
Demystifying The Coding Interview

Strategies to help you get the job you want

Dave Kanter

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Demystifying The Coding Interview

by Dave Kanter

Copyright © FILL IN YEAR O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Shira Evans and Louise Corrigan

Production Editor: FILL IN PRODUCTION EDITOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2025: First Edition

Revision History for the First Edition

YYYY-MM-DD: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098173654> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Demystifying The Coding Interview, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author(s) and do not represent the publisher's views. While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17365-4

[FILL IN]

Table of Contents

Preface.....	ix
1. Preparing for the Coding Interview.....	13
How to use this Book	13
Study Plan	14
During the Interview	15
Backup Plan	15
The Importance of Doing the Work	15
Developing Intuition	16
2. Big O Notation.....	19
Big O Introduction	19
What is Big O Notation?	20
The Shapes of Big O	20
$O(1)$	21
$O(n)$	22
$O(n^2)$	23
$O(\log n)$	24
All of the shapes together	25
Discussion of Big-O	26
Big O is a Limit	26
Operations	26
Constant Time	28
Logarithmic Time ($\log n$, $n \log n$)	28
n^2 , n^3	28
Space	29
Best, and Worst Case Complexities	29
Average Case (Big Theta (θ) Complexity)	30

Building Intuition: How is Big O used?	30
Example Questions	30
Refactoring for Big O	30
“What Big O is This?”	30
“Give an example of this Big O”	30
3. Strings.....	31
Strings	31
Breaking Down a String	31
Accessing a String	33
Typecasting	34
Built-in String Functions	34
Example Questions	37
Sliding Windows	38
Example Questions	38
Array Terminology	39
Array Numbering	41
Arrays and For Loops	41
Modifying (or not Modifying) Arrays	41
Basic Array Functions	42
Array Copying	44
JSON (JavaScript Object Notation)	44
Array Searching, Sorting, and Manipulation	44
Sorting	44
Searching	44
Sample Questions	45
Array Rotation	45
Permutations	46
4. Linked Lists.....	47
Linked Lists	47
Linked List Definition	47
Singly Linked List	47
Doubly Linked List	47
Developing Intuition: What are Linked Lists Used For?	47
Example Questions	47
Find	48
Insert	48
Delete	48
Reversing a linked list	48
Item Swap	48
Merging linked lists	48

5. Stacks and Queues.....	49
Stacks and Queues	49
FIFO v. LIFO	49
Stack Definition	49
Queue Definition	49
Developing Intuition: What are stacks and queues used for?	50
Example Questions	50
Balanced parentheses	50
Find the second smallest item	50
Implement x using a stack	50
6. Heaps.....	51
Heap Definition	51
Developing Intuition: What are heaps used for?	51
Max heaps and min heaps	51
Example questions	51
Is heap a max or min heap?	51
Convert max heap to min heap	51
7. Hashes.....	53
Hash definition	54
Building Intuition: What are hashes used for?	54
Reasoning about Hashes	54
Real world example	54
Hashes and encryption	54
Salts	54
Example Questions	54
Word Count	54
Palindromes II	54
Anagrams	54
etc...	54
Using AI to Study for Hash Problems	54
8. Trees.....	55
Trees Definition	56
Building Intuition: What are trees used for?	56
Trees and heaps	56
Reasoning about trees	56
Tree Operations	56
Insertion	56
Deletion	56
Balancing	56

Types of Trees	56
Binary Trees	56
Properties	56
Search	56
Sort	56
Other Tree Types	56
Red and Black Trees	56
AVL Trees	56
Tries	56
Other trees	56
Example Questions	56
Finding height of tree	56
Tree longest path	56
Balanced tree	56
Merging trees	56
9. Graphs.....	57
Graph Definition	58
Building Intuition: What are graphs used for?	58
Graph data structures	58
Arrays	58
Matrices	58
Reasoning about graphs	58
Depth First Graph	58
Breadth First Graph	58
Weighted Graph	58
Dijkstra's algorithm	58
Example Questions	58
Closest path	58
Furthest path	58
Knight walk	58
Flood fill	58
10. Functions and Recursion.....	59
Functions 101	59
You will be asked a question about functions	59
What is a function?	59
Building Intuition: Reasoning about functions	60
Functions basics	60
Function prototypes	60
Constructing functions	62
Receiving function data	62

Returning function data	62
Closures	62
Functions Advanced	62
The mystical “this”	62
Constructor functions	62
Generator functions	62
Function chaining	62
Side effects and functional programming	62
Currying	62
Higher Order Functions	62
Building Intuition: Functions Ideas	62
Functions and state	62
Function refactoring	62
Function composition	62
Example Questions	62
Swap two numbers in place	62
Refactor this function	62
“Program to an interface, not an implementation”	62
Recursion	62
Introduction to Recursion	62
The Elements of Recursion	62
Base Case	62
Recursive Case	62
Reasoning about Recursion	62
When you (might) have a recursive pattern	62
When is it time to use recursion	62
Don’t paint yourself into a corner!	62
Example Questions	62
Factorial	62
Fibonacci	62
Making Change	62
Reversing a String	62
11. Search and Sort.....	63
Search	64
What does “search” mean?	64
How is search used	64
Why is search important	64
Building Intuition: Reasoning about Search	64
Types of search	64
Linear search	64
Binary Search	64

Other types of search	64
Example Questions	64
Partitioning	64
Solve x problem using y search	64
Search optimization	64
Searching and data structures	64
Sort	64
What does “sort” mean?	64
How is sort used?	64
Why sort is important to understand	64
Building Intuition: Reasoning about Sort	64
Types of sort	64
Insertion sort	64
Bubble sort	64
Quick sort	64
Merge sort	64
Example Questions	64
Anagrams	64
Partitioning	64
Optimization	64
12. Dynamic Programming and Greedy Algorithms.....	65
Introduction to Dynamic Programming	66
Why Dynamic Programming Matters	66
Building Intuition: Reasoning about Dynamic Programming	66
Solving Problems using Dynamic Programming	66
Memoization	66
Recursion, revisited	66
Example problems	66
The Climbing Stair problem	66
The Knapsack problem	66
Longest common substring	66
13. “Parallel Thinking” Questions.....	67
Reasoning Questions	67
Active Listening Questions	67
Logic Questions	67
14. Using Generative AI to Study.....	69
Reasoning Questions	69
Active Listening Questions	69
Logic Questions	69

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/title_title.

If you have a technical question or a problem using the code examples, please send email to support@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://www.oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://www.oreilly.com/catalog/catalog_page*.

For news and information about our books and courses, visit *<https://oreilly.com>*.

Find us on LinkedIn: *<https://linkedin.com/company/oreilly-media>*

Watch us on YouTube: *<https://youtube.com/oreillymedia>*

Acknowledgments

Preparing for the Coding Interview

If you've looked for a job as a developer anytime over the past 10 years, you've almost certainly been asked to take a coding test. The very phrase, "coding test," strikes fear into the hearts of developers from Palo Alto to the Passaic. If this describes you, fear no longer. The book in your hands (or on your screen) is your guide to how to take and pass coding tests. With some time, effort, and energy this book will hopefully help you build the knowledge and skills needed to answer coding interview questions like a pro.

How to use this Book

One question I see asked all the time online goes something like this: "I've been studying LeetCode for a few weeks now, and I can't even answer the most basic questions." Should I quit? The reason for this usually turns out to be because the question asker has made the assumption that LeetCode is the place to begin. Except it isn't. Without a basic knowledge of data structures and algorithms — also known as "DSA," — you won't get far on LeetCode at all. It's like trying to swim against a strong ocean current. You'll make no headway and only tire yourself out.

This book is the DSA primer you might not have gotten if you don't have a computer science degree, or if you got one in the days before coding questions became required for most programming jobs. Coding boot camps, for instance, are a great way to gain some basic knowledge about how to get started as a developer, but they aren't necessarily focused on teaching the broader concepts covered by DSA. This book aims to bridge that gap.

Study Plan

My first job as a computer programmer began with a phone call on Friday afternoon. Would I be willing to come down on the following Tuesday and interview for a position with a newly formed tech company? I eagerly said yes, hung up the phone, and realized I had no idea how to do what they were hiring me for. So I went into the storeroom at work and “borrowed” the manual for the software I was expected to know how to use. I read the whole thing from cover-to-cover twice over the weekend, and somehow managed to string together a small demo that convinced the company to give me the job. I’ve been a programmer ever since.

My advice to you: Don’t do that.

You have to give yourself enough time to be able to not only read about DSA, but to sit down and write some code and try some things and hopefully really absorb the content this book has to offer.

To that end, I advise coming up with a plan that you can realistically accomplish. Maybe you already have some knowledge of DSA, and maybe it will only take you six weeks for the book to sink in. If you’re like me and maybe you like to spend more time with things in order to fully understand them then perhaps a schedule of 15 weeks is more appropriate. I can’t say. I don’t know you, and I can’t see you. But I do want to help you.

Sit down soon and maybe skim through this book one time. Take a moment to make an estimate of how long it will take to read it through. Maybe you don’t need to know everything in every chapter. Maybe you can already tell that some chapters will go quicker than others. Whatever you decide, write it down. This gives you a baseline for how long you can expect to take to finish.

Then build a plan around that longer time period. Will you put two hours a day into this book? 15 minutes on the bus ride to work? Marathon two-day sessions every weekend? Figure that part out and write it all into a calendar — preferably one that you can easily modify.

As you work through the first several chapters, check back in with this plan. Is it going faster or slower than you think? Maybe everything is going exactly as you planned — great! But if the time is not what you thought it was, go ahead and readjust your calendar to make a more realistic estimate.

The other thing that matters is consistency. However you plan to study, make sure you make it a habit.

After you’ve honed in on a realistic assessment, you’ll have some way to track your progress. You’ll also know when you can start looking for work, and when you can

plan to attend an interview and be effective. The more diligent and patient and careful you are in your planning, the more likely you are to reach your goal.

If this is a Friday and you have the interview on Tuesday, good luck!

During the Interview

There unfortunately is not a single way in which coding tests are administered by companies. What you are asked can change from company-to-company, and even from interviewer to interviewer. You might be put into a room and asked to solve three questions on your own in an hour, or the interviewer might say “Let’s solve a problem together” and take you through a team exercise.

If it’s just you and a coding test and a clock then spend a moment thinking about how to maximize your time. The questions will be of different difficulties. Oftentimes you will not be able to answer all of the questions in the time given. Do you feel more comfortable answering several hard questions or a single easy one? If it’s a group exercise, would you like to spend more time talking or more time coding?

There are a lot of resources online about how to present yourself in interviews from how to write a resume to how to dress to how to speak. Spend some time looking at these so you’re not surprised by what you find once you get into the interview room.

Backup Plan

Things don’t always go the way we expect, even (especially) when they seem like they will. The odds are, you won’t get the job. Expect it. You might know exactly why you didn’t get the job or you might have no idea. Don’t take it personally, it really doesn’t matter and likely has little to do with you as an individual person. The only thing you can do is regroup, figure out how to improve, and figure out what to do differently the next time. After you leave the interview, take a moment and reflect. Write down the questions if you can remember them. Go home and look up anything you didn’t understand. Fortify yourself for the next time. The re-plan, re-apply, and expect to do better the next time!

The Importance of Doing the Work

There is no learning without doing. You can read this book many times over but unless you have a photographic memory, you will have a hard time learning data structures and algorithms without going through the exercises and trying things out for yourself.

It’s incredibly important that you sit down in front of a computer and try the things you need to learn from this book. At the end of most chapters I will give ideas for

ways to expand on the concepts that have been covered on your own. As someone who always skips over these kinds of things, I'm asking you not to skip over these things. Spend some time with the ideas. If they're too easy, find a way to modify them to suit your skill level, or find a LeetCode problem that utilizes them. If they're too hard, break them down into smaller problems that you can solve — an idea I'll come back to more than once in this book.

I also suggest you “grow before you know,” which means working on problems that seem a little more challenging than you can handle. The more you push yourself, the faster you'll learn. It's an uncomfortable feeling, and I suggest you get used to it!

Developing Intuition

When I first decided to learn math as an adult, outside of school, I was given some really good advice by a very smart person. She told me not to worry so much about the specifics of equations, but instead to try to understand what the equation is trying to do. $y = 1/x$, for instance, must describe a limit since x can never reach zero because anything over zero is undefined. I can see that graph in my head. If I can't see it exactly, I can understand the basic idea it's trying to convey.

x is a number that will get smaller and smaller and smaller but will never hit zero, so it describes a curve that glides along the y -axis, peaking at one, before it turns to never reach zero along the x -axis. I'm not a human calculator, I just found a way to understand what anything over zero must represent. It forms a picture in my mind that I can understand.

I tell my students, “Don't start coding before you understand what you're trying to do.” Because if you don't you're going to be just like the x in that graph. You can push yourself further and further and further along an axis, but you'll never reach it.

If instead you stop for a moment and try to take a concept and put it into words that you can understand, then you'll be more prepared to see when it's useful in a coding interview. If you pause and try to really unpack what it's saying and what it's trying to do, you won't have to memorize it because you can simply re-create it from your understanding.

The several sections in this book that begin with the heading “Developing Intuition” are written to get you to come to this understanding. Take some extra time with these. Putting in a little effort in the short term might save you lots of time in the long run because you won't have to struggle to understand what you're being asked.

“I see that x at the bottom of the fraction, so I know x can never be zero. Because it's in the denominator of the fraction I know that the bigger the number gets, the smaller the equation gets. $1/1$ is pretty small, $1/10,000$ is even smaller and $1/1,000,000,000$ is tiny. Aha! This equation must describe a limit approaching zero!”

I didn't look that up, I didn't memorize that, I only have a general idea of what the resulting graph might look like. I just used my understanding and a little bit of intuition to explain that example, and I'm pretty sure I'm right. The more you can do that with the concepts in this book, the sooner you'll master the ideas needed to prevail.

Big O Notation

After reading this chapter the reader should understand what Big O notation is, why it's important, how to answer questions about Big O notation, and some tips for making better guesses.

Big O Introduction

I'm going to start this chapter with a confession: I hated math growing up. I once gleefully told my 12th-grade precalculus teacher that I would never need to know one single thing he taught me. Mr. Wittekind, on the off chance you're reading this book right now, you were right and I was wrong.

In my biased opinion, and with a further apology to Mr. Wittekind, I think this has a lot to do with the way math is taught. Children are asked to memorize a bunch of formulas and concepts, without much of an idea of why they're important or even what they are. If you ask most American high school students about algebra, they can likely tell you something vague about equations, quadratics, or slopes of lines. If you ask most American high school students "What *is* algebra?" I'm guessing they won't know.

Unlike computer programming, it can be hard to make mathematical concepts immediately appreciable and understandable. I can teach you about a for loop, and then show you a cool example where you can make a computer say your name ten times. I can teach you $y = mx + b$ and you might not understand why that's important and useful to know for quite some time, if ever. It's certainly hard to see why immediately.

No wonder I was always drawn much more to computer programming than mathematics.

Now that that's out of the way, there is of course a strong link between mathematics and computer science. In many cases, the latest discoveries of one have been found thanks to ideas or methods derived from the other. This doesn't mean you have to be a math whiz to be a great computer programmer, but a little bit of knowledge of math can certainly never hurt you.

Big O Notation is one of those little bits of math. You don't need to solve long sets of problems or memorize lists of equations to understand and use Big O. There are some general principles you have to understand, and once you understand them you can add the Big O tool to your arsenal.

Understanding Big O is a vital skill for getting through coding interviews, so this book is going to start with it. Rather than taking some crazy deep dive into every equation imaginable, I'm going to give you a few "Rules of Thumb" to keep in mind for a coding interview.

What is Big O Notation?

Big O (short for "Big omega" or the Greek letter Ω) is a topic that can make developers tremble in their stylish sneakers. While the phrase "Big O" itself means something specific it is also sometimes used in a general way that means "algorithmic analysis."

This is exactly what Big O is for, analyzing the speed of algorithms. The most important idea behind Big O and related terms is that they measure speed in a general and relative, and not specific, way. What "general" means is that Big O is great for comparing algorithmic approaches or finding algorithmic optimizations.

What Big O is not used for is measuring exact run times. If you need to do that, programming languages and libraries contain execution and runtime timers that can tell you exactly how long something takes. Where most people get stuck is on the idea that Big O is a precise way to measure speed. Think of Big O as more of a benchmark. Big O is a bar that can be set to see if an algorithm can be improved.

The Shapes of Big O

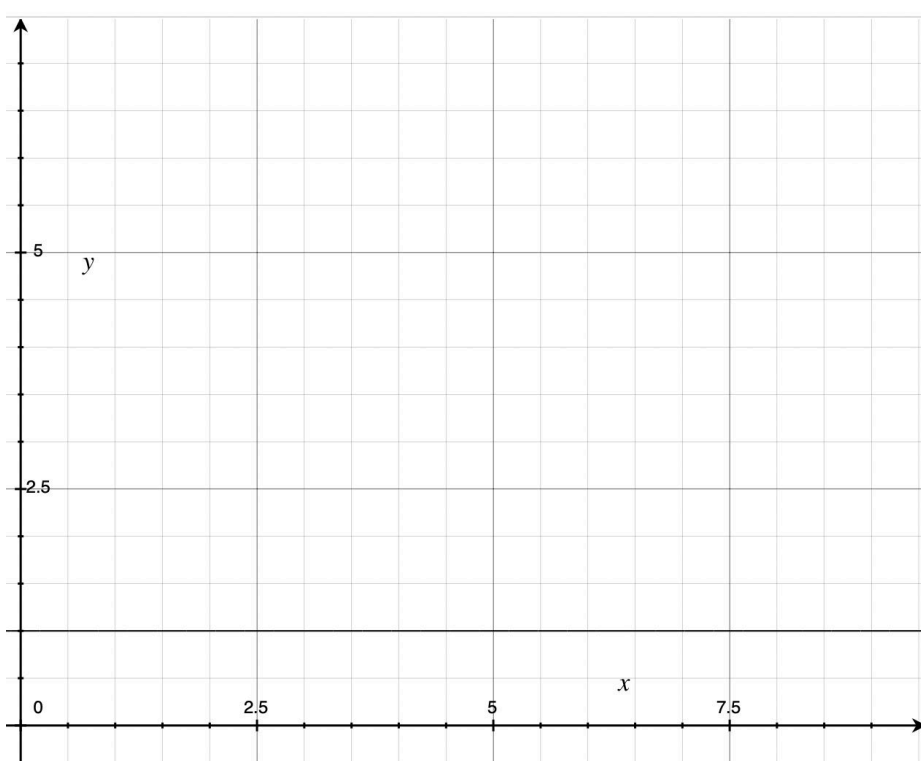
Let's dive in. There are five Big O "shapes" you need to know. By "shapes" I mean slopes of a line on a graph, but I think it's easier just to consider them as shapes. If there's one thing to memorize in this book, it's these shapes — especially as they relate to one another. Yes, there are more. Yes, it's more complicated than what I'm about to say. But if you can understand these five principles you've gained most of what you need to know to get through a coding interview.

I can't promise you're not going to be asked to explain how to derive Big O equations during your coding interview. But it's unlikely. What you might be asked is to

describe the efficiency of your algorithm in terms of Big O, and whether or not it can be improved.

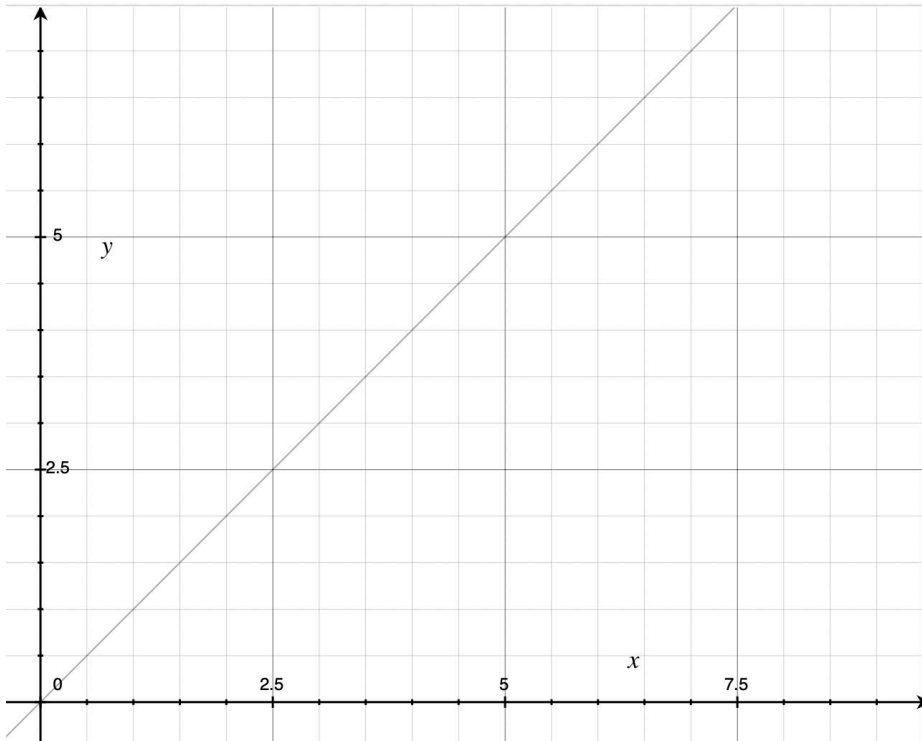
In each of these graphs, the Y or vertical axis is the number of inputs. The X or horizontal axis is the number of operations that need to be performed to accomplish a given task. Usually, it's not possible to decrease the number of inputs, so what makes an algorithm more or less efficient in terms of Big O is a reduction in the number of operations. As such, “flatter” graphs are generally preferred to “steeper” ones, as described below.

$O(1)$



With $O(1)$, no matter how many inputs you have, there's only one operation that needs to happen. Generally speaking, it's hard to beat this approach. Most algorithms don't run with this kind of efficiency, but some do, and they'll be discussed below. Notice that the shape of this graph is flat, and you can't get more efficient than that.

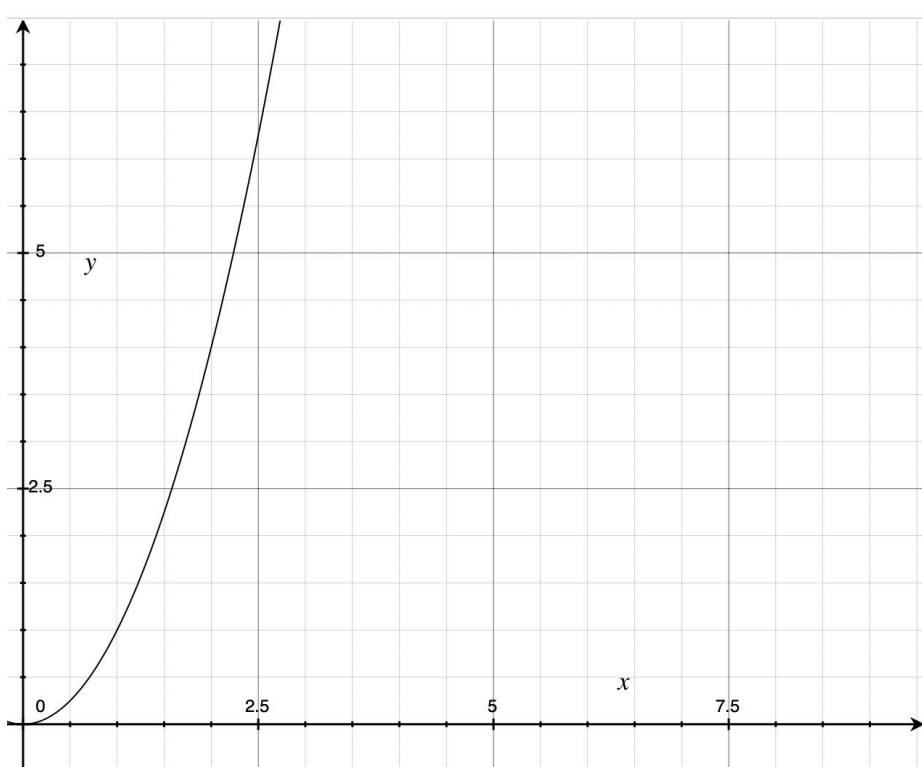
$O(n)$



Notice the shape of this graph. The number of inputs (y-axis) equals the number of operations (x-axis). An example of this is printing a list of names from a database. If there are 100 names, there will be 100 calls to `print()`. If there are a thousand names, there will be a thousand calls.

$O(n)$ can be very efficient if there are only a few inputs. That's a great place to start, but when you discuss Big O, you must always consider it at scale. If there are 100 inputs or one thousand, as mentioned above, it still might be more efficient than other approaches. But what if there are 500,000, or a million, or a trillion? The efficiencies of $O(n)$ might diminish considerably compared to other approaches.

$O(n^2)$



$O(n^2)$ is the least efficient of all of the “shapes” discussed in this section. $O(n^2)$ (also called “quadratic”) means that every operation must be done *number of inputs* multiplied by *number of inputs* times. If you have 100 inputs, it will take 100^2 or 10,000 operations to complete the task. Notice that the shape of this graph takes off quickly and goes nearly vertical. The addition of even a small number of inputs will quickly cause the number of operations to skyrocket. An example of this happening is in nested for loops. Keep in mind that it gets worse than $O(n^2)$. Poorly built algorithms can take on $O(n^3)$ or $O(n^4)$ or even worse, so try to avoid these approaches if you can. There are even worse metrics, including $O(n^n)$ and $O(n!)$. We’ll discuss these more later in this chapter and point out examples in the algorithms sections as they arise.

$O(\log n)$

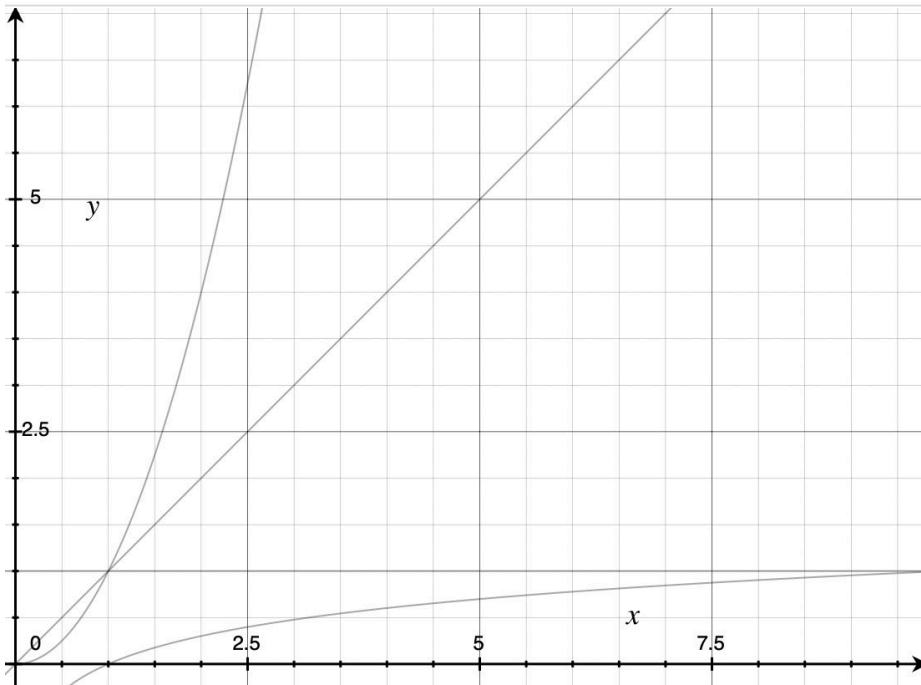


If you don't remember logarithms from high school, a logarithm is the number something would have to be raised to in order to come up with a given number. To put that in a hopefully less confusing way, think about it like this: what would 2 have to be raised to in order to get 8? Since 2^3 is 8 the answer is 3 times, and so $\log_2(8) = 3$. There's more to logarithms than this, of course, but right now we're focusing on the shapes. Notice how slowly $O(\log n)$ grows, flattens as it grows larger, and how for a small number of inputs it can be even more efficient than $O(1)$.

$O(\log n)$ arises from algorithms that split the relevant numbers of inputs under consideration by half, repeatedly. You'll soon see that this is the case with some searching and sorting algorithms, and also with binary trees. (In general if the word "binary" is involved in the description of an algorithm, it's $O(\log n)$.) This is a highly desirable efficiency and one that should be sought wherever possible.

Closely related to $O(\log n)$ is $O(n \log n)$, which is less desirable than $O(n)$ but certainly superior to $O(n^2)$.

All of the shapes together



Here are all of the shapes together. Take a moment and compare each shape to the others, and go back and re-read the description of any shape in isolation if you don't understand what it means. These shapes are going to form the basis of the rest of the discussion of Big O, so really take a moment here if you need it. If someone asks you “Which is more efficient: $O(n^2)$ or $O(\log n)$?” you should see the two shapes in your head and immediately know the answer.

Nothing builds understanding like doing. If you want to cement the relationship between the equations in your mind, find a graphing calculator application or website and graph these shapes yourself. Zoom in and out on the graph and the points at which the lines cross, and come up with situations in which one operation would be preferable to another. Which shapes are better for smaller or larger numbers of inputs?

Discussion of Big-O

I started with the shapes so that you could keep them in mind as we discuss Big O in depth, so that it's far less of an abstraction than what's in computer science textbooks. Now that you've seen the shapes, it's time to discuss in more detail what they mean.

Big O is a Limit

Big O is not about measuring the actual speed of an algorithm. For starters, that's not an easy thing to do universally because different computers come in different memory and chipset configurations.

Big O is a type of “asymptotic analysis,” which is a way to mathematically describe limiting behavior. “Limiting behavior” means the best or worst a process can possibly do. For instance, when you buy a car you're usually told what your expected gas mileage will be. This doesn't mean your car will always (or ever) get this exact gas mileage. The number given is usually the absolute best the car can be expected to do, under a very specific set of conditions. But depending on any number of factors — passenger weight, tire pressure, weather, road conditions — your car won't do as well as what the number promises. And no matter what, your car will not do better. So the number describes a “limit.”

Big O also describes a limit, but unlike the example above Big O actually describes the *worst* an algorithm can possibly do. If the graph describes the *best* an algorithm can possibly do, that's referred to as Big Omega or Big Ω notation. The average case is described as Big Theta, or Big Θ notation. This is not a book about asymptotic analysis, and there are a lot of them out there, so let's stick with Big O as a general term for algorithmic analysis and the five shapes we considered earlier.

To sum it up, if an algorithm is said to run in Big $O(n^2)$ time, that algorithm in its worst case will run worse than an algorithm running at Big $O(\log n)$ time. To think about it another way, an algorithm running at Big $O(\log n)$ time will *usually*, but not always, run better than an algorithm running at Big $O(n^2)$.

These descriptions should give you a sense that the idea behind Big O is to compare two approaches as generally “better” or “worse” in terms of expected performance.

Operations

In presenting the shapes to you I said that the Y-axis represented the number of inputs, while the X-axis represented the number of operations. It should be clear that the number of inputs is the number of pieces of data processed, but less clear is what is meant by the number of operations. More specifically, what exactly is an operation?

For the purposes of Big O, operations are the things that have to happen for the algorithm to execute. Because for the purposes of tech interviews Big O is about *trends* and not actual timing, think of an “operation” as a constant value. There could be three things happening, or four, or ten, but we’re more interested in how many times this *group* of things has to happen, than in the number of items in the group.

As an example, consider the multiplication table you may have learned in grade school:



Here’s a general example of how this might be accomplished. Ignore the fact that there are no headers or proper spacing or optimizations and just focus on the fact the code contains a function definition, two for loops, and two print statements. There’s nothing complicated here: the code runs through two for loops and for each iteration through the loop prints out the number in the “i * j” position, much like the multiplication table itself does. The multiplication table in this form can be considered a “two-dimensional” piece of data.

```
---
def multiply(limit):
    for i in range(1, limit + 1):
        for j in range(1, limit + 1):
            print(f"{i * j}", end=" ")
        print("\n")
---
```

How many operations is this? You could consider that the function definition is one, the two for loops are two, and the two print() statements are two resulting in 5 operations. But as far as Big O is concerned, the answer is it doesn’t matter. For the purposes of Big O there are two for loops, which means that whatever you pass into `limit`, the number of operations in this code is $limit^2$, or n^2 times. The running time is $O[n^2]$.

What if you make the following modification to the code to add a header and a notification that the program has finished? Does this change the number of Big O operations involved?

```
def multiply(limit): print("Multiplication Table from 1 to ", limit)
    for i in range(1, limit + 1):
        for j in range(1, limit + 1):
            print(f"{i * j}", end=" ")
        print("\n")
    print("Finished!")
---
```

If all you’re adding is two lines of code, Big O remains at $O(n^2)$. Why? Because as I mentioned earlier, Big O is about the *trend* of the code running time, and not about the running time specifically.

So why is this important? It's important because Big O allows us to talk about the *optimization* of code in terms of *overall* improvements to the running time of the algorithm. Again, because of the inherent differences between computers, the comparison between the running times of algorithms can be pretty meaningless. The `multiply()` function above is a very simple (I will not use the word “trivial” one single time in this book, other than right now) example that I picked to specifically illustrate $O(n^2)$, but as we go through this book we will see some algorithmic approaches that improve Big O in terms of either time or space complexity.

Pause and think: Can the multiplication example above be improved so it doesn't use two for loops? Can it be improved so it doesn't use for loops at all? Does this change the number of operations required to run this code?

Now that I've presented you with some introductory ideas regarding Big O, it's a good time to consider some further specifics.

Constant Time

“Constant Time,” or $O(1)$ means that the number of operations remains the same no matter how many inputs are passed to it. As such, this is a highly desired state of algorithmic optimization. It should come as no surprise to you, however, that it's an incredibly hard state to obtain, and for some problems may simply not be possible.

Looking once again at printing the multiplication table above, no matter how you refactor the `multiply()` function, it's going to have to run at $O(n^2)$. It has to in order to print the required numbers the required number of times. It cannot be optimized further.

But there will be examples in this book of starting with a “brute force” algorithm and optimizing that algorithm in order to flatten the curve. For instance, the Rabin-Karp algorithm discussed briefly in Chapter 3 changes a $O(n^2)$ algorithm into $O(1)$ time

Logarithmic Time ($\log n$, $n \log n$)

Logarithmic time usually describes algorithms that split the data set in half with each iteration. For example Binary Search algorithm, discussed in Chapter 12, works by splitting the data set in half every time loop through the data is performed. The height of binary trees, discussed in Chapter 9, can also be derived logarithmically. Most (but not all) of the time we're dealing with a base-2 logarithm, as is expected from a repeated reduction by half.

$$n^2, n^3$$

“Polynomial” time often occurs when a process is run through multiple loops, like in the Multiplication Table example above. If every loop through means every item

needs to be considered once (or more than once), you're in polynomial times. Of the five examples of Big O shapes given in this chapter, polynomial time is the most complex (and so the "steepest"), but there are even steeper curves including $O2^n$ and $O n!$. There will not be too many examples of such complex algorithms given in this book, but they're out there.

Space

Big O notation in this book will refer to the complexity of the running time of the algorithm, as has been discussed throughout this entire chapter. It is possible, however, to also discuss Big O in terms of the space required to run the algorithm. Does the algorithm have to make a copy (or multiple copies) of the data to run, or can it use the existing data "in place" without the need for additional copies?

At the beginning of my career the internet was awfully slow, storage memory was a lot more expensive, and "plug and play" hardware had yet to come along, and so it was cheaper and faster to mail our spiffy presentations out to customers on 3.5" floppy disks than to ask them to sit through an hours long download. The capacity of a 3.5" floppy disk is just under 2MB. So everything we did had to fit within that memory space.

You might be glad those days are gone, but for programmers of embedded systems, microcontrollers or IoT devices memory can still be a precious commodity. So algorithms that unnecessarily duplicate data in order to process it may need to be optimized to handle space as well as time complexity.

Don't be surprised if you go for an interview at a company that focuses on these types of products and you're asked a question about improving space complexity. Even if you don't plan to apply for these kinds of jobs, a 100MB web page is a bad practice that happens all too often because developers don't understand space complexity. Computer memory is a lot cheaper than it once was, but that doesn't mean it's not precious!

Best, and Worst Case Complexities

For the last part of this chapter, I would like to focus on some different-than-best-case, um, cases. I briefly mentioned these above and will now expand a little bit on these different boundaries. The changes in boundaries can come from improvements to algorithms, or by using entirely different, better or worse algorithms, as you would expect. But they can also come from changes to the data. For instance, the Quicksort algorithm has a best case (Big O) of $\log n$ but a worst case (Big Ω) of n^2 . The only difference between the two cases is whether or not the data is already sorted. Because of the way the Quicksort algorithm works, it actually runs the slowest *when the data*

is already sorted. This idea will be considered further when we examine Quicksort in Chapter 12.

Not all algorithms necessarily have a best and worst-case scenario. An algorithm that runs in constant time, for example, has a best case that is identical to its worst case.

Average Case (Big Theta (θ) Complexity)

This is also sometimes called the “exact case.” If you specifically measure every instance of every iteration of an algorithm and chart it, that could be considered an example of Big Theta. It would also result in a fairly crooked line that might represent a linear trend, or might be show that the operating system speed is being adversely affected by having too many browser windows open. Because we’re looking to understand general trends and not exact algorithmic analysis, in this book Big θ will mean “average.”

Building Intuition: How is Big O used?

In coding interviews, Big O will usually be used to test one of two things: Do you understand the complexity of the algorithm you’re looking at, and/or can you reduce the complexity of the algorithm you’re looking at?

For example, you might be given a problem that requires you to sort an array. As a first option, you might choose to use the Quicksort algorithm which, as discussed above, has a worst-case running time of n^2 if the data is already sorted.

The questions regarding the complexity of Quicksort could come up in a number of ways: Do you know the average (Big Theta) time of this algorithm? Do you know the best-case (Big Omega) running time of this algorithm? Do you know of an algorithm that runs faster than Quicksort? What has to change for the Quicksort to become more or less effective? Do you know why Quicksort runs in Big O when the array is already sorted?

Example Questions

Refactoring for Big O

“What Big O is This?”

“Give an example of this Big O”

While this book assumes you’ve had an introduction to computer programming, this chapter and the next will contain an exploration of some basic computer science concepts in order to create a baseline of understanding for the content to follow. If you come across something in this chapter that you already know feel free to skip ahead.

Strings

Strings are (of course) one of the most commonly used data structures in programming. The documentation of most programming languages starts with the ubiquitous “Hello World” program, which usually involves rendering a string.

In Python, for instance, all that’s required to render text to the console is a simple `print` statement:

```
print("Hello World")
```

What a great 10-character example! Or is it 11? 13? Something else? To understand how to manipulate strings, you must begin by understanding the way in which they’re constructed and represented by computer operating systems and programming languages.

Breaking Down a String

In most programming languages a string is nothing more than an array with a character at each position.



Arrays are covered in the next chapter, but let's begin by discussing what is meant by a "character." If you're already familiar with this subject, please feel free to skip this section.

Whenever you press a key on your keyboard, like I'm doing to write this book, a character comes out on the other end. Without going into too much detail on exactly what happens during this process, the electrical signal generated by the key is sent to the operating system, which then uses a character "lookup table" to look up the character represented by that signal. The two most commonly used lookup tables for computers that use Latin-character-based languages (like English) are ASCII and Unicode. Again, there are piles of documents written about these two systems and others, and if you're interested a quick web search will show you more than you ever needed to know about these two systems.

The positions of the characters in this table are direct descendants of the teletype systems that were widely in use in the middle part of the 20th Century. If you're watching a movie and you see a chattering machine spitting out the latest news or a top-secret message that needs to be seen immediately by the people in charge onto a sheet of paper one line at a time, that's a teletype machine. It might be hard to imagine, but not all that long ago computers were only able to output to printers and not screens.

The size of a character in most operating systems is based on a byte. A byte contains eight bits. A bit is a symbol that can contain a one or a zero. Since each bit can represent two states, the number of signals that can be represented by a given number n is 2^n , where n is the number of bits.

For instance, 2 bits can represent 2^2 , or 4 discrete states: [0, 0], [1, 0], [0, 1], and [1, 1]. Three bits can represent 2^3 or eight discrete states. 2^4 is 16 states, 2^5 is 32 states, etc. While there are leetCode questions that cover binary numbers, they're beyond the scope of this book. There are piles of information available that cover computer operating systems and binary mathematics, and if you have more of an interest in the subject I encourage you to look them up.

A byte, which contains eight bits, can represent 2^8 or 256 discrete states, and so most character systems are based on a representation of 256 characters. Usually, the first 128 of these characters are usually reserved for "regular" keys on a keyboard, while the second 128 are reserved for "special" characters. So when you press that *d* key on your keyboard, it sends a numeric symbol to the operating system which then looks up which character is represented by that number. In the case of an English keyboard, the lowercase *d* is represented by the decimal number 100. (This is not the same as the uppercase *D*, which is represented by the number 68.)

There's nothing special about the positions of these characters. If you grew up speaking a language other than English, you've probably used a keyboard where the

characters are in different positions than they are on an English keyboard. That same *d* key on a different keyboard might instead stand for “M”, or “7”, or “Ü”, or “ä”. It all depends on how the keys are mapped in that lookup table. The reason why should be clear: it’s easier to change a look-up table than it is to rebuild a keyboard!

So when computer programmers talk about “characters,” this is what is meant. A series of symbols mapped to a lookup table, that translates them to the corresponding electrical signal represented by a keyboard key. None of that is likely to come up as an interview question, but it’s a basic idea that it’s important to understand as a developer.

Another important thing to know is that not all characters are visible characters. The “space”, “tab”, “return”, and “enter” (not always the same as return!) characters — and others — are mapped to the lookup table the same as visible characters, and have to be accounted for when you’re parsing strings. These characters are sometimes called “escape” characters since they can only be represented in strings using a symbol preceded by the character “\”. A “tab”, “space”, and “newline” are meaningful symbols to many programming languages and so you can’t always just type them into a string. Instead you must type a tab character as “\t”. A return is “\r”, newline is “\n”, even a space must sometimes be escaped as “\ ” to mean a string as a character and not a symbol that a programming language needs to consider.

These are all things to keep in mind when you hear the word “character.” When you hear the phrase “array of characters” you should immediately think of a string, and vice-versa. And when you think of either you should picture that array as potentially containing the visible and non-visible characters that can be found in a table stored somewhere in the operating system.

Here’s another look at that image above one more time:



Accessing a String

As mentioned before, usually the first thing a programming student is shown how to do is generate output using some kind of variation of a “print” statement. You won’t be asked interview questions about that. What you might be asked questions about, however, is the ways in which you can manipulate strings to get the results you need to accomplish the task you want.

I recently was working for a client that needed data parsed from an Excel document to the web. Among many of the things complicating this task was that we on the programming team had no control over what was in the Excel document. This means

that many of the cells in Excel were “polluted” with data that kept our program from running properly.

For instance, many of the cells contained line returns, or the “\n” character. This interfered with our program, because when we wanted to put a piece of data into a specific place if we printed it with the line return it would knock all of the data following out of alignment. I used the Python Panda’s `strip()` command to get rid of these unwanted characters, but even this was limited in its effectiveness as it only considers items at the beginning and end of a string. The team eventually had to use some custom lambda functions to get the results we wanted. Parsing string data is absolutely something you will do more than once in your programming career.

Typecasting

Everyone knows that $2 + 2 = 4$, but while that’s fine for English or mathematics, that’s not necessarily the case at all when it comes to programming languages. For starters, “=” is the assignment operator, which means that $2 + 2 = 4$ will return an error. Additionally, do you as a programmer mean the *expression* $2 + 2 = 4$ or do you mean the *string* “ $2 + 2 = 4$ ”? Notice that one is in quotes and the other is not, because one is a string and the other is not.

Context is important in computer programming. Compilers really can’t be bothered trying to figure out what it is you, the programmer, are *trying* to do. They don’t care. They’re literal, and logical.

If you’re trying to evaluate whether $2 + 2 = 4$, the correct expression is:

```
2 + 2 == 4 True
```

Enter “2” + “2” == 4 however, and what seems true is now false.

```
>>> “2” + “2” == 4 False
```

The reason for this is that like most programming languages, Python treats strings and symbols separately. 2 without quotes is the “symbol” for 2. It means the number 2 which is exactly what you think it means. 2 fingers, 2 sisters, 2 dollars.

“2” is the string representation of 2, which doesn’t mean anything to Python except that it’s a string.

Built-in String Functions

Most programming languages contain a number of functions for manipulating strings. One of the more basic and widely used is the “concatenation operator,” which connects one string to another.

```
myVar = "Hello" + "world!"
```

The plus in this instance does not mean addition, it means concatenation, or joining together. Eagle-eyed readers may have noticed that this concatenation may have an unwanted result.

```
print(myVar) HelloWorld!
```

Oops! Where did the space go? There are multiple ways to account for it, including adding it to the first or second string in the concatenation, but the object here is to make you aware that you need to account for it when processing your data.

Processing strings is an important part of data science, and so knowing how to do it is a fundamental skill all programmers must possess. As such, you absolutely should know how at least the basics of string processing, and the built-in functions most languages contain. Built-in string functions are also a great place to start building up your programming interview skills, as knowing how to build each of the following functions from scratch will help you gain a lot of skill in string and character processing.

While these functions may differ in name from language to language, they're all available to you as a programmer interested in manipulating strings:

`length()` or `len()` or `sizeof()` These are different names for functions that return the length of the array that is passed to them. This is incredibly useful for processing strings because it allows your code to handle strings of whatever size is passed to it.

```
---
myVar === "Hello world!"
print(len(myVar))
12
---
```

`toUpperCase()` or `toLowerCase()`

These functions will change every character in a string either to upper or lower case, as the function name suggests. This can be invaluable for data entry.

`split()` This function will split an array element into pieces according to a delimiter. When the delimiter is left empty, every element of the array will be split into an element of its own. This function has different implementations, depending on the language, so examples of splitting strings into individual characters in different languages are shown below. The C++ standard libraries don't have a straightforward way to accomplish this task, and the easiest way to do this in Java uses a regular expression, so if you're focused on those languages you will have to either look up these implementations, or, better yet, build your skill by writing your own!

Python:

```

myString = "algorithms"
print(*myString)
# Prints ['a', 'l', 'g', 'o', 'r', 'i', 't', 'h', 'm', 's']

```

JavaScript:

```

console.log("algorithms".split(''))
// Prints ['a', 'l', 'g', 'o', 'r', 'i', 't', 'h', 'm', 's']

```

`splice()` This function allows you to choose a very specific part of a string, selected by an index.

```

---
slice()
slice(start)
slice(start, end)
slice(start, end, step)
---
```

In languages that implement `splice()`, the data is returned without modifying the original. This can be useful as a “stripping” operation, say to remove a line return and dollar sign from the beginning of every string in a group of strings. Python contains the most robust splicing function, allowing programs to choose by an index that can be passed in a number of useful ways.

Be careful when using these functions as they all work slightly differently, and the difference usually has to do with how the range of items is chosen. For example, in Python, you can use a “:” to select the range of the items you’re interested in. Arrays start at 0, and the second value in a splice parameter is the item *before* the one in the number.

```

---
>>> myVar = ["turtle", "gecko", "frog"]
>>> myVar[1:2] # select from position to before position 2
['gecko']
>>> myVar[:3] # select everything before position 3
['turtle', 'gecko', 'frog']
>>> myVar[:2] # select everything up until position two
['turtle', 'gecko']
>>> myVar[2:] # select everything after position 2
['frog']
>>>
---
```

There are more concepts that are applicable to strings that will be discussed in the upcoming section on arrays, including `slice()` and `reverse()`.



Modifying data

I will revisit the construction of some of these functions in the upcoming “Arrays” chapter, but hopefully this has helped you start to see the importance of these basic string manipulation problems and why they might come up in programming tests.

Example Questions

These next few example questions will give you a taste of what’s coming in the “Searching, Sorting, and Manipulation” part of this chapter. Try to solve the following problems without looking ahead to that section. Or, if you get really stuck, come back and solve these problems after reading the next section.

Finding a Character or Word in a String. *Create a function that takes a string and a character and returns the position of that character in the string. For instance, given the string “One tall house, one tall ladder” and the character “a”, the function would return 4. Notice that there are three “a”s in the string. How would you return the positions of all of them? Does your function account for capital and lower-case letters?*

Similarly, create a function that takes a string and returns the position of a word. Given the same string as above, find the position of the word “tall.” Again, as above, account for the fact that there is more than one instance of “tall” in the string. What if the word you’re given is part of a larger word? The word “add” can be found inside the word “ladder.” Does your algorithm work for the cases in which this is and isn’t a valid solution to the problem? Can you think of any other things that should be considered in solving this problem?

Reversing a String Without `reverse()`. *Write your own function to reverse a string. For instance, if passed the string “apple,” the function should return “elppa” Can you do the same for a sentence, turning “Look at the moon” into “moon the at look?” Again, can you think of any other conditions that should be considered?*

Palindromes. *A palindrome is a string of text that contains the same characters in the same order both forwards and backward, excluding spaces and punctuation. A simple example is the word “level”, but the sentence “Evade, Dave!” is also a palindrome because without the capitalization, comma, space, and exclamation point the letters are “evadedave” which are the same forwards and backwards. Write a function that determines whether or not a given string is a palindrome. If you get it working with the example given, search the internet for other examples of palindromes to test your function.*

- Consider adding coding questions to the end of sections **

Reversing. Reversing

Sliding Windows

This one comes up a lot, good basic technique to know.

Example Questions

Here are some example questions

Maximum and Minimum Element. Given an array, can you return the highest or lowest element according to a given criteria?

Example: What is the largest number in the array [7, 2, 8, 6, 5]?

Longest Substring Without Repeating Characters. Given a string *s*, find the length of the longest substring without repeating characters.

Example 1:

Input: *s* = "abcabcb" Output: 3 Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: *s* = "bbbb" Output: 1 Explanation: The answer is "b", with the length of 1.

Example 3:

Input: *s* = "pwwkew" Output: 3 Explanation: The answer is "wke", with the length of 3. Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Food for Further Thought. Look up the Rabin-Karp algorithm for finding a substring in a string. This is an algorithm you probably use all the time, like when you press CTRL-F to find text in a longer document. The Rabin-Karp algorithm works by converting the substring to a hash value which allows it to run in $O(1)$ time. Hashes will not be covered until Chapter 8, but try to begin thinking about Rabin-Karp now. Is it a suitable algorithm to be used in a coding interview? Why or why not?

Let's begin with the simplest linear data structure, the array. In less abstract languages like C++ or Java, arrays are manipulated directly from their addresses in computer memory by pointers. For our purposes, we will only consider arrays as being manipulated and addressed by the abstraction known as the array's "index." I mention this abstraction as something to keep in the back of your mind when you consider the original purpose of some of the data structures and operations you're about to encounter.

Levels of Abstraction

Programming languages can be categorized by levels (or layers) of abstraction, which means abstraction away from the underlying hardware and chipset that a computer uses to store data and instructions. "Low-level" languages, like assembly, are only one level of abstraction from the hardware that makes up a computer, and can be used to manipulate a computer's hardware directly using what is known as "machine code."

Low-level languages tend to make more sense to computers than humans, though, and it was realized long ago that abstractions were needed to connect human thinking and language to computer actions. "High-level" languages, like C++, take human-readable language and compile it into machine code instructions that can be used to manipulate a computer's chipset directly. There's a higher abstraction than this though. "Interpreted" languages, sometimes referred to as "scripting languages", contain an even higher level of abstraction, as these languages don't interact with the computer chipset at all. Scripting languages are instead fed to an application, like a web browser, that attempts to "interpret" their instructions into machine code. It's well worth your time to understand layers of abstraction as a programmer, and it goes far beyond what can be explained in a single side note.

Array Terminology

Arrays are close in equivalence to what is known in mathematics as a "set." A set is a list of items that have some relation to one another, that can be represented by a single notation. In strongly typed languages, arrays can only contain the same type of data, but in weakly typed languages, arrays can contain mixed data types. Arrays have some special terminology related to their specific use:

Element, index, set

<diagram of an array> We looked at a simple example of an array in the last chapter, with an image showing how an array is a string of characters.

GRAPHIC:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

In their original versions, and as is still the case in languages like C++, arrays worked hand-in-hand with data “pointers”, or reference objects that “point” to specific memory addresses. Once an array is created in C++, the programmer can “dereference” each address in memory to get the value stored there. Because a pointer points to a specific type of data, incrementing a pointer moves it in memory the length of that piece of data.

In C++, for instance, an integer usually (but not always!) occupies four bytes of memory. So pointing a pointer at the beginning of an array and then incrementing it by one moves it over four bytes — to the beginning of the next item in the array. Because this type of array manipulation has become language-specific, we’re not going to use pointers in this book. We’ll instead settle for array items simply being referenced by their index numbers.

<GRAPHIC>

//// THERE IS SOME UNIVERSAL MEMORY SIZE THING BEING DESCRIBED
HERE, WORTH IT’S OWN SECTION ////

The length of an array is one of its most important properties. Many array operations take the length of an array into account for their operations. Most programming languages will throw an error if you try to access an element in an array that is beyond the array’s “scope,” or length.

Weakly vs. Strongly Typed Languages

The difference between weakly typed and strongly typed languages comes down to how strict the language requires programmers to be when mixing data. In strongly typed languages, like C/C++ or Java, data mixing is not allowed without creating specific data types to support the mixing. In weakly typed languages, this mixing is allowed, accepted, and even encouraged!

If you’ve used Typescript you’re used to a hybrid of both, with Typescript being a language that adds strong data typing to Javascript. There’s some controversy over whether this is desirable or should even be allowed. Is Typescript a useful tool or simply meant to make Javascript more palatable for coders coming from more strongly typed languages, like Java?

In C++ an array must be of a specific data type, and only that data type is allowed in the array. Here is an integer array in C++. It can only be used to hold five integer values, no more, and no other type of data is allowed.

```
int intArray[5] = {33, 22, 11, 18, 96}
```

In weakly-typed JavaScript, all different types of data can be mixed together, and the language interpreter doesn’t mind at all:

```
mixedArray = [33, 22, 3.15159, "hello", false]
```

The reasons for these differences are logical, of course, and mostly have to do with the way memory is allocated by different programming languages.

Array Numbering

In most programming languages, arrays are numbered from zero. This is because arrays are meant to be thought of as contiguous elements in a computer's memory, as explained in the section on pointers, above. If arrays started counting from 1, there would be space unaccounted for in that first position in the memory that exists between 0 and 1. It's the same reason why the 19th century starts in 1800. You have to account for the First Century somehow, and no one wants to be from the "Zeroeth Century."

That means the first element in the array `testArray` is located at array position 0. If `testArray = [21, 31, 41, ...]` then `testArray[0]` is where to find 21, `testArray[1]` is where to find 31, `testArray[2]` is where to find 41, etc.

It's not uncommon for programmers to use an "offset" variable to account for the difference in counting from 0, but I think it's pretty easy to just add or subtract 1, as needed.

```
someArray = []
for (int i = 0; i < someArray.length, i++):
    print "The item at position", i + 1, " is " someArray[i]
```

Arrays and For Loops

When I taught an introductory Java class, I used to tell my students that "arrays and for loops go together like peanut butter and jelly." What I mean by this is that they go together well, as a for loop is one of the easiest ways to traverse through an array by index, as shown in the example just above.

One day when I was giving this example, a student cringed and said "Ew, why would you mix peanuts with jelly?!" I realized this was something that was not commonly done in her culture, and that not all foods are American foods. So if "peanut butter and jelly" makes you cringe, please feel free to pick your own food analogy: "Rice and beans", "fish and rice," "flour and butter", or any two foods that blend together well in your mind will do. The point is if you have to iterate over an array (peanut butter), use a for loop (jelly)!

Modifying (or not Modifying) Arrays

In strongly typed languages, arrays can't be modified in length or data type once created. This leads to the desired outcome of an array always being what the program-

mer says they are. For a short time in the history of computer programming, this became an undesirable thing, as the rise of web programming created a generation of programmers that didn't need or want to worry about data types in their arrays since it was the browser's problem.

In that way where all old things become new again in computer programming, there has been a recent shift towards “functional programming,” which insists that programmers regain control of arrays and not simply modify them according to their moods. In “pure” functional programming you should not modify an array at all, but create a new copy every time you add or remove something from it. This keeps the array in a certain state and ensures that past states can be re-created from older copies. If you've ever worked with a library like Redux you might be used to this approach.

In the programming interview, make sure you understand what you're being asked to do, and treat your arrays with intention. Don't just blithely and blindly push information into and out of an array without understanding the problem you're being asked to solve.

Additionally, in some languages you can't simply copy an array, because the copy will copy not the array but will only copy its pointers. The pointers in the copy will continue to point to the data to which the original array was told to point. This means that modifying the data in the new array will modify the data in the old array, and vice-versa. This type of copy is called a “shallow” copy, as opposed to a “deep” copy. It's important to know which type of copying your language supports, and how to get a deep copy if you need a deep copy (or a shallow copy, if that's what you want instead).

Basic Array Functions

Most languages contain a library of functions for arrays and questions about these functions are fairly commonly found in coding tests. These are functions that allow programmers to modify arrays in ways that make them more useful to programmers.

As you read through these examples, keep in mind that these functions — and really any functions — are not created by magic. They have been added to the language by programmers trying to make their jobs easier by making the language more useful. As you read about array functions, try to think of how you might create them yourself. Or better still, create them yourself and look inside the language specification to see how the problem has actually been solved.

Array functions fall into two general categories: modifying array data and retrieving array data. As mentioned earlier, the length of the array is important for making most of these operations work properly.

Arrays vs. Objects vs. Tuples vs. ???

In the beginning was the array, and no other data type could be used for grouping related data.

Recently many programming languages have added “objects” as primitives, which are data types where the information is relational. Items in an object can be indexed by more than just a number, and can actually be specifically named. We will explore this idea more deeply when we get into hashes in the next chapter. In some languages, like JavaScript, there is no “array” primitive data type — all arrays are actually objects!

If that’s not already confusing enough, most modern programming languages, like Python, have realized the need to combine multiple data types in new and more useful ways. In addition to arrays, Python contains lists, tuples, and dicts, all of which are array-like data structures with specific properties for specific uses.

This book is not geared toward teaching primitive data types, so make sure you understand the different types and functions available in your specific language. And remember, if you don’t see an array data type or function you like, you can always create your own!

Array modification functions include the following operations:

`length()` Retrieve the length of the array, usually to do something else with it. Sometimes shortened to `len()`

```
demoArray = [1, 2, 3, 4, 5]
print(len(demoArray)) # returns '5'
```

`for` loops Every programming language contains a function that allows for simple iteration, or “counting.” Iteration is a process that has been available in programming languages from the very beginning, as it’s essential to making computers work.

`for/in` Some programming languages — usually the weakly-typed ones — support `for/in` operations, which automatically consider the length of an array in iteration.

```
items = ["loaf of bread", "container of milk", "stick of butter"]
for item in items:
    print(item)
```

`push()` / `pop()` These functions are called different things in different languages but they are for pushing an element into the end (or beginning) of an array and for retrieving an element from the beginning (or end) of an array.

`slice()` The slice function allows you to

`splice()` The splice function is useful for

`reverse()` Many programming languages contain a function for reversing a string. This can be useful for searching/sorting algorithms, which will be covered in chapter

1. Sliding window “Find the longest substring with k unique characters in string.”
Linear data structure, subset, condition.



Knowing how to program basic built-in functions is useful because you can see how they work and because the techniques in built-in functions usually contain processes that optimize those functions. These functions are often added to programming languages because of how often they're used, and this means they're the basis for a lot of programming interview questions. So download the source, dig through the source code, read the spec, reverse engineer, and figure out how to program them yourself!

Array Copying

Deep copy v. shallow copy

JSON (JavaScript Object Notation)

Array Searching, Sorting, and Manipulation

Now that you know the basics of using arrays with built-in functions, this next section will focus on essential algorithms for working with data in arrays. Much of computer programming is about searching, sorting, and manipulating data, and many programming interviews will require you to understand the basics of how to implement these algorithms. Even if you aren't directly asked questions about these algorithms, they are incredibly useful tools to have in your programming toolbox and they can be used to solve many different types of programming problems.

Sorting

Sorting is just what it sounds like: sorting data according to a given criteria.

Let's start with the most straightforward sort algorithm to understand

Searching

Searching is when you find an item or items in a set of data. We'll talk more about search in Chapter 12, but we can start the conversation here in our discussion of arrays. The main differences that determine the difference in whether of not to prefer

a given search algorithm is the quantity of data being searched, and whether or not that data is in a specific order to start.

To try to gain some intuition for the differences in search algorithms, consider this simple example. Let's say you decided to go to the pound to get a puppy. There are one hundred kennels containing dogs to choose from. You walk past all the kennels and you see a dog you just love named "Harry."

The animal trainer who works at the pound hands you a stack of papers and says you have to find Harry's paper to take him home. The papers aren't sorted. Right off the bat it would clearly be easier to search through the stack if the papers were already sorted. But stop for a moment and think about how you're going to do that? Are you just going to page through the papers from A-H until you find Harry? Start the search at Z? Something else? Would you search through the papers the same way if there were only 10? Or 10,000? But it's not any of those things; it's a stack of 100 unsorted papers. So how are you going to go about finding "Harry?" Stop and consider these questions for a moment before moving on.

Let's start with a simple search: the binary search. You may have used this one before if you've ever played "Guess the Number." Someone says to you, "I'm thinking of a number between one and 100 and if you can guess the number in 6 guesses or fewer, I'll give you \$5. After every guess I'll tell you if my number is higher or lower than the one you choose." Does it matter that you're told whether or not the number is higher or lower after every guess? Why? How would you go about solving this problem?

Even people who have never played the game before will quickly come up with winning strategies. It doesn't make a lot of sense to guess numbers in order because that would give you only about a 5% chance to guess the number before running out of guesses. You're also only going to guess numbers based on the clues provided to you by the person who makes the offer. If you guess 71, and she tells you it's going to be higher than 71, it makes no sense to guess 20 on the next turn.

The foolproof way to win this game every time is by conducting a binary search. The reason it's called a binary search is because every guess reduces the remaining number pool into two parts, and then eliminates one of them from further consideration.

Following this strategy, your first guess should be 50. It cuts the remaining number pool in half.

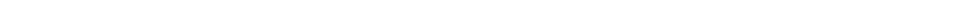
Sample Questions

Array Rotation

Rotate

Permutations

ABCDE = EBDCA



Linked Lists

After reading the chapter the reader will understand lorem...

Linked Lists

Link your lists

Linked List Definition

Link your definitions ipsum

Singly Linked List

One list with one link to rule them on

Doubly Linked List

Two links to rule them all

Developing Intuition: What are Linked Lists Used For?

Think and grow lists

Example Questions

Here are the examples you might find

Find

Find this is

Insert

Insert is a this and that

Delete

Delete and you need to know

Reversing a linked list

Reverse play the reverse card

Item Swap

You go this way, I go that way.

Merging linked lists

All together now.

Stacks and Queues

After reading this chapter, the reader should understand the ...

Stacks and Queues

Stacks and Queues are two kinds of data structures that

FIFO v. LIFO

FIFO v. LIFO

Stack Definition

A stack has a few different meanings in computer science, but when it comes to data structures, a stack is a “last in, first out” way of ordering data. The analogy commonly given is of a stack of plates in a cafeteria, where the last plate placed on the stack is the first one that comes off the top. Earlier we talked about “push/pop” functions available for arrays, and these are often used with stacks as well. A plate (or a pancake, or data) can be pushed onto the top of the stack, and that same plate (pancake, data) will be the first one popped off the top.

Queue Definition

A queue is a “first in, first out” data structure akin to standing in line. The first piece of data pushed into a queue is the first piece of data to be taken back out of it again.

Developing Intuition: What are stacks and queues used for?

Stacks and queues are often used to indicate *priority* in the processing of data. It's not hard to see why. Imagine an elevator that only goes to two floors, say the ground floor and the 10th floor. Ten people enter the elevator on the ground floor. Because of the way elevators — and society — are constructed, the last people to board the elevator will be the first ones to leave it. The elevator creates a priority by the way it is boarded. It's simply impossible for the people who first board the elevator to exit first, as they will be furthest from the door. This is an example of a stack.

A queue is even easier to understand. We've all waited in line. It even seems “unfair” for people to cut in line; the first in must be the first out!

There are, of course, exceptions to these rules. Pause for a moment and try to think of a real-world exception to each.

There are a few ways to use stacks and queues in computer science, but most often they are used for assigning priority to tasks. They can be used to determine the priority of packets in a network or to determine tasks that must be completed before other tasks begin.

Is a for loop an example of a stack or a queue? What about a nested for loop? What about the browser event loop?

Example Questions

Here is an example of the sample that you're ample

```
print('hello world')
```

Balanced parentheses

How can you balance parentheses

Find the second smallest item

Find the second-smallest number in an array of integers. Can you scale this to the third smallest? Third largest? Can you scale this to other sorts of ordering, like alphabetical order?

Implement x using a stack

How to implement x

Heaps

Heaps and heaps and heaps

Heap Definition

A heap is a data structure that

Developing Intuition: What are heaps used for?

Think and grow heaps

Max heaps and min heaps

MINIMAX and heap

Example questions

Before you get your heap on here's what you need to know.

Is heap a max or min heap?

IS IT MAX OR MIN

Convert max heap to min heap

Min is max and max is min

1. Top k elements. "Find k largest elements in an array" Input is linear data structure. Solve using heap.

After reading this chapter, the reader should understand the definition of a hash ipsum.

Hash definition

Building Intuition: What are hashes used for?

Reasoning about Hashes

Real world example

Hashes and encryption

Salts

Example Questions

Word Count

Palindromes II

Anagrams

etc...

Using AI to Study for Hash Problems

Trees

After reading this chapter, the reader should understand the definition of a tree...
ipsum

Trees Definition

Building Intuition: What are trees used for?

Trees and heaps

Reasoning about trees

Tree Operations

Insertion

Deletion

Balancing

Types of Trees

Binary Trees

Properties

Search

Sort

Other Tree Types

Red and Black Trees

AVL Trees

Tries

Other trees

Example Questions

Finding height of tree

Graphs

After reading this chapter, the reader should understand the definition of a hash, a tree, and a graph, and what each is used for in programming. The reader will be given some idea of what kinds of questions to expect on a coding test about these data structures and strategies for solving them.

Graph Definition

Building Intuition: What are graphs used for?

Graph data structures

Arrays

Matrices

Reasoning about graphs

Depth First Graph

Breadth First Graph

Weighted Graph

Dijkstra's algorithm

Example Questions

Closest path

Furthest path

Knight walk

Flood fill

Functions and Recursion

After reading this chapter the reader should know how to use both recursive and non-recursive functions, how to reason about functions, and what type of functions problems to expect on a coding test.

Functions 101

You will be asked a question about functions

While there are many ways to define and describe functions I've never heard one more accurate than "functions are the building blocks of code." Functions are a primary and essential part of coding that you should absolutely expect to use in a coding interview. Even if you're not asked a question about functions directly, you will almost certainly need to use a function to solve a harder problem.

What is a function?

Functions are usually the first "advanced" things a programmer learns. Functions in programming are similar to the function equations you may have learned in high school algebra, which looked something like this:

$$f(x) = 2x$$

Learning about functions often included exercises to create tables of the expected results. The table for $f(x) = 2x$ might look like this:

x	f(x)
1	2
2	4

x	f(x)
3	6

Functions in programming perform a similar, um, function, allowing data passed in to be transformed in a way specified by the programmer. Here is the above-mentioned example high-school algebra function in Python:

```
def doubler-function(value):
    return (2 * value)
```

Function implementations are language-specific, with C++ and Java requiring functions be set to data types, and JavaScript containing several ways to create functions, depending on what you're trying to accomplish. In every language, however, functions are just like the ones you encountered in high-school algebra. They take a value (or values), transform them according to a set of rules, and return the result.

Building Intuition: Reasoning about functions

Functions basics

Function prototypes

In strongly typed languages, functions are usually defined by a *prototype*, a block of code that defines the function in terms of the data it can expect to receive and return. In this way, function prototypes are used to enforce type checking, to make sure that the values that are being passed to a function are what it expects to receive. This is an example function prototype in C:

```
// Function prototype
float multIntAndFloat(int num1, float num2);
```

The first `float` in this prototype indicates that it will return a float. The `int` and `float` keywords preceding the two values being passed in means these values *must* be an `int` and a `float`. If you pass or return any other type of data, the compiler will return an error — or at least a warning, depending on the C implementation you're using.

Weakly typed languages don't require function prototypes, but there are ways to require them, of course. The Typescript programming language is probably the best current example of creating type-checking, for all data and not just functions, in JavaScript.



Try coming up with examples of situations in which it would be important to check data passed to a function. Are the checks limited to checking data types? If your language doesn't support function prototyping, how can data checks in functions be enforced?

Constructing functions

Receiving function data

Returning function data

Closures

Functions Advanced

The mystical “this”

Constructor functions

Generator functions

Function chaining

Side effects and functional programming

Currying

Higher Order Functions

Building Intuition: Functions Ideas

Functions and state

Function refactoring

Function composition

Example Questions

Swap two numbers in place

Refactor this function

62 | Chapter 10: Functions and Recursion

“Program to an interface, not an implementation”

Recursion

Search and Sort

After reading this chapter, the reader should know the basics of search and sort algorithms, how to reason about search and algorithms, and general guidelines for choosing the right search and/or sort algorithm to use to solve the problems that might arise in a coding interview. This chapter will build on ideas presented in earlier chapters.

Search

What does “search” mean?

How is search used

Why is search important

Building Intuition: Reasoning about Search

Types of search

Linear search

Binary Search

Other types of search

Example Questions

Partitioning

Solve x problem using y search

Search optimization

Searching and data structures

Sort

What does “sort” mean?

How is sort used?

Why sort is important to understand

Building Intuition: Reasoning about Sort

Types of sort

Insertion sort

Dynamic Programming and Greedy Algorithms

After reading this chapter, the reader should know the basics of dynamic programming, how to reason about dynamic programming, general guidelines for solving problems related to dynamic programming, and how to use dynamic programming to solve problems in a coding interview.

Introduction to Dynamic Programming

Why Dynamic Programming Matters

Building Intuition: Reasoning about Dynamic Programming

Solving Problems using Dynamic Programming

Memoization

Recursion, revisited

Example problems

The Climbing Stair problem

The Knapsack problem

Longest common substring

“Parallel Thinking” Questions

Put some introductory text here.

Reasoning Questions

Active Listening Questions

Logic Questions

Using Generative AI to Study

Put some introductory text here.

Reasoning Questions

Active Listening Questions

Logic Questions

About the Author(s)

John Doe does some interesting stuff...