From now on, I decide to make a long term plan that I will learn the basic computer science courses by myself. What makes me create this plan file is that I come to realize that sometimes I must make my hands dirty to move those ideas in my mind to practice. This is important, and this is real. That I can see what I am doing, what I will do and what I will become in the near future. So here is the plan, not just a to-do list.

"Aim for 100-200 hours of study of each topic, then revisit favourites throughout your career".

Subject	Why study?	Book	Videos
Programming	Don't be the person who "never quite understood" something like recursion.	*Structure and Interpretation of Computer Programs*	Brian Harvey's Berkeley CS 61A

Subject	Why study?	Book	Videos
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.	*Computer Systems: A Programmer's Perspective*	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 challenging problems.	*The Algorithm Design Manual*	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.	*Mathematics for Computer Science*	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.	*Operating Systems: Three Easy Pieces*	Berkeley CS 162
Computer Networking	The Internet turned out to be a big deal: understand how it works to unlock its full potential.	*Computer Networking: A Top-Down Approach*	Stanford CS 144
Databases	Data is at the heart of most significant programs, but few understand how database systems actually work.	*Readings in Database Systems*	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.	*Crafting Interpreters*	Alex Aiken's course on edX
Distributed Systems	These days, *most* systems are distributed systems.	*Designing Data-Intensive Applications* by Martin Kleppmann	MIT 6.824

If you think these are too much, then just focus on just two books: "*Computer Systems: A Programmer's Perspective*" and "*Designing Data-Intensive Applications* by Martin Kleppmann". The 9 books may take me several years, but these two books may be not.

- Computer Architecture

  first: *Computer Systems: A Programmer's Perspective*

  github exercise: DesmondoRay/CSAPP-3e: Solutions of Computer Systems: A Programmer's Perspective ,Third Edition (github.com)

  then: *Computer Organization and Design*

  video: https://www.youtube.com/watch?v=9y_sUqHeyy8&list=PLhMnuBfGeCDM8p XLpqib90mDFJI-e1lpk

there are so many resources to collect, so I will do it later, now let's make a copy about the original advice.

- ## 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.

  Since this guide was first published in 2016, one of the most commonly asked questions has been whether we'd now recommend recordings of a more recent iteration of 61A taught by John DeNero, and/or the corresponding book *Composing Programs*, which is "in the tradition of SICP" but uses Python. We think the DeNero resources are also great, and some students may end up preferring them, but we still suggest SICP, Scheme, and Brian Harvey's lectures as the first set of resources to try.

  Why? Because SICP is unique in its ability—at least potentially—to alter your fundamental beliefs about computers and programming. Not everybody will experience this. Some will hate the book, others won't get past the first few pages. But the potential reward makes it worth trying.

  If you don't enjoy SICP, try *Composing Programs*. If that still doesn't suit, try *How to Design Programs*. If none of these seem to be rewarding your effort, perhaps that's a sign that you should focus on other topics for some time, and revisit the discipline of programming in another year or two.

  Finally, a point of clarification: this guide is NOT designed for those who are entirely new to programming. We assume that you are a competent programmer without a background in computer science, looking to fill in some knowledge gaps. The fact that we've included a section on "programming" is simply a reminder that there may be more to learn. For those who've never coded before, but who'd like to, you might prefer a guide like this one.

- ## 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.

Our favorite introductory book is *Computer Systems: A Programmer's Perspective* , and a typical introductory computer architecture course using the book would cover most of chapters 1-6.

We love CS:APP for the practical, programmer-oriented approach. While there's much more to computer architecture than what's covered in the book, it serves as a great starting point for those who'd like to understand computer systems primarily in order to write faster, more efficient and more reliable *software* .

For those who'd prefer both a gentler introduction to the topic and a balance of hardware and software concerns, we suggest *The Elements of Computing Systems* , also known as "Nand2Tetris" . This 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, we suggest either returning to CS:APP, or considering 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.

- ## 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 algorithmic problem solving and typically succeeds in fostering similar enthusiasm among his students and readers. In our opinion, the two more commonly suggested texts (CLRS and Sedgewick) tend to be a little too proof-heavy for those learning the material primarily to help with practical problem solving.

For those who prefer video lectures, Skiena generously provides his online. We also really like Tim Roughgarden's course, available on Coursera and elsewhere. Whether you prefer Skiena's or Roughgarden's lecture style will be a matter of personal preference. In fact, there are dozens of good alternatives, so if you happen to find another that you like, we encourage you to stick with it!

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.

- ## 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.

- ## 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 lack of clarity and general student unfriendliness.

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

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*. For Linux, we suggest Robert Love's fantastic Linux Kernel Development.

A great way to consolidate your understanding of operating systems is to read the code of a small kernel and add features. One 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* previously available via Stanford's MOOC platform Lagunita, but sadly now only available as unofficial playlists on Youtube.

- ## 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 recordings 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're adamant about 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.

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.

Our suggested introductory text is the excellent *Crafting Interpreters* by Bob Nystrom, available for free online. It's well organized, highly entertaining, and well suited to those whose primary goal is simply to better understand their languages and language tools. We suggest taking the time to work through the whole thing, attempting

whichever of the "challenges" sustain your interest.

A more traditional recommendation 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.

If you elect to use the Dragon Book, it's almost essential that you cherry-pick the topics, ideally with the help of a mentor. In fact, our suggested way to utilize the Dragon Book, if you so choose, is as a supplementary reference for a video lecture series. Our recommended one is Alex Aiken's, on edX.

- ## 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.

Our suggested book for self-study is Martin Kleppmann's *Designing Data-Intensive Applications* . Far better than a traditional textbook, DDIA is a highly readable book designed for practitioners, which somehow avoids sacrificing depth or rigor.

For those seeking a more traditional text, or who would prefer one that's available for free online, we suggest Maarten van Steen and Andrew Tanenbaum's *Distributed Systems, 3rd Edition* .

For those who prefer video, an excellent course with videos available online is MIT's 6.824, a graduate course taught by Robert Morris with readings available here.

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

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.

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, but with a focus on systems. In our experience, these will be the highest ROI topics for the overwhelming majority of self-taught engineers and bootcamp grads, and provide a solid foundation for further study. Subsequently, 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.

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

When this guide was first written in 2016, the OSS guide had too many subjects, suggested inferior resources for many of them, and provided 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. In the subsequent years, the OSS guide has improved, but we still think that this one provides a clearer, more cohesive path.

freeCodeCamp is focused mostly on programming, not computer science. For why you might want to learn computer science, see above. If you are new to programming, we suggest prioritizing that, and returning to this guide in a year or two.

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, after which you can learn about the libraries/tooling/ecosystem on the job.

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 SICP?

Look, just try it. Some people find SICP mind blowing, a characteristic shared by very few other books. If you don't like it, you can always try something else and perhaps return to SICP later.

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.