Teach Yourself Computer Science

If you're a self-taught engineer or bootcamp grad, you owe it to yourself to learn computer science. Thankfully, you can give yourself a world-class CS education without investing years and a small fortune in a degree program 🚀.

There are plenty of resources out there, but some are better than others. You don't need yet another "200+ Free Online Courses" listicle. You need answers to these questions:

- **Which subjects** should you learn, and why?
- What is the **best book or video lecture series** for each subject?

This guide is our attempt to definitively answer these questions.

TL;DR:

Study all nine subjects below, in roughly the presented order, using either the suggested textbook or video lecture series, but ideally both. Aim for 100-200 hours of study of each topic, then revisit favorites throughout your career 🚀.

Programming

Don't be the person who "never quite understood" something like recursion.

Best book: *Structure and Interpretation of Computer Programs*

Best video lectures: Brian Harvey's Berkeley CS 61A

Computer Architecture

If you don't have a solid mental model of how a computer actually works, all of your higher-level abstractions will be brittle.

Best book: *Computer Organization and Design*

Best video lectures: Berkeley CS 61C

Algorithms and Data Structures

If you don't know how to use ubiquitous data structures like stacks, queues, trees, and graphs, you won't be able to solve hard problems.

Best book: *The Algorithm Design Manual*

Best video lectures: Steven Skiena's lectures

Math for CS

CS is basically a runaway branch of applied math, so learning math will give you a competitive advantage.

Best book: *Mathematics for Computer Science*

Best video lectures: Tom Leighton's MIT 6.042J

Operating Systems

Most of the code you write is run by an operating system, so you should know how those interact.

Best book: *Operating Systems: Three Easy Pieces*

Best video lectures: Berkeley CS 162

Computer Networking

The Internet turned out to be a big deal: understand how it works to unlock its full potential.

Best book: *Computer Networking: A Top-Down Approach*

Best video lectures: Stanford CS 144

Databases

Data is at the heart of most significant programs, but few understand how database systems actually work.

Best book: *Readings in Database Systems*

Best video lectures: Joe Hellerstein's Berkeley CS 186

Languages and Compilers

If you understand how languages and compilers actually work, you'll write better code and learn new languages more easily.

Best book: *Compilers: Principles, Techniques and Tools*

Best video lectures: Alex Aiken's course on Lagunita

Distributed Systems

These days, *most* systems are distributed systems.

Best book: *Distributed Systems, 3rd Edition* by Maarten van Steen

Best video lectures: □

Why learn computer science?

There are 2 types of software engineer: those who understand computer science well enough to

do challenging, innovative work, and those who just get by because they're familiar with a few high level tools.

Both call themselves software engineers, and both tend to earn similar salaries in their early careers. But Type 1 engineers grow in to more fulfilling and well-remunerated work over time, whether that's valuable commercial work or breakthrough open-source projects, technical leadership or high-quality individual contributions.

> The global SMS system does around 20bn messages a day. WhatsApp is now doing 42bn. With 57 engineers. pic.twitter.com/zZrtSIzhlR
>
> — Benedict Evans (@BenedictEvans) February 2, 2016

Type 1 engineers find ways to learn computer science in depth, whether through conventional means or by relentlessly learning throughout their careers. Type 2 engineers typically stay at the surface, learning specific tools and technologies rather than their underlying foundations, only picking up new skills when the winds of technical fashion change.

Currently, the number of people entering the industry is rapidly increasing, while the number of CS grads is essentially static. This oversupply of Type 2 engineers is starting to reduce their employment opportunities and keep them out of the industry's more fulfilling work. Whether you're striving to become a Type 1 engineer or simply looking for more job security, learning computer science is the only reliable path.

> Lol oh but they were.... pic.twitter.com/XVNYlXAHar
>
> — Jenna Bilotta (@jenna) March 4, 2017

Subject guides

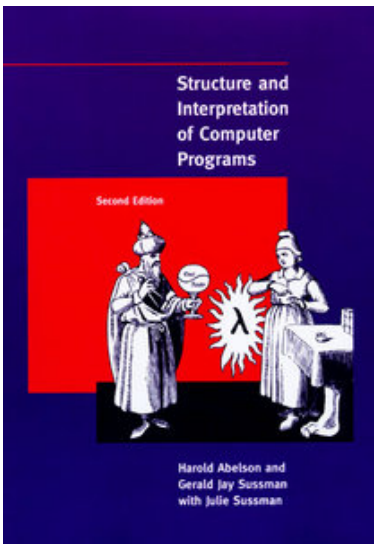
Programming

Most undergraduate CS programs start with an "introduction" to computer programming. The best versions of these courses cater not just to novices, but also to those who missed beneficial concepts and programming models while first learning to code.

Our standard recommendation for this content is the classic *Structure and Interpretation of Computer Programs*, which is available online for free both as a book, and as a set of MIT video lectures. While those lectures are great, our video suggestion is actually Brian Harvey's SICP lectures (for the 61A course at Berkeley) instead. These are more refined and better targeted at new students than are the MIT lectures.

We recommend working through at least the first three chapters of SICP and doing the exercises. For additional practice, work through a set of small programming problems like those on exercism.

For those who find SICP too challenging, we recommend *How to Design Programs*. For those who find it too easy, we recommend *Concepts, Techniques, and Models of Computer Programming*.



Computer Architecture

Computer Architecture—sometimes called “computer systems” or “computer organization”—is an important first look at computing below the surface of software. In our experience, it’s the most neglected area among self-taught software engineers.

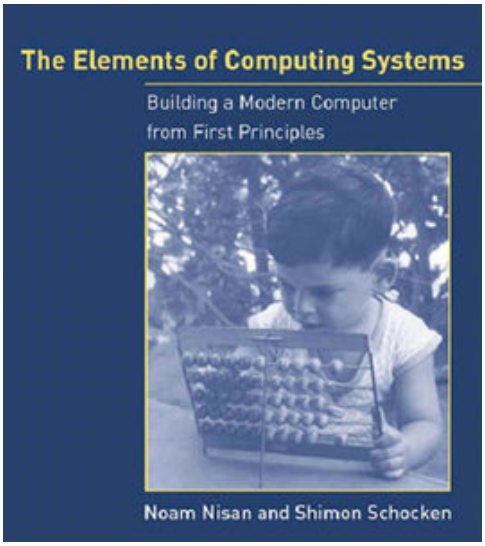
The Elements of Computing Systems, also known as “Nand2Tetris” is an ambitious book attempting to give you a cohesive understanding of how everything in a computer works. Each chapter involves building a small piece of the overall system, from writing elementary logic gates in HDL, through a CPU and assembler, all the way to an application the size of a Tetris game.

We recommend reading through the first six chapters of the book and completing the associated projects. This will develop your understanding of the relationship between the architecture of the machine and the software that runs on it.

The first half of the book (and all of its projects), are available for free from the Nand2Tetris website. It’s also available as a Coursera course with accompanying videos.

In seeking simplicity and cohesiveness, Nand2Tetris trades off depth. In particular, two very important concepts in modern computer architectures are pipelining and memory hierarchy, but both are mostly absent from the text.

Once you feel comfortable with the content of Nand2Tetris, our next suggestion is Patterson and Hennessy's *Computer Organization and Design*, an excellent and now classic text. Not every section in the book is essential; we suggest following Berkeley's CS61C course "Great Ideas in Computer Architecture" for specific readings. The lecture notes and labs are available online, and past lectures are on the Internet Archive.



Hardware is the platform

– Mike Acton, Engine Director at Insomniac Games
(watch his CppCon talk)

Algorithms and Data Structures

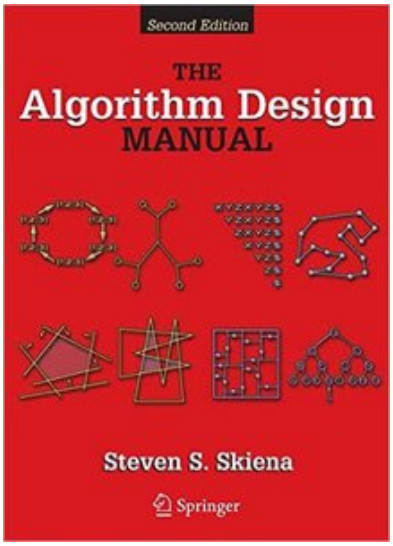
We agree with decades of common wisdom that familiarity with common algorithms and data structures is one of the most empowering aspects of a computer science education. This is also a great place to train one's general problem-solving abilities, which will pay off in every other area of study.

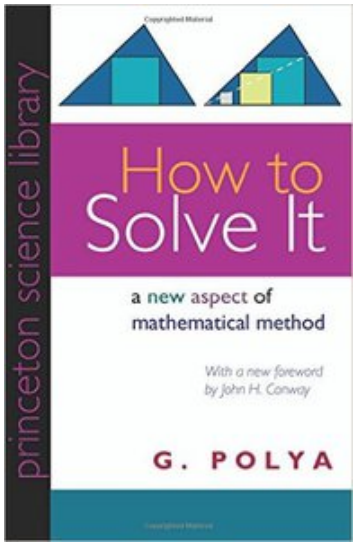
There are hundreds of books available, but our favorite is *The Algorithm Design Manual* by Steven Skiena. He clearly loves this stuff and can't wait to help you understand it. This is a refreshing change, in our opinion, from the more commonly recommended Cormen, Leiserson, Rivest & Stein, or Sedgewick books. These last two texts tend to be too proof-heavy for those learning the material primarily to help them *solve problems*.

For those who prefer video lectures, Skiena generously provides his online. We also really like Tim Roughgarden's course, available from Stanford's MOOC platform Lagunita, or on Coursera. Whether you prefer Skiena's or Roughgarden's lecture style will be a matter of personal preference.

For practice, our preferred approach is for students to solve problems on Leetcode. These tend to be interesting problems with decent accompanying solutions and discussions. They also help you test progress against questions that are commonly used in technical interviews at the more competitive software companies. We suggest solving around 100 random leetcode problems as part of your studies.

Finally, we strongly recommend *How to Solve It* as an excellent and unique guide to general problem solving; it's as applicable to computer science as it is to mathematics.





> **I have only one method that I recommend extensively—it's called think before you write.**

— Richard Hamming

Mathematics for Computer Science

In some ways, computer science is an overgrown branch of applied mathematics. While many software engineers try—and to varying degrees succeed—at ignoring this, we encourage you to embrace it with direct study. Doing so successfully will give you an enormous competitive advantage over those who don't.

The most relevant area of math for CS is broadly called "discrete mathematics", where "discrete" is the opposite of "continuous" and is loosely a collection of interesting applied math topics outside of calculus. Given the vague definition, it's not meaningful to try to cover the entire breadth of "discrete mathematics". A more realistic goal is to build a working understanding of logic, combinatorics and probability, set theory, graph theory, and a little of the number theory informing cryptography. Linear algebra is an additional worthwhile area of study, given its importance in computer graphics and machine learning.

Our suggested starting point for discrete mathematics is the set of lecture notes by László Lovász. Professor Lovász did a good job of making the content approachable and intuitive, so this serves as a better starting point than more formal texts.

For a more advanced treatment, we suggest *Mathematics for Computer Science*, the book-length lecture notes for the MIT course of the same name. That course's video lectures are also freely available, and are our recommended video lectures for discrete math.

For linear algebra, we suggest starting with the Essence of linear algebra video series, followed by Gilbert Strang's book and video lectures.

> **If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.**

— John von Neumann

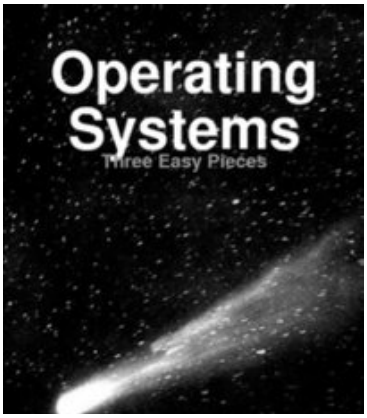
Operating Systems

Operating System Concepts (the "Dinosaur book") and *Modern Operating Systems* are the "classic" books on operating systems. Both have attracted criticism for their writing styles, and for being the 1000-page-long type of textbook that gets bits bolted onto it every few years to encourage purchasing of the "latest edition".

Operating Systems: Three Easy Pieces is a good alternative that's freely available online. We particularly like the structure of the book and feel that the exercises are well worth doing.

After OSTEP, we encourage you to explore the design decisions of specific operating systems, through "{OS name} Internals" style books such as *Lion's commentary on Unix*, *The Design and Implementation of the FreeBSD Operating System*, and *Mac OS X Internals*.

A great way to consolidate your understanding of operating systems is to read the code of a small kernel and add features. A great choice is xv6, a port of Unix V6 to ANSI C and x86 maintained for a course at MIT. OSTEP has an appendix of potential xv6 labs full of great ideas for potential projects.



Computer Networking

Given that so much of software engineering is on web servers and clients, one of the most immediately valuable areas of computer science is computer networking. Our self-taught students who methodically study networking find that they finally understand terms, concepts and protocols they'd been surrounded by for years.

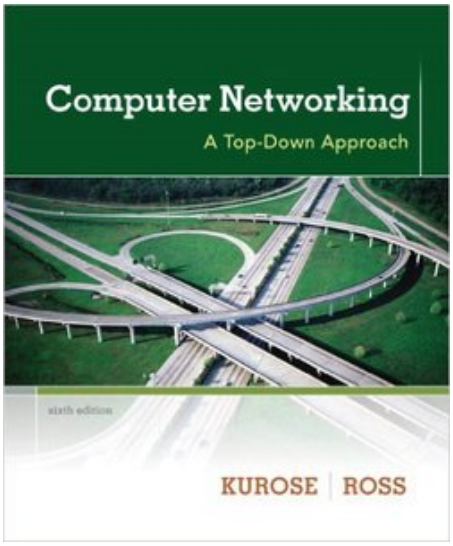
Our favorite book on the topic is *Computer Networking: A Top-Down Approach*. The small projects and exercises in the book are well worth doing, and we particularly like the "Wireshark labs", which they have generously provided online.

For those who prefer video lectures, we suggest Stanford's *Introduction to Computer Networking course* available on their MOOC platform Lagunita.

The study of networking benefits more from projects than it does from small exercises. Some possible projects are: an HTTP server, a UDP-based chat app, a mini TCP stack, a proxy or load balancer, and a distributed hash table.

> **You can't gaze in the crystal ball and see the future. What the Internet is going to be in the future is what society makes it.**

— Bob Kahn



Databases

It takes more work to self-learn about database systems than it does with most other topics. It's a relatively new (i.e. post 1970s) field of study with strong commercial incentives for ideas to stay behind closed doors. Additionally, many potentially excellent textbook authors have preferred to join or start companies instead.

Given the circumstances, we encourage self-learners to generally avoid textbooks and start with the Spring 2015 recording of CS 186, Joe Hellerstein's databases course at Berkeley, and to progress to reading papers after.

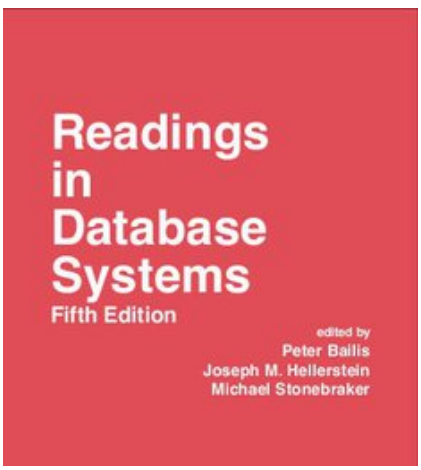
One paper particularly worth mentioning for new students is "Architecture of a Database System", which uniquely provides a high-level view of how relational database management systems (RDBMS) work. This will serve as a useful skeleton for further study.

Readings in Database Systems, better known as the databases "Red Book", is a collection of papers compiled and edited by Peter Bailis, Joe Hellerstein and Michael Stonebraker. For those who have progressed beyond the level of the CS 186 content, the Red Book should be your next stop.

If you insist on using an introductory textbook, we suggest *Database Management Systems* by Ramakrishnan and Gehrke. For more advanced students, Jim Gray's classic *Transaction Processing: Concepts and Techniques* is worthwhile, but we don't encourage using this as a first resource.

It's hard to consolidate databases theory without writing a good amount of code. CS 186 students add features to Spark, which is a reasonable project, but we suggest just writing a simple relational database management system from scratch. It will not be feature rich, of course, but even writing the most rudimentary version of every aspect of a typical RDBMS will be illuminating.

Finally, data modeling is a neglected and poorly taught aspect of working with databases. Our suggested book on the topic is *Data and Reality: A Timeless Perspective on Perceiving and Managing Information in Our Imprecise World*.



Languages and Compilers

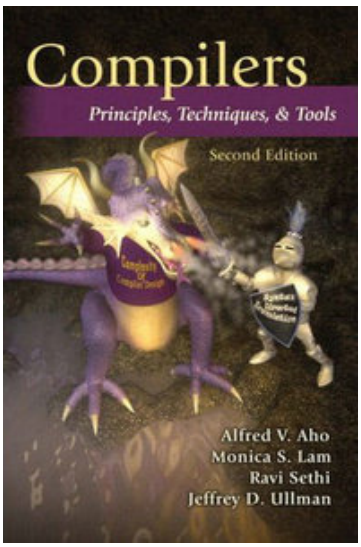
Most programmers learn languages, whereas most computer scientists learn *about* languages. This gives the computer scientist a distinct advantage over the programmer, even in the domain of programming! Their knowledge generalizes; they are able to understand the operation of a new language more deeply and quickly than those who have merely learned specific languages.

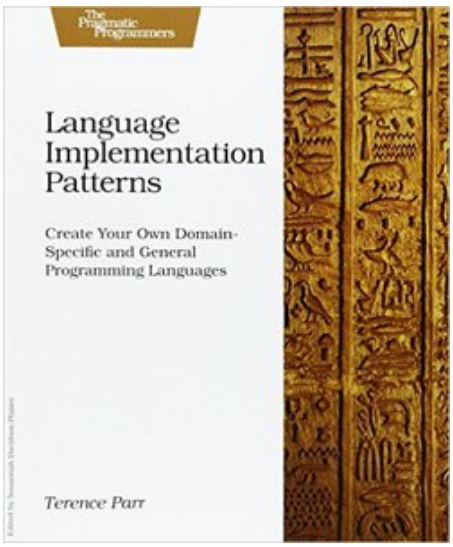
The canonical introductory text is *Compilers: Principles, Techniques & Tools*, commonly called "the Dragon Book". Unfortunately, it's not designed for self-study, but rather for instructors to pick out 1-2 semesters worth of topics for their courses. It's almost essential then, that you cherry-pick the topics, ideally with the help of a mentor.

If you choose to use the Dragon Book for self-study, we recommend following a video lecture series for structure, then dipping into the Dragon Book as needed for more depth. Our recommended online course is Alex Aiken's, available from Stanford's MOOC platform Lagunita.

As a potential alternative to the Dragon Book we suggest *Language Implementation Patterns* by Terence Parr. It is written more directly for the practicing software engineer who intends to work on small language projects like DSLs, which may make it more practical for your purposes. Of course, it sacrifices some valuable theory to do so.

For project work, we suggest writing a compiler either for a simple teaching language like COOL, or for a subset of a language that interests you. Those who find such a project daunting could start with Make a Lisp, which steps you through the project.





> **Don't be a boilerplate programmer. Instead, build tools for users and other programmers. Take historical note of textile and steel**

> **industries: do you want to build machines and tools, or do you want to operate those machines?**

— Ras Bodik at the start of his compilers course

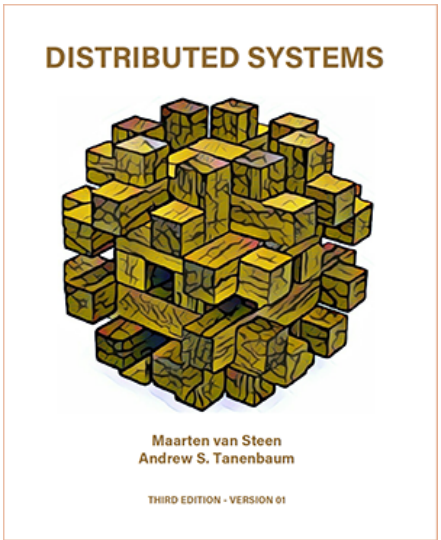
Distributed Systems

As computers have increased in number, they have also *spread*. Whereas businesses would previously purchase larger and larger mainframes, it's typical now for even very small applications to run across multiple machines. Distributed systems is the study of how to reason about the trade-offs involved in doing so, an increasingly important skill.

Our suggested textbook for self-study is Maarten van Steen and Andrew Tanenbaum's *Distributed Systems, 3rd Edition*. It's a great improvement over the previous edition, and is available for free online thanks to the generosity of its authors. Given that the distributed systems is a rapidly changing field, no textbook will serve as a trail guide, but Maarten van Steen's is the best overview we've seen of well-established foundations.

A good course for which some videos are online is MIT's 6.824 (a graduate course), but unfortunately the audio quality in the recordings is poor, and it's not clear if the recordings were authorized.

No matter the choice of textbook or other secondary resources, study of distributed systems absolutely mandates reading papers. A good list is here, and we would highly encourage attending your local Papers We Love chapter.



Frequently asked questions

What about AI/graphics/pet-topic-X?

We've tried to limit our list to computer science topics that we feel *every practicing software engineer* should know, irrespective of specialty or industry. With this foundation, you'll be in a much better position to pick up textbooks or papers and learn the core concepts without much guidance. Here are our suggested starting points for a couple of common "electives":

- For artificial intelligence: do Berkeley's intro to AI course by watching the videos and completing the excellent Pacman projects. As a textbook, use Russell and Norvig's *Artificial Intelligence: A Modern Approach*.
- For machine learning: do Andrew Ng's Coursera course. Be patient, and make sure you understand the fundamentals before racing off to shiny new topics like deep learning.
- For computer graphics: work through Berkeley's CS 184 material, and use Computer Graphics: Principles and Practice as a textbook.

How strict is the suggested sequencing?

Realistically, all of these subjects have a significant amount of overlap, and refer to one another cyclically. Take for instance the relationship between discrete math and algorithms: learning math first would help you analyze and understand your algorithms in greater depth, but learning algorithms first would provide greater motivation and context for discrete math. Ideally, you'd revisit both of these topics many times throughout your career.

As such, our suggested sequencing is mostly there to help you *just get started*… if you have a compelling reason to prefer a different sequence, then go for it. The most significant "pre-requisites" in our opinion are: computer architecture before operating systems or databases, and networking and operating systems before distributed systems.

Who is the target audience for this guide?

We have in mind that you are a self-taught software engineer, bootcamp grad or precocious high school student, or a college student looking to supplement your formal education with some self-study. The question of when to embark upon this journey is an entirely personal one, but most people tend to benefit from having some professional experience before diving too deep into CS theory. For instance, we notice that students *love* learning about database systems if they have already worked with databases professionally, or about computer networking if they've worked on a web project or two.

How does this compare to Open Source Society or freeCodeCamp curricula?

The OSS guide has too many subjects, suggests inferior resources for many of them, and provides no rationale or guidance around why or what aspects of particular courses are valuable. We strove to limit our list of courses to those which you *really should know* as a software engineer, irrespective of your specialty, and to help you understand why each course is included.

freeCodeCamp is focused mostly on programming, not computer science. For why you might want to learn computer science, see above.

What about language X?

Learning a particular programming language is on a totally different plane to learning about an area of computer science — learning a language is much *easier* and much *less valuable*. If you already know a couple of languages, we strongly suggest simply following our guide and fitting language acquisition in the gaps, or leaving it for afterwards. If you've learned programming well (such as through *Structure and Interpretation of Computer Programs*), and especially if you have learned compilers, it should take you little more than a weekend to learn the essentials of a new language.

What about trendy technology X?

No single technology is important enough that learning to use it should be a core part of your education. On the other hand, it's great that you're excited to learn about that thing. The trick is to work backwards from the particular technology to the underlying field or concept, and learn that in depth before seeing how your trendy technology fits into the bigger picture.

Why are you still recommending the Dragon book?

The Dragon book is still the most complete single resource for compilers. It gets a bad rap, typically for overemphasizing certain topics that are less fashionable to cover in detail these days, such as parsing. The thing is, the book was never intended to be studied cover to cover, only to provide enough material for an instructor to put together a course. Similarly, a self-learner can choose their own adventure through the book, or better yet follow the suggestions that lecturers of public courses have made in their course outlines.

How can I get textbooks cheaply?

Many of the textbooks we suggest are freely available online, thanks to the generosity of their authors. For those that aren't, we suggest buying used copies of older editions. As a general rule, if there has been more than a couple of editions of a textbook, it's quite likely that an older edition is perfectly adequate. It's certainly unlikely that the newest version is 10x better than an older one, even if that's what the price difference is!

Who made this?

This guide was written by Ozan Onay and Myles Byrne, instructors at the Bradfield School of Computer Science in San Francisco. It is based on our experience teaching foundational computer science to hundreds of mostly self-taught engineers and bootcamp grads. Thank you to all of our students for your continued feedback on self-teaching resources. Thanks too to Alek Sharma, Omar Rayward, Ammar Mian and Tyler Bettilyon for feedback on this guide.

You may also like to join our mailing list:

email address Subscribe

hello@bradfieldcs.com
San Francisco, California