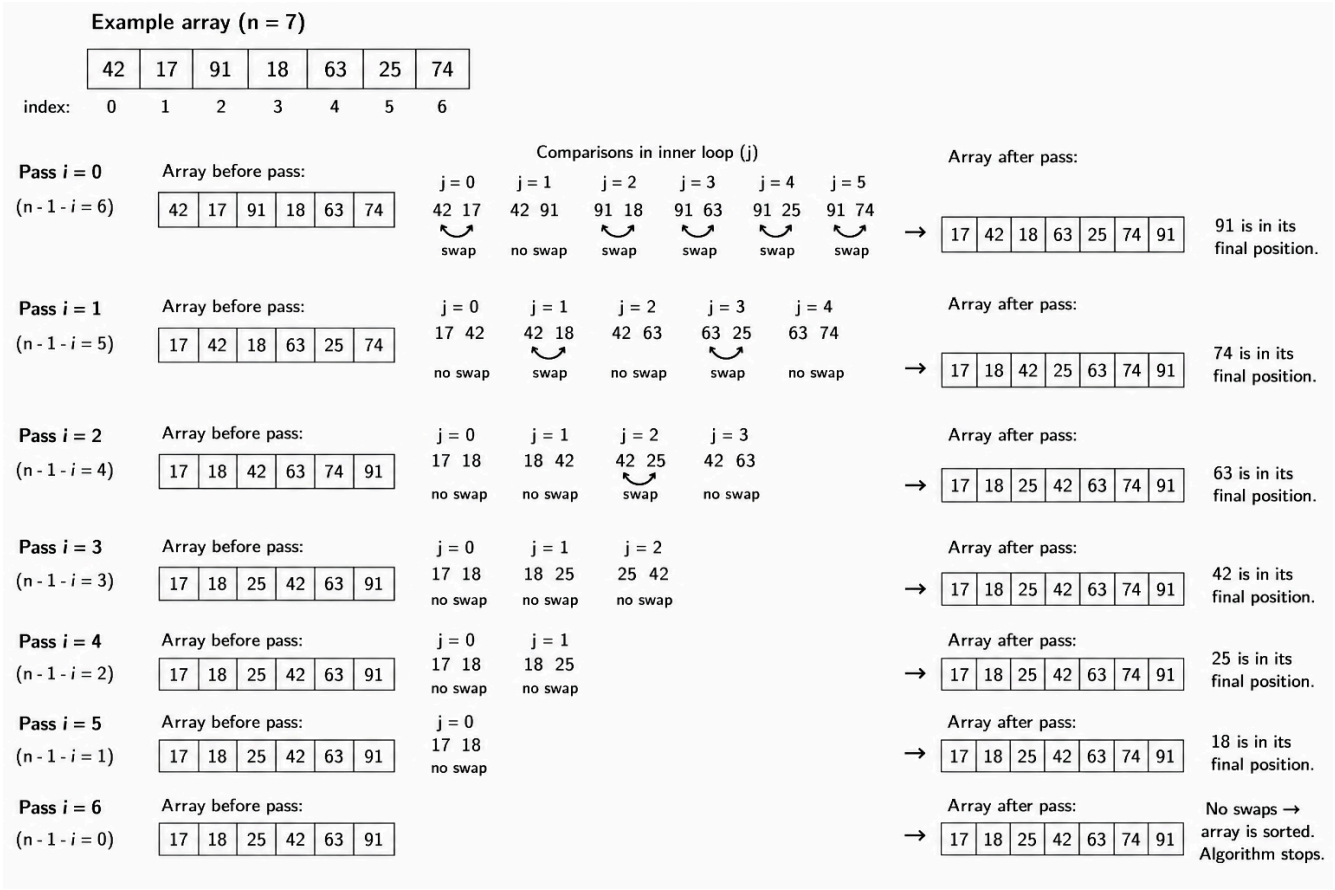
2. This is less consistent, but it still holds true that if we haven't swapped anything throughout the current inner loop then it means that every element was less than the one after it and therefore everything was sorted. We don't need to check everything else at that point.

    - Is it worth having the extra overhead of checking whether it's sorted? This is a valuable, but very practicality-oriented question. Basically, what we need to ask is what are the odds that we will ever have that clause engage. And that depends on the data. If, for example, it used to be sorted and we're just tidying it up, then it's probably very helpful. On the other hand, with perfectly randomized data, the odds that it's the permutation in which all the numbers are in the correct order is 1/n! (1 in n-factorial). So, for a three-element array that's 1 in 6. For a 1,000,000 element array (say, from a medium sized database), it's really big—like a percentage with more zeros than most computers can comfortably display at once. So, maybe we don't need that clause.

We've alluded to it, but there is actually one place where Bubble Sort is sometimes (rarely) the best solution. (Consider how many swaps might be made and when the !swapped clause might execute)

Otherwise, Bubble Sort is easy to remember, easy to implement, and sometimes that's enough.

Example



Linked List Bubble Sort

```
void bubblesort (struct node* head)
{
      struct node* outer = head;
      while (outer != NULL)
      {
            struct node* inner = head; // 1

            while (inner != NULL && inner->next != NULL) // 2
            {
                  if (inner->payload > inner->next->payload) // 3
                        swap(inner, inner->next); // 4

                  inner = inner->next;
            }

            outer = outer->next; // 5
```

```
        }
}
```

Linked Notes:

1. We don't have iterator values, so we're going to have pointers traverse our chain of ->next links.
2. This part gets a little jury-rigged. Basically, we need to be sure that the conditional at // 4 doesn't try to compare the last node in the list with something that's beyond the list. But what if inner is NULL? C has a specific, defined behavior that it breaks if statements at the first false statement, so we won't get to the next if. This is non-ideal and doesn't always transfer well to other languages. It's probably clearer to break it into a separate if statement.
3. We have to compare the ->payloads, which is the equivalent of arr[j] > arr[j+1]
4. Just a little hand-waving here. We could keep track of prev and swap the actual nodes themselves. Or, in this case, we can just swap the payloads and leave the nodes where they are. We're not worried about it for this part of the algorithm, though. That 'whatever is most appropriate, but we don't care at this given moment' is why abstraction is so helpful in Object-Oriented Programming and it has inculcated itself into most programmatic thought.
5. Turns out that outer was basically nothing more than a counter variable, but because we don't have an n, we don't know how long the array is.

For a quick side exercise, how would you include the prev node in order to swap the actual nodes in the chain? Also, can you come up with ways to adjust this code so that we can include the !swapped escape? What about to never compare the nodes that are already slotted in the right place (this one is a little tricky, you'll need extra pointers)?

Which do you prefer? Do you prefer the array, with its direct access, or the node, which is more self-contained. As a side note, that's part of the appeal of the linked list: each node acts independently and we can create complex behavior our of self-encapsulated behaviors.

Selection Sort

As you might expect from any escalating series of algorithms, each of these algorithms is a little more complex and demonstrates a slightly more optimal sorting experience (in the general case). Selection Sort is exactly what the name sounds like. It selects the next value we want. And builds the list iteratively that way.

Side Note: Where do these algorithms get their names from? Most of this introductory knowledge is considered part of the public domain now. But, originally, they would have been proposed by mathematicians or computer scientists. Selection Sort, for example, was proposed in the 1950s as being novel in use and

application. And then it's been refined over the decades as people slowly distill what it does best and how to describe that as concisely as possible. Names and approaches change over time.

Let's consider the sort process holistically. Imagine you're sitting in front of a big pile of random length blocks and you need to sort them by size. Are you going to lay them out in a line and then iteratively cycle through them comparing each one to the ones next to them? Or are you going to look through all the blocks, find the smallest one, and put it in the beginning of your ordered area. Then the second, and the third. While the former is indeed a situation that would work, this way makes more sense to us as a human, right?

So, we need to try to see if we can simulate that process and whether or not there is any utility in doing it this way.

```
void selectionsort (int* arr, int n)
{
      for (int i = 0; i < (n - 1); i++) // 1
      {
            int min = i; // 2
            for (int j = i + 1; j < n; j++) // 3
                  if (arr[j] < arr[min]) // 4
                        min = j; // 5
      }
      swap (&arr[i], &arr[min]); // 6
}
```

Selected Lines:

1. Outer Loop, notice that we cap it at n – 1, since we know that we'll have an i + 1 inside the inner loop.
2. This is what we might called a posit statement. We posit that the element at index i is the smallest element of all the elements higher than it. The rest of what's in the outer loop is disproving that. Side note: The distinction between creating a primitive data type inside the loop (and thus recreating it every loop) and creating it outside is valid, but we're trying to keep things simple. Feel free to test how much affect it has in practice.
3. The inner loop. Notice that once we've passed an iteration on the outer loop, we don't care about that element anymore, because we know it's the smallest.
4. We're making a comparison, and testing the posit we made earlier. min used to be i. Maybe it still is, or maybe not, but if we scan through the array and every time we find a smaller number, we update min to say that it's the new smallest, by the end, we'll know we have the smallest.
5. We'll update min to be the smallest so far, aside from the values that have already been selected and slotted into place. This is where we make the actual swap.

So, let's take a look at this. We have an outer loop and an inner loop. So far, it still looks like $O(n^2)$, right? But, what's inside that inner loop? In Bubble Sort, it was a comparison and sometimes an actual swap. The comparison and the swap are both fairly cheap. But, think about the cost of a spaghetti noodle. A spaghetti noodle is incredibly cheap, but a factory producing those noodles for a year is fairly expensive. So, even though having a cheap spaghetti noodle become slightly cheaper by using a new kind of flour is kind of irrelevant, it adds up over the whole factory full, right? And, spaghetti noodle experts probably know whether it's worth their time to make the flour or the water or the salt or any other ingredient cheaper based on the relative value of the ingredient to the noodle.

The same thing happens here. A swap is cheap. A comparison is cheap. A million comparisons are still cheap but a million swaps are probably much more expensive (computationally).

Best First

We have considered the Brute Force algorithm design method, so here's the Best First method. But typically, there are some holistic improvements we can make or base assumptions to change. In the case of selection sort, we assume that the comparison is much cheaper than the swap (not actually a given, but probably true more often than not). So, suppose we have an array of 1000 elements. Because the comparison is inside the inner loop, we will perform $n^2 = 1{,}000{,}000$ comparisons. But, because the swap is in the outside loop and not the inside loop, it'll only occur at $n = 1{,}000$ swaps. Well, technically it'll be closer to 500,000 and 1,000 because the inner loop iterator resets to the value of the outer loop iterator. Compare that to Bubble Sort's 1,000,000 and 1,000,000 respectively, and it's apparently that Selection sort is(careful) faster.

This is the Best First design methodology. The general pattern here is that we are willing to do all of our analysis up front (typically in each cycle) and then to pick the overall best decision to engage in. Then, the cycle for the next problem will be a little smaller (in our example, we picked the smallest of 1,000 elements, and the second cycle will only have to pick the smallest of 999 elements).

Assumptions

The tradeoff to the Best First methodology is deciding that the extra analysis before acting means that you have to take fewer actions. Alternately, you're presuming that the savings in the active phase is substantively larger than the cost of the analysis.

How would you make that decision? Well, at some point, it'll be fairly simple to count how many times various functions run. You could literally set a global variable and increment it by one at the beginning of the function, the counting itself takes some time, but that should be minimal and offset by the fact that all the measured functions are being measured the same way. Then you can run some timing functions (check the time before the function runs, after the function runs, and then getting the average run time) and that can give

you an indication if this assumption is warranted. It typically is, but the why of that is a little more involved, so don't feel compelled to read the rest of this chapter.

Your process operates in the scale of billions of operations per second. But your memory access speed is typically going to run in the range of thousands of MB/s. Since your processor will operate on a 64 bit bus and is probably operating on multiple decisions at once, the processor comes out a little ahead in the speed. That may not be a perfect rule, but it's generally a good assumption: the CPU is the fastest thing on a computer.

This is compounded by the fact that the memory is physically distant from the CPU and frequently has to page between several layers of cache. The point being that the processor is running at several operations per nanosecond and the memory is going to lag behind by several dozen to several hundred nanoseconds. The result is that evaluating data that is in memory requires finding and then transmitting the data out. But writing involves finding the data, transmitting the data in, and then a physical mechanism to change the physical medium. This is a severe reduction.

The point is, assuming that any write activity will be slower than solely a read activity is generally a solid assumption. But, there are situations in which this isn't the case. For example, suppose we were looking at sorting pictures on a smartphone based on how much red was in the picture. That analysis would take a fair amount of computation, but the actual swap wouldn't be switching places on the disk, it would just be switching pointers to file locations. So, the Best First methodology's assumptions would lead you astray in that case.

How do you know when that's going to happen? A little bit of thinking, a little bit of analysis, and a lot of testing.

Challenges

The code from above sorts the data from smallest to largest by going from the beginning to the end. Consider how you would adjust it to sort it from largest to smallest by going from the beginning to the end.

How would you adjust it so that you sorted from smallest to largest by going from the end to the beginning. And largest to smallest by going from the end to the beginning.

Is there any utility in these variations, or will they always be the same?

Spoiler warning: What if you tried sorting from smallest to largest from both directions at once?

One last challenge: Can you come up with all of these, but with a linked list instead of an array?

Insertion Sort

This is a bit of a specious distinction, but we generally consider Insertion sort slightly faster than Selection sort (which is considered faster than Bubble Sort). Whereas Bubble lets bigger data slowly bubble to the top and

Selection grabs each value that needs to come next, Insertion Sort effectively builds a new array out of the pieces that were already there.

So, suppose we've got a series of runners and times at a major marathon, and we want to put together a big display board that lists every runner and their time in order from fastest to slowest. Are you, the human in a very physical (although imaginary) scenario, going to go through every one of the hundreds or thousands of entrants to find the smallest entry and then do that again for the next smallest entry?

Or are you likely to take the first entry and say 'done, I have an ordered stack of running times that is exactly one tall!' Then you'll look at the top entry in your stack of unsorted ones and say 'is this faster or slower than the one in my sorted stack?' and you'll put it in the sorted stack as appropriate. And then you'll look at the third and figure out if it goes on the sorted stack at the top or the bottom or in the middle. The point being, that you're building the sorted list rather than trying to rearrange values.

We can manipulate this a little bit and recognize that this isn't necessarily a 'new' stack, because maybe you're just rearranging pages inside a folder or file folders inside of a cabinet. But the point is that the framing of the Insertion sort is still an array that needs to be sorted. But how we're going to handle it is different.

```
void insertionSort (int* arr, int n)
{
      for (int i = 1; i < n; i++) // 1
      {
            int key = arr[i]; // 2
            int j;
            for (j = i - 1; j >= 0 && arr[j] > key; j--) // 3
                  arr[j + 1] = arr[j]; // 4

            arr[j + 1] = key; // 5
      }
}
```

Side Track: Array Size

Suppose you have an array of n elements. Let's further suppose n is 50. How big should you make your array? The answer is probably 50. There's nothing wrong with that answer. But what if you need to add a 51st element? You need to create a new array, copy all of the elements to the first 50 elements of the new array, and then finally you can add the 51st element in that new array. Now, what happens if you need to delete the 3rd element? Well, you've got two options: You can copy over to a new array and copy over everything but the 3rd element or you can just not card. You've allocated 51 chunks of memory but you only use 50.

What's the downside there? The biggest one is obviously that you're wasting that single element. And that's

a 2% overhead in something like an array of integers. That's a consideration that needs to be made: which is worse to have a 2% wastage in space concerns or to have to use the computation time to create a new chunk of memory (for a moment, you'll have a 100% waste as you have two simultaneous arrays) and then the copying portion. All for a single element to be added?

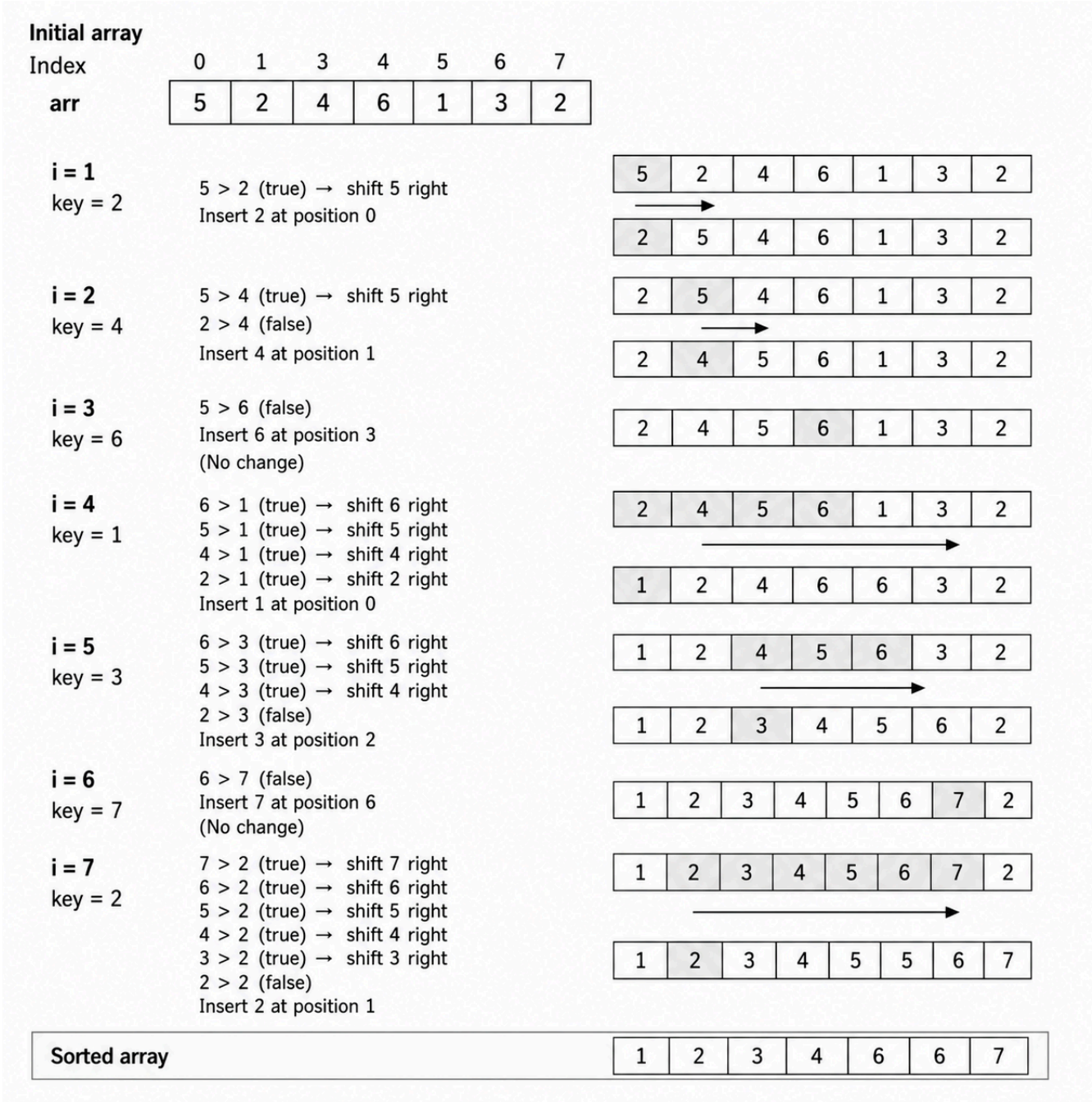
What if we had an array of 500? 5,000? 5,000,000? Saving the memory space of a single array, is it really worth saving the 0.00002% in space by copying over a million values? Obviously, the answer becomes a resounding 'No!' This is especially true when we go back to the archetype we noted earlier of an array of pointers, the overhead of space constraint shrinks even further. We can go on, but it'll belabor the point. The point is that there is an elegance in 'wasting' that space, and the tradeoff (as very frequently will be the case) is in time versus space.

In the case of static arrays, the shortcut almost every program uses is to just declare really large arrays. You need to store 50 elements? How many will you likely every need to store? 1,000? Then let's just make all of the arrays 1,000 in size and then we can just change the n of array length, knowing that we can change that n within the array as long as it never goes above 1,000 (or whatever our MAX_ARRAY_LENGTH is set to for our program).

So, if we don't need to worry about actually exceeding the length of the array because more space is allocated than we needed, why do all of the algorithms worry about it? Because it doesn't matter whether the technical applications are sidestepped or not, the algorithm should generally be correct outside of those concerns. If we say that the array is of length n, then we need to be able to respect that length of n (as always, there are exceptions, but it's standard to assume not in an intro algorithms course).

This is important right here, because in Insertion Sort, we said we're building a new array. And, we are: at any given point, we have two subarray, with the pivot point at *i*. Everything before i is sorted, everything after i is unsorted, and i is what we're slot into place.

Example



Insertionotes

1. The outer loop controls the main progression of the algorithm. In this case, `i` also serves as the dividing point between the sorted and unsorted portions of the array. Notice that `i` starts at 1 because we assume that a single element by itself is already sorted.

2. At each step, the array is effectively divided into two parts: the sorted portion and the unsorted portion. As the unsorted portion shrinks, the sorted portion grows directly beside it. The index `i` marks this division: everything before `i` belongs to the sorted portion we have already built, and everything from `i` onward belongs to the portion that has not yet been fully processed.

3. Because `arr[i]` is the key value we want to insert into the sorted portion, we first store it in the variable `key`. This is necessary because the elements larger than `key` will be shifted one position to the right, which may overwrite the original value at `arr[i]`.

4. The inner loop works backward through the sorted portion of the array. As long as `j` is within the array and `arr[j] > key`, the element `arr[j]` is shifted one position to the right. This continues until we find the correct position for `key`. This part could also be written as a `while` loop, and that might be easier to read because the loop condition has two parts: `j >= 0` and `arr[j] > key`. However, since the previous examples used `for` loops, this version keeps the same style.

5. The statement `arr[j + 1] = arr[j];` performs the shifting step. It is safe to use `arr[j + 1]` because `j` begins at `i - 1`, so `j + 1` begins at `i`, which is a valid index. The condition `j >= 0` prevents the loop from accessing positions before the start of the array. Finally, once the correct empty position has been created, we place `key` back into the array: arr[j + 1] = key

Since we're bringing these up in a particular order, you can guess that each one is faster than the previous. Again, we haven't gotten into what 'faster' is, because we want the term to be nebulous for you for a bit. It's an error to equate 'faster' with a universal gradient. But, let's look at it again: Bubble Sort was $O(n^2)$, Selection Sort was $O(n^2)$, and Insertion Sort is also $O(n^2)$? It's got an outer loop and an inner loop, so how can we say that one is faster than the other?

Remember that for selection sort, the primary focus was that we pulled one of the components out of the inner loop and only had it run once every outer loop. For insertion sort, we're doing the same thing by having the swap only occur every outer loop. But the comparison is also approximately half as frequent in selection sort.

In Bubble Sort, we saw that if we skipped the comparisons that were guaranteed to already be in order, we could cut the number of comparisons in half. In Insertion Sort, we're seeing that if we have a situation where we only have to make the comparisons up to a point, then we can reduce the total number of comparisons even further.

However, whereas there was a fixed amount of increase in naïve Bubble Sort to brute force Bubble Sort and a fixed amount of increase in Bubble Sort to Selection Sort (this is not strictly true, there's a whole field of study on evaluating different cases and how they reflect on selection sort), there's a different situation here. How much we save on moving from Selection Sort to Insertion sort is questionable.

We will always have the same number of swaps. We discussed why earlier, but if we presume that swaps are the really expensive part of the algorithm, that's why Bubble Sort fares worse, it'll always be $O(n^2)$ swaps versus the O(n) of selection and insertion sort.

Variable Time Savings Analysis and Methodology

In many ways, it doesn't actually matter between selection and insertion sorts. If the discrepancy between comparison and swap is high enough, saving time in the comparison portion could be almost irrelevant (it'll never be actually irrelevant, but how much time do you want to save for an improvement of 0.001%?).

This is where the algorithm analysis portion starts to come in. And algorithm analysis is incredibly easy in practice. Side note: It's obnoxiously difficult in theory and for theorists, but for practitioners it's super easy. We're going to bracket our analysis into three catered categories:

- Worst Case Scenario: in the case of a sort, what's the worst case scenario? Suppose that the 'random' data was really just sorted backwards but there was no way for us to know. In this case, we'll end up with the same number of swaps, but the most comparisons.
- Best Case Scenario: in this case of a sort, this is the case where the data is already sorted! This is why Bubble Sort is sometimes called fastest for mostly sorted data: a) it has that exception of 'if the data is already sorted, leave' (but we can fix that by literally checking for sortedness before we start) and b) because in the very niche case of data where there's only a few data points that are out of position by literally one place, we might be able to sort the data with less than n swaps (but again, judicious additions of if statements can compensate for this in other sorts). The point is, even though best-case seems obvious, it's part of our bracketing.
- Average Case Scenario: if there are different exemplars, this case might need to be more than one case, but generally you can get away with at least an initial overview with just one. 'Average' in this case typically means most normal, or middle of the set, or expected middling results. It's not subjective, it's just 'what represents what the average thing to happen will be'. This is important to clarify because when we says 'random data' we don't actually mean 'random data'. Random typically ends up with a number of clusters and hills and valleys. Instead, we want pseudorandom data that is uniformly randomized as our 'average' test case. The nice part of that is that pseudo-random data is much easier to simulate than actual random data (although actual random data is more common in reality).

And then the analysis from here is simple! We just run the algorithms and count how many times each operation will occur. Broadly, and this is not exact, but at the level we're at, we can generally ignore the structural code that just greases the wheels. It's too much to get into, but that's one part because it's too picayune to worry about for most applications and one part because as the other code increases the structural

code will probably increase proportionally. Again, those aren't exact, but for now, and until you have reason otherwise, don't worry about the structural stuff (like the for loop or the i++ or whatnot).

Worst Case Scenario

- Bubble Sort: ½ $O(n^2)$ comparisons and ½ $O(n^2)$ swaps because it basically has to touch everything once per inner loop, but the loop size shrinks with the outer loop iteration.
- Selection Sort: ½ $O(n^2)$ comparisons and O(n) swaps. Each element just gets put in the correct place, but the method to find that correct place is effectively the same as bubble sort.
- Insertion Sort: ½ $O(n^2)$ comparisons and O(n) swaps. In this case, there's no benefit to using insertion sort. There might, in fact, be a slight loss based on the structural code we just said not to worry about it. So, we're not going to worry about it, because it would be minimal enough to not be a problem.
- So, in the Worst Case Scenario, we don't change how many comparisons are being made. This makes sense, it's sort of like when you've got 300 pages left to write in your book, it doesn't matter how you try to repackage it to yourself, it's still another 300 pages of book, right? RIGHT?

- Best Case Scenario
  - o Bubble Sort: ½ O(n2) comparisons and 0 swaps, because everything's already in order. If we added the standard 'no swaps in the last inner loop, then break' clause, it would be 0 and 0. This is all a bit of a misnomer, though, because we can modify any sort to do that.
  - o Selection Sort: ½ O(n) comparisons and O(n) swaps. In essence, unless we add something to stop it, it'll try swapping every index with itself. Is it worth stopping that from happening? Try it and see! What's the best way to avoid that, in the sort method or in the swap method?
  - o Insertion Sort: ~0 comparisons and O(n) swaps. This would be where it really shines. Remember how the inner loop scans backwards through the sorted portion to find where the pivot value goes? If the data is already sorted then key value will always be bigger than anything in the sorted portion. And then it'll try to swap it with itself, so same as the Selection Sort in that regard.
  - o Overall, in the best case scenario, we should just assume that there are going to be 0 swaps, because in practice we'd probably add the code to stop a self-swap if it was an issue, they just aren't part of the canonical algorithms because that adds a little too much to memorize.
- Average Case Scenario
  - Bubble Sort: ½ $O(n^2)$ comparisons and ½ $O(n^2)$ swaps. Same as the previous two scenarios.
  - Selection Sort: ½ $O(n^2)$ comparisons and O(n) swaps. The swaps will generally probably still happen, because the odds that a value is in the exact right position drop down at a factorial rate and aren't worth noting.
  - Insertion Sort: ¼ $O(n^2)$ comparisons and O(n) swaps. The same number of swaps will happen as Selection Sort, but because we're only going to be ever evaluating half of the sorted half of the array, we can drop the comparisons down to an average of ¼ instead of the existing ½.

- So what we can see in the perfectly average scenario, we see an increasing ability of insertion sort to save some time over selection sort. How much time, depends on what proportion of the overall execution time is taken up with the comparisons. However, that just changes whether it's a presumably small or a presumably negligible savings, but never a worsening.

Side Note: All of these O()'s we've been mentioning are part of Big-O notation. And it deserves almost an entire chapter by itself. You're welcome to read ahead, but the details are less useful than the practice. What you'll find when we get to those details is that this idea of adding coefficients in front of the O is actually silly. However, it's also communicative of what's going on. So, please recognize that we'll have better tools later, and using this framing will get you laughed at during a Snooty McSnootyFace Academic Convention. But, for regular practice, people will understand what you're trying to convey, which is the primary metric of importance in the vast majority of situations.

What do we observe overall? The term is 'monotonic' that is the term for when something never gets worse, but it may stay the same. So, in every single case, Insertion Sort is better than Selection Sort, and in almost every single case Selection Sort is better than Bubble Sort. In practice, just presume the cases where Bubble Sort is better are going to be so obvious that you'll call your friends and tell them about this super weird situations at work where you actually used Bubble Sort. And one of them will point you to Binary Trees and suggest you consider B-Trees instead.

Design Methodologies Revisited

So far, we've discussed Naïve Algorithms, which are when you're just finishing the task and don't care about doing it well. We've talked about Brute Force algorithms, where you're specifically noting that you want to touch each part of the process along the way, possibly for ancillary reasons. And, we talked about Best First algorithms where the algorithm does some analysis first and the overhead in the analysis is outweighed by the fact that the globally correct answer is selected to the local problem.

Insertion Sort is what we call a Greedy Algorithm. The Greedy Design Methodology says 'just pick the best one right now, and don't worry about whether it's perfectly correct'. So, you see in this case, the Greedy potion could be framed as which of the element to put into the correct place next (the next one, duh!) or that, given the next element where should it go (the right place, duh) and then the algorithm single-mindedly engages in that solution.

What's interesting is that the Greedy Design Methodology is frequently a good algorithm method! Especially for simple, framed, or highly constrained problems, it turns out that the solution is frequently baked into the problem. Alternately, we can create a set of constraints such that the greedy methodology is fast and correct in designing the solution. Basically, in human terms, it would be like 'do the hard tasks first, so that if you run out of energy you only have easy tasks left' only it's nothing alike in rationale and the fact that the same outcome occurs is pure happenstance.

What's the difference between Best First and Greedy? In Best First, we want to find the next best step by considering our overarching goal. Greedy Algorithms, on the other hand, aim for immediate gain, which means that when the Best First algorithm could do all of the analysis and come back with the most obvious solution most of the time, a Greedy Algorithm will do well. For example pathfinding with no maze walls.

So far, all of our approaches have been very linear. We're thinking in terms of 'how to address all of the iterative parts of the problem'. And there are a number of other design methodologies that do so.

Remember that the goal here is to find good solutions to problems, not to obsess over labels. In practice, the labels are only useful for communication. For example, which of these algorithms is which is mostly a matter of rhetoric. Any of the three searches we've covered could be argued to be any of the design methodologies we've discussed. Further, the boundaries between the methodologies are kind of contrived and fuzzy (for example, Best First vs. Greedy vs. Local Neighbor).

Lastly, you'll find that people will tend to describe algorithm design methodologies like they're the algorithm itself. That's fair, but it would be probably more useful to consider these design methodologies as observations of behaviors that certain optimal solutions share, and what characteristics might lend them to apply to a novel problem.

Merge Sort

This is the first topic that is complicated in the text. Don't worry, just remember that there are people you can ask and tutorials you can review to gain alternative explanations. But, notice that in the previous section we said that the existing design methodologies were very linear. We can see that because when they were looking at the number of comparisons, it was always a question of some variation of looking at every other element, for each element in the array. That linear approach for each element is tricksy, and maybe we can do better?

Logarithms Review

Let's take a brief moment and review how a logarithm works. Because this is going to be a big deal in a moment. We don't need to be able to solve something like zf(x) = yx. That would be frustrating to do on a computer for routine programming. Instead, we just need to know how they generally work. Remember that for a lot of things, we can look up the mechanics or program a computer to do them for us (hey! That's the name of the industry!). But knowing the basic principles of why and how are what are important.

So, remember that in math we have functions like $f(x) = x$. Or $f(x) = x^2$. Or $f(x) = x^3$. Or even the dreaded $f(x) = 2x$. Without seeing them, you should be able to visualize their general shape. Quadratic functions are the parabolas where both ends point in the same direction. Cubic functions look like parabolas, but the ends point up and down instead of both in the same direction. And more importantly, they're steeper. Quartic functions are like parabolas, but even steeper, etc. In fact, in $f(x) = xc$, as c gets bigger, the curve gets steeper.

Mostly, when we talk about how long a function takes, we only care about the positive values, as we're looking at number of inputs versus time. And we generally can't have a negative amount of time or a negative amount of inputs. So, (0,0) is almost always the start point, and we're concerned with how fast the function escalates.

In there, f(x) = x has a special place. That's out baseline. What it means is that for every input, it only takes us a single operation to process it. Later on, we'll see that's basically impossible, and it'll always be some ratio, like f(x) = cx, or for every input, it will take some number (c) of steps to complete. But, and take this on faith for now, for various reasons, when we analyze functions, we tend to just generalize any cx to just x. Any cx^2 to x^2, etc. But the point is still, that we count up how complex a function is by, among other things, how many nests deep a for loop we have. So:

```
for (int i = 0; i < n; i++)
{
}
```

Is an O(n) operation. But:

```
for (int i = 0; i <n; i++)
{
      for (int j = 0; j < n; j++)
      {
            for (int k = 0; k < n; k++)
            {
            }
      }
}
```

Is considered an $O(n^3)$ operation. And as that exponent increases, the computational complexity increases. We want lower computational complexity, because that means our program operates faster (in real business terms, that means less computational resources spent and, if nothing else, less electricity purchased). So, we want a lower nesting of loops. This is a severe overgeneralization, but it's basically true. All computational complexity is, at the end of the day, some complex version of counting nested loops.

This, in turn, is why we really don't like $f(x) = 2^{n.}$ Notice it's the opposite of $f(x) = n^2$. But that is substantial. Whereas in n^2, when n increases to n+1

, the amount added is only about 2n+1

. In contrast, 2^n doubles every time n increases by 1. That means the new value becomes 200% of the

previous value, or equivalently, it has increased by 100%. This is exponential growth, which becomes much larger than polynomial growth such as n^2.

So, all of this is the negative side of computational complexity. We have f(x) = x and that's a straight line, and everything else is above it. Bad, boo. Get off the stage! What about if we looked at what was below the line? We'd be looking at faster than linear time! But what would that look like? There are a lot of functions that are below f(x) = x. But most of them are impractical.

One, however, is relatively common: f(x) = log(x). log() stands for logarithm. Logarithm of what? There are two answers: 1) it doesn't matter, all of the important values are smaller than f(x) = x and 2) the answer's 2, this is computer science, we like binary and 2. There's 2 boolean values, so it's always 2 in CS. The former answer is also true, but that smacks of math, we like to just use 2.

So, if 2^x means multiplying 2 by itself x times, what if we wanted to ask the question in reverse? Suppose I have the number 900. How many factors of 2 do I need before I get past 900? That is the idea behind a logarithm. Exponents ask, "What number do I get if I multiply this base by itself a certain number of times?" Logarithms ask, "How many times do I need to use this base to reach a certain number?" For example, since 2^9= 512 and 2^{10}=1024, the value of $\log_2(900)$ is somewhere between 9 and 10. In this application, we round up to 10 because 9 factors of 2 are not enough to reach or exceed 900, but 10 factors are enough. We will discuss why this later.

The important part is this. Suppose we have 1,000,000 inputs, if we have an O(n) algorithm, it will take us 1,000,000 steps to complete. If we have an $O(n^2)$ algorithm, it will take us 1,000,000,000,000 steps to complete. If we have an O(log(n)) algorithm, it will take us 1,000 steps.

This means that O(log(n)) is even better than O(n), but it's a little more nuanced. One last point, we can mix n and log(n). For example, if we have O(nlog(n)), that means that we're looking at more than linear (O(n)) time, but not as much as quadratic ($O(n^2)$) time. Hopefully you can guess where this is going, since we've so far covered a bunch of $O(n^2)$ operations.

PS This shouldn't have been too bad. It was mathish, but that was just a long and drawn out way of reminding you that logarithms are the inverse of exponents in a different way that roots are the opposites of exponents. A root removes the exponent and leaves the number. A logarithm removes the base and leaves just the exponent.

Binarystreet's Back Alright

Let's jump into a game you've played before. The game master is thinking of a number between 0 and 100. What's your best guess? Everyone knows that it's 50, because even though each number has an equal chance of being right, at least with 50 you'll never be that far off. If we play this game as an iterative one where the game master has to tell you if the number is higher or lower than yours, it means that you can repeat the process in the leftovers until you get to the number. In fact, it's almost like an algorithm.

```
define guessinggame (max, mysterynumber)
{
     a = 0, b = max
     guess = (a + b) / 2
     while (guess != mysterynumber)
           if mysterynumber > guess
                a = guess
           end if
           if mysterynumber < guess
                b = guess
           end if
           guess = (a + b) / 2
     end while
     return guess
}
```

How many guesses will this take? Each guess cuts the search space in half, so the number of guesses grows like

\log_2(\text{max})

$\log_2(\text{max})$, rounded up. For example, 100 takes at most about 7 guesses, 1,000 takes at most about 10 guesses, and 10,000 takes at most about 14 guesses. This is why logarithms matter: even when max becomes 10 times larger, the number of guesses only increases by about 3 or 4.

Compare this to the Bubble Sort level equivalency of the same algorithm:

```
define stupidguesser (max, mysterynumber)
{
     guess = 0
     while (guess != mysterynumber)
        {
           guess ++
         } end while
     return guess
 }end
```

This version is simpler and easier to write, but it performs much worse. If max is 100 and the mystery number could be anywhere from 0 to 100, this version may take as many as 100 increments in the worst case. On average, it will take about 50 increments.

It's important to qualify that word 'faster' again. Suppose the mysterynumber was 0 and max was 100.

How many iterations of the while loop with the first version take? 7 (50, 25, 12, 6, 3, 1, 0. How many iterations of the while loop will the second version take? 1 (0). So, in some cases, the slow one is faster!

Actually, we can just count up how often that is. The nice part about computer science is that it's frequently a discrete environment, which means there are only a fixed number of possibilities. If max is 100, there are only 100 possibilities for mysterynumber! Let's try them all!

```
mysnum:0,fast:7(50,25,12,6,3,1, 0), slow iter:1 (0).
mysnum:1,fast:6(50,25,12,6,3,1),slow:2(0,1).
mysnum:2,fast:7(50,25,12,6,3,1,2),slow:3(0,1,2).
mysnum:3,fast:5(50,25,12,6,3),slow:4(0,1,2,3).
mysnum:4,fast:6(50,25,12,6,3,4),slow:5(0,1,2,3,4).
mysnum:5,fast:7(50,25,12,6,3,4,5),slow:6(0,1,2,3,4,5).
mysnum:6,fast:4(50,25,12,6),slow:7(0,1,2,3,4,5,6).
mysnum:7,fast:6(50,25,12,6,9,7),slow:8(0,1,2,3,4,5,6,7,8).
mysnum:8,fast:7(50,25,12,6,9,7, 8),slow:9(0,1,2,3,4,5,6,7,8,9).
mysnum:9,fast:5(50,25,12,6,9),slow:10(0,1,2,3,4,5,6,7,8,9,10).
mysnum:10-100,fast:<=7,slow:>10.
```

You should be able to write a quick program to show this, only all cases and with nicer output. But the point is that there are only 7 cases where the slow version is faster. This is why we talk about faster or slower algorithms even though all that matters is what actually happens. When we're planning ahead, we want to know what percentage of cases will happen faster with one algorithm versus another. A little bit of analysis and we can see that the slow function is only faster in a max of log(n) cases. Since log(n) grows much slower as n grows than n does, the disparity will just increase over time.

Side Note: This is important. Please don't just skip over this as being obvious. For example, what if we had a special case where we were only ever searching with an n such that log(n) > 50% of n and therefore the slower search was actually faster in that specific case?

MergeSort algorithm

So far, you should have seen the disclaimer that we're not going to be doing things linearly. And further, that binary search is a question of O(log(n)) rather than the previous linear search of O(n), right? And, as a reminder, O(log(n)) is 'better' in the sense that in most cases, it will perform faster than an O(n) algorithm.

Now, imagine you have a deck of cards that are all randomly shuffled together. This is going back to our set of physical examples to help visualize what's happening. This deck of cards is completely random. Maybe it even has some Yu-Gi-Oh cards mixed in. Who knows, but there's 64 cards there for some reason! Well, we know that eating a whole pizza is difficult, but eating a single 1/8th slice is routine, right? Could that apply here? Maybe we can split the deck of cards in half, and sort each half. So far, this is making the task harder,

but (and this breaks the analogy a little) you can see how sorting half of a deck could be a little easier than half the difficulty of sorting the whole deck, right?

Well, what about these half decks? Do we dare to split them again? And then we'd have four quarters of a deck. Really, we have two halves split into halves, which is important, but not to this analogy. So, what do we do with the quarters? Could we split them again? Basically, we've come up with a rule: we don't really have a good idea of how to sort 64 cards when we're holding them in our hands: swapping adjacent cards (bubble sort) seems like a hassle and rough on the cards, finding the biggest card and moving it to the bottom of the deck is a lot of rifling through the cards, similarly, starting a new stack of cards and inserting each new card into the new deck seems like we'd get tired of looking through the sorted deck pretty fast.

So, we've got these eighths of a deck (or, again, halves of halves of halves of the deck). Following our logic, we'd split it again (sixteenths)! And then again (thirty-seconds)! Now we have thirty two subdecks each with two cards. What should we do? I mean, it's only two cards in each deck, surely we coul– Split them again! Now we've got 64 subdecks each 1 card each. What's actually important here is how we were arranging them. You can figure out something (maybe a pyramid style, maybe a tournament bracket style, maybe a branching tree), but the important part is that we don't have 64 sub decks. We've got halves of halves of halves of halves of halves of halves (that's 6 halvings).

What's something else we already know about single cards? They're already sorted! Each card in its subdeck is already entirely sorted! So, what if we combined them back up? We go back to thirty-two subdecks of 2 cards each, but this time when we combine the subdecks back together, we put them back together in order. That should be easy, right? They're already in order, so if we've got subdeck A and subdeck B and they're going into a new subdeck C, the card we add to subdeck C next would just be the smaller of top cards of A or B, right? Then we take those 32 subdecks of 2 cards each and merge (there's that word again!) them together into 16 subdecks of 4 cards each, but again, they're in order. And then 8 subdecks of 8 cards each, but now they're in order. And then 4 subdecks of 16, 2 subdecks of 32, and then 1 subdeck of 64. Each step of the way, we maintained the sorted order of the subdecks as we merged them together.

This is MergeSort. We call it that, because of all the merging we did on our way back up. But what's interesting here is that when we split the decks apart, to get each card down to its single card subdeck only took 6 actions. Since we know (or we trust and then verify) that the merging process will result in a sorted deck, we could also phrase it such that it only took us those 6 acrions (or log(64)) to figure out where in the deck the cards should go.

We count it as taking us log(n) comparisons to find where each of the n cards should go, but we still need to address each of the cards in turn so this algorithm (and likely no algorithm) will get below O(nlogn) in this regard.

This is also a longer bit of code, also using a for loop here would be more confusing.

```
void merge (int* arr, int left, int middle, int right) // 1
{
     int i = left; // 2
```

```
    int j = middle + 1;

    while (i <= middle && j <= right) // 3
    {
        if (arr[i] <= arr[j]) // 4
        {
            i++;
        } // 4
        else
        {
            int key = arr[j]; // 5

            for (int k = j; k > i; k--) // 6
            {
                arr[k] = arr[k - 1];
            }
            arr[i] = key; // 6

            i++; // 7
            middle++;
            j++; // 7
        }
    }
}

void mergeSort(int* arr, int left, int right) // 8
{
    if (left < right) // 9
    {
        int middle = left + (right - left) / 2; // 10

        mergeSort(arr, left, middle); // 11
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right); // 11
    }
}
```

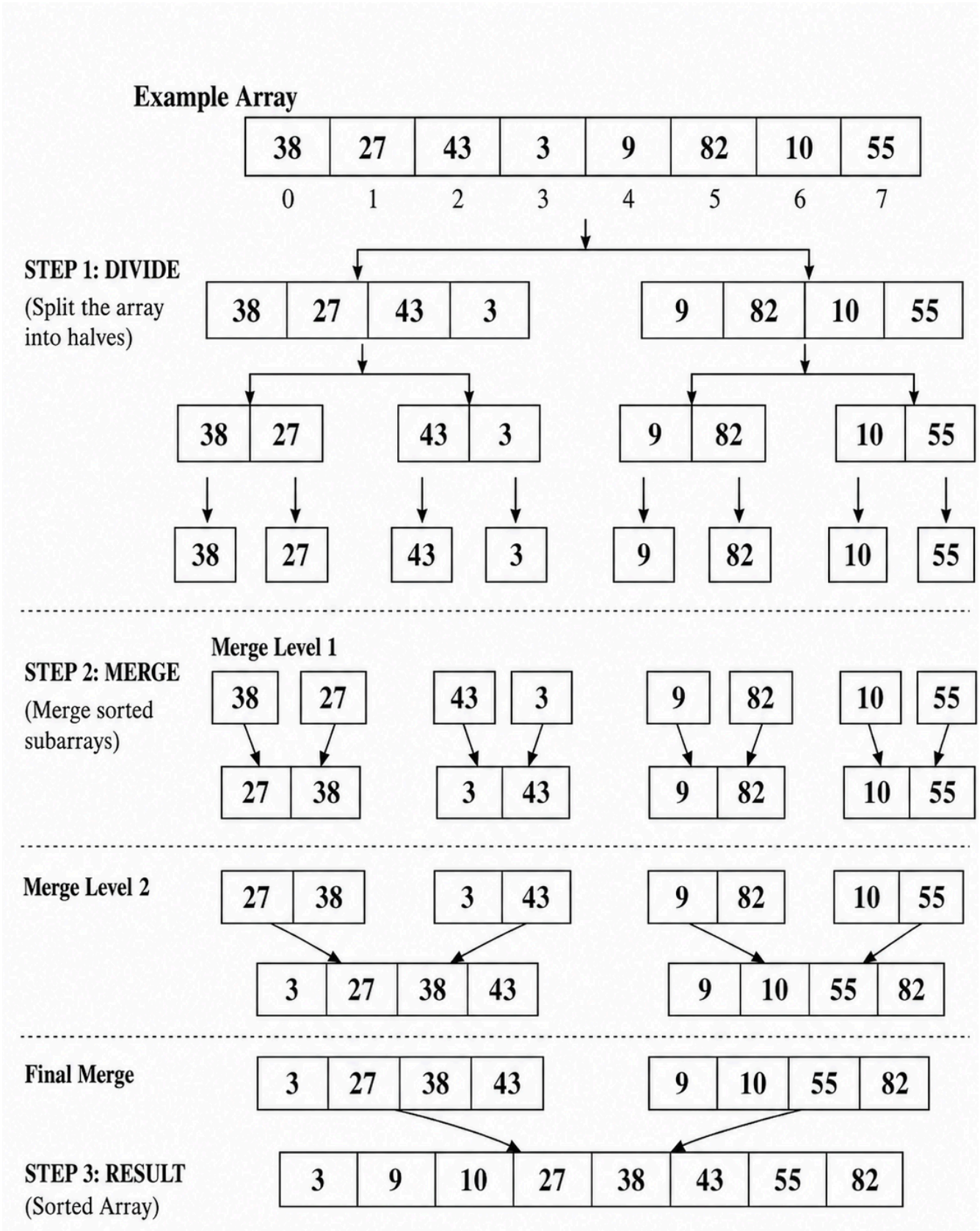
First, let's face the obvious: this is a recursive function. It's calling itself. Drawing pictures will help a lot. Let's merge this with some notes:

1. This is probably a bit of a shock in the arguments department. But, remember that we're going to be splitting and resplitting our array into halves of halves of halves and so forth. So, we need to know not the length of the array, but what index the specific sub-deck we're dealing with starts and stops. The 'mid' is actually not necessary. We could calculate it at the beginning of the merge() function. But!
    1. It's standard not to. People recognize mergeSort in this pattern as given.
    2. Including mid in the argument implies that we're going to be merging data from left to mid and then from mid+1 to right, right there in a self-descriptive function definition. Notice that we've covered the communicative aspects first.
    3. Since we know that mid is calculated on the same arguments for both the left and the right, it would be silly to recalculate it each time, when we don't need to.
    4. It provides us a little extra functionality! Imagine if we kept everything the same, but changed how mid was calculated (maybe to 1/3 of the way through instead of 1/2) our merge() function would be exactly the same! Less efficient, but the point is that we've abstracted out the calculation of mid from the execution of the function by precalculating it and letting merge() just trust that the calculation was as intended without any particular bias one way or the other.
2. We define i and j here. Couldn't we just use left and middle? Absolutely! But don't. It gets much more confusing. It's a question of scope. left, middle, and right aren't really constant values or fixed in any official sense. But, conceptually, they kind of stand as fixed values within the function. So, by not changing them, we can come back to them later if we need to. And, in this case, we will need to. We could get around that need in a couple ways, but it always just devolves into messiness. It's not a perfect rule, but it's fairly useful, unless you're specifically changing the value of an argument because the algorithm needs you to, it's much easier to just keep the arguments as fixed guides. (But, again, if space constraints demand, you might need to. It's why it's not a hard rule, just a strong guideline.)
3. This is why we need to use a while loop. We're going to be cycling through two iterators, where each cycle only one of the iterators moves. In fact, if either of them hits their limits (that is, the left iterator gets to the middle or the right iterator gets to the end) then the remainder of the other partial array is already in order. It's not worth proving, but that's why we don't need a more exhaustive setup here. It's also why we can't rely on a for loop, because then we'd need a conditional inside of the for loop, which would be awkward.
4. Just like the subdecks of cards, now we've got an array divided into parts. We're sorting this array 'in place', which means we're not using extra arrays or anything, we're pretending the only storage we have available is what was given to us. So, we have the first half of the array and the last half of the array with markers pointing to where the 'top' of the subdecks are (everything to the left, or below i, is part of the already merged portion). If i is the next smallest value, it's already the next value in the array, so we can

just move on. There are comparisons we could make to the deck of cards, but it gets convoluted and the analogy breaks down. In short, when arr[i] is the lesser value, we kind of don't need to do anything except have the left side of the array check its new value.

5. We used the variable name 'key' here to show that it's that middle spot that we're kind of pivoting around like in the insertion sort. slightly different application, but same general idea that we need to pull a value out of the array so that we can make space 'lower' in the array.
6. This should look familiar, as it's the same shifting mechanic we had in InsertionSort.
7. This is the opposition to when the left array marker is smaller. Now the right array marker is smaller and we're moving it to the left with // 5 and // 6. What this means is that the old index for the marker at i is out of date since it's been shifted to the right. And the old index for the marker at j is out of date since that value just got put into its sorted place and we need to check the next value. And then the spot halfway between them needs to increase as well since we're using that to compare when i has gotten to the end of it's left subarray.
8. Everything so far has been the unwinding portion of the recursion. But since the actual mergeSort() calls merge(), we wrote it first. However, even though mergeSort() has fewer lines, it's potentially more conceptually confusing. Notice that we're only getting left and right as arguments this time. This is because the previous iteration of mergeSort called this one to split either the left or the right half of the bigger array. It's like Inception, we're going into dreams of dreams of dreams. In this case, suppose we're at recursive level 3: recursive level 2 actually spun off two versions, one from left-level-2 to middle-level-2 and the other from middle-level-2 to right-level-2. But on level 3, left-level-2 is now left-level-3 and middle-level-2 is now right-level-3 (and in an effectively unrelated memory stack, middle-level-2 is the new left-level-3 and right-level-2 is the new right-level-3).
9. This is a bit silly. We're actually just checking to make sure that we don't have a singleton. left should always be less than right, it would be silly if it was bigger than right. Something definitely went wrong, and the best thing our function could do then is bail anyways. But what's going to happen more often is that the size of our subarrays has reduced to a length of 1. In that case left == right, and therefore we won't go into this if case. And, further, we don't need to sort the singleton. So, in essence, we're calling the function to just confirm we don't need to do anything and the call will immediately return. It's sort of like a win by default.
10. This is a standard average function. Essentially, we want to know the range between the two numbers (bigger minus smaller), we'll split that range in half, and then we'll add it back to the smaller number.
11. We could shortcut this as 'how do I sort an array? That's basically like sorting two half arrays! You want me to do two things? No way!' and then we create new functions to sort each of the half arrays, who respond in the same way until we get down to the singletons and then the analogy breaks down. But it's basically just like Futurama Season 6 Episode 17, Benderama. Alternately, we're saying 'following these same rules, sort the left half' and then 'following these same rules, sort the right half' and then 'ok, now that both halves are sorted, quickly merge them together! Done!

Example


Example Array
38 27 43 3 9 82 10 55
0 1 2 3 4 5 6 7
STEP 1: DIVIDE
(Split the array into halves)
38 27 43 3
9 82 10 55
38 27
43 3
9 82
10 55
38
27
43
3
9
82
10
55
STEP 2: MERGE
(Merge sorted subarrays)
Merge Level 1
27 38
3 43
9 82
10 55
Merge Level 2
3 27 38 43
9 10 55 82
Final Merge
STEP 3: RESULT
(Sorted Array)
3 9 10 27 38 43 55 82

Divide and Conquer

This is kind of the big name in Algorithm Design. Divide and Conquer is why every academic Computer Scientist you've ever interacted with has been excited by recursive functions. The theory that any given algorithm will work is much easier to prove with recursion when it uses divide and conquer.

The general idea is based on human behavior. There are tasks that we don't want to do. Imagine writing a term paper. It needs to be 20 pages long? That sounds a lot better if I just imagine it as 4 shorter 5 pages papers that I can stitch together. 5 pages? That sounds too long. How about 5 papers that are 1 page each? The point is that because you can conceptualize a 1 page paper, it might actually be faster for you to overall write 20 pages as 20 papers that are 1 page each and stitching them together than it would be to try to write the single monolithic 20 page paper.

Another argument might be painting a room. Suppose you have to paint a big, giant auditorium. You get given a pile of paint buckets and a bunch of rollers, and told to get to work. Now, imagine if you have a friend come and help you. There's some social aspect that will make it feel like it takes less time, sure. But it might also practically take less than half the objective time if both of you are painting together. for example, maybe one of you paints the high parts and one paints the small parts. Or maybe you split it by walls, so trips to the paint buckets are smaller. Or who knows. What about 4 people? 8 people? There's a practical limit, but with computers, there's not always a practical limit.

Why does this happen? Because sometimes we have problems that are difficult to solve. For example, sorting an array. The brute force solution basically wants us to try everything. But we tend to get bogged down by issues like permutations or exponential growth. For example, how many different ways can a set of numbers be arranged? The answer is n-factorial (n!). And that grows really fast: 1! = 1, 2! = 2, 3! = 6, 4! = 24, 5! = 120, 6! = 720, 7! = 5,040, 8! = 40,320, 9! = 362,800, 10! = 3,628,800. The issue is that only 1/n! of the permutations is the sorted array, and we have to find out which one that is.

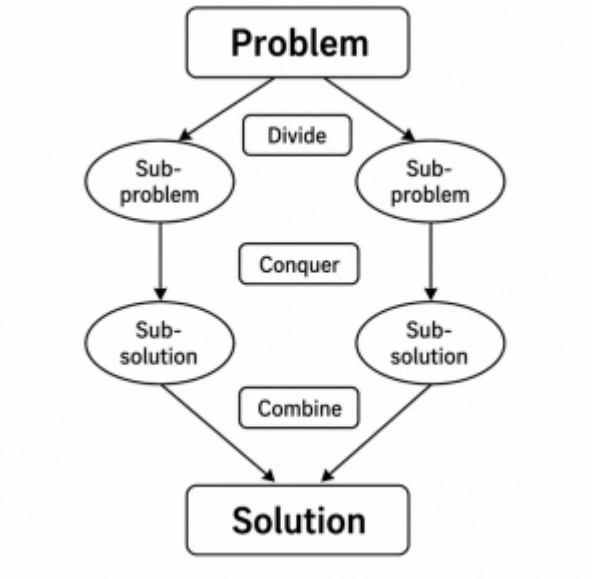
So, we add some constraints, and that's why Bubble Sort can do it in $O(n^2)$ time instead of O(n!) time. For example, we know that if any two numbers are out of order, we can increase the odds that we get to that 1 in n! sorted array by just swapping the two adjacent nodes. This is all going a bit far afield, but part of algorithm analysis and design is this process of figuring out constraints and why the constraints are there. Why is a good question.

But something funny happens with these complicated questions. 20! is 2,432,902,008,176,640,000, but what if we tried to sort the two half lists? In that case, there's only 10! permutations in each half, right? So all we have to do is check 3,628,800 permutations twice. So, we can get two half-sorted lists in 2*10! = 7,257.600 permutations. Now, we've just changed the problem, though, right? We've changed the problem from 'find the one sorted permutation out of all of the permutations' into 'take two lists of sorted numbers and smoosh them together so that the end result is still sorted'. The tradeoff we're making (it works in this case, it doesn't work in other algorithms) is that the time it takes to smoosh the two sorted lists together is smaller than the extra time it would take to evaluate all of the permutations.

This example is a little farcical. It's actually a lot farcical. But we're justifying why something works by demonstrating the extremes. And these are the extremes. We can take that pair of 10! permutations where we have to select the correct 2 from all of the permutations and change it to a quartet of 5! permutations. That's 4 * 120 or 480 permutations where we have to pick the four corret ones.

Now, again, what we're comparing this to normally is not statistics and randomly picking one. We're comparing it to Bubble Sort or Insertion Sort. And in that case, the allegory side breaks down because they don't have a common source. That's why we use things like Big-O notation (O(nlogn) vs. O(n)) since it serves as a common unit between them.

So, the point is, divide and conquer is any time you can take a problem and break it into parts, where each part is mathematically similar to the whole problem. The proof also becomes easy, because we can prove the entire process as long as we can prove the base case (typically an input of 0 or 1) and we can prove that the single recursive function works for just that step. It's the ultimate expression of proof by parts.



Divide and Conquer

Better Merge Sorts – TimSort

There's a lot of information to know about MergeSort. Too much to continue on in this chapter. But, you'll want to look at loosening some of our constraints. What if we don't have to do an in-place merge sort and we can have extra memory space applied? What if we had some common patterns that we had stored somewhere? What if the sorting needed to occur in a matrix of some sort? What if we used a linked list? What if we... the point is that there's a lot of more advanced variations that are worth researching.

One specific one we want to mention is typically called TimSort. Remember when we looked through the binary search times for different values and some of them, the 'slower' iterative search was faster than binary? That is a reflection of a general pattern. When we say that a function is larger or smaller than another, we're typically saying that 'for any arbitrary large number, the output of one function is larger than the output of

another function.' But for smaller values, it's not always true that the 'larger' function is larger or that the 'faster' program is actually faster. So, take MergeSort? All of that overhead we talked about? It isn't always as substantial when the size of the array is small.

In particular, when n < 4, MergeSort is slower than InsertionSort. So, TimSort has a call out. Instead of, in mergeSort() 'if (left < right)', it's 'if (right – left < 4) {} else { insertionSort() }'. That one adjustment to the base case practically speeds up the entire process by a substantial amount. And, it's a good object lesson in what to watch for and test lots of different algorithms across lots of different constraints.

Sorts Blending Into Linked Lists

Notice how some of the examples we gave had different implementations based on linked lists or arrays. If you don't mind a little further fourth-wall breaking, this chapter is already long enough and we don't need to do every variation for each one. However, you might want to consider how you might implement each of the four sorts using each of the Linked Lists we're going to cover here. The question you're asking is, does this add value to the process? That value can be just about anything: maybe it uses fewer resources, happens in less time, takes advantage of a specialized hardware call, provides handles for external functions to attach to, is easier to explain, is easier to communicate with, is easier to prove, is easier to maintain and update, etc. The primary requirement is that you can explicitly articulate what does or doesn't add. And the way to practice that is to try lots of variations, and then list what things you observed.

What's in a Linked List?

We have used linked lists frequently so far, but we haven't actually considered what it is. In principle, any data type needs to have five components., in no particular order:

- The structure. This is the bit with the payload and the next. It doesn't have to have just those two things, but we almost always simplify it to that level when discussing data structures and algorithms in a vacuum, because it otherwise gets complicated. The payload can be in situ (that is, in place, as a bunch of other attributes) or it can be a separate data structure with the node just being a container.
- Syntactical Sugar – This is the sweet stuff that makes the whole thing taste better. What exactly you'll need is difficult to predict. But, for example, you'll need a head node to start. Do you need a tail node? Maybe, depending on application. You'll probably want functions like isEmptyList() or a length function or anything else that lets you keep track of your list better.
- Retrieve – You need to be able to find a given node in the list and read the data in it in some manner.
- Insert – You need to be able to add a node to the list.
- Delete – You need to be able to delete a node from the list.

There is no fixed answer on this. What we've got here is the sort of statistical mean that you'll find in most implementations and theory discussions. But, for example, what about an 'edit' function allows you to change one of the nodes?

Atomic Functions

Read, Write, Delete are considered the atomic functions in almost every application from disk access to databases to algorithms. This is for two reasons: The first is that they can't be split any further, and the second is that everything else is made up of some combination of these functions.

How would you break 'read' down into a smaller aspect? 'Find' and then 'Return'? Yes, but typically in order to find a given node, something is scanning what's inside the payload, so you haven't really made 'read' smaller, you've just shifted where the read action happens. Sometimes, we might call it 'retrieve' instead of 'read', but in that case we're just changing the name not the function. We can play this same game with write and delete.

Similarly, suppose you wanted to edit a given node: you could just read to retrieve the node, delete that node from the list, and then write a new node that's the same in every way except your updates. That sounds complicated. After all: update is just one command, but using the atomic functions is three. And that's absolutely true! But remember what we talked about a moment ago about complexity and how having more of something sometimes increases complexity by an exponential amount? So, the amount of basic, atomic functions we can reduce a system to frequently allows us to reduce the complexity of any proof in that system (or, if we're designing hardware, reduce the complexity of the actual wires on the circuit board) by that same exponential reduction. Further, this doesn't change the fact that you can divide edit into smaller pieces, so it can't be atomic.

However, let's take something like Databases. The Database Atomic model is CRUD: Create, Read, Update, Delete. They actually have an edit function as one of their four, not three, atomic functions. What gives? Well, these are observations, not requirements. There's no Atomic Law & Order of two separate yet equally important groups. Instead, there's just what's expedient. And, with a lot of debate, most database management systems make the same decision: if they elevate update to the equivalent of an atomic function, then they can optimize it at the same level of the other three. And, in that case, improve the overall speed of a database. This makes a lot of sense when updating an existing database is just as or more important than creating/deleting records.

The point is, none of the rules in this section are 'this is what you must have'. In fact, to a certain degree, it extends to the entire text: these are just tools. It's up to you to decide what applies when, and where you want to use them. But, again, the actual process of programming, computer science, and algorithm design is complex enough that there is not (yet?) a grand unified algorithm.

Singly Linked List

Let's quickly go over 3 different types of linked lists. The first is a singly linked list. We've already covered this, but let's give it a little more exhaustive treatment. Note that obviously the 'int' data type in all of these examples is kind of silly, but it's the easiest to present the overall concept.

```
struct node {
     int payload;
     struct node* next;
};

struct node* head = NULL;

struct node* read (struct node* head, struct node* target)
{
     struct node* curr = head;
     while (curr != NULL)
     {
          if (equals(curr, target))
          {
               return curr;
          }
     curr = curr->next;
     }
     return NULL;
}

void write (struct node* head, struct node* target)
{
     target->next = head->next;
     head->next = target;
}

struct node* delete (struct node* head, struct node* target)
{
     struct node* curr = head;
     while (!equals(curr->next, target))
```

```
        {
                curr=curr->next;
        }
        struct node* ret = curr->next;
        curr->next = ret->next;
        return ret;
}
```

Let's go over this quickly:

- We've got the node structure. Obviously, since it's a singly linked list, there's a single node. The payload is unimportant without context, but it can be anything.
- There's a head node, because that simplifies everything the most. The question remains for you to answer: suppose add a tail node. What does that benefit us? (this is not a rhetorical question)
- The read function is simple, it finds the node using an undefined 'equals' function that is a way to abstract out the details of figuring out whether two nodes are equal to each other (remember the difference between shallow and deep copies, and when we say ptr1 == ptr2, it'll be false becasue the pointers themselves are different, and there's no instructions for determining equivalency in the value the pointers are pointing to.
- The insert function is simple, it just inserts the node. If the list isn't ordered in any way, then it doesn't matter what the head is, and we could just make the new element the head! Except... then we'd probably want to return the new head to be sure that the calling function is using the new head. But, if we want to separate the form and the function and it doesn't matter where the new node goes, we'll just do the fastest element that retains the current head. Not because the current head is important, but just from expedience (speaking of expedience, this code isn't valid, it's missing null checks, among other things, in order to highlight the specific actions that are happening. How much does it extend the code to add those checks? How much does it extend the runtime?) This adds an important question: how much overhead is taken up by inserting the node in the correct place for the list to remain sorted, and is it worth maintaining a sorted list or occasionally sorting one? There isn't a right answer, but the answer that there's no benefit to maintaining a sorted list probably needs more justification than otherwise.
    - Side Note: Notice that we aren't always aligning our terminology. Here we say 'insert', but in the code, we said 'write', and elsewhere we say 'create'. It's sort of like in math: do you use a *, a x, or a · to indicate multiplication (and do you call it product, times, or multiplication)? They all mean the same thing, and there aren't math police that are going to arrest you for using the wrong symbol. Your metrics are clarity of purpose, ease of use, and only occasionally remembering that some things don't work in certain contexts (for example, printing a * in a situation unrelated to CS will make mathematician's raise an eyebrow, since * is shorthand for · when we didn't have easy access to printing ·). But, mostly your emphasis should be on problem-solving and knowing everything

available, not adhering to narrow definitions that are rarely universal.

- The delete function gets tricky. notice how we have to introduce an extra bit there, we need the ret function because otherwise, we lose the deleted node. We're returning the deleted node for two reasons. We normally free() the deleted stuff and we want to show that sometimes you pass the responsibility of free()'ing a pointer to the calling function. Secondly, we frequently don't really mean 'delete' as in 'forever', we mean 'delete' as in 'remove from this context' because, for example, you want to put it into a different linked list.
    - It would also be good practice, in practice, to set ret->next = NULL since it's no longer tied to that linked list. It's not necessary, since we're deleting it and don't care. But just to tie up loose ends, it falls under the category of correctness more than necessity.

Doubly Linked List

In this case, when we say 'doubly', we typically mean backwards and forwards, although there are other options (for example, skip linked lists are very common).

```
struct node
{
     int payload;
     struct node* next;
     struct node* prev;
};

struct node* head = NULL;

struct node* read (struct node* head, struct node* target)
{
     struct node* curr = head;
     while (curr != NULL)
     {
          if (equals(curr, target))
          {
               return curr;
          }
          curr = curr->next;
     }
     return NULL;
```

```
}

void write (struct node* head, struct node* target)
{
     target->next = head->next;
     head->next = target;
     target->prev = head;
     target->next->prev = target;
}

struct node* delete(struct node* head, struct node* target)
{
     struct node* curr = head;
     while (!equals(curr, target))
     {
          curr=curr->next;
     }
     curr->prev->next = curr->next;
     curr->next->prev = curr->prev;
     return curr;
}
```

Let's take a second look at this:

- We add one extra pointer, and this one is to the previous elements in the list. We're going for brevity, and trust you can add the checks for null and empty lists on your own, because they're tangential at the moment. Notice that by adding another pointer, we've effectively increased the size of the node by 50%! But, that's just because our examples are all int. In practice, the node would be some more elaborate data structure, maybe kilobytes in size. For example, maybe the node is a page of text along with all of the meta data for that text. In that case, maybe being able to flip forward and backward quickly is useful and increases the overall speed of the program even though the algorithm doesn't need it to function. In that case, adding a couple of bytes for a prev pointer is trivial compared to the time savings.
- We can still get by with just a head node. We can always find the tail by traversing the ->next links. But, and this question is even less rhetorical than last time we asked: Is there any benefit to having a tail pointer to counter the head pointer in a doubly-linked list?
- Notice that the read is exactly the same. The biggest benefit is that when finding a given node, it's much easier to find the previous node than before. In a singly-linked list, to find the node before a given node, you'd need another function that's exactly like 'read', but iterates over equals(curr->next,

target), not equals (curr, target). So, the tradeoff is whether it's worth it to use a little or a lot, depending on application) extra memory in order to not have to run a new linear search for every new attempt. Remember to add null checks.

- The Insert node just needs to compensate for the fact that prev is there. As a side note... if we are adding the node to some random location (random in this sense in that the second node of the list is not of specific importance), why not add it to the end? Is there a simple way to do that, such as with a special tail node that acts as the opposite of head? What utility might there be in that?
- Deleting is much easier with the prev. Although, we do get the cumbersome double pointers. We could avoid that by assigning some temporary values. However, being able to follow double pointers is a pretty common expectation. More than that is a bit gauche. But while good style suggests avoiding double pointers, simple situations like this are still fairly self-descriptive.

Circular Linked Lists

```
struct node
{
     int payload;
     struct node* next;
};

struct node* head = NULL;

struct node* read (struct node* head, struct node* target)
{
     struct node* curr = head;
     while (curr->next != head)
     {
          if (equals(curr, target))
          {
               return curr;
          }
     curr = curr->next;
     }
     return NULL;
}

void write (struct node* head, struct node* target)
```

```
{
     target->next = head->next;
     head->next = target;
}

struct node* delete (struct node* head, struct node* target)
{
     struct node* curr = head;
     while (!equals(curr, target))
     {
          curr=curr->next;
     }
     curr->prev->next = curr->next;
     curr->next->prev = curr->prev;
     return curr;
}
```

We'll just keep going around and around with these same notes:

1. The structure of a circular linked list goes back to the singly linked list, because the use case for a circular linked list and the use case for a doubly linked list are relatively small, but the overlap between them is even smaller. What's the most striking component here? There is no NULL pointer, so we can just keep traversing around the list forever. In fact, consider a game where the number of monsters on the screen is so many that they can't be processed every frame. Instead, the game is designed to process as many as it can in the time it has alloted to updating monsters, and then it stops. In that case, it would be convenient if the new head was already the next unprocessed node.
2. Head is not really the first element of the list. Well, it is, but since there's no NULL termination, the entire list rotates and the 'head' is just where we start, not the beginning. The wheel of links turns and heads come and pass, leaving prevs that become tails. Tails fade to generic nodes, and even those generic nodes are almost forgotten by the time the head that gave it birth comes again. There are neither true heads nor tails in a circular linked list, but this was a head. (The editors complained: this is a parody of the iconic opening line of The Wheel of Time series.)
3. The biggest difference with the circular linked list is that we don't know what the appropriate stopping point is. We either have to leave a marker behind or we have to move the marker until we hit the head again. Either way, it doesn't really matter, because where the head is doesn't matter in a circular linked list, just that the structure exists.
4. Similarly for write, it's exactly the same. If it doesn't matter (that's a good question: is the assumption that it doesn't matter where we add the new node to the linked list ever valid? Is it generally valid? Is it

valid in a specific application?), we can just insert it the same was as a regular linked list.

- Note: if you've tried using any of this code so far, you'll notice that you've run into a problem: the code doesn't work! The code as written presumes the linked list already exists. But it doesn't handle when there is no list. How would you modify each of these in order to insert the very first node? You'll likely need: head->next = head and/or head->prev = head.

5. For delete, we run into a couple of issues that we haven't brought up yet. What if the node isn't in the list? And what if we're deleting the last node in the list? Again, how would you modify the code to handle those two situations?

Circular Doubly-Linked Lists

You have all of the tools you would need to write a circular doubly-linked list. These are not uncommon, for what it's worth, when linked lists are used. But the question you need to ask yourself towards the end of every chapter is not 'can I write that', absolutely you can! You can write anything. the question is 'should I write that?' So, before you write a circular doubly-linked list, come up with a use case and explicitly describe why it would be valuable to make a) a linked list, b) a doubly-linked list, and c) a circular doubly-linked list. Your solution is likely at least a little contrived, which is fine. But to the extent it's feasible, it would also be good to explain how it justifies the tradeoff in time and complexity.

The point is not that any given solution is bad or good, it's always a question of tradeoffs. You do not want to be able to quote a thousand solutions, you want to be able to solve a thousand problems. That involves understanding the 'why' behind the structure and applying that understanding more so than relying on rote memorization of use cases. We hire programmers to regurgitate code. We hire computer scientists to solve interesting problems. (*cough* and we hire MIS majors to install software they bought from someone else *cough*)

XOR Lists

It's also important to point out that just because a data structure exists doesn't make it useful. Side tangent: one of the most frustrating aspects of working in technology is the clueless CIO. Not every CIO is clueless, but frequently it is true that they've been in the industry long enough to no longer be on the cutting edge or only truly understand a part of their domain. And the result is that when they go to conferences or trade shows, they're easy to convince to buy a program with large promises of effectiveness, because they're in an authority decision-making role and willing to commit company to solve a problem they didn't know existed. This is

a legitimate problem, and why some technology initiatives are constantly being redirected through different technology stacks.

This also applies to the lowest level. Just because an algorithm is clever doesn't make it good. Neither does the fact that someone is trying to promote the use of a given data structure or algorithm type mean anything further than they have a vested interest in making that algorithm popular (maybe it's their academic focus, for example). XOR lists are an example of this. In an XOR list, we have a doubly-linked list, with a prev and a next. Except that there's only one pointer. And the address of that pointer is ->next XOR ->prev. This means that the overhead of having a doubly linked list is removed, since we only have one pointer! Genius! It's basically a free conversion from singly to doubly linked lists!

Except... there's never anything free when we get to the low level side of things. In this case, it means that we never actually know ->next unless we happen to already know ->prev. And we can't find ->prev unless we already happen to know ->next. And it means that every time you want to traverse the list, you have to run an XOR operation. So, really, it's not 'free'. It's 'free' if you only pay attention to space constraints. But the tradeoff is in the amount of computation time that it takes. (which, might also reflect in space, since there'll need to be a temporary place to store the results of the XOR, but that brings us into the weeds of the analysis).

The short of it is that the extra complication is almost never worth it. In reality, when space is a constraint, computation is probably a constraint (not enough room for memory means not enough room for transistors). The extra code required to handle the XOR is also a consideration. In short, except for in very, incredibly niche scenarios an XOR list is never actually useful. It's complicated and 'elegant' solely for the sake of being complicated and elegant, not for the practicality of it. (And, arguably, even in the cases where it might technically be superior, it's such a niche and fragile solution it may still not be an ideal application).

The point of all of this is to test and verify utility. When it comes to the communication and maintenance side, there's a significant 'gut instinct' aspect. You should be able to explicitly describe why the code is less maintainable or elegant, but if it seems too complicated to really be useful, you should track that feeling down (even if you're wrong, it would be good to know why and where you misunderstood).

Practical Applications

Linked Lists are never really used in industry. Here's where they're used in industry:

- Low-Level Systems Programming. At the very lowest level, we kind of don't trust arrays. We want small, modular, testable data structures. This is doubly true when the data will be written to and read from or changed in irregular intervals. So, for example, device driver lists, or interrupt stacks, or processing queues (wait, no, that last one is not exactly a linked list. We'll get there shortly, though). These are highly reviewed, highly engineered, very low-level system functionalities. You'll want to brush up on your Linked Lists any time you're going to study, talk about, or practice operating system design.

    - The common element here is, really, the distrust of an array when the operating system can't know at compile-time (at the Linux Headquarters or the Microsoft Secret Lair) what hardware options your individual computer is going to have combined with the peculiarities of memory addressing when it comes to hardware and interrupts versus the need for the operating system to track them.
- Undo systems, or iterative change patterns. For example, when you push ctrl-z three times to unwrite the last three lines you wrote, the application almost certainly has a linked list that's handling this. (Alternately, the operating system provides it as a function of the initialization of a program.) Saving something as an iterative change log is interesting. Essentially, imagine you're working on a document that's not just large, but super large. Think thousands of pages, and your job is to go through and change all of the indeterminant 'him' and 'her' to a gender neutral 'bro'. Saving the document every time you make a change seems like a lot of churn. Instead, the document before you started making changes would be saved, and then there'd be a ->next pointer that points to a data structure containing the location and content of the change (delete a word, add a word). After you've processed the whole document like this, then you can apply the iterative changes into the original document in a recompilation or remux process.
    - The thing that these have in common is that we don't know how many changes are going to be made, so we can't allocate an array without either wasting a ton of space or running into regular problems with content limits.
- Concurrent Data Structures is a bit of a misdirection. Linked Lists are prominent in the authoritative texts on when multiple programs are acting on the same data, or multiple threads are working in the same program or multiple users are trying to edit the same document. Anything along these lines may make use of linked lists rather than arrays. This is not because the array is impossible, so much as that the established convention is to use linked lists, based on the literature and previous examples.
    - The key idea here is that since the linked list nodes are separate data structures, and the array is a monolithic data structure, using locks or flags to designate individual nodes as being in use by a different actor is cleaner and easier than having to flag the entire array.
- Functional Programming uses linked lists a lot for their internal representation. The reason for this is because, despite occasional renaissances, functional languages are kind of boojie. It's for academics or for proofs that are going to take so much computing power that the lack of efficient low-level structures is not a problem. This is ironic that both extremely low level and extremely high level programming tends to like linked lists.
    - The key idea here is the provability. Functional languages are typically used to be able to prove new ideas or to otherwise provide a mathematically rigorous environment. And the internal representation reflects this, as the recursive nature of the linked lists are much easier to prove.

Wrong Thinking

As one last example, let's engage in some wrong thinking. But this is the sort of wrong thinking that employers

want to see. MP4s do not have linked lists. But MP4 does contain different video codecs. (It's complicated: MP4 is a wrapper that neatly associates all of the stuff like video and audio and subtitles that all use their own standards.) Most modern video codecs encode the video because raw video is incredibly expensive. For example, the 3 seconds at the beginning of a movie where it's just a black screen before the credits roll at HD would be 60 frames at 1920 x 1080 (that is, 1080p) would be (1920 cols * 1080 rows * 3 seconds * 60 fps * 3 bytes per RGB pixel =) 1.12 Gigabytes,. That's just for a black screen and that's just not tenable. Instead, the way that most video formats work is to have occasional key frames and then it encodes which pixels change between that keyframe and any given keyframe.

More keyframes equals more data storage but faster processing, fewer keyframes results in smaller sizes but more processing time to display.

Doesn't that suddenly sound a lot like the undo linked list or the iterative change log? We used a linked list to represent that! (Again, a reminder, there's no linked list here, this is not about the linked list, it's about the problem solving). What could we use from the linked list world that would let us make better videos?

What about doubly linked lists? Doesn't it make a lot of sense that if we have a keyframe every, say, 8 frames, we could represent any non-keyframe by how each pixel changes from the previous keyframe but also from the next keyframe? The algorithm of how it does that is what differentiates, say, x264 from x265 from av1. But the idea, broadly, has a lot in common with linked lists. So, with the caveat that you can't force the analogy, that connection leads to the introduction of bi-directional transition frames (b-frames), which leads to a huge file size reduction. We can tell this, because when people are recording their videos, it obviously can't refer to the future keyframes, since they haven't been recorded yet. Instead, it only has the previous keyframes. And the file size and processing time both get reduced by re-encoding the video once the entire videos been encoded.

The point is, that employers don't care that you know all of the details of the algorithms off the top of your head. They want to know that you've got the tools available that you can use to make disparate connections and apply that problem solving to the problem at hand in a way that is better than the off-the-shelf solution.

AI Examples

Let's consider how AI systems work in practice at the highest level. Because AIs run their code across maybe trillions of calculations, it's going to be most effective as low level code. And that means algorithmic thinking and application. But, there needs to be some high level principles applied, too so that the code as a whole is working synergistically instead of at cross purposes. So, while this isn't about the specific data structures at the low level, it's about the philosophy behind their choices. We want to highlight two areas that are super important: hybridization and modularity.

Before we get into that, let's split up the training cycle (we're looking at each cycle, not the whole product) of an LLM into the following phases:

1. Parse the training input
2. Run the transformer on the training input
3. Compare model predictions to training input
4. Adjust weights in response to the correctness of the simulation

This is purposefully non-technically framed, because we're going to make our two points here in a moment.

Hybrids

Most of the algorithms we cover have pros and cons. Not only that, but because algorithm design is frequently performed in a theoretical environment (like when we teach physics: most early concepts are explained in a vacuum on a frictionless plane), those pros and cons can be evaluated in extreme detail. This means that it's very rare for there to be any situation in which 'you should always use such and such basic algorithm' is a fair statement. Instead, it's almost always the case that it's worth spending some time creating a set of classes, and each class can use a different algorithm based on which is most efficient or effective for that class. We've already demonstrated this with one example: mergesort is typically never used in a pure form, instead it's almost always mixed with insertion sort below a certain threshold.

AI does the same thing. Each of the four stages outlined was broken down that way to demonstrate a given model that might be used. For example:

1. There are a number of different tokenizers that we might use. But, depending on the application, we might not want to use just a single one. For example, some tokenizers are better at identifying individual words or dropping out lexical grease words (stop words or low-information density words) or identifying proper names, etc. We might even run several different tokenizers in parallel to see which one produces the most effective output.
2. Transformers are typically focused on converting the words into ideas. How they do this is varied. But we could give two extreme examples. Consider what would happen if someone was trying to train an AI on slam poetry as well as on mathematical functions (maybe they were shelved next to each other in the discount bookstore that the training dataset was purchased from). There's a huge difference in the emotional weight of one versus the other. Wouldn't it make more sense to use different types of transformers? Or, alternately, wouldn't he math version need to eschew the transformer entirely and just run an analytical engine or some sort?
3. When we use say we're going to compare model predictions, which model is that? Broadly, it's the neural network, but could we have several neural networks running at the same time and choose the one that's the fastest, the most accurate, or results in the highest fidelity from the users? How do we compare the prediction to the actual? In statistics, it's typically done with residuals, but for example, those residuals might be L1 or L2, they might only be across one dimension. Which one is most effective is going to be

the one that, through testing, leads to the most optimum outcome.

4. Finally, we typically call the process of updating the weights in the neural network the process of Backpropagation. But, that's not the only way (to be fair, it's almost always the best, so much so that it's almost axiomatic at the moment), and the specific mechanic to update the weights? This is where the different models come into play, maybe they use adaptive gradient clipping, maybe they use 1bit methods, maybe they use some sigmoid function instead of backpropagation altogether, etc. But more importantly, the entire neural network doesn't need to use the same backpropagation method. Once we've started to differentiate different parts of the neural network as different foci, it makes more sense to begin to apply different weight update algorithms that are more effective for different tasks (describing a picture versus creating anew picture versus a quantitative analysis of a picture, etc).

The point is that there's rarely a time when an algorithm can't be improved by introducing different algorithms and then applying it in various ways. When AIs are running on the tightest of margins and are competing with each other and their own previous iteration, eking out every small advantage is useful. Normally, we might say that the increase in complexity of the code was an important consideration, after all twice as much code results in twice as much code maintenance. And that's true with AIs, but the code base is typically so technical that adding extra technical code is an expected cost.

Modularization

You've noticed that we keep separating as much as we can. This is a design principle that we would generally associate with OOP, but it applies elsewhere as well. In general, we want to design a situation where we don't care what's 'under the hood', as it were. But we could swap out each method and no other function would notice. For example:

1. Maybe we don't use a tokenizer at all, or maybe we do. Maybe we try several models or maybe inside of the tokenizer, we consider different sorting algorithms to see whether any apply elsewhere as long as the output is the same, it won't matter!
2. Transformers are a relatively new idea. It's what sparked the current rush forward in AI technology. A lot of it was old concepts, but the paper 'All you need is Attention' essentially gave everyone permission to use those old concepts and just pour more processing speed into them. What about Natural Language Processing, then? Could we, instead of a transformer, incorporate an NLP algorithm of some sort? Could we use them both in some hybrid approach? The point is, if we have designed our program modularly, we could just swap the functions in and out and the rest of the program would work as well.
3. For the model predictions, the entire model can be tried. In fact, every iteration is a separate model, so we want to separate it out in such a way that we don't need to worry.
4. Part of testing an AI is trying lots of different A/B testing on several iterations of different weight

updating methods. But that means that we want to separate out the concept of there being a set of weights in the first place with the method we use to create those weights.

What does all of this have to do in practice? It is almost pure improvement to introduce modularity into code that is not effectively modularized. This is doubly so in something complex and theoretical like the backend of an LLM trainer. But, it also provides you with a context for finding improvements. You can write new functions for each of these modules and compare short term and long term gains or losses in using the new approach in various situations. Know what different algorithms are available for each of the main modules in a large scale project like this is vital!

Scope

Lastly, this is all subject to scope, like anything else. That scope can go all the way down to the individual functions being swapped out for other functions or hybridizing between different outcomes in each iteration of a loop. Or, it could be carrying several different overall neural networks and trying all of them at the same time to determine which results in the most customer satisfaction during an interaction . This latter is called a/B testing and most online tools use it all the time. for example, YouTube lets creators A/B test different thumbnails for videos to see which ones perform better. And this goes all the way up to major websites running two versions simultaneously to see which one gets more compliments and/or fewer complaints.

Media Attributions

- bubblesort-1
- insertion sort
- MergeSort
- divide_and_conquer_diagram

CHAPTER 6 - YO DAWG, I HEARD YOU LIKE STACKS!

Let's review what we know so far:

Data Structures and Algorithms are intertwined together. Data Structures are the "things", and algorithms are the "actions". When we have a single atomic piece of data, we call it a primitive data type. When we combine different pieces of data all smooshed together, we call it a data structure. Similarly, a single instruction is a statement, but a group of instructions that meet certain requirements to solve a problem is an algorithm.

Mostly, we've covered the C programming language. Why? Because while we can write algorithms in any language, most of those are routine programming tasks. The actual science of computer science is concerned with proving algorithms from a mathematical basis. While we need to know that such proofs exist and understand how to apply the algorithms elsewhere, we don't need to focus that much on the mathematical proofs side (yet). The C language, however is most effective in the current environment at low level programming, and algorithms as we're studying in this text have the most effect on low level programming.

Lastly, most of what we're looking at are established proofs that have been made for decades, so the important part is not reinvent the wheel but learn when to use existing algorithms and how to apply them or adapt them to problem we may face in your own programming.

So far, we've looked at several type of linked lists, which are a sort of emergent behavior. We, as humans, see the idea of the linked list that exists, but really the data structure is just the node. A node is the combination of a data and a pointer to another node (or some variation thereof). The actions that occur typically just happen at the node level. This is contrasted to the regular array data structure, which is just a monolithic block of memory that associates several of the same data structure in sequence. Neither is 'correct', both are 'correct', it mostly matters with regard to what works best in a given situation.

We've also covered several broad algorithms, but the things we've given the most time to are sorts and searches. These are the first entry into the study of algorithms for most treatments, because they're simple, well-researched, illustrative of a number of concepts, and easy to both memorize and adapt to different circumstances. The small divide we have between different algorithms is what design methodology they represent (a misnomer, the design methodology applies to the actual design, not to classifying them after the fact, however we can learn in a pedagogical perspective by trying to classify them after they're already designed). The big divide we have, so far, between algorithms is whether they are bad algorithms (exponential time), regular algorithms (polynomial time, such as $O(n^2)$, $O(n^3)$, or even just $O(n)$), or good algorithms (typically some variation of logarithms). Binary Search and Merge Sort are the best example we have of the latter so far.

Let's briefly step aside and consider one characteristic of arrays and linked lists before we change it to see what happens.

When we have an array, we have a data structure that we call RAM, that is Random Access Memory. This means that it takes exactly the same amount of time (O(1), or some constant amount of time) to access any address as it does any other random address. This is a more complicated topic than just one side paragraph, but the idea is that we want to be able to access any part of the array and being able to do so quickly is important. A linked list, on the other hand, is called SAM, that is Sequential Access Memory. This means that it takes the same amount of time to reach the next adjacent chunk of memory (O(1), or some constant amount of time), but that it takes more to access memory locations that are further away. The idea is that we can still access all of the memory, we just don't worry too much about being able to do it quickly. Notice that the common presumption we made was that we want to access all of the elements of the aggregate data type.

Let's continue on and look at some other basic data structures and then some algorithms that utilize them.

Stacks and Queues and Stacks and Queues and Stacks and Queues

So, in an array or a linked list, we want access to anything that the data type has available. That's actually a much narrower situation than we'd expect. Think about the day of a mega-CEO: do they keep a mental copy of their entire day broken down into 5 minute blocks, or do they only worry about what's happening next and trust their executive assistants to manage the rest? The answer is certainly the latter. There are a number of situations like this, where we don't need to know everything, we just need to make sure that everything is recorded somewhere, and then we only need to worry about the next thing that's happening.

There are two classic examples of this, but first, let's make a bunch of nonsense words:

FIFO, FILO, LIFO, LILO

We've established that we don't need access to the entirety of the data. We still want to be able to read, write, delete values (although, we may rearrange what that looks like, implicitly) as they're still the atomic operations of note. The term we might use for this is the edges of the data. Everything in the middle is there, but we can only access it through the edges. What's the edge of a line (presumably you've been imagining arrays and linked lists as some form of a line. If you have been imagining them as something else that's cool, this just doesn't really apply. If you haven't been imagining them at all, it might be good to at least draw out pictures to visualize data structures), except for just a single point? In fact, we could also call these the endpoints. However, distinguishing between the middle and the edge also has some longer-term benefits for figuring out what parts of data you might need in other circumstances. The commonly used term for stacks and queues specifically is the end point.

So, imagine one of two scenarios: you're working a deli counter or you're working a trauma unit in a triage center. Yes, that did escalate quickly, but the connotation is also sort of accurate. For the deli counter, customers come in and form an orderly line (in the UK, the term is actually 'queue' for an American line, it's where the term for the data structure comes from). The person who has been there the longest gets served first. Then they leave, and the person who's been there the longest gets served first. Then they leave, and so on. The fact that they're actually physically standing in a particular arrangement is just a rough indicator of the conceptual idea that fairness is to serve people in order of arrival.

Contrast that to a triage area. Let's structure it so that one doctor is responsible for classifying patients in order of who gets sent where for further treatment in order of severity. We're not worried about that part (although, that would make an interesting data structures question and project, wouldn't it?). We're only worried about this one doctor's role. The goal is speed and urgency, and let's pretend that the classification is almost instantaneous. What does that doctor do with the rest of their time? They work on the patient in front of them. Why? Because running back and forth between different patients takes time. There's a triage team bringing patients in and moving them around. So, as a new patient comes in, the classifier moves over to the new arrival, makes their classification and treats that patient until a new patient comes in. If they finish treating a patient or it gets moved to its destination, the doctor moves to the next most recent patient in the line. There's always work to do, but the kind of primary interest is: the most recent thing comes first. Other people will focus based on how long they've been waiting or based on severity, but that initial panicked handling happens based on recency.

So, in the deli counter, we call this structure First-In-First-Out. that is, the long someone has been standing in queue (that is, the first one there), the sooner they'll be handled (that is, the first one handled) and booted out the door to make room for the next person to be served. The triage unit–slightly contrived to highlight the example–is called LIFO, or Last-In-First-Out. That is, the most recent one needs to be handled the quickest, because for any of a number of reasons, it's cleaner to just handle what's come in most recently. Maybe older stuff ages out, or maybe it's just a question of getting even a little bit of processing done, or any of a number of other issues.

This is not a rule, just an observational guide, but queues tend to be useful when consistency and orderliness is prioritized. Stacks tend to be useful when speed or recency is most important.

But, obviously, we've seen F's and L's listed, why didn't we address FILO and LILO? It's partially just a marketing thing. What does Last-in-first-out mean? It's the same as a stack: the one that's been there the longest will continue to be there longer, but now we've focused on the moldering bottom end of the stack rather than the fresh top end. LILO is the same, we're focusing on the slowest part that we're not actively working on, not the idea of what's next.

As an aside, this is also great life advice! If you can break a problem into lots of individual tasks, you can order them and then either take each one in turn, only worrying about the next task, or you can sort them in the order they came in and handle the most recent one (presumably the most interesting) based on recency. Either way, focusing on the next concrete action step is an excellent approach to completing large projects.

Pop and Push

When we're playing with stacks and queues, we typically use two words: pop() and push(). Think of it like a can of Pringles or one of those rolling soup can holders at a grocery store. We pop off the top Pringle or we pop out the next can. When we push a chip (this is why it's not a great analogy, but it is fairly common) back into can or pushing a can of soup into the loading part of the soup can rack. Other examples:

- Stacks
  - Receipts put on one of those paper spikes in old movies
  - Cans of tennis balls
  - Water in a jug (although this is a little murky)
  - A stack of wood to be processed that likely built up over time, and then is processed through from the top
  - A stack of dirty dishes.
  - Typically anything with only one end point that acts as entry and exit.
- Queues
  - Cars on a train arriving and departing a station.
  - Scenes in a movie
  - Orders piling up in a kitchen
  - Watching a movie series marathon, probably
  - Milk cows being moved through the rails into the milking machine
  - Typically anything with a separate entry and exit point

The key point is that regardless of structure, we abstract away inner workings of the data structure and only focus on the fact that pushing into/onto (both are used regularly) a stack or a queue means adding something to the data structure. popping off of or out of a stack or a queue means removing the next thing from the data structure.

What this means is that while we'll probably still have the three atomic functions of read, write, and delete for our debugging/administrative use, the atomic functions we expose are simplified: pop() and push(). pop() will just be to read the next value and then delete it, push will be to take a value and add it into the appropriate end. This is, in a lot of ways, the reason why stacks and queues are popular, processing speed increases dramatically because there are so many fewer possible operations.

Stacks

We cover linked lists before stacks, because stacks are easiest to conceptualize as a linked list. Whereas in a

linked list we're worried about traversal, a stack is relatively easy: we only need to worry about head. If head == NULL, we have nothing left in the stack. If head != NULL, then head is the only option to get handled next. We don't need to worry about how long the stack is (within reason, the actual physical memory of the device will end at some point), because all of that is abstracted away. (Recall that when we say 'abstracted away', we are just hand waving away some critical question and declaring it unimportant.)

What's interesting about this is our node structure. You should be familiar with this by now, but here's the node for a stack:

```
struct node
{
     int payload;
     struct node* next;
};
```

And, we have a few extra functions we need to make it work:

```
void stackPush (struct node** head, struct node* target)
{
     target->next = *head;
     *head = target;
}

struct node* stackPop (struct node** head)
{
     struct node* ret = *head;
     *head = (*head)->next;
     return ret;
}
```

Notice that the code is incredibly short. By limiting the options for how to interact with a stack, we've reduced all interactions to constant time. There is no O(n), no for loops, no searching, no sorting. The only thing we do is push and pop from the stack. There is a lot of power in that, if the model works for the use case. Stacks are popular for exactly this reason: their simplicity. Further, because they're such a simple structure, they're easy to error check on and easy to design around.

But obviously this can't be it, right? There's got to be a tradeoff in some level. The obvious one is the constraint that was placed on it: we can only access the endpoint, and that endpoint is LIFO. If that doesn't apply, we can't use it. The other situation where this isn't always useful is when there is a space constraint. We don't control, as written, what happens when we just keep adding more nodes onto the stack. For example,

NASA would never approve code in a probe that used this, because as the processor degrades over time, there's nothing stopping the sensors from pushing onto the stack faster than the processor can pop and process the nodes, leading to a stack overflow. What is a stack overflow? Nothing in particular, it's the generic term for when the size of the stack is too big for the space that it can fit in. Maybe that's allocated space, maybe it's the full memory, maybe it overwrites something critical, maybe it just does a null-pointer error as mallocs() return NULL. This latter issue is easy to solve from a technical perspective: just don't allow it to grow above a certain length and test for NULL. But, how the program itself should handle that situation from a higher level is the actual security-conscious programming question: sure, the stack refuses to grow above a certain size, but what should the application do with the node it was going to create?

The answer is always application-dependent. For example, in network devices, it would seem like a queue would be appropriate. But the network devices are normally trying to satisfy as many connections as possible. If they went with FIFO, and their queue was being pushed to faster than it could be popped from, eventually every connection would be served out of date data. But by using a stack, it can be sure that it's always serving the new stuff to its intended destination as quickly as possible and those connections will be happy, while the old stuff where the connections would be upset will just continue to get older but the connections won't get any more upset. This also solves the problem of what to do with a stack that gets full: just delete the oldest stuff, it'll never get served anyways and it might be out of date by the time the network device gets to it.

That's just one solution and one application, the actual implementation needs to be considered in terms of what works for that environment, what the industry or application standard is, and what conventions might apply to the data types. Notice how with many of the data types, we're immediately suggesting what to do when things fail. It's a poor plan to expect failure, generally. But, on the flip side, if everything was easy and nothing failed then we wouldn't need data structures in the first place. To a certain extent, the mere presence of data structures implies that a better way was needed.

There's one thing that's probably got you scratching your head: What's with all the ** and (*)'s?

Pointers to Pointers

When we pass a reference to a function as an argument, we are passing it like this: void foo (int* x). This means that if, inside the function, we do *x = 8, then the original x from the calling function will now change to 8 as well. So far, we've just been treating pointers as pointers and interacting with them as such, for the most part. But it's important to remember that we can actually adjust the values themselves using the dereferencing operator: *.

That's important to note: it's what we use to denote when we've got a pointer. But it's also an operator in its own right. so, when we've got int** x, we're saying that we have a pointer to the pointer to x. That sounds silly, but what if we want to pass a pointer such that the pointer itself can change to a function? Then we'd need: void bar (int** x). And now, in the function if we want to do the same thing as before, we'd need to use

**x = 8. But, we could also do something like *x = someOtherPointer, and the original pointer in the calling function would now be pointing somewhere new.

This is, generally, to be avoided. In fact, you should go through a lot of permutations to avoid double pointers like this. It is very common in systems level programming, so it's not a forbidden art or anything. However, it is confusing and more difficult to read. There are situations where there are no other options or the other options are even less clear. But creating both solutions and evaluating it is probably the best solution there.

Finally, notice that we have the (*head). Why do we need the parentheses? So, there's two answers to that: 1) you should use parentheses more often, they're an explicit way to denote what order of operations you intended; even if the code works without it, it's probably more readable with a few extra parentheses than are strictly necessary. 2) It's to denote order of operations. Remember that because * is an operator, then -> is almost certainly an operator. Which one takes precedence? Some people probably know off the top of their head. But there is little reason to keep that factoid in active memory when the easy solution is to just use parentheses to denote what gets grouped with what. (In this case, the parentheses are actually necessary, as -> is a higher priority than *. But that doesn't matter, because the principle is: use more parentheses for communicative reasons.)

Better Stack Code

With the caveat present that **'s are not wrong, just confusing and therefore difficult to read, here's a different version:

```
struct node* stackPush (struct node* head, struct node* target)
{
      target->next = head;
      return target;
}

struct node* stackPop (struct node* head)
{
     if (head == NULL) return null;

     struct node* = temp;
     head = head->next;
     free(temp);
     return head;
}
```

This one is a little different. We're hoping to demonstrate that there is no one stack code. There is a stack concept, with a thousand different implementations. In this case, notice that the push function returns the new head, which means that we don't need the **. But it does mean that the calling function needs to know to set the head = stackPush(head, target);. Similarly, the stackPop just removes the first node from the list. We added the checking code here, specifically to highlight the fact that the pop just needs to remove the node. Depending on the specifications, it might not be necessasry to actually return the popped node. Why would we have the stack then? Because we can always just access the top node by handling the head pointer. So, in this case, we're telling the calling function to just treat head like the node it wants to process, and to use pop just for the deleting part.

Which one is correct? Well, the latter is simpler, but requires knowledge of the internal mechanics a little more. The former is less readable, but separates the internal workings from the external workings a little better. We could go on and on with alternate implementations. In fact:

Array Stack

What if we just used an array? What if we know that if we get to a stack size of 100, we're never going to catch up, so we may as well delete the older stuff, or move it to a different function to be processed?

```
#define MAX 100 // 1
struct node
{
     int payload; // 2
};

struct node* arr[MAX]; // 3
int head = -1; // 4

void arrPush (struct node* target)
{
     if (head >= (MAX - 1)) // 5
          return;
     head += 1; // 6
     arr[head] = target;
}

struct node* arrPop ()
{
```

```
     if (head < 0) // 7
          return NULL;

     struct node* ret = arr[head]; // 8
     head -= 1;
     return ret;
}

struct node* arrPeek() // 9
{
     if (head < 0)
          return NULL;

     return arr[head];
}
```

We've got an array of notes to cover here:

1. In general, it's good practice to avoid random constants in your code. But sometimes, you just need a random constant. This is where the compiler directive #define is useful. It means that you can describe your numbers with labels in the actual code. For example, we picked 100 arbitrarily, so it's nice to describe it with an arbitrarily vague name. But, alternately, if it was picked as 'MAX_STORAGE_DEVICE_ALLOCATION', that would be good to know. Generally, use a define or a const like this if you use the same constant number more than once in your code. The compiler will do all of the replacements before it even begins the parsing phase, so there's literally no efficiency loss when using #define.

    - In this array-based stack, notice that we've imposed a hard limit of 100 nodes. The stack can never go larger than that, because the memory isn't allocated. This is one of the areas we mean when we say that C is memory insecure: unless every single accessor function to this array is on board with the idea of never going past the bounds, we will eventually have a array out of bounds situation (and even then, it might be a long time before that error actually affects something that will cause the developers to be able to notice). The correct answer is 'just don't program it wrong', which is true. That is the correct solution. But it's a bit naive to assume that an organization with a hundred developers of different levels of experience will all program exactly correctly.

2. Notice that the data node loses it's next pointer, because it's not a linked list. We have the monolithic array handling our order and greater stack structure.

3. This is an array of pointers, we'll discuss why this is common in a moment. Arrays like this are a bit of a devil's bargain. Think of it like a zombie apocalypse and your job is to keep those pesky zombies out of your house. In an American zombie movie, it's set in the suburbs and you just board up a couple of windows and doors and voila, you've got a zombie-proof area to discuss emotional backstories and character development. In a Korean zombie movie, it's set in an apartment building and you have to clear out floors and there's always another way for the zombies to get in through the ducts or the sewers under or up through the building structure from a floor that hasn't been cleared. From a design perspective, we really like the fact that we can closely guard a structure and define how it interacts with everything else. But from a practical situation, we like arrays because they're fast, easy to implement, and intrinsically bounded.

    - When we say 'intrinsically bounded', what do we mean? The linked list structure can grow infinitely as long as we keep pushing new nodes onto the stack and we have to actively develop a way to limit that growth. The array, on the other hand, can only grow to the size of the array and we have to actively develop a way to make it grow infinitely. Both of these are in a theoretical context.

4. There is no 'null' in this context. One of the interesting parts of using an array like this is that we are never really going to delete the old data (we can, but the basic algorithm won't). Instead, we just mark it as sort of not really valid anymore, since it's beyond the end of our array. In this case, when we want the entire array to be not real anymore (or yet), we can just set the upper end of the valid data to below zero. Since -1 means that we can add an array and just increase the upper end of the valid data by 1 like the rest of the algorithm will expect, it's typically cleanest to just use -1.
5. This is the first time we've done a NULL check like this, so we explicitly included it. Essentially, we've got two limits. There is a 'hard' limit at arr[MAX] and a 'soft' limit at arr[head]. The soft limit will shift with the pushes and pops, and the algorithm is based around the idea that it shouldn't be iterating past that soft limit. But we also need to define our exit scenario when we try to make the soft limit go past our hard limit.

    - Side note: we had a bug here for a long time, with head > (MAX – 1). The thinking was that we want to escape when the soft limit was higher than the hard limit. But since we're adding a new node with the push function, we don't want to check if we're past the hard limit, we want to check if we're at the hard limit. And that subtle preposition was enough to cause us to implement the code incorrectly for a while.
    - How did we find it? We were using some of our code analysis tools that look for memory leaks and out of bounds errors.

6. Again, notice that the only operation we have to do in order to extend the size of the array is to increment

the array size by one.

7. The opposite bound is both hard and soft, and that's when we have negative array indices, which we would typically assume means that there's a problem. There's a whole set of algorithms that handle how to construct errors encoded in the negative numbers returned here. But that would be in a later chapter.
8. When we pop and push, we're doing it at the highest valid array index that we've noted contains relevant data. But couldn't we do it at array index 0, instead? Yes, but then we would need to shift everything over to the right. Right now, the advantage of the array is that it's a single operation (O(1)) to push something on the stack. But if pushed to index 0, it would be an O(n) operation as we shifted everything over.
9. Peek is an interesting concept. Hardcore, old-school, grizzled, purist veteran coders would suggest that stacks only have push and pop. And if you want to look at the top element of the stack, you should pop it off to interact with it and then push it back on if you wanted to keep it. That's fine and all, if that's how the specific structure is defined. But, realistically there are a number of instances where we just want to look at the top element. For example, suppose we've got some sort of timer involved and all we want to know is whether or not it's time to address whatever is in the head of the stack? It would be wasteful to pop it, check it, push it, when we could just take a peek. 'Wasteful' in this sentence is, again, a bit leading. It depends on the application and the specifications of what's needed.

    - There is a whole sociological component here as well: people feel comfortable having the structures be the same and mirror each other. Notice that pop takes the place of delete and push takes the place of write, but we completely ignore the read atomic operation! We aren't, we've just combined it with the delete operation to make the pop operation. That tight binding is part of what makes the stack useful, in its restrictive interactions. But, a lot of people intuitively know that there are 3 atomic operations and really want to reflect that in all data structures. There's nothing wrong with that, and both approaches are reasonable. Again, it's about problem solving, not rote recital.

Which one do you prefer? The linked list stack or the array? It's OK to have a preference, but there's an old adage: when the only tool you have is a hammer, every problem looks like a nail. Just because you have a preference shouldn't override real and material reasons to pick one application over another.

Oversized Arrays

Here's the standard approach to resizing an array:

1. Given an array, A, of size n
2. Allocate a new array, B, of size n + c (some constant sized, necessary increase)
3. for i = 0..n-1
    1. Copy A[i] to B[i]

4. for i = n.. (n+c-1)
    1. Copy a new value x to B[i]
5. Delete A

That is perfectly fine, it is exactly and optimally space efficient. But, look online briefly and compare a 8 GB memory stick to a 16 GB memory stick, then evaluate the increase between a ~$300 processor and one that costs twice as much (processor speeds will change faster than memory, so we can peg memory to an absolute value, but it's easier to peg the processor to an approximate cost). You'll probably find that the relative cost of processing speed, right now at the time of publishing, is much more expensive than the cost of memory and especially disk space.

What does that tell us? It's an important enough point to get another subheading:

An Era of Computing Plenty

If this were the Israelite Exodus, they'd be promised to be led to a land flowing with Processors and RAM. Consider the average processor and memory for a super computer in 1965 would have probably topped out at a single Mhz and the memory size would have been crazy large at 256 KB. These are supercomputers, mind. There's a reason Bill Gates is famously attributed as mispredicting the idea that we'd never need more than 640 kb of memory for a desktop. Compared to historical computing standards, even that is a huge amount of memory. This machine would have cost around $10 million.

The average desktop computer in 1990 would have performed at around 20 MHz and had around 2-4 MB of memory. These machines would have run $2,000-$3,000. They would have run MS Dos with possibly a Windows application-level system that wasn't consistent enough to be the main operating system for the computer.

The modern tiny device in 2020 that fits in an average pocket and costs around $200 and performed at around 2 GHz, with around 3-4 GB of RAM. At that point, the biggest point of contention in the operation of the device was the fact that the battery only lasted so long and the screen was too tiny to do 'real' work.

You've probably seen these sorts of progression before. But let's apply it to algorithm design for a moment. How much space does an array of 1,000,000 pointers take? 8 MB. The phone that fits in your pocket in 2020 probably had enough free memory to hold thousands of those arrays at any given moment. And, this is an important question, but why would we care about optimizing for space when there's almost literally always more space available for the program to draw from? Further, we could compare the size of the data type. Suppose we were considering HD textures that were loaded on screen and needed to be immediately available. If each texture takes somewhere between 1 and 16 MB (that is, a 1k to a 4k resolution), what does it matter if we use up literal hundreds of wasted bytes (an overhead rate of less than 0.1%) to maintain faster access speed for a game that might be focused on smooth framerates and intended for high end gaming PCs?

This is a very real phenomenon. It's almost a rite of passage for experienced developers to look at modern

software and complain about how bloated everything is. It is, but it's also the fact that resources are so much more abundant than 10 or 20 or 30 years ago that the bloated, inefficient software still runs better, faster, and more robustly than even the lean and elegant software used to run.

How can you apply this? When you're developing, it's important to always ask about what your tradeoffs are. If processing power is more expensive than space, then allocating extra space and not always using it is preferable to being space-optimized (so use an oversized array). If memory, disk space, or network bandwidth (it's a transmission rate, but in another way it's a space-constraint) is more expensive, then it's better to use something like a Linked List structure for your stack.

What this is not: this is not an excuse for wanton waste. There is some of that in the industry, because another factor is time/cost versus development time. It's faster to write unoptimized code and when a company is paying coders by the hour or line of code, the company may prefer to rely on the prevalence of computing resources instead of writing efficient code. But, that's a business problem, not a computing problem. Within the realm of computing, we are saying that you can be inefficient in one metric if it means a gain in another metric. This is important: it has to be a tradeoff, not just laziness.

Queueueues

First off a reminder: mistyping Queue is something every computer scientist has done. Most of what applies to a stack also applies to a queue. The biggest distinction is that a queue treats the push operation (basically) the same, but the pop operation is different. We'll go ahead and jump right to the version that mirrors the 'better' stack code. You can backtrack if you need to, or look up examples that use the ** operator.

```
struct node
{
     int payload;
     struct node* next;
};

struct node* queuePush (struct node* head, struct node* target)
{
     target->next = head;
     return target;
}

void queuePop (struct node* head)
{
     struct node* curr = head;
```

```
    while (curr->next->next != NULL)
    {
        curr =  curr->next;
    }
    struct node* temp = curr->next;
    curr->next = NULL;
    free(temp);
}
```

Again, we're omitting the NULL checks and the empty list checks, because you should practice by adding them in. Also, it saves space and attention to focus on the things that matter. Notice how, in this case, popping takes up an O(n) operation instead of the stack's O(1)? That's either intrinsic to the queue structure or we need to change the node to accomodate it. As long as we're going to change the node structure, we'll make everything a little more robust:

"Better" Queue Code

```
struct node
{
    int payload;
    struct node* next;
    struct node* prev;
};

struct queue
{
    struct node* head;
    struct node* tail;
};

struct queue* createQueue  ()
{
    struct queue* q = malloc(sizeof(struct queue));
    q->head = NULL;
    q->tail = NULL;
    return q;
}
```

```
void queuePush (struct queue* q, struct node* target)
{
     if (q == NULL || target = NULL) return;

     if (q->head == NULL)
     {
          q->head = target;
          q->tail = target;
          target->next = NULL;
          target->prev = NULL;
          return;
     }

     target->prev = NULL;
     q->head->prev = target;
     target->next = q->head;
     q->head = target;
}

struct node* queuePop ( struct queue* q)
{
     if (q == NULL) return NULL;
     if (q->tail == NULL) return NULL;

     struct node* ret = q->tail;
     if (q->head == q->tail)
     {
          q->head = NULL;
          q->tail = NULL;
     }
     else
     {
          q->tail = q->tail->prev;
     }

     ret->next = NULL;
```

```
        ret->prev = NULL;
        return ret;
}
```

Notice that we're treating each example slightly different. This is to implicitly expose you to more ways of coding the same constructs and help you to find the style that you like the best. We encourage you to rewrite any or all of the examples with the slight variations that you prefer. This version has a slight conceit, though. Typically, we like to think of data structures as being self-assembling. That is, it's a set of nodes and the behaviors that implicitly create a semantic construct of a queue based on the emergent behavior of the defined methods. However, in this case we explicitly defined a concierge object (not a standard term, but the authors' own; there doesn't seem to be a standard term, although 'interface', 'opaque data handler' or 'controller' are all common) that provides extra utility. Is that good or bad? From a provability perspective, it's a little inelegant. From a readability or utility standpoint, it's pretty common and useful.

Why wouldn't we always use it, then? Ultimately, it's because it very narrowly and specifically limits how we can use our structures. That's good in a lot of cases, but it does reduce the reusability a bit. Further, it also requires some extra overhead in the form of the concierge object itself. It's not a lot in this case, but in other cases it might be.

Regardless, notice that the structure of this queue is still O(1) in both directions at the expense of a little extra overhead in space usage, since we have to have the ->tail and ->prev pointers. Final note: if we only ever have a single queue, we could probably get by with just using global variables instead of the struct queue. However, global variables get a little unwieldy across large projects. But, for example, in an embedded environment, it might be just fine to have a global queue of, say, instructions to execute. On the other hand, in an embedded environment, having a concierge type object that controls the queue explicitly would provide room for rate limiting or size limiting the push operation.

Queue Array

Just like the stack, queue's have a frequent implementation as an array. With the queue, however, we need to accomodate the two different endpoints that we're interacting with. If we were just going to shift everythign every time either the head or the tail changed, we may as well just reallocate the array. Instead, we want to maintain the O(1) nature.

```
#define MAX 100
struct node {
        int payload;
};
```

```
struct node* queue[MAX];
int head = 0;;
int tail = 0;

void queuePush(struct node* target)
{
     if ((tail + 1) % MAX == head) return;
     queue[tail] = target;
     tail = (tail+1)%MAX;
}

struct node* queuePop ()
{
     if (tail == head) return NULL;

     struct node* ret = queue[head];
     head = (head + 1) % MAX;
     return ret;
}

struct node* queuePeek()
{
     if (tail == head) return NULL;
     return queue[head];
}
```

The %, remember, is the modulus operator, and gives the remainder after integer division. But, the question is, how optimum is that process, or is it just a stupid C trick to do it this way? Further, how would you denote in the documentation ahead of time that tail and head being equal is when the struct is empty, but when tail is immediately prior to head it's full (hint, the return type will be NULL and that's all other functions need to know. So it'll just be an explanatory part of the documentation)? You should explicitly write the documentation and test to see if modulus is as fast as any other basic integer operator.

Regardless, this array implementation is a rotating implementation. We're slowly incrementing the entire queue and then wrapping back around to the beginning. The only fiddly bits comes when the array becomes full and we have to check for that. How can we check that the array is full and not empty? Well, that's why we end up wasting the one index, and the practical max size is MAX – 1. Since avoiding that means creating

an extra global int that denotes size that we increment and decrement with a push and pop, respectively, it isn't really a waste. Incidentally, how would you implement this with a global variable that is size, to track where the rotating vector is? Could we get away with just a head index, then?

As a side note, this global structure is not ideal in many situations. The general term 'queue' is awkward. However, when programming a game or something similar, we'd probably have global variables with more descriptive names like 'inputQueue' or 'soundEffectQueue' that we would all track independently, and therefore might need an extra argument for the function after all.

Implementation Concerns

Array v. Linked List

We've discussed it tangentially, but let's take a look at when you would decide to use an array-based implementation versus a linked list based implementation. The question largely comes down to the nature of the processing that will happen for each entity. If the processing needs to happen for each entity, and it's convenient to do it in turn, and the entries don't change very often (at the scale of processing speed, so no more than every few seconds), then a linked list is probably appropriate. But, on the other hand, if the processing is going to hop around between random elements, or spot instantaneous access to random elements is important, then the Array's random access is probably the most effective.

And, realistically, that's the most important part there: the high level decision making about how the data is going to be access. That's a human decision that needs to be made and communicated across a team. Technically, there's not a lot of reason to pick one versus the other. There are adherents to both schools of data structure and there are algorithms that will apply to both. But, what's important is the software engineering decision about which side to go for. And that's a question of Random Access or Sequential Access. Those, in turn, are representative of a larger design pattern that we need to discuss.

We could make a connection to eager versus lazy evaluation. This is more of a programming language concept, and relies on the idea of whether or not an expression should be evaluated immediately upon assignment or if it should just be stored and evaluated when it's used later. They both have their uses, and it primarily focuses around things like '$f(x) = x^{2}$'. How do you evaluate the entire, infinitude of that equation? Obviously, the answer is you can't, and the more you try, the more likely it is that any answers you pre-populate are going to be useless. Sure, you could immediately evaluate the fact that $f(1) = 1$ and $f(2) = 2$. But if the user later needs to evaluate $f(1.5)$, then you wasted the values earlier.

On the flip side, in our stacks and queues array examples, we included the #define statement. That replaces every instance of MAX with the constant 100. Since sometimes our expressions were MAX – 1, the code doesn't compile the equation '100 – 1', it eagerly evaluates it before it even gets to the parser and sends the

parser the constant '99'. That's not exactly what we mean by 'eager' evaluation. But it's along those lines: sometimes it's important to not store how we got to a conclusion, just the conclusion itself.

Neither of these approaches is right, rather they have their various applications where it makes more sense.

Let's broaden the scope one more time and talk about proactive versus reactive. This is not a Computer Science specific topic, it's just general problem solving. You've probably been told in your life to be more proactive. The idea is that if you wait to react to a situation, it's likely too late already. For example, in martial arts, if you're always reacting to what your opponent does, eventually you won't react fast enough. Instead, you want to set it up so that you're driving the engagement and your opponent is reacting to you. That makes sense in martial arts. But, for example, it doesn't make sense in child-raising. Nobody wants to be a small child whose parent is like 'I know you are only 5, but I've already arranged for you to go to a specific college, and you're going to train to be a podiatrist'. Sure, it might be a lucrative field, but that's not how human nature works: you can get to the same conclusion, but by reacting and guiding, and not by proactively dictating.

Similarly, for design methodologies, different design paradigms hold sway in different contexts. Sometimes, we want to be proactive and cycle through each and every one of our entities every time, because we'd rather do small amounts of calculation every time to make sure that we never have a situation where we need to catch up by a lot. Other times, we only want to calculate over the specific elements that are important right now. For example, on one level, your monitor is constantly processing every pixel on the screen. It's cycling through every single pixel from top left to bottom right (why? Because CRT monitors had an actual electron tube that had to move in a pattern, and that was the pattern it used: each pixel from left to right and each row from top to bottom) and insuring that the correct ratio of primary colors is excited to get the appropriate color to display. However, your graphics card doesn't do that it typically only processes the parts of the screen that change or might change and leaves everything else the same. For example, if you're reading this book as a PDF, you likely have a menu bar across the top of the screen and 99.999% of the time, that top bar is exactly the same. So unless the graphics card gets an indication that it's going to change, it's better off focusing all of it's processing power on the parts that will change.

The point is, all of this is just a question of how you want to implement the various solutions. Once again, it's a reminder that you need to have as many tools at your disposal, and just pick a solution based on the best information you have, and targeting the best outcome you can anticipate.

Let's talk about a few specific implementations:

Undo Stack

In Modded Minecraft, Direwolf20 has a mod called Building Gadgets. Most of the gadgets have an undo feature. How does it work? You could check his github page to find out. But, let's speculate a little bit:

```
struct changedBlock
{
```

```
     int x;
     int y;
     int z;

     state* previousState;
     action* someAction;
};

#define MAX 1024
struct changedBlock* undoHistory[MAX];
int undoHead = -1;

void undoHistoryPush(struct changedBlock* p)
{
     if (undoHead == MAX - 1)
     {
          free(undoHistory[0]);
          shiftLeft(undoHistory); // Is there a faster solution here?
          undoHistory[undoHead] = p;
     }

     if (undoHead < MAX - 1)
     {
          undoHead += 1;
          undoHistory[undoHead] = p;
     }
}

struct changedBlock* undoHistoryPop()
{
     if (undoHead >= 0)
     {
          undoHead -= 1;
          return undoHistory[undoHead + 1];
     }

     return NULL;
```

```
}
```

We're taking lots of liberties. First off, the Minecraft code is written in java and not C. And, we're not tying it to the real code, but you're welcome to compare solutions. We're presuming that there is some state struct that is available in another library, and that the action that was taken with the gadget is defined in some custom struct.

For each gadget, and there are several of them, they might all share the same undo array, or they might each have their own. For this simulation, we're supposing that they all share one, because that seems more interesting and easier to avoid contradictions between the devices. Now, the functions would do something like:

```
struct actionConfirmation* rightClickBinding (gameState* g, heldItem* i, acti
{
     struct changedBlock* p = malloc(sizeof(struct changedBlock));
     p->previousState = copyState(b);
     p->x = b->x;
     p->y = b->y;
     p->z = b->z;
     p->someAction = doAction(i->currAction, b);
     undoHistoryPush(p);

     return SUCCESFUL_ACTION;
}
```

And then when we push the Undo button, it'll do something like the following. Notice that there's no targetBlock, because that comes from the undo action itself.

```
struct actionConfirmation* revertState (gameState* g, heldItem* i, actionStat
{
     struct changedBlock* p = undoHistoryPop();
     if (p == NULL) return NULL_ACTION_TEMPLATE;

     block* b = globalBlockRetrieve(p->x, p->y, p->z);
     doAction (revert(p->someAction, p->previousState), b);

     free(p);
}
```

Freeing up to 1024 redo actions is probably not a major problem in this simplified example. Since the redo

history is capped, clearing it directly is simple and reasonable. Could we use a third garbage-collection array? Yes, but it may be overengineering for this example. A garbage-collection array would store objects that should be freed later, when the program has time to process them. This might make sense in a performance-sensitive system where freeing memory immediately could cause a delay. However, if we wanted delayed cleanup, a linked list or queue would probably be more natural than a fixed-size array. Another common solution would be an object pool, where old undo records are recycled instead of repeatedly allocated and freed.

These examples are only meant to give the flavor of how undo and redo systems might work. In real code, we might not see functions literally named `push` or `pop`. The actual code may use Java collections, command objects, events, packets, or engine-level callbacks. However, the underlying data-structure idea is still similar: save enough information before an action happens so that the action can later be reversed.

Overall, this example is not realistic in the sense that it matches the actual Building Gadgets code line by line. But it is realistic in the sense that it feels like the kind of logic a game engine or modding codebase might need: actions are registered, block states are saved, undo records are stored, and predefined success or failure values are returned.

Task Queue

Suppose you're writing a document in a word processor and you want to set up some macros. These macros might type your name automatically, or perform a series of font changes, or who knows what sordid actions might happen in a Word Processor. Internally, the Word Processor might store these actions as a Queue. Why? Because as you add the actions, you will want them to be repeated in the same order you added them (FIFO), but at the same time, the program doesn't need to know how long the macro is going to be at the beginning when the first key is pressed, so it can't allocate a fixed amount of space for it. Thus, you might get something like this:

```
struct macroPress
{
     KEYBDINPUT ki;
     struct macroPress* next;
};

struct macroPress* MacroKeyList = NULL;

void buildMacro ()
{
     while (PeekMessage(...) != MACRO_END_KEY)
     {
```

```
            macroPressListPush(buildKeyInputFromMessage(GetMessage(...)));
      }
}

void macroPressListPush(KEYBDINPUT ki)
{
      if (MacroKeyList == NULL)
      {
            MacroKeyList = malloc(sizeof(struct macroPress));
            MacroKeyList->ki = ki;
            MacroKeyList->next = NULL;
            return;
      }

      struct macroPress* curr = MacroKeyList;
      while (curr->next != NULL)
            curr = curr->next;

      curr->next = malloc(sizeof(struct macroPress));
      curr = curr->next;
      curr->ki = ki;
      curr->next = NULL;
}

void macroPressListPop()
{
      struct macroPress* curr = MacroKeyList;

      while (curr != NULL)
      {
            SendInput(buildMessageFromKeyInput(curr->ki));
            curr = curr->next;
      }
}
```

There's a bunch wrong with this code. The biggest issue is that it's silly to write code for any given version of Windows in a textbook that is intended to be long-term viable. The '...' is just a hand-waving way to say

'whatever the Windows arguments are', but otherwise the functions are a reductionist version of what this might look like in Windows. Visual programming in C is like playing Dark Souls on Ironman Hard Mode, you really need to know a lot of the internal systems really well if you want to get anywhere. Futher, the macroPressListPop() should probably be macroPressListPlay() or something. But the point that we're trying to demonstrate is that no matter how much we mutate the structure, it's still a queue at its core. The actual pop mechanic will be when the user deletes the whole macro, but that doesn't change the fact that we still want to read through the queue in order.

That does bring up the question: could we do this as a stack?

Mutability

Data can often be represented in multiple ways, and many data structures can be adapted to fit a particular problem.. Consider temperature: is it 0^{o} C or 32^{o} F? Both are correct, but which one gets displayed doesn't matter! All that matters is that the underlying data is there, everything else is presentation. Algorithms are like that, as well. There are not algorithm police that are going to come by and complain because your code doesn't measure up to an academic level of quality. Instead, you'll be judged on whether it accomplishes its goal, it is readable, reusable, and secure/robust.

Sometimes you need to have two things in mind, and this is one of those areas: the data structures and algorithms we're covering are more or less fixed with a widely accepted form factor. But, at the same time, they're infinitely mutable and changeable as you need them to adapt to your situation. One fun game is to ask an LLM 'how many variations of <some data type> are there?' Any one you pick will have dozens, and what an LLM 'knows' can broadly be considered to be public domain knowledge (this is up for debate, but in the context of computer science algorithms, it's a fairly accurate idea to say these are all public knowledge). So, it's likely not counting all of the specific, proprietary implementations for specific applications. Pick one of the wierd variations it lists and ask how many variations of that variation there are. In our test, we got 24 variations of a linked list, "plus dozens more" in a single response. For a circular doubly linked list, we got 12 immediate variations with 'many more' variations list based on 3 different axes.

The point is, remember the exact algorithms when you can, but most of them will be widely available. Most employers want to know about your problem solving ability, and some love to throw in 'trick' scenarios where the vanilla data structures won't work and you'll need to adapt it on the fly. As a side note, if you want another odd queue structure to add to your list, consider a deque: Double Ended Queue.

Priority Queues and Heaps

Let's finish up with something completely new. What's a heap? In one sense, it's sort of an unstructured data thing. It's a catchall term for when we want a category of things, but we're not worried about the details. For

example, when we talk about the 'memory heap', there's probably some detailed structure there, but we don't care about it as a programmer at the moment, we just need to reference the fact that the memory exists.

But, this text is not about Data Things and Algorhythms. If we want to be more structured, we can start doing fun modifications of our Linked List. One of the questions from an earlier chapter was: what happens if we have more than one ->next for a linked list node? Let's try it!

Basic Tree Structure

Remember that a linked list, and any linked-list based data structure (like sometimes when we implement Stacks or Queues) uses the following data structure:

```
struct node
{
     int payload;
     struct node* next;
}
```

As a reminder that we've made several times, it doesn't have to be an int payload. It's just easier to write random numbers into the examples than it would be to try to fit complex data structures there. But, realistically, most situations would call for something more complex than just an int. Here's that same data structure, but targeting a tree:

```
struct node
{
     int payload;
     struct node* left;
     struct node* right;
};
```

Notice how we have to change the values. We could make them anything (up/down, top/bottom, charm/strange), but since we like to draw pictures to represent these data structures, we'll orient them left and right to emphasize that the final diagram will kind of sort of look like a child's drawing of a christmas tree (it's really just a triangle, but 'binary triangle' sounds wierd). These are typically drawn as a 'tree', with a single node at the top, and an increasing number in every layer beneath that (but never more than double). It is also called a binary tree when there are only two unidirectionally linked nodes. We like this particular tree type, because it's simple and it allows us to skip through it easily in a binary search pattern, but with the autonomous self-assembling linked list conceit.

A heap is a semi-ordered data structure. In any given tree or sub tree, the top node is the smallest one

(specifically in a min-heap, which is what we'll stick with; a max-heap is literally just a > instead of a <), but we don't guarantee order more than that. This means that in order to find any given node in our tree, we need to search every node in the tree and haven't gained anything. However, sometimes, we're not worried about finding any given node. We're only interested in, say, the smallest node. For example, maybe we decide that the next task we need to work on is the one with the least amount of time before it's due. We don't need an intricate list ordered for every tasks due date. We just need to know what we should be working on next (or, what our priority is).

Also, recall that sometime sit was easier to use the following:

```
struct node
{
     int payload;
     struct node* next;
     struct node* prev;
}
```

We can get by without it, but doing so increases the complexity of our functions, so it's frequently easier to have access to the previous node for when we want to delete a node or so on. The equivalent would be:

```
struct node
{
     int payload;
     struct node* left;
     struct node* right;
     struct node* parent;
}
```

What's the point of this structure? We want it to do two things: we want to be able to find the next thing we should prioritize as fast as possible. Since that's going to be the root of the node, that's easy. Second, we want to be able to pop that root node off or push a new node onto the heap fairly quickly. Again, we aren't going to guarantee an order, just that at any given moment, any given root node is the smallest for its subtree.

In fact, when the tree is organized with two nodes like this, we find that it operates push and pop in log(n) steps. Let's take a look:

```
void insertPQ (struct node** root, struct node* target)
{
     struct node* curr = target;
     // Find the first 'open' slot through a breadth first search.
     // Insert into the first open spot.
```

```
     // "Bubble Up" by swapping the new node with any parent that's smaller t
}

struct node*  peekPQ (struct node* root)
{
     return root;
}

struct node* popPQ (struct node** root)
{
     struct node* ret = *root;
     // recursively pop the smaller of the left or right node off their subno
     // replace the root node with the popped node
     return ret;
}
```

We're not going to actually write this code out, because it's long and obnoxious and detracts from the point of demonstrating how a binary tree can be designed around these linked nodes. Notice that the push node uses a breadth first search to find a place in the tree to add the node. We'll cover that in the Trees chapters. But also notice that the pop node moves recursively down. This is an easy way to visualize or diagram what's going on. The problem is this: because we are trying to create a structure that self-assembles into complex behavior (that is, implicitly creates the priority queue out of the interactions between the nodes), we don't actually have a good idea of how long the code is.

This is actually a very normal situation in practice. But it's too complicated for right now. All we want to demonstrate is two things:

1. The general structure of tree-based data structures.
2. That we can still take something complicated like a tree-based data structure and represent it in an array.

To that latter end, let's take a look at how we would make that representation.

Abstract Data Types

Let's take a minute and define some terms. So far, we've been muddling together implementation and concept.

- Abstract Data Type – When we talk about abstract data types, we're talking about the data structure as a nebulous idea. We describe its behavior, but we don't worry about how or if it's implementable. We

can describe what would happen if we had this data type and how it would behave, without detail. An example might be a Magnetic Monopole in physics. That's a source of magnetism that only has a North or a South pole. In reality, all magnets are bipole magnets, we always have not just a north and a south pole, but they always match exactly. But physicists can ask the question 'what if not that?' And that's where they propose the magnetic monopole and can theorize about the sci-fi sorts of benefits we'd get if we could find one in nature. The fact that the monopole isn't real doesn't stop them from understanding how the universe works a little better. Notice that in each of the following descriptions, we don't talk about how to do these, just what's being done:

  - A stack is an Abstract Data Type: we know that we can only push and pop from the the same end, and that it is FILO.
  - A queue is an Abstract Data Type: we know that we can push and pop from opposite ends, and that it is FIFO.
  - A sorted list in an Abstract Data Type: we only need to know that it's a group of values and they're monotonically increasing or decreasing.
  - An unsorted list is an Abstract Data Type: like a sorted list, but we don't guarantee that it's sorted.
- Implementation – This is when we talk about the specific nodes or arrays or detailed definitions for how to implement the system. This is where we get the detailed breakdown of what data structures will be included and how we're going to implement the various functions that we need.
  - A sorted linked list is an implementation.
  - A binary tree is an implementation
  - A min-heap is an implementation

Broadly, an Abstract Data Type could be loosely described as goals an outcomes. We then create a concrete data structure and an algorithm to make those things happen. So, an implementation is specifically the data structure combined with the algorithm that results in the abstract data type being possible. This is a little obfuscated by the fact that the concrete data structure can, itself, have many different implementations.

So, in this current section, when we talk about a Priority Queue, we're talking about the Abstract Data Type that has a set of data, but the biggest or smallest is always available. Notice that this is different from a Queue. Whereas a queue was focused on having the first value pushed on to the queue be the one that's next available and a stack was focused on having the last value pushed on to the queue as the one that's next available, the priority queue is focused on something in the data itself to determine what's available.

We're using the concrete data structure of a min-heap to define behaviors on how to make this priority heap happen optimally. We could just throw everything into a big pile of data and search for the smallest one every time. But, with just a little organization, we can make sure that the smallest is always available in O(1) and that we can add any value or remove the root value in O(log(n)) time.

Finally, the specific mechanisms behind the min-heap is what we're looking at with the nodes or array implementation. Let's move on to the specific array implementation:

Priority Queue as an Array

```
int arr[MAX]; // 1
int size = 0; // 2

void pushPQ (int value)
{
     if (size >= MAX) return;

     int i = size; // 3
     size += 1;
     arr[i] = value;
     while (i > 0 && arr[i] < arr[(i- 1) / 2]) // 4
     {
          swap (i, (i - 1) / 2); // 5
          i = (i - 1) / 2;
     }
}

int peekPQ()
{
     if (size <= 0) return -1;

     return arr[0]; // 6
}

void popPQ()
{
     if (size <= 0) return;

     size -= 1;
     arr[0] = arr[size]; // 7
     int i = 0;
     while (TRUE) // 8
     {
          int left = 2 * i + 1; // 9
          int right = 2 * i + 2;
```

```
            int min = i;

            if (left  < size && arr[left] < arr[min])
                 min = left; // 10
            if (right < size && arr[right] < arr[min])
                 min = right; // 10

            if (min == i)
                 break; // 11
            swap (i, min);
            i = min;
      }
}
```

Let's prioritize a few of the lines here:

1. This is just a standard array of integers. It could be any of a million different data types, even a node if we strip out the ->left and ->right, since those will be intrinsic in the array.
2. By default, there's no data in here. In this case, we're setting size to be literally how many values we've added, this is in contrast to other data structures we've used that had -1 for the size when they were empty. Why would we change that? Just to show both sides, or is it intrinsic to the heap structure? You could prove it by changing either the stack/queue to index at 0 for empty or this min-heap to index at -1 for empty.
3. This is the key for why the array implementation is so much easier. Because we know what the size of the heap is, acting within the oversized array, we can always access the last filled index (and therefore the first empty index) immediately. This is vital to creating a 'full' tree, which is where we want every layer to be filed left to right before we move on to the next layer.
4. i is not the size right now, it's just the index we're looking at. Basically, we put the node at the furthest node away from the root. (Notice we're still using the terminology from the linked list, because it's still how the abstract data type tends to be conceptualized, even if we implement with an array.) And then we're going to use the same process from Bubble Sort to swap it with it's nearest parent whenever it's smaller than that parent so that it bubbles up to the top. We're saying 'move to the point where you're the smallest root node of the biggest sub tree you can'.
5. The details of swap don't matter and would be changed based on different data types. Note that we could either pass the indices to swap or the pointers to the values and have it swap the payloads. Either is fine, and both are specious when the data type is current just an int.

    - More importantly, what's the (i – 1) / 2 bit? Basically, we're storing each row of the tree in its own

little subarray. so, the root node is stored at arr[0]. The two first layer nodes are stored at arr[1] and arr[2]. Their children nodes will be arr[3-6], etc. Using this, we can always find the parent node by using integer division (which will omit the remainder, therefore dividing odd nubmers by 2 will give us the same answer as an odd number – 1 / 2). We have to preface it with number – 1, essentially, because we have a zero-index for the root node.

6. Peek is almost always super easy.
7. This is a little sneaky, but basically, we're swapping the biggest existing node into the root spot and then the old root back to the 'worst' spot. But then, since we reduce size by one, we basically declare that where the old root now is, is no longer part of the actual heap. And then, we'll need to take a moment to have other values bubble up to where the root node is, since it's probably not the smallest node.
8. We'll do this on an infinite loop. Why? Here's a better question, what's the difference between an infinite loop with an if statement that leads to a break in the middle and a regular loop with a standard conditional? In this case, not really much except that we want to do some stuff, then check the conditional, and then do some more stuff. We could rearrange it so that we use a traditional boolean conditional for the while, but it's frankly not as clear, despite needing this explanatory note for the infinite loop that gets a break.
9. This is the equivalent of having the permanent ->left and ->right nodes in the linked node version. Except, because we always know where the left and right are based on the structure of what we've put together, we don't need to retain them permanently, we can just recalculate them every time. what's the tradeoff? It means that we do have to spend some computation time recalculating the child node indices every time, when we could have just stored it (in fact, you can use a hybrid approach where you include the indices as attributes for the data structure, you would just need to make sure to update those indices every time the node moves position. It's not a great idea, just a note that you can).
10. Because the old root node has disappeared, we don't need to do the previous equivalent of shiftRight(), we've already made the room and can just assign the values.
11. We exit the node when it turns out that neither child node is actually smaller than the current value. This will happen by default when there are no child nodes, because we put that as an escape class earlier in // 10.

That was a lot of algorithm and data structure to get to a basic point. And that basic point is that we can encode complex looking arrays that take 2 or more dimensions to draw in a 1D array. More specifically, it is relatively common to map Tree structures onto single dimensional arrays.

Hologram Side Venture

Imagine you have a 2-dimensional integer plane. How many dimensions do you need to represent it in a data

structure? If you said "2," you're got an answer, but not the minimum answer. It turns out that you can represent any 2-dimensional cartesian coordinate system with a 1D array. For example:

```
[(0, 0), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1)
```

It should look like a spiral if you plot those points out. and you can continue on in that vein infinitely, or until you hit MAX_INDEX. So what's happening here? This has to do with what we call 'mapping'. We can map data from one format to another. There's no problem using a 2D matrix to list these values out. But there's also no problem using a 1D matrix. It's all about complexity and what we're looking for. So, in the case of a tree being represented as an array, we can represent the data as 1D data, even though it's a higher order than 1 dimension geometrically. We could find a lot of ways to analyze it, but that starts running into fractal geometry, not computer science.

The point is, that we call this kind of mapping a hologram in physics: where we represent a higher-dimensionality data set in a lower dimensional structure. We aren't going to get sidetracked by orders of infinity or minimum dimensionality analysis. But those are things you could get sidetracked with on your own. (In CS, it's just called dimensionality reduction, which is a much more boring term. Feel free to use hologram and make it a standard term!)

It's fairly common to represent data in lower order spaces, because it makes it much easier to run calculations. for example, you use a derivation of this all of the time: spheres are 3D objects, and we live on a sphere, but we use a 2-d coordinate system to mark latitude and longitude. And with latlon data, we can identify any place on the planet. Fun fact! That's what a knot (nautical miles, the speed of boats) is different than land-based mph: because knots take into account the curvature of the earth even though it's small for small distances it's still there.

Side Note: there's a physics theory that actually postulates that the entire universe is actually 2-dimensional and that we perceive 3-dimensions because of this same hologram effect.

Practical Application

Ultimately, what are we looking for in any given implementation? There are two metrics that we can always reliably use:

- Readability – This is the ability of any one to read the code that you wrote. Further, it's also a measure of how understandable the code is. If you, for example, include the code *ptr->member, it may be technically correct. But, the reality is that most people aren't going to immediately look at that and know that you want the dereferenced value of the member of the ptr. They're going to have to stop, read it, process it, decide what's most likely, and then hope that they guessed right (and maybe they even stop to test to be sure they're right). Even if they know the right answer, the specific format looks confusing.

- Reusability – This is a measure of how specific the solution is to the target environment. Frankly, a lot of C code is very custom-built. There will be some libraries, but the only reason to write C code in the modern environment is to take advantage of low-level calls that won't always translate over well. However, that doesn't mean that reusability is unimportant. Because, at some point, the specifics of the hardware will change and the code will have to be updated. Using an algorithm that takes advantage of some off-brand timing issue for the specific processor and lets you cut out certain system calls is fine, but it means that the code will never be ported, it'll have to be rewritten from scratch.

The point of this is that there are always tradeoffs. You need to decide whether the tradeoffs are worthwhile for any given situation. But, to the extent feasible, trying to adapt common knowledge algorithms to specific situations is typically your best bet to satisfy both of these situations. The most technical or niche the data structure or algorithm, the harder it will be for others to read since they won't know what to expect, and the harder it will be to take the code and apply it to new situations.

That's not to say that very targeted solutions are incorrect. To a certain extent, the entirety of driver programming could be said to be very targetted since it has to run on very specific hardware. Again, though, they have patterns and they have structures that they can use to make sure that the overall approach falls in line with readers' and porters' expectations.

AI Examples

We talked a little bit ago about data mutability. What exactly does that mean? Consider the following two data structures:

```
struct dog
{
     int IDnum;
     int heightInInches;
     float weightInLbs;
     enum Breed type;
     float ageInMonths;
     enum Placement placement;
};
struct entity* bestInShowDogs[NUM_DOGS];
```

This is versus a rotation of this data:

```
int IDNumBase = SOME_DEF_VALUE;
```

```
int heightInInches[NUM_DOGS];
float weightInLbs[NUM_DOGS];
enum Breed type[NUM_DOGS];
float ageInMonths[NUM_DOGS];
enum Placement placement[NUM_DOGS];
```

Which one of these data structures takes more memory space for, say, 500 dogs? Aside from some non-significant variation based on IDNum, which could be optimized away based on specifics but we wanted it to be clearer what connected to what, they should both use about the same amount of data space (assuming each base data type is 32-bits). That's 6*4 = 24 bytes per dog across 500 dogs = 12,000 bytes of data for the first data structure and 500 * 4 = 2,000 bytes per attribute across 6 attributes = 12,000 bytes of data for the second data structure.

What is important to remember is that the data is just there. The computer itself is, effectively, agnostic about what the data is, what it means, how it's arranged. The data, like a Turing Machine, just enacts single instructions on the data put in front of it. It's the humans programmers who are giving semantic meaning to the code (Side Note: This is also true of LLMs, they're just a statistical central tendency model that happens to be huge and that we anthropomorphize into a semblance of personhood. Maybe at some point it will be qualitatively more, but right now it's the same as looking at the front of a car and seeing a face. But this is a bigger issue for later). What we can do, however, is aggregate the data together and physically arrange them next to each other.

It's sort of like stuffing envelopes (or manufacturing muskets for Eli Whitney or assembling cars for Henry Ford: data, logistics, and optimization are, in many ways, all the same) for a wedding. Nobody wants to stuff the 1,000 envelopes that someone has deemed need to go out. So, instead, people set up an assembly line. We intuitively know that having one person fold, another stuff, and another seal is much faster than having three people alternate between folding, stuffing, and sealing.

It's the same way with the data: we should be arranging the data in a way that makes the most sense for the processing that we're going to do. Let's consider two applications:

```
void DogToString (struct  dog* d, char* ret, size_t retSize)
{
      snprintf(ret, retSize, "Dog #%d, Breed: %s Height: %d inches Weight: %.
           d->IDnum, BreedToString(d->type), d->heightInInches, d->weightInLb
}

void PrintDogPlacements ()
{
     char buff[1024];
     for (int i = 0; i < NUM_DOGS; i++)
```

```
      {
            DogToString(bestInShowDogs[i], buff, sizeof(buff));
            printf("%s", buff);
      }
}
```

And this would be opposed to some calculation like this:

```
float averageDogHeight()
{
      float sum = 0.0f;
      for (int i = 0; i < NUM_DOGS; i++)
            sum += heightInInches[i];
      return (sum / NUM_DOGS);
}

void DogAveragesToString (char* ret, size_t retSize)
{
       snprintf(ret, retSize, "Number of Dogs: %d, Breed: %s Height: %.2f inch
            NUM_DOGS, BreedToString(maxBreed()), averageDogHeight(), averageDo
}
```

As a slight clarification: an array is an aggregate data type of the same component data types, whereas a struct is an aggregate data type of disparate data types. So, for example, there's nothing stopping us from doing either:

```
struct someListOfNumbers
{
    int x1;
    int x2;
    int x3;
    int x4;
    int x5;
}
```

And that data would be stored in memory sequentially. In fact, theoretically you could (but don't, it's a really bad idea for maintenance and readability):

```
struct someListOfNumbers* s = malloc (size_of( struct someListOfNumbers);
int* arr = (int*) s;
```

```
for (int i =0; i < 5; i++)
     arr[i] = i;
printf("%d", s->x3); // should print 2
```

Similarly, we could do the reverse:

```
int x[5];
struct someListOfNumbers* structarr = (struct someListOfNumbers*) x;
structarr->x1 = 0;
structarr->x2 = 1;
structarr->x3 = 2;
structarr->x4 = 3;
structarr->x5 = 4;
printf("%d", x[2]); // should still print 2
```

The point, again, isn't that you should engage in what are typically called Stupid Programming Tricks to be extra fancy in your code (although, the annual http://www.ioccc.org is a good way to test skills for reading and writing code in a non-serious environment). The point is more to demonstrate that the data is literally just a bunch of 1's and 0's sitting around in memory, and that the order of them is pretty obvious.

The way that the computer will cache and store the memory, though, is a little more complicated. And, even though we call arr[] and someStruct->attribute O(1) operations, the reality is that if we're bouncing around between memory addresses or just using sequential memory addresses, the time is different in actuality. It's much easier to do for (int* p = somePtr; p < somePtr + n; p++) than trying to track down various random parts in memory.

This all brings us back to: when we train an AI and run lots and lots of calculations, it makes sense to save those nanoseconds when we can, and we can use a column-ordered data structure (the set of arrays) instead of a row-ordered data structure (the array of structs). It's basically a question of how you would like to sequentially access the data. And while this is a fairly cut and dried set up, there are lots of more murky situations. For example, how many NUM_DOGS would it take before it made more sense to copy the ->heightInInches value to a separate heightInInches[] array before processing? Probably there is no point for a simple int like this. But for processing some network graph, or evaluating portions of an image, or using computer vision on short segments of surveillance footage, it might start to become important.

And, in fact, what we see is that a lot of the DataBase Management Systems (DBMS) that target big data applications do exactly this: they store the data locally in column order instead of row order and that's one of the reasons why AI, Machine Learning, and Data Analytics tend to get specialized programming languages to interface with their specialized DBMSes.

CHAPTER 7 - MY ORDER! IT'S FULL OF TREES!

Let's talk about tradeoffs for another moment. We've discussed them several times by this point, but frequently, what we're asking is not a question of doing something better, it's a question of doing the same thing. It might be the Law of Equivalent Exchange, but that's a bit too rigid. So, let's consider some basic ideas that we'll come back to later when we discuss computational complexity. But there are two metrics that we typically have to worry about: space and time.

In order to store X integers, we need to use 4x bytes, because ints in C are set to 32-bit values. But, that's just the data sitting there. We've already seen that the swap function requires that we have N + 1 units of data in order to shift around N units of data. That's why our swap function had to have three values: To swap A and B, we needed a value C so that we could do:

```
void swap (int* a, int* b)
{
      int c = *a;
      *a = *b;
      *b = c;
}
```

So, in order to act on the static 2 integers, we needed a space of 3 integers. Further, we needed a time of 3 integers. This not a perfect example, because there's not really anything we can do to make this better or worse.

But what about when we were looking at arrays? we saw that we can try to save space by using exactly the amount of memory we need. For example, if an aggregate data type currently stores `n` integers, we could allocate an array of exactly `n` integers. This gives optimal space usage, but in practice it can be inefficient. Every time the data structure grows, we may need to create a new array, copy all existing elements into it, and then delete the old array. Instead, we often allocate an array with some extra capacity. So, even if the data structure currently stores `n` integers, the underlying array may have space for more than `n` integers. This uses more memory, but it saves time because we do not need to resize and copy the array every time a new element is added. This is an example of a tradeoff between space and time. It is important to remember that these decisions depend on which resources are limited, expensive, or available. For example, many businesses produce and store constant streams of data, but they may not analyze all of it immediately. In that case, storage may be relatively cheap, while computation may be more expensive.

Again, the point is that when we find a clever way to do something, it's frequently not an actual savings,

it only looks like it from a localized perspective. But from a broader scope, it's apparent that the tradeoff is coming from something else.

However, and this is the point: this is not always the case. The Philosopher's Stone for Computer Scientists is finding a calculation method that is either faster or uses less space without an increase in the other. We would call this an absolute optimization instead of a comparative optimization (in line with absolute competitive advantage versus comparative advantage in economics).

Binary trees are one of the fundamental data structures. Just by shifting the order that we store data in, we can reduce the computational complexity of certain sorts of operation, but it doesn't necessarily take up more space. Frequently, we choose to make it take up space, so that we don't also have the array resizing problem, but that's a separate issue.

Binary Trees

A binary tree is an Abstract Data Type that has a fairly simple description. It's just a hierarchical structure such that each node has at most two child nodes. That's it, everything after that becomes part of the implementation. In fact, we've already looked at this structure in a couple of different aspects. But, let's take a look at it a little bit more rigorously. Let's begin with some labels of convenience:

- **Left and Right** – These are typically the child nodes of any given node. They don't necessarily have to be filled, but these are the names we give them. Again, this is mostly convention, and because when we draw it on paper, we tend to draw it from the top of page to the bottom, and that means that each child node is leading off from the left or right side of their parent node.
- **Child** -A child is a node that comes directly below another node in the hierarchy. In this case, it's because the 'tree' kind of looks like a family tree, and It also fits naturally with recursive implementations, where a function may call itself on a node's children. A child `node` can be described as any node such that: `child == node->left || child == node->right.`
- **Parent** – this is the node immediately above a child node. Every node has a parent except the root. Same thing: when we draw it or code it, we call it the parent because it's what spawns the nodes below it in a diagram or at least the processes that handle it. We don't necessarily need to explicitly note the parent relationship, but it frequently helps.We can define the parent relationship as any node such that: `parent->left == node || parent->right == node`
- **Root** – The root node stands at the apex of the tree diagram. We wills say that it has no parents, but as we'll see what we really mean is that it has not parents in the current scope.
- **Branch** – another name for child nodes. Sometimes, we want to emphasize that a node only has a single child or a single branch off of it.
- **Leaf** – these are nodes that do not have any child nodes. leaf nodes are important because these are end

points in most algorithms, or base cases in most recursions.

- **Interior nodes** – These are nodes that are neither a leaf nor the root. They're just somewhere in the middle of the tree. These are what the bulk of the logic of any loop iteration or recursive call will focus on.
- **Full node** – A full node is a node that has exactly two children. Broadly, this is sort of the normal or expected situation in a binary tree. The exceptions are nodes with a single branch and leaf nodes.
- **Subtree** – A subtree is a tree formed by choosing some node and treating that node as the root of a smaller tree. For example, a node's left child is the root of its left subtree, and its right child is the root of its right subtree.Subtrees are especially important in recursive algorithms. When we call a function on a child node, we are often asking the function to solve the same problem on a smaller tree. This works because every subtree has the same general structure as the original tree.
- **Degree** – this is just a measure of how many child nodes any node can have. So, obviously, in a binary tree, the degree of the tree is 2, because any given node can have 0, 1, or 2 children.
- **Full binary tree** – this is a tree such that the degree of every node is 0 or 2 children. Alternately, a perfect binary tree is a full binary tree where the number of nodes is equal to $2^n - 1$.
- **Height** – The height of a tree is the length of the longest path from the root to any leaf. In most tree-based algorithms, we generally want the height to be as small as possible because a smaller height usually means fewer steps to reach a node. However, keeping a tree short may require extra work to rearrange or balance the tree, so there is a tradeoff.
- **Depth** – For any given node, how many links are between the node and the root. The root has depth 0. A child of the root has depth 1, and so on. Height describes the tree or subtree as a whole, while depth describes a particular node's position within the tree.
- Level – All the nodes that share a same depth. For example, the root is on level 0, its children are on level 1, and their children are on level 2.

Phew, that was a lot of random words! There's a reason we haven't done that so far. But, hopefully, we've addressed binary trees and linked lists enough that most of these terms should click into place for 'oh, that's what word to use to describe that' more than something conceptually new.

However, let's do a quick example of a specific type of binary Tree, the Binary Search Tree, or BST. We'll use a lot of these vocabulary words in context here.

Binary Search Trees

Notice the heading level. These are important enough to be on the same level as the generic concept of a Binary Tree. Remember in Merge Sort when we had that log(n) moment? What did we say, that we could find the

appropriate slot for a node in no more than log(n) comparisons, right? We can rotate that around and, given a sorted tree, we can find any given node in no more than log(n) searches.

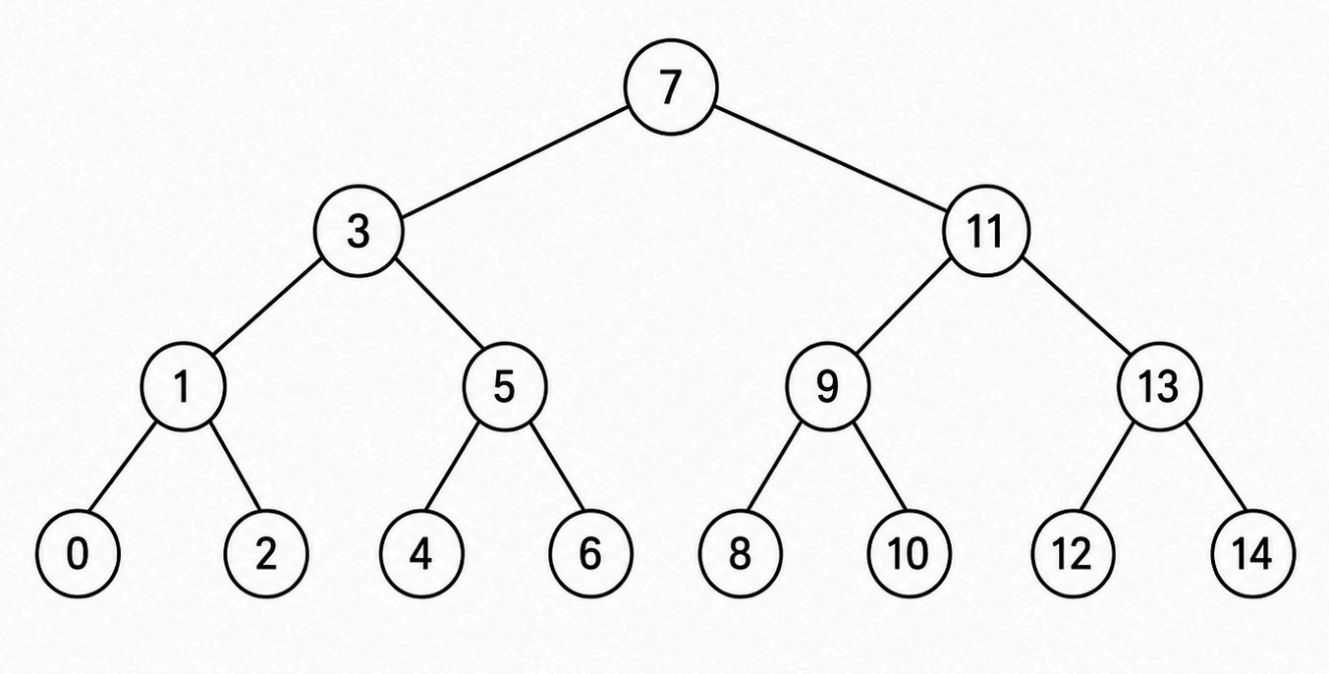


Fig 7.1 Binary Search Tree

Figure 7.1 shows a perfect binary search tree. It is a perfect binary tree because every level is completely filled. Notice that each full layer has (2^l) nodes, where (*l)* is the 0-indexed layer number. For example, layer 0 has 1 node, layer 1 has 2 nodes, layer 2 has 4 nodes, and layer 3 has 8 nodes.

What do the numbers represent? In a basic example like this, the numbers represent the actual payload stored in each node. However, they could just as easily represent links to other data structures, strings or individual words, indices in a larger array, or other forms of data. The values shown here are sequential for simplicity, but they do not need to be sequential in general. This example simply gives us a clear starting point for understanding the structure of a binary search tree.

Now suppose someone says, "Let's play a game. You can move from any node to any adjacent node by following the edge between them." An edge, in this case, is the line that connects two nodes. We will cover graphs later, but for now it is useful to know that a binary tree is a special kind of graph.

Suppose you always start at node 8, and the other person can choose any other number in the tree. They have to tell you the number they chose, and your job is to move from 8 to that number by following the edges.

For example, if the target is 3, the path is: 8 → 9 → 11 → 7 → 3

That takes 4 edge moves. If the target is 11, the path is: 8 → 9 → 11

That takes 2 edge moves.

This kind of question is about distance between nodes in a tree. It is slightly different from the usual way we talk about searching in a binary search tree. In a normal binary search tree search, we usually start at the root.

In this example, the root is 7. Because the tree is balanced and perfect, every path from the root to a leaf has the same length. That means any search starting at the root will take at most 4 node visits, or 3 edge moves, to reach a value in this tree.

This is one of the main advantages of balanced binary search trees. When the tree is balanced, the number of steps needed to find a value grows slowly as the tree gets larger. Instead of checking every value one by one, we can use the structure of the tree to eliminate large parts of the search space at each step.

Balanced Binary Search Trees

This is the important part: when we have a balanced binary search tree, we start to approach log(n) comparisons in order to find any node. This is the ideal state. However, it takes work to get there! What if we have to add another value: 15. We could add it to the first spot on the fifth layer. In that case, our worst search time is now 5, because there's one case where it takes us 5 hops to get there. But, even that one case is enough to qualify for 'worst case scenario'. The formula gets a little intense, it's:

```
# comparisons = floor(log(n) + 1)
```

So, log(15) was 3.91, but our worst case was 4. why is that? Because we can't have a partial search. It's a discrete action, meaning that we either made a comparison or we didn't. So, what we really want is to round that value up. But! That won't work either, if we have ceiling(log(n)), then what about when we added that 16^{th} value in the previous paragraph? Now our worst case scenario is 5 comparisons. log(16) is 4, and ceiling(4) is still 4. So, we have to add 1 to it, but that means that we're now on the other side of the divide and have to floor to get back to our discrete value. The equation feels messy, but it's correct. Technically, we should add a divide-by-zero check in there of 'for all values n >= 1', because log(0) is undefined and log(<0) is a negative value, where having a negative number of comparisons is nonsensical.

Side Note: However... does it really matter? If you go to your boss at work and say 'Look, I found out how to get free power!' and you take a power strip and say 'see, these parts provide power, right? And this plug requires power, so we can just plug it in here!' Everyone should look at you like a crazy person or assume you're not serious, right? So, if you write out a quick formula and say 'such and such log(n) for all n', but technically it should have been for all n > 0, for the most part, the answer wouldn't have made sense anyways, because negative answers don't make sense. In practice, there's a presumption that nonsense answers don't need to make sense. So, you've got one of two approaches to take and to recognize as part of an organizational culture: either it's important to be precise and include the specifics like the constraints of an answer because that's the seriousness of the task or making obvious statements like 'don't take the logarithm of 0' is a waste of time and the company doesn't pay you to waste time. Both are valid to a certain degree.

Regardless, the point is that this worst case scenario description only necessarily holds true for a balanced tree. For example, when we put that 16^{th} node in, what if we put it in as the root, instead? Now, we've got a 5

layer deep structure, still but every search will take 1 extra comparison. Further, what if we add a 17th node? If we add the 17th node to the 5th layer we maintain the balanced node. but if we shove it in at the root, we have layer 0 with the value 16, layer 1 with the value 15 on the left, layer 2 with the value 7 on the left, and the rest continuing as normal, but only on the leftmost branch. Now, our search will take a max of 6 searches!

We could take this to an extreme, what if all the values were always stored in ->left, and ->right was always null. Then our worst case scenario would be on level n with n assigned nodes and n possible worst case comparisons. Suddenly, we're looking at a linear search and we could have just used a regular linked list, right? And, in fact, what we find is that there's a spectrum: one end has a linked list with the worst case scenario for the structure of n elements, and the full and complete Binary Search tree is the optimal structure of n elements.

Your spidey senses should tingle for a moment. I said 'optimal' and didn't define my metric. In this case, we mean specifically optimal with regard to search time with a binary search algorithm.

Height

Ok, so what about height. Notice that the worst case number of comparisons when searching in a tree has consistently been the node with the most depth. And, that's what we find is that the depth of the node with the most depth = worst case search time in binary search = floor(log(n) + 1).

Why a Linked List at all, then?

Here's a valid question: Trees are not that complicated. Why, then, do we even bother teaching Linked Lists? Sure, it would take a little more effort to describe two pointers, but that's not necessarily worth the effort. We could say for the historicity, but there's lots of algorithms that used to be popular (Threaded Trees) that we don't even touch because they're kind of silly. We could go with prevalence in practice, but really outside of some systems level applications they really aren't that common. So, then, there has to be a practical reason why we would have Linked Lists in the first place, right?

The short answer is mostly: because they are used for Stacks and Queues. But we can expand that answer to ask questions about insert. How long does it take to insert a node into a stack or a queue? O(1) time, right? Literally 3 operations (which, remember, is the same as constant time, which simplifies to just O(1)). How long does it take to insert a node into a sorted binary search tree? That phrase is actually mostly redundant, as a binary search tree is sorted, by default. The answer is O(log(n)).

We haven't covered the specific algorithm, but we saw that it was the case in the merge sort that the number of comparisons to find the 'slot' for the next unsorted elements was O(log(n)), right? You could, if you were inclined to experiment, try to figure out the algorithm on your own before we get to it. It is no shame not to be able to figure it out. Remember that someone, at some point, invented most of these ideas. For example, we could generally point to 1953 as the introduction of general hierarchical structures, but the paper by Adelson-

Velsky and Landis that is generally credited with defining Binary Search Trees didn't come out until 1962. That's 9 years it took for the entirety of the computer science industry to invent what is now a relatively basic concept. The dates aren't specifically important (although, in this case Donald Knuth is one of the modern heroes of CS, and Trees was his area of expertise in the early 60's), except to compare how long a solution took to come around. If you can figure out any idea in this book on your own before we get to it, you should be pleased with yourself. We're doing our best to be leading, but that doesn't change the fact that it's akin to solving an Agatha Christie novel from first principles instead of just knowing that The Butler did it.

We have two possibilities: insert of O(1) or insert of O(log(n)). The solution should be obvious, right? We want O(1), so why then do we have trees? Well, what about finding a specific element. First off, we can't do that at all in a stack or a queue. But let's relax that a little bit and switch to searching for a specific element. In the stack/queue/linked list option, it's O(n), but in the Binary Search Tree, it's O(log(n)). So now it's obvious the other way! We want trees!

Once again, we have to return to the idea of tradeoffs. One of the most basic ideas in software engineering is a needs analysis. You need to know what the user is going to be doing and what the practical outcome is. For example, take a database of prices of produce at one of the mega grocery stores. Which is going to happen: a price change happens requiring writing a new entry to the database, or someone buys a piece of produce requiring a lookup in the database. Obviously, the latter is going to happen much more often. Ideally, on the order of thousands (millions?) of times more often. So, in that case, the speed of lookup is vital, because looking things up happens in real time, but storing new data probably happens while the store is closed or during inventory or during system maintenance or any other time when responsiveness isn't an issue. In that case, having a robust system that optimizes for search time is better (it won't actually be a BST, Binary Search Tree, it'll be a more advanced and detailed variation).

On the flip side, let's consider something like a system log for a high security area at a large venue. They might have hundreds or thousands of people coming and going, triggering dozens of RFID sensors to track movement, and that data needs to be stored in real time. But the only time you'd actually review those logs would be for an anomalous event that requires time to evaluate. We want the storage to be fast and are OK if the reading takes a little longer as a response. In that case, we'd optimize for storage time and not worry about search time as much.

Hopefully, you're catching on that while there are a few instances of better or worse, for the most part we're looking for tradeoffs. We're willing to compromise on metrics we don't care about in order to increase metrics we do care about. One might argue (it would have a significant amount of truth, but not be entirely correct) that the majority of algorithm design cutting edge development is about trading low-value resources for high value resources and determining how to best do that more optimally or cutting closer to minimum standards without crossing them.

Side tangent: Consider the Busy Beaver problem. In short, imagine a very simple Turing machine that writes only 1s and 0s. Given a machine with (n) possible states, we ask: what is the longest number of steps such a machine can run and still eventually halt, rather than running forever?

This question is closely related to the Halting Problem. The Halting Problem asks whether there is a general algorithm that can determine, for every possible program and input, whether that program will eventually stop. Turing proved that no such general algorithm exists. The Busy Beaver problem gives us a concrete way to see how quickly this kind of question becomes impossible to solve by ordinary computation.

For small values of (n), the problem is manageable. The known runtime Busy Beaver values are (S(1)=1), (S(2)=6), (S(3)=21), (S(4)=107), and (S(5)=47{,}176{,}870). The jump from 4 states to 5 states is enormous. Even though the machines themselves are extremely simple, the number of possible machines grows very quickly.

At first, this looks like a brute-force problem: list every possible machine, run each one, and keep track of the longest-running machine that eventually halts. But the hard part is not the machines that halt. The hard part is proving that the remaining machines will never halt. For (n=5), researchers did not simply run every machine forever. They used a large collaborative effort, symmetry reductions, specialized algorithms called deciders, and formal verification to prove that all remaining cases were handled. So the Busy Beaver problem is a useful reminder: sometimes the algorithmic challenge is not just "run the program faster." Sometimes the real challenge is proving whether a computation will ever finish at all.

The point is not to detract from the solution, but in this case, what was precious was algorithmic elegance (as in, we don't have a more elegant algorithm than brute force, yet) but what was relatively cheap was social connections and the spare computation capacity of a crowdsourced mob. Thus, the meta-algorithm was crowdsourcing, and that was the correct solution from a strategic sense.

Making Sorted vs. Keeping Sorted

This brings up an excellent question about trees. It's really about all data types, but since we're talking Binary Search Trees, it's particularly poignant. What's the average search time for any given node in a Binary Search Tree? We've already established that it's O(log(n)), right? But that comes with strong assumptions. Specifically, we assume that the array is sorted. This brings up an interesting question: how are we insuring that the tree is in sorted order at all times?

The answer is pretty obvious: when we insert a node, we do an sorted insert. That is, we don't put the new node in at the beginning, end, or random location like we might with a regular sort. Instead, we specifically find where the node goes and insert it there. We've previously said that this will take O(log(n)) to accomplish, but what does that mean? We've got a few special cases:

- The tree is null. In this case, analysis doesn't really matter, because we'll just put the node in the only place possible: the root.
- The new node goes into a new leaf spot. In this case, we put it in the beginning or the end, and it should be a direct log(n) traversal steps to get to where it goes.
- The new node goes into a random intermediate node. Well, we have a problem here, because if we've been

> building our tree correctly, all intermediate nodes already have 2 child nodes. So, the new node will have to take the place of the interior node, and the previous interior node will have to become a child node, but that means that child node might also be an intermediate node and need to supplant one of it's children, etc.

We aren't going to get into a solution for this right now. But there are a few tools at our disposal, some new and some old. But, generally, we split it into two steps, the insertion, and then another for rebalancing to make sure that we minimize the height of the tree. Insertion into an ordered tree is log(n), and rebalancing the tree is log(n). This means that we consider the whole operation to be log(n).

But let's ask another question: if we're inserting nodes into a tree, we have to insert n nodes, where n is the number of nodes in the tree. Inserting nodes into the tree is an O(log(n)) operation. And we have to do it for n nodes. So n*O(log(n)) operations is an O(nlog(n)) operation. Which, in turn, means that assembling and maintaining the order of the tree is no cheaper than just sorting the tree in the first place. So why would we bother? And the answer comes in the iterative nature. If we already have the sorted tree, it will only take O(log(n)) steps to insert and maintain order. But if we want to sort the tree, then all of the O(nlog(n)) steps need to happen at once.

This is a bit superficial, and it proves what seems intuitive: maintaining the sorted quality of a tree over its lifetime is as expensive as sorting it in the first place. However, what's more important is that we haven't lost anything: maintaining the sort order is, within a small tolerance, no more computationally expensive than occasionally sorting. So, what this tells us is that whether we maintain a sorted tree or sort an unsorted tree is entirely application dependent. But, we typically have a preference for retaining a sorted tree because the fringe benefit of being able to search in log(n) time is powerful.

Balance

We are also introducing a new constraint here. We've mentioned it several times. The balance of the tree is how even both branches of the root are. There are specific metrics that we'll go into in the next chapter. But, for now, what we want is to maximize balance (or minimize the imbalance). We'll use terms like rotation or restructuring to make the tree more in line with what we need in order for these sorts to work.

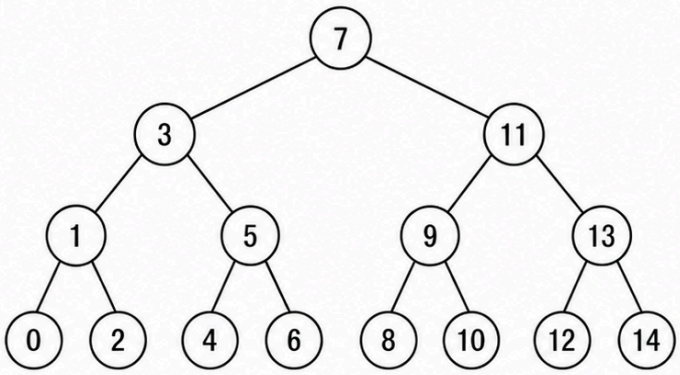


Figure 7.2 a

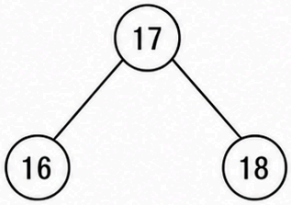


Figure 7.2 b

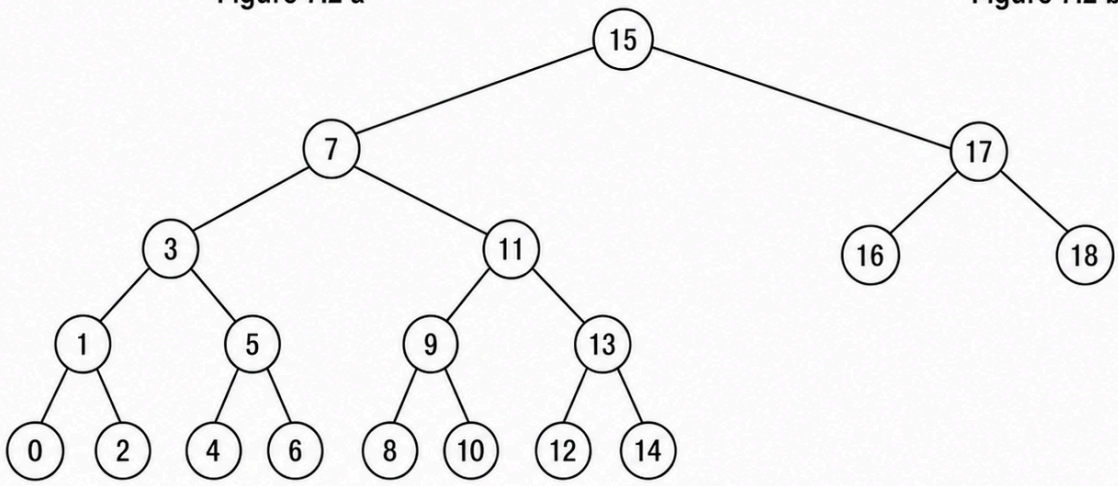


Figure 7.2 c

Consider Figure 7.2 a & b Now suppose we take the roots of both trees and make them children of a new root node at layer 0: 15. The larger tree rooted at 7 becomes the left child of 15, and the smaller tree rooted at 17 becomes the right child of 15.

This gives us an imbalanced binary search tree figure 7.2 c. The left side is taller than the right side. However, in this small example, the imbalance is not a serious problem. The deepest value on the left side can still be found in no more than 5 comparisons, including the root node 15. That is still much better than using a non-tree structure, where searching through all 19 values one by one could take up to 19 comparisons.

So a little imbalance is fine. The real problem appears when the imbalance becomes larger and larger. What if the right side has height 4, but the left side has height 8? That may still be acceptable. But what if the right side has height 4 and the left side has height 400? At that point, the tree still follows the binary search tree rule, but the worst-case search becomes much slower. This is the main issue balanced binary search trees are trying to solve. The question is not whether the left and right sides are exactly equal. The question is whether the height of the tree is still small enough to keep searching efficient.

There are special cases where imbalance can be useful. For example, in frequency-based encoding such as Huffman coding, more common values are intentionally placed closer to the top, while less common values may be deeper in the tree. But in the general case, we want to prevent one side of the tree from becoming too deep. Otherwise, the tree can start behaving more like a linked list than a search tree.

The short of it is that there isn't a good general answer. And in the case when we don't have a good general answer, we tend to revert back to risk averse analysis. So, because we are giving a general answer, we want to say that the tree should always be balanced to minimize search time and insertion time. That's a fine answer, and more specifically it maximizes the chances that we can fit a search (or repeated searches) into a given frame of computation time for, say, an interactive application.

But, we can give a more self-referential answer. Rebalancing the tree on an insert is an O(log(n)) operation. But rebalancing the tree after an unknown number of inserts and an unknown amount of imbalance is an O(n) operation. So, we have to consider the fact that if there are 1 million values in the tree, we may be trading out momentary O(20) operations for a later O(1,000,000) operation. That might be OK, depending on the context. But, on the flip side, it may also push the constraints that the program needs to operate in.

We will return to Balance, but for now recognize that it is frequently a desirable attribute to build and maintain in a Tree.

Challenge and Definition

Notice we haven't gotten to much code. This is a challenge that you should try but not expect to succeed at. It's within your skillset by this point, technically. But that 'technically' is doing a lot of heavy lifting. It would be a good experience for you to attempt to write a Binary Search Tree before we cover the code, and then you can compare yours to ours. What should a Binary Search Tree have?

- The struct we gave above, possibly modified for a different data type.
- Insert – this will maintain the sorted and balanced nature of the BST
- Search – this will return the value in no more than log(n) time.
- Delete – this one is optional. You don't have the algorithms or tools to delete and maintain the sorted nature, yet. But you could either look ahead for them or you could basically recreate the tree every time you delete a node (for now).
- Traversal – We'll cover this next, but an example of this would be printing each node in order.
- Min – return the minimum value in the tree.
- Max – return the maximum value in the tree.

Traversal Order

We've been sneaky so far. We keep talking about traversing the tree. But what do we mean by that? This is the wrong example, and in fact, we're going to come back to this in the Graphs chapter, because the concept has application across multiple different applications. But, it's most applicable to Graphs. Until then, let's consider how we might traverse the tree with the analogy of a maze. The tree isn't really a maze, but the rules for beating a maze are the same. How do you get to the end of the maze? You probably follow the rule of keep your hand on the wall. But, which hand, left hand or right hand? What if we told you that it didn't matter, if holding your hand on the wall would get you to the exit, either hand will work, there's just a 50/50 chance that one hand will be faster than the other but you can't know that ahead of time.

What are we really doing when we say 'keep one hand on the wall'. We're giving a set of priority instructions for how to make turning decisions. What if, however, instead of finding the exit, we wanted to visit every part of the maze and the priority decisions about which direction to go dictated the order we'd visit them. So far, none of this matters, because we haven't distinguished between different parts of the maze and suggested why the order would matter. That's where the analogy breaks down, so leave it there for now (as mentioned, we'll come back to it in the Graphs chapter, where we'll also solve circular paths that never actually lead to the exit).

When we traverse the trees, we have three real options (aside from things like 'don't visit any node', or 'visit the first node infinitely', we mean 'real' as in the only ones that matter). They're named very sensibly:

- preorder traversal – In preorder, we 'visit' the root node, and then we preorder traverse the left child and then we preorder traverse the right child.
- inorder traversal – In inorder, we sidestep around the root node, and then we inorder traverse the left child before we 'visit' the root node, and then we inorder traverse the right child.
- postorder traversal -in postorder, we sidestep around the root node, and then we postorder traverse the left child, and then we postorder traverse the right child, and then we 'visit' the root node.

This seems a little arcane, why would we do this? It's a question of writing style (we're rearranging the problem, we'll get to the ultimate why in a moment):

- infix notation – When we list operand operator operand, we're using infix notation. So, for example, 6 / 2: 6 is the left child, / is the root, and 2 is the right node.
    - So, we visit the left node first, get a 6, add the root node, get 6 /, and then visit the right node for 6 / 2. (Six Divided By Two)
    - This is how humans talk and think and write, for the most part.
- prefix notation – When we write some procedures and when the compiler processes operators, it convert everything to prefix notation. For example: Divide (6, 2)
    - So, we visit the root node, get the divide operator /, visit the left child to get the numerator or first

argument / 6, and then we visit the right node to get the denominator / 6 2. (Divide Six By Two)

- postfix notation is dumb – We use it sometimes in functional languages, and some people swear that it's super useful because you can read it linearly and prepare your brain for the appropriate operation.
    - When we have a stack, sometimes it's cleaner to pop off the two operands before popping off the operator. Alternatively, if you pushed postfix into the stack, when you read the stack like an array, you'd see prefix notation.
    - When you study formal logic, you'll be exposed to certain types of problems that don't bother the rest of us mere mortals. Postfix solves some of them.
    - We visit the left node to get an operand 2, visit the right node to get the other operand 6 2, and finally we reach the root node and get the operator 6 2 /. ("Grunt grunt, six! two! Dividum!" there's not actually any English sentence that makes it work.)
    - It's also called Reverse Polish Notation, which sounds like a joke but it's not. In contrast, Gary suggested the following Shakespeare: "Out, damned stack! Out, I say! For when the operand precedes the act, what ghostly register remains?!" And that about sums it up, if you think about it.

Now, if you've studied Context Free Grammars even a little bit, you'll notice that these structures seem a lot like rules for a CFG. And, in fact, that's partially what we're trying to create. You'll notice that we made a subtle shift: so far we've only been using numbers. But now we're using numbers and operators. Let's presume that the basic pattern is prefix notation: Operator (Operand, Operand). What's an operator? Typically a function or one of the operator symbols (which is just code for a symbol). Sometimes we might have unary operators or ternary operators that handle things a little differently, but leave those for later consideration. Just recognize that your entire program is made up of operators and operands, at some level (that level is the point when the compiler starts refactoring the parsed tokens).

We also need to ask what constitutes an operand. That's a little more complicated, but we could simplify it and say that an operand is any number or the result of an operator clause. So, we're getting some recursion in here.

We could ask a few questions like why we only have three when we would have six if we did the right child first sometimes. The answer is yes, that's true. But, what you'll find is that distinction is practically meaningless. Like distinguishing between Thing 1 and Thing 2 in The Cat in the Hat, there's no practical distinction between left and right. It's just a distinction of convenience and they're otherwise indistinguishable. In practicality, what we're really doing by distinguishing between left and right in the various notations is literally distinguishing between 'the one we do first' and 'the one we do second'. So, it's not that we don't have right-first notations, it's just that they resolve into the exact same pattern and only 3 are unique.

This Just In: Monty Python Say "Get On With It!"

What's the point? Just like anything else, trees can store lots of things. But, that's not the point. We've covered two ideas:

1. When you 'traverse' your tree, you need to pick a method to do so, and there are three options.
2. When we organize data into the tree structure, to a certain extent, the root is intended to resolve, in some way, everything below it, even if that is just by concatenating data. But, we can do many more things with time.

Trees are one of the most powerful tools you have at your disposal. We'll see later that linked lists are just constrained trees. Binary trees are also just constrained trees. Trees in general are just constrained graphs. And graphs are just constrained sets of relationships. (all of these amongst many other interactions) If we want more fidelity with the real world, degrees of freedom, or unconstrained calculation, we moved higher on the list. If we want to simplify problems, bound complexity, or increase our ability to confirm our findings, we move down on that list. Trees are frequently a nice balance point between all factors. That last statement is worth the amount of electrons it took to write it, because it doesn't give any clear indication of when to use trees. Just, consider tree structures early if you need to use fundamental data structures.

Side Tangent: Have you done any game design? In a modern game design engine like Godot, Unity, or Unreal, have you noticed how the Scene Editor tends to be a hierarchy? How do you think they model that hierarchy internally?

For a pure theory issue, consider some of the smartest people we've had historically. What did they spend their time on? Donald Knuth is one of the most influential Computer Scientists. He wrote The Art of Computer Programming and has served as a figure of awe for generations of computer scientists. He spent decades studying tree data structures.

So, we've got a few more things that we'll come back to later:

- Which traversal order makes the most sense for a Binary Search? There's only three, consider the consequences of picking any.
- There are two more tangential ideas to traversal order: Breadth First versus Depth First. We'll return to this idea in Graphs.
- What happens when your data isn't balanced, won't stay sorted, needs to be sorted in multiple ways, or doesn't even come close to tree structures?

Printing

By now, we have seen that traversal notation tells us the order in which we visit the nodes of a tree. That also means it tells us the order in which the data stored in those nodes will be printed. For example, suppose we use

the binary search tree from figure 7.1. If we print the tree using **in-order traversal**, we visit the left subtree, then the current node, then the right subtree. For a binary search tree, this gives us the values in sorted order:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
```

That is one of the most useful properties of a binary search tree. The structure of the tree already stores the values in a searchable order, and in-order traversal allows us to print those values from smallest to largest.

Recursive Tree Thinking

We've gone a bit far afield. But it's important to grasp that Binary Search Trees are important. But they're the beginning. This means that the minute you say a constraint like 'stays sorted', someone goes 'ok, but what if not that'. And if you say 'it has to stay balanced!' and someone else goes 'ok, but what if not that?'

So, all of this brings us to some basic principles of a tree. And those principles can be aggregated into one idea: All trees are fractals. Every node is the root of a tree that has the same fundamental properties as the whole tree. This is true of many data structures, but in a lot of ways it's most interesting with a Tree.

So, how does this apply? Consider our notation examples from a moment ago. What is in the tree? You have two options:

- A node with the divide operator in it and two children: 6 and 2.
- A node with the number 3 in it and no children.

Which one is correct? Not only are they both correct, but from the perspective of the tree, they're exactly the same! For as much as the traversals and the notations might seem different, notice one thing: they retain the order of the operands, and the importance of the operator over the others. But, we could apply it in other contexts.

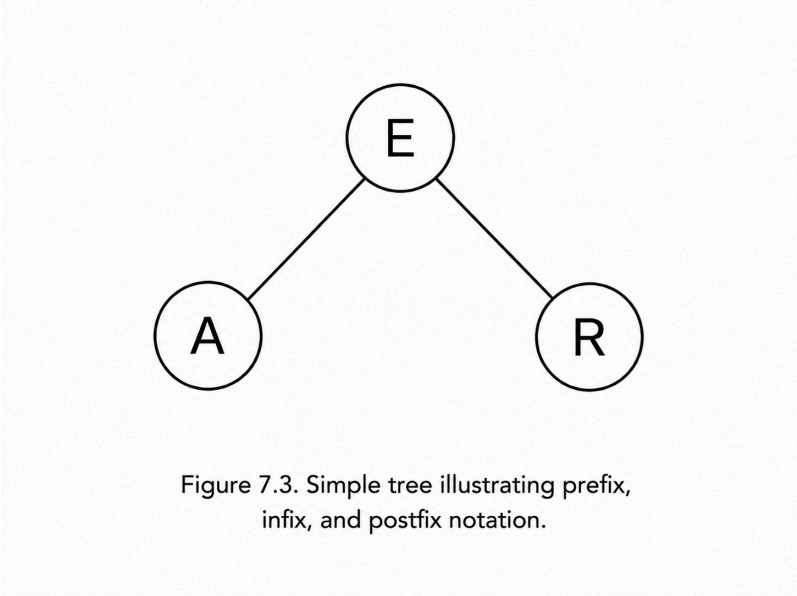


Figure 7.3. Simple tree illustrating prefix, infix, and postfix notation.

Consider a tree in Figure 7.3 , In prefix notation it spells 'EAR', the thing that brackets most peoples noodler. In infix notation it would be 'AER', as in a pseudo-magical fantasy term for elemental air. In postfix notation, it would be 'ARE' as in 'what are you doing allowing the previous word!?'

So, here's a good challenge: What's the shortest tree you can create and the longest tree you can create such that all three words: prefix traversed, infix traversed, and postfix traversed are all valid words (say, according to the Scrabble dictionary). Shortest starts to get fairly easy. There's only 17,576 possible 3-letter combinations, 456,976 possible 4-letter combinations, etc. But how would you go about proving what the longest one is? If only there were a way to organize words by the number of letters they had...

Regardless, the point is that as you evaluate each root of a subtree, you're effectively aggregating the entirety of that tree into a single concept. This also means that many things (we'd say 'everything', but obvious things like 'size' won't stay the same) that is true of the whole tree is also true of each subtree. The mechanism for evaluating, changing, substituting, balancing, sorting each of the subtrees applies to the main tree.

What does this have to do with recursion? Your base case is ,

```
if node.left == NULL && node.right == NULL: return node.payload
```

Your different traversals are:

```
return someConcatenationFunction( traverse(node.left), node.payload, traverse
```

Recursion non-tree thinking

As we've seen, the Binary Search Tree is useful for a lot of things, but two of the biggest benefits are:

- It provides you with a way to reach any known node in faster than linear time (O(log(n)).
- It provides you with a set of evaluative aggregations that allow you to collapse complex sub-trees into single values.

Now, what if we could combine those two benefits with the advantages of the Random Access nature of an array? Turns out, other people have asked the exact same question! And the result is a series of mappings from the linked version to a linear array version that retains most, if not all depending on application, of the benefits of the original. We've already explored this a bit with the Min-Heap from the previous chapter.

The Question of Tradeoffs

This is also a good time to return to the question from the beginning of the chapter. What are the tradeoffs of using a tree rather than just a standard linked list or an array? Is it an overall improvement, or what out-of-demand resources are we trading for what in-demand resources by using a tree structure?

Binary Search Tree Code

There's a few things that we need to take note of. We'll cover the pointer code and then jump into the array version of the code. This is likely the last time we'll do this. The data structures are pretty standard in the abstract with pointers and such. But, as we've already noted, being able to port it to other languages or use it in a more practical sense will largely rely on using arrays or a similar data type. There are tradeoffs to everything, true. But the tradeoffs for using the abstract-correct pointer setup is almost always more speed and memory-safety than is worth the tradeoff in the extra space and possible inconvenience of the array-based solution. In short, after this, we'll only give one implementation and trust that you can recognize that a conversion is possible and could do the conversion with some preparation based on application, need.

Pointers

```
struct node
{
     int payload;
     struct node* left;
     struct node* right;
     struct node* parent; // 1
};
```

```
struct node* insertBST (struct node* root, struct node* target) // 2
{
     if (root == NULL)
     {
          root = target;
          root->left = NULL; // 3
          root->right = NULL; // 3
          root->parent = NULL; // 3
          return root;
     }

     if (target->payload < root->payload) // 4
     {
          root->left = insertBST(root->left, target); // 5
          if (root->left != NULL) root->left->parent = root;
     }
     else
     {
          root->right = insertBST(root->right, target);
          if (root->right != NULL) root->right->parent = root;
     }

     return root; // 6
}

struct node* searchBST (struct node* root, struct node* target)
{
     if (root == NULL) return NULL;
     if (equals (root, target)) return root; // 7

     if (target->payload < root->payload)
          return searchBST(root->left, target); // 8
     else
          return searchBST(root->right, target); // 8
}

struct node* minBST (struct node* root) // 9
```

```
{
     struct node* curr = root;
     while (curr != NULL && curr->left != NULL) // 10
          curr = curr->left;
     return curr;
}

struct node* maxBST (struct node* root)
{
     struct node* curr = root;
     while (curr != NULL && curr->right != NULL)
          curr = curr->right;
     return curr;
}

struct node* deleteBST (struct node* root, struct node* target) // 11
{
     if (root == NULL) return NULL;

     if (target->payload < root->payload)
          root->left = deleteBST (root->left, target); // 12
     else if (target->payload > root->payload)
          root->right = deleteBST (root->right, target); // 12

     else // 13
     {
          if (root->left == NULL) // 14
          {
               struct node* temp = root->right;
               if (temp != NULL) temp->parent = root->parent; // 15
               free (root);
               return temp;
          }
          else if (root->right == NULL) // 14
          {
               struct node* temp = root->left;
               if (temp != NULL) temp->parent = root->parent; // 15
```

```
                free(root);
                return temp;
           }

           struct node* next = minBST(root->right);
           root->payload = next->payload
           root->right = deleteBST(root->right, next);
     }

     return root;
}

void inorderBST (struct node* root) // 17
{
     if (root != NULL)
     {
           inorderBST (root->left);
           handle (root->payload); // 18
           inorderBST(root->right);
     }
}
void preorderBST (struct node* root) // 19
{
     if (root != NULL)
     {
           handle(root->payload);
           preorderBST (root->left);
           preorderBST (root->right);
     }
}
void postorderBST (struct node* root) // 19
{
     if (root != NULL)
     {
           postorderBST(root->left);
           postorderBST(root->right);
           handle(root->payload);
```

```
        }
}
```

We've got a few branching comments:

1. We can implement it with or without the parent node. In our opinion, the vast majority of cases will have the extra overhead to allow you to include ->prev or ->parent or whatnot. There are a few cases where it makes sense not to. And there are a few systems where the memory and computation capacity margins are so narrow that it might not be feasible. But both of those situations should be considered edge cases, and your code should very much more be customized to adhere to specific goals. So, the most utility comes from presenting the simpler code with the back links. That being said, this particular implementation doesn't use ->parent, but that's by happenstance not by design.
2. We pass the root, but then we also return the root node. One argument would follow the philosophy of David Hume. In this case, we can semantically draw a link and intuitively see that the tree is basically the same just with some small changes. But, by another sense, we're also creating an entirely new tree that is foundationally different and only happens to coincidentally carry many of the same nodes and edges as the previous tree. Point is, it is human nature to think that changing something slightly is faster and better than reconstructing it from scratch. But in computing, as we get further and further advanced, it becomes more likely that it's as fast and sometimes more accurate to just create the entire structure from scratch. (For example, digital cameras and smart phones no longer take pictures. They estimate the scene based on input from an optical device that's no longer relying on pure optics.) So, like Hume derived from Hiraclitus, we can never dip into the same river twice.
3. We can either presume that target come initialized with everything NULL or we can specify it here. This may be wasted computation. But, adding extra security checks is never– let's rephrase that: within reasons, adding extra security checks is probably not a substantive waste of time. In this case, we included it because you would want to do so when creating a library when you aren't sure how the end user is going to use it. Further, it gives us a chance to remind you to do your NULL checks.
4. The presumption here is that the smaller 'half' of the tree is going to go to the left. That can just as easily be reversed, as long as it's consistent.
5. The logic here is simple. Is the new node smaller than the current root? Then it belongs somewhere in the left subtree. Otherwise, it belongs somewhere in the right subtree. This is the central idea of a binary search tree. In a balanced tree, this decision removes a large portion of the remaining search space. In an unbalanced tree, however, the benefit may be smaller, because one subtree may be much taller than the other. The recursive call keeps moving down the appropriate side until it reaches a NULL child pointer, where the new node can be inserted.
6. Again, we return root. This is important because recursive insertion may change the root of a subtree. For example, if the subtree was originally NULL, the inserted node becomes the new root of that subtree. Returning root allows each recursive call to reconnect the updated subtree to its parent. This is why root-

>left = insertBST(root->left, target) or root->right = insertBST(root->right, target) works correctly.

    - This does bring up a potential problem we aren't addressing: How do we know that the node that we're inserting only has a single node in it? How would we handle it if we got passed an entire multi-node tree? You should be able to solve this by using a completely different argument for the insert function (think payload, rather than node), by rejecting the invalid argument through early checks, or by parsing through the tree and inserting iteratively. Is there a faster way than iteratively (O(n)) to do the latter?

7. This is a bit of hand-waving. Why is it that we'll check inequalities with a <, but the equality gets a its own function? For both of these, it came down to ease of reading. But you should be sure to check your code and make sure it's consistent. In this case, both are acceptable, but not mixing between them. In the general sense, it's probably (though not definitely) more reusable to dump all of the boolean comparisons to separate functions for a deep comparison instead of a shallow comparison.
8. This is the heart of the benefit of a binary search tree. At each node, we compare the target value with the current node's value. If the target is smaller, we search the left subtree. If it is larger, we search the right subtree. In a balanced BST, this process can reduce the remaining search space very quickly, giving O(log n) search time. However, in an unbalanced BST, the search path may become long, and the worst-case time becomes O(n).
9. We include minBST() and maxBST() because they are useful operations on a binary search tree, and minBST() is also needed for deletion. The minimum value in a BST is found by repeatedly following the left child pointer. The maximum value is found by repeatedly following the right child pointer. These operations run in O(h), where h is the height of the tree. If the tree is balanced, this is O(log n). If the tree is badly imbalanced, this can become O(n).
10. This function deletes the first matching node it encounters based on the comparison logic. If we wanted to delete all nodes with a matching value, the complexity would depend on how duplicates are stored and how many matching nodes exist. Deleting one node costs O(h), where h is the height of the tree. If there are m matching nodes, deleting all of them may require repeated searches and deletions. In a balanced tree, this could be approximately O(m log n). In a badly imbalanced tree, it could be much worse. So, for this chapter, we focus on deleting one matching node. Handling duplicates or deleting all matching values requires an additional design decision.
11. There are five cases we need to deal with. The root == NULL case occurs when we recursively reach the end of where the node would be and it is not there. The second case is when the current node is not the target, so we continue to the left or right. The remaining three cases occur after we find the node: it has zero children, one child, or two children. The delete function is the longest because it has to handle each of these cases.
12. The else precludes any of the previous code.

13. The if (temp != NULL) check is needed before assigning temp->parent. If temp is NULL, then trying to access temp->parent would cause a NULL pointer error. It is true that in C, calling free(NULL) is safe and has no effect. However, that is not the main issue here. The important point is that we must not dereference a NULL pointer.
14. When the node has two children, we replace its value with the smallest value from the right subtree. This node is called the inorder successor. The original copyNode(root, next) placeholder should copy only the payload or key data from next into root. It should not copy the left, right, or parent pointers, because copying those structural pointers could corrupt the tree. In a simple integer example, the code can be written as root->payload = next->payload. After copying the payload, we delete the original successor node from the right subtree.
15. For a binary search tree, inorder traversal is the traversal that prints the values in sorted order. It visits the left subtree, then the current node, then the right subtree. Preorder and postorder do not print a BST in sorted order, but they are still useful. Preorder traversal can be useful for copying or serializing a tree. Postorder traversal is useful when freeing or deleting an entire tree, because it processes the children before the parent. So, inorder traversal is the most important traversal for sorted BST output, but the other traversal orders still have valid uses.
16. handle() is just a quick 'do something' function that might very well just be 'printf()'. But, this way we can avoid the formatting and it becomes a little more obvious where to swap out custom handlers.
17. Again, preorder and postorder don't matter. Could you design a BST so that they were valid for preorder and postorder? The answer is 'yes, but don't bother it's worse in almost every respect'. But, it's worth thinking about it in case you think of a unique situation in which i might be helpful or if you come up with a novel conceptualization (that's good science: new things get a novel method or novel data, but never both, and never neither).

There is one huge and glaring issue with this implementation. For all that we talked about balance throughout this chapter, this code doesn't balance the BST! It turns out that the actual implementation is a whole separate field of study for maintaining balance versus maintaining sortedness. So, consider this code to be a baseline. Is the extra time in maintaining precise order worth the extra overhead compared to the priority queue from an earlier chapter? Failing that, consider whether the extra overhead is worth it for maintaining balance in the next chapter.

On Moving From Pointer to Arrays

As we've mentioned with arrays, in practice we want to take advantage of both the advantages of a pointer-based structure while at the same time maintaining the advantage of the random-access arrays. However, whereas linked lists, stacks, and queues are very rarely actually implemented with actual pointers in practice, trees do regularly get 'properly' implemented. We would still consider it uncommon, just not as rare as with

the linear structures. Which makes sense, because the linear structure is literally just a constrained array: in a linked list you can only access the next or previous, in a stack you can only access arr[0], and in a queue you can only access arr[0] and arr[max], so why not take advantage of the underlying array for the backend of the library?

Further, one of the things that we can't stress enough is that all of these data structures can be implemented in other languages. But most languages don't have true pointers. At best they tend to have simulations that are still high level interpretations of the underlying pointeriness of a pointer. But by 4th-gen languages, they don't even both simulating it. The closest we have in visual, high level, or low-code environments is handles to other objects. But whether it's a pointer or a value or a reference is all abstracted away and handled 'under the hood' by the runtime environment.

But, this is not really a big deal, in fact, it actually doesn't matter. Let's consider what an array is. An array is a set of values that are consequent. That is, they are all right next to each other. Could we define memory as a set of values that are all consequent? So, if we take the entirety of the memory heap and subsection it, we'd get a set of memory addresses that, in most ways, acted with the same context as the main memory heap, just with a smaller set of addresses available. So, what's the difference between taking a subsection of memory and declaring an array? Aside from the dramatic flair of one over the other, they're basically the same.

Now consider what we're doing when we create a linked list or a pointer-based tree. We're creating a set of pointers that designate where the next node in a series of nodes is, based on where the memory manager allocated new values. What if, instead, we could define a fixed and deterministic way of determining what those memory addresses would be? We'd have the same advantage of the array links, but we wouldn't have to 'waste' space by having the actual link, we could calculate it each time. This is dangerous territory: we've already established that we'd almost always rather spend space in order to save time. And in this case, we're spending time in order to save space. But! The saving of space is just a side effect of the fact that we have a fixed and clear understanding of where the data is.

Because, in fact, we could then subsection the memory heap (by creating an array) and defining our own memory management routine (by creating the deterministic set of pointers to the next link in the chain) and we basically still have a tree or a linked list, it just looks different. But otherwise, it's exactly the same.

Now, there are a few big caveats there that don't really affect the conclusion:

- Memory doesn't really act like that. It's more complicated. But, for most programming, we just treat it like a series of addresses we can just go from one to the next on. Worrying about whether we can really treat an entirety of 64-billion addresses as contiguous is moot.
- Saying that the array based and pointer based structures are basically the same is a little jarring, and might be arguably too fragile a statement to make. That 'basically' is doing a lot of heavy lifting in that sentence. However, the point is that anything we can do with one we can do with the other. So, we can treat them as the same and only consider the ease of implementation, and ancillary benefits of the implementation.
- We don't really save space bey eliminating the pointers. We don't have to record ->next, but that's

effectively compensated for by the fact that we need to define an array. In fact, it's overcompensated for, because we're going to create the array to be larger than necessary to avoid the previously noted static array sizing problems.

- However, if you don't admit to the space we lose by allocating the array, we won't admit to it either, and we can all pretend we've found a magic algorithmic trick.
- Or, "If you don't count the cost, you can pretend it doesn't exist."
- Or, "If you declare the array overallocation irrelevant, it will help improve your performance metrics when it comes time to publish."

Again, we want to emphasize that there are practical and real reasons to use the original pointer and recursion-based solutions. Particularly in low-level and systems level programming. But in the vast majority of development environments, there are already libraries that exist, and those libraries will almost always prefer a managed array-based solution behind the scenes. But again, that giant over-allocated buffer is magic, not a real cost of using high level languages.

Array

```
struct node
{
     int payload;
};

struct node* arrtree[MAX]; // 1
bool occupied[MAX]; // 2

void insertABST (struct node* target, int i) // 3
{
     if (i >= MAX) return; // 4

     if (occupied[i] == false) // 5
     {
          arrtree[i] = target;
          occupied[i] = true;
          return;
     }

     if (lessthan(target, arrtree[i])) // 6
```

```
            insertABST(target, 2 * i + 1); // 7
      else
            insertABST(target, 2 * i + 2); // 7
}

int searchABST (struct node* target, int i)
{
      if (i >= MAX) return -1; // 8
      if (occupied[i] == false) return -1; // 9
      if (equals(target, arrtree[i]))
            return i;

      if (lessthan(target, arrtree[i]))
            return searchABST(target, 2 * i + 1);
      else // 10
            return searchABST(target, 2 * i + 2);
}

int minABST (int i)
{
      while (2 * i + 1 < MAX && occupied[2 * i + 1] == true) // 11
            i = 2 * i + 1;
      if (occupied[i] == true) // 12
            return i;
      else
            return -1;
}

int maxABST(int i)
{
      while (2 * i + 2 < MAX && occupied[2 * i + 2] == true) // 11
            i = 2 * i + 2;
      if (occupied[i] == true) // 12
            return i;
      else
            return -1;
}
```

```
void deleteABST(struct node* target, int i)
{
     if  (i >= MAX) return;
     if (occupied[i] == false) return;

     if (lessthan(target, arrtree[i]))
     {
          deleteABST(target, 2 * i + 1); // 13
     }
     else if (greaterthan(target, arrtree[i]))
     {
          deleteABST(target, 2 * i + 2);
     }
     else // 14
     {
          if (occupied[2 * i + 1] == false && occupied[2 * i + 2] == false) /
          {
               occupied[i] = false;
          }
          else if (occupied[2 * i + 1] == true && occupied[2 * i + 2] == fals
          {
               arrtree[i] = arrtree[2 * i + 1];
               deleteABST(arrtree[2 * i + 1], 2 * i + 1);
          }
          else if (occupied[2 * i + 1] == false && occupied[2 * i + 2] == tru
          {
               arrtree[i] = arrtree[2 * i + 2];
               deleteABST(arrtree[2 * i + 2], 2 * i + 2);
          }
          else // 17
          {
               int mini = 2 * i + 2;
               while (occupied[2 * mini + 1])
                    mini = 2 * mini + 1;
               arrtree[i] = arrtree[mini]; // 18
               deleteABST(arrtree[mini], mini);
```

```
            }
      }
}

void inorderABST(int i) // 19
{
      if (i >= MAX || occupied[i] == false) return;

      inorderABST(2 * i + 1);
      handle(arrtree[i]);
      inorderABST(2 * i + 2);
}
```

Let's handle an array of issues:

1. There are a thousand decisions that go into making example code. The data type could be just an int, we didn't need to wrap it in a node. But, in practice, nobody would bother doing this with primitive data types. The closest might be if they have their data in column order such that they have a dozen different arrays that hold the primitive data from each attribute and a central function that maintains it all. But that's obviously a much more niche implementation than we'd consider. For the most part, any tree in practice is going to be interested in data with some level of context, not just raw numbers. So, the goal is to show that there's lots of comparisons, but the reality is rarely as simple as a <.
2. This is an interesting approach, so we included it. One of the things that's common in most large data applications is that things rarely get deleted. They typically get marked as deleted. So, in this case, if we were just holding raw integers, there'd be no way to know if the number was a real number of not, so we'd have to have a secondary array that holds that information. With the pointers, we could set values to NULL, and that would handle it. But, and this is a contrived example, suppose we have a myriad of structs that are sorted by several different trees by different attributes and we want to be sure that no struct becomes a memory leak, so we won't be free()'ing any of the data. Instead that's handled by a different, unlisted, set of functions. Regardless, it is true that we can frequently just use a secondary array to hold metadata, and that way we can, for example, run a garbage collection app when we have free time, and not in real time.

    - Side note, because this is a bool, it only needs a single bit. But a bool in most programming languages takes up a byte. If space were of concern, we could store this as an array of int[MAX / 8 + 1] and each bit would be the value of (occupied[i / 8] << (i % 8)) % 2. That is, the value at the appropriate group of 8 bits, we shift it over by the number of values within the 8 group that we know the occupied bit is, and then we take the result and mod it by 2. That will give us the

appropriate bool. But, notice that's ~5 operations for one value. The tradeoff being every bool taking 8x the space. Which one is used is very much an application-specific decision.

3. Notice that these will all be recursive functions, so we need to tell it what i we're currently considering, but we also need to give it the target that we're looking for. This, again, can be in a myriad of forms. Typically, you'd have a lot of these functions wrapped in some wrapper functions so that everything was formatted appropriately for the functions.
4. Rather than our checks int he pointer based implementation, with an array-based implementation we are always going to be checking for out-of-bounds errors. MAX would likely be a globally defined variable for the max size of arrays. It's a whole other question about what to do when the size of the tree is going to be large than what would be allowed with the MAX array size, but a lot of that is a social question: can we increase the size of MAX reasonably? Can we introduce a second LARGE_MAX for expected large data sets? Can we reduce the size of the tree in some manner? The worst outcome is to reduce functionality for arbitrary constants. The second worst outcome would be to create a complicated data structure that involves stitching separate arrays together.
5. This line could also be if (!occupied[i]). That has the benefit of being concise and still clear. However, another school of thought notes that a single character is not the most clear way of denoting any given action and that an explicit declaration of == false would be more readable. The first school of thought thinks the second school of thought is dumb and codes bad, and maybe they should just learn to read the code right the first time. This is one of those arguments that if you have an opinion, it's likely a strong opinion.

    - Generally, when we're checking whether occupied[i] is true or false is a secondary check for bounds. Because we don't have a fully balanced tree, it's possible something got messed up. It shouldn't have, because we're doing everything right. But, for example, suppose the data got read from a saved file that was tampered with, or was retrieved from a remote program that had different expectations. Checking is better than not checking.

6. See? Didn't < in the tree based one look better? But this is probably more normal. How we compare the values is largely repetitive code that needs to be kept track of. Most higher level languages do it for us, but in C we'd pick one of several inequality functions tied to the specific struct.
7. What's particularly neat about this approach is that it's a hard 1-to-1 map for the pointers in the earlier version. It's an indication that even though the number of elements will grow much faster with depth than, say, a linked list, we still can hold the same number of elements, we're just organizing them different. Because 2 * i + 1 grows at twice the rate of i, we get twice the space to store the twice as many nodes per level. It's worth taking some time to trace out a couple of these to get a good feel for how the expansion occurs.

- Reiteration: You can just plug these algorithms and they'll work. But, getting an intuitive feel will make you more able to adapt it and even remember it later. The why in this case is more important. The what can be looked up with a quick search for "Tree left node in array" or something similar.
- Donald Knuth codified the standard form of this approach in TAOCP. Nice.

8. It's a good idea, a necessary one even, to explicitly note whether your array-based tree is expected to return the actual entities or the index to find the entity. Broadly, without any other input, it's better to return the index rather than the value. You can use the index to find the value. But finding the index with the value is more difficult. This is an asymmetric mapping, so better to return the upstream side of it. That rule doesn't always hold true. But in this case, since we don't know what the calling function wants with the answer, but we do know that we can go index -> value with a single lookup, it's a fair bet that it's worth doing that way. This means that -1 is a valid error for index (and why just returning the raw numbers leaves little room for error handling). But if you're returning the struct pointers, you'll need to return NULL on the error.
9. This is one part error-checking. If we're currently roaming around the unoccupied parts of the array, something may have gone wrong and we need to stop. But, this is actually also part of the algorithm: we keep looking until we go past the parts that are occupied, and then the -1 means 'not found' in addition to 'error'. In higher level programming with a runtime environment overseeing things, we might distinguish between this more. But at the lowest layer, it's sufficient for 'it didn't work, do something else', the calling function doesn't necessarily need to disambiguate between errors of structure and errors of data.
10. This else is unnecessary. But, we want to be explicit about the fact that we did one thing and now another. It's not a catchall situation, it's literally the opposite, when the if did not activate. Alternately, it's the disjoint flow path, not the default fallback path. It's not necessary to be explicit with this, but there are benefits to implying what's occurring through the explicit code.
11. That's a lot of wow. So, even though it's a little bit of data hiding, a better approach would have been to use something like #define LEFT(i) (2 * (i) + 1). The parentheses make it nice and packaged so there won't be unexpected order of operations errors. Remember that #define replaces the first value with the second value prior to compilation, so we haven't changed the code at all. We've just made it a lot easier because this line becomes something like if (LEFT(i) < MAX && occupied[LEFT(i)]). We need the LEFT in both places to be sure that we aren't about to go past the min/max values. And, unlike an linear array, we don't know by default what the largest or smallest values are (0 and max – 1), but this condition gets crazy when it's written out denotatively.
12. We went back and forth a bit about including this section here. It's unnecessary. But we wanted to throw a quick nod to secure programming. Suppose you're acting over a large data source that has multiple people, applications, or threads operating over. Not only could one be a malicious user who is trying to change things to make parts of the app fail, but just routine operations when lots of apps are interacting on the same data. So, sometimes, it's worth being overly defensive by checking that something's still valid

even though you just proved it was valid. You're really proving that it hasn't changed. This is especially true when the first pass takes a lot of time per cycle, we would want to be able to have a quick check that nothing has changed. But, none of this is really part of the pure algorithm side of things. Just a quick side note that the pure algorithm and data structure study isn't normally enough: you'll want to include those practical aspects and know when and where are the most effective to put them in.

13. This is just a standard redirection. We're cleaning out the easy bits, which are 'is this not the node we're looking for? Cool, pass the buck!'
14. This is bad programming. It's great for algorithms! Algorithms will break down into deeper and deeper cases as needed. But, when it comes to actual, readable code, most people would avoid more than a few nested code blocks like this. This, in particular, is confusing, as the if{} elseif {} else block is entirely contained inside of this else{}. The why of that is because everything inside this outermost else{} is part of the target == arrtree[i] clause, so it needs to be set apart to demonstrate that. But, there's no difference between that and if {} elseif {} elseif {} elseif {} else{}, the case breakdown is exactly the same. The difficulty in readability is bad in both cases.

    - There is a movement called 'Never Nesters' that insist that you should never have one control structure inside of another control structure. Like any other coding philosophy, it's probably problematic when taken to extremes. But, there is a lot to be said for breaking nested conditionals or loops into smaller sections.

15. The case when we are deleting a leaf. We'll just mark the leaf as deleted!
16. The cases where there's only one branch. We'll move that branch up one an then delete it in place. This will ripple everything below up a layer.
17. The case where there are two branches. We just need to find the smallest value on the right side and move it up to replace the otherwise deleted node. Then we delete the old version of the smallest value (or rather, mark it as deleted).
18. We could probably use minABST() here to find the smallest node on the right side. That's a better implemented solution, but it was more complicated when showing the example than just explicitly doing it.
19. You can probably extrapolate pre and post order on your own.

So, notice that the code is a little cleaner in the array version. This is part of the aspect of information hiding. Because we know the shape and the size of the tree, since it's all available at a first-order retrieval, it's easier to visualize how everything is working. With the pointers, it's easier to see how the individual units become a tight set of proofs that create the emergent behavior of the grander structure. Luckily, we can read the proofs and program with the more practical version.

Non-recursive Version

We won't do it a lot, but just as the occasional reminder that all recursive functions can be written iteratively (and it's frequently a good idea to do so):

```
void insertABST (struct node* target) // 1
{
int i = 0;
while (i < MAX) // 2
{
if (occupied[i] == false) // 3
{
arrtree[i] = target;
occupied[i] = true;
return;
}
if (lessthan(target, arrtree[i])) // 4
i = 2 * i + 1;
else // 4
i = 2 * i + 2;
}
// Error // 5
}
```

Iterative Notes:

1. Notice how there are fewer arguments. That doesn't necessarily imply a simpler algorithm. But, you'll notice that the algorithm is simpler. Why is that? Because it's relying on the structure and correctness of the previous two versions and can just do what was needed to be done. That's possible, but it might also need some extra security checks. Where would you see them? Regardless, this is a standard pattern that we see: the algorithm or data structure is created and proven and articulated in academic theory and then the actual implementation tends to be much more straightforward.
2. The only thing we're looking for is did we go beyond the bounds of MAX. This is actually an interesting limit, because it happens regardless of how full the array is. If we have a MAX of 1024, we'll hit that when the level of any given insert reaches the 11th level. That means we could start faulting as early as the 11th value that's inserted into the array. This is why it's frequently considered important to keep the tree balanced, because then we also know there aren't any random vacuoles throughout the rest of the tree.
3. When we hit the first unoccupied node, in contrast to where we were with the recursive version, we want to just make that the place where we insert our target node.
4. There are only three real cases in this variation, and these two are that the inserted node is smaller or

greater than the current iteration's index's node.

5. The most likely error is that the array is imbalanced and we tried to insert beyond MAX. But, that's actually hard to check whether it's happening, so we just want to post an error. There isn't a lot we can do with C, because there's noone to report to when there's an error. The error just happens. So, instead we'll just fprintf("Array overload in insertABST()") and then maybe a little diagnostic data. But, realistically, this could be an expected and known behavior that doesn't even need an error, we've just reached the limit for the hardcoded maximums, and that's it. There are ways of addressing it, but the question is whether it's worth doing it, and that's a situational question.

What's more fun is that you should be able to replace this in the the recursive version. Aside from the argument difference (which could be fixed with a quick wrapper function that calls one or the other randomly), they should return the exact same results.

Practical Applications

Let's talk libraries. All libraries aren't made the same. In fact, most libraries are a security vulnerability. But, at the same time, suppose you go into your job as a programmer and tell your boss 'Bossman, I know that there's a thousand different solutions that do the exact thing that I want, but I really think we should spend six months writing some code that will result in an early prototype that's equivalent to a library that's already been released and is a standard tool for the vast majority of developers and that new coders we hire will have to spend a few months learning before they're familiar with it'. It just doesn't matter, the business case will absolutely override any security justification you were trying to give. So let's go over some of the common libraries.

glibc

You'd be surprised what's in the standard, default C library. It's the library that lets you use NULL for pointers, use malloc() and free() without having to directly interface with the memory hardware itself. It's got a ton of basic, routine functions. Tree traversal is now largely considered routine, so there are a series of functions that implement and act over a self-balancing binary search tree. You can search for 'man tsearch' for more information. 'man' in this case is short for manual and is the command in *nix systems that displays information about a function.

std namespace

In C++, if you're using the std namespace, you can also use the std::map and std::set classes. These provide more advanced functionality than the standard library in C. Specifically, you can define your own data types in

them for both the search index and the stored value. Reading through the cppreference.org site about std::map is a small blueprint for the rest of this text. We won't be able to go as far in one book as what a team of experts have developed across their career. But we'll sure try!

TreeMap<K, V> and TreeSet<E>

GeeksforGeeks' explanation of the java objects is fairly easy to understand. It starts to ramp up quickly, but most of the terminology we've covered in this text so far, their treatment is just much denser. It's a good halfway point. Java likes their TreeMap implementation so much that they use it as the basis for most of their 'ordered' (there's a difference between ordered and sorted, but that distinction doesn't matter in this case) data structures. It's a little more complicated than what we covered here for three reasons: 1) it needs to be able to handle lots of different data types, 2) it needs to handle a lot more error situations, and 3) it's much more comprehensive and feature-filled.

BTreeMap<K, V> and BTreeSet<V>

Rust is a contender to replace C. It might manage to do so, although there's still a lot of inertia behind C. Regardless, because it's a newer, modern language, there's a lot more history in the field of algorithms that it can draw from for its core concepts. It follows a lot of Java's approaches. Similarly, it's also borrowing heavily from where we're going. What we find is that BSTs just aren't robust enough for real world use. They might be useful for simple implementations in a small context, but Rust, for example, uses a later variation on the Binary Search Tree called the B-Tree.

Python

This is the highest level language on this list. It turns out that it doesn't have a native tree structure. Why is that? Because the goal of python is not for you to fiddle with the internal mechanics of things. Instead, you just trust python that when you say 'store a bunch of data, please. Oh! And sort it!' that Python will do it in an efficient and correct manner. High level languages are abstracted away from data structures and the level of algorithm we're considering here. You can still create a similar structure, but it's a bit contrived. There do exist a few libraries like bintrees that are mostly helpful for teaching the concepts of binary trees, but don't perform any better than the built-in python data types.

The point is, there are lots of options, but how do you evaluate them? You need to understand the simple data structures first before you can decide whether the tools that are available are useful for you, or if you'll need to adjust them or if you'll need to download a third-party library to do things in a more customized fashion.

AI Examples

When we say "AI", we mean lots of things. We've mentioned this before. When you put AI in your corporate speak brochure or on your resume, you're probably heavily implying some Isaac Asimov robotics laws or some sort of self-actualized GPT-LLM. But there are lots of other applications of AI. Next chapter we'll cover Natural Language Processing (NLP), which was the leading contender in realistic chat engines until the attention model came in vogue and statistical engines could be brute forced into sensibility with large amounts of computation. But another often overlooked model is the Expert System.

Here's a couple of examples you've seen before. You're walking through the toy aisle at the store, and you see a little gizmo that will guess the pokemon you're thinking of within 20 guesses! Or you see one of those videos online that says 'can we guess your state from these basic questions?' Or, you used to play the board game Guess Who until you realized that the game was set up to be winnable with an optimized set of questions. Or you use webMD or a car trouble diagnostic tool or a bank's automated voice line.

There are two ways we can simplify these down.

Solution-First

How many pokemon are there? 1,025? Geez, that's a lot of pokemon! Suppose you pick one (Psyduck, please be psyduck) and decide to leave it out so you have a nice round 1024. You can disambiguate them each in 10 questions and 2047 nodes(1023 interior nodes in levels 0-9 and 1024 leave nodes in level 10). If you gave yourself 20 questions, you wouldn't have a balanced tree anymore, but you'd have more room in various wplaces to disambiguate between similar pokemon.

So, how does this data get stored? As a series of question and answer objects with left/right links for no/yes answers.

Now, what's the difference between differentiating between 1024 pokemon (seriously, Psyduck is never the one they pick) and the various potential ailments that the human body has? An absolute ton! But, here's the reality: Zebras. When you hear hoofbeats, you think horses not zebras. Just because someone might be suffering from cadmium poisoning from acute and long term exposure due to shellfish consumption, it's probably much more likely that the shellfish they ate was bad. Most people who go to the doctor get medication for sinus infections, the flu, or general aches and pains. So, without minimizing the amazingness of modern expert systems for health diagnoses, it doesn't take a lot of end states to create an effective decision tree for this. And that decision tree can be reordered (sorted) to emphasize more common diagnoses over less common ones.

What's the most effective use of an expert system like this? By experts! The idea of an expert system is that it simulates the reasoning of an expert with the presumption that experts just know more things than the non-expert and that's why the expert can make better distinctions between cases. But the reality is that experts

are humans just like anyone else. So, combining their existing expertise with the routine and consistent set of questions that the expert system provides leads to a massive increase in the effectiveness and accuracy of their task.

Makes one wonder what sorts of expert systems might exist for programming...

Problem-first

The flip side of this is that if we list out all of the problems that might occur and then start categorizing them into what makes them different, we have the solution-first mode. But if we focus on the problem, we can say 'out of all of the wrong things that might happen, what categories are there?' So, for a vehicle, we might say: body damage, mechanical, cosmetic. For cosmetic, we might break it down into smells, looks, and cupholders. For the mechanical section, we might break it down into won't start, starts but won't do anything, does thing but badly, etc. The point is that we can develop an expert system not by finding a solution but by more precisely delineating how to identify the problem.

Top-Down or Bottom-Up

These are actually representative of large problem-solving methods. We can tie it back into AI, since AI is always going to solve problems, and it might go with either of these approaches. But, that connection would be a little specious. This is just a target of opportunity to point out that for large problems, starting with the solution domain and combining subsets until you have a tree that leads to any individual solution is one way (bottom-up). the other way is to start with the most generic problem and start differentiating different variations of the problem until all of the subsets of problems lead to single solutions or an acceptably small number of solutions (top-down).

Cost-effective

Regardless, it doesn't take a lot of overhead, the structure is similar to the BST we've covered in this chapter, and organizing it is largely just a question of figuring out how to represent the decisions that are being made. But, it takes a lot less coding, seems just as impressive to most non-programmers, and costs a lot less in time and resources to interview a couple of experts and codify what they do into a simple interface than to try to build up an entire general purpose chat engine.

Consider the average task. When you go to pay a cashier at the convenience store for your slushee, you have to say words and they have to say words. Think about how much time and effort was spent for them to learn to talk so that they can tell you how much your slushee costs. But there are ancillary reasons why they learn to

speak moreso than just providing for your upcoming sugar rush. But does the computer really need to talk to accomplish its task?

One of the things to keep in mind is that you can claim 'AI' applications for things that aren't GPT or LLM. And expert systems design paradigms are improving as fast or faster than GPTs, so it's worth exploring what models will apply to any given problems you might want to solve.

Media Attributions

- BST
- balanced tree
- traversal illustration

CHAPTER 8 - YOU DONE MESSED UP, TREE-TREE-RON!

In our last episode, our intrepid adventurers had just discovered the joys of Binary Search Trees, and how they seemed to improve with every iteration. But, beware, for the same ease with which they inserted and deleted could cause them great woe. Should they dig too deep, they might unleash the unbal[anced]rog.

Here's a thought exercise. Imagine a Binary Search Tree that starts off empty. Insert 50 into it. Then insert 40. Then insert 50. Then insert 40. Repeat that, say, 15 times. Like, you have overrun your buffer. But your MAX for your array was 1024, so how are you overrunning it with only 30 numbers input? The problem falls into what we call dense versus sparse storage. Some data types like a hash table (a few chapters away) are intended to be sparse so that they operate well. Others like Linked Lists are dense, in that every node has to hold data.

One metric we could use is equivalent to the electrical engineering term of signal-to-noise ratio. Some of the space we allocate directly corresponds to the data we're covering. But, for example, when we have an array of pointers to different structs, all of that array is not directly correspondent to the data itself. It's just overhead. It's space we allocate because it makes interacting with the data easier. Calculating the direct, relevant data over the entirety of the data allocated is a good metric to keep in mind. It's not a fixed rule, but there is a sort of connotative correlation that the data structures with less overhead can be operated over faster. This is more ,so at the very low levels, where the time it takes to skip over the overhead data to get to the actual data becomes meaningful.

But, that's not the metric that we're looking at. What we really want to see is that there is a spot available for us to put data in, and that data is intended to go into, but there's no data there. It's an empty location. This is sparse data, and it's frequently a bad thing. Interestingly, it's not always a bad thing as a number of different algorithms will rely on its presence. It's a bit of a balancing act. The best instance of this we've seen so far is the MAX oversize of an array. The bigger we make the oversize, the more we'll be able to dynamically adjust the used portion within the static space. But, the flip side is that it literally costs money to put memory into a computer, to run electricity through the RAM sticks, and opportunity cost in other things we could be using with that memory. So, nobody minds a 30% overhead to help make your data structure's array more dynamic (in fact, they might congratulate your frugality!), and probably nobody would mind a 3-5x overhead. But it's when you start getting to 100x or higher that the egregiousness is really apparent. (Best to have some hard numbers you can rely on to show how big you expect the array might grow.)

So, let's take some time to find ways to improve the density of the trees that we create. Ideally, we'd like to see 100% density in the used part of the array that represents the tree, and we have a minimum amount of the overall array that is occupied (in the terminology of the previous examples).

AVL Trees

Let's start with a quick recap, and remove some of the demonstration code from the last version.

Minimal Binary Search Tree

```
struct node // 1
{
     int payload;
     struct node* left;
     struct node* right;
     // 2
}

struct node* insert (struct node* root, struct node* target) // 1
{
     if (root == NULL) return target;

     if (lessthan(target, root))
          root->left = insert (root->left, target);
     else if (lessthan(root, target))
          root->right = insert (root->right, target);
     else
          return root; // 3
}

struct node* delete(struct node* root, struct node* target) // 1
{
     if (root == NULL) return NULL;

     if (lessthan(target, root))
          root->left = delete(root->left, target);
     else if (lessthan(root, target)
          root->right = delete(root->right, target);
     else
     {
```

```
            if (root -> left == NULL)
                root = root->right; // 4
            else if (root->right == NULL)
                root = root->left; // 4

            struct node* temp = min(root->right); // 5
            root->payload = temp->payload;
            root->right = delete(root->right, temp);
        }
        return root; // 6
}
```

Let's look through a few more notes:

1. This is not a full BST implementation. The BST implementation is in the previous chapter. however, we're massively reducing the code so that we can focus on what's new with the AVL. Notice that as we've gone along in this text, we've been a little sloppier with the code. It was important in the earlier chapters to have simple concepts and robust code. But as we're moving to more complex concepts, we're trusting you to engage in basic security coding. A good exercise is for you to copy this code and explicitly grade it, showing what places the improvements need to go. But, for the sake of brevity, we're going to move on.
2. We got rid of the parent structure, because we won't need it for AVL. The reason we're using this pointer node structure is because it's easier to highlight what's happening during a rotate operation. We will need to add a 'height' attribute. You should be able to follow along and translate all of this over to the array structure on your own. Remember, what's happening is staying the same, the only thing that changes is how we access the data.
3. Notice how simple the insert node is when we boil it down? It's really just four lines: If we didn't find it, return an error. If it's on the left side, search the left side. If it's on the right side search the right side. Otherwise, it's right here. Notice that this will only return the first instance of the value. Also, notice that because we're using the 'lessthan()' function here, and we could use any function we want, we can actually adjust the BST to insert and search in other mechanics. That's why we only used the one function here, so that, for example, if we were doing statistics, we might want to return true if it's within a statistically insignificant range of the other value rather than exact equality. But, for now, we're just using ints and exact equality is fine.
4. This should probably 'free' the data. There's actually a lot to unpack here, and we have before. You must guarantee that the values are free()'ed. But, it's possible this section isn't where we do that. When we overwrite a pointer, that is an excellent place to a) check if we need to free() anything in the code, b) double check whether we should be doing that right here, design-wise, and c) making sure to include a comment either way about why we are or are not free()'ing the pointer at this point. We've reminded

you enough, this line here should set off some alarm bells as you read it: we overwrote a pointer without freeing the overwritten pointer.

5. This is actually a convoluted swap. It's also the case in the previous code. Can you trace how it's the same as the swap() code we introduced in the first chapter?
6. Delete is always more complicated than insert. It's the biggest reason we typically 'mark as deleted' rather than actually deleting, because the actual delete code is frequently complicated. But, in this case, remember that the general pattern of a tree is almost always (not always, just almost) some permutation of: handle the ->left, handle the ->right, handle the root. So, in this case, we're checking if the target node is on the ->left, on the ->right, then assuming it's the root. Handling the ->left and ->right is easy. Handling the root is three cases: check if the ->right is NULL, check if the ->left is NULL, and then deal with the root portion by rippling the deletion down the ->right side and replacing the root with the min value on the right side (again, why ->right? It's just a toss-up, either would be fine.)

The Problem

Again, we've mentioned that there's a problem with this code. The biggest issue is when the data is sparse. An alternate way to phrase that in the pointer version is that as the tree becomes more unbalanced, it approaches the linked list but with all the extra overhead of the unused branches. For the array based structure, we've already noted that for very unbalanced trees, we run out of depth because the array can only handle 2^{depth} nodes.

Another way to phrase that is that as the array-based implementation becomes unbalanced, the amount of wasted overhead approaches some variation of $2^{(depth - 1)} - 1$. That's pretty substantial. That means that, by the 10^{th} value we insert, we're wasting 511 array entries. Is that really a problem? After all, it's only 511 pointers, which in a 64-bit system is only 32Kb. But, by the 11^{th} value we insert in the several imbalanced system, we're wasting 64Kb per inserted node. By the 12^{th}, 128Kb per inserted node, etc. By the 20^{th}, we're wasting 32Mb per node. The point is not whether the system can handle it.

This the important part: we don't care about the facts of the matter, when we're approaching these problems from the theory side. Notice that this is a book and not a computer. We could say 'Ok, pretend we took the entire mass of Jupiter and converted it to sophont matter that was basically a pile of transistors and RAM, so none of our inefficiencies matter!'. So, the 'what-is' is not as important as 'what will inevitably be'. In this case, we can see, that doubling with ever single inserted node is going to be crazy large. For example, even a simple database that had 121 simple values in it would overrun a memory bank the size of Jupiter (assuming each gram of mass was ~1Mb).

The point being that we don't care about the waste right now, we care about the fact that it will, inevitably, be unsustainable. Even if it turned out that we were way out of line, and our exponentiation function was only $1.01^{(dept - 1)}$, exponents are crazy enough that this overflows at 7027 inserts.

Compare that with a much more computationally reasonable n^2 function. We don't overrun that memory

bank the size of Jupiter until we get to inputs in the 10^{18} range. And for logarithmic functions or linear functions... the point is that our analogy is a little silly, if for no other reason than a Mb of RAM is closer to a milligram in weight, and by the time we got to Jupiter and were converting it with a Von Neumann Machine to sophont matter, it would be at sci-fi levels of extreme. But, that's neither here nor there, hopefully:

It's clear that the trend is what's important. And because we don't know what the implementation will be, when we're discussing the growth and dangers of algorithms, we have to consider it from how fast it is growing, not from what it is with any given static input. Luckily, this Big-O notation (O(n), O(log(n)), $O(2^n)$) will address that. We'll get into why it works in a couple of chapters.

All of this is important to understand why algorithms act the way that they do. It helps to determine why is it that the array we thought would hold 1024 values only holds 10 values. In that case, you could instantly note the relationship (even close to this relationship) of n to log(n) and guess where the error might be.

Hesitantly, we can also say that none of this matters for any given implementation. What matters for the implementation is readability, reusability, and whether it works at all. If it doesn't work, no amount of theory is going to make it work. And if it does work, it doesn't matter how much waste there is as long as it's in bounds.

We say that hesitantly, because in practical reality, the difference between a human programmer and automated code is exactly that: the automated code can't tell when it's being ridiculously inefficient for given tasks, but a human can. Plus, while we haven't talked about elegance, we are going throw out there: elegance is an important stylistic point.

So! Let's narrow down the effects of this exponential growth problem here:

Effects of the Problem

In practice, there are two big issues with not full filling out the array. The first, we've talked ad nauseum now: the sparsity of the data structure. In fact, we could, if we wanted to, generally create a system for measuring the amount of space that something uses. So, an array always uses some variation of O(n) space. By definition, it will always use the amount of space allocated to it. In contrast, when we use a complete and balanced tree, we have a data structure that is O(n) in space usage. Which, if our math is correct, is the exact same amount of space as the array! It seems like we could confirm this was the case by having a way to map from the tree structure to an array implementation of it. Hint: that's sarcasm, we provided that mapping last chapter. This is an example of how we prove that two things are equivalent.)

Space

However, that's a special case. A complete tree is a special case of a balanced tree. And a balanced tree is a special case of a tree. So, if we take those restrictions away, that's when we start running into issues like we've described. Specifically, if we look at the best case scenario for a balanced tree, it's O(n) use of space, but the

worst case scenario for a balanced tree is $O(2^n)$. We've just explored both of those. And here's a practical reality:

We have no way of knowing, without any prior information, where on the spectrum of O(n) <-> $O(2^n)$ any given tree will be without the constraints of balance. That inability to provide bracketing (or constrain the solution space) is problematic, because it means we don't know how much computational resource to allocate to the tree. If we allocate enough for the worst case scenario (technically impossible at 2^n for all useful values of n, but hypothetically) then we're leaving the processor to sit around and twiddle its thumbs when it's doing something much more balanced. On the flip side, if we allocate resources for smaller, we'll have situations where we run out of resources and the program might crash.

Side note: So, what we end up realistically wanting is to be able to frame, break up, or constrain problems such that the range of possible problem sizes is relatively fixed and known. The narrower the bands of problem we can assign the easier it is to put this together. In short, most industry applications of programming want to be able to reduce the variation in processing time for any given problem. It is probably not perfectly accurate, but it might be arguable to suggest that reduction in variation is more important than making the fastest cases faster. (Alternately, increasing the speed of the slowest cases is more important than increasing the speed of the fastest or even average case. Your mileage in that aphorism might vary, though.)

Side note: the average case for the space used by a tree is actually kind of interesting. How tall is the tree? Suppose we have a 50-50 chance at any given traversal for the insert() function to go left or right. (This is an informal proof, but it follows the actual literature in the field) That means that as the tree increases in height, the odds that a new insert() increases the height also decreases because it has to reach the deepest node in the tree. (This is countered by the fact that as the tree fills in with non-height increasing insert()'s, the chance increases, but for simplicity sake, we're skipping that bit — which is fine, because it's also a smaller order term, and our average case analysis outcome won't be affected.) So, there's 1 option out of a myriad of binary choices, right? If we have a series of binary choices, that's $1 / (2^h)$. This is just like the odds of a series of coin flips all being the same is $1 / (2^f)$ where f is the number of flips. So what is h? It's the height of the tree! What's the height of the tree, we'll shortcut to the end of that tangent to say that, on average, it'll be log(n). After all, that's the point of the binary tree. So, the odds that we increase the height of the tree on a random new addition is $1 / (2^{\log(n)})$. The exponent and the log cancel out, so it's really 1 / n. Thus the odds that any new entrant will increase the height of the array is 1 / n for the generic case.

Time

But what about solving time? We have the same bounding problems. To make this easier, let's presume infinite space. Not because we'll have infinite space, but if we crash the program because of a space constraint, that's not the time analysis' fault. So, for any insert, we can go as low as O(log(n)) and as high as O(n). This doesn't seem bad, but remember that if logarithms are the opposites of exponents, then we're running into the same issues: O(log(n)) when n is a million gives us a rough time estimate of 20 to complete the insertion. O(n)

when n is a million gives us a rough time estimate of 1,000,000. So, it's less dramatic because the O(n) isn't getting carried away, but the ratio between outcomes is still substantial. And it means that we're running into the same issue in terms of how much computations resource to assign to the task.

For example, suppose we have some sort of movie database that's international, and there are around 20,000,000 media entries listed on it. Suppose that there are 100 visitors per second looking up some value. That means that we're looking up 100 entries out of 20,000,000 per second, and the informational page can't even start to be displayed until the target is found. If we don't have any bounding for our giant Binary Search Tree, it means that it might take 25 comparisons to find through to 20 million comparisons to find. So the question is, does this need a dedicated processor just for the search feature? In the former case, it's kind of trivial, as it's a total of 2500 operations per second out of (probably) billions. But in the second case, it's billions of comparisons out of a pool of billions, and we'd need the dedicated processor. Nobody minds if it needs the specific dedicated processor to make everything faster. But it would be frustrated to find out that it's because the binary tree was just badly designed, and the processor wasn't needed most of the time.

So, again, we see that being able to more tightly constrain the problem size is a better outcome.

Solution

Luckily, this means that what we want is the same for both situations, and it kind of makes intuitive sense as well. We just want to not waste all of those empty array entries, or the equivalent in pointers that point to NULL that comes from having an extremely unbalanced tree.

So, the reason why we want this balancing operation to occur is to make sure that if we are choosing between recursively searching the left or right halves of the tree, we're really cutting the problem size down into as close to half as possible. This reduces the amount of time the problem takes, and it also reduces the amount of space it takes because it's reducing the overall height of the tree.

It's kind of important to note that balanced and complete are separate metrics. A tree is balanced if the height of the subtree is approximately equivalent on the left and the right ('approximately' in this case meaning off-by-one, because we have to allow for odd numbers and partial layers). A tree is complete if every layer but the last is full of 2^{layer} nodes. These two metrics are different, but it turns out that it's practically more difficult to balance a node without either making or assuming that it's complete. So, we'll end up doing both of those things.

Again, remember that what we're trying to do is narrow the brackets of the completion time and space used of the algorithm. Frequently, that bracketing will also reduce the maximum completion time (in fact, kind of always). But, this is a good example, because it also increases the minimum completion time. The tradeoff is that it takes extra computation to constantly be rearranging the tree to be sure that it's constantly balanced, sorted, and complete. However, the expectation (and the reality), is that the overhead is vastly outweighed by the reduction in worst-case-scenario.

Tree Rotations

Let's jump into what a 'rotation' is. Imagine that the root, ->left, and ->right nodes are all in a triangle configuration (Or a Triforce! Side fact: did you know that the triforce is a 1-layer Serpinski Triangle, which is, itself a recursive representation of a geometric figure with an area of exactly '0', constrained by the finite nature of its recursive depth.). We're not actually going to do a rotation that makes any sense in this context. But you need to be set up and then dashed. The word 'rotate' here is a bit of a misnomer. It will not look like a rotation in any local sense.

But, in a global sense, we're sort of pushing the whole tree over. Imagine a large tree, hundreds of nodes, and one single node is slightly out of place, because we just inserted it sort of willy nilly. So, we're rotating all of the values around in a giant cycle (not a circle), and the size of that cycle, how many nodes it entails, how much it expands will depend on essentially how unluckily placed that out of place node was. But, because the tree was ultimately pretty well sorted, we likely won't see lots of changes. However, the idea is that if we have too many nodes on one side of a root (whether the whole tree or a subtree), then it's less about figuring out where to move the newly inserted node, and more about how to cycle that root node out so that the most middlest (average in the median sense) node is in the root position.

What's the point of this? Essentially, we're doing the equivalent of a tree bubble sort. Only instead of a single swap, we're rotating three values so that the middlest value is now on the top, regardless of how it was.

We'll try a bunch of situations, but one thing to remember is that the presumption for any self-balancing tree (notice what they did there? A BST is not a Self Balancing Tree?) is that there are certain constraints on the insert and delete functions that will only allow situations that can be resolved by the basic AVL algorithm.

Atomic Functions

Note that we're not really worried about the traversal functions or the search functions. These are read-only functions, so as long as the tree is sorted, balanced, and complete, they can just assume that they're running optimally. It's the insert and the delete functions that need to be sure that they leave the tree in the same sorted, balanced, and complete state afterwards that they had before they started. Try writing out the PreConditions, PostConditions and other heads for the following functions.

Another important caveat: we always presume that a self-balancing tree starts out sorted, balanced, and complete before any of the functions run. Why? Because otherwise we have to sort, balance, and complete the tree. And that's not the algorithm. In this case, we're solely maintaining the existing specific descriptors. How can we be so sure? We start with an empty tree.

An empty tree is ordered (at least nothing is out of order), balanced (both sides are equally NULL, although that's technically undefined since the root is NULL), and complete (because the height is 0, we have to define it as complete, since 2^{-1} is undefined behavior for counting numbers). More calculatedly, a tree of size 1 is

always ordered (for the same reason we said in the Sorts chapter: you can't have a single value that's got the wrong equality to its neighbors, it has no neighbors), it's balanced since both subtrees are of size 0, and it's complete because the 1st layer is the only layer and has $2^{1-1}=1$ nodes in it. So, when the tree is empty, it's sorted, balanced, and complete and then if we insert a node it is of size 1 and is sorted, balanced, and complete by definition. At size 1, we start Sorted, balanced, and complete, and if we delete a node, we go to the empty state which is also sorted, balanced, and complete. Or we insert a node, and then we have size 2, which isn't guaranteed to be sorted, balanced, and complete. But, we are assured by the AVL tree algorithm that, given a sorted, balanced, and complete tree, the output tree will be sorted, balanced, and complete. So, the tree at size 2 will be algorithmically guaranteed correct.

The point is that all of the functions will, effectively, be the same as the BST outside of the struct definition, the insert function, and the delete function.

Definitions

We've given these piecemeal before, but as a quick recovery:

Let T be a binary tree. For any node n, define:

Nodes(T) = the set of all nodes in T

L(n) = the left subtree of node n

R(n) = the right subtree of node n

p(n) = the payload, key, or value stored at node n

height(∅) = 0, where ∅ represents an empty subtree

- Sorted (inorder)A binary tree is sorted if, for every node, all values in the left subtree are smaller than the node value, and all values in the right subtree are larger than the node value. This is the property that makes an inorder traversal return the values in sorted order.

  $\text{Sorted}(T) \Leftrightarrow \forall n \in \text{Nodes}(T),$

  $[\forall x \in \text{Nodes}(L(n)), p(x) < p(n)] \wedge [\forall y \in \text{Nodes}(R(n)), p(y) > p(n)]$

  Important: The condition must apply to the entire left and right subtrees. Comparing only left(n) and right(n) is not enough to guarantee that the whole tree is sorted.
- Balanced

A binary tree is height-balanced if, for every node, the heights of the left and right subtrees differ by at most one.

$\text{Balanced}(T) \Leftrightarrow \forall n \in \text{Nodes}(T), |\text{height}(L(n)) - \text{height}(R(n))| \leq 1$

AVL note: An AVL tree maintains this balance property after each insertion and deletion by using rotations.

- Complete Binary Tree

A binary tree is complete if every level is completely filled except possibly the last level, and all nodes in the last level are placed as far left as possible.

A clean formal definition uses level-order indexing:

index(root) = 1

index(left(n)) = 2 · index(n)

index(right(n)) = 2 · index(n) + 1

If m = |Nodes(T)|, then T is complete if the occupied indices contain no gaps:

Complete(T) ⇔ { index(n) | n ∈ Nodes(T) } = {1, 2, 3, ..., m}

Clarification: Completeness should be defined by level structure, not by referencing the balanced property. A complete tree is usually height-balanced, but completeness is a separate structural property.

Implementation

```
struct node*

{
     int payload;
     struct node* left;
     struct node* right;
     int height; // 1
};

int height (node* n) // 2
[
     if (n != NULL)
          return n->height;
    return -1;
}

int balance_factor(node* n) // 3
{
     return height(n->left) - height(n->right);
}

struct node* rotate_left (node* n) // 4
{
     node* r = n->right; // 5
```

```
     n->right = r->left; // 5
     r->left = n; // 5
     update_height(n);
     update_height(r);
     return r;
}
struct node* rotate_right(node* n) // 4
{
     node* l = n->left; // 5
     n->left = l->right; // 5
     l->right = n // 5
     update_height(n);
     update_height(r);
     return l;
}

struct node* balanceAVL (struct node* n)
{
     update_height(n);
     int b = balance_factor(n); // 6

     if (b > 1) // 7
     {
          if (balance_factor(n->left) < 0) // 8
                n->left = rotate_left(n->left);
          return rotate_right(n); // 8
     }
     else if (bg < -1) //7
     {
          if (balance_factor(n->right) > 0) // 8
               n->right = rotate(n->right);
          return rotate_left(n); // 8
     }
     return n; // 7
}

struct node* insertAVL (struct node* root, struct node* target)
```

```
{
     if (root == NULL) return target;

     if (lessthan(root, target))
          root->left = insertAVL (root->left, target);
     else if (lessthan(target, root))
          root->right = insertAVL(root->right, target)
     else
          return root; // 9

     return balanceAVL(root); // 10
}
```

Let's rotate through some of these notes:

1. The first change is pretty simple, we just want to track the height of the tree. We can do this a thousand ways. But when we've got the pointers to structs, it's almost always easiest to just insert the height as an attribute in the struct.
2. This is just a helper function. In general, we could do lots of these values with ternaries, but that's awkward. Pulling the code into a separate function like this is easier than extending the code every time we needed height to include this code.
3. Just kidding! height() is only ever called from here! So, balance_factor() is the helper function. We also don't define update_height(), because that's a fairly arithmetic function. It just looks at the two child subtrees and adds 1 to the max height between them. You should be able to write that fairly quickly now. Notice that we could write it without height as an attribute at all! How much extra computation time would that involve, and is the tradeoff always worth it or only sometimes?
4. Generally if you see a qualifier like rotate_left, you're going to be able to guess there's a rotate_right. Now, we're going to look at it here in the insertAVL() function, but essentially what we want to do is just add a little push to make the whole tree lean a little more left or a little more right (respectively). What does it mean when we say 'rotate'? It's not what you'll find in the literature, but it's a fairly accurate lay description: you're moving one node from somewhere on the too-heavy side of the root to the too-light side of the root when there's an imbalance. So, rotate_left takes a node from the right and moves it to the left, and vice versa. How do we decide which node to move? We'll see in a moment that it's based on the difference in heights.

    - What this function does: it swaps the root node with it's child to the left or right (opposite of the direction it's rotating). Why? We'll break it down a section further down.
    - Take rotate_left(). The root's ->right gets shifted down to its ->right->left. Then the former ->right

replaces the former ->right->left to point to the root. And then we return the former ->right, which tells everyone before that the former ->right is the new root. So, by pulling the ->right up to the root, we can kind of feel how it's nudging everything slightly left.

5. We're going to get into what's happening here in the next subsection. But you should draw a picture to see what's happening. Try a tree that's a balanced one between a-c, an imbalanced one with a-c on the left of d and nothing on the right. then try a tree that's balanced between a-g. It's easier to see what's happening in a balanced tree than to try to break down the different cases right now. Basically, you're trying to see how running rotate_left and rotate_right mess up your balanced trees, and that will give you an idea of what's going to happen later.
6. Balance factor is just a quick helper function. We can do this any number of ways, but it tends to be fastest overall to cache the value in a set variable at the beginning of the function. It's literally just the difference between the heights of the ->left and the ->right subtrees.
7. We have three cases to consider when we're balancing. If the difference in heights between the trees is -1, 0, or 1, we'll call it as balanced as it's possible to be and just do nothing (the last case in the code). If that difference is a 1, it means that, with one caveat, we need to rotate right because the left tree is too tall. If that difference is a -1, it means that with one caveat, we need to rotate left because the right tree is too tall.
8. There are 4 cases we need to consider, and we'll give them more detail in a later section.
9. This has been present in the previous code, but it hasn't really been an issue until now. This is bad for readability, but it's pretty common for this particular algorithm, so we've included it here. There are 3 cases. The first if statement activates, recursing through and then we fall out of the the if-elseif-else and drop to return balance(). The second if statement activates, recursing through then we fall out of the if-elseif-else to return balance(). The third case is when the values are even and we're basically absolving ourselves of responsibility and saying 'we can't insert, this value is already here' and we just return without needing to balance() so we don't fall out of the if-elseif-else in this case. This style of code is poorly written. Even though it's more code, it would be much more readable to include the return balance() in each of the first two if cases. Or, have a separate callout that 'if equals(root, target)' to specify that we're bailing in that case.
10. All of these extra functions are to add this one node here.

Let's ask one quick question. Given no other information, how often are we going to need to balance the entire tree? We have two cases: an even number of nodes in which case our root is one of two possible middle balance points or and odd number of nodes in which case our root is the only possible middle balance point. Since we're talking even/odd, we can say that we're always talking a 50/50 split. In the second case, we'll never need to balance. In the first case, when we add a node it'll either be on the slightly imbalanced side or the other side. So there's a 50/50 chance that the added value will be greater or smaller than what we needed given that

we will already know which side will cause an imbalance. This means that, overall, there's a 1 in 4 chance that we need to rebalance the tree.

But that's the thing, when we do need to rebalance the tree, we want to check every time to make sure that the small problem doesn't turn into a big problem. Again, it's not about the small imbalance, it's about bracketing the overall cost of searching and inserting values.

But, we need to consider more than just that one situation. Suppose we have a perfectly balanced tree with 50 nodes to the left and 50 nodes to the right. If we insert the node smaller than the root, it goes to the left. But that side has an even number! So even though the main root node had 101 nodes and wouldn't need to be balanced, the left side might need to be balanced! Generally, we have the same four cases, all with equal chances of occurring (in this example, it's a 50/50, but that's because we already limited the options for dramatic effect). Further, even if that balance doesn't happen, the ->left->left branch might also only have 24 nodes, which means it might need to be balanced. So, for ->left, there's a 1 in 4 chance of needing balancing, but that situation can only occur half the time (Because the other half, it's the right node) which means that it's 1/8 of the time. And 1/16 of the time for the nodes below that. and 1/32 of the time for the nodes below that. But, there are two branches of probability for each situation in which a sort hasn't been guaranteed yet. So, the probability is doubled, since we need to sum the odds over all possible scenarios. Overall, the sum of all of those chances of there being some need somewhere for a balancing to occur will approach 1as the number of layers increases. (It's essentially the opposite of the Serpinski triangle: building up to a certainty out of a number of unlikelihoods).

What's happening here?

There are so many ways to describe this, we're going to try a bunch. Pick the one that resonates with you, don't feel like they all need to make sense:

Medians

Imagine instead of a tree, you've listed out every single number in order on a number line. Suppose you have 7 numbers listed. The 4th number is the median, right? That means there are as many numbers below the median as there are numbers above the median. But now you add a new number (it doesn't matter, but we'll pick one for the example) below the median. Now you have 8 numbers, and the 5th one is the 'median'. In statistics, it's the mean between the 4th and 5th numbers. But since we are using discretized values, that option isn't available to us. We have to pick one of the actual numbers and say 'this is the median!' So, which is it? The 4th or the 5th? In short, either one is just as accurate as the other. Since we've already got the 5th it's not worth changing. But now, let's say we add another number below our 'median' at the 5^{th} (why below? Because if we added above, balance is maintained and we don't need to worry, the algorithm only kicks in when we get out of balance). So, we've added the number below and we're now calling the 6th number out of 9

the 'median'. That's definitely not right. So, and this is sometimes (if it's not, don't make it confusing) the confusing part: the median stays in the same place, it's like we're pointing at a fixed place. But then we rotate all the numbers by nudging them to the right a little bit. Exactly enough to scootch everything over by 1 position. So now the place we're pointing to is the 5th position.

Why do we do it like this? Because with the tree, the 'root' is always the middle. It's by definition the middle of the tree. We've declared it balanced and complete, right? So, if it's starting to slightly lean one way or the other, we have to 'rotate' all of the numbers the other direction. Like when you're on a balance beam, and you windmill your arms the opposite way you feel like you're about the fall. We're just compensating.

What's nice about this analogy is that it's super clear that we have two possible states: we are exactly correct as to which one of the values is the true 'median' or we're off by a half-position. Alternately, if we're in the correct position now, the next node will make us off by a half-position, but we're still 'balanced' in that we have a tolerance of being off by one to account for odd sizes. Whether we go up or down doesn't matter, we were correct, and after the insert we're going to be wrong but we don't need to rebalance anything. The other case is that we're off by a half a position. In this case, after the insert, one of two things will have happened: we will either have shifted by a half-position again and be correct now or we will have shifted by a half position to be off by a full position. In the former case we are obviously in balance again, and in the latter case we need a rebalance. Again, when we do that rebalance, we're pulling a new value to be where the median is and shoving everything else over, not specifically moving the median.

The place this analogy breaks down is that it's rippling recursively through each layer of the tree. So, there's medians, quartiles, octiles, etc, and each one is able to pull smaller sections around.

The Next One

At any given point, we assume that the entire tree and every subtree is balanced. This is only about balance, not completeness, because if we're just adding each node in sequence, completeness will automatically happen. So, if we have some evidence that tree is not balanced, what do we want to do? We want to shift it back into balance. So, we're going to move the 'middle' to a more balance-y point. We don't know what the next number is, in a tree structure. but what we do know is that the ->right branch is bigger than the root. So, if we move the ->right to the root, and the root to be the new ->left, we've maintained that relationship. Now the new ->left is still smaller than the new root. And we kept the new roots original ->right.

Contradiction

Suppose we have the balanced binary search tree shown in Fig. 8.1.

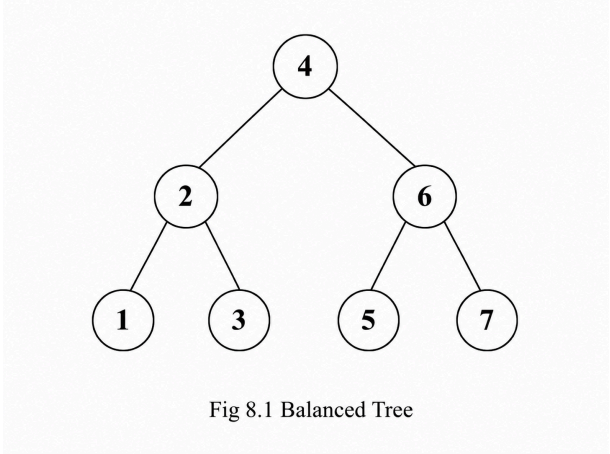


Fig 8.1 Balanced Tree

We would like to make this tree left-heavy while still preserving the binary search tree property. As it stands, the tree is balanced, sorted (remember the inorder traversal), and complete.

Notice that the height of `4->left` is 2, and the height of `4->right` is also 2. To make the tree left-heavy, we need to increase the height of the left subtree while decreasing the height of the right subtree.

There is one important constraint: The tree must remain a valid binary search tree.

Let's first focus only on the top three nodes: 2, 4, and 6. The tree is balanced because 4 is the root of this subtree. If we want the tree to lean to the left, which node should become the new root?

Clearly, 6.

Suppose we simply move 6 to the top, as shown in Fig. 8.2.

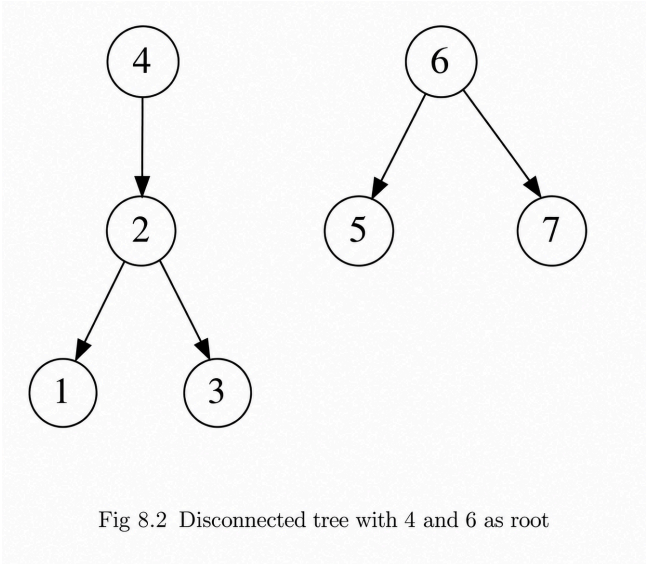


Fig 8.2 Disconnected tree with 4 and 6 as root

Now we have two completely disconnected trees. That clearly cannot be our final result, because we have lost the connections between the subtrees.

So where should node 4 go?

Since 4 < 6, the binary search tree property tells us that 4 must become the left child of 6. We can temporarily ignore node 5 while we determine where 4 belongs. The intermediate result is shown in Fig. 8.3.

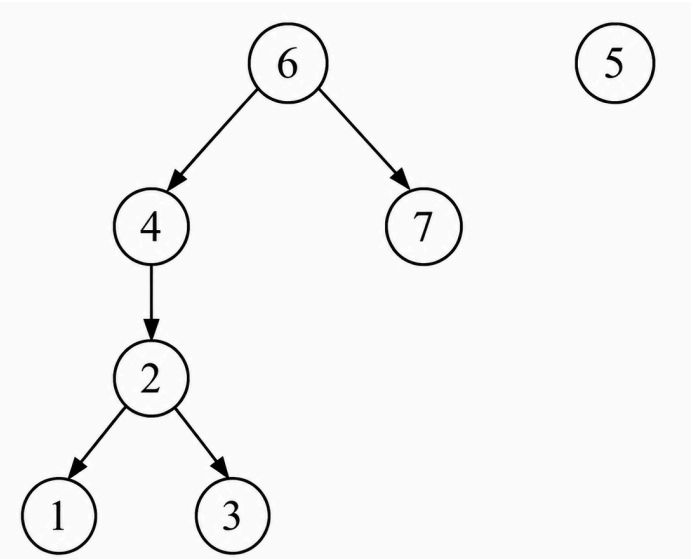


Fig. 8.3 After promoting 6 to the root and inserting 4 as its left child; node 5 is temporarily removed.

However, simply removing 5 creates another problem. Every node must remain in the tree, so we must find the correct location for 5.

Again, the binary search tree property gives us the answer.

5 < 6, so it belongs somewhere in the left subtree of 6.

5 > 4, so it must become the right child of 4.

After reinserting 5, we obtain the tree shown in Fig. 8.4.

Now compare the subtree heights. The left subtree of 6 has height 3, while the right subtree has height 1. The tree is now intentionally left-heavy while still satisfying the binary search tree ordering.

Suppose we now want to restore balance.

We cannot simply undo each step in reverse order. Instead, we repeat the same reasoning with the opposite choices.

Looking at the top three nodes—6, 4, and 7—which node should become the root if we want to rotate the tree back to the right?

The answer is 4.

Since 6 > 4, node 6 must become the right child of 4. Once again, we temporarily set aside node 5 while determining the new structure. Because 5 > 4 but 5 < 6, it must become the left child of 6. Drawing these intermediate steps mirrors the figures shown for the left rotation.

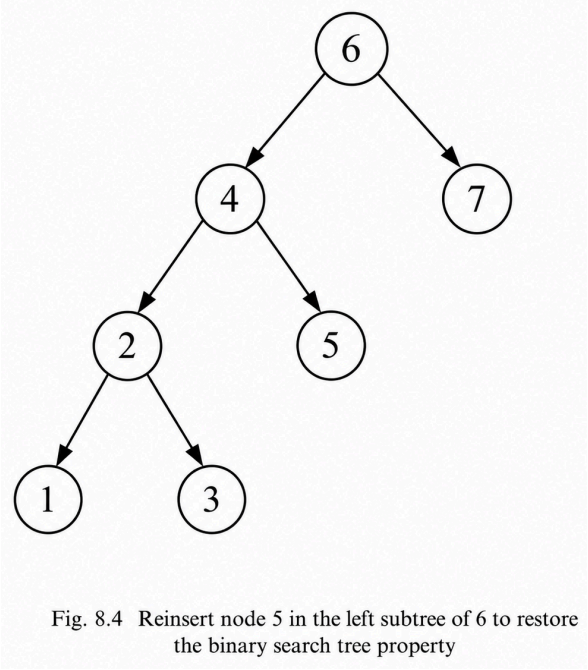


Fig. 8.4 Reinsert node 5 in the left subtree of 6 to restore the binary search tree property

This entire discussion is simply an intuitive explanation of what the rotate_left() and rotate_right() functions accomplish. The actual implementation is much shorter because it does not physically move nodes one at a time. Instead, it updates a small number of pointers so that the tree immediately reaches the desired final configuration.

The 4 Sort Cases

There are four cases. Basically, we want to know whether the new node that got inserted is all the way on the left, all the way on the right, or if it zig-zagged a little bit.

LL

Suppose there are 10 nodes in the tree. And the root node is at the 6th node. Now, suppose we insert a node that is the new smallest node. Every left subtree is going to be a little more imbalanced. We can fix that by just rotating everything to the right, it'll pull everything up and then we can recurse down to be sure that it helped. We know that this happened because the height of ->left is +2 compared to ->right. But, the height of ->left->left is equal or greater compared to ->left->right.

RR

The same difference but flipped. We have 10 nodes, the root node is at the 5th node. We insert a node that is

the new largest node. Every right subtree is going to be a little more imbalanced. We can fix it by just rotating the root left, and then letting it propogate down the tree on the right side. We know that this happened because the height of ->left is -2 compared to ->right. But the height of ->right->left is less than the height of ->right->right.

LR

So, the trick here is that we have to pretend that the left subtree is its own beastie. Essentially, in LL and RR we're assuming that any cleanup that needs to happen will propagate down, and the sort will be maintained because the appropriate rotation will keep everything there. But when it's mixed like this, we have to rearrange ->left a little to be sure that the correct node is going to be scootched into place. We need to do a pre-scootch, if you will. So, suppose we have 10 nodes, the root node is at the 6th node. We insert a node that is just barely less than the root. It'll go on the left side, but it's going to make the right side a little too heavy sometimes. So, we have to hop down to ->left and rotate it left (and left everything propagate down), and then we'll rotate the root node right with the newly arrange node there.

RL

Same ting, but flip right and left.

Accomplishment

This is a common pattern and one you want to emulate. 'Let's assume everything's right. After my function runs, everything will still be guaranteed to be right'. You can't control the data coming in, but at a certain point we have to presume that our program is following it's own internal constraints. So, especially with programming, don't presume that you can fix stuff later. Security needs to be baked in at the beginning, and algorithmic correctness needs to be insured from the first set of functions.

AVL Tree Delete

Recall our delete function from the beginning of the chapter. We just need to modify it a little bit. We're going to do the same thing. Like Liam Neeson, we have a particular set of skills. We will look for it, we will find it, and we will ki–delete it. But! Because this is an AVL tree, we are just going to make sure that in the unwind phase of the recursion, we just do a quick balance. No biggie. However, you should consider: what's the difference between a downward-propagating balance() and an upward-unwinding balance() chain?

```
struct node* deleteAVL (struct node* root, struct node* target)
```

```
{
     if (root == NULL) return NULL;

     if (lessthan(target, root))
          root->left = deleteAVL(root->left, target);
     else if (lessthan(root, target)
          root->right = deleteAVL(root->right, target);
     else
     {
          if (root -> left == NULL)
               root = root->right;
          else if (root->right == NULL)
               root = root->left;
          struct node* temp = min(root->right);
          root->payload = temp->payload;
          root->right = delete(root->right, temp);
     }

     return balance(root); // 1
}
```

One measly note?

1. This is it. The entirety of AVL's contribution to the delete function is using one of the helper functions that were already there from Insert. Remember, however, that this will engage several times as it goes back up.

This one was nice. Isn't it nice when you already automagically made things happen elsewhere and you can just rely on your previous correctness to be correct in a different capacity?

Min/Max Heaps – The ReHeapening

Some of your brain juices should be flowing How does everything we just did apply to heaps from a couple of chapter ago? This is the sort of connections we need to be able to make. This is what adaptation and being able to apply new data structures to existing problems looks like. Let's consider the logic trail:

1. We learned about Linked Lists, which at that stage ended up being used as overly complicated arrays. We

need them because they help us explain why (not how, in practice) other data structures work.
2. Linked Lists are slow compared to arrays, but how might that help us? What if we wanted to have O(1) lookup by constraining what we could look up? So, we changed the problem to fit the solution we cam up with.
3. Since we had Stacks and Queues, we looked at the next evolution: Priority Queues. But that turned out to be a whole different structure under the hood. However, the end state was the same as the queue: we always know where things lie.
4. In order to really get where Priority Queues were going, we had to fully dive into the Trees structure, specifically Binary Search Trees.
5. We had a problem with the Binary Search Trees: they get too big in the worst case scenario and everything falls apart!
6. So we moved to AVL trees, which allow us to be in the best case scenario and solve all of our problems!
7. If AVL trees help us with BST problems, is it possible they might help us with Priority Queues?

* – When we say 'is' or 'is not' in this case, the engineer inside of all of us wants to pipe up and say 'Ack-schoo-all-lee... it's just not guaranteed to be sorted, or guaranteed to be balanced. But it could accidentally be those things'. We don't allow that sort of negativity and self-correcting to rear its head, so we politely acknowledge it, ignore it, and continue on with the understanding that not guaranteeing something in CS is as good as presuming it's not there. It's mathematically wrong, but practically much easier.

And that's where we are. Our explanation of Priority Queues was a little flawed, because we were trying to focus on what they do and how they do it moreso than how they worked with a full step-through of the process. Let's return back to Priority Queues, and specifically let's look at the implementation behind them, the Min/Max Queue.

To be clear, the specific train of thought we're focusing on is steps 5, 6, and 7 from above–

Spoilers: step 8 is 'What if I don't want it to be a tree, and I want the links to go every which way?' That's called a graph and it's in the next chapter! There's a whole field of math called Graph Theory that is fun, complicated, but way beyond the scope of this book. It's important to note that what we're doing here is creating a highly constrained situation (linked lists that perform like arrays) and then slowly removing those constraints in a logical manner so that we have useful data structures every step of the way.

–and we're going to refactor those steps in a completely different format:

1. Linked Lists are a worst-case situation from the time analysis, and finding anything is a O(n) operation. They are sorted and anti-optimally imbalanced.
2. Binary Search Trees are an average case situation from the time analysis, and finding anything is either a much better O(n) or a poor O(log(n)), depending on your assumptions. They are sorted, but neither balanced nor complete.
3. AVL Trees are a best case scenario for the time analysis at O(log(n)). they are sorted, balanced, and

complete.

4. The next thing we want to look at (hint: it's Min-Max heaps) should be what happens if we create a structure that's not sorted, but is balanced and complete?

This is where the real Min-Max Heaps come in. We go back to re-evaluate what that Priority Queue could do better now that we understand how to do the AVL trees. But then secondly, we also want to explore different variations of the labels we've applied to the different structures. Technically, we could explore what happens if one is balanced but not complete or vice versa, but we've already looked at it a bit and they're different labels but effectively inseparable in this regard.

Definition

To save some logical progression, we'll jump to a couple of conclusions. The question we post (what's not sorted, but is balanced and complete) is kind of useless without a constraint. Essentially, if we don't have a way to know whether to explore the ->left or ->right subtrees, we need to search every node and then we may as well have a linked list. So, we need to add a constraint, which is:

```
Min Heap: payload(root) < payload(children(n))
Max-Heap: payload (root) > payload (children(n))
```

They're basically the same, but this applies to all the nodes. We're saying that for any root, in sort order, it's always the least or the most of any of it's children. And, remember, because that's true for all of the nodes, it needs to hold true for all of its children. this is nice, because we get almost all of the benefits of an AVL tree without needing to do a lot of the rotations! Maybe some rotations. In fact, let's rename them to 'heapify' and then we can say Heaps don't use rotations, they just sometimes need to heapify. But since we don't need to enforce a strict sort, just a general order, we can make sure that the tree is balanced and complete without too much overhead.

We'll just cover the min-heap. The max-heap is almost always exactly the same just with >'s instead of <'s

Heapify

Here's your analogy, and it may take some leg work. Have you ever driven by road construction at about 5:15pm? The work day has kicked off, and there's just one poor intern stuck in the front loader or mechanical shovel, and it looks like they're just taking the bottom of the pile of whatever material it is (gravel, sand, select fill) and putting it on the top. Why are they doing that? There are two reasons, equally interesting. The first is sort of like telling the new mechanic to get a pint of blinker fluid, where you tell the intern 'we need those heaps piled bottom side up'.

The second and more realistic reason is that over time and as they pull shovels and trucks loads out of the pile, it starts to flatten. They track material all over the place, they make the heap less steep, and it generally just

gets more surface area. And that's the key, the amount of surface area directly corresponds with how much the wind or rain can mess with the properties of the material. Plus, it takes up more space. Plus, all of the stuff that got tracked everywhere has the potential to mix with the other materials and become waste. So, it's frequently worth it to set the new guy (or whoever needs to pick up more hours) to stick around and reshaping the heaps. They're gambling that the extra hour they have to pay and operate the machinery is made up for by the reduction in waste. Thus, it's a very common practice.

Interestingly, that's the exact same set of tradeoffs we're using when we use the rotations in the AVL tree or we use the heapify operation for the min-heap.

Let's take a look;

```
void heapify (struct node* arr[], int n, int i) // 1
{
      int min = i;
      int l = 2 * i  + 1;
      int r = 2 * i + 2;

      if (l < n && lessthan(arr[l], arr[min])) // 2
            min = l;
      if (r < n && lessthan(arr[r], arr[min])) // 2
            min = r;

      if (min != i) // 3
      {
            swap(&arr[min], &arr[i]); // 4
            heapify(arr, n, min); // 5
      }
}
```

Let's cover a whole heap of notes:

1. Notice that we've switched to the arrays. You can do heaps without arrays, but it's much easier to demonstrate when you know what the last node in the heap is. We'll show why in the next code segment. But, in short, it's easier to demonstrate what's happening and, in this case, it's almost always an array in practice. The AVL tree is not normally done with arrays, because the rotations are much easier with the pointers.
2. We assume that the root is the smallest node and then we try to prove that's not the case by contradiction. As a side note, this is an excellent analogy for another proofs technique. It doesn't tend to be used as much in algorithms, but you will sometimes find some of the end points. Basically, it goes like this

'assume that the opposite of what I want is true. Here's the logical contradiction that occurs when we assume that. Therefore the opposite must be true, which is what I wanted in the first place'. It's passive aggressive with social interactions, but it's invaluable with formal logic.

3. This brings us to a key point: we assume that if we were called to heapify something, and this little mini-tree is already a heap, we don't need to do any more. We're relying a lot on the fact that if everything else is held to high standards, then we can trust all the other parts are already valid.
4. Swap here is a magic hand-waving thing that we've done before. We don't care how the swap happens (whether it's by value, by reference, shallow, or deep), it'll get figured out in implementation.
5. This is important, though, we changed the order of the heap. If this small mini-tree was not a heap, we need to check that where we moved the old root to is also a heap.

This is a much easier operation to follow. Partly, because the utility is lower: we don't need to worry about whether everything is sorted, just this one little narrow microcosm. Hopefully, you can already see the tradeoffs between the simplicity of the heap (and lower computational overhead) versus the full AVL tree.

Full Heap

```
void heapify_up (struct node* arr[], int i) // 1
{
     if (i = 0) return; // 2

     p = (i - 1) / 2l; // 3
     if (lessthan(arr[i], arr[p]))  // 4
     {
          swap(&arr[i], &arr[p]);
          heapify_up(arr, i); // 5
     }
}

void heapify_down (struct node* arr[], int n, int i); // 6

void insertHeap(struct node* arr[], int* i, struct node* target) // 7
{
     arr[*i] = value;
     heapify_up(arr, i);
     (*i) += 1; // 8
}
```

```
struct node* popHeap(struct node* arr[], int* i)
{
     if (*i == 0) return NULL;

     struct node* ret = arr[0]; // 9
     arr[0] = arr[*i - 1];
     (*i) -= 1; // 8
     heapify_down(arr, *i, 0); // 10
     return min;
}
```

This is called a bait and switch! You thought we were just going to have 1 heapify operation! But now we've got 2 different ones! What's that all about? Well, it turns out that the heapify we gave earlier is the normal one. It makes sense, and we could use it in both directions. However, in the worst case scenario, there's only 1 swap, in the best case there's 0 swaps for both heapify functions. But when we look at comparisons, there's 3 comparisons for the one (left, right, and then whether there was a change) but only 1 comparison for the other. When the functions are basically only 3 or 1 comparisons, that's something around a 66% reduction in the function. This is not a time complexity analysis or a Big-O operation, you can literally count how many times things are going to happen. Both functions are O(1), because we simplify whatever is inside the parentheses. But, that doesn't change the more direct observation of the time the function will take.

The short of it is, we have a second function because the tradeoff of having a single extra function that's basically 6 lines long is worth being able to, half the time, have a significant reduction in the time that one of the heapify operations takes. You can guess what the difference is just from the name, but if not, we'll explore it in about lines.

Now, let's pop off these last few notes about heaps:

1. When we insert a node, we're going to insert to the bottom of the heap, and then let any out-of-heap nodes bubble their way up. Again, it's like bubble sort, but just one value and it's going up the short-ways with log(n) comparisons. So, it's not terribly inefficient, even if it shares one slight contextual similarity to an inefficient algorithm. (And, actually, Bubble Sort is arguably not really inefficient. It's more like the baseline. We can just do so much better that it gets bad rap).
2. This is a little awkward. We'll give you a little workaround here in t a moment. But, 0 is the sentinel value for the range of the array. Because we're going 'up' the array, the end point will be the 'left' side, which is going to be the 0 index. But that means we don't need to know how long the array is (n), because we'll never pass the right side of the array by dividing the index and going left. So, technically, we don't need to pass the max_index of the array if we don't want to. You might still, just to preserve consistency and to make sure there's not any constraints later when you want to adjust how you handle the internal

mechanics. But it's also Ok to not pass the length of the array. How do we know we'll always hit 0? Remember that for integer division we drop the remainder. so, we'll always hit 1 or 2. 2 / 2 = 1 and 1 / 2 = 0(.5, drop the .5).

    1. This means you can also use this as your conditional when you convert this from recursive to iterative loops.

3. We'll assign the index of the parent value, because we do use it twice. It's a toss up whether it needs to have a separate value. Many people will say 'if you use it more than once, you need to create a pattern for it'. But, on the flip side, since we need a third line to include this and the (i – 1) / 2 is so recognizable in this context (maybe even more recognizable than the variable name which changes from context to context), it might be more readable to do it without the temporary variable. Either way, the compiler will choose the most efficient version during the actual compilation process so it doesn't matter which you pick. Except: you should always pick what's most readable. By definition, what's most readable is whatever coding style guide you have to follow for work or organizational purposes, secondarily whatever way is most consistent with your previous code, and thirdly whichever you decide as a judgement call.
4. Notice that when we're going up, if we assume that the heap was correct before we started, and that every operation always leaves a correct heap when it finishes, then we will never have a situation in which the new node we insert on the right will be less than the node on the left. So, we only need to compare to the root node.
5. The recursive phrase. This only happens when the index > 0, so we'll never have an array out of bounds or an infinite recursion.
6. This is the code we covered in the previous subsection, but it's cleaner to specify that it's heapifying down.
7. Notice how much of the benefit of the heap is based on the presumption that we know what the current index is. So, we can just insert there. How do we know what the index is? We'll just be sure that it's correct after every operation.
8. This is where we guarantee that we've maintained that order. We increment the size of the heap on insert, and we decrement the size of the heap on a pop. If we were programming overly defensively, we might periodically have it re-count the size of the array, but that's conceptually hard to do. Since we don't really 'delete' values, we only ever overwrite them and change the bounds of the valid portion of the part of the array that represents the heap, it's difficult to tell what's real data and what's old data.
    1. We could increase complexity by adding a known stop value like MAX_INT arr[i + 1] and maintain that value. But that makes the array more complicated and doesn't add any actual utility. It would only be useful if you specifically expected there to be, say, a concurrent overwrite error.
9. As usual, make sure we hold the returned node separate. Notice we don't need to worry about the various pointers, because the pointers are inherent in the index relationships between i and (i + [1,2]) / 2.
10. In short, we pull off the root node of the whole heap. Then we swap the furthest out node into the root. Then we reheapify the whole thing, but we only ever need to take one path downward when the local roots are bigger than any of the children. Remember, we just need to adhere to the rule that every root

is the smallest node in its subtree, so it doesn't matter if it goes left or right. We'll just track it down like that.

So, overall, inserts get an O(log(n)) because we only ever need to trace through one vertical traversal through the tree that represents the heap. And delets get an O(log(n)) because we also nly need to trace through one vertical traversal through the tree that represents the heap, just in the opposite direction.

Helper Struct

It might be helpful to, instead of having a global variable of struct node* arr[], instead to use this:

```
struct heapArray
{
	struct node* arr[MAX_SIZE];
	int n;
	int i;
};
```

But then you'll also need an instantiating function (this is the equivalent to a constructor, and what a constructor gets morphed into when it gets compiled from C++ or java or any other Object Oriented langauge into C):

```
struct heapArray* newHeapArray()
{
	struct heapArray* ret = malloc(sizeof(struct heapArray));
	ret->n = MAX_SIZE;
	ret->i = 0;
	return ret;
}
```

In general, we want to avoid global variables. But the specific implementation for how any of this applied shouldn't be conceptually different at its core, just in its details. The idea is that all of these data structures that we're covering as part of the field of Algorithm Analysis and Design are more or less universal and can be adapted to fit any situation and coding environment.

Reiteration of Completeness

What's interesting here is that the heap isn't necessarily balanced. Or rather, the heapify function won't

balance your tree. We presume that it's balanced. Just like the previous sequence for the AVL tree. We assume that the empty heap is balanced and complete. And then if we just add a node at the next available slot, we get a heap of size 1 that is balanced and complete by default. If we insert any nodes, we always just add them to the next available slot, which means that a complete and balanced tree will continue to be complete and balanced. If we delete a node, we make sure that it's the last available node that's deleted, insuring that the node is balanced and complete still.

The point is that we don't guarantee order, we just follow our one simple constraint of a min or max heap. But otherwise, the balance and completeness are inherent in the fact that we only ever interact with the node that will insure we maintain our balance and complete attributes.

Practical Applications

In Zen Buddhism, the idea of a Koan is intended to startle your brain out of it's normal pattern of thinking with, reductively, absurdity. So, you've heard the question about 'what's the sound of one hand clapping?' Or, perhaps you've heard about the monk Budai. Taking great liberties, someone asks Budai what the meaning of Zen is, so Budai puts down his carryback. The guy goes 'I don't get it, what does that mean?' So Budai picks up his sack and wanders off. Here's a similar question:

Why 2? Why do so many things in computer science revolve around 2?

Let's not answer that question, let's answer a different question. Imagine being in an office job, and the Big Boss asks whether something can be done, what kind of answer does he want? We could be cynical and say 'he only wants to hear Yes!' But, a little more prosaically, there are two: 'yes, and here's why' and 'no, and here's why'. The why is important, but in the mind of the decision maker, they want answers broken down into two categories. Suppose you're producing a data visualization for your boss. You can cover dozens of scenarios and have as many different outputs as you can fit. But, typically, the Big Boss wants to know 'which ones worked and which ones didn't?'

The point is that there's a simplicity in the decisions between acting and not acting. Did you know that ninjas (that is, literally students of the modern school of Ninjutsu, the Bujinkan Dojo) have to write papers in order to advance in ranks after a certain point? One high ranking Shihan wrote a paper about how the English proclivity towards three verb tenses was a misnomer and that there were really only two tenses: present-past and present-future. Either what you've done or what you will do.

Regardless, all of this could also be boiled down into Claude Shannon's Mathematical Theory of Communication. In it, one of his claims is that binary is important because it is the smallest alphabet possible for encoding information (that is, the one with the fewest number of unique symbols) and that it is possible, for any piece of information, to determine the minimum number of bits necessary to represent that information without loss.

What's the point of all of this? Not much, it's just some anecdotal bits for you to keep in the back of your

brain. Sometimes it helps to have a set of good anecdotes to talk about, and for most people, this is as in depth as you need to go in those stories for most people to be interested (or not interested). Feel free to make use of them, should the opportunity arise to tell pithy stories! We won't be able to get to an ontological 'why' through these, though.

Let's consider the opposite direction. We can at least answer 'why' Binary Trees are so effective. To do that, let's consider the opposite. We already discussed the idea of proof by contradiction. This won't quite be that, but it's a good place to start. Let's presume that 2 child nodes was not the most effective approach. Maybe it's 5 or 10 or 255. Either way, we just need to iterate through the numbers and if we find even one that's more useful than the Binary Trees, we've won!

So, let's look at the pros and the cons. Why do we like binary search trees? Because while we tend to write it as log(n), it's properly $\log_2(n)$. That is, Log-base-2 of the number of inputs. But, if we had, for example, a trinary tree: one that has three children, we might have a significant improvement. For example, suppose that we had a tree with 1 million entries. The depth of a balanced and complete binary search tree would be 20 layers deep. But if we had them as a Trinary Search Tree, we'd see that a balanced and complete tree was only 13! Sure, we'd still be using 1,000,000 nodes, so the space would be exactly the same (ignoring the slight variation in NULL pointers that would be 3 for each of the ~530,000 leaf nodes and 2 for each of the ~525,000 binary leafs), but the computation time would be around a third faster with the trinary! And that increase increases as we increase the number of nodes, since the base of the log function would also increase.

However, we run into three main problems:

- Nobody else uses trinary trees. If it was true that 'more child nodes is better', there'd be a split in the available algorithms and there wouldn't be an agreement on how best to do things. So, there's really a strong rationale behind the fact that because more people use Binary Trees, more people use Binary Trees. It's called the Network Effect. Consider: suppose there was a technical reason why you could gain a small benefit by using slightly different inputs, but you'd have to develop the algorithm from base mathematics principles to take advantage of it. Just like almost everyone else in a production environment, you'd be better off using the widely used and supported version.
- The algorithms would be much more complicated. In fact, the algorithms would be more complicated in the order of $2^{n(n-1)}$, which is the number of combinations that might occur between the various child nodes. Remember in AVL, when we wanted to maintain sorted, we had to rotate? And in the rotation there were 4 combinations: LL, RR, LR and RL. If we go to three, we need all possible combinations (LLL, LLR, LLC, LRL, LRR, LRC, LCL, etc...), and so forth. So, the actual calculation time for insert and delete becomes prohibitive quickly. That's not a problem if insert and delete rarely occur. But the tradeoff requires a much higher ratio of reads to writes to be viable.
- Finally, and this requires bit more inquiry, where do the splits occur in order to make everything even?

Suppose you want to split a set of numbers that are from 0-100. The immediate guess is at 50, right? But,

and this is also an important ideology, we typically want to have the data be self-defining and not related to an external standard. 50 is a fine split for uniform data, but what if the data was grades in a 3rd grade class and there were 2 below 50, but 26 above 50? It would make more sense for the split to be at the average of the grades, median(grades) (or mean(grades) for a sometimes faster substitution).

Digression to practical applications… in the Practical Application section

But where do we make that split for tripartite data? Well, obviously at the first and second third splits, T1 and T2. However when we start adding the data, it gets more and more difficult to check where the splits are. We need to check the heights, and again, the rotate might need to rebalance and put something from the first third into the third third (for example, suppose there are two inserts into the first third, one insert into the second third, and then a third insert into the first third, balance comes from moving it to the third). But, we could get around a lot of this algorithmically. The issue just comes in the fact that the number of comparisons increases dramatically, even still.

```
--Start--
```

One interesting application might be a situation where the number of similar values is common. For example, maybe the tree is sorted by the year of manufacture for cars. It would be silly to have all of those trees bounce between different values. So, you might decide to have a third branch that's for values that are equal to the current node.

Of course, then the question comes up with what sort of data structure to put as the third branch. It could just be another subtree. But it could be an array since all the values are equal. It could be a linked list. Maybe it's a completely separate tree, but sorted based on some other value (like, month of manufacture, VIN number, license plate, make/model).

```
--End--
```

This is how we do data structures in practice. We start with an idea and then we extrapolate what to do with various bits. Maybe it's the same, maybe it's different, let's brainstorm! What's even more interesting is that a lot of the current LLMs are fantastic at brainstorming these structures. Between Start and End, try pasting that into an LLM and ask it what kind of structure it is. And then ask it to improve that structure. This back and forth, before the code is even written, is an excellent way of refining and adapting data structures and algorithms to specific applications.

Bringing it back

So, the answer basically comes down to: it's expensive and it's harder to track. It turns out that keeping track of 2 things is just much easier, and the simple conditionals make things much cleaner. You should have started getting pretty comfortable breaking down the two or three cases for most of the functions. Throwing in extra cases means extra comparisons that are being made.

But, the most important issue really boils down to: it's not common. Given the opportunity to write something that nobody else understands or one where you can have discussions with colleagues about how to improve your code, always go with the latter. It means that when someone else takes over the code base, they have a solid grasp. It means that when you have to justify your decisions, you're not constantly explaining from scratch. And, most importantly, it means that you aren't reinventing entire swathes of a codebase for minor gains that may be outweighed by the inefficiencies of a new algorithm over well-optimized ones.

Side Note: This isn't to say don't ever try something new. Just that the fact that everyone else does something, there is institutional and industry knowledge around specific solutions is a significant source of inertia and it's worth considering the cost of fighting against it.

AI Examples

The same questions applies to AI. Why 2? But it requires some backstory.

AIs are trained to do specific tasks. Maybe in the future they can do other things. But right now, an AI is always trained to do a specific thing. That 'specific thing' isn't always what you thin of. For example, when we want to know 'what animal is in this picture?', we can't really train an AI to do that. We can train an AI to say 'is it a giraffe?' 'is it a bonobo?' 'is it the mythical chimera?' And, essentially, have it check through each of those, and then it can pick the one that has the highest levels of confidence in the answer. So, it's definitely a Tarsier Mole based on the hypothetical numbers we're looking at.

We can design systems that will train multiple AIs at once, for example, we can give it lots of different pictures with different labels on what they are (the training data) and then use that to reinforce which ones it can correctly identify in unlabeled data (the testing data). But, the point is that each of the artificial intelligences is being tested over different things. These are called Artificial Specialized Intelligence (as opposed to Artificial General Intelligence), although the current common term is Narrow AI.

Side Note: When an AI can be trained to train itself in tasks is one of the many definitions of an AGI. It's also the precursor to the singularity event.

Statistical AI

One caveat: we don't know how the internals of a neural network work. Anyone who explains differently is trying to sell you something (probably literally).

How does one of the big AI systems like a chatbot or an image generator work? You can look into the Transformer model, which is the 'new' model (it's been around for a long time, it just recently got a huge bump with a specific paper called 'Attention is all you need' that took advantage of the massive gains in computational capacity since the model was first introduced). But that and the attention model are a little outside the scope of the text.

Instead, let's look at the data structure that they use to model it. They use a large multi-variable matrix. When we say 'multi-variable', we mean thousands or more dimensions. This is the conceptual space that the AI operates out of. Trying to track what's going on is, by the most successful current exploratory approaches, similar to evaluating human psychology through brain imaging techniques. We can take the giant matrix at any given point and run it through various filters to see which areas are still active.

One infamous result right now is the Golden Gate Bridge effect. One team found that when they trained their AI, they could isolate a region in the matrix that corresponded to the AI's desire to talk about the Golden Gate Bridge. If they turned up the numbers (weights) in that region, the AI would almost always bring the conversation back to the Golden Gate Bridge, but if the numbers were reduced, the AI would never mention it even when directly referenced.

The interesting part is that it's still a binary question: Golden Gate Bridge or Not Golden Gate Bridge. And the values were all related to whether or not it would discuss that idea. Now, there's a strong likelihood (but no proof) that these numbers would also affect other concepts that the team just didn't find. Or, possibly, it has nothing to do with anything and it was all coincidence. The mechanism is not understood, just the specific effect in that controlled environment. But, again, it's interesting that the floating point numbers that were used were just there as a finely graded weight towards a specific binary decision.

The point is that the more complicated the program becomes, the more important it becomes to remember to bring the outcomes back to simple outcomes, and that almost all conclusions are going to be binary in nature.

The Models

The point is those matrices. If we have thousands of dimensions, we can encode them in a string of vector values. But there are thousands of different states we need to consider, each of which gets its own vector. And, what's worse, any generalization here is likely to be deprecated before you read it. So, let's narrow down to just the actual values:

What do those values do? They're largely there in order to break an outcome down to a binary value. sometimes it's whether something is above a certain threshold, sometimes it's if it's the biggest value, other

outcomes. But the point is that they're huge levels of processing to more accurately predict whether a specific factor should be true or false.

So, why would you feel compelled to ever be limited to a single data type? Right now, floats are the most common value to put there. But what's to stop you from experimenting with simple trees, integers, 1-bit boolean values, the outputs of separately executed programs, etc.

And this applies to all of the data in this chapter: You're not limited in scope to what we present. You can experiment with a lot of things. Just remember that the Golden Triangle still applies: Between fast enough, small enough, and exactly correct, you can have two of the three (if that). If you meet all three, it suggests that you should be pushing a little harder for a more comprehensive process.

Media Attributions

- Bal_tree8_2
- 8_2
- 8_3
- 8_4 (1)

PART III

PART C - THE TRUE GRAPHS ARE THE FRIENDS WE MADE ALONG THE WAY

Up to this point, we've been presenting information in two ways, akin to the two problem-solving approaches:

- Top Down when we just assume you can keep up with terminology and application, and we start using it regularly, explaining spot mechanics until we get to the big explanation. Big-O is an example of this, that we explain in Chapter 10.
- Bottom Up when we meticulously pre-explain many concepts so that the explanation itself will make more sense. Most of the Sorting algorithms work like this.

However, we've largely gone over the principles that we want to cover at the lowest level. So, luckily, these next chapters will go by much faster. We are going to assume that you can apply the material from the previous chapters and just jump into the explanations and applications straight off.

CHAPTER 9 - GRAPH... FINDS A WAY

Once again remember that a textbook has a lot of similarities to a story. There's the introduction, the rising action, the climax, and then the docuument. We are no past the rising action, and you should feel that we are now approaching the climax of the action. In some ways, this is the simplest set of chapters, because a lot of the 'yah, but they never use this in practice' is replaced with 'this is basically what they do, with a few caveats and updates'.

In short, you should feel comfortable with data structures and algorithms. You may have to create a data structure from scratch, but more likely you'll be adapting an existing and known algorithm to your specific data set, and you'll just be adapting the data structure used by the algorithm to adhere to your data. That should be a comfortable process for you, since all of the previous chapters have been discussing how different algorithms work.

As a review, we have two broad categories of implementation: Pointer-based nodes and the array. Sometimes we like the pointer-based nodes because we don't know what the eventual structure or size will be, or because adjusting the relationships between nodes is easier when it's solely a local affair. Sometimes we like the array implementation because we want to track some metadata about our emergent structure (it's actually less of an emergent, implicit structure and more explicit here) like length or max value. It's almost always a question of which is easier, measured by readability and reusability.

In terms of broad strokes structures, we covered:

- Basic, informally defined (from a structured algorithms perspective) code in C, possibly with some light structs, but not with any formal algorithm or data structures logic.
- The individual Node structure, a wrapper for other data.
- Linear data structures, and their sort and search options.
- Tree data structures, and their various permutations of sorted, balanced, and complete.

You should feel a sense of dimensional expansion there. It's not a perfect analogy, but it's roughly no dimensionality, 0-dimensional points (the nodes), 1-dimensional lines (the lists and arrays), and then what comes next? It would be nice to say '2-dimensional', but we don't really see that. What we see is more of a halfway point between 1-dimensional and 2-dimensional. If we think of 1-dimensional as moving freely on a line and 2-dimensional as moving freely on a cartesian plane, the tree structures don't really fit that.

Graphs

So what does fit more with the 2-dimensional nature? This is where we start to introduce the concept of Graphs. Graphs are still constrained, they aren't really a coordinate system. Instead, they fall much more in line with, say, a flowchart. We have a set of nodes and from any node we have a set of links that go to a set of other nodes. Generally, but not always, these are going to be uni-directional links. You can move from node A to another node B but not back. Or more precisely, if you can move back from B to A, there's an explicit link.

If we were going to study this in a formal logical sense, we would be describing relationships, like the below:

```
Given nodes A, B, C, D, E, F, G, H
A -> {B, C}
B -> {A, B, C}
C -> {D}
D -> {E, F}
E -> {G, H}
```

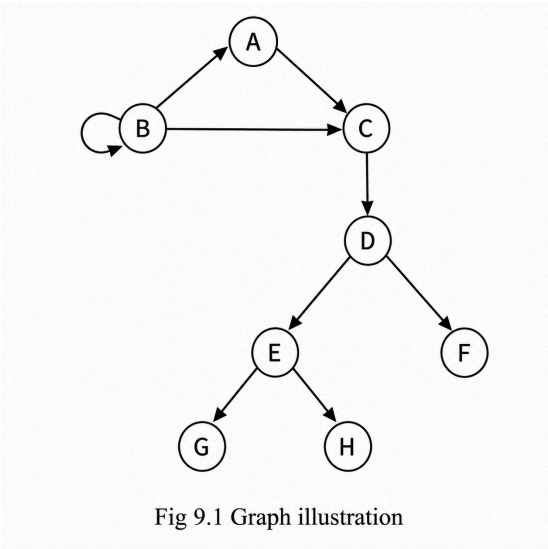


Fig 9.1 Graph illustration

So this would read something like 'A can go to B or C, B can repeat B, go back to A or continue to C. C goes to D, and D can go to one of E, F, G, or H'. This might be a broad overview of a state engine for an input verification and classification system.

Caveat: We're going to heavily imply, several times, that we 'move' from one node to another. This is purely pedagogical rhetoric. And sometimes it's actually real. But the truest definition is just that there is a relationship, not necessarily that we're going from one to the other. This, of course, is belied by the fact that we often 'traverse' the graph just like a tree! But that's an applied idea not an intrinsic one.

Directed/Undirected.

It's important to note that there are two different kinds of graphs: directed and undirected. In a directed graph, the arrow matters. That is, for example, in the previous node we can only go from the left node to the node on the right. We can also call this a unidirectional or a simplex relation (or one side of a half-duplex connection).

An undirected graph, on the other hand is one where the nodes are just linked together and direction is not implied, just the connection. This means that if it's a state machine (sometimes represented as a graph), the fact tat there's a connection from A to B and a connection B to A are both implied by either relation. We can call this bidirectional or (full) duplex relations.

Here's the trick question: which one is more constrained? It turns out that even though 'undirected' seems like it would be less constrained, that's not actually the case. We're imposing the constraint that all relations have to be bidirectional. But, that's definitely a constraint, how do we know?

Suppose I have the bidirectional relationship between A and B. I can represent it as A -> {B}, B -> {A}. However, suppose I have the unidirectional relationship A -> {B}. How do I represent that with a bidirectional notation? We can't. And, in fact, we can also demonstrate that any bidirectional relationship can be represented by two unidirectional relations. But a unidirectional relation can't be represented in any context by bidirectional relations.

Since it's trivial to represent bidirectional with unidirectional (you literally just need to note the there and the back again, like a Hobbit's Journey), we'll be using unidirectional throughout. Sometimes there are unique concepts that apply to bidirectional. You just need to be aware that it's a possibility and figure out how to handle it.

Relation vs. Edge

We keep referring to the lines that connect the different nodes as 'relations'. Why is this ? It comes back to Databases, which are a basic subfield of computer science as well. When we have a tool like SQL, it's a Relational Database. What are those relations? They're the same as these relations, just a completely different application! But most of the math still holds for all of it. That math is a branch called Set Theory, which in turn uses a set of rules called Relational Algebra. When you see $\bowtie$, $\cap$ or $\cup$ symbols, they represent the join, intersect, and union operations respectively.

However, in Data Structures and Algorithms, we tend to just call them edges. They aren't really mathematical or geometric edges. In DSA, they're their own thing. It's just because it's a visually similar line segment to what we have in geometry and because when we're talking about the nodes themselves the very outer perimeter is sort of touching a set of other nodes very outer perimeters in a defined manner. So we talk about their edges, and it's just easier to call the line between them an edge. So, from here on, we'll use 'edge', but we started with relation to make the specific callout that they are like but not the same as (or the same as,

but not at all alike?) the similar concept from Databases. Both are technically correct, but edge is much more common.

Mathematically, and from a proof nature, the fact that any one node has an edge connecting it to another node is a property of one or both of those nodes. It's just a descriptive fact about them. But the edge itself has no real semantic weight or value in its own right. We might also say that the edges are contingent on and derivative to the existence of the nodes. But they are not a thing themselves.

In programming, however, it's practically impossible to represent them like that. It can be done, and we'll explore that a little bit. But, realistically, every node is a thing and every edge between nodes is its own thing. In fact, it almost becomes true that in programming, because the edge is the important part, the nodes themselves become contingent on and derivative to the existence of the edges. That's not perfectly true, but it's close to the mark. Either way is dependent on the specific application and graph type we're dealing with.

This is not going to be a problem in the text. We will typically use the programming approach, however a lot of graph algorithms explanations 'in the wild' sometimes borrow a little too heavily from the mathematical roots, which means that you need to be aware. If you can't figure out how to implement an algorithm, consider if it's because the edges aren't explicited given semantic weight and doing so will frequently clear up confusion.

Degrees of Freedom

How many different linear structures are there? There's a fair dinkum, sure. But, there's only so many different ways you can permute 'a line'. How about the tree? Same thing, there's lots of variations from B-Trees to Black-White trees to tries to Binary Search Trees. But the hierarchical and frequently constrained nature of the trees makes them basically similar. In both cases, we like that! We like the fact that the constraint lets us define explicitly what activities can be done.

But graphs? They're a much more unconstrained version of the former data structures. What this means is that there are a ton of different graph applications, implementations, goals, and algorithms. Don't be overwhelmed, but also don't be afraid to dive in and look at what the intentions are. After all, at their core, the transfomer model, attention, and neural networks are all just specific graphs. Of course, saying that a modern LLM's neural network is 'just a graph' is sort of like saying a modern CPU is 'just a circuit'. Still, every one of these steps we've taken is the stepping stone towards multi-trillion node networks.

What's after graphs? It gets a little more abstruse. We might call it just 'unconstrained graphs', such that maybe the nodes aren't real or the edges aren't real or we add something like a coordinate system so that there are theoretically an infintie number of nodes. For example, a voxel system might be an 'unconstrained graph' in the sense that each node is in relation to each other node based on physical distance in a 3D volume. But the description gets a little strained (but still mostly correct) at that point. We could also call an unconstrained graph a 'map', as in 'mapping values from input to output'. In some ways, we're saying that the Unconstrained Graphs keep the node and lose the edge, while the mappings keep the edge and lose the node. But, just like

anything else it gets murky. We'll explore a few of these in later chapters. Just keep the notion for now that a graph has the nodes and the edges. but no rules as to how they go together, and that's what makes it a graph.

The point is, your ability to manipulate and adapt a graph is higher. If you're looking for a complex data type that doesn't have an immediate analogue as to how to represent it? Try graphs. We'll look at Time Series Data in a later section.

Weighted/Unweighted

So, it's important to remember: the edges are concepts in their own right. They have two end points (this is a constraint, feel free to look up what removing that constraint looks like). But what if there was more? Typically if the edges themselves are just the connecton between two nodes, we call this an unweighted graph. For example, if you were going to consider the relative accuracy of 'the boy _____ the ball' and you had to choose between 'holds', 'carries', or 'bears', none of them is syntactically more correct. It's more a question of is or is-not correct. So, we can't say 'the boy 54446 the ball', because that's not even English. But those three words are not just the same type of word, but they are synonymous to each other so it just doesn't matter.

Incidentally, a Context Free Grammar is also a type of graph. Notice that the 'rules' look a lot like relations or edges.

A weighted graph is the opposite. A relative amount of edge-ness gets applied to that edge. What the edge means is sort of like listening to Morpheus: The edge means something; what it means you can't explain, but you feel it and you've felt it for the entire graph. Or we could just say that the weight only has meaning when it's in context of the other weights of the other edges of the graph and that the semantic meaning is applied externally by the agent observing it (that's us, the humans). But, mathematically, it's still true that the comparative value of the nodes is only in relation to each other. that's why we call them weighted, but maybe some examples would help:

- On an airlines map, each travel hub is connected to each travel hub by specific airlines, and the cost, fuel usage, or distance is the weight that they might use when determining the best way to get someone from one location to another.
- In a classic video game rules-based AI system (not typicaly a neural net, more of a state diagram), there are different weights to determine the likelihood of any given action.
- In A*, the edge between any two nodes is the conceptual weight associated with traversal from one to the other, and the algorithm wants to create overall traversal paths that minimize that total weight between any two non-adjacent nodes. It's a map-based algorithm that we'll use shortly.

Graph Example Code

There are actually three different versions that we need to cover quickly.

Adjacency List

Basically, in this version, each node keeps a list of each node it connects to. This is the obvious approach, and something between a naive and a brute force approach. It's benefit is that it's modular and that each node self-describes itself.

```
struct node // 1
{
     int payload;
     struct node* next; // 2
};

struct edges // 3
{
     struct node* head;
};

struct graph
{
     int i;
     struct edges* arr; // 4
};
```

Let's run through some of the edge cases of understanding:

1. They're still just nodes. No matter how fancy they get, it's the same basic idea: we've got payloads and we're organizing how they are related to other.
2. The entire set of nodes is just a big, old linked list! This particular linked list isn't really the relations, it's just a way to be sure that we can touch each node. For example, what if there was a way for a single node or a set of nodes to be completely disconnected from the rest. We'd still need to know it existed but we couldn't guarantee we could get there from the nodes. So this is our best approach.
3. We don't need to have this be its own struct. We could just always refer to the node* head or whatnot. But, realistically, we want to have the self-descriptive code and also separate out the implementation from

the structure. What that means is that right now we're using a linked list to interact with the edges. But we can't guarantee that our use case won't change or that our needs won't change, in which case maybe we change what's contained inside of this struct. The result is that if we have it as a struct, we just change a few of the functions that interact with it when the internals change. But if we have direct access to the *head as an attribute of graph, then we run into issues when it changes.

    - This is not just a justification to avoid the situation where we have an array of linked lists, but we pass it by having to have pointers to pointers. It really is cleaner to do it this way.

4. The graph itself is an array of pointers to a set of edges. We'll want to initialize it with a `malloc(MAX_NODES * sizeof(struct edges))` or something similar. But the point is that we're going to have an iterable array of these edges that we can cycle through in order to produce the graph in whole later.

So what's going on here? We're going to need a few extra functions. We need a few extra helper functions to make all of this work. We'll need an initialization function for the graph itself that allocates the array and sets i to 0. We'll need a function to be able to add edges to this graph. And then when the edges refer to nodes that don't exist yet, we'll need to add them to the main linked list. We could delete the edges, but that's not really normal. Generally, we generate the graph from, say a file or a set of inputs, and once it's generated we run our calculations or algorithm on it from there.

Delete Side Trail

The point is that the deletion doesn't always happen after we've started modelling the graph. Sometimes we set up the algorithm so that the node is deleted ahead of time and then we just recreate the graph. Why is this?

Broadly, edge deletion can lead to cascading effects. Consider the graph in Fig. 9.1. If we delete the edge $C \to D$, should we also delete node D and the nodes reachable from it $E, F, G,$, and H? The answer depends on the purpose of the graph. In some applications, unreachable nodes may no longer be useful and can be removed. In others, the nodes should remain even though they are disconnected from the main graph.

We might instead delete a node only when the removed edge was its only incoming connection. However, this raises another question: should an entire disconnected component or cycle also be deleted? For example, if $A \to B$, $B \to C$, and $C \to A$ form a cycle that is disconnected from the rest of the graph, should that cycle remain? Detecting and removing such structures requires additional traversal and computational effort.

The point is that these questions are all answerable in concrete examples, but difficult to have in a context-less example here. The end result being that we frequently just don't see the delete portion for the general algorithm. Which is fine, in its own right. Just remember to look up or develop the delete algorithm that works.

Challenge

We need to move on to the other examples. So, your challenge is to use this set of structs and create the helper functions that mirror our later functions, but with this adjacency list function. Further, if you wrote a program that created the graph, populated the graph with relevant edges and nodes, and then displayed the graph (typically just as a set of rules), that would be a good exercise.

Adjacency Matrix

This is the cool one. You should remember how we could represent a tree or a heap as an array. We can do the same thing for these edges. What is the edge, when it's unweighted? It's just the presence of a connection between one node and another node, right? If we have presence or not presence, isn't that just a binary value? So, suppose we have n nodes. The total number of connections we might have is `n (n - 1)` nodes. Or even `n * n` is we count self-directed loops. So, what if we had n arrays, each one was n long.

Matrix Indexing

There are a number of ways to do this. We can create an array of arrays. We can create an array the size of the matrix and then deal with each individual node. Or we can do the latter but then type cast it back to the former. That third option is a trip, so we aren't going to do that. And, in fact, the first two options both have downsides. The issue is that when we want to pass this matrix between functions, we run into the same function-passing issues we've always run into.

Suppose:

```
struct node** mtx = malloc (MAX_NODE * MAX_NODE * sizeof(struct node*));
```

This creates one large, contiguous array of pointers. Because each entry is only a pointer, it may require less memory than storing complete `struct node` objects in every position. However, for a large value of `MAX_NODE`, the matrix can still consume a significant amount of memory. We can access the entry in row `i` and column `j` in either of the following equivalent ways: `mtx[i * MAX_NODE + j], or (mtx + i * MAX_NODE + j)`. Both expressions refer to the same matrix entry. A three-dimensional matrix can be flattened in a similar way, but it is best to become comfortable with two-dimensional matrices first. Regardless of the number of dimensions, a macro or helper function can make indexing easier and more readable.

```
#define MTX(i, j) mtx[(i) * MAX_NODE + (j)]
Then you can write: MTX(i, j) = some_node;
```

That tends to be the one of the most common approaches in low level programming. We are compromising

between the random-access nature of the array and the single extra lookup it takes to get to the actual data structure. This is one of the unavoidable uses of the ** operator. Because, ultimately, the array/matrix is a pointer to a set of pointers.

But, sometimes we want it to be even more immediately accessible. So, we'd do:

```
struct node* mtx = malloc(MAX_NODE * MAX_NODE * sizeof(struct node));
```

And, in that case we have the actual nodes immediately available. Luckily, we can still use either the `mtx[i * MAX_NODE + j]` or `**(mtx + i * MAX_NODE + j)` syntax, because the pointer math of '+1 is the size of the data structure' is implicit in the declaration of the pointer to given structures (which is why, in the previous method we needed to declare it as a ** so that it knew the data type was the size of a pointer).

Finally, if we really wanted to use the [][] notation (which is not necessary), we can use the following

```
struct node*** mtx = malloc(MAX_NODE * sizeof(struct node**));
for (int i = 0; i < MAX_NODE; i++)
{
      mtx[i] = malloc(MAX_NODE * sizeof(struct node*));
}
```

And this will create us an array of allocated arrays. What's different here? We still have the random access ability, but because it's an array of pointers to other arrays (even if that phrase is slightly redundant), those other arrays might be anywhere else in memory. So, we lose a little bit on the lookup time. This is not a very common approach at all for that reason, plus nobody wants to be the monster that introduces the *** operator. However, you might see:

```
struct node** mtx = malloc(MAX_NODE * sizeof(struct node*));
for (int i = 0; i < MAX_NODE; i++)
{
     mtx[i] = malloc(MAX_NODE * sizeof(struct node);
}
```

This will allocate arrays of the actual values rather than pointers to them elsewhere. But, since they're all normal, it's fine.

We cheated a little bit in this example, though. It's nothing but surprise reveal after surprise reveal! We won't be creating a matrix of nodes, because we only need the single array or linked list of the nodes. Instead, we want the array to be (to start) of type bool! Since bools are actually smaller than pointers, it's almost always going to be the case that we implement a low level matrix of Booleans like:

```
bool* adjmtx = malloc(MAX_NODE * MAX_NODE * sizeof(bool));
```

Side node: if we wanted to be super optimized, but horrible for readability, reusability, or maintainability, we could even just do: `bool* adjmtx = malloc(MAX_NODE * MAX_NODE),` because `sizeof(bool)` is almost always 1 byte.

Correction: when we use * or ** or *** to demonstrate that the type of variable we're declaring is a pointer, it's technically not an operator. ISO/IEC 9899 (the C standard) specifically dictates that when we use the * in the declaration, it is a 'type indicator' and not an operator. It is our opinion that 'type indicator' is a confusing thing to throw in the middle of the already confusing matrix math from a moment ago. However, * is technically sometimes an operator and sometimes a type indicator, which means *** is technically not a monstrous operator, it's a monstrous type indicator.

Matrix Adjacency Code

This is the one that's the most fun, so let's go through all of the code:

```
void main (void)
{
     bool* adjmtx = malloc(MAX_NODE * MAX_NODE * sizeof(bool));
     for (int i = 0; i < MAX_NODE * MAX_NODE; i++)
     {
          adjmtx[i] = false;
     }

     adjmtx[0 * MAX_NODE + 1] = true;
     adjmtx[0 * MAX_NODE + 2] = true;
     adjmtx[1 * MAX_NODE + 0] = true;
     adjmtx[1 * MAX_NODE + 1] = true;
     adjmtx[1 * MAX_NODE + 2] = true;
     adjmtx[2 * MAX_NODE + 3] = true;
     adjmtx[3 * MAX_NODE + 4] = true;
     adjmtx[3 * MAX_NODE + 5] = true;
     adjmtx[3 * MAX_NODE + 6] = true;
     adjmtx[3 * MAX_NODE + 7] = true;

     for (int i = 0; i < MAX_NODE; i++)
     {
          for (int j = 0; j < MAX_NODE; j++)
          {
```

```
                printf("%d ", adjmtx[i * MAX_NODE + j]);
            }
            printf("\n");
        }
        free(adjmtx);
        return;
}
```

This should give us the output:

```
[ 0, 1, 1, 0, 0, 0, 0, 0, 0, ...
  1, 1, 1, 0, 0, 0, 0, 0, 0, ...
  0, 0, 0, 1, 0, 0, 0, 0, 0, ...
  0, 0, 0, 0, 1, 1, 1, 1, 0, ...
  0, 0, 0, 0, 0, 0, 0, 0, 0, ...
  0, 0, 0, 0, 0, 0, 0, 0, 0, ...
  0, 0, 0, 0, 0, 0, 0, 0, 0, ...
  0, 0, 0, 0, 0, 0, 0, 0, 0, ...
  0, 0, 0, 0, 0, 0, 0, 0, 0, ...
  ... }
```

Depending on how big MAX_NODE is. Speaking of which, we're talking memory here. So, if we thought that 1024 was an appropriate size for the trees before, that means that 32 would be an appropriate size for `MAX_NODE` in this adjacency matrix setup. This is because suddenly we have an $O(n^2)$ use of space as opposed to the tree or the linked lists $O(n)$.

There's no struct, how can this work? That's the beauty! It doesn't need the struct! This is entirely an implementation independent solution! We can use this adjacency matrix regardless of what our original source looks like. One might even be able to set up an array of nodes and access them using the heap or the BST code we used earlier, and then also have a set of edges that also connect between the nodes. It would contravene the application we've used for one of the other of these. But, that just means you'll need to adapt one to fit the other!

Weights

This is also, arguably, the easiest one to add weights to. Instead of just using a bool to indicate the presence of the edge or not, we can use a float (or an int or even a string if you had a specific calculation method) the amount of whatever the edge is measuring. That's, typically what the weight means. Distance, mass, cognitive

displacement, any other kind of displacement. It's realistically less common in the grand scheme of things to have AI-style meaningless weights, although not rare by any stretch.

How would you adjust the other two methods to add weights?

Edge Array

What's the takeaway from the last method? The data structure doesn't have to be in the data structure to be useful! Remember in one of the recent chapters we talked about sparse versus dense data. When we have a matrix like this, it's sparse data. Technically we could call it dense because the fact that there's not a connection is just as true as the fact that there is a connection. But it's an incredibly rare problem that asks 'what are all of the correlations that don't exist between unrelated values?'

So, that being said, let's pretend that it's information sparse. Is there some way that we can distill just the positive statements out of the matrix? Absolutely! And it's literally exactly what it sounds like. We just declare:

struct edge
{
int src;
int dest;
};

And then we have a denotation of the struct. What's nice is that with the ints like this, we can utilize it with any of the implementation methods whether it's a linked list or an array. The int for the id of the node is either the count of how many ->nexts we need to follow in the linked list or it's the index in the array that we're visiting. Of course, we could go in there and swap those out to pointers that point to the nodes in question, which might work in other scenarios. (You should notice that as we progress through this book, the vagaries are increasing. This is because you should be more and more aware that there are no right answers, just solutions that apply in certain contexts. You should try to supply the context yourself as you read these).

Now, we could either do

```
struct graph
{
     int currNodes;
     int currEdges;
     struct edge* edgearr;
     struct node* nodearr;
};
```

or some variation thereof. Or we could go with the previous solution:

```
void main (void)
{
     struct edge* edgearr = malloc(MAX_EDGES * sizeof(struct edge)); // 1
     int numEdges = 0;

     edgearr[0] = (struct edge){0, 1}; // 2
     edgearr[1] = (struct edge){0, 2};
     edgearr[2] = (struct edge){1, 0};
     edgearr[3] = (struct edge){1, 1};
     edgearr[4] = (struct edge){1, 2};
     edgearr[5] = (struct edge){2, 3};
     edgearr[6] = (struct edge){3, 4};
     edgearr[7] = (struct edge){3, 5};
     edgearr[8] = (struct edge){3, 6};
     edgearr[9] = (struct edge){3, 7};
     numEdges = 10;

     for (int i = 0; i < numEdges; i++) // 3
     {
          printf("(%d -> %d)\n", edgearr[i].src, edgearr[i].dest);
     }

     free(edgearr); // 4
     return;
}
```

Just a couple more edge cases:

1. We're showing lots of different implementations of the same things. At this lowest level, sometimes we want to entangle the various parts of the structure. Other times, we don't and we just need the common factor of, say, index to link the different bits together. In this case, you can use the graph structure above or this array here.
2. This is so stupid. Don't do this. But, we've mentioned before that an array is just a set of contiguous values in memory. And a struct is just all those values shoved together into a contiguous list. So, we'll just use the syntax for a literal (as in a compound literal) array, and then cast it as the struct. The stored series of 1's and 0's stays the same, but it lets C see that it can put it in the same place as the structure. Playing games with casting is generally a bad idea. You have to cast sometimes, it's why it's there. But stuff like this is just silly and would be more readable with a construction helper function.

    - Then we change the numEdges. This static input of data is pretty standard for proof-of-concept code.

3. It feels silly to write a function that just creates a data structure and then exits. So, most of the time we get these print functions. In case you were curious about why any example code you ever read always seems to have these print bits at the end. It's not because we're obsessed with printing out values, it's just so that we've done something to demonstrate that it worked.
4. Don't forget to free your malloc!

Variation

By this point, you should see that you can create a struct or not create a struct and it's still a data structure. You can create side functions and a library or just do everything from main(). You can use integers, structs, floats, bools, or a number of different data types. You can explicitly create arrays or matrices or arrays of arrays, and you can interact with them via pointer or [] syntax. What's the common denominator? The most effective solution is going to change based on the specific context of the problem you're solving.

Also, fun fact, the graphs in particular don't even care of there's a node at this point. We can actually perform a bunch of our proofs just by demonstrating certain configurations of the edges. But that's a different book.

Search

Another reason we use search is like the aforementioned print command. Searching is a very common task in computer science. But it's also one of the topics that can be understand as applicable regardless of the how generic the solution is. So, we need to ask the question: how do we find a specific node within a graph like this?

Side Note: You need to ask first whether you need to find a specific node. It's not a hard question to answer, as it's normally immediately apparent. But, frankly, sometimes all you need to do is follow certain patterns of traversal and finding a specific node is not important. For example, we could model a giant maze as a graph, each turning point in the maze linked with it's left, right, straight and/or back options. We don't need to find a specific node, we just need to keep going until we're out.

This approach will work for all graphs. That includes trees and linked lists (but it's silly to apply it to linked lists). There are two variations: Breadth First Search (BFS) and Depth First Search (DFS). The difference is easy to comprehend in a tree. In Breadth First Search, you check every node in a layer from left to right first before you move to the next layer. In Depth First Search, you check every node starting with the root, then the leftermost branch, then the rightermost branch (although the specific order doesn't specifically matter in practice, just that the current node is quick and easy and cuts off extra calculation since you're already there).

This is obviously catered to two different variations: depth first search is easier with pointers but not impossible with an array implementation and breadth first search is easier with the array implementation (try it, you literally just search it from i = 0 to i = n!) but not impossibile with a pointer structure.

In general, we could also look at it like this:

```
define Breadth First Search
     while queue is not empty
          i = pop(queue)
          if i is the search target
               return i
          endif
          for each child of i
               push(queue, child)
          endfor
     endwhile
     return NotFound
end

define Depth First Search
     if the current node is equal to the search target
          return the current node
     for each child of curent node
          if searching the child returns a valid node
               return valid node.
          endif
     endfor
     return NotFound
end
```

Which one to use is a matter of (you should know what comes next) the context of the problem you're solving. You should work to anticipate where the most likely place for any searched for node is. If the node is likely to be anywhere and you're using an array structure, do a breadth first search (the array shortcuts the psuedocode) it's going to have less overhead and you don't know where it is. Similarly, if you're using the pointer approach and you don't know where it's going to be, you may as well use the depth first search. On the flip side, if for some reason you're more likely to search for newer entrants or higher entrants or whatever makes a leaf node a leaf node, or a root node a root node in your structure, that's a good approach.

Alternately, sometimes you can narrow the data down. If you have any clues about the data, there's no reason to search the whole tree. All of this certainly presumes you're searching non-sorted or non-sortable

data. But, for example, a prediction chain is going to have some ideas of where to look. for example, if you've got a tree that has weighted branches, certainly go down the higher weights first (unless you're looking for the least likely). Regardless, if you notice how convoluted the sentences get when we try to cater to all bases, this is why the real answer is always just the simplest: it depends.

We have a problem, though. while this works in trees, there's one slight option that exists in graphs that trees specifically avoid.

Cycles

What's a cycle? We've presented some of them, but essentially a cycle is any traversal path that revisits an already visitied node. In the tree, we can only ever go down, so there's no cycle. But even in our consistent exaple from earlier, there are two cycles right there: A->B->A, or B->B. This is a problem that is hugely consequential because there's no way for us to know at the outset that there is going to be a cycle. We won't know there's a cycle until we're in the infinite loop. There's only really one strong solution to this:the breadcrumb trail.

Basically, we create a list of every node we've already visited. Before we visit a new node, we check to see if it's on the list. If it is, we don't search that node again. What we find is that no matter how tightly interwoven the graph is, it will slowly straighten out and act like a tree structure. And thus, we can treat it exactly the way we did with the tree. Nice!

Search Code

Let's revisit our Matrix Adjacency code again. We'll add a global nodearr[] array as a global variable and return an int to the located portion.

```
int DFS (bool* adjmtx, int current, struct node* target, bool* visited) // 1
{
     if (visited[current] == true) return -1; // 2
     visited[current] = true;
     if (equals(&nodearr[current], target))
          return current;

     for (int i = 0; i < MAX_NODE; i++) // 3
     {
          if (adjmtx[current*MAX_NODE + i] == true)
          {
               int ret = DFS(search, i, target, visited); // 4
```

```
                if (ret > -1) return ret;
            }
        }

        return -1;
}

int BFSTree (bool* adjmtx, struct node* target)
{
        for (int i =0; i < MAX_NODE; i++)
            if (equals(&nodearr[i], target)
                return i; // 5

        return -1;
}

int BFSHardMode (bool* adjmtx, struct node* target)
{
        bool visited[MAX_NODE] = { false }; // 6
        int queue[MAX_NODE]; // 7

        int head = 0;
        int tail = 1;

        while (head < tail) // 7
        {
            int current = queue[front];
            head += 1;
            if (equals(&nodearr[current], target))
                return current;

            for (int i = 0; i < MAX_NODE; i++)
            {
                if (adjmtx[current * MAX_NODE + i] == true && visited[i] == fa
                {
                    queue[tail] = i;
                    tail += 1;
```

```
                    visited[i] = true;
                    // 9
                }
            }
        }

        return -1;
}
```

Let's get into this code a little deeper:

1. This is a lot of arguments. Low level code can do that, but it's mostly just the matrix of edges, the current node for recursive purposes, what we're searching for, and then we need to add the visited array to check for cycles. You should notice that if we just remove the visited argument and the first two lines, we get a suitable DFS function for a tree (or any graph we can guarantee doesn't have any cycles).
2. The very first thing is, have we been to this node before? If so, we know that we won't cover new ground by following the edges that lead out from it. The second thing is to mark down that we've now visited this node.
3. The loops are pretty simple. Since we only want to check the edges of one node at a time, we just need to check one row of the array.
4. We need to check for each of the nodes and conceptually most of them won't have the node we're looking for so they'll return a -1. Notice, though, that this is the most basic level of cycle checking. There are more complicated versions. Why would we want more complicated versions? Mostly for weighted graphs where it's not whether we can find something but finding the shortest path to it. But that's another topic.
5. Cheeky. We just wanted to show that trees are super simple versions of graphs. This function is almost fun to write!
6. Same thing, only because we're not recursing in this case, we don't need to worry about passing the visited array. Putting a value in {} like this and assigning it to an array basically sets all of the values of the array to that value. This is what makes an experienced coder get higher pay than a less experienced coder: constantly finding these small little nuances and tidbits of coding knowledge and being able to apply it.
7. Did you notice the queue? We could actually just call on the queue functions we developed back in chapter 6. But you should have noticed that the theme of this chapter is simplicity. Most of these algorithms and data structures aren't intended to be big giant things you memorize. They're processes that you intuitively learn. For example, you didn't memorize how to throw a football, you just practiced a bunch of times and adjusted variations in hand placement until you improved it. (Even if you're don't feel athletic, if you've thrown a football twice, you've improved at least once in your ability to throw a football). In the same way, just a couple of lines of code are enough to put some of these tools into any given framework.

8. We try to avoid compound conditionals. But we also try to avoid deeply nested control structures. This was an unfortunate confluence of both. Just notice that we're checking if there's a link to the node at index i and also if we've visited the node at index i. If neither is true, we go through. Notice that this one calls before we do anything, but the BFS calls after the function is called. No strong rationale, just demonstrating that both are valid. Can you write code to swap and have BFS check after doing something and DFS check before?
9. Notice that we aren't actually doing the comparison here, we're not actually engagin in any checks or whatnot. Instead, we're just pushing the target values into the queue and managing the queue.

BFS and DFS are important concepts. It's almost always important in any graph or tree to test both search versions to see which one gives better results. There'll almost always be a different, but whether it's repeatable or consistent is what you're looking for.

Lastly, we call this 'Search', but it's not really about search. What if we wanted to visit every node of the graph and do some sort of processing on that node? These two algorithms are also good processes of making sure that we can hit each node once and exactly once.

Practical Applications

We've mentioned weights and edges a number of times here. We've even bounced around a very famous problem. It's called the Travelling Salesman problem. It goes like this: Imagine you're a salesman and you have to fly to 5 different cities by the end of the summer so that you can visit a potential client in each one. What order you go doesn't matter, what time you get there doesn't matter. All that matters is that your company's travel booking platform has a fixed cost to visit each city regardless of time or anything else. And, if your travel budget is under a certain amount by the end of the summer, you get to keep the leftovers! So, the question is, how do you visit all 5 cities for as cheap as possible?

The solution is simple: Just try to exhaustively list all of the options. This is the brute force option, and it's incredibly effective! You will absolutely find the best answer! In this case, you've got 5 cities, right? You can try pattern 1-2-3-4-5, 1-2-3-5-4, 1-2-4-3-5, 1-2-4-5-3, etc. Every single variation. that's 120 patterns, which is definitely worth the time it takes for the budget prize at the end. But, suppose that you were supposed to go to 6 cities? That's suddenly 720 routes. 7 cities? 5040. At just 10 cities you have to check 3.6 million combinations. Babymetal's 2025 US tour covered 24 cities! Their poor manager must have calculated 620 sextillion variations! That escalated quickly!

The Travelling Salesman is a notorious problem because it's not just a hard problem to solve, in its class, it's considered the provably hardest. So much so that if anyone came up with an optimum solution, it would literally revolutionize the entire of Computer Science and the derivations of that would invalidate almost everything in this book. Luckily for a certain pair of authors, nobody plans on that happening.

What's more important here is that we don't consider the Travelling Salesman problem difficult. We consider it impossible from a computational standard. It's classified as O(n!) or N-Factorial time (we can theoretically solve it in 'only' exponential time using backtracking, but it doesn't change our conclusions). And that 24 cities optimal route is 24!, or $6.2*10^{23}$. Fun fact, their 2024 global tour was 51 cities, and that's $1.5*10^{66}$ different possible routes. For comparison, if we could calculate 4 billion routes per second (and the time it takes to calculate each route gets harder as the journey size increases), the 24 city tour would take around 4.8 million years, or ~40 times as long as humans have been a going concern. The 51 city tour would multiple times longer than from now until the heat death of the universe.

And the point is that we don't even care about those questions. Google needs to know how to get their street-mapping van to every single one of the 19,495 incorporated cities in the US. Go ahead and try to find a calculator to calculate 19495!. It's about 10^{75168}, which is pretty big and there's no way that Google can get their surveillance van to every little city in any–

But wait a second, Google does get to every little city. That's why we can use streetview almost everywhere now. So if the calculations are impossible, how do they calculate it? They don't calculate the factorial of 20,000. That's ludicrous. Instead, they redesign the problem. They don't need 1 vehicle to go to 20,000 places. They need 20,000 vehicles to go one place. Now they've reduced the problem from a traveling salesman problem to a tautological one. There are thousands of intersections in the city, how do they get to all of them most optimally? They don't. They find a good solution, maybe using a greedy algorithm or a best first algorithm. Divide and Conquer isn't ideal for this, but maybe they split the problem into smaller subproblems. Then they iteratively improve their method for problem solving. This is a case where the perfect is the enemy of the good and if they wait for a perfect solution they won't get anything done.

A real life example is package delivery routing algorithms. One of the things that has been a quiet revolution in mapping utilities is that most urban routing apps have started to optimize for right turns. In an urban environment, when there's lots of starts and stops and turns anyways, from a pure graph theory perspective, it turns out that most good-not-best solutions were approximately equivalent. Until someone pointed out that a right turn literally takes less distance than a left turn in an intersection and if there are that many turns, it turned out to be substantive in aggregate.

The answer is constraints. And that's the real practical application here. There are two kinds of answers:

- The Best Answer
- a good answer

What's the difference? Sometimes we can prove that a good answer is no worse than, say, 20% less efficient than a theoretically optimal answer. But calculating the theoretically optimal answer will take a million times as long. So the best answer is no longer the best answer because the money you saved by finding the optimal route got wasted on electricity to pay for finding the optimal answer! Again, it's always about tradeoffs.

The art of properly constraining a problem is one of the most useful skills in computer science. When you

heard a client or a stakeholder tell you about what needs to be done, start to ask questions about what would be good enough. Because as often as not, the client thinks they need more accuracy than they really do. For example, suppose that a client comes in and wants a tank designed for them to the highest and most precise specifications. But when those specifications are delivered, it turns out almost no manufacturing facility can meet those high quality (the German Tiger I tank, for example).

We'll come back to this idea. But one of the most practical applications in all of computer science is: Contain the impossible until it becomes performable.

AI Example

We've tried to simplify problems as much as possible in this chapter. So let's close by making them more complicated. One of the types of data that is almost ubiqitous is time series data. This can take a lot of forms, but the most common is just a date range. One of the most common applications of this is what you'd expect: first, last, average. But we also see a lot of counting. For example, how many incidences occurred in a given time period. Or, even harder, 'what is the optimal window of time to select in an analysis on how many incidences occured in a time period?' That latter is the same complexity problem as the Traveling Salesman is.

Regardless, there are a lot of data analyses that occur and we need to have good ways of storing data. So let's review five ways we can store the time series data so that it's easy for different types of analysis that feed into ML or Analytics. Sometimes AI, but only as one more data source, not something it's particularly known for.

1. An Append Only Array, or more likely a circular buffer – Some of these data entry points come in fast enough that they push the rate at which data can be written to drives. For example, high end HD video might be writing frames so fast that even the comparatively minor metadata is hard pressed to keep up. Or network stacks that are logging every piece of data that fits certain criteria with a time stamp. In this data structure, there's a 'head' pointer (think read/write head on a tape deck or video scrubber) that just moves along to the next available slot in the array. Data comes in, it gets written and the head moves to the next. When it gets to the end, it either stops or it circles back around to the beginning. It just depends on whether it's intended to be a buffer or a long term storage Either way, because there's less calculation involved and the head just moves, it's a very optimal way of writing very fast data that's coming in and time series stuff can be very fast in some applications.
2. Columnar Storage – We actually explore this a little bit in this chapter. Typically we think of data structures as being monolithic. All of the data on an entity is recorded in one location and then the next entity gets the next location, etc. But in columnar format, all of the data of a given attribute is stored in sequence and then the next attribute in sequence, so on. So rather than all of the demographic data about 'Mr. Smith' being the primary organization method, it's all of the age-related data of all of the sample set of whom Mr. Smith is a member, and then it's all of the birth dates, and then it's all of the dates of their

last doctor's visit, etc. Columnar data makes displaying data about individuals much slower than row-order. But many applications don't care about individual records, they only care about aggregate data, which this is much faster for.

3. Segment/Interval Trees. One of the sorting methods we didn't get into was the 'bucket sort', which isn't a sort so much as a constraint making things faster. Basically, in bucket sort, you decide that you don't need everything sorted, you just need different groups sorted. So, for example, you might dump everyone who responded with different levels of college education into different 'buckets' of data, and then the sort within that bucket is much faster. Interval Trees are similar: rather than a strict order, each child of a given root is set up to separate out the data into different gradations. So, for example, if you only wanted certain time series data but couldn't erase the rest. Or, perhaps you wanted to sort records based on whether they were recent or ancient. Or maybe you had a pile of HR complains and you needed them sorted by which boss was in charge. If you had enough complaints, it might warrant needing to use a constrained sort like this.
4. Log-Structured Merge Tree (LSM Tree) is frequently overcomplicated, but it's essentially a paging system. Rather than worrying about putting the data down on disk, you set up a series of triage points. You collect the data like a small Append Only Array in a high speed cache, and then when it's full enough, you dump it down to a slower but faster cache. But you insert it like it's the actual data structure you ultimately plan to use. And then when that one gets full, you dump to a slower tier of memory, and then eventually you get to the writing to disk part, which is slow. But, hopefully the number of times that you polished the structure and made sure that everything was in order and the fact that it's a big bulk data transfer was enough to make the disk write time optimal and less than if the data were being written in real time.
5. The easiest and arguably most effective approach is less about speed of writing and more about speed of reading. How precise does your time keeping need to be? You can save a lot of space by cutting off extra precision. For example, most time stamps measure down to the millisecond. But if you're looking at data that can only happen every few seconds, does it matter? Or, for example, if you're talking about someone's age, it's frequently the number of milliseconds since Jan 1, 1970 that they were born minus the number of milliseconds since Jan 1, 1970 of today. Nobody measures age in milliseconds. So just cut it out. Same for measurements that include time when only the day matters. By reducing the number of bits in the data representation, you're increasing the speed that the analysis can occur.

    - The other side is valid too! You can take a frame from video encoding. Only write the number of seconds since the last minute mark. Or if you know that all of the data that you're encoding is going to be very tightly groups, just set a central value and then measure everything from that central value. That way you get much higher precision from much smaller amounts of raw data.

Media Attributions

- graph_illustration

CHAPTER 10 - MATH IS MORE ART THAN MATH, MORTY!

Welcome to chapter 1! Well, sort of. See, the idea of 'practical data structures' or 'practical algorithms' is a bit of a misnomer in most modern academic contexts. Here's the best example: why are there stances in Martial Arts? That sounds like a silly question, the answer is: 'to teach you how you might want to stand'. But, if you're in a fight, how effective will it be to just stand there, no matter how interesting or obscure your stance is? Shouldn't you be moving around and giving yourself the advantage of motility?

Here's a fun test: ask this question to any dojo you might be interested in joining. Except for BJJ. They'll just look at you like a crazy person for asking questions and even suggesting that they have stances. For everyone else, their answer should be something along the following.

We don't learn stances because we ever expect to stand in them. Instead, they constitute the transitional states through which we move. By learning a stance, we get our bodies comfortable in the momentary places we've identified in our style as being convenient to standardize. It's like putting your fingers on the 'home keys' of the keyboard: you don't keep them there when you're typing, but by always returning there, you've standardized how your hands should move to continue. In the same way, stances are not meant to be stayed in, they're meant to serve as both a starting point for understanding the movement of The Path, but also as a self-centering mechanism mid-movement.

So, the same thing happens with Data Structures and Algorithms. The previous chapter was the closest to what you should actually be using in the real world. Just make the general context fit the application you're making. But, always, you should be moving and adapting to the specifics of your current data environment, not trying to adjust your data environment to fit the rote data structures you've learned or read about.

This is what made this textbook worth writing and the value you should be getting out of it. We started with the practical 'do it like this' and then moved to 'modify it like this'. And only now are we going to start with the abstract theory that answers 'it's like that because...' whereas most texts start with the abstract theory and then jumps into the practice. Hopefully, you'll find that because we've been using and conversing with an increasing presumption that you can just use this theory, you'll quickly assimilate the new rationale. Ideally, we've set up a situation where your brain is now hungry for the things that connect the terminology with the meaning behind the terminology.

And, this may be a controversial claim, but don't worry too much about fully understanding the theory. That's really the domain of the academics and the experts who've gotten good enough at the day-to-day programming that they have the time to think about the deeper meanings. The vast majority of the workforce in development has a reliable understanding of the parts of the theory that struck them most, but nobody expects them to grasp the entirety all the time.

So, without further ado... Theory time!

Constraints

Computer Science is great, because it has essentially revolutionized every other industry. When we look at projections and how we look at the future, best practice tends to be to give three options: best outcome, worst outcome, and expected outcomes. It might be completely subjective, but surely that's because Computer Science (and the mathematicians before us) made such strides in effective arguments based on their experiences.

So, as you might expect, when we measure the effectiveness of an algorithm or a data structure, we're always going to be looking at how it performs in the best case scenario, the worst case scenario, and the average case scenario. These all have special names. Don't get too hung up on the names or which one is which. That is easy to look up, it's enough to know that there are differences:

Why then is the section titled 'Constraints'? Because it's not really about the performance of the algorithm in any given instance. Instead we're proving certain constraints that exist in the metrics that we look at for the execution of the algorithm. It's always the dynamicism of the algorithm as it completes that's of concern, never just the static data structure (with a couple of caveats). It is incorrect to say 'best case scenario'. It is much more accurate to say that 'the best case scenario will perform at or better than the Theta bound'. One problem with explaining this is that it just doesn't occur in the real world very often. In the real world, variation is such that we can't talk about organism growth or whatnot in scales that we see regularly.

All of that leads back around to the idea that while it's incorrect to consider the bounds to be the best, worst, and average case scenarios, it's much more visceral to think in those terms. So remember the little coda that these scenarios are the scenario in which the bounds will most tightly align to. But otherwise, use the internal monologue that makes the most sense to you. We are going to get pedantic and picky for a moment, though.

These three bounds are as follows:

- O (Latin Letter O, for Order) – This represents the upper bound of a given algorithm.
- Ω (Greek Letter Omega, for last) – This represents the lower bound of a growth function.
- Θ (Greek Letter Theta, for normative) – This represents the tight bounding of a growth function.

This is all coupled with the fact that we can do the analysis as either 'big' or 'little'. This part is the fun part. Suppose you buy a bottle of shampoo that's one of those travel ones. How much would you obsess about getting the last drops out versus the last drops out of a giant refill canister? That's kind of the gist of Big-O, but to get to why it's an issue, we need to contextualize all of these bounds.

Scaling

Nobody really cares bout the solution to any one problem. For example, sorting a list with 50 elements and bubble sort takes 2500 swaps and comparisons. At the time of writing, one of the slowest processors still being made was the Texas Instruments MSP430 at 32kHz, which would accomplish that task in possibly less than a sixth of a second. The point is that, broadly, the questions that we can visualize are kind of trivial. What about a thousand elements? That will take a thousand seconds and be done on most processors in less than a microsecond. A million elements? Now we're talking! Sorting that might take almost a quarter of a second!

And here's the problem. Try to visualize a million things. You know what a dollar or a Pokemon card or a soda can looks like. Now imagine 10 of them in a row. You need to be able to visualize each one. Now make ten rows total for a square. Now stack those squares for a hundred. Can you still see each individual can and all 100 individual cans at the same time? Likely not. Humans don't scale with large numbers well, empirically.

So, when we talk about how an algorithm does, what we want to know is how it performs when it's really big. Now, that "really" is what we call an imprecise term. So, let's make it more specific, we want to know what happens when the input to an algorithm is arbitrarily big. That is, we can predict how long the algorithm will take no matter what input is in there. In order to do this, we model the ratio with an input of `n` and an output of how long it will take on a graph. That graph is... complicated.

Bounds

Think about how music works. A 4/4 time signature is fairly straight-forward. And a 5/4 time signature is complicated. But a 6/8 time signature is easier again. But then a 7/8 is incredibly complicated, yet a 16/16 time signature is handleable. But, that doesn't change the fact that if we had the number 1645 in the time signature, we'd definitely know it was super complicated. And we'd also know that it was nowhere near as complicated as 78334. The specifics of the intricacy don't change our understanding of the general trend that the more beats per measure the more complicated the music will be.

The same thing happens with the algorithms. As the input increases, we know that the complexity will increase. We don't care about the minutiae of whether any one sized input is strictly more complicated than an input that's one bigger (or less complicated than one smaller). We want to know how an input of 10 compares to an input of 1,000 compares to an input of 100,000,000 compares to... any arbitrarily large input.

That's actually an impossible question to answer. It's like a king summoning a prophet and saying 'I had a dream, I know now what. Interpret it for me.' So instead of trying to being specific, we start asking what questions we are able to answer. And, in this case, we can't answer how fast it will perform in an arbitrary environment, but we can create a function and say this is the bound of the performance. So, for example, at 999,948 input, the time it takes might be 1,999,896 but at 999,948949 the time might be 4. But, can

definitely guarantee that it will be less than 2 * the input. WE can't give more information than that, but we can guarantee that.

Upper Bound

When we define the upper bound, we're defining a rate at which the actual execution time will grow less than. That's a fancy way of saying that the worst case scenario in every situation will always be less than the upper bound. why can't we specifically estimate? Because for an input of any given size, the actual structure can be totally different. We've already seen how a tree can become incredibly lopsided based on the input order. Or that the different sort functions perform differently based on how ordered the data was originally. Since we've just said [an arbitrary input of] an arbitrary size, there are too many unknowns.

So, that's what O analysis is. We're examining what order (as in tier or level) the upper bound of an algorithm is. This does not mean that the algorithm will perform in that time. It means that we are mathematically guaranteeing that the algorithm will operate in less than the time indicated by the Order. This is where we get Big O from, but we're not there yet.

The colloquial understanding of the Upper Bound is that it's the worst case scenario. Distinguishing between the two is incredibly pedantic and the actual distinction is sort of like when someone says 'You can't say 'to boldly go', because that's splitting the infinitive and you can't split the infinitive'. And someone else goes 'I choose to rebelliously split the infinitive' and what people are missing is that there is a historical precedent that the original linguists established rules for how English works and presumed that English should work like Latin where it was impossible to split the infinitive because the infinitive was a single word, so even though English has a two-word infinitive, they insisted that it was gauche. And society, writers, modern linguists all promptly ignored them and used words in the way that's most useful to them.

It's not quite, but mostly like that. You will be wrong in the absolutely most archaic and academic sense to say O Analysis is the worst-case scenario. In practice, mostly nobody cares and will completely accept that statement at face value.

Lower Bound

Omega is typically the symbol in literature for the last of something since it's the last letter of the Greek alphabet. When we organize something sometimes the last is the least. Therefore when we talk about the least amount of time that an algorithm might take we're talking about Omega Analysis. Omega Analysis is, broadly, the opposite of O Analysis. We're trying to demonstrate a growth curve in which every input of every given size will perform faster than.

One way to look at it is that, by definition, all functions are faster than 0 steps. So all algorithms are, by definition $\Omega(0)$. And all functions are faster than 1, or take at least 1 step, if just to return their success state.

So all functions are faster than *O(1)*. But what about linear? When we search an unsorted list, we need to examine every element in the list, right? So, as the size of the list grows, the size of the number of checks we'll need to make grows, and the value of *f(n)* where *f(n) = n* grows. (But wait, you say, what if we find the value halfway through, then it's only *n / 2*. There's only one appropriate response to that, and similar to Brave Sir Robin, we'll run away and come back another time).

Tight Bound

Theta is kind of the symbol for 'normal stuff'. Like, we use Θ when we have polar coordinates, because we can only have an input of of so large (the circumference of a circle). So, we use it when we want to define the tight bounding of the algorithm. Not every algorithm will have a tight bounding. But, when we say that the Theta Analysis of an Algorithm is n^2, we're saying that the absolute fastest it will run at any given input size will be of the growth rate $f(n) = n^2$. And the worst case scenario is $f(n) = n^2$. And the average case is $f(n) = n^2$. We're not just analyzing the 'tight bound', we're specifically stating also that there is a tight bound, and it's some growth function.

Term Size

Let's go back to the bottle of shampoo from a little while ago. Why don't you care about the last few drops in a giant bottle of shampoo but you do care in a small bottle? (Assuming you care, please just assume this is a vital matter to you. Like, you regularly #shampooislife) It's a matter of scale. As the size of the shampoo bottle increases, the relative utility of each drop of shampoo decreases. So, just like humans have a hard time scaling up numbers, we have a hard time scaling down proportions. We either over or under presume the relative importance of small fractions of a whole (this is similar, but only tangentially related to how a toddler will always tell you that the tall 8oz glass holds more liquid than a short and thick 8oz glass).

But, there's a reason why we do that! However, it's complicated, theoretical, and way outside the scope of this textbook. We can demonstrate a similar derivative problem to that issue, though. Imagine we have a function $f(n) = n^2 + n$. Now, suppose n = 1. $f(n) = 2$, right? But how much of that 2 comes from n^2 and how much comes from n? 1 and 1, so a 50/50 split. Suppose that $n = 10$. Then $f(n) = 110$. And now we're at a 91/9 split. And when n = 100? 9901/99 split. What about when n is arbitrarily large? Well, we don't know, but we can very easily see that as n grows infinitely, the ratio between what parts of $f(n)$ come from n^2 grows closer to 1 and what come from n grows closer to 0. Reductively, we substitute 'arbitrarily large' for 'when n grows infinitely' and we get a general idea of what we're looking at. The result is that for any random n, in a lot of ways, it's only true for us to give the same answer that came when n was infinitely large. So, even though the growth rate is technically $f(n) = n^2 + n$, because at an infinitely high n, $f(n)$ is also equal to n^2, we just say

that the entire algorithm grows slower than $f(n) = n^2$, because that is true-ish when n = infinite, we say that's the growth rate overall, because that's what it will eventually be.

Phew, that was a long and meandering paragraph. In short, for any function, if we take all of the terms (a term is the bit separated by a plus/minus and not by a multiply/divide (or any higher order math like an exponent or root or log or factorial), we only tend to respond with the biggest of those terms. We can just say because the smaller ones become less consequential as the values increase.

Big versus Little

That previous section wasn't strictly true. We mentioned that each analysis can have a big version and a little version. It's a super small, incredibly picky distinction. But, technically the Big version of each analysis method (Big-O, Big-Ω, Big-Θ) is looking for the growth rate where the bounds are less than or equal to (or greater than or equal to). We call this an inclusive bound. So, for example, $n^2 + n = n^2$ as n becomes infinitely large. Thus, it's true that a function whose growth rate would otherwise have an upper bound of $n^2 + n$ can be said to have a Big-O of n^2. Not perfectly, but broadly, we only drop the small terms for the Big versions of the bounding analyses.

The Little version of each analysis method (Little-O, Little-Ω, and Little-Θ), however, is the opposite, it's only strictly less than. Remember that $n^2 + n = n^2$. It is not true that $n^2 + n < n^2$. Therefore when we do the Little-*O* analysis, we have to say $n^2 + n$. We don't get to drop the smaller term.

What about constants and coefficients? Same thing, as n grows infinitely large, $3n^2 = n^2$. Because with the ratio of the growth rate between what the 3 contributes versus what the n^2 contributes, the 3 is always the same, but the n^2 is growing quadratically. However, with Little-O notation, the 3 is still important, because it's still a strictly exclusive bound. It has to be strictly greater or less than.

In Practice

Notice that we've used Big-O notation a lot in this text. And this is the first you've heard of Little-O, Big-Θ, Little-Θ, Big-Ω, and Little-Ω. There's a reason for that. We rarely see the latter used in practice. There's two reasons for that. Big-O is easy to remember, but remembering 3 or 6 metrics for each algorithm would be tricksy.

But there's a more practical reason. when we're writing an algorithm, if it runs too fast we're excited and maybe add a few more features. But if it runs too slow, we need to make it run faster. Alternately, we might say that the practical utility of all of these algorithms is never going to be based on what the lower-bound of their execution is or what the normal case is, it's almost always going to be the upper bound. That is, we need to make sure we've got enough head room for the slowest that the algorithm will operate. If we just always assume it will go the fastest it can go, our application will run shoddily.

Similarly, when we're considering, we want to typically know the Big-O, because it's just a single term and that's easy to run and compare. We get a general, broad spectrum understanding of how the algorithm performs without having to engage with the intricate details. This is typically sufficient, because a lot of the details will change with actual implementation, based on the details of the environment we're implementing in. For example, if we implement an algorithm in java, the amount of memory allocated to the Java Virtual Machine might be more important than the time it takes to complete the algorithm, so focusing on the intricacies on the theoretical general case would have been non-optimal.

One last distinction

For the most part, we stick with Big-*O* analysis, because that's what most people are practically concerned with and we'll continue to do so. Broadly and colloquially, we'll say that Big *O* is, essentially, the worst case of the algorithm. That's not correct, it's more correct to say that Big *O* is slower than than the slowest that the algorithm will perform. We can use the term 'worst-case', but only colloquially and with the recognition that the exact details are more complex.

Why? Because we can actually talk about the worst case operation of a function! Suppose we talk bubble-sort. But then suppose we talk bubble sort over reverse sorted inputs. It's the same set of steps, but the problem domain is different so the entire analysis is different. So we can talk about:

- Big *O* of Bubble Sort = $O(n^2)$.
  - Little *O* is $o(n^2/2 – n/2)$., which we might write as $o(n^2)$
- Big *O* of the worst case scenario (reverse sorted) of Bubble Sort = $O(n^2)$.
  - Little *O* is $o(n^2/2 – n/2)$ which we might write as $o(n^2)$
- Big *O* of the average case scenario (uniformly distributed random values) of Bubble Sort = $O(n^2)$.
  - Little *O* is $o(n^2/4 + n)$, which we could say is not $o(n^2)$, but $o(n^2+\varepsilon)$ would be theoretically accurate. Meaningless, but accurate..
- Big *O* of the best case scenario (already sorted) of Bubble Sort = $O(1)$.
  - Little *O* is $o(1/n)$, which approaches 0 as n increases, and it's weird, but we could theoretically say $o(1)$.

Each of these different contextualizations is actually a different problem and big-O applies in there. But now we've broken it down into so many sub parts that it becomes hard to convey in practical environments (but excellent for an academic paper!). In fact, all of these variations are ways that we use to get a better handle on how the algorithm will function. Ideally, little theta, where we can tightly bound in an exclusive bounding is kind of the most optimal setup. It means that we can know exactly how much computing resource to assign to any given task based on size. But that situation happens rarely and real-world messiness often gets in the way of abstract perfection.

The idea here is that you've gotten just a taste of the computational complexity measuring. Broadly, what we want to do is just use smaller algorithms. The size of the algorithm is important. $O(n^3)$ is a bad algorithm, but sometimes it's as good as we can do. $O(n^2)$ tends to be fairly normal if we're doing anything real with a set of data where everything relates to everything else. Similarly, we might have $O(n*m)$ data when we have two data sets that interact with each other in a many-to-many relation. $O(n)$ tends to just be for output, with no processing. $O(nlog(n))$ or $O(log(n))$ is kind of the gold standard: it's when we know that we've got the theory down about as good as it's going to get for all practical purposes.

After that, the specifics will depend on the details of the implementation. That's the part where breaking out of a function when we find out there's no chance that we'll find an answer or we've already found answer, etc. Once we know the Big-O, in other words, we need to work on optimizing the Little-O. But, realistically, nobody optimizes for Little-O, because every moment spent contemplating the algorithm in the abstract is a moment that could be spent programming it in a very real environment. Again, it takes a theoretical role for anything more than Big-O to be more than tangentially necessary.

Speaking of which, notice that the Little O's are messy up in the previous list? And that we can simplify it to an easier form, but that reduces its utility. In short, don't worry about Little O, just look it up when someone else mentions it. You can always compare like with like for analyses. So, for example, you don't need to know the details of what little-theta is, exactly, to know that $\theta(n^{2.543} + 4n + 2)$ is faster than $\theta(n^{2.539} + 4n + 8)$.

Wrap-Up

Big-O notation means we keep only the dominant term of a growth function, dropping smaller-order terms as well as constant factors and coefficients. It's good to know that the discarded terms exist, but unless you're going to write and formally analyze algorithms for a living, it's better to come back to the finer details after you've mastered the basics. For now, focus on understanding Big-O notation, learning how to estimate it, and eventually proving the Big-O of a given algorithm.

The informal nature of the notation is why you'll sometimes hear people say things like, *"Well, it's still O(n log n), but it's a bigger O(n log n) than the other algorithm."* Strictly speaking, that statement isn't mathematically correct both algorithms belong to the same asymptotic complexity class. What the speaker usually means is that one algorithm has a larger constant factor or performs more work per iteration, making it slower in practice despite having the same Big-O complexity.

Complexity Groups

This is the single nerdiest thing that everyone should remember and be able to explain later, just for the sheer abstract joy of it. We're going to cover the equivalent of the philosopher's stone, but Computer Science instead of alchemy.

Every single math problem you can ever imagine is all smooshed together into a giant conceptual space. When we talk computer science problems at this layer, we're broadly talking about complicated math problems. They don't always translate well, but they're still math problems at this point. Don't panic about the math! We're just going to talk about them in conceptual details, not too much calculation. It's important you know why things works at this point, the how and the mechanics can come later if you choose.

Also, this is a fun bit of nerd-out, so we're going to cover a bit more than is strictly necessary. For the lulz.

Time, Space, and Turing

We've talked about time and space tradeoffs before. What do we mean? Let's talk Turing for a moment. This is a brief aside that would also be covered in an intro primer to the Theory of Computation. There was this guy Alan Turing. It's important to know he existed for a bunch of reasons. He's a hero of Computer Science, his biographical details are relatively interesting and intersect with social issues, The Big Prize in Computer Science is named after him (it's often referred to as the Nobel Prize of CS), and he formalized (or maybe reformalized) a large swathe of Computer Science such that how we think about programming is almost entirely derived from his work. This doesn't mitigate anyone who came before, because he was obviously working off of their material. But, the framing of Computer Science as we know it is all built on his work.

And, formally, he defined what a Turing Machine was. We aren't going to go into too much detail, suffice to say that it's a conceptual model of what a computer is at it's most abstract. It has a read-write head, and that read-write head will perform a set of actions based on directives which are selected between based on its current state and the inputs that it reads off of an infinitely long tape. The key here is that when we drop it down into this fundamental conceptualization, we can really get to the heart of what matters. We aren't worried about electricity use, we realize that electricity use is a derived metric. We don't worry about how long it took for a programmer to work on the code, that's a derived value. The only things that matter at this conceptual level is how long it takes to complete a program (time) and how much data on the infinitely long tape the program uses (space). This is why we can have programs that complete after the heat death of the universe: we don't know the answer, but we know how to program the conceptual machine to keep chugging along.

This isn't really how a computer is made, assembled, works in the real world, etc. It's how we program it, and it's the theory underlying every other aspect of Computer Science. It's why Computer Science as a field is safe regardless of the current state of AI, because even when AGI is developed it will still take a while to get good at these core computational conceptual models.

Ok, so the point is that we only care about two metrics: Time and Space. It's almost like a Doctor Who quote or something. We've talked so many times about the tradeoffs. This is why those are the tradeoffs we always bring up. Sure, there's a tradeoff between acting within a small enough footprint that our program can operate within the L1 cache and not need to get paged out to L2 or that we can operate fast enough that we don't need to worry about garbage collection or OS context switching or something. But you can see that those are just exaggerated ways to say that we're trading out time and space concerns.

But, also, to a certain extent, those are units. We only have so many units to trade away. For example, one of the tradeoffs we made in Merge Sort was to show the smallest space possible, because it's the easiest to understand what's occurring. However, in practice we can make Merge Sort much faster by creating new arrays in new memory for every recursion and then just merging it all together at the end in the original space. It's the same Big-O, there's not a lot of change in the complexity analysis. But, it's just faster because in place we basically read and write and delete every value *O(nlog(n))* times, but when we use new space we just need to read and write and then when we free we just pretend like we never used that space *O(nlog(n))* times. But, think about how much more space we use, every iteration allocates a new sub array. But is it worth it? In short, right now memory is cheap and processors are expensive. Longer-term, businesses will typically ask 'does it need to happen fast?' and we can almost infinitely increase storage space but the amount we can increase processing for a single process has a point of diminishing return. (We should always hope for a paradigm shift based on actual objective improvements, because that improves everyone's capabilities.)

The point is, no matter how much we want to trade space to make the algorithm faster, we can't go beyond a certain point. For example, in Merge Sort, we're already using O(n) space. And the processing is happening in *O(nlog(n))* time. So, we could potentially (this is theoretical, and not what the current best algorithm uses) increase to using $O(n^2log(n))$ space, but any more than that and we'll have to use more time creating that extra space in order for it to work! (Of course, we could presuppose that it gets the huge amount of added space as part of it's input, but even reading that extra data will increase the complexity analysis for the time.) The point is tradeoffs are worth it up to a point. And, if we were going to be sassy, we might even say that there were tradeoffs to choosing different tradeoffs.

The Big Complexity Bullseye

Ok, so imagine a dart board. Which is an impure but kind of apt metaphor. The creative component of coming up with new algorithm ideas is conceptually like throwing a dart at a dart board. You know what you're aiming for, but sometimes you're off. There's a lot of skill and practice that involves getting to where you can hit the dart where you want it and to make the algorithm fit inside the constraints you want. But there's also a lot of luck involved in both, too.

So, from inner to outer, we're going to cover each of these areas:

- Constant Time – Think of the function that prints out a character input by a user. 3 steps: input, store, output. Done. No matter what we do, it can't go faster than a fixed number of steps. We tend to call this O(1). Technically it isn't following Big-O constraints and we should call that one O(3), but nobody does (some people still call it O(c) for constant, no matter what number it is). There are very few algorithms that non-trivially execute in constant time. Examples tend to be 'accessing an element of an array by index' (because that's to oppose finding a node in a linked list or a BST) or we can do some clever things with bitwise math (there was a period when people were enamored of XOR as a way to store two data

types). If you've done any IT, subnet masks are an example of constant time ways of segmenting networks to reduce traffic noise.

- Polynomial Time – Everything from linear time (searching for an element in an array) through to $O(n^{SomeReallyBigNumber})$ fall in here. Conceptually, they are always smaller than exponential or factorial values because over time, exponential and factorial will always grow larger. Without going into the specifics (and there are a ton of exceptions and nuances, but they're mostly specific and one-off situations). The point is that when the largest term in the complexity function is just a constant exponent, we consider the algorithm to be pretty simple. Most algorithms we cover in this book so far have been P problems.
    - But that's not to say we don't try to improve them! In April 2024 a team defined a method that brings the best time for matrix multiplication from $\omega(n^{2.371552})$ down to $\omega(n^{2.371339})$. An AI tool found a method for multiplying 4×4 matrices, specifically, in 47 steps rather than the previous 49 step approach.
- Nonpolynomial Time – This is, mostly, everything bigger than polynomial time. Instead of nested for loops, we're typically looking at algorithms that need us to select from every possible permutation, or checks for every possible decrypting key or so on. In fact, while the reason we call it non-polynomial time because it doesn't have a polynomial term as its largest term, the actual definition is that it's a problem we can't solve in polynomial time but we can check the correctness of in polynomial time. That means, for example, that we don't use too much space to solve the problem in NP time, because the checking algorithm will need to be able to evaluate it in P time. One example of this is finding the best move from any given game state in chess on an *nxn* board. It's solvable, but not in any realtime context.
    - There's a special term in here called NP-Hard. We'll come back to this.

Skip the rest of these if you want.

- Polynomial Space – PSPACE is basically the same as NP time. But it's not exactly the same, there are a few problems that aren't NP time, but are still solvable in PSPACE one example (and these get obscure really fast) is called Quantifiable Boolean Formula. SAT is like it, it basically asks 'given a complex boolean equation, is there a set of inputs that makes it true?', but we can solve SAT in NP Time. However, if we add the condition that we want to also know a specific variation of 'for all values' and 'for some values' of the equations, it becomes PSPACE.
- Non-polynomial Space – NPSPACE doesn't exist. Well, it does, obviously we've set up a pattern, right? We can put the words together. But, and this is math you can get into on your own if you like but it's too much for a cutout here. Savitche's Theory basically says that PSPACE is the same as NPSPACE. Or, that anything that would be in NPSPACE can be mapped to PSPACE using a quadratic (which is a polynomial) amount of space. It's absolutely the sort of breakthrough that makes problems categorically easier.

- Exponential Time – EXPTIME is absolutely real! You just have to believe! But belief is all there is. We have no existing examples of a problem that is solvable in NPSPACE but not in NP time (or, perhaps it's more accurate to say that if it can't be solved in NP time then it falls to EXP time and we don't have a way to prove the space issue). Every single one we've tried turns out to not be in NPSPACE. So, it's not that we don't have EXPTIME, it's just that they require too much space so they don't fall in this category. But, we also don't have a proof that there is no such problem, or that EXPTIME = EXPSPACE.
- Exponential Space – EXPSPACE problems are, essentially, everything weird and hard and huge. So, for example, given an arbitrary game where every move by any player has an exponentially growing number of options, is there a way to define that player 1 has a OTK. This isn't Magic: The Gathering no matter how much we might ban those cheesers who put together the 0-turn win decks. Instead, we're talking about any arbitrary game (incidentally, Magic: The Gathering is almost the perfect example of this). Another example is a constrained Halting problem: does a Turing machine solve a given problem in at most a specific amount of space.
- Double Exponential Space – 2-EXPSPACE problems are not solvable in EXPSPACE but instead, the exponent is itself EXPSPACE, space of $O(2^{2^{\wedge}n})$.
- Elementary Space – ELEMENTARY is an arbitrary number of nested exponents. So, for example, a complex recursive process. Think, for example, elaborating all of the layers of possible Inception in the hit movie Inception, but across every person and every person that's created by the inception of every person.
- Primitive Recursive – this is where we can define that the problem will eventually stop and we can even count the number of recursions.
- Recursive – This is where all we know is that it will definitely stop. Probably. No, it will definitely stop. 100% of the time, we may have to wait for several universes to come and go, but it would definitely stop sooner or later. "Finite time" is a sucker's bet.
- Half Halting – These are problems where when the answer is true, we'll get there eventually, but we won't know if we the answer is not true. For example, the Mandelbrot Set (the fractal pattern you see in High School Math posters) is like this. Those black and colored spots? The outside edge is the ones that 'escape' in a reasonable amount of time. The middle black spot is the stuff that we calculate for a while and then give up on.
- Impossible – The Halting Problem. This is the general question of 'will an arbitrary Turing machine finish for any arbitrary input?' The only way to know would be to run every single input. But one of those problems is, itself, a simulation of the Turing machine.

The general pattern is that C ⊆ P ⊆ NP ⊆ PSPACE ⊆ NPSPACE ⊆ EXPTIME ⊆ EXPSPACE. ⊆ is the symbol that means 'is a subset of'. So, technically it would be true to say that 'all P time problems are also NP time problems'. But nobody would care to do that because it doesn't help anything. It mathematically makes

sense and ties everything together. But conversationally and problem-solving-wise, it is not helpful. You can internally replace the ⊆ with a < and get a colloquially equivalent understanding.

Precision of Terms

One of the things that's hard to write in a section like this clearly is all the hedging and hemming and hawing. There are a lot of exceptions and specific variations that cross to a different problem class. Further, for most of the classes, they aren't provably separate. We're relying on the fact that there is no proof that says they're the same. It's like if it takes you one and a quarter tanks of gas to mow the lawn, you can't prove that it's impossible in less than a tank. But you haven't found it yet despite trying really hard, because it would be nice to not have to refill the tank. You plan on always using a tank and a quarter, but you don't have a firm proof that there's not a better way.

In the same way, mostly we erred on the side of explainability. If you run into a situation where your job is based on being able to differentiate between a problem that can only be solved definitely in an arbitrary case while not using the entire available mass of the universe converted to computronium and you need to make it happen while staying within using the entire available mass of the universe, then you're probably not going to succeed at your job. So, we don't mind taking some liberties in those descriptions for illustrative purposes.

Broadly, there are specific proofs for a few areas. And, generally, the proof looks like 'if P is a subset of NP, are there any problems in NP that we can prove are not solvable in P time'. And there aren't any. So, that's fun, but there are still a ton of problems where we don't know what the solution in P time is. We haven't proved it's not in P time, we just don't know what it is. And what this means is that talking about them in hard terms like 'NP is the set of problems that can't be solved in P time' is technically incorrect but still how most people will conceptualize it.

-Hard and -Complete

There are two special subclasses of problems. It's important to make a distinction first.

That ⊆ symbol means that every problem in P is also in NP. And every problem in PSPACE is also in the half-halting space. What we're describing is akin to the Big-O notation. When we describe the class of the problem, we're saying that it's solvable within that bound. For example, bubble sort. It's solvable within 2-EXPSPACE, because O(n) space is in the arbitrary size of $c^{(e^d)}$ when c is equal to n, d is equal to 1 and e is equal to 1 (n^1^1 = n). It's stupid math problems, but it's important for a fundamental goal of CS. And you may not need to perform that goal, but you need to know that it exists.

-Hard is a special subset of any space (sort of, this is in-practice, not in-definition). It's sort of the opposite of 'no harder than'. When we say that a problem is NP-Hard, for example, we're saying that the problem is at least as hard as the hardest NP problem. So, for example, the halting problem is always an AnyClass-

Hard problem. QBF is NP-Hard even though it's classically an example of PSPACE. Basically, we're saying it's almost certainly not in the easier category. So, the Travelling Salesman is NP-Hard, but while we can solve Merge Sort in NP time (remember that since O(nlog(n)) is smaller than NP, it's within NP time), it's not "really" in the NP time category. There's only a very slight and pedantic difference between saying something is 'NP' and it's 'NP-Hard' in normal conversation. For example, nobody is going to talk about the recursive semi-halting state of array access. It might be technically true, but it's complicated. It would be like explaining to someone how hard it could be to put pants on because using magic to do it is so hard when we don't have magic; it's a nonsense way of approaching a conversation even if the specific statements are technically true.

In short, just keep that -Hard are basically the problems in a class that aren't part of the smaller class inside of it.

-Complete is the set of -Hard problems that every other problem in that space can be mapped to. So, for example, the Travelling Salesman problem takes a little finesse, but we can express it as the Efficient Knapsack packing problem or the Hamiltonian Path problem in graphs. We can, for example, represent someone printing their name to the screen as an overly complicated permutations problem. It would be silly, but we could do so. What that means is that for any NP-Complete problem, because every other problem in the space can be mapped to any NP-Complete problem, finding an efficient algorithm for any NP-Complete problem applies to the whole category.

Here's an excellent example: Travelling Salesman brute force is a factorial time problem. But, Knuth and several others (Djiksta, Naur, etc) all wrote about using backtracking as a recursive method to solve SAT or Travelling Salesman in $O(2^n)$ time instead of O(n!) (write a quick program to give a comparison of 2^n v n! for all n from 1-100 to see how much that improvement works). The point isn't the details, the point is that $2^n <$ n! and now we can solve any NP-Complete problem in $O(2^n)$ time!

This is the philosopher's stone we mentioned. We can't prove that it's not possible to solve the NP-Complete problems in P time. So, someone could come up with an algorithm that solves an NP-complete problem in $O(n^{100})$ time, and that would mean that every NP-complete problem would be solvable in $O(n^{100})$ time. That would be revolutionary, even if it might be a while before we could probably take advantage of it. More practically, we might also find a solution to a problem that occurred in $O(n^{\log(n)})$ time or maybe $O(1.9^n)$ time. They're not polynomial time, but they're still faster than the extant solution. It's sort of like being an alchemist and transmuting lead into silver instead of gold. Maybe not legendary ranked, but certainly epic ranked.

And there are lots of these! Could we solve EXPTIME in NPSPACE? Could we prove that PSPACE is always solvable in NP time (as in, checking the solution is a polynomial time problem)? We have lots of guesses, but no actual answers.

Practical Application

We've talked before about the most effective strategy you can employ. Because this is about computational complexity, there's not a lot to be applied in practice. It's sort of like if the CEO of a company said 'we need to reduce the price of our raw goods or we won't be able to compete with our biggest competitor!' The first person to respond 'but that's hard to do, and I don't waaaaanna!' gets fired. Instead, the solution is to find workarounds.

What these complexity analyses do is give us a few tools to analyze how we can make the workarounds or constraints. So, for example, take a look at SAT versus QBF. SAT is solvable in NP because we just want to know one solution. But QBF allows an unrestrained variation of 'all values' and 'some values'. Or, for example, the EXPSPACE example of the halting problem, but within a certain amount of time. The Halting Problem is unsolvable, but saying 'will it happen within a certain exponential period of time?' is a solvable problem! It's the same reason why we can have pictures of the Mandelbrot set: because the colors come from how long it takes for a given input to 'escape', and the outer edge is a default color when the algorithm gave up after 10 or 50 or 1000 steps.

So, what's the takeaway? Just stop. For any hard problem you're trying to solve, give yourself a time limit. This is both programmatic and problem-solving. But, if you can't solve the problem in a certain reasonable amount of time, just move on to the next problem. If you absolutely need a solution, give what you've got in that reasonable amount of time.

What's an example? What is the square root of 7? You'll stop listing digits after a while. Why? Because at a certain point the precision doesn't matter for any practical task any more. calculating Pi is a non-halting problem. We just keep getting more numbers. But what application needs more than a dozen digits? Consider it, if we had a circle the size of the orbit of the Earth, we could define it down to the precision of a Planck length (the resolution of the universe in one theory) with just 47 digits of Pi. So, we don't ever need the correct answer for Pi. We need the answer '3.14'. Sometimes, '3' is a good enough answer, and a heck of a lot faster.

The point is, one of the best skills to learn in programming and Computer Science is when to just stop because the actual answer is less important than having a answer.

AI Example

Continuing on this example of when is good enough, almost all AI (both the colloquial term and the precise term) is computationally hard. But, we can actually split it out. Most of the conventional algorithms we have are NP problems when it comes to creating or training. But they're only P when it comes to executing the outcome. This is classic NP example: solving the problem in hard, but checking the problem is true is easy.

What does this mean? Emphasize precision in the evaluation of an existing AI (whether it's a statistical model or NLP or an optimization or an expert system), but during the training take as many shortcuts as you

can. These shortcuts sound bad when phrased like that. But, the reality is that when the training process is happening in exponential time, then being able to, say, estimate a log by using sqrt. Or estimate sqrt with just some divisions:

```
float fakeSqrt (float f)
{
      float absf = f & 0x7FFFFFFF;
      if (absf < 100) return absf / 4.0;
      if (absf < 1000) return absf / 8.0;
      if (absf < 10000) return absf / 32,.0;
      if (absf < 100000) return absf / 100.0;
      return absf / 300.0;
}
```

That's horrible, don't do that. Mostly. Kind of. Well, think about it: how accurate do your values need to be? This is not a general question, this is a specific question for a specific situation. Is it possible that your values are from 0-10 and you could literally just precalculate your outputs? Are your values from 0-1 and you can just batch them into effectively 1 significant digit?

The answer is to test and see how much error is introduced, or what the instability looks like as the input sizes get bigger. What we're doing is not trying to cheat or lie. We're trying to find good approximations that simulate the complex functions. The tradeoff here is time for accuracy. But, realistically, being able to increase the speed by an order of magnitude means that you can increase the data size by an order of magnitude (for symmetry's sake, but in actual practice it'd be less). The increase in the data may actually make the outcome of the entire process more accurate even thought the individual sections are shorter.

So, the point is, emphasize accuracy for evaluating a trained AI. Emphasize speed (and therefore data set size) over all else for training the AI.

Side note: an even faster and well known Sqrt shortcut is actually just x >> 1. But you'd really need a much more constrained input set to be able to leverage that effectively.

CHAPTER 11 - A BIG BALL OF WIBBLY WOBBLY MATHY WATHY... STUFF

Sometimes subjects are exciting. A little boring and nerdy, but also exciting. Subjects can have more than one adjective. And, because the subject is so exciting, we got off on a huge tangent earlier. However, these algorithm design methodologies need to be focused on instead of just explained in passing on the way to another topic. But before we do that, we need a huge prepend to this. So we're going to break protocol and move the Practical Applications section up to the top.

Practical Applications

When we use algorithms in practice it is always (that's a bold claim, but we stick by it) different than in theory. The theoretical approach is claiming what can be done with the math outside of pesky things like silicon and electrons. But, the minute we introduce actual computing platforms we have issues. For example, is a print statement O(1) or O(length)? It's one line, but internally we all surely realize that there's a for loop that is iterating over how to print each character. For that matter, wouldn't each character be a matter of each pixel being rendered to the screen (at a very low level)? So, what gives?

This is a bit of the hand waving that we do in Algorithms. The algorithms are all true within their context. And then the practical and applied people write programs or libraries that implement them. But there are always technical tradeoffs that occur, and we've already seen a bunch of them. However, we're also going to start with a quick overview of practically what you want when you start designing or implementing algorithms. Or, as every LLM wants to tell us during the current iteration: "Now you're thinking like an algorithm designer!"

Hybrid is always the Answer

You'll never see a pure algorithm. Or, the "pure" algorithm is really just a coat of paint on two different algorithms crammed together to make them look continuous and like they're the same thing. Timsort is the primary example. But, it's pretty obvious that when one sort method is faster when n is less than or equal to 4 and the other sort methods are faster when n is greater than 4, maybe it's worth a single 'if' statement to check! But there are lots of examples.

One example we never really cover in a survey text like this is Introselect. We have already mentioned that

trees can get lopsided when they're not balanced. And it turns out that the growth rate of the function stack can follow a similar possibility of nightmare out-of-control growth. So, for the most part, we don't want our programs that are based on recursion to suddenly print an error on a bright blue screen: 'Oh by the way, we failed in the worst possible way, and your computer is toast now. So sorry, but I guess next time you'll think twice about entering a worst-case input!'

So, IntroSelect basically limits a Quickselect algorithm by depth and balance. Quickselect is sort of like a non-permanent quicksort that we use to quickly find the k^{th} smallest node in an array. That's obviously different than finding a specific entry (and that distinction applies to our next practical application). But it also means that we won't know anything about what the order isof the nodes unless we order it somehow. But now we've mixed sorting in with our searching, and suddenly we have a pedagogically nonviable algorithm. However, the point is that when the temporary quicksort is operating, it might get out of control or imbalanced and then we need a fallback algorithm to take over. Quicksort is fastest most of the time, but within a certain range, it turns out to be not as good as others (by potentially a large margin) and we drop out and switch to the backup sort (typically something akin to heapsort or median of medians).

Side Note: when would this matter? What if we want to display the top placers of a marathon. We don't need to sort everyone, we just need to find and sort everyone who finished 10^{th} place and higher. So we quickselect on k = 10 and go from there. In fact, we could even quickselect and find the top 50, top 100, top 500, etc. Every notice how displays tend to get less precise as the category size grows? They're using Quickselect like this because it's faster for something like a dynamic web display.

Side Note: Heapsort just turns the data into a heap and then forms the array by popping the top node off the stack.

Change the Problem

This is one part a constraint, and one part a shift in focus. For example, switching from finding the specific element in the array to finding the positional element is not really adding a constraint. It's a completely different problem! It's like the difference between (continuing the metaphor) saying 'I want to run a marathon in 2:55:18.4' and 'I want to run a faster marathon than Johnny Hayes' and 'I want to place in the top 3 in the Boston Marathon'. They're objectively different goals even if they all revolve around the idea of 'run really fast for a really long time'. The subtle distinctions are important. One is empirically viable, one can be measured without even a stopwatch, and one makes people go 'Who's Johnny Hayes' and if we're lucky we'll be able to see a spike in the traffic on Wikipedia! As of time of writing, the daily average for his entry was 28.

Another example is factorization of binning. Sometimes, you don't need the data in order, you just need to know how many of each type there are. So, you can get through the array in $O(n)$ time instead of $O(nlog(n))$ time. Or, recent research into 1-bit neural nets where the weighted edge is just a boolean value instead of a whole floating point number.

The point is that when you see a nail, it feels like every tool in your hand is a hammer. That's why you see

people trying to pound on something with the butt of a screwdriver or a crowbar or something. Sure, it might work, but there's just no precision there and there's lots of collateral damage. In the same way, when we see a list and there's some factor of order there, it's easy to think that no matter what our actual goal is, we need to get to that goal through a regular sort. It's important to remember that we can change data structures, we can change algorithms, but we can also change the problem definition. And that problem definition part is kind of outside of the scope of 'Computer Science' frequently, but it is absolutely within the scope of a typical developer's job duties. In fact, some might say it's the largest part.

Iteration is King

You won't get there the first time. This is not intended to be accusatory or cause you to doubt yourself. But, in practice, you should recognize that you will always do better on your second attempt at code. And better still on your third attempt. Maybe not your fourth attempt. But the pattern is that on your n^{th} attempt, you will generally perform better than all of your *0..(n-1)* attempts. This is so much so that you should expect and anticipate that iterative process!

When was the last time you took an English class and the instructor said 'you have to turn in your rough draft, I don't ever want to see any editing!' Unless it's a highly specialized writing exercise, that just won't happen. Similarly in coding, unless we are doing a highly specialized coding exercise we will expect to revise our code several times. Most places have code reviews where others can evaluate and suggest improvements to your code. We, as a public, ow expect there to be new versions of active programs.

This is both good and bad for the exact same reason. There's nothing so permanent as a temporary solution that works. So, there's a temptation to write sloppy code 'just for now and it'll get fixed before the next version'. But, avoid that programming pattern. On the other side, as long as we're engaging in broadly good programming patterns, there's no reason why the first version has to be exactly right, use the exact correct algorithm, be designed exactly right. There's time to iterate and improve. There's a lot to say for good design patterns before the first line of code is written. But, that doesn't preclude the fact that every professional environment will almost certainly expect that the program won't be ready on the first iteration.

And, again, that's freeing! It means that we have the time to look at how the program is operating and figure out how to customize or adapt the algorithm more exactly. We can even replace the algorithm. We can stress test our code and as long as we remember to always look, we can fix security or logic errors as it goes through. This is that process we expect in the process of adapting general algorithms to our specific environment.

Exhaustive algorithms are always an option

We have been calling them brute force, and that's an algorithm we're going to go into in the next section. But, there are different degrees of 'brute force'. Typically, when we say exhaustive, we mean something closer to one

that combines the worst aspects of naive and brute force. Instead of even trying to solve the problem, we look at our expected output and we just try every possible output to see if it works.

This is the worst algorithm type. But!

1. Sometimes there is no better solution. Password cracking is, generally, just pure random chance which characters are used. For people who make bad passwords, there are a few tricks, like checking for dictionary words first. But for so-called Strong passwords it's literally just a question of trying every possibility.
2. Sometimes there are specific issues with the dataset that don't mesh nicely with the best algorithms or the algorithm is just hard and deadlines come up fast.
3. The actual problem set is so small it's not worth trying anything faster, because the robust code is easier to read and reuse.
4. Sometimes we just need a placeholder because we need to focus on other parts of the code.

The point is that exhaustive solutions are the worst performing solutions, by definition (one way of defining the different classes of problems is in how big the solution space is).

Problem Reduction

The last answer is one of the most common ones. Why is it that you can only get 255 rupees in the original Legend of Zelda? Because the problem of 'how do we store the amount of rupees the player has picked up?' was difficult to make happen within the time and space constraints of the game, so they reduced the scope of the problem and said 'well, he just can't get more than 255! 1 byte of data, aha! Problem solved! I said good day, sir!' This is kind of just a variation of the constraint option, though.

However, when we talk about problem reduction, we're typically talking about some form of pre-processing. For example, if you wanted to create a program that automatically downloads album art and jacket information and sleeves and such from the Internet, you don't want to wait until the last minute to do all the processing. Instead, as the albums are added, you'll do some basic fingerprinting or during otherwise low computational programs so that when the time comes, the fingerprint can just be sent to the album content site rather than processed and then sent to the site. What was maybe an $O(nlog(n))$ algorithm seems to the user like it's $O(1)$, because the data was available before the user suggested creating it. Of course, that means that there's a lot of data created on spec. But, again, that's where the planning and design of the application come in.

Analogize

Every single algorithm is based on a real world problem, uses some conceptual framework to explain how it works, or otherwise uses a metaphor to make it plucky. Even the current most effective Matrix multiplication algorithm is called the laser method.

There are two sides to this. The first is that coming up with new problem solving approaches is absolutely most effective if you can associate it with a more concrete example. Nobody knows exactly what backpropogation actually is other than associating some numbers with some new numbers. But if you came up with a new version of backpropagation and called it the lawnmowing method because you visualize it as like cutting grass when you adjust the numbers to be like other numbers, you've got the beginnings of a good algorithm method on your hand. (This is literally what the attention/transformer model is. It's not really 'attention' that's just the metaphor that they use to explain the semantic web they build out of the parsed input.)

The other side is to avoid getting too caught up in the metaphor when you're learning how an algorithm works. Some of them don't really look like the thing they're describing, or they are filtered through a specific cultural or social experience, or they were created by a crazy person. For example, the phrase 'syntactic sugar' is intended to mean that some parts of the syntax in a programming language are there to make it more convenient to program in that language. But, others see that metaphor and just start applying it to any random idea and the meaning gets jumbled and confused because what is a rational metaphor for one person is not for another. So, only accept the metaphor with the context of the meaning behind t.

Takeaways

Notice that we spent a lot of time talking about the planning and design side of the programming. The programming is fairly easy (and becoming moreso as LLMs become better at some of the programming concepts and gain longer memory), but being able to constrain and preprocess and design and manipulate the code before even sitting down at a keyboard is one of the most important parts of programming. Discussing and communication about the algorithms in question are all key to solid implementation in most programming environments. Even solo devs need to be able to read what others are saying about various algorithms, they won't just know the answer out of the blue.

Brute Force

So far, we've differentiated Brute Force from exhaustive and naive algorithm design methods. But what does that mean exactly? We've already mentioned that one example of that is, essentially, not doing dumb things. For example, in the Bubble Sort algorithm, we only sort the parts of the array that aren't sorted. It's silly to

think we should check the already sorted parts of the array for whether there are elements that are not sorted. (Alternately, if there are elements that aren't sorted, there's a bug in the algorithm or the implementation that needs to be fixed. The correct solution is not to mitigate the error.)

Two of the most influential algorithm methods in modern history are backtracking and dynamic programming. These are brute force algorithms design methods! But they're so influential that we're going to cover them specifically in the next chapter!

Other examples are frequently fairly niche, although they can be helpful with some application. Most brute force algorithms that are worth remembering have a sort of twist to them. For example, the N-queens problem involves putting n number of queens on a chessboard that it is n x n squares in size such that no queen threatens another. There are probably less practical problems in the world, but not a lot. The point is, however, that the brute force method is just 'try every variation of the starting setup'. And the starting setup is to just put the queens on the board in a particular pattern of one queen per row. Basically, they trimmed the solution space by skipping past all of the obviously wrong solutions until they got to the first reasonable one and then started from there. (For example, it's not a valid solution to the n queens problem to only have 2 queens on the board.)

We talked about fingerprinting songs before, but that's exactly how a lot of apps work. It's now a reasonable feature to include in an app that it can record an audio clip and return what song it is. They aren't literally playing the song and comparing it to literally every other song in existence. However, they are running it through a hashing algorithm (another technique that's upcoming) and comparing the hash against the same hash of every other song. Both would come up with the answer in O(number of songs in existence) time, but the former will take exponentially longer for each comparison than the latter.

Slide and Compare

We're snagging this one because it's easy to see how it relates to other problems. So consider what other problems you might solve with this algorithm. But, "Is this string a substring of this other string" is a common task, right? Checking "is this character inside this other string" is an easy task. You should be able to write a linear search that solves it quickly. Can you write the equivalent of a linear search for a series of characters, in order?

So, for example, given P = "son" and T = "Clarkson", you should return either a true or a 5, depending on the context. But if P = "son" and T = "Kelly", you return a false or -1. Let's give it a shot:

```
int substring (char* p, int pn, char* t, int tn)
{
     for (int i = 0; i < tn - pn; i++)
     {
          for (int j = 0; j < pn; j++)
          {
```

```
                if (p[j] != t[i + j]) break;
                if (j == pn - 1) return i;
            }
        }
return -1;
}
```

We shouldn't need to go through the code line by line any more. But you can see that the code is a nested for-loop. We can guess that it's brute force because there's just two strings and it somehow manages to look like a quadratic function. But, how would we represent this? If ^{p} is length 3 and t is length 8, then $^{O(n2)}$ would be 64 and it obviously won't have that many loops. So, in this case we refer to it as *O(m * n)*. That's still quadratic, because it's still the product of two variables, but we're acknowledging that they have different sources.

Also, this is a clever use of one of our techniques earlier of changing the problem. We could have the first for loop be for `(int i = 0; i < tn; i++)`. But then inside the second for loop, we'd need to add a `if (i + j > tn) return -1;`. But if we preclude the possibility of an array out-of-bounds completely, even better, right? So, here is a policy question for you to ask yourself: should we still keep the out-of-bounds check there so that later programmers don't need to stop and read it several times to know that it's handled? There are three valid (maybe more) and equally usable answers.

But, you should see that the code is just checking with a moving start point whether it can explicitly match the substring. What other approaches could you use this for?

Side Note: There are actually two better, more improved algorithms. The Knuth-Morris-Pratt(KMP) algorithm from 1977 (just to note how old many of these basic algorithms are) as well as a backtracking algorithm. But they're still brute-force. The backtracking is still *O(m*n)* but frequently has more overhead and the KMP version does some pre-processing to allow it to skip already-checked sections. The code for the former is more complicated and the code for the latter isn't too complicated but it is too long for this chapter. However, KMP is widely available online.

Basic Math

Turns out math itself is a brute-force algorithm most of the time! Take adding two numbers. Think about how you learned to add 3-digit numbers. You start from the right and iteratively move to the left. That's exactly how the computer does it, too! The biggest difference is that the computer uses bits instead of digits. In general, we even have a term for it, we can have an 8-bit adder or a 16- or 64-bit adder. They're complicated to look at the diagrams for, but they're just variations of the normal full adder.

This is a bit of a tangent, but it doesn't have to be that way! One of the best parts of computers is that it's not 'real' math it's discrete math! Although the Big-O notation is a little nuanced at the circuit level, there's actually a whole can of worms you can open to look at better adding algorithms at the bit level. For

example, the standard full adders operate at $^{O(n)}$, but the carry-lookahead version only goes through *O(log(n))*. And, even better, if you've got an 8-bit computer why are you calculating the addition at all? Couldn't you just create an array of 256*256 =65536 bytes and just look up the answer every time? `A + B = lookuptable[A*256+B].` When would be a good time for that tradeoff?

Divide and Conquer

Divide and Conquer is one of the easiest patterns to recognize. For the most part when we have recursion where it recurses down two different tracks, we're talking about Divide and Conquer. When we split off a small part of the problem and check it then move on to the next it's the same thing. In general, when you have a problem and you subdivide it in any way to a smaller problem, it's Divide and Conquer. Merge Sort is probably the most famous example where it says 'sorting two lists half the size is faster than sorting 1 big list!'

The analogy is obviously based off of military tactics. If your army of 10,000 meets the enemies army of 10,000, it's very possible that you'll end up with nobody on either side. Like a Bronze Age Mutually Assured Destruction. So, strategy is the process of trying to set up situations where your bigger piece of army can go against their smaller piece of army. You want your 8,000 to be in a stalemate with the 9,000 while your 2,000 actively fights their 1,000. Instead of losing 1,000 soldiers, you'll likely lose 500 or less. The point being that in a head on collision, your 10,000 will zero out against their 10,000. But if you can divide them, you can conquer them. This is the much less elegant expression of classic Sun Tzu.

The secret here though is the recognition: an army is a group of little mans. They're like pokémans but without the 'the wild pokémon fell unconscious' part! If we have 10,000 little army mans, we have an army. If we have 2,000 little army mans, we have an army. They're similar when we're dealing with them in this extremely abstract conceit. Tacticians, of course, make their living off of differentiating exactly why one group of 16-44 soldiers is different from this other group of 16-44 soldiers. But, we algorithmists know that they're mathematically similar.

Here's the dirty secret, and it's going to sound like the kid from the Matrix who says 'There is no spoon'. But it's something like "Do not try to find the subproblem, instead realize the truth." "What truth is that?" "There is no subproblem. Then you'll realize that it's not the problem that divides, but you who observes the division." The problems are all metaphors to begin with. Just choose a different metaphor that already has an easy division in it, don't try to force the division into one that's hard. Worst case scenario? It doesn't work, and then you just try a different metaphor.

Obviously Binary Search from the Binary Search Tree is the platonic ideal of a Divide and Conquer algorithm. Notice, however, that it took a lot of setup. The array wasn't just magically divisible, they had to reframe it as a tree instead of a line.

Reversal

This is a bit of a tough on to set up. Because we can do the following pretty easily:

```
char* reverse (char* s, int n)
{
     char* ret = malloc(n * sizeof(char));
     for (int i = 0; i < n; i++)
          ret[n - i - 1] = s[i];
     return ret;
}
```

That executes in *O(n)* time, it's brute force, because there's just not a lot faster than that when we need to address every element of the array. But! What if we were talking about video that needed to be reversed or audio that needed to be sampled backwards or even just an incredibly high-resolution image that needed to be mirrored. Maybe we don't want to flip it, and we need it to be reversed in-place. Well, in that case we could do:

```
void reverse (char* s, int n)
{
     for (int i = 0; i < n / 2; i++)
          swap(s[i], s[n - i - 1]);
}
```

That's almost even easier. Although, if we look at the swap, we see that it does o(3*(n / 2)) swaps, so it will probably run 50% slower than the not in-place version But! What if we had that very large file that needed to be reversed in place, and also we needed to demonstrate divide and conquer. And maybe we've got a high end processor with a large number of cores that can multi-thread through the process. Then we might do:

```
void reverse (char* s, int start, int end)
{
     if (start >= end) return;
     int middle = (start + end) / 2;

     reverse(s, start, middle);
     reverse(s, middle + 1, end);

     int halfway = middle - start + 1;
     for (int i = 0; i < halfway; i++)
```

```
            swap(s[start + i], s[halfway + i]);

}
```

So, there was a divide and conquer algorithm lurking under there after all! Only, if you trace it, you'll notice that there's a for loop that iterates over the entire sub array in the function. For every function. So, every element of the array is going to be swapped n times. That's an $O(n^2)$ operation, maybe we can make it better?

```
void reverse (char* s, int start, int end)
{
     if (start >= end) return;
     swap (s[start], s[end]);
     reverse (s, start + 1, end - 1);
}
```

Well, that's not really better, it's just the same as the in-place version, but the iterative loop got swapped for a recursive loop. And it's still $O(n)$. But you'll notice that the function call stack int he previous version only ever went $log(n)$ deep. And this one goes $n / 2$ deep. So, this fourth algorithm is trading better time for worse space compared to the third algorithm.

The point is, just because we can make it Divide and Conquer doesn't make it better. But, we can frame lots of problems as an actual divide and conquer right in the middle. But we can also frame it as a divide and conquer where we're just slicing of single iterative bits at a time and it's just a for loop but with recursion. We mentioned several chapters ago that all iterative loops can be written as recursive and all recursive loops written iteratively. It's just a question of determining which one is more efficient (the recurisve algorithm might be better, but it's frequently the case that refactoring it as an iterative loop that otherwise follows the recursive formula will operate faster and use less memory).

In general, if you're not dividing and conquering in even swathes of the input set, it's probably not going to be better.

Math

Turns out, a lot of low-level operations are really slow. For example, consider multiplication. You're just repeatedly adding a number to itself several times. If addition happens as O(n), then the multiplication will be $O(n^2)$. But, we can make it better! We can basically split the values in half, create 4 temporary values based on known quantities, and then multiply the haves recursively. The downside is that it's complicated to set up in text. You can look up Karatsuba multiplication.

In fact, at the bit level there have been a ton of advances as people applied more interesting approaches to low level distributions. Discrete Fourier Transformation is a foundational shift in how we can represent

information. Fourier Transformations are, essentially, a way of recognizing that all sine waves are variations on each other. Thus, we can add and subtract them together and multiplex/demultiplex them at will to maintain several channels of information in the same basic signal. Fast Fourier Transform allows us to do DFT in $O(nlog(n))$ time instead of $O(n^2)$ time. Fourier, like Karatsuba, is a teaser for more advanced study in data structures and algorithms.

Greedy Algorithms

There are sort of three primary ways we can visualize Greedy Algorithms. We've mentioned it before and we'll mention it again here: all of these distinctions between algorithm types, even algorithm complexities are slightly arbitrary. There is no spoon. But, in order to have meaningful conversations about how to improve our algorithms, we've collectively agreed on some terminology so that we aren't always just point and grunting.

So, for a Greedy Algorithm, you can try considering them in one of three ways:

- Take a look through the input and grab the input that brings you simplistically closest to your goal. For example, the Greedy Algorithm for the Knapsack problem (remember it's one of the NP-complete problems, so the best algorithm for the correct answer is $O(2^n)$. This will get us a good answer, not the correct answer) is to pick the biggest item that needs to be packed and put it in the knapsack. Then the next biggest item that will fit, etc. This way we can solve the problem in O(n*m) time where n is the size of the input and m is the size of the output. Do not try this with a Handy Haversack.
- Take a look through the input, do some calculations or rely on some basic logical presumptions, and grab the input that brings us to the goal after a little bit of preprocessing. So, for example, the Travelling Salesman's Best-First Algorithm (remember it's one of the NP-complete problems, so best algorithm for the correct answer is $O(2^n)$. This will get us a good answer, not the correct answer) involves creating a priority queue (remember that was based on a min-heap) of potential paths, and then selecting one based on evaluating which one has the smallest cost so far melded with an underestimate of remaining travel cost.
- Finally, we can just grab the first element and make it work. We can fix it later, but we just get started on the first element and go from there. The Graph Coloring problem (it's one of the NP-complete problems, so best algorithm for the correct answer is $O(2^n)$. This will possibly get us a good answer, but only sometimes the correct answer) has a Greedy Constructive algorithm. Basically, just pick a node on the map and assign it the first color. Iterate through the entire map going to the first available node and assigning a color. The answer is either: that it failed and had to use extra colors than necessary, that it repeats from the start until it finds a valid option, or it returns the largest colored set it could before it failed.

Greedy Algorithms tend to be the best when you want to get a good answer, but you don't care if it's technically the best. When would this be the case? Suppose you're cleaning the floor of your kitchen and you finish. Are you really 100% done? No, you're probably more like 90% clean. Good enough for a day's cleaning, but you're not going to get it 100% clean every day. But sometimes you perform a deep clean where you get it way cleaner. It's not 100% clean, it's still only 99% clean. And maybe one day you get out a toothpick and a tooth brush and really go after all of the tiny little corners to get every speck of dirty of the floor. And now you're, like, 99.99% clean. At the point when you spend several days and gallons of hydrochloric acid to absolutely get that last 0.01%, you maybe realize that when you said 'clean', you really meant 'mostly clean'. And you just want a 'clean enough' floor. It needs to be above a certain standard. And your 'clean enough' is likely different than the 'clean enough' for other kitchen floors. But, it's not worth the extra time and attention to make the floor 100% clean.

So, when you know that you can get a pretty good answer in $O(n)$ time, but finding the exact answer is $O(n^2)$ time, maybe you just stick with the pretty good answer? This is even more so when we look at NP problems or self-diagnostic processes for computers. Hash Checksums are similar to that: error checking an entire downloaded file would be technically correct and take as long or longer than the download in the first place, and the checksum value itself would be as large or larger. But we can get away with a hash checksum that's only, say, 1024 because the odds that any given change happens to have the exact same value is low and for people to target a malicious change in a file and have the checksum match across several algorithms is ludicroius. Of course, saying something security-oriented is ludicrous sounds like 'famous last words'. But, still, it's beyond current technology.

String Tiling

Suppose we're doing something like checking for plagiarism. Let's stop for a moment and really examine our modern life. Now that you know all about algorithmic complexity, time analysis, different algorithm design methods, how does anything get done? For example, how do we take a student paper, put it into a website, and have it literally check if anyone else since the dawn of mankind has ever written those word? There's no zinger here. Architects are the ones least excited about actually living in skyscrapers, Air Traffic Controllers don't like to actually fly, and security engineers haven't figured out why we all haven't died in a robot uprising instigated by rogue state APTs. Once you're an expert in sausage making, actually eating sausage takes a bit more gumption than for the average person. Do not look up what's in chorizo. Just enjoy it until your innocence is shattered.

If you keep your eye open on the technology front, you'll realize that many companies super top secret method is really just a basic algorithm used at scale and with a fancy GUI. But, for example, why is it that certain plagiarism checking websites that get aggressively marketed and nobody in Academia would actually say was kind of a bad actor out of fear of retaliation– why is it that their plagiarism reports are just two things:

the longest section of copying and the percentage of the document that was copied and they make a big deal out of exactly how many words counts as "plagiarism"?

```
void stringtiling (char* p, int pn, char* t, int tn, int k, Match* results)
{
     bool* pmarks = calloc (pn, sizeof(bool));
     bool* tmarks = calloc (tn, sizeof(bool));
     int count = 0;
     int max_length = min (pn, tn);

     for (int curr_k = max_length; curr_k >= k; curr_k--)
     {
          for (int i = 0; i <= pn - curr_k; i++)
          {
               if (marked(pmarks, i, curr_k)) continue;

               for (int j = 0; j <= tn - curr_k; j++)
               {
                    if (marked(tmarks, j, curr_k)) continue;

                    if (strncmp(&p[i], &t[j], curr_k) == 0)
                    {
                         if (count < MAX_MATCHES)
                              results[count] = new_match (i, j, curr_k);
                         count++;
                         mark_range(pmarks, i, curr_k);
                         mark_range(tmarks, j, curr_k);
                    }
               }
          }
     }

     free(pmarks);
     free(tmarks);
     return count;
}
```

That is a highly nested function. Hopefully you can see why we would want to conceptually simplify it. Given

two documents of arbitrary length, we're essentially trying to find the longest string inside of there that is common to both documents. It's greedy because, if we're honest, we don't really care about every little minor transgression. We want to know what the absolute largest chunk is. So, you'll notice that it starts with the maximum possible length (assuming one document is wholly a subset of the other) and then starts going down from there. That way if it gets interrupted or we hit the maximum amount of time we're willing to let the algorithm run (MAX_MATCHES), we've found the most egregious bits.

Just to be clear, p and t are the two strings in question. pn, and pt are the lengths thereof. results is the list of whatever data structure the transgressions are listed in. And k represents the minimum length for a match. The arrays of bool are to mark places we've already found a match so we know we don't even need to check there (this is important), so that the shorter matches don't double-trigger inside of the longer matches. calloc() is like malloc, but it zeroes out all the data before giving us back the pointers; we typically don't use it because it's much slower than malloc since it has to do an O(n) wipe of all of the data in the allocated memory.

Side Note: this isn't actually very useful a function. We could copy 10 pages, but replace every space with two spaces and it would be visually indistinguishable for most people but this algorithm would find nothing. We can modify it to use words instead of characters (arrays of character arrays, just split on spaces), we could use some sort of fuzzy logic or wild cards to find matches that are mostly the same, we can add a pre-filter to remove special characters and change everything to lower case, etc. There are lots of variations we can add here.

Think about the other way around, though. Suppose we started with the smallest matches. We'd have a million false flags for every time both documents used the same single letter sequence. So, that's what the k is for, it means we'd start with, say, matches of length 30 and go from there. Except that if there was a section were a person copied 1000 characters from one document to the other, we'd trip it 971 times, and then 970 times when we iterated to matches of length 31, and then 969 more tmies at length 32, etc. So, by starting with the longest matches and then marking the boolean arrays to go ahead and skip those portions, we can zip through the otherwise marked up documents.

Either way, this function already operates at $O(n^3 * m)$ where m is the longest match length. Technically, we'd say it was $O(k * n^2 * m)$ where m is the longest match length and k is the number of match lengths checked. Since that's kind of complicated, it would be moderately inaccurate but reasonable to call it $O(n^4)$. But, it's also widely varying. think about the difference betwen one where there are lots of different short copies between them compared to one where an entire document is a copy from the other. The former would take closer to the upper bound and the latter would be almost just $O(n)$.

AI Example

Fourier Transforms are a wierd bit of a magimatics. Essentially, any sound wave can be represented by a combination of other waves. Think about the movies you watch. the final audio is a single wave length hitting your ears. But someone made that audio by assembling the individual audio from each different character's personal mic, the background clatter, and the musical score. Plus, don't forget the Foley artists, once of the

most fascinating jobs in history. This is not a provable sort of claim. It just sort of is, and we rely on it, and a large part of a modern technological society is built around relying on it. It's along the lines of the "theory" of gravity is so obvious that we don't need to prove it to make planes fly and satellites orbit and antigravity machines works.

But, what happened in the 1960's was that this whole signal processing bit became important to the cold war, and we started putting a lot of resources behind it. One of the things we realized was that we could basically represent anything as a sine wave, as long as it was more or less repeating in generality. And then we realized that we can synthesize the variations in these waves and then desynthesize them. So, we can encode large numbers of 'channels' of data in a single carrier wave. It quickly became on of the most underlying algorithms that girds the modern technological infrastructure.

This can't be a textbook about Fourier Transforms, there are literally entire books about the process.

And, in 1965 we came up with FFT, which was the Fast Fourier Transform, an algorithm that let us do these fourier transforms just slightly slower than the speed of the signal itself. The code isn't even that complicated. It requires some more advanced math to understand, but if we hand-wave that part away, we get:

```
void fft (complexnum* arr, int n)
{
     if (n <= 1) return;

     complexnum* even = malloc (n / 2 * sizeof(complexnum));
     complexnum* odd = malloc (n / 2 * sizeof(complexnum));

     for (int i =0; i < n / 2; i++)
     {
          even[i] = arr[i * 2];
          odd[i] = arr[i * 2 + 1];
     }

     fft(even, n / 2);
     fft(odd, n / 2);
     for (int k = 0; k < n / 2; k++)
     {
          double angle = -2 * PI * k / n;
          complexnum twiddle = new_complexnum (cos(angle), sin (angle));
          complexnum t = complex_multiply (twiddle, odd[k]);
          arr[k] = complex_add (even[k], t);
          arr[k + n / 2] = complex_sub(even[k], t);
     }
```

```
        free(even);
        free(odd);
}
```

Don't worry about it if you don't want to. What we're getting at is that this is the O(nlog(n)) of the Discrete Fourier Transform of $O(n^2)$. DFT basically requires us to take the signal and runs from all the possible frequencies and checks to see how much of each frequency is present. FFT does the same thing, but it is essentially using that magic math to skip over the fixed chunks that it's already calculated.

This is interesting for AI stuff because: why can't AI do that? What is it that stops us from being able to calculate parts of the backpropogation or to predict what weights will be so that the actual training will just need to be confirming, or to develop a way of multiplexing different layers of the neural network together so that they can be calculated at the same time and then demultiplexed? There are so many variations of this that are, right now, in the realm of science fiction. It's not even really 'theory', it's just 'why aren't we, yet?' with the answer being 'we don't know how, yet!'

The other answer is that we don't need to use AI to multiplex and demultiplex signals together. That's literally what the algorithm is for. It's a deterministic algorithm that will always give us the same output based on the same input. But, modern GPTs are incredibly good at simulating types of data. So, when we talk about creating audio streams or video segments or similar streams of data, we're talking in a very real manner about something that should be able to be split into component parts or that should be mixed into other inputs. Being knowledgeable enough to freely use fourier transforms on streams of data will allow you to step in and out of the expensive parts of calculation (the neural net) and supplement it with cheap bits of algorithmic simplicity (FFT).

Side Note: There's also a lot of similarity between how we can analyze time series data and doing predictions or analyses based on fourier transforms, but that's well into the advanced spheres of the industry.

The point is: if you've got a lot of AI experience or want a lot of AI experience, being able to demonstrate facility with Fourier transforms is one of those things that those who know will be impressed at your ability to meld topics. And those who don't get it will be impressed that you know two different but complicated topics.

CHAPTER 12 - IF EVERY ALGORITHM WERE PERFECT, WE WOULDN'T HAVE PROGRAMMERS

More algorithm design techniques! Here's the thing about these last few chapters. The first several chapters were almost lyrical. There was a story of increasing complexity. We went from unstructured to structured. We moved from individual data structures to linear structures to tree and heap structures. And then it culminated in the graph chapter! Because we approached everything in an incremental way- that's incremental like the clicky games, not incremental like the loop style- it hopefully makes more sense when we drop to the theoretical stuff.

Side note: the theory is difficult. It's possible that this is your first exposure to upper level computer science concepts. There's strict programming at the lower level, and then there's the science of computation at that upper level. There's several subfields that all interrelate and combine to form a semi-cohesive whole. They feel different to specialists, but if they didn't combine on some level, then we wouldn't have computers that did cool stuff. The problem comes that there's a big catch-22 between Theory, Databases, Operating Systems, Algorithms, Languages, Systems, Networking, Security, Visualization, and Interfaces. To study any one of them in a thorough sense requires at least a little knowledge of all of the others! After all, we won't understand why Data Structures exist at the lowest level if we don't understand how Language theory informs the design of compilers. And we don't understand why the Time and Space complexity is the metric we consider unless we understand the Turing machine from Theory.

All of that to say: don't worry too much about picking up every nuance. Your goal is to remember that things exist, but you can always look up how exactly they work later. Remember, we're all cyborgs in the modern era with half our brains on the Internet. (This isn't a new trend: people who excel have had half their brains in the library for the last 200- no, 400- no, 1,200- no, at least the last 2,073 years!). Ideally, you understand more of the topic with less effort than in more formalized treatments of the material.

So, all of this theory will culminate in the next chapter when we discuss proofs. Instead of static divisions, we opted to use a progressive division. The culmination of the lyrical portion is the first chapter of Part C, because the theory stems from the practice (or vice versa) so we wanted a reminder of everything we've covered at the beginning of the second half. Similarly, the second half of the second half will start with the culmination of the theory as we look to apply that theory (derived from previous practice) into more targeted applications.

In fact, if we did our job right, you should be able to take the book and (aside from internal referencing, reinforcement patterns, and other current pedagogical best practices) subdivide it in halves recursively and find that each partial textbook stands on its own merits to the extant of its scope. Which is coincidental to mention

now, because this chapter is also about recursion! We're going to be addressing the recursive algorithm design methods.

Dynamic Programming

Dynamic Programming is easy: it's recursion, you just remember the answers to problems you've already solved. This increases the amount of space you have to use, but it reduces the amount of time it takes potentially super-linearly. We actually hinted at one of the most basic answers int he previous chapter, but let's rabbit trail for a moment.

Truth Tables

We know from systems and discrete math that there are basic truth tables that we follow. For example:

```
  | T | F
---------
T | T | F
---------
F | F | F
```

is the truth table for AND. But what's the difference betwen that truth table and the following truth table:

```
  | T | F
---------
T | F | T
F | F | T
```

At their core, all binary math is not really math but a simulation of math. It's just logic tables that we extenrally apply semantic meaning to either because it makes sense to how we are simulating the world around us or because it is empirically useful. The point is that the basest mechanics of the computer are built on truth tables that are defined values not calculated values. We know that 0 + 1 = 1 because we defined the output of that function to be the XOR operation (and the value of the carry bit to be AND). We know that because we can simulate the fact that it reflects the non-discrete math operations of adding 0 and 1. But saying that the XOR/AND combination there means addition is semantically extrinsic, not intrinsic.

So, what's to stop us from making truth tables that are bigger? Consider that if the idea of a processor is to force electrons through transistors in increasingly complex patterns, couldn't we create a truth table that was something like:

```
    | TT | TF | FT | FF
--------------------
TT | TT | TF | FT | FF
TF | TF | TF | FF | FF
FT | FT | FF | FT | FF
FF | FF | FF | FF | FF
```

Now all of a sudden we have a 2-bit AND truth table that can calculate complex outcomes twice as fast! What's to stop us from making a 4-bit or an 8-bit or a 64-bit processor? We're processing the values at the same time by using cheat-y look up tables instead of any practical application. (To be fair, the 'lookup tables' of a processor are based on the electromechanical properties of the material that make up the transistor, which is an entirely different field or more properly several different fields all acting in concert). There aren't really 'lookup tables' for these low level operations, it's more that the engineers provide a declaration of how their transistors or subcircuits will work (so it's not a 64-bit transistor, it's just a circuit that provides reliable results), and we rely on the circuit responding reliably.

There is a huge downside to this: the lookup table grows massively based on the size of the input. It's why, in practice, we develop Lookup Table Processors specifically to act with O(1) on higher-order bitwise operations, but for more general purpose processing we still stick with 1-bit transistors. Substituting the truth table for the properties of the transistor materials is a good idea, but it falls prey to the same reason why bits are encoded as 1's and 0's. The difficulty in producing reliable and consistent measurable outcomes gets more difficult the more variations there are.

Dynamic Programming Arrays

The former section was true, but it was the metaphor. If we make bitwise circuits faster with lookup tables, why don't we consider simulating that in the higher level code? And that's exactly the practice that dynamic programming attempts. One example is called the Integer Partition Problem. This is where we want to know how many ways there are to add numbers together to get a certain value. for example, to get 3 we can do 3, 2 + 1, or 1 + 1 + 1. For 4, we can get 4, 3 + 1, 2 + 2, 2 + 1 + 1, or 1 + 1 + 1 + 1. This problem leads into other questions about how the number is made up and we can use variations on it for lots of different outcomes. But, ultimately, we're comparing every number from 1 to n for every number from 1 to n up to n times. That sounds like an exponential problem, or $O(2^n)$.

The brute force solution is, essentially:

```
int integerpartition (int n, int i)
{
```

```
     if (n == 0) return 1;
     if (n < 0 || i == 0) return 0;
     return integerPartition (n - i, i) + integerPartition(n, i - 1);
}

# do we need this psuedocode?
define Integer Partition (n, i)
     if n = 0 return 1
     if n < i return 0
     total = 0
     for k from i to n
          total += Integer Partition (n - k, k)
     return total
}
```

It's the standard recursive approach to solving a problem where we just basically pop off one input and then ask it to solve again for the rest. But try tracing it out and what do you find? If we send it 5, it will calculate the values for 4, 3, 2, and 1 and then add one to them (that's what the n = 0 conditional is for). But how does it calculate 4? By calculating 3, 2, and 1. But how does it calculate 3? By calculating 2 and 1. See how the algorithm just repeats the same calculations over and over again?

The problem is that what we're dealing with is a stateless problem. It doesn't know where it is, nor does it know what other recursive trees might exist. It only knows how to solve the right here and right now. That's great for recursive problems where we are, say, splitting an input in half. But when we're shaving off a single value, it's a bit much. So, what if we could create a state? This is called the Memoization approach (the first dynamic programming style), because we're sort of just making a memo of where we've been. Literally all it takes is an array and a quick 'have we already solved this problem yet?' And we get:

```
int integerpartition (int n, int i, int** arr)
{
     if (n == 0) return 1;
     if (n < 0 || i == 0) return 0;

     if (arr[n][i] == -1) arr[n][i] = integerPartition (n - i, i, arr) + inte
     return arr[n][i];
}
```

Now, all of those extra calls effectively get subverted and we can execute the operation in $O(n^2)$ time, with the tradeoff being the size of the array leaving us at $O(n^2)$ space. But that's not all, there's a second variation

of Dynamic Programming called Tabulation that will perform even better. Essentially, this is the difference between a top-down and a bottom-up problem solving method. As always, context determines which will be most effective, but in this case, notice how we're keeping a big array that's mostly empty but then we fill it in with values. And, more importantly, those values stick around even after we're done with them?

What if we just constructed each array and then finished with it? Then we get:

```
void integerpartition (int n)
{
      int arr [n+1] = { 0 };
      arr[0] = 1;

      for (int i = 1; i <= n; i++)
      {
            for (int j = i; j <= n; j++)
            {
                  arr[j] += arr[j - i];
            }
      }
}
```

This is the classic version of the tabulation version of the dynamic programming function. Notice that the big difference here is, effectively, that we switched from recursive to iterative loops. And when we do that we realize that we can get by with just a single array and just keep track of the current value, since everything's in the same function scope.

So, in the brute force recursive method, we have an NP problem that is 2^n (notice that it's not NP-Hard or NP-complete, part of that explanation is because the 2^n is not necessarily due to required calculations so much as it is from unnecessary repetitions). But just by keeping track of our previous answers and then using those, we drop to $O(n^2)$ with a space of $O(n^2)$! And then finally, we reduce the space to $O(n)$. This is how the algorithm evolved in the literature. But, it's also the sort of iterative improvement you'll see when you move from concept to implementation to eureka moment to eureka moment.

Other Examples

We actually see equivalent applications all over the place. For example, when you're playing a computer game, and it takes a moment to load. This is the part where they put in all of those 'reticulating splines' and 'aging cheese' and 'constructing pylons' messages. What they're really doing in the background is grabbing the textures and rigging and other components of the 3d characters in the game and combining them together so

that when the character appears on the screen, there's not a recursive deep dive as it tries to ready everything immediatly.

Or, for example, most games will precalculate things like 'XP to next level'. The reason why iterative metrics like that are always based on an easy to figure out formula ($level^{1.15}$ or 2^{level} or something), it's so they can tell you what the next target is based on what your current target is. This is a tangential, but sequential variation of these.

One of the most interesting parts is Greedy Fibonacci. Zeckendorf postulated (and then proved) that you could write any integer as the sum of Fibonacci numbers. This is both cooler and not as cool as it sounds. It's not really that cool because we've already seen that you can represent any number as the sum of other numbers, so this is just a filter and saying 'aha, but you can do it with just the Fibonacci numbers, too'. But it's cooler than that, because it turns out that we can do it as a combination of unique sets of Fibonacci values. For example, let's take a random number: 104. That's greater than 89 but less than 144, so we call 89. The remainder is 15, which is greater than 13 but less than 21, so we add 13. The remainder is 2, which is 2. So, 104 is 89 + 13 + 2. And there are no other combinations that make it.

So, you should be able to write a quick function to find the unique representation of any input number using just Fibonacci numbers. Using dynamic programming, finding any given Fibonacci number should eventually be O(1). But we can also just use a giant array of numbers, and since every number will eventually be used to represent a larger number, it helps to have already calculated them. So, the general algorithm would be for input n: find the next lowest Fibonacci number, f. Then add that in sequence to the already recorded solution for $n - f$. 3 steps! It just takes $O(n)$ to build up to that point.

Backtracking

Backtracking is, so far, the hardest set of algorithms we've avoided covering as well as being the most difficult to conceptually distinguish. The nice part is that once you get a good feel for what constitutes backtracking, you should be able to just retain that knowledge. But, until then, it may feel frustrating to figure out new algorithms. One common pattern is: 1 – not understanding backtracking, 2 – everything is backtracking! 3 – nothing is really backtracking... 4 – Acceptance and Incorporation.

So, let's set some rules:

1. Backtracking is always recursion. That's the whole point, although we may convert it to an iterative version, the proof and the permanent version are recursive.
2. We have some constraints to the problem. These may be intrinsic to the problem, but more likely they're there to help us solve NP problems in polynomial time. These constraints are trivial to identify when we've crossed them.
3. We frame the meta-algorithm as making a series of choices until we get to an answer. We can both make

a choice and undo that choice if we need to.
4. When we've broken a constraint, it's trivial to notice it and we can immediately cross off any further choices in that subtree.
5. When we've broken a constraint, 'pruned' that subtree, and backed out a layer, we undo what we did.

The classic example is combinatorial problems. For example, how many license plates have words on them, or more specifically: how many different words could a license plate have if we rearranged just that license plates' characters. You know you've done it before, with a license plate like 'IMAWINR'. They didn't really think that through. (Side Story: because mandarin is such a homophonic language and license plates are completely random, people sometimes have to deal with their license plate saying 250. Which is homophonically similar to 'I am a dummy.)

Sudoku Solver

Sudoku is just an exercise in combinatorics. We've got 9 rows of 9 numbers. There are special rules for each combination. If you aren't aware, sudoku requires:

- Each number 1-9 is unique in each row.
- Each number 1-9 is unique in each column.
- Each number 1-9 is unique in each of the 9 snapped 3×3 sub grids,

And then the puzzle comes because the designer fixes some of the numbers but not all of them. A good sudoku puzzle has a unique solution. What's interesting about sudoku, though, is that not every puzzle has a unique solution, that's just one of the (primary) distinctions between a good and a poor puzzle. People love to spend their time meticulously tracing through all of the various permutations and logic traps and gotcha moments. And then we, the evil computer scientists who like to ruin people's fun like the Peculiar Purple Pieman of Porcupine Peak lives to ruin Strawberry Shortcake's day, come in and automate the whole thing.

The rules for the sudoku solver are pretty simple:

1. Pick the next empty cell of the board (typically an array of 81 ints).
2. Try placing each number 1-9 in it.
3. Check if the most recent action breaks one of the existing rules.
4. if it was a valid move, then recurse to the next step.
5. if it's not valid, backtrack (undo the move, and if all the current moves have been tried, return that invalid state up the recursive stack)
6. Repeat until solved.

It would be nice if we could point to any one bullet number and say 'that, that is what makes it backtracking'. But it's really all of them together. In order to be backtracking, we're essentially going to start by picking an arbitrary next portion of the puzzle to solve such that we'll eventually get to all of it (and this is one of the constraints we'll give ourselves, that it has to happen in order). Then we have a choice of what to put there, any of the possible symbols (and the fact that we do it in order is another constraint). We can clearly see when we make a mistake because we can check rows, columns, and sub grids. We recurse to the next step. And we backtrack. There's no part of this algorithm that isn't quintessentially part of the backtracking paradigm. And that's part of why backtracking is hard to recognize: it needs all of the parts, or it's not backtracking (and probably not very effective).

```
bool valid(int* arr, int row, int col, int num)
{
     for (int i = 0; i < SIZE; i++)
          if (arr[row * 9 + i] == num || arr[i + col * 9] == num)
               return false;

     int subrow = row - (row % 3);
     int subcol = col - (col % 3);
     for (int i = 0; i < 3; i++)
          for (int j = 0; j < 3; j++)
               if (arr[(subrow + i) * 9 + subcol + j] == num)
                    return false;

     return true;
}

bool sudokusolver (int* arr)
{
     for (int i = 0; i < SIZE; i++) // 2
          for  (int j = 0; j < SIZE; j++) // 2
               if (arr[i * 9 + j] == 0)
               {
                    for (int num = 1; num <= 9; num++) // 2
                         if (valid(arr, i, j, num)) // 3
                         {
                              arr[i* 9 + j] = num; // 4
                              if  (sudokusolver(arr)) // 1
                                   return true;
```

```
                                arr[i * 9 + j] = 0; // 5
                            }
                        return false; // 5
                    }
            }
            return true; // 6
}
```

There are a few berry interesting points here:

1. This corresponds to the first requirement of backtracking. It's going to be recursive. And remember this part: because we're recursing like this, we're effectively going to try ever possibility. We haven't actually increased the worst possible case or the strict interpretation of Big-O.
2. These lines are the constraints, we're going to try i, j, and num in order. We don't have to do it like that, we could try best first or we could do some analysis to see if any of the rows or columns or sub grids have fewer empty spaces then than the rest and therefore possibly more a smaller solution subdomain. But, we're relying on fast decision making in this case to demonstrate the backtracking. Adding more complicated choice-choosing heuristics might improve performance, the only way to find out would be to ideate an approach and try it. You should ideate an approach and try it.
3. We need to check if the solution we're attempting is valid. for example, if the 1st row, 3rd column has as part of the puzzle itself, then the very first attempt (putting a 1 in the 1st col of the 1st row) is going to be invalid. This is how we reduce execution time: every solution we deem invalid in the top left cell is 80^9 solutions we've pruned away without even trying them because we know they'd be invalid.
4. Here is where we commit to the instantaneously valid (that is valid right now, no guarantees about in the future) choice and recurse to attempt the next cell.
5. The if statement at // 1 came back and said that whole attempt it went down was invalid, meaning it tried every number 0-9 in the next cell and and undetermined number of recursive follows, so we are going to revert back to the stage we had before we got here. What should we set the value to? Since we know that none of the values we've tried so far (1 – num) were valid, none of them are the 'previous' state. So, we'll set it to 0. This is not a fixed stricture of backtracking, sometimes the permutation just wants you to undo a swap, or return a temp value back into the solution-so-far. But this is why Sudoku is such a good example.
6. There's only two times when we return true: when we guessed the next possible value and the next recursion came back and said 'hey, that works!' and when we've gotten to the end of the puzzle and haven't run into any contradictions yet (which is the win state). It's easy to forget that we're returning true. Of course, the actual solution is the passed-by-value array of `SIZE * SIZE` elements.

Backtracking can become a routine technique that you apply regularly to problems. But finding a novel way of doing it requires a creative component in addition to the routine craft of the process. It's one of the things that distinguishes practitioners from theoretical academics: the latter want to cultivate that artistic, creative spark and the former want to cultivate the craft.

Looking at this problem, what's the search space for a solution? That is, what's the Big-O of the brute force algorithm? $O(SIZE^{(SIZE*SIZE)})$, which in the standard case is 9^{81} possible solutions (reduced a small amount by the number of clues present). That's $2.2*10^{77}$, which would take, roughly, 7 tredecillion times the remainder until the heat death of the universe at current technology. So, kind of not viable to try to calculate.

What about the backtracking version? Big-O of $O(SIZE^{(SIZE*SIZE)})$, which is much better! No, obviously it's the same. (This next part goes back to the difference between the terminology 'best-case/worst-case' and 'upper bound/lower bound' from the Time Complexity chapter.) The average case changes from $O(SIZE^{(SIZE*SIZE)/2})$ to $O(SIZE*SIZE^3)$. That's only 531, 441 possible options, solvable potentially under a microsecond. So, what's the difference? Imagine the following puzzle:

```
9 8 7 | 6 5 4 | 3 2 1
2 4 6 | 1 7 3 | 9 8 5
3 5 1 | 9 2 8 | 7 4 6
------+-------+------
1 2 8 | 5 3 7 | 6 9 4
6 3 4 | 8 9 2 | 1 5 7
7 9 5 | 4 6 1 | 8 3 2
------+-------+------
5 1 9 | 2 8 6 | 4 7 3
4 7 2 | 3 1 9 | 5 6 8
8 6 3 | 7 4 5 | 2 1 9
```

Only, remove the top couple rows and enough others to make it a puzzle. The puzzle doesn't matter, just check the fact that it will have to evaluate 8/9ths of the search domain (roughly) before it finally gets to a 9 at the top. And then it has to evaluate 7/9ths of the remainder before it tries to get to the correct value in the second block. In fact, we can estimate that it's going to be at least 9! * ~9^{81}, So, this is up there in terms of absolute worst case scenarios for the sudoku solver. It will take a very long time to get to the solution. There are mitigations, but so far none of them really solve the problem in all cases faster than the exponential time. Typically, we put in a timer and just give up, because even the self-evaluation of 'am I close to a solution' is an exponential problem. (Side question: out of our problem solving mitigation techniques from the previous chapter, which ones would you suggest applying here? Are they worth the increase in average-time solving in order to mitigate worst-case time?)

However, most problems are not going to be worst case like this. Most problems will have random numbers in the various fields, and we'll frequently find the invalid scenarios early (semi-randomly, based on the

distribution of clues) enough in recursive depth to mitigate some of the exponential aspects. So, we call it an average-case of $O(SIZE^3)$ (and just don't mention the worst-case O()), because it still has to cycle through three nested loops.

Side Notes: If you ever try to publish a sudoku solver, do it in reverse order and then you'll get a reputation for solving unsolvable problems (of course, the caveat is that you might get known for failing at "simple" puzzles). Also, a 6×6 sudoku board only takes around 327 million years to solve! A 12×12 sudoku board takes 2.4 lines of text just to show the number for the size of the search space (174 digits long). A 3D 9x9x9 sudoku board has a solution space that takes an entire page just to describe, and the only way realistic explanations of those numbers are stupid. Imagine an ant that has an average speed of 1 step every billion years, and it has decided to walk laps of the milkyway galaxy. It would finish trillions of laps and the search space would still only be barely scratched. Big numbers are a guilty pleasure for a lot of people.

Back to Backtracking in general

The secret to finding a good and practical backtracking solution is to find problems where the repetition is so high that a sizable amount of overhead can be used and it will still reduce the amount of the search space that needs to be searched. In fact, a better way to think about it is: what's the most efficient way to cross off the largest chunks of the search space as being invalid? We have two last thoughts.

Donald Knuth proposed his Algorithm X, which is a genericized version of backtracking. He created the Dancing Links data structure to intertwine with Algorithm X. Anything of the form sudoku, mosaic tiling, Tetris, etc., are all solvable with Algorithm X. But, an entire book could be written on just backtracking using Algorithm X. In fact, Knuth did write a book, The Art of Computer Programming, Volume 4, Fascicle 5. Any given portion of TAOCP is worth parsing through. You should never feel bad about not understanding any of it in a sitting. Remember that it has taken Dr. Knuth over 60 years to write down the stuff he already knows. So, for you to read that same stuff which you don't know yet in less than 60 years seems appropriately reciprocal.

Here's a quick list of other backtracking algorithms. Just figure out where each of the five components are, and you'll be the majority of the way to understanding the algorithms:

- Subset Sum, given a list of numbers and a target, is there a subset of the numbers that equals the target.
- Permutations, print all unique permutations of an input string (this is a semi-common test or interview question)
- Word Search Solver, see: evil computer scientists ruing newspaper games
- Palindrome Partition, figure out what the smallest number of slices to an input string are necessary so that all the resulting strings are palindromes
- Knight's Tour, move a knight to every square on a chess board (there's way more possible solutions than you'd expect)

- Balanced Parentheses, print all valid combinations of matching braces
- Cryptarithmetic Solver, this is where each letter is a digit, but it looks like two words equal another word.

Random

A side note: randomness plays an important part of computer science. So, let's quickly address the different ways that randomness applies:

Input

When we are playing a game, we use a random number generator as input into a program in order to make the game not exactly the same every time. There are lots of applications for random number generators. For the most part, we don't use random numbers inside of the algorithm itself. We typically think of algorithm analysis and design as deterministic. In general, we would assume that any random values in an algorithm would be of the form of: provable, deterministic algorithm section + random value + provable, deterministic algorithm section and we prove that the deterministic sections will work for all possible values of the random value.

Output

So far, we've talked about using a deterministic set of algorithmic steps. But, there is the opportunity to use randomness to aid in algorithm design. the problem is that random is actually much more complicated than just randomness. What do we mean when we say 'a random number'. For example, if we were going to trace the path of a unit of water vapor as it evaporates, we'd have a fairly random trail, but they still clump together as clouds as a confluence of random factors leads to temporary reductions in uniformity. So, when we talk about 'random', we typically don't want to use that as an approach for solving problems, because the randomness will frequently clump together and the outcome will be no better than a routine approach.

Uniform Distribution

When we think of 'random' in a computer sense, we're normally thinking of a uniform distribution. That is the order of the values is indeterminant, but the distribution is even across all classes. And there's something to be said for this approach, as long as there's a strong justification. For example, we may know from psychology that people are less likely to pick a PIN code that starts with a 0 or a 1, because it seems obvious. So maybe always starting with a 1 to try to crack a PIN is non-optimal and we should use a random value instead, but

we need to be sure that we get to all values. In this case, what would likely be the most effective approach is to pre-populate the 'random' distribution and then to just cycle through it as an alternative order for the symbols.

Outcome

The point is that we just wanted to make a note that Computer Science is generally the science of systematically finding solutions to problems. Not hoping and guessing that things will work. We can program those solutions, but that's not really the science of computation. It might be most effective to consider that algorithms are how we respond to the randomness of the input not part of the solution itself. Just like anything else, there are exceptions to this guideline, but absent a compelling reason, it's probably best to assume that the random component is part of the programming and implementation and not the actual algorithmic conceptualization.

Practical Application

One of the things that we've perhaps hit too many times is that most of the abstract and academic concepts are not immediately applicable in the majority of development environments. Again, it is necessary to understand these theoretical constructs so that you can talk the talk with others. But, you won't just be dragging and dropping your practice Linked List code into a production environment and be ready to go.

Now, that is. Or in the foreseeable future. But, linked lists were fairly common in community projects in the 90s, for example the old ROM2.4 codebase was filled with it. Standardization has allowed for better code constructs. But, what once was might come again. So, don't walk into a development environment and demand to be taken to their linked leader list. But, maybe keep in mind that they may be the solution for different computing environments in the future.

Regardless, here is a list of algorithms you should absolutely know inside and out if you want to impress people. These are all algorithms that could be dragged and dropped into the high level or low level environment of a huge number of modern projects:

- A* pathfinding – This is the algorithm you want to use if you're trying to get from one place on a 2-d map to another. It starts at the end point and then does a breadth first search of the likely very large map to the starting point. Along the way, it marks how far away each node is from the end point with some weighted values involved. So, for example, impassible terrain or slow terrain gets calculated. But, it doesn't just have to be maps. Anything that can be represented as, for example, a cartesian plane with weighted values on it can be used with A* to find the shortest path between two points, with its efficiency based on the heuristic equation given to it.
- Dijkstra's Algorithm – This is also a pathfinding algorithm, but it's between nodes. Think less map

traversal and more 'finding an optimal route in Risk to troop transition from one country to another'. The idea is that rather than a map or a plane or coordinates, it's a set of interconnected nodes (the actual graph data structure) with no positioning outside of the weights of the edges. Dijkstra will find the shortest path between two noes quickly.

- DFS/BFS – We've talked about these in a previous chapter. But being able to apply them to any given structure quickly is a common task. They're more for surveying all of the data than finding a path (broadly).
- Disjoint Set Union (DSU) – This is essentially a way of reducing the size of sets of data based on the relationships between them. For example, if you have a set of nodes and some are related to others, but the relationships are relatively sparse (that is, it's not a full mesh), what you'll find is that there are distinct groups of continuity. So, when later searching, why would you search through the entire set when you can just search through just the disjoint (that is no relations) set that holds the node to see if the other is in there. This is the primary algorithm that drives every search and social media company.
- Tries – These are trees, but instead of being binary, they have an entire symbol set. It's used for lots of tasks, predictive word completion being one of the most common. It starts with the root node and then, for example, it has 26 child nodes, one for each letter. When the user types a letter, it traverses to that letter's node which is the new root node. This way, it can use a quick heuristic to travel at $Log_{26}(n)$ to make lots of quick guesses as to what the user is typing to display to them.
- Markov Chains – What a Tries is for word completion, a Markov chain is for next word prediction. This is the tool that is used to make suggestions on most cell phones for what the next word will be (possibly superseded, but it's so simple that many single word predictive algorithms still use it). It only worries about what the most likely word to come after the current state (word) is, but most applications train the weights for transitioning from the current to next state based on historical data.
- Splines – Splines and Bezier curves are a way of simulating motion. We say simulating motion, because actual movement doesn't look natural in most cases the minute it veers from a straight line. So a spline is calculated to be the curve such that there is no 0 value in the 1st, 2nd, etc., derivation. The higher the continuity (generally) the more normal the motion will look to human eyes. Being able to talk conversationally about splines will particularly impress non-technical people.
- Ray Casting – this is possibly one of the most fundamental algorithms in game design. If you want to learn just enough math to understand the most basic parts of a game engine, ray-casting is a good start. It's the method for finding which object in a 3d (most commonly) space a line will intersect with first. This is how Minecraft knows which zombie you're clicking on with your sword. It's also used with robotics, computer vision, lidar, etc. It's a very simple but versatile bit of applied math.
- K-Means and Newton-Raphson – Almost everything you'll find in the field of numeric methods will be optimally covered in a standard library in almost every language. However, it turns out that you still need to know how they work so that you can explain what the findings mean and why you might be getting spurious results. Being able to talk through these two algorithms will cover 90% of the field,

although a lot of that will be the ability to adapt to variations. However, once you understand the goal, understanding the variations is easy.

There are so many cool algorithms that are worth learning. You should categorize them into classes according to how much you need to know. One approach would be:

1. Meh – It's OK to decide you don't care. You can't learn literally every algorithm any more than an English major can read literally every book. Although Book Goblin tries.
2. Mentionable – You recognize the name and you can talk about what they do, but you don't worry about it more than that.
3. Mechanics light – You can explain how it works, but it may take some hemming and hawing as you go back and insert steps that you forgot on the first pass.
4. Mechanics heavy – You are practiced at explaining how it works, because you've practiced. This is how you inspire confidence in your abilities with coworkers, potential employers, and clients.
5. Memorized – Bubble Sort. You have memorized bubble sort. Anything else short, sweet, applicable to your repeated projects and in your regular language, you should memorize and be prepared to write at the slightest excuse (without being That Guy) and maybe you get known for your obsession with it. Three, tops.

There's a mythical level called Mastery. You've written books on the subject. But only very special people meet that level.

AI Example

Approximation and Heuristics

What do we do when we can't compute and answer, we can't reduce the solution domain effectively, and we're at the doom spiral of knowing that we have several more steps of exponential growth problem to solve? This is exceptionally frustrating for things that humans excel at. For example, when we want to know how to get from point A to point B, humans instinctively know that circling the block 40 times and then making a left-hand turnout of the neighborhood is not correct. But the computer doesn't intuitively know that. And, in fact, there's no guarantee that circling the block 40 times isn't the appropriate solution until after we test whether or not that was the solution. This does, in fact, leave us susceptible to never finding an answer as the algorithm gets sucked down a rabbit hole of circular side trips.

The first step, of course, is the artificial constraints we've already considered. But those can only take us so far. The thing is, all of the really interesting problems are non-deterministic (in this case, that means that

they're exponential or worse). AI problems are frequently of this category. For example, 'is this a picture of a lemur?' is an interesting question, but it either brings in Computer Vision or statistical pre-training to come up with an answer. In a lot of ways, we've just been solving the problem by throwing more transistors at it.

However, because the interesting AI problems tend to be non-deterministic, the field of AI could be (reductively) described as the field for finding solutions to non-deterministic problems without having to search the entire possible search space.

This field is called heuristics. It's not a standard framing, but one way to understand the difference between algorithms and heuristics is:

- Algorithms generate a set of equally valid solutions, ideally a single solutions.
- Heuristics select from a set of equally valid solutions, all other factors being equal.

So, we could just randomly pick one. But heuristics is the process of coming up with any extra information that can help us inform our decision. For example, suppose that a chess playing algorithm narrows down the next possible move to one of four possibilities whose score is all exactly the same on the next board. It could probably pick the exact move that would maximize the number of win states if it had the time and processing power to delve all the way to the most conclusive resolution. But the point is that it doesn't have that time as often as not. But it needs to pick an option (realistically, this doesn't happen, because the heuristics are built into the score, but we're presuming this is before the heuristics). It will, for example, pick the option that has the smallest number of moves in the next turn. Why? Because risk management suggests that the projections will be more ineffective the further away the end of the game is, so the range of possible error is higher the longer the game will continue, so it will pick the equal option that reduces the game as much as possible. This is, broadly, completely outside of the scope of evaluating the game board, but it's better than random (this is assuming it's better than random, but high tier chess heuristics are secret, so knowing what's used in practice is a bit of guesswork).

When we can narrow the problem down to a set of good solutions, heuristics is the way we pick the good solution. And being able to use the term confidently is one of the tells that AI industry people are looking for (not so much a positive tell so much as that using the term incorrectly or non-confidently suggests a lack of familiarity).

Here's a sampling example:

```
int best_length(int* arr, int n, int target)
{
      int best = arr[0];
      int best_diff = abs(target - best);
      int max = sqrt(n);

      for (int i = 0; i < max; i++)
```

```
    {
        int sample = rand() % n;
        int val = arr[sample];
        int diff = abs(target - val);
        if (diff < best_diff)
        {
            best = value;
            best_diff = diff;
        }
    }
    return best;
}
```

Samples are a heuristic in themselves. We can't census an entire population, so the entire field of statistics has risen in the social sciences to demonstrate when it's more likely that the laws of math are wrong than that some statistically significant finding was due to random chance. It is computationally infeasible to find all solutions, so we choose from many good solutions. The good solutions are to only select a portion of the population.

Side note: There are two heuristics here. The first is, which boards should we compare if we don't have time to compare all the boards to find the ones that re the closest in length to our needs. The second is how many boards should we compare and rather than just pick a number we use a value that is pegged to the total number of boards to insure that we have some scoped scaling. We'd use log(), but sqrt() is faster.

Incidentally, that's why we have a token nod to Randomness in this chapter. Sometimes we do just want to use random numbers in heuristics. It should just be 'randomly pick', because there's almost always a way to be slightly better than random.

And that leads to the last issue: when we're picking, we're picking between choices that leave some level of uncertainty due to computational complexity. So, multiple options equally good, all other factors equal, uncertainty due to computational complexity, non-deterministic problems. These are the hallmarks of heuristics. And to apply them to AI, you'll need to figure out how to get them to run millions of times a second.

PART IV
PART D - BEING THE ONE IS JUST LIKE BEING IN MATH

Can you feel the impending completion of topics? We're almost there! This part is a microcosm of a standard theoretical textbook in the field. We'll start with Proofs and then move into iterative and quasi-linear structures. Then we'll move into the more expansive graph-like structures. Finally, we'll cap everything off with a big helping of security. Security is at the end, because it needs to be included, but it's not really part of the data structures and algorithms field. However, there are obviously basic algorithms and structures that will help with it.

That being said, security needs to be baked in. You must ensure that you have a solid grounding in basic aspects of secure programming before you get started into any development environment. Learning secure coding practices is a lifelong process. There are principles that you'll learn early that you'll keep finding new ways to apply. But there are also constant new threats and new things to be aware of while at the same time some old concerns become addressed systematically. We'll get into all of that, but it's important to remember that it's there and that it's part of the self-directed ongoing learning you'll want to consistently improve.

Otherwise, while these chapters are the shortest of the textbook (purposefully, as attention and excitement turns to confidence and a less frenetic interest), they're also the juiciest.

CHAPTER 13 - SHOW ME THE PROOF!

This is the part where we'd like to say 'here are proofs in 3 easy steps'. But, just like any other creative activity in or outside of math, the number of creative outlets blooms substantially and very quickly. Why? Because the outcomes are fixed, 'we want an algorithm that solves a problem', but the methods to get there are more or less unregulated. It just has to start from the basic premises (axioms) and build up a set of logical steps where each step is necessarily true from the one that preceded it.

This is not, however, a routine activity for the average developer. Reading and writing proofs in detail is unnecessary. The most intense that the average proofing process gets is to acknowledge that a paper with a proof was published (in, say, the 1970's) and that other academics all rely on it. There's a reason that the common programming paradigms are common: the proven algorithms are widely available to anyone who wants to use them. And the uncommon programming paradigms tend to be part of the institutional knowledge that companies don't like to spread around.

The point is that the ability to write a proof is one that can, effectively, be developed when the need arises because you've developed something new. We'll cover the process in general, but mostly that's as a hat-tip towards reading proofs. And reading and understanding proofs is the sort of task that comes with practice. In order to be better at reading proofs, read more proofs just like in order to play jazz better, play more jazz.

It's a balancing act of making sure you have the basic concepts, know that certain categories exist, and know where to come back to later when it is important.

What are we Measuring?

The big value that we're measuring is: correctness. When we say that an algorithm is correct. This is, from a high level, the only proof that matters. You want an algorithm that gets you the right answer, or at least an answer that you can use and pass on to the next step in a process. For example, in a business case where you need to write SQL code for a database query about clients whose recency score on an RFM analysis is too old, you just want to guarantee that you got at least all of the answers and nobody falls through the cracks. All of the more formal proofs are below level.

So, keeping in mind that this is not the algorithmic definition of correctness, being able to demonstrate that a function or an algorithm will always get the correct answer is a valid approach at the high level.

But, we also need to define some terms about proof:

Correctness

Correctness is, to a certain extent, what we already mentioned at the high level. At the low level, correctness is also the measure of whether the algorithm finds the correct answer. This is sort of a silly proof, because nobody would try to publish or share their algorithm unless it was correct. But, there are variations. For example, we might have an algorithm that doesn't find the correct answer, because we've redefined the problem so that it only has to find several good answers without actually spending the computation to find which one is the correct one. For example, we might change a mapping app to find 'fast routes from A to B' from 'the fastest route from A to B', recognizing that the practical difference in finding every shortcut through a parking lot or a drive thru in order to save 3 seconds on a 45 minute drive is probably not a good use of electrons. In that case, we would say that the algorithm does not display correctness. And, in that regard, it actually is a useful question to ask a search engine or LLM, 'does X algorithm display correctness?'

Side Note: Herbert A. Simon has an excellent concept on constraint programming that specifically addresses many constraints that are either too difficult to model simultaneously or produce no overlapping satisfactory area of a graph. The term is 'Satisficement', but it's from the realm of Management Science rather than Computer Science, so consider it to be taboo. And all the more alluring for it.

We can also refine the concept a little further:

- Partial correctness – It produces the answer or nothing, but. The but is important: sometimes it isn't producing anything because it's engaging in a multi-billion year exhaustive search of a non-differential search space. It's 'correct' in that it hasn't produced an answer yet, but that's because it just keeps going For some of these algorithms, we don't even know that it ever will finish. Properly, the question is whether it is in an actual infinite loop, and not just a really long one.
- Total correctness – It produces an answer, the answer is either correct or correctly that there is no answer, and it always terminates. We could even mark that latter as 'it always terminates in a reasonable time'.

That brings us to:

Terminating

This is actually an important question. We've implied a question several times but never really solved it. Because the solution tends to be a little icky (and it takes up a lot of space without adding any new concepts). Most of our search algorithms will return the first value that matches. But what if we want to find all the values that match? Those are, frequently, completely different algorithms! The reason is that the find the first tends to automatically halt. But when we want to find all of the matching elements, we need to redefine the problem such that we narrow the search space to a much smaller domain.

This is the actual reason why binary search is so important. If we're just looking for a single node, it's neat

that we can search an array by splitting it in half and so on. But, we can also just check every node, no big deal. But when we want to find every element of the array (or every element that satisfies cetain conditions) of complex data structures, we really do have to ask whether it terminates. So, for example, what if we aren't searching for a single node but for every combinations of N nodes that meets a certain criteria: the set is a derived complex structure whos Big-O value might be too hard to search for.

So, in some cases, an algorithm that performs slower but always terminates maybe better than one that performs better in the average case but doesn't terminate.

Complexity

We had a whole chapter on this. We will be mostly discussing how to create a proof for Completeness. However, most formal or academic papers that discuss the correctness of an algorithm will also discuss the time and/or space complexity of an algorithm. Anything from the standard set of O(1), O(log(n)), O(n), O(nlog(n)), O($n^{1, 2, 3, 4 \text{ or so}}$) should be fairly easy to understand. But anything with multiple log values or n in the denominator or floating point exponents or n in the exponent may be too complex to casually peruse.

We've covered what these look like in depth, but to review:

- Time Complexity – Not how long, exactly, an algorithm will take but what the bounds are on the completion of the algorithm with the reduced upper-bound being the formal stand-in for the colloquial 'worst-case scenario'. You should be familiar with Big-O notation and able to compare them.
- Space Complexity – Anything with a polynomial will probably be fine on a modern general purpose computer. Optimization is important, but that's beyond the algorithmic proofing side. You may need to limit yourself to logarithmic, linear, linear-logarithmic space complexity with tight bounding of the size of n for embedded or low-computation applications.

Remember that complexity is not a question of if the algorithm is correct, or whether it will finish. Complexity presumes both (because both were already explicitly proven), and is just expressing a bounding on execution time or space.

Determinism

Determinism is not necessarily the same as whether something is deterministic. But they are related, which is why we use the similar term in this different context. Leaving aside the rabbit trail of Theory of Computation terminology, an algorithm displays determinism when the same input always leads to the same output. For example, if you ask for the 5^{th} entry in the binary powers series should always give you 32. Asking for the 5^{th} tallest kid in a Kindergarten class will give you different answers for different classes. Or even, possibly, different

weeks. Mostly, if we have a determinant input, we want to have a determinant output. If we're playing with random numbers, it makes the proofing weird.

However, there's a more important issue. Determinism isn't really concerned with the answer (not always, but most of the time, correctness presumes determinism) itself. It's concerned with the state of the output. Consider a factory assembly line for climbing rope. Every, say, 100,000 feet of rope produced, they test a 10-foot section for tensile strength. How do they test it for tensile strength? If it needs to hold a 200-lb person, they don't just put 200-lbs of tensile force on the rope. They pull on it until it snaps and measure how much that required. It's a non-deteminant outcome because the status of the rope is different post-testing than pre-testing.

- The answer to the question 'will it hold a 200-lb person' display determinism when the answer is 'true' but not wen it's false.
    - When it's true, you have the answer that was tested and demonstrated, but you also have a rope.
    - When it's false, you have the answer that was tested and demonstrated, but your rope may be all saggy or loose or otherwise no longer pristine.
- The answer to the question 'how much will it hold?' does not display determinism, because the answer to 'did it break' is always true, eventually. And, at the end of the test, you'll have a floating-point answer in some unit of force, but you'll also have a broken rope.

Why are we going into detail here? Because whether there are side effects of an algorithm is important to know. But, on the flip side, it's also important to know if we don't care about the side effects. For example, if an algorithm is faster if it just overwrites an original array, but we don't care about retaining the temporary data that was in the original array, the tradeoff of no longer demonstrating determinism is worth it for the reduction in complexity.

Purity

An algorithm demonstrates purity when it displays both correctness and determinism. The phrasing is a little bit outdated, so sometimes we purposefully conflate it with the very similar and related term (but not really a 'proof' term) of Referential Transparency. This is in the same vein as why the terms 'whitelist' and 'blacklist' are now more properly called 'accesslist' and 'denylist'. There is some arguable justification for why some might respond negatively to the historical terminology. But the correct outcome by the industry is less focused on that and more focused on 'the new term is less idiomatic and more descriptive anyways'.

Regardless, a function displays referential transparency when, in order to demonstrate correctness and determinism, it also does not access any memory or storage outside of its function stack. Alternately, we could say that it doesn't reference any external values.

It's a little complex, because algorithms should normally list all of the values they need, and that would be

considered passing them as arguments. So, by proxy, we might suggest that algorithms, by definition, display purity. And that's why the intentional conflation with a programming term tends to be reasonable, because then we can focus on whether a function displays referential integrity rather than what the algorithm as a whole is doing.

This is a very specious distinction. In short, don't access memory outside of a function's internal scope if you can help it.

Soundness and Completeness

Every different system has its own variation of what statistics calls Type I and Type II errors (or alpha and beta errors, based on the symbol to represent them). Type 1 errors are when you make a statement that's not true. And type II errors are when you don't say something is true but you should have. So, for example, 'there are no answers' when there was one is a Type I error. It's not about positive or negative phrasing. It's about the accuracy of the statement. Type 1 is making an inaccurate statement. Type 2 is not making an accurate statement.

So, in algorithms we frame this around the answers that an algorithm produces. And algorithm is:

- sound, if every answer it produces is correct.
- complete, if it produces every correct answer.

Ideally, we want algorithms to be sound and correct. However, again, it's not absolutely necessary. For example, suppose you're a door-to-door salesman. You never need a 'complete' list of everyone who might be receptive to a new set of encyclopedia's in your sales zone, you only need enough leads to last you for the day. On the other hand, within reason, if a bunch of people who never expressed interest in a new encyclopedia and weren't good leads get mixed in there it's considered part of the business.

In the same way, algorithms may only return enough answers to fit the requirements because finding every answer is computationally challenging. Or, they might find too many answers because stopping and checking is too much effort. However, generally the latter issue is easier to correct. It's easier to prove that something is not a correct answer than it is to prove that an algorithm is not overlooking something. It is, for example, why the search for Mersenne primes takes two paths: one looks for every bigger mersenne primes while the other tries to show every prime below larger and large numbers.

Safety and Liveness

These are a bit more exotic metrics, but they should be brought back. They're not really metrics of an algorithm, per se. More like what happens in an ongoing function. For example, we might use them to

measure the performance of the functions on the Voyager 1. Safety measures whether or not it's provable that no crashes will occur (like memory buffers filling up or array bounding errors or divide by zero). Liveness measures whether or not it's making progress on a task. So, for example, the Voyager 1 is mostly sound but we realized that after 60 years in space cosmic rays and vacuum ablation are taking their toll on some of the subsystems and external factors are causing it to no longer always display safeness. In the same token, because it's past the heliopause, there's not really 'liveness' for its planetary observation hardware.

Types of Proofs

There are so many different types of proofs. We're only going to cover 'the important ones'. They're all important, if that's the proof that you need for an algorithm. But there is a sort of hierarchy of how complicated a proof is. When a mathematician says that something is elegant, they typically want a three-line proof by induction, not a probabilistic proof that took 50 pages of setup to demonstrate. The latter is valid, and proving something is statistically likely within 5 9's (99.999%) is something that companies will rely on. One might even suggest, although it removes some capacity for novel approaches, that all proofs are derivations of the two important ones.

Proof by Induction (The Important One)

We've referenced this method twice so far. This is, by far, the most robust method of proof that is preferred in most cases. In fact, because we've called it 'proof by parts', and each part can be proven in different way, we might even say that even proofs that rely on other methods to address some of the parts are really still a form of proof by induction.

So, what does inducction mean? Let's make a quick analogy. And this applies to problem solving, proofs, predictive analytics, or just math. We can extrapolate, which is where we take a set of data or lines and extend it out to estimate what inputs outside of the scope of our measured data would likely output. Or we can interpolate, which is where we estimate what inputs inside of the scope of our measured data, but that weren't explicitly measured, would likely output. So, if we have two points of data: (1, 1) and (3, 3): extrapolation is when we guess that (10, 10) would be a valid point and interplation is when we estimate that (2, 2) is a valid point.

One thing to note is that in the modern, statistics heavy world, we have a tendency to trust these inferential or predictive statistics with an increasingly critical eye. The statistics take too much analysis to fit into sound bites, so they get endlessly recycled into minimally truthful reflections of their original conclusions.

Proofs are not the same thing. A proof by induction is a proof of values that we haven't explicitly measured, but we are always basing that proof on necessary and logic-driven steps from the one before them. The term that is frequently used is 'provenance'. Statistics provide a break in provenance. The steps before the statistical

leap might necessarily lead to the steps after the statistical leap. But logical steps always necessarily lead to the next step. This is an entire distinction in paradigm. And, importantly, proofs are necessarily open and visible, all outcomes are arguable and mistakes can be pointed out from the superficial (not bad, just 'on the surface layer') delineation of the proof.

What actually is Proof by Induction?

Proof by induction relies on the inductive step. We start off with a base case. This base case gets proven through direct proof. For example, you might prove that the Fibonacci sequence value for the first number is 1. Why? "By definition." What's the second value of the fibonacci sequence? 1. Why? "By definition." By definition is the easiest direct proof, it's one step and it's essentially like that part of a Ryan George bit where he goes "That's the name of the movie!"

We also need to demonstrate how we're going to get from any given generic case to the base case and prove that it will happen every time. A common inductive step is to say that some function f(x) is based on some function f(x – 1), and all integers will eventually get back to f(1), which was the base case.

We also need to set up some constraints, or situations in which our algorithm not only doesn't work, but it isn't intended to work. So, for example, that's why some math functions say 'for all non-zero real numbers', because they know that one of the values will go in the denominator and that will cause a divide by zero error. So, like any good manager, they delegate the handling of 0 to whatever exists outside of the proof. More common in Computer Science is something like 'for all positive integers' since negative numbers don't go into the domain.

Lastly, we also need to define a success or failure condition. This is what tells us whether it's worth inducting further. Broadly, the success state is something like 'some condition is true before we start. That condition is true after we finish'. that is, broadly, the inductive part of 'inductive step'.

Oh, look at that! We've just describe the four components of a loop, all the way back in the first chapter. Inductive proofs are just loops. Or, alternately, all loops are really an algorithm in their own right that's so low level that we just take it granted that loops work, so much so that they're baked into the infrastructure of most languages.

Regardless, look for those four parts or create those four parts, and you've got a proof by induction. You can see that it looks more similar to a recursive loop than an iterative loop and thus you can also see why academics whose role is to provide these proofs like recursion so much. Incidentally, that also implies that an iterative loop is actually the transformation of the base case, which is the recursive loop. The recursion came first and then came the iterative loop. Iterative are 'higher level' than recursive, but the gain is that they're easier to teach, learn, read, and correct. (Of course, when we say iterative loops are "higher level", we mean that Assembly Languages is higher level than principled math or circuitry, so it's a nuanced position.)

Loop Invariant

This proof type can go here without being its own section. Loop Invariant is the name of the proof style for an iterative loop instead of a recursive one. Any Proof by Induction can be refactored to be a Loop Invariant Proof and any Loop Invariant Proof can be refactored as a Proof By Induction. They're basically just the same.idea in different form factors.

Proof by Deduction (The Other Important One)

Proof by deduction is simple. We just try every value. Computers are binary machines, which means that all of their data is a stream of 1's and 0's. Further, all of the streams are bounded in time at some point. That means that for every single possible input, there are only a fixed number of possibilities. This is why Discrete Math is the mathematical underpinning of Computer Science.

Sometimes, we don't need to prove every possibility. If we have a boolean input, we can prove that the algorithm gives the correct response for an input of true, and we can prove that it gives the correct response for and input of false.

Alternately, for a single byte, we can prove that it gives the correct input for 0, the correct input for 1, the correct input for 2, ... , the correct input for 254, and the correct input for 255. The actual deductive proof would have involved literally listing every single value, and the '1, 2, skip a few, 255' method we just employed is a form of induction. But (a). you can fill in the blanks without needing to waste paper or electronics, and (b) you can see that it's sometimes tedious and requires some extra meticulousness. For example, the Kepler Conjecture Project is 500MB of proof to demonstrate deductively the densest method for packing spheres.

Side Note: This is exactly the sort of proof that AI is excellent at.

Direct Proof

This is not a formal logic text. The field is too broad to even get started and any attempt to summarize in a single section is laughable. Suffice to say that the historical field of Formal Logic is extensive, math-based, and the basis from which every step is proven, inductive or deductive. But, here are some of the conclusions you might come to:

- Q.E.D – Quod Erat Demonstratum – thus have we demonstrated or that which was to be demonstrated. You demonstrate that something is true through a series of logical and necessary next steps or deductive reasoning. This is sort of a summary conclusion that 'we handled all the parts, there were no parts unhandled.'
- Proof by Contradiction – Suppose we have two options: true or false. We presume false and then run

through a series of logical steps to show that it always leads to a contradiction, thus the answer must be true. This proof relies on adequately demonstrating the problem domain (how many possible inputs there might be) and the inevitability of the contradiction.

- Algebraic Property – This is mathematical lingo for 'we mathed it based on math too simple and common knowledge to explicitly state.' For example, we don't need to cite axiom to add or number theory to count. Certain things are presumed.
- Implication – This is one of the formal logic bits, but it's intrinsic enough to be worth explaining. The idea is that it's given 'if A, then B'. And then we posit that A is true. Therefore, B must be true. We've essentially taken a step back and moved the proof from trying to prove B to trying to prove A, which might be easier.
- Inequality – This is how the bounding works. We don't need to necessarily prove that, say, a given truck is small enough to fit under an underpass. We could, for example, just prove that the truck was made following legal requirements, and that the legal requirements state that all trucks must be shorter than some value lower than the height of the underpass. We aren't exactly proving something, we're proving that it's in a range and then that everything in that range is valid.
- Set Theory – Basically, this is where all of the numbers come from, where ideas like distributive properties and commutative properties come from.
- Construction – There are some proofs that amount to 'there exists a thing', so all you have to do is demonstrate what it looks like. For example, you might disprove that all prime numbers are odd by construction 'there exists an even prime number. 2 is prime and even. Q.E.D.'

There are so many more, but again, these are just a brief overview of an entire course. But, direct proof is how we go from any one step to the next step or any final step to the conclusion.

Hybrid

Lastly, as always, it's important to note that a lot of proofs are just hybrids. For example, we might have a situation where an input can be one of four values, and for two of the values, we prove a different input inductively. For one of them, we prove the other input by direct proof, and the last one, in turns, gets broken down into more cases and then each case is addressed. Again, proofs involving rare numbers are like this , where they disregard large swathes of numbers but then treat numbers near the target differently.

Another example is the TimSort proof. It's inductively proven that MergeSort is faster for n >some threshold, but then it iteratively proves that Insertion Sort is faster for the smaller values. this is why Python has a threshold of 32: they know from empirical testing that MergeSort was slower for their implementation less than 32.

Slight Caveat

A lot of the terms in this chapter (as with many of the chapters) have been declared authoritatively. But that's because a nuanced detail of, say, terminology drift over the centuries and across languages is a bit much for a survey exposure to the topic. So, to the extent that you may find a different description of things. What matters is what happens, not how it's labelled. These categories are a semantic attempt to classify natural mechanics that don't always have fixed structure.

What to Prove

This is easy. All of the metrics in 'What are we Measuring?' are proven by any or all of the methods in 'Proofs'. Each metric will need its own proof, and sometimes a proof that the proof is necessary in the first place.

Practical Application

From a programming or applied perspective, we might see these proofs as being a sort of meta-algorithm. But it's not very useful for routine activities in the development field. After all, the metric is 'does the company require we use this package?' not 'where is the formal proof for every function in this package be proven to run in a given time complexity?'

So, here's a different approach to meta-algorithms: the Software Development Life Cycle (SDLC). Regardless of whether you use Agile, Scrum, Six Sigma, Lean or any other project management approach, you're likely going to be referencing a derivation of the SDLC. Essentially, it's a cycle because when you finish, it goes right back to the beginning again for the next business cycle. The stages are:

- Analysis – In this phase, you analyze the current situation and develop a set of problems that need to be solved. This can be done through code analysis, user needs analysis, noting which imported packages have had updates, reviewing the bug trackers, etc.
    - The input is the situation that has been maintained for some period of time.
    - The output is a set of goals that will ideally be attained or goals that need to be filled.
- Design – In this phase, you take the set of problems or goals and you create a plan for how to accomplish those things. This involves finding the specific solutions to the problems listed or making choices on how to approach goals and then delineating the exact steps to get there.
    - The input is the set of goals or problems to be addressed.
    - The output is a specific plan to attain those goals.
- Testing – In this phase, you try some prototyping or proof of concept or further check to see whether the

solution you've designed will work and that it will work at scale. This would be, for example, where you start to note that a function doesn't run fast enough for the planned usage, and therefore it needs to be fixed.
    - The input is the specific plan on how to attain the goals.
    - The output is a set of improvements on the plan or a binary decision on whether it's feasible to proceed.
- Implementation – In this phase, you take the modified plan and begin to implement it at scale. You take all the test code and make it production quality, you build out the documentation and the libraries and broadly incorporate good programming techniques.
    - The input is the plan and acceptance to proceed.
    - The output is a new situation.
- Maintenance – In this phase, you take the new situation that's an improvement over the prior situation and maintain it. It's the most basic setup, but it frequently takes the most time. Depending on the applicaiton, it might be a few weeks to several years.
    - The input is the new situation.
    - The output is the same situation that's been maintained for some period of time.

It's important to remember that almost all of the practical world of development is filled with 'meta-algorithms' like this. They're not really algorithms because, for example, they don't have a start or a stop point, they don't have a specific goal or a list of conditions. They're just a cycle because they're listing how things will proceed. But, they're akin to algorithms, so it's worth bringing them up as a note.

Math and Computer Science tends to think in terms of correctness and processes that will have a conclusion. Business tends to think in terms of cycles and going concerns and continuous improvement.

AI Example

A paper recently found that hummingbirds are genetically adapting to the proliferation of hummingbird feeders in California. Basically, because it's much more efficient to feed from the feeders than flowers, the birds that have straight, longer beaks are more successful and the wildlife ecologists can empirically confirm that change. What's fun as a Computer Scientist, though, is to say 'write a real world artifact that optimizes the gathering of nectar' and then imagining what it would look like. Would it look like a hummingbird? Who knows! But it turns out that lifeforms are remarkably adept at problem solving over generations. That's literally what Natural Selection is.

But imagine if a Computer Scientist were going to create an algorithm that was provably correct at optimal nectar gathering. First, in the real world, what constitutes 'optimal' for something like gathering nectar? Most fit? Most likely to survive? Most likely to develop unnecessary plumage to attract female hummingbirds?

Most prolific? By volume? By volume to mass? By volume to calories expended? Most likely to cause brain size to expand leading to super-hummingbirds able to leap tall buildings in a single humm?

So, first off, the idea of correctness for a broad ideation like 'nectar gathering' is difficult. And it's factored by the fact that it's the wrong question: what we really want is a critter that is capable of succeeding across multiple different environmental pressure environments. Or maybe that's not what we want. The point is, the answer of what we want is confusing, but we'll know success when we see it based on how much prolific the critter is.

So, what is our input? The DNA of a hummingbird. So, the search space is $4^{1,000,000,000}$ to search the entire genetic space and find the outcome that is most able to solve the constrained metric of 'fitness' according to natural selection, indirectly measured by procreation rates. Biology works on long time scales, but that is not a viable search space when the space between loop iterations isn't microseconds; it's about a year.

So, what do we have? An ill-defined problem, an ill-defined solution, an indirect measurement to proxy for the outcome, and an impossibly huge search space despite the inexactitudes. This isn't just a heuristic issue, it's a meta-heuristic issue. Here's the thing: natural selection has already solved the problem, and it was in less than a billion steps. If we could be as effective in coming even close to that space in that period of time with our computer algorithms, we'd be stoked! So, how does nature do it?

Meta-Heuristics

Meta-Heuristics is the realm of dreams and analogy. Whenever you see a 'their startling new computer algorithm works like a...' sort of news story, it's likely talking about a meta-heuristics approach. The idea is that we want to simulate what some natural process does, and we're taking a good stab at how the natural process works. Here's some examples.

- Genetic Algorithms – Based on Natural Selection. These are algorithms that create a huge set of attributes (like DNA base pairs), a couple of specific instances of 'pretty good' solutions, and then recombine those good solutions in random ways. It's like genetics, because the 'fitness' of each mostly-good solution is checked against a rubric of some sort, and only the best outcomes are brought on to produce the next generation. We hardly search any part of the solution space but we can very increase the quality of the outcomes. These are used in things like AB testing, game features (like landscape generation), or waveform cleanup.
- Simulated Annealing – Based on how metals harden/cool. We start of with what we feel is already a pretty good solution in the giant solution space. And then we change it just a little bit. If that results in a better outcome, we keep the change. If it doesn't result in a better outcome, sometimes we keep the change based on some probability curve. Metal cools at different and weird rates over time. So performing different steps in the treatment process at different times helps, but there's a sort of activation cost to get to the next level of cooling, it has to release more heat than the temperature necessarily reflects. So, what

we're trying to do is simulate that by not falling into a 'local minima or maxima' that would seem like it's better than its immediate neighbors but with just a little nudge would be better. It's like metal cooling because we start at the beginning with a very high probability of attempting the worst solution so we can explore more of the solution space. But with every iteration we reduce the 'temperature' mechanic and become less likely to try flowing to a less optimal but new area.

- Neural Networks – Based on how brains work. One idea of how the brain works is that it's essentially a bunch of individual little computers (the neurons) that are passing information back and forth between each other through the dendrites. We form neuronal pathways be repeatedly experiencing the same thing (even if it's just in memory). So, without arguing whether that's actually how it works in the real world, we try to simulate the theory in AI. Essentially, we're trying to create the robot brain by simulating the performance of neurons with semi-independent neurons. We'll get into the details more in a later chapter, but this is also a meta-heuristic whose goal is to skip calculating a large portion of the potential solution space and try to find a shortcut to the solution.
- Ant Colony Optimization – the idea that we can find optimal paths by simulating how ants follow pheromone trails, but shave off corners.
- Swarm Operation – the idea that bird flocks are many, but they seem to follow a simple enough set of rules to seem like one giant organism.
- Basin Hopping – Sort of based on the idea of electron energy states and how they move from one place to another to facilitate system stability.

There are way too many meta-heuristics to count or include. But these are common ones with easy to understand metaphors. Creating a couple of sample projects that solve basic problems like simulated locomotion or computer vision or whatnot are excellent projects to add to the portfolio. Neural Networks are just one approach (that happens to be the most popular and lucrative at the moment) to what is a large and diverse arena.

CHAPTER 14 - THERE ARE NO MISTAKES, JUST HAPPY LITTLE ALGORITHMS

When we want to talk about data structures, one of the things that we've been glossing over is that there's a fuzzy line between the implementation and the goal. So, for example, when we talk about a hash table, we can talk about the general idea of a data storage method in a sparse lookup table. Or we can talk about specifically the process of MD5 (Message-Digest Algorithm 5), a hashing algorithm that produces a fixed-length hash value from input data. The thing is that the theory is the abstract concept of what we're doing, but the practice has dozens of different applications. It's like how SQL has an official, but theory-based design document, and it's up to each implementation to translate into practical reality. That's why we have so many different SQL DBMSs that are each slightly incompatible with each other.

So, we're going to do our best to discuss how Hash algorithms work in this chapter and do some disambiguation. But first, let's break everything down into a few sub pieces. Side Note: Part of what we're trying to show in this chapter (the whole is hashing, so just a part) is how we can recombine different aspects of an abstract data structure and the results are very different outcomes.

The Parts of A Hash

The Problem

Consider an array. If we have the index of an element, we can find that element in O(1) time. But if we don't know what the index is, just that it's definitely in the array, it might take us up to n steps to find the value, when it's in the last position of the array. So, someone said 'Wouldn't be nice if we could do that faster, say in O(log(n)) time?' and thus Binary Search was born. In this case, we still have the situation where we can access by index in O(1) time, but we can also access by searching for a specific value in O(log(n)) time. And, there's no downsides (other than a little extra conceptual difficulty, preprocessing, and structural massaging through rotations, but that just takes practice)!

So, obviously, there's a next question, right? Wouldn't it be nice if we could access our data by index in O(1) time, but still access it by searching in O(1) time? And that is what a hash attempts to do! It does not succeed. But it gets close, we just have to accept a substantive amount of extra complexity in the process.

The analog would be to imagine that you're a network technician at an ISP (Internet Service Provider) and you get a call from customer service saying that a customer's house doesn't have Internet. In our cynical

analogy, because that's all the call said, we walk up and down and look at every single house in a given neighborhood until we find the one where the network isn't working and then they can flick the remote on/off switch twice to get it working again. That's O(n) time. However, if the person gives their specific address, you can walk along one street until you get to the appropriate turn, skipping all the houses down the wrong street, and keep only making correct turns until you get to the house in question to find, say, the MAC address of the NAP to flick the remote on/off switch to get it working again. (This is assuming 99% of the time errors are fixed by just tuning it off and then on again.)

But we all know that's not really what happens. When they first install the Internet to that house, they write down all of the information about them. And then someone looks it up in a big table. But how do they know which house corresponds to which place in the table? (One way, and literally how it might have been done in the era of file cabinets and Rolodexes), is to keep an index that maps each house to its location in the table would be to take the house number, then the first couple letters of the street written as numbers (So 1234 Main Street might be arr[12341200], or maybe even mtx[1234][1200]), and voila! We've got an index! We could put that on a big map or in a giant table or who knows what. But now when we want to look up someone's MAC address, all we have to do is the O(1) operation of looking it up on the chart!

What's the difference between looking for a file in a file folder in a file drawer in a file cabinet, searching through a big stack of customer records, and looking it up on a chart? Nothing in real life, just that the analogies tend to get conceptually linked to specific search algorithms. And table or chart lookup by index is the O(1) array access. (Incidentally, this is the same reason Amazon and Google know your address by latitude and longitude and not just street number: it's easier to just locate you than it is to try to track down street names and such.)

So, the problem is: we went from O(n) to O(log(n)), wouldn't it be nice if we could go to some form of O(1) with a search function in addition to array lookup? What this entails is coming up with a way to convert the payload data (value) into a consistent index value (key) that we can use to access it immediately by a simulated index.

Hash Function

So, the first part of the process is that hashing function. Most of what we have covered so far has been effectively lossless. When we convert from a linear array to a binary search tree, we lose the values of the original indices, but they weren't important the data was . So we'd consider that to be Lossless, we can still go back to an unordered array without a loss of actual data. But when we hash, the tradeoff here is that we do lose some data in the process. That's an important caveat to start with. We don't care about the data we're losing, because we keep the original. It's akin to how your LinkedIn page has a big picture of you for the main profile, but in long lists, people see the same profile picture but smaller. There are actually fewer pictures in the smaller image, but nobody cares because the larger image is also stored.

Suppose, then, that we have data that is 64 bits long. And we have 100 elements. If we wanted to look it up

by value in O(1), we'd have to have an array where the index was 64 bits long. That largely defeats the purpose. Stepping back, we'd have an array of 100 pointers, but to access them in O(1) time, we'd need an array that's the same size as the memory that the pointers point to, leaving nothing extra for the pointers to be pointing to. It's non-effective.

So, instead, we need a way to map a large possible set of values into a smaller set of array locations. If we have 100 elements, we might create an array that is 1024 elements long and place those 100 elements into pseudo-random-looking locations. The pseudo-random value is produced by a deterministic function on the data itself, meaning that the same input will always produce the same array index. For example, we could do something simple like take the first 10 bits of the data and use that as the new array index. This is simple and easy to repeat. If the result is `0b0000010000`, then we store or look up the data at `arr[16]`.

What if we want to look up to see if we have a value stored at `arr[0x4A8c3824]`? No problem, we just look up index `arr[0b0000010000]`, which is `arr[16]`. Well, that's a problem. This is called a collision and we handle it in one of two ways. We'll cover Collision handling in another section, but we can work around it by coming up with something a little more robust. In fact, the ability to distribute the simulated indexes more uniformly across the array is the hallmark of good hashing techniques. So, because we start using memory at the beginning of the memory stack, we're almost certainly going to have more addresses at `0x0000xxxx` than at `0xFFFFxxxx`. So, just using the first few elements is not a good hashing technique. It is a hashing function (called Truncation), and it's a fast hashing function, but it's not a good hashing function.

So, the hashing function is the first part of this hashing concept. In general, we're taking a large numerical space and mapping it to a small numerical space (notice that we're covering this towards the end of the text, because mapping like this in unstructured space is less constrained than graphs). The key is to do it consistently and as uniquely as possible. And then we can use it as an O(1) lookup of the data!

Checksums

If we have three components of a hashing algorithm, we can consider what happens when we take each part on its own. Essentially, if we can, we want to double dip on the development of the math. So, what's the point of the hashing function? It reduces the size of the numerical space. Now, imagine that we have a byte of data. That is 8 bits, so the number of possible values is 2^8 or 256. Here's the question we want to know: how can we guarantee that the 8 bits we're reading are the 8 bits we set previously? This is, for example, a big question in long-term data storage or in space craft where solar radiation sometimes changes bits.

Introducing the Hamming Code. It's fairly simple: We add a 9^{th} bit there that checks to see if anything has changed. We call this the checksum bit, because the sum of all of the digits needs to equal an even number. For example, if we have 10110011, then we can add a 9th bit as a 1 and we now have 6 1's (so the error-checking number would be 101100111). If we had 10001101, then we'd add a 9^{th} bit as a 0 and we'd have 4 1's still (so the error-checking number would be 100011010). The point is that if any single bit accidentally

gets corrupted and turned to the other value, we'll know it on the reading side because there'll now be an odd number of 1's!

The big presumption here, of course, is that "bit flips" are rare. So rare, in fact, that we might have 1 in a byte, but having 2 would be very rare. So the tradeoff is that we use 12.5% more space to represent data, but we get a (100-error rate)% cover of any given error. And, as we get more confident, maybe we switch to having an error checking bit every 16 or 32 bits. (Or, if it's a noisy environment, maybe we drop down to every 4th or even ever 1.) Going into too much detail starts looking at RAID algorithms and different striping patterns. Regardless, we can see that by including one extra bit we have the ability to strongly increase the probability that we can see an error.

These are called Parity Bits. To get the Hamming Code, we need to introduce several parity bits, each one covering a different arrangement of the data bits and rather than summing, we do it with XOR across the checked bits. Hamming wrote an entire book about it, but the point is that it is an overly complicated hashing function. Of course, if we consider the data -> parity, it's a reduction from, say, 8 bits to 1 bit. But, if we consider it, as Hamming's original example entailed, 4 bits -> 7 bits, we get a hash space that's bigger than the data space!

Security Checksums

What if we brought that concept forward a huge amount? What if we introduced a hashing function where we take a 100Gb file and introduce a hashing function that represents that file in, say, 64 bits. We can then see that within rough range of $1/2^{64}$ chance of being wrong, any changes to the original file will result in a different hash. Alternately, how many different 100Gb files are there? 2^{37} different possibilities. In order to check whether or not there has been an error in transmission or a malicious change, the brute force approach would be two download it twice to confirm and in the case of a tie, a third time for confirmation. That's mostly untenable. So, instead, we use more complicated checksums. The longer the hash output, the less space in the original search space each hashed outcome represents and the more likely we are to be able to notice a difference.

A good hash will even have small changes in the original file demonstrate massive changes in the hashed outcome (i.e. if a single byte is changed, say in the address of the update web address, then the hash will be more than a single byte different).

To disambiguate: we use a hashing function to generate a checksum value. It's exactly like the hamming code, but with a much more extreme input space -> output space ratio. CRC is a common one to check for environmental transmission errors. Frankly, making a suggestion on valid 'cryptographic' hashes is a bit of a gamble since they regularly get deprecated. But, right now SHA-256 is a common check for, say, files and such.

The point is that we're doing this not to find data in O(1) but so that we can download a simple check from the original source of a file or data stream to check that what we downloaded is still valid. Notice the lack

of algorithms: these are math-intensive, low level functions that can be looked up from primary sources most accurately.

Hash Tables

A hash table is basically an array, but like an array++. Just like when we create an array we can define our own dynamic array data structure that handles the resizing and iterating for us (that's what a List is in, say, Python), hash tables are a variation of arrays that handle the hash-specific ideas. In short, when we store a value into a hash table, we typically don't want to separate the steps, like hash_value = hash_function(payload); hash_table[hash_value] = payload. Instead, we just want to say 'insert(hash_table, payload)' and then later 'retrieve(hash_table, payload)'. Notice how we're kind of abstracting the concept of indexes a little bit. We're completely doing away with the whole 'how do we find it' part. That will become important for dictionaries.

Regardless, we need our hash table to handle a few things:

- We'll handle collisions on their own, but the hash table has to be able to handle collisions.
- We need some form of insert and retrieve function.
- We need it to represent the data internally somehow.
- We need to handle self-evaluation, automatic resizing, and iteration, but these are all closer to 'nice-to-haves' more so than intrinsic.

These are all standard functions, most of which we've addressed or should be simple to write on your own.

Dictionaries

There isn't really a thing as a hash table on its own (Alternate phrasing: the existence of a hash table presumes the presence of a hashing function). The idea of the hash table kind of requires a hash function to work. But, what happens when we don't have a hash function. In that case, we need to more specifically distinguish between the indexing value and the payload value. Broadly, we call this a Key-Value Pair (KVP). Alternately, we might phrase it as KVP(key) => value. One common example we give of this is a dictionary (the actual book). If we want a definition, we need to know the word. And the definitions are listed in alphabetical word order. The word itself is the key and the definition is the value. Or, perhaps, the key is the input and value is the output.

What we're looking at here is, however, more of a tight coupling. It's not that there is a calculable connection between the key and the value, there is a defined relationship between the key and the value. So, we don't use a hashing function to determine the key, we define the key.

In a dictionary (the data type), we treat it like an array, in that Dictionary[Key] returns Value. And, in turn, we can assign Dictionary[Key] = Value. What's interesting is that internally, we aren't actually doing anything

interesting. We still just have two arrays. For each entry we add, we just add the key to one array and the value to the other array but they both have the same index. This is the practical implementation, not the abstract data type.

In many languages, the term is just a 'map' (like in C and C-derived languages), but Dictionary is a more specific term while not being more idiomatic. So, we prefer that term for the general concept.

In practice, Dictionaries are frequently not ordered. However, they can be ordered. The sort operation and the balancing operation are performed on the elements in the KeyArray, while the ValueArray just follows.

This is both a side note about the concepts of Dictionaries, as well as noting that Key-Value Pairs is a concept that gets used regularly in Computer Science. It's any situation in which we represent a value with a representative key, and the interface favors giving the key exactly rather than a further representation of the key (like an integer). Key-Value Pairs always map from a key to a single value, but multiple keys can map to the same value.

Collisions

So far, we can implement a hash table using a hash function, and those are all we need. Except, sometimes it doesn't work. If everything is perfect, then the hash function converts the payload into a hash value. Then the hash table stores payload in the index of the hash value. If the hash space (possible outputs of the hash function) is large enough, we can almost guarantee that each payload will get its own value. The problem with that is the qualifiers 'if' and 'almost'. We can't accept the proof if it has 'if' and 'almost'. Instead, we have to break it into two classes: success and collision.

In a success scenario, no big deal. But in a collision scenario, we have to find a way to handle it. Recall that we just have to be consistent and repeatable in our handling. There are three issues to consider:

- Insert or Retrieval when there is no value there. Insert means there was no collision, retrieval means that the payload is not in the hash table.
- Insert or retrieval when there is a value there, but it's different: This is a collision.
- Insert or retrieval when there is a value there, and it's the same: Insert likely means we update the payload, retrieval means that we found the item.

Hashing Functions

The actual problem solving revolves around preventing collisions through manipulation of the size of the space and the uniformity of the hashing function. Here are two simple hash functions, but they get more complex and implementation specifics:

```
For all chars in a string, sum the numerical value of the chars, mod by hash
```

```
Given some hash value, h, for each char in a string, h = h << 5 + h + char.
```

Notice that we're trying to quasi-randomize the value in any feasible way, but such that we can always follow back to the same value. The first one does it with an immediate calculation and then domain reduction. The second does it through an iterative process that deterministically ambiguates the character values.

But, for example, notice that "cat" and "act" would both result in the same has value for the first. But, the first one is takes n time and the second takes 3n. When we're trying to calculate over billions of data points, those extra operations per cycle might matter! Of course, the flip side is that the tradeoff, based on context, is that having too many collisions and having to handle them might be more than the overhead of the longer hashing function.

Collision Handling

We've discussed around collisions, but not how to actually handle them. This is an active area such that many approaches exist. As always, covering many different application-specific solutions is difficult here. You should always look up what the industry standard is for your particular set of data. But, we can cover three of the most common general patterns:

1. Next Available – We just find the next available index to insert the value into. It's simple and repeatable! What happens when we go to retrieve the value? We just check the hashed value and then every sequential index until we get to an empty one! It increases lookup time dramatically in a crowded hash table until we get to O(n) at 100% population density. But, this is fantastic for a sparsely populated hash table.
2. Of course, it makes deletion much more expensive in a crowded hash table, because it means that we have to recheck everything in sequence after the deleted node until the first empty node to be sure that they're all re-indexed, otherwise we'll get orphan nodes when the now-deleted empty index stops the retrieval check.
3. Secondary Hash – We introduce several hashes, and then we cycle through them in turn. So, theoretically, if there's a value in the first hash index, we just use the secondary hash function, and the tertiary, etc. Typically it seems like two is about as much as is feasible. The problem is that it means that every lookup needs to have two lookups. So, the tradeoff is that every retrieval is slower, but if collisions are infrequent or the hash functions are simple, the overhead might be worth it.
4. This works best when instead of first-come-first-served, there's a priority system. So, the most likely to be accessed (the field of Operating Systems has a number of algorithms about how to predict the most likely for paging) is given the primary slot and the less likely given the secondary slot.
5. This is also more attractive when the write frequency is less imbalanced with the read frequency.
6. Chaining – This is very common in practice. Each index points to a linked list. If that pointers is NULL, it's obviously empty. But if it's not NULL, then every value that hashes the same ends up in the linked

list, no alternatives needed.

7. This works well for crowded hash tables where the distribution of collisions is fairly even. We can approach a situation where the average lookup isn't quite O(1), but approaches something like O(n / (hash space size)). So, it's an iterative improvement on linear search, and depending on the ratio of n to the hash space, it might be (but only in specific, never general, cases) faster than Binary.
8. What is the worst case lookup for each of these? O(n)! In Next Available, we might have 0 empty slots, which means that a retrieval has to check every index to see if the collision handling dumped it there. In chaining, every insert might hash to the exact same value (this would imply the hash function has a bug, in practice, but it's not technically impossible) and it's just the same lookup as a linked list. (Side Note: can you design a situation that's exactly the same but, by definition, worst case lookup is O(log(n))? What's the tradeoff?) For Secondary Hash, it's ostensibly O(m) where m is the number of hash backup functions. But, unless the priority system causes new inserts to delete old collisions, it will need a secondary collision handling mechanic. But, in practice, it's more complicated.

Hash table Resizing

We call the amount of populated indices in the hash table the Load Factor. The magic number seems to be about 0.75. There's no hard objective and empirical justification for this, just that it's the point when the hashing collision handling cost overwhelms the utility of having the hash table. Every single collision handling methodology has a probability model or performance curve associated with it. If not, some statistics (or paying a statistician) will quickly create the likelihood of a collision based primarily on load factor. This method assumes a perfect hashing function, and could be modified to account for data that doesn't hash uniformly, etc.

But, in practice, the assumption is that a Load Factor of 0.75 is the magic number. It seems to just be an industry standard to start there, and that's the answer you'll get. The problem with justifying that number is that because it's the 'known' factor, it's difficult to tell where in the mathematical justification the presumptions are creeping in. So, rather than justify it, we'll leave it at: the industry standard is to assume a target Load Factor of 0.75 and then adjust from there. High performance is considred to be a Load Factor of < 0.5, and slow performance at > 0.9. Again, these are just 'common knowledge' values that aren't worth arguing for or against. For high profile tables, it's worth calculating the specific collision rate for that specific implementation.

And that's, realistically, where the lack of empiricism in the rule of thumb comes from. Small variations have large effects on the performance of a hash table when we're trying to optimize. If we optimize a large enough hash space then there'll practical never be any collisions. Incidentally, this is the exact same justification for the ludicrously large IPv6 IP space! Trying to randomly find a valid IP address through a ping sweep is theoretically impossible (or at least improbable). There is always a tradeoff in overly large hash spaces. Most of the time, it's the amount of memory required. But that doesn't apply to IPv6, instead it's the increase in the

size of the header for every single packet sent (which suggests that 'no collisions ever' was a minimum criteria of the protocol).

Networking

Collisions on their own is also a thing. Any time that we have a random event or a mapping of any kind, we have the potential for a collision. In the box office, it's when two blockbusters come out on the same weekend. It's why Americans remember that The Dark Knight came out, but the movie adaptation of Mamma Mia is hazy. For Networks, the issues is taht there is a shared medium. Most network protocols are 'broadcast' on the local leg because there's no way to hide the physical properties of the, for example, electricity being pushed into the shared copper wire. Every computer on the local line has to have access to find out whether the signals are for them. So, even though it may be meant for the router to be forwarded onto the Internet, everyone has access.

So, what happens when two computers try to send signals at the same time? (Or, within the short window when the signals will interfere with each other.) There's a whole field of collision detection, because even knowing that there was a collision was a process that had to be developed. But broadly , when a collision is detected, all of the computers involved wait for a random amount of time before attempting to retransmit. This works because network communication is a time-based process: once the transmissions are separated in time, they no longer interfere with each other. However, this approach does not apply to hash collisions, since hashes are not based on timing.

But, the fact that collision detection and handling occurs outside of the context of Hashes was the important bit.

Practical Application

When we talk about key value pairs (a dictionary is a more constrained version of a hash table), we need to refer to databases. A database is, at its core, just a key-value pairing on the order of a Dictionary. Except that's entirely too reductionist to be practically useful.

Or maybe that's not correct, and Databases are really just a giant data structure. Several attributes all smooshed together and then arranged in a large array. And then each array is a different table.

Of course, the data might just be a giant raw matrix.

The point is that the reason there are so many different database management systems (DBMS) is because there are so many different ways to conceptualize what a database is. This is the practical reality of this text (we've given you several, pick the one you like the best!): Algorithms and Data Structures don't matter except that you can explain them. (The following is a set of very bold claims) That's the reason why math used to be the lingua franca of computer science and computerdom in general. It was such a new, niche, and arcane field

that the only way that people could conceptualize of these ideas that there weren't enough people to talk to about was to standardize the nomenclature. The most standard nomenclature is math, therefore it was math. But, as programming has ubiquitized, it has become more and more its own lingua franca. And as people are more comfortable around computers, the ability to talk about them in plain English becomes more clear.

Regardless, the reason we've brought up databases in this section is because the entirety of a database is a key-value pairing. The order that the records are sorted by and stored on disk is literally called the Primary Key. When we refer to an entity from another table by proxy through using one of its attributes (i.e., we refer to which students are in a class by using a student_id number, we don't recreate the entire student in the class table anymore) we're using a Foreign Key. Any unique attribute or combination of attributes is a Candidate Key. We can ever create secondary sort orders for a table by using, say, a secondary linked list whose ->next order is the Secondary Key.

What's the value? The entire entity (or row).

Every DBMS is seeking to provide a standard and consistent way of just manipulating the interaction between these key value pairs. The field of study for how to do this is called Relational Algebra or Set Theory, depending on how abstract you want to take the math.

The practical application of this is: what's the difference between a dictionary and a database? From the programming perspective, almost nothing! We send a key, we get a value. One of the nice things about programming is that we can make those conceptual distinctions that let us avoid the sticky wicket that is database administration.

AI Example

If you explain to someone in the field of AI that this is what you are doing, they won't appreciate it. But, there is something to be said that being able to specialize in creating code that can be run incredibly fast is potentially a skill that can be cultivated, excluding all else, and still lead to expertise in the field of AI. We're not saying that just low level programming is AI, just that if that's all you did, you could still potentially be a productive member of any AI team even if you didn't understand anything else they were doing. Most of the math, in the immediate scope, is fairly simple (literally one of the most optimized functions is called Softmax, and it's a custom 'which one is bigger' function). That's how important being able to optimally run low level functions very fast is. Blue hedgehog fast.

That all being said, here's the paradigm shift that started to occur in the mid-2000's, is continuing to occur, and will continue as long as Moore's Law is driving the industry. Once upon a time, the rule was that your decisions were only as good as your data. The pyramid is that Good Data -> Good Information -> Good Decisions. But what we're finding is that when we scale up the amount of data we're processing, we suffer a qualitative shift and not just a quantitative shift. Now, More Data -> Good Information -> Good Decisions.

(If you want to be ahead of the curve, the cusp we haven't figured out how to cross yet is More Data -> More Information -> Good Decisions.)

What does this look like in practice? Take Feature Hashing. This is the process of taking attributes that are complicated and storing them. Suppose we wanted to evaluate the tone of a document. computers are famously non-emotional. So how does it interpret tone? We use something called a lexicon that maps words to associated emotive words. So, for example, we might take the declaration of Independence and map it to Plutchik's Wheel of Emotions (a non-scientific, but still very popular emotional framework). We'd see that "When" correlates to Anticipation, "in the course" correlates to Trust, "of human" correlates to trust, "events" correlates to surprise, etc. And then we could count up how much of each word there are. There may be as few as 8 emotions depending on the layer of plutchik the lexicon dives to.

In a higher level language like Python, as we review the document, we'd parse each word or phrase and get back "Anticipation", so then we'd say mydict["anticipation"] += 1, but we'd do it for each word in the document. Not complicated, and at the end we'd just count up how many of each type of word are there and declare that the entirety of the document most strongly evinces 'hope'. Or, we might even just output the entire dictionary for the most full-scoped analysis. Done-zo!

But! That's kind of a small document. When we do textual analyses from a quantitative perspective, we want quantity over quality. So, what if we wanted to look at how the legal system had evolved over time in the United States? There are a number of law libraries, and recent versions might have hundreds of millions of words each (they're the kinds of books you buy by the foot in the used bookstore). And suppose we wanted to analyze every historical version on top of that. We're talking about scanning and calculating over billions of words! This means that the dictionary update of incrementing mydict["anticipated"] might actually take long enough to affect the total operation.

Feature hashing is when we just hash the value using a fast hashing function. And we get some number and mod it by some relatively large number, say 1000. We don't care about 1000 integers if we're running billions of operations . The size of the corpus (body of text) is so huge that a few extra bytes are not the issue. But that means that we don't need to worry about any aspect of hash table collision, and we never need to handle it. So, we can just run the quick, say, sum-of-chars hash and dump it into the hash table quickly.

It won't happen in this example, because we only have 8-24 different emotive words, so the likelihood of a collision is small and since we know it ahead of time, we could even just directly and statically map it to an array anyhow. But, lexical analysis like this is just a standing for, say, a large ETS process where we don't know how many attributes we will have to store, but we just want to count them. So, we'll just hash the value store it and if the original value stored at an index is 0 we know it's a new value and we store it as a potential value.

In this case, since we don't know what's coming, how do we handle collisions? The short answer is: We don't! When the dataset is large enough, occasional errors like we see in a lot of disparate datasets that are being merged together in an ETS process don't matter! In fact, like any other tradeoff, we have to make decisions about fidelity. The only metric that matters is 'how accurate are the predictive analytics that occur as a result of this data'. The balance, then, is not stopping to worry about every little error when fixing the error would

take enough time as to cause us to import less data, when the amount of data we'd be unable to import would aversely affect the accuracy of our predictive analytics. In short, as long as fixing the error takes more time than just importing another data point, it's counterproductive to fix the error!

CHAPTER 15 - THE FUTURE IS NOT SET. THERE IS NO AI BUT WHAT WE MAKE

We've been touching on different topics about and around AI in every chapter of the text. That's not quite what this chapter is going to be about, though. In this chapter, we're going to talk about the theoretical data structures that drive the mechanisms of AI. But, these won't be the practical aspects like in the final section of each chapter. These will be the much more generic, idealized structures that start with before we can make the AI. A good analogy for modern largescale AIs (notice we're avoiding saying GPTs or LLMs, those are just one of the many types of AI system that might be integrated into a modern organization) is the CPU. A modern CPU is full of billions or trillions of transistors. They're tiny, laser-etched, and barely worth being called 'components'. But with so many of them together, we can start to develop a modern processor. However, people learn, in the beginning, on single transistors that they can hold in their hand and put on a breadboard. The scale changes, and the outcomes are qualitatively different, but it's still the same quantitative scaling path to learn the large environments.

So, we're going to start with probability that presages the data structure and explains where we got the initial ideas. Then we'll jump into Neurons which are the specific implementation of the node structure we've been building so far. Then, we'll cover the entirety of the Neural Network and how it all goes together. This all ties together into the final outcome, which is 'what do we do with all of this?'

Naive Bayes

This is not a text on statistics, however learning more statistics will help you in both data structures, algorithms, and AI. So, we aren't going to exhaustively derive what Naive Bayes is, just that we're going to go through it. Before we do that, though, we want to mention historicity. Bayes wasn't really a hero of Computer Science, but a lot of his work revolutionized math, science, and applied statistics. We use Bayes' theorem any time we want to deal with representative samples and such. Also, you have got to really credit a guy whose Wikipedia article has a portrait that says "The only known portrait that is probably of Bayes... but it is doubtful whether the portrait is of him." The man was a specialist in probability.

What is probability? You can read many books and papers on the subject. They'll explain to you a number of reasons that the naive understanding is incorrect. However, as a working definition, you can keep whichever understanding you have as long as you recognize that there is a better and more circuitous phrasing. For our purposes, probability is the measured number of occurrences of a given event divided by the total number of measurements. This is a very practical definition.

Independence is the property of an event to not be affected by another event. So, we consider a coin flip to be independent of any other coin flip. We'd also consider the probability of a particular specimen of Butter Leaf Lettuce planted by a home gardener in Arizona going to seed to be independent of the number of satellites launched by non-governmental sources. Sure, chaos theory might suggest there is a relationship. But, we are only interested in what we can observe and measure. How do we prove independence? We have to establish a common frame. Frequently, that's time and that's why time series analyses are so common (everything correlates to time). But, these unrelated things don't help us.

Instead, suppose we get cornered by the mob, go on a wacky but PG-13 adventure, and at the climax of our Action-Adventure story, everything comes down to whether or not a coin flip comes up heads. Can we predict whether it's going to be heads or tails? Classic probability says there are two outcomes: heads or tails, therefore it's a 50/50. But, Naive Bayes (we've skipped many steps and jumped right from base math to the algorithmic side) suggests that there are a lot of different metrics we could look at. When was the coin produced? What kind of coin? If it were graded by a numismatist what grade would it get? What angle is the coin before it's flipped? What surface is it going to? We can collect a ton of data, and we want to use that data to predict the outcome of the coin flip because the heroine is being held hostage and the bomb is ticking in the orphanage and all of our wacky hijinks are at stake! (Fade to black)

Remember that probability is based on previous evaluations. So what we need to do is establish a way to prove that these features we're describing have some impact on the class of the outcome (heads or tails). Let's pick one: that the grade is A+. So, we measure how many times we got a heads out of all of our observations (P(H)). We measure how many of the observations were of an A+ coin (P(A+)). We measure how many times we got heads when the coin was A+ graded (P(H|A+). We measure how many A+ coins were flipped when we retroactively know they turned out heads (P(A+|H)). The question is does P(H|A+) equal P(A+|H) * P(H) / P(A+). If it does, we are independent! We succeeded at proving independence! Which kind of stinks, because that means we can't use the grade of the coin to predict the outcome of the flip.

Now, suppose we prove that every single thing we can think of is completely independent of the outcome of the coin flip. Except one thing: is the coin flipper left handed or right handed. For whatever reason (we don't guess reasons in this level of stats, we just measure. Bayes is a semantics-free zone), right-handed people more frequently flip heads and the flipper is right-handed! We can't guarantee whether it's going to be left or right. But suppose that we had 1,000 random people flip a coin each. If 60% of them are right handed and right-handed people flip heads 60% of the time, how much money will we make? We can only answer on average, which is why our example is always '1,000 people' and never '1 single coin flip'.

Luckily, as we've been progressing towards, AI is frequently about making billions and billions of small calculations, so you should be able to see where this is going.

Let's give a more exact answer. Suppose we're calculating email for an entire organization. We look at the contents of every email that is deleted within 2 seconds of viewing (it's never quite this cut and dried in practice, but we have to start somewhere). We compare that with the contents of every email that is not deleted at all. So, we start with a fairly exhaustive analysis where we do a Bayesian analysis to see if we can prove non-

independence between the probability of any given word and the probability that an email is saved or deleted within 2 seconds. This will give us a list of words, each of which give us a probability that the email is spam or definitely not spam. This then gets applied to every new email after the training period and we check what the probability is that any given email is spam multiplied by the probability that a word is in the email given that it's spam, for each word that is in the email.

The point is, we're trying to classify outcomes. Is it spam or not. We'll use this as a simple example of binary classification.

Perceptrons

The key here is that classification. We're looking for reasons to classify something as one thing or another. If we flip a coin and everyone is aware of all of the odds around it, then the gambling returns will be a 2:1 rate. That is, you get double your bet if you win and nothing if you lose. Thus, what we want is a way to do better than blind, random chance by using any set of measurements or observations to inform our decision. Random, classical probability events aren't a good example for this because we haven't found any ancillary features that affect which class we should predict.

But, what about things that just look ancillary? This is what Naive Bayes does. Without the math, the explanation is a little clunky, but it's a way of demonstrating that there is a statistically significant amount of non-independence between two observations. The perceptron is, broadly, intended to do the same thing. These are solving the same problem. However, the perceptron never really caught on, because it's not particularly effective.

Essentially, given an output that can be either 1 or 0, we classify it as 1 if all of the input values multiplied by their associated weights are greater than 0. Or, the sum of the inputs is positive rather than negative.

The classical perceptron-based neural network looked like an set of input measurements, each in their own node. They all have an edge connecting them to the perceptron with a weight for each value. This weight could be adjusted in a self-reflexive equation to help optimize the outcome. The perceptron would then, in turn, be connected to one or more output nodes which had weights for each of them. This is not a very effective neural net. It's got some input nodes, some output nodes, and a single hidden node in the middle. We call it hidden, because it doesn't have any values that are directly input or output, so it's not 'visible' from the outside of the neural net. The outputs were fairly simple and tended to get muddied. Even something like trying to simulate the output of XOR was impossible.

But, the secret sauce comes when we connect the perceptron to other perceptrons! What if we had 3 perceptrons. Each input node had 3 edges, each one connected to one of the perceptrons. The perceptrons would then each connect to the output nodes and the output nodes were classifiers that used the same perceptron function of summing the weights times the input values.

Perceptrons are a bit of a dead end, because they, essentially, just simulate the dot product from Linear Algebra. But they need to be trained and refined to get to that point instead of just being calculated.

Neurons

So, we moved from the Perceptrons to Neurons. A perceptron is a type of neuron. but because we were starting to link the perceptrons together and then to replace them with other things, we started using the term Neuron to represent the class of activation functions that we would use in place of the perceptron. (We've been talking classifier, but activation is effectively the partial unit of the classifer unit. Instead of classifying some internal value, it just passes on its raw value to the next layer.)

When we say 'classifier function', we mean functions like Naive Bayes, like Perceptrons, like Sigmoid functions, etc (on their own they're classifiers, but as part of the neural network, they're 'activation'). We have two perspectives: mathematically, we just see it as the function. But programmatically, we look at the function as just being a function of the node. And each node can run the same activation function as the rest or they can each get their own unique activation function, or anywhere in between.

When we say that the perceptron never really caught on, that's a little harsh. It's had a few resurgences, but other functions have consistently been more successful. For example, the Sigmoid function (named because it looks like an S on its side. We calculate it as:

$$\sigma(x) = 1 / (1 + e^{-x})$$

Why e? You'll need to look into numeric methods, but it's because e^x is its own derivative. So calculus is easyish! This is known as Euler's number, and the inverse $\log_e(x)$ is called ln(x) (for natural log). It's derivative is just 1 / x. Notice that our sigmoid function has the x under in the denominator. However, otherwise, x typically represents a weighted sum of the previous neurons. What this provides us is a fast and easy calculation to determine the value of any given node during any given execution of the neural network.

It is a bit disingenuous to talk about Neurons on their own. A neural network can be viewed as one large composite function, but we usually describe it as a collection of smaller computational units called neurons to make the structure easier to understand. Each neuron takes input values, computes a weighted sum, adds a bias, and then applies an activation function.

In a fully connected network with (m) hidden layers and about (n) neurons per hidden layer, there are approximately (mn) hidden neurons. The number of connections between hidden layers is roughly proportional to (mn^2), because each neuron in one layer may connect to every neuron in the next layer. So, under the hood, a neural network is not magic. It is a large mathematical function built from many smaller weighted-sum and activation operations connected together.

So, the point is this: A neuron is the unit that we swap in and out that controls how we adjust those summation functions. The weights are what get adjusted in order to improve the answers. The neural

network is actually the overall giant function. And the point of the entire process is the classifier: we want to know (effectively) should the output be 0 or 1. For example, is there a bonobo in this picture or not.

So, on the programming side, we tend to model the neural net as a directed graph. The weights that denote the relative value of each edge between any two sequential layers of the network are the data that we retain. The neural network itself is programmatic: the weights only make sense in the exact same neural network (mostly, we can design around that, but the point is that they're not general purpose weights sets).

Inputs

What are the inputs to a neural network? Let's imagine a neural network that is being trained to classify whether or not an audio clip contains a synthesizer wave of someone singing 'Mr. Blue Sky'. The inputs are likely the raw waveform data for each timestamp in the audio file. However, it could be lots of things. It could be derived data, such as graphical representations of the sound or some sort of waveform analysis. It could also be metadata like recording data, file name, etc. In practice, it's frequently all of those things.

This is for the input layer. For each hidden layer, the inputs are the layer before them. This becomes a little tricky, because each node can be handled uniquely but they're also intrinsically part of the whole program.

Outputs

What's the output of the entire neural network? That's a little bit more nuanced. A neural network is a classifier. It determines whether something should be a 0 or a 1. Could we link neural networks together or run them in parallel and get answers to slightly different questions. For example, we might have one neural network trained to determine if there is a primate in a picture and if that classifies as true, then it runs several neural networks in parallel answering whether there is a gibbon, a sasquatch, or a silverback in the picture, all while another several have run to determine which color is dominant (with each color getting its own neural network and the answer being a blend of the outcomes).

So a neural network can only ever answer one question. Traditionally, we would also say that it can only ever answer one true or false question. But there may exist proprietary activation functions that perform effective non-binary questions. But the point is that a network does exactly one thing. But many neural networks stitched together into a giant neural network can seemingly perform many tasks simultaneously and answer more complex questions like 'what sentence would most naturally follow a given question' or 'how many balloons are being held by this picture of a sad clown?'

So, classically, an activation function passes its output to another layer of activation function. But the neural network output is a classifier that determines one thing. When we say 'neural network' we typically mean the entire chain of many neural networks that all produce a more complex outcome.

Caveat Operator

We just don't have time (or space) to get into every variation of neural networks. There are many classes of neural network that are designed to create arrays of output or to output multiple outcomes, or sequences of value. But, practically, this does not invalidate this analysis. Part of the issue is that nobody knows what is happening inside of the neural network. So, predicting that the inside of the whole neural network is really a series of individual binary classifiers is as valid as claiming there is a holistic data transfer throughout. Research is still ongoing and there is no consensus on a number of aspects of how or what the neural network is producing. As a result, we've presented the most simplistic view, because otherwise it's easy to get caught in the weeds to start. Be sure to pay attention to primary sources, and not the claims of people selling something.

Neural Networks

So far, everything is a bit static. Let's change that. We can see how data moves from each neuron to the next neuron(s). But how do the weights and the values get there in the first place? This is an oversimplification, but we have two modes of operation. In the first mode, we're training the neural network and the process we're going to focus on is backpropagation. The other side is literally just called propagation, and it's when you're operating the neural network as expected to make predictions or take action.

Backpropagation

Neural Networks are famous for the idea of backpropagation and they popularized the idea. But you do it all of the time, both physically and in your programming. For example, suppose that you go to pick up a glass and you completely crush it. Next time you pick up the glass, you'll almost certainly be gentle. On the flip side, if you pick it up too gently and drop it, you'll pick it up harder. To a certain extent, this is what adults consciously do after certain kinds of injury as they learn to use muscles again.

Alternatively, as you're programming, have you ever gotten the wrong output? Maybe the spacing was off or the timing was a little suspicious. So you adjusted some values and got it to work a bit better? And so you tried going a little further and it looked even better. But then you tried a little further and it was much worse, so you back off a bit? You're incrementally adapting what you did in the past to affect what you want to output in the future. That's a fairly apt understanding of Backpropagation.

Take a neural network, we'll start with a going concern: one that's already been trained a little bit. Sometimes it gets the answer to a question right, sometimes it gets the answer wrong. It will never be 100% accurate, because that's not how statistics works (and remember that we started with Naive Bayes not because it was necessary, but to set the statistical tone). While the neural network is being trained, it has to be trained on data that someone already knows the answer to! For example, if we show it a picture of a snow leopard and

it says 'that's a monkey', we inform it that it's incorrect (there are a lot of behavioral modelling theories about skinner boxes and such, but that's well beyond the low level we're considering). When it's informed that it was incorrect, it goes from the output neurons back through hidden layers in reverse order and updates the weights so that the activation functions that activated would have been less likely to activate or so that the activation functions that didn't activate were more likely to activate, depending on how the overall network was incorrect (modern training also has a discriminator that will limit how many of the various paths it goes through, since it doesn't need to even check a whole swathe of 0's or 1's sometimes).

So, backpropagation is the process of updating each of the weights by a little bit so that they're more likely to get the correct answer the next time it attempts that same task.

Side Note: we don't always need to 'know the answer'. But there always has to be some level of correction. For LLMs as they're being conditioned after the initial raw language training, for example, the correction comes when they show the response to a human who says 'yes, that was a pleasant response' or 'no, that was a horrible response'. This is why most neural networks have a hard time talking about socially unacceptable things: any answer that smacks of aggression or violence or dirty words tends to be disincentivized by the human reviewers. This is why LLMs and image generators that center on vices tend to be open source and developed with alternative organizational structures.

That's it. The entirety of the training process is updating the weights to make it more likely to get the answer right the next time. And the emergent behavior that occurs when there are trillions of neurons is the modern AI landscape. The causative steps in between are the current developmental and experimental areas. The reason training takes so long is twofold. 1) In order to train a sufficiently large AI to brute force through the statistical barriers of 'sufficient size' to simulate realism, there needs to be huge datasets and that just takes time and electricity to process. 2) There are a lot of neurons there that need to be backpropagated through and, as should be apparent by now, some of the backpropagation algorithms can become more in depth than just a simple nudge one way or the other for the weights.

Lastly, how does this start? The earlier presumption was that the neural network was already partially trained and we were updating its values. So how does it get started? This is also an area for discovery at the moment. But, and a lot of people will not like this answer, it basically comes down to guessing. The initial neural network will almost certainly be incredibly wrong very frequently. So, to a certain extent, it doesn't matter what the initial values are, they'll all need to be updated eventually anyways (after all, if we were good enough to 'guess' correctly, why would we need to train the neural network to begin with?). The goal, then, is to be as close as possible to reasonable starting values as possible. There are algorithms to determine reasonable starting values, and it's a bit of an exaggeration to say that they're just a fancy way of guessing. But they're basically just fancy ways of guessing.

Propagation

After the neural network is trained, how does it operate? Exactly as we'd expect! We have a series of nodes, the

first layer is a series of input nodes. Whatever the input is (a picture, a prompt, sensor data) gets structured and transformed into the appropriate format and then pushed into the input nodes. The input noes calculate (theoretically simultaneously, but sequentially in practice) and populate their outputs. These outputs are then multiplied times their edge's weight, typically all summated and fed into the next node's activation function. Variation between different nodes will happen naturally through backpropagation and magic (again, magic is literally as valid a suggestion, at the time of writing, as any other explanation of the internals of a neural network), but they can also be encouraged by using different activation functions for different nodes. This process repeats for each hidden layer until the output nodes. Then the outputs are processed into a recognizable format and output to the user.

Forward propagation is relatively simple in comparison to backpropagation, there is much less calculation, and the calculation that does occur is very highly optimized. Most of the processing that happens in both forward and back propagation is floating point in nature and therefore GPUs are much more optimized for the calculations over CPUs, but either can do the calculations. This is why anyone can run even very large LLMs, but only a few groups have the capacity to train them.

Form Factors

So, what do we want these neural networks to do? Let's go through some common current models:

- Generative Pretrained Transformer. The name is exactly what it sounds like, it uses the transformer model to convert textual input into the attention model of concepts. It's pretrained so that all of the weights are already where they're going to be. And it's used to generate new words. This is novel in that it generally doesn't try to retrieve answers, it creates them probabilistically, and the training method just makes sure that it's typically right. The point is that it is good at transforming text into a different form of text that otherwise holds certain factors constant.
- Large Language Model. This is the superset of GPT that's trained across a huge amount of text. Typically the text will encompass something like 'every available webpage, consenting e-mail accounts, scanned books, and human interactors'. This means that while it can do what a GPT can do, it can also act with very little input to produce simulated environments.
- Generalized Adversarial Network. This is how we create image generators. We train two neural networks. One's job is to evaluate what's in an image and make a guess as to what it's supposed to be. The other tries to create images and feeds it to the other. We've covered this in an earlier chapter in more detail.
- Diffusion Model. In a diffusion model, we take a normal picture and then we slowly turn it into static, which is an 'easy' process, we just slowly remove information from the pixels. 'Easy' in this case just means through traditional algorithmic methods rather than AI methods. Since we have a series of steps it took

to turn an image into static, we then reverse them and train the AI to turn the static into images. "Static" being the random pixels that an untrained neural network will likely generate.

- Reinforcement Learning Models. These models will continue to adjust their weights (typically a subset of their total weights) as they are executed in order to slowly refine the model.
- Retrieval Assisted Generators. These are models that are attached to a database of some kind. This database typically includes a corpus of documents or the codebase for a company. The AI will query the database first, which means that it's a good way to provide specific guidance as to the scope of what should be covered.

There are others, but these are the current popular terms that are also likely the base level for learning about future terms sans a paradigm shift in the manner of the Attention is All You Need paper. However, what's important to note is that despite the popularity in recent years of AI, we've been using "AI" for decades. It's what allows us to have movie special effects, it's what lets photoshop function, it's what produces game characters and limited chatbots prior to GPTs. The big distinction is that anything based on a neural network is ultimately a successor to Naive Bayes. And that means it's always statistical probability-driven.

Practical Applications

We talked about how to train an AI. But where does the training data come from and where does the starting data come from? There are two big divisions in AI (more so Machine Learning and Analytics). Picking one and specializing in it, then being able to talk more authoritatively about that subdomain is probably a good decision. But knowing sufficient amounts about both is also a fine decision. Being able to articulate the reason for the outcome is an even better skill to have.

Unsupervised Learning

In unsupervised training, the model is given no prior instruction (or limited prior instruction), just a framework for evaluating its own performance. Unsupervised training tends to take the form of a goal (like move a computer model as far to the right as possible) and a set of constraints (you can only make wheels and lines, and you're subject to a physics simulation). The model uses some form of meta-heuristics to slowly mutate it's attempts and evaluates what it's doing as it goes.

Alternately, an unsupervised model might be given a goal (find statistical significances) and a problem domain (a specific set of tables) and told to find all possible outcomes. How it does it is sometimes dictated more or less specifically. This is a much more deterministic approach, but it follows the trend that more data will lead to good solutions. So, just like probabilistic neural networks gain a large amount of accuracy with large data sets and neural network sizes, unsupervised learning models also benefit from large sample sizes.

We tend not to call this framework 'AI' from a technical jargon perspective. This is much more in line with Machine Learning. This is why it's called learning and not training, because there doesn't tend to be feedback. The heuristic or self-evaluation model is built into the process. Machine Learning can be used to design neural networks, and neural network can be used to design Machine Learning experiments. So, they are all tightly wrapped together, but they are different specialties that tend to get conflated by the general public.

Supervised Learning

Similarly in the Machine Learning domain, Supervised Learning is the process of posing specific problems, postulating answers, and asking the model the best way to justify those answers. The justification might be non-existent or bad, and thus the model is suggesting it's not a feasible postulate. Or it might collect data from a wide variety of places to support the conclusion through its set of built int rules.

Frequently Supervised Learning is also the process of interactive approaches. The Machine Learning model will run some analysis for a bit and then at a set point it will come back and ask for some guidance. This largely takes the form of 'don't follow this path and further and please add this path to consider'. That is a little to anthropomorphic a description, but it's largely valid. There's no agency here, just an iterative set of evaluations through a set of data.

Sometimes, we also consider it to be supervised learning when we use real-time interactive data displays like OLAP or other analytics tools to 'drill down' into the data. The Supervised Learning model knows how to manipulate the data in a way to display what the user is asking for or to visualize the data in an effective or pleasing manner. This field is a bit of a stretch to be called AI or Machine Learning as it's very deterministic and mechanical. But it is frequently lumped in, categorically.

Mixed Methods

So, broadly, unsupervised learning is for finding every single statistically significant conclusion and asking a human to narrow it down to the ones that matter (for example, it may report that there is a statistically significant change that someone who buys one brand of ketchup does not buy another brand of ketchup). Supervised Learning is a human presenting a set of conclusions and asking the AI to find reasons why it those conclusions might matter.

We tend to prefer to alternate between the two models, producing questions and statistical findings without provenance. And then as those questions are structured and the findings provided with post hoc theorization and epistemic scaffolding.

AI Example

So, what non-statistics models exist?

The most common one is Natural Language Processing. This is a bit of a misnomer, as modern GPTs and other statistically-driven models have coopted the phrase to describe what their models are doing. However, NLP purists claim that is a retcon and NLP is never going to be statistically based. Instead, the entirety of their approach needs to be context free grammar based (or a similar deterministic computational theory) and that provability at each step of the ruleset is paramount. The most common tool right now is called NLP++ and is available in several IDEs for higher level a languages.

Most Meta-heuristics can be developed on their own. For example, protein folding tends to follow a genetic algorithm approach as it tries to find proteins that fit into certain patterns or biochemical reactions that will produce certain proteins.

Rules-based AI is literally the system of creating a set of rules that simulate behavior. For example, any RPG player can tell you that guards are clever enough to notice you, but dumb enough to think it was the wind the moment you go back into stealth. No matter how complex the ruleset, they'll still be relatively simplistic. But, it's much easier to error check a rules-based AI that has markers than a neural network that just does things (and while that's getting better, it's still not comparable in robustness).

Constraint Satisfaction Problems are a way of designing optimal solutions. This is frequently put together with other AI tools in order to generate a set of inputs and constraints that are strict or loose enough to be effective. This is also one of the areas that most closely aligns with many other social science fields.

K-Means, Decision Trees and Random Forests. These are statistical models, but they're much more heavily invested in the descriptive statistics side than the inferential or predictive. Essentially, given a group of data, their goal is to iteratively divide that data into smaller and smaller subsets until it can create a set of mathematical rules to determine how to get there. It's, for example, a classic way of weather prediction based on historical trends or bank loan approval or disapproval. The latter is especially important since the banks need to follow federal non-discrimination laws and need to be able to justify decisions.

The point of these models is that it's good to have a backup plan. AI right now is heavily dominated by Neural Networks and GPTs. But that shouldn't stop you from being sure that you understand how alternative models work. This lets you apply lessons from those other models to GPTs, and vice versa. It also gives you multiple areas of expertise in case the market becomes saturated. Career-wise, do not run in the same direction as everyone else without having an alternative in case you get outcompeted.

CHAPTER 16 - THE CAKE IS A 301 REDIRECT ERROR

In the broader world of information security, we break security threats down into three categories. Natural disasters are the sort of environmental issues that you can't avoid, you can only mitigate. Benign users are authorized users who may only be making a 12-egg Omelette with 11 eggs. And Malicious users are the people who started the trend of pineapple on pizza. Between these three, we have a good framework from which to consider how different threat vectors operate at the high level.

This is a good model for retaining and pulling out as you're trying to spot check an application. It doesn't replace an exhaustive and targeted security analysis. It is, however, the sort of generalized ideas that you can keep in mind as you progress.

The secret is, you don't need to be perfect in your security practices. In fact, you can't be perfect. But through consistent (not constant) attention to detail, you can avoid mistakes or bad practices that would otherwise cripple your implementation. This applies at the low level as well as the high level, but how the categories are applied change.

Quick Security Framework

- Natural Disasters – When we talk natural disaster, we're typically talking about abnormal, amoral activities. These are unavoidable strictures of operation that need to be mitigated or avoided.
- Benign Users – When we talk benign users, we're really more talking about routine operations that are poorly controlled or have legitimate unhandled edge cases. Remember that it's hard to make something foolproof, because fools are so darn clever.
- Malicious Users – When we're talking malicious users, we're talking about operations that are happening outside of the norm or outside of the scope of the program.

Let's dive into these and then come back to take a look at what needs to be considered in a more in depth fashion.

Natural Disasters

At the high level, when we talk about natural disasters, we're talking about routine situations that there's no

way to plan for or avoid. For example if you operate in Florida, you'll need a hurricane mitigation plan. Will a given hurricane hit? Maybe, maybe not. Will a hurricane hit where you are some point? Definitely. So addressing natural disasters involves producing structural systems to respond or mitigate the damage that a hurricane can cause (for example, non-windowed server rooms) or adjusting locations (for example, having an offsite backup or moving routine operations to a hardened, underground former salt mine).

The high level is a bit squishy in context and deals with everything from lightning strikes and political unrest to hiring a cleaning crew who isn't informed that they shouldn't plug a vacuum into the main server room circuit. But at the low level, those concerns are so far outside of the purview of algorithm design that we can't even begin to address them. When the entirety of our data structures are handling whether or not there is a positive or a negative bit in the highest order bit, radar alerts are a bit out of scope.

So, instead, we'll focus on the mathematical framing. This is your first check for secure programming. It's not enough to just ask, we need a checklist! You should check for these things and explicitly ask these questions. Some methodologies will insist that you note an answer explicitly somewhere in a log or documentation. These are questions you ask for each line (it's a filter, so it goes quickly)

- Do you use an array in the line?
  - Do you have a max array length available?
  - Is there a different array bound, and are you checking it or constraining each time you access the index?
  - Are you iterating over the array? Does the iterator value get adjusted each time?
- Are you using pointers?
  - Do you check for null values before you access any part of the pointer?
  - Do your reference/dereference operators appropriately align?
- Are you using any syntactical tricks to get around coding rules?
  - Is it necessary? Can you write within standard coding convention?
- Are you using any variables in this line?
  - Is there a chance that the variable overruns or underruns?
  - Is the variable bounded appropriately?
- Are you referencing external entities, such as enums, functions, or structs?
  - Will your code always have access to that entity?
  - Does the use of the entity align with both the bounding and intent of the entity?
  - Are the arguments passed to or from the entity correctly typed and scoped?

Remember that when we say 'chance' or 'is it possible', we're not talking about intention. We're talking about, is it possible, given any input, for this situation to occur. You should also cater any list to an application you're running. This is just the most common denominator list of concepts to address. But, for example, if you were doing inter-OS network programming, endianness might come into play. If you were doing a lot of bitwise

operators, be sure to create a check list for how your bitwise operations are bounding. If you use a lot of recursive functions (want secure code? Don't use recursive functions), are they being bounded by function depth rather than just data value? If you are running numerical methods, are your functions stable and if not, do you have markers to constrain them when they are not stable?

The point is, most if statements are cheap (within reason. A simple == or <= between two direct memory accesses and/or literals, for example). It will affect the performance of your code to liberally sprinkle them in the beginning of your functions. However, it is less impactful to your code than having your program crash because of an unnoticed improper sign bit. And tracking something like that down after the fact can be frustratingly tedious.

Benign Users

At a high level, a benign user issue occurs when normal system operation or ordinary user behavior unintentionally produces a bug. For example, automated tools may place multiple generated data files into ZIP archives. Other tools may then collect those archives and place them inside additional ZIP files, creating several layers of nested archives. A user might later rename files within those layers using more descriptive filenames—not that we are suggesting this recently happened. This may work correctly in one operating system or utility but fail in another. Although ZIP archives are intended to behave consistently, differences in operating-system path-length limits, software versions, and archive utilities can cause unexpected problems. In this case, the combined directory paths and filenames became too long for one of the tools or operating systems to process correctly. As a result, files that could originally be archived could no longer be extracted by the average user. Nobody intentionally did anything wrong. The problem resulted from a combination of normal actions, environmental differences, and software limitations.

Other examples are forgotten passwords, inputs that are the wrong character set, or typing values into the wrong input box. The average user is intending to use the program wrong, but they are a security threat through their own lack of expertise (this is not disparaging, developers are sometimes the inexpert users in other contexts).

At the low level, what this means is that even if everything is mathematically sound, we need to check for content. We're, essentially, moving up the stack. Instead of just form, we're looking for the function of the data that's held in the data structures. This means examining the content. But we're asking the same questions:

- Does the function accept arguments from other functions?
    - Are the bounds of those values guaranteed by the passing function?
    - If not, are the values filtered here?
- Does the program accept input from the user or an external data file?
    - Are the contents of the input filtered for format? (For example, saved as a 7-bit word, but read as an

8-bit word)
    - Are the contents of the input filtered for content? (For example, prohibited words or values)
    - Are the contents of the input filtered for length? (For example, data values that overrun the internal data type)
    - Are the contents of the input filtered for structure? (For example dangerous characters like a "<!–" in HTML or anything following a "\" in C)
- Does the program execute for the expected period of time?
    - Does it fail gracefully?
    - Does it shutdown appropriately?

There are additional contexts to consider. For example, a data-transformation tool may need to determine the maximum time required to convert an input into its output form and whether its size, time, or resource limits should be adjusted.

Malicious Users

Malicious users are the category that we give to the attempts by actors to create situations that are never intended. For example, what if someone could hypothetically hold a magical micro magnet up to the side of a device exactly right so that the data values in storage get adjusted to their appropriate target? Could they create a situation where they gain access to privileges or data that they are not supposed to have? A classic example is Captain Crunch, who got his handle by finding out that if you used a toy from inside the box of Captain Crunch cereal, it would perfectly mimic the audio tones that payphones used to determine whether or not a long-distance call had been paid for. This story is so dated now. Cereal boxes no longer contain toys. Pay phones are no longer pop culture relevant, long distance calls are already free, phone data is digitized and has attached meta data instead of encoded audio waveforms. But, it's still a story worth reading. In fact, anything by Kevin Mitnick is worth reading in the security field, particularly for non-security specialists.

At the upper layers, we're essentially concerned with people artificially creating environments in which any of the previous security questions are breached. It's difficult to create a framework to adjust this, and it's largely a question from after the algorithm writer operates. But we can check a few things.

- Is there a chance that an external application might gain access to data allocated to this program?
    - Can it be checked or otherwise checksum?
    - Does the data need to be checked before each use or will the results already be outdated by the time the function runs?
- Does the function run in a deterministic manner with regard to the security system?
    - If so, is it possible to add a level of variation to prevent external users from understanding the internal structure of the memory stack?

    - If not, where is the internal directory that determines what values are where and is it vulnerable?
- Is there memory ownership (this is more for Rust and ownership model languages) that can be coopted?
    - If so, are we correctly de-allocating the memory in the correct place to prevent ownership ambiguity?

We debated whether it was worth it to include this section. But, it's important to ask the questions that you know to ask. For example, at one point there was a novel attack method called the Row Hammer attack where, reductively, assigning and then flipped large chunks of adjacent memory repeatedly allowed access to adjoining data. But, basic tools prevent that from being an issue. Essentially, internal code is not really something that can protect from external threats. It requires external tools. But, over time, you should start to look for programming patterns that might protect data in sensitive environments (Or course, it might not be an issue, for example on a air gapped computer in a secure area, if someone has access to act like a malicious user, there are physical security problems in play).

External Tools

There are three layers of external tool we want to use. Again, we'll present them as a layered approach, looking deeper in the content of the data structures:

Obfuscation

This layer of tool typically doesn't do much. We use obfuscation tools on, say, large JavaScript projects in a web environment. Because the user downloads the JavaScript file and has access to all of the code, we want to keep them from easily being able to manipulate the function calls. There are better tools, but this one should have almost zero impact on the code. Essentially, if you have a function called 'struct proprietaryStruct* SuperSecretFunctionMethodCall()', it will change it to 'struct aabde43e* fe9533eef()', but it will cascade these changes. Cascading, in this context means making sure that every other reference is the same.

Essentially, this is what compilers already do. When the code is tokenized and ready to be compiled, all of the function names and other handles are stripped away and they're just left with internal representations in the byte code. So, an obfuscator just does this with the plain text versions of the code. It's relatively straightforward to de-obfuscate. All you need to do is painstakingly unravel all of the function calls and variable names to see what they do! But, again, as long as the handles are all internally and consistently the same as the readable code, there should never be a functional issue.

One example of this is how people produce mods for games that don't have mod support. This is particularly easy in java, since the java code can be decompiled and returned to what looks like obfuscated code and as the functions' purposes are learned, the names and the names of their associated variables can be

changed to something more readable. This is how early Minecraft mods were developed before they got semi-official assistance from Mojang.

Obfuscation Upgrade

Obfuscation is not a security measure. At best, it's like locking the door to a house. It says 'we would like you not to open this door'. But anyone with a $4 lockpicking set off of eBay can get through that, and anyone with a rock can get through the building-code mandated windows nearby. So, a lock isn't really 'security', it's a polite request to not actively try to circumvent the polite deferral. In the same way, obfuscation just makes it more difficult to trace the code, not impossible and is a polite imposition against doing so.

However, we can upgrade it a little.

So far, we've seen that data structures are just like arrays, but the data is of disparate types. Suppose we have a data structure like {10, 14}, and a data structure like:

```
struct c0043fedde
{
int 54cab4ec;
int 54cab5ed;
}
```

We could look at where the code is used and make a strong guess that the values are for an integer based coordinate struct. But, which comes first X or Y? If the obfuscation tool can change the order of X and Y, then it becomes harder to disambiguate some things. If one is a char* and the other is an int, it's fairly obvious which is which, but higher level tools can obfuscate that as well. The key is to be sure that it's cascading throughout the code consistently. This also makes it difficult to use a memory scanner to see what's there.

We can also adjust who it obfuscates for. For example, if it uses a different function name for every person who downloads it (that is, it obfuscates it differently for each person), it would be more difficult (still not impossible or even infeasible) since people would have a harder time talking about it. Depending on the code, however, it may not be a good idea to dynamically change the order of the struct attributes or memory stack allocations since that change will aid in understand what's happening. Good for other approaches, but not always obfuscation.

Lastly, we can add spurious data in there. For example, maybe we create a function that is never called, or we use a little extra memory for values that are changed periodically. It's a waste of computation, but that tradeoff may be attractive for organizations that want to double down on the obfuscation (with the next step being legal protections: "We went through all this trouble, and you still tried to get around our protections. That's a violation of the DMCA!").

This is all superficial changes. We're not actually adjusting how anything works, just what we call everything. So, we would call this superficial because it doesn't look at the content, just the label of the content.

Memory Address Randomization

The biggest tool we have in our security toolbox at the low level is memory address randomization. This is a little more in depth than the previous one. We start to affect how the code runs, because we're affecting where data gets allocated inside of the physical memory chip. Whereas optimization is just a coding tool, memory address randomization has to happen at the compiler level. In fact, in almost all current compilers, it's on by default. You would only turn ii off if you were writing code to a very specific embedded system and you needed different addresses to be located in specific places. Or, for example, you didn't want the memory randomly distributed because you know that certain physical components were more likely to suffer wear and so organizing the memory in a specific way prolonged longevity. But, broadly, we almost always want the memory address randomization enabled. It's so endemic that most processors and systems will run it with comparable speed to non-randomized.

So, how does it work? You've probably written a simple game before where you called for a random value, but the random value came back exactly the same way every time, so the game ran exactly the same. You either forgot or didn't know yet that you needed to seed the random value in order for it to produce different values.

Memory assignation is like that. when you start a new function, the memory stack gets allocated. And then the same variables get allocated in the same order every time. So, if you were running Bubble Sort (void bubblesort (int* arr, int n) at the default, then every time, the first 64 bits of the memory stack are going to be the pointer to arr. The next next 32 bits will be n. the next 32 bits will be the value of i, the next 32 bits will be j. And then depending on how you implement swap, it might change, but it will at least be internally consistent. The point is that this happens in the exact same way, every single time. You will always have the passed arguments be the first memory addresses and then the variables assigned in the order they're declared.

Similarly, in the entire program, the different functions and global variables are all allocated memory in the exact same order that they're declared (which is why you can declare the header of the function before defining the body, and get rid of 'function not declared' errors).

But, what if not. What if we could set it so that they were randomly declared in random memory spots throughout the RAM of the computer?

This bypasses one of the most common hacking attempts. Imagine an ATM, one of millions deployed that are exactly the same. Same memory spaces, same processor, same OS, same OS version even. It's feasible to think that getting ahold of one and monitoring how the memory works would allow one to know the static allocation of all of the others. And thus, using some hacking tool (the details don't matter, just whatever tool is common and effective), if we get even minimal elevated privileges on a random other ATM machine, we can access those pieces of memory to learn privileged information. For example, if the account number and authorization of the ATM is always stored in the exact same place, we could access it and use another tool to spoof the operations.

Again, randomization sidesteps this. It means that we can't just know where the data is by default. This is, again, not going to prevent any hacking attempts. What it does is slow the hacking down and raises the

skill and access level required to compromise the system. Presumably nobody's going to let someone loiter outside an ATM and hook a home-made device to the machine for 20 minutes without alerting someone (or the machine itself may be designed to alert someone).

This is a non-superficial version, because changing the memory addresses reaches a little deeper beyond just the labels of functions and data structures to adjust the actual operation of the code. It also requires a little higher level of vertical integration: the compiler has to be designed to handle this and sometimes the processor or BIOS has to be involved.

Other Compiler Tricks

There are a lot of really cool tools at the compiler level. These are really more in the compiler design arena, but because they affect how the algorithms will get compiled and you should always be aware of the tools available, we'll briefly cover several:

- Stack Canary – Every return address gets preceded by a custom value. If another bit of code tries to overwrite a buffer in order to get access, it will change the canary value and the calling function (that's receiving the return) will know that it can't trust returned value.
- Control Flow Integrity – We didn't cover it, because it's much more advanced C than we needed, but we can create pointers to, say, a function call. And in this case, we send a quick check to be sure that the called function (that is referenced via pointer, remember) is the correct function.
- Memory Sanitization – This adds a bit of a concierge function that monitors memory access and will sound an alarm if buffer overflows are enacted (for example, an array out of bounds).
- Position-Independent Execution – This is another compiler function that should always be enacted. The others might have reasons not to, but this one just makes all pointers relative to a fixed instruction pointer, rather than absolute memory addresses. This means that the operating system doesn't need to worry about rearranging where the whole program stack gets moved to, but can break it up and vary the storage locations. It also generally helps it play nicely with higher order operations like the operating system.

Sandboxing

When we say sandboxing, we typically need a high level of integration. For example, java runs a specific JVM program that monitors everything that is occurring. With C, we don't tend to have that. But we can add libraries that dd a certain level of sandboxing. To a certain extent, that defeats the purpose of programming in C (the low-level optimization). Broadly, for software sandboxes, you should probably rely on languages that have one by default.

However, that's not to say that the operating system, the bios, or the hardware doesn't have a sandboxing approach that is effective. Essentially, we can segment off some parts of the memory or other resources and a

lower level actor (the processor or the bios or the operating system, for example) will refuse access to code that originates outside of the original tool.

Of course, the ultimate level of vertical integration is when humans get involved. For the highest level of security, we start talking about air-gapped computers and the physical security surrounding them. This would be for highly sensitive data processing or for research areas. If you need a keycard to get into a physical area, chances are that the computers inside of it should disconnected from the network (or at least, there should be a class of computers that are not networked).

Comments, Documentation, and Communication

We'd like to end the book about where we started: with a high level holistic look. The most effective security tool at your disposal is the comment. Other code reviewers can't tell if your code has an error or an intentional approach if you don't have comments describing what's intended. You should follow a comment and/or documentation methodology that consistently presents the intent and rationale behind the decisions made when coding.

This goes hand in hand with writing code that follows standard coding conventions. You should not be writing rogue code that just makes sense to you. Vibe coding is not a sufficient level of rigor for production code. And personal style is not really what should be involved in the code side (although absolutely in the communication and content side). Instead, the key is consistency, readability, documentation. Those are the most effective security tools at the low level or systems programming level.

Lastly, this all means that you should never neglect your communications skills. Every single hiring manager has the same goal. They want to hire members of a team. And, in the general case, team members need to be good communicators. This means not only do you need to practice but you need to develop a portfolio (even if that's just a set of anecdotes) demonstrating, for example, how you solved a security-related bug through your communication with a teammate.

Practical Application

One of the most effective practical tasks in cybersecurity is to fail securely. There are two, brief, analogies to use here. In a prison, who will get their way? The prison guards or the prisoners themselves? In terms of systematic issues, the prison guards will prevail, because they've got a lot of tools and procedures on their side. But, when it comes to capitalizing on edge case cracks in those systems, the prisoners will always win. Why? Because the prisoners need to win once in order to win, but the prison guards need to win every time in order for the prisoner to lose. Over time (same as Big O, as we extend the practice out to infinity), there is no win condition for the guards and the prisoners will always have won at least once. This is why every prison guard quickly learns to keep at least one spare change of clothes at work.

Similarly, we need to ask how a good bank vault works. Typically, a bank vault has three goals. The first is to delay an intruder. At some point humans had to make the bank vault using human tools and human machinery. So then, human tools and human machinery will always be able to get back in eventually. The second is to be visibly broken. Nobody should use a bank vault that can be broken into non-destructively. If someone breaks into a bank vault, there are explosion marks, or cut marks, or drill marks so that the next person to see it will know 'oh, someone was in here'. It's the real reason to use a residential deadbolt: so that it's obvious when most people try to bypass the lock (bonus goal: attempted break-ins will leave a mark, not just successful ones). Thirdly, the bank vault needs to be designed to notify someone. In the modern environment, that might be as simple as a silent alarm when vibrations reach a certain level or motion detectors are tripped after hours, etc. These three goals can be considered to compromise security design: Delay, Display, and Deploy.

In the same way, our goals for cybersecurity is the same. We cannot control where our code gets used. It is always a valid strategy for the bad guys to appropriate our computer and run it in their own environment. They only need to crack our security measures once, whereas we need to succeed against their every attempt to be inviolable. Similarly, our code should be designed to: delay an malicious user (obfuscation and randomization), display them (canary values and control flow integrity), and deploy (for example, some Control Flow Guard algorithms will immediately quit the program when it displays a security event). The latter is more common with higher level systems that are much more comprehensive like the security module of an ERP.

When designing programs, these are the things to keep in mind. At every step of the design process, the question should be asked 'how can we delay an malicious user, how can we display compromises, and how can we notify someone?' Even at the low level, you can make log entries about oddities for later review. This is a secure programming practice, but it will also help you identify weak points of your code. The point is that we are going to fail when someone wants to break our code, but our goal is to fail securely.

AI Example

Let's end the text with a single interesting algorithm. One of the cardinal sins that developers can make is to have both Alpha and Beta errors (or Type I and Type II errors). Users can adapt to knowing that false positives might exist, or that false negatives exist. But it's very difficult to deal with both 'it might not have found all the solutions', and also 'it might have made up some of the answers'. Consider a classic AI tool (using Naive Bayesian classifiers, most often): the Spam Filter. If it blocks some of the spam, but you still sometimes get spam, but sometimes it also lets legitimate e-mails go to the junk folder, what's the point of using it? There are no perfect spam filters (although ones trained on large data sets are getting pretty good), since it's a probabilistic engine. But, if the choice is to completely optimize for accuracy or to have a lightly lower accuracy while completely avoiding one of the two error types, then the latter is preferable.

So, a bloom filter is one tool that we can use in AI to completely avoid false negatives with the tradeoff is that sometimes we have false positives (this philosophy is not the singular reason, but a contributing reason why AIs hallucinate by inventing truths instead of refusing to acknowledge the existence of things that exist).

```
// Define 3 hash functions
bool bloom[MAX_BLOOM]

void insert (struct node* value)
{
     bloom[hash1(value)] = true;
     bloom[hash2(value)] = true;
     bloom[hash3(value)] = true;
}
bool query (struct node* value)
{
     if (bloom[hash1(value)] == false) return false;
     if (bloom[hash2(value)] == false) return false;
     if (bloom[hash3(value)] == false) return false;
     return true;
}
```

So, we can see that we're, essentially, increasing the runtime of insert and read by 3, but since all three values have to agree in order for the value to be acknowledged as present, it's a good check. It does bring up several questions. What if it was just that at least 2 of the three had to agree? What if we only used 2 hash functions or 4 hash functions? How would either of those change the utility of the bloom filter? Then, how could we adjust it so that it never has a false positive, but it might have false negatives? Can you create a standardized approach that allows for either a preference for false positives or negatives to be set?

Lastly, can you explicitly note the tradeoffs that these variations present? If so, can you cater those tradeoffs to low level repetitive calls, such as with AI training?

This is where you can add appendices or other back matter.