What every computer science major should know

Given the expansive growth in the field, it's become challenging to discern what belongs in a modern computer science degree.

My own faculty is engaging in this debate, so I've coalesced my thoughts as an answer to the question, "What should every computer science major know?"

I've tried to answer this question as the conjunction of four concerns:

- What should every student know to get a good job?
- What should every student know to maintain lifelong employment?
- What should every student know to enter graduate school?
- What should every student know to benefit society?

My thoughts below factor into both general principles and specific recommendations relevant to the modern computing landscape.

Computer science majors: feel free to use this as a self-study guide.

Please email or tweet with suggestions for addition and deletion.

Update: Thanks for the suggestions and reminders! I'll incorporate them as I receive them to keep this a living document.

Portfolio versus resume

Having emerged from engineering and mathematics, computer science programs take a resume-based approach to hiring off their graduates.

A **resume** says nothing of a programmer's ability.

Every computer science major should build a **portfolio**.

A portfolio could be as simple as a personal blog, with a post for each project or accomplishment. A better portfolio would include per-project pages, and publicly browsable code (hosted perhaps on github or Google code).

Contributions to open source should be linked and documented.

A code portfolio allows employers to directly judge ability.

GPAs and resumes do not.

Professors should design course projects to impress on portfolios, and students, at the conclusion of each course, should take time to update them.

Examples

- Edward Yang's web site.
- Michael Bradshaw's web site.
- Github is my resume.

Technical communication

Lone wolves in computer science are an endangered species.

Modern computer scientists must practice persuasively and clearly communicating their ideas to non-programmers.

In smaller companies, whether or not a programmer can communicate her ideas to management may make the difference between the company's success and failure.

Unfortunately, this is not something fixed with the addition of a single class (although a solid course in technical communication doesn't hurt).

More classes need to provide students the opportunity to present their work and defend their ideas with oral presentations.

Specific recommendations

I would recommend that students master a presentation tool like PowerPoint or (my favorite) Keynote. (Sorry, as much as I love them, LaTeX-based presentation tools are just too static.)

For producing beautiful mathematical documentation, LaTeX has no equal. All written assignments in technical courses should be submitted in LaTeX.

Recommended reading

- Writing for Computer Science by Zobel.
- Even a Geek Can Speak by Asher.
- The LaTeX Companion.
- The TeXbook by Knuth. (Warning: Experts only.)
- Notes on Mathematical Writing.

- Simon Peyton-Jones's advice on How to Give a Good Research Talk.
- My advice on how to send and reply to email.

An engineering core

Computer science is not quite engineering.

But, it's close enough.

Computer scientists **will** find themselves working with engineers.

Computer scientists and traditional engineers need to speak the same language--a language rooted in real analysis, linear algebra, probability and physics.

Computer scientists ought to take physics through electromagnetism. But, to do that, they'll need take up through multivariate calculus, (and differential equations for good measure).

In constructing sound simulations, a command of probability and (often times) linear algebra is invaluable. In interpreting results, there is no substitute for a solid understanding of statistics.

Recommended reading

- Calculus by Spivak.
- All of Statistics: A Concise Course in Statistical Inference by Wasserman.

The Unix philosophy

Computer scientists should be comfortable with and practiced in the Unix philosophy of computing.

The Unix philosophy (as opposed to Unix itself) is one that emphasizes linguistic abstraction and composition in order to effect computation.

In practice, this means becoming comfortable with the notion of command-line computing, text-file configuration and IDE-less software development.

Specific recommendations

Given the prevalence of Unix systems, computer scientists today should be fluent in basic Unix, including the ability to:

- navigate and manipulate the filesystem;
- compose processes with pipes;
- comfortably edit a file with `emacs` **and** `vim`;
- create, modify and execute a Makefile for a software project;
- write simple shell scripts.

Students will reject the Unix philosophy unless they understand its power. Thus, it's best to challenge students to complete useful tasks for which Unix has a comparative advantage, such as:

- Find the five folders in a given directory consuming the most space.
- Report duplicate MP3s (by file contents, not file name) on a computer.
- Take a list of names whose first and last names have been lower-cased, and properly recapitalize them.
- Find all words in English that have `x` as their second letter, and `n` as their second-to-last.
- Directly route your microphone input over the network to another computer's speaker.
- Replace all spaces in a filename with underscore for a given directory.
- Report the last ten errant accesses to the web server coming from a specific IP address.

Recommended reading

- The Unix Programming Environment by Kernighan and Pike.
- The Linux Programming Interface: A Linux and UNIX System Programming Handbook by Kerrisk.
- Unix Power Tools by Powers, Peek, O'Reilly and Loukides.
- commandlinefu.
- Linux Server Hacks.
- The single Unix specification.

Systems administration

Some computer scientists sneer at systems administration as an "IT" task.

The thinking is that a computer scientist can teach herself how to do anything a technician can do.

This is true. (In theory.)

Yet this attitude is misguided: computer scientists must be able to competently and securely administer their own systems and networks.

Many tasks in software development are most efficiently executed without passing through a systems administrator.

Specific recommendations

Every modern computer scientist should be able to:

- Install and administer a Linux distribution.
- Configure and compile the Linux kernel.
- Troubleshoot a connection with `dig`, `ping` and `traceroute`.

- Compile and configure a web server like apache.
- Compile and configure a DNS daemon like bind.
- Maintain a web site with a text editor.
- Cut and crimp a network cable.

Recommended reading

- UNIX and Linux System Administration Handbook by Nemeth, Synder, Hein and Whaley.

Programming languages

Programming languages rise and fall with the solar cycle.

A programmer's career should not.

While it is important to teach languages relevant to employers, it is equally important that students learn how to teach themselves new languages.

The best way to learn how to learn progamming languages is to learn multiple programming languages and programming paradigms.

The difficulty of learning the **n**th language is half the difficulty of the (**n-1**)th.

Yet, to **truly** understand programming languages, one must implement one. Ideally, every computer science major would take a compilers class. At a minimum, every computer science major should implement an interpreter.

Specific languages

The following languages provide a reasonable mixture of paradigms and practical applications:

- Racket;
- C;
- JavaScript;
- Squeak;
- Java;
- Standard ML;
- Prolog;
- Scala;
- Haskell;
- C++; and
- Assembly.

Racket

Racket, as a full-featured dialect of Lisp, has an aggressively simple syntax.

For a small fraction of students, this syntax is an impediment.

To be blunt, if these students have a fundamental mental barrier to accepting an alien syntactic regime even temporarily, they lack the mental dexterity to survive a career in computer science.

Racket's powerful macro system and facilities for higher-order programming thoroughly erase the line between data and code.

If taught correctly, Lisp liberates.

Recommended reading

- How to Design Programs by Felleisen, Findler, Flatt and Krishnamurthi.
- The Racket Docs.

ANSI C

C is a terse and unforgiving abstraction of silicon.

C remains without rival in programming embedded systems.

Learning C imparts a deep understanding of the dominant von Neumann architecture in a way that no other language can.

Given the intimate role poor C programming plays in the prevalence of the buffer overflow security vulnerabilities, it is critical that programmers learn how to program C properly.

Recommended reading

- ANSI C by Kernighan and Ritchie.

JavaScript

JavaScript is a good representative of the semantic model popular in dynamic, higher-order languages such as Python, Ruby and Perl.

As the native language of the web, its pragmatic advantages are unique.

Recommended reading

- JavaScript: The Definitive Guide by Flanagan.
- JavaScript: The Good Parts by Crockford.
- Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript by Herman.

Squeak

Squeak is a modern dialect of Smalltalk, purest of object-oriented languages.

It imparts the essence of "object-oriented."

Recommended reading

- Introductions to Squeak

Java

Java will remain popular for too long to ignore it.

Recommended reading

- Effective Java by Bloch.

Standard ML

Standard ML is a clean embodiment of the Hindley-Milner system.

The Hindley-Milner type system is one of the greatest (yet least-known) achievements in modern computing.

Though exponential in complexity, type inference in Hindley-Milner is always fast for programs of human interest.

The type system is rich enough to allow the expression of complex structural invariants. It is so rich, in fact, that well-typed programs are often bug-free.

Recommended reading

- ML for the Working Programmer by Paulson.
- The Definition of Standard ML by Milner, Harper, MacQueen and Tofte.

Prolog

Though niche in application, logic programming is an alternate paradigm for computational thinking.

It's worth understanding logic programming for those instances where a programmer may need to emulate it within another paradigm.

Another logic language worth learning is miniKanren. miniKanren stresses pure (cut not allowed) logic programming. This constraint has evolved an alternate style of logic programming called relational programming, and it grants properties not typically enjoyed by Prolog programs.

Recommended reading

- Learn Prolog Now!
- Another tutorial.
- miniKanren.

Scala

Scala is a well-designed fusion of functional and object-oriented programming languages. Scala is what Java should have been.

Built atop the Java Virtual Machine, it is compatible with existing Java codebases, and as such, it stands out as the most likely successor to Java.

Recommended reading

- Programming in Scala by Odersky, Spoon and Venners.
- Programming Scala by Wampler and Payne.

Haskell

Haskell is the crown jewel of the Hindley-Milner family of languages.

Fully exploiting laziness, Haskell comes closest to programming in pure mathematics of any major programming language.

Recommended reading

- Learn You a Haskell by Lipovaca.
- Real World Haskell by O'Sullivan, Goerzen and Stewart.

ISO C++

C++ is a necessary evil.

But, since it must be taught, it must be taught in full.

In particular, computer science majors should leave with a grasp of even template meta-programming.

Recommended reading

- The C++ Programming Language by Stroustrup.
- C++ Templates: The Complete Guide by Vandevoorde and Josuttis.
- Programming Pearls by Bentley.

Assembly

Any assembly language will do.

Since x86 is popular, it might as well be that.

Learning compilers is the best way to learn assembly, since it gives the computer scientist an intuitive sense of how high-level code will be transformed.

Specific recommendations

Computer scientists should understand generative programming (macros); lexical (and dynamic) scope; closures; continuations; higher-order functions; dynamic dispatch; subtyping; modules and functors; and monads as semantic concepts distinct from any specific syntax.

Recommended reading

- Structure and Interpretation of Computer Programs by Abelson, Sussman and Sussman.
- Lisp in Small Pieces by Queinnec.

Discrete mathematics

Computer scientists must have a solid grasp of formal logic and of proof. Proof by algebraic manipulation and by natural deduction engages the reasoning common to routine programming tasks. Proof by induction engages the reasoning used in the construction of recursive functions.

Computer scientists must be fluent in formal mathematical notation, and in reasoning rigorously about the basic discrete structures: sets, tuples, sequences, functions and power sets.

Specific recommendations

For computer scientists, it's important to cover reasoning about:

- trees;
- graphs;
- formal languages; and
- automata.

Students should learn enough number theory to study and implement common cryptographic protocols.

Recommended reading

- How to Prove It: A Structured Approach by Velleman.
- How To Solve It by Polya.

Data structures and algorithms

Students should certainly see the common (or rare yet unreasonably effective) data structures and algorithms.

But, more important than knowing a specific algorithm or data structure (which is usually easy enough to look up), computer scientists must understand how to design algorithms (e.g., greedy, dynamic strategies) and how to span the gap between an algorithm in the ideal and the nitty-gritty of its implementation.

Specific recommendations

At a minimum, computer scientists seeking stable long-run employment should know all of the following:

- hash tables;
- linked lists;
- trees;
- binary search trees; and
- directed and undirected graphs.

Computer scientists should be ready to implement or extend an algorithm that operates on these data structures, including the ability to search for an element, to add an element and to remove an element.

For completeness, computer scientists should know both the imperative and functional versions of each algorithm.

Recommended reading

- CLRS.
- Any of the Art of Computer Programming series by Knuth.

Theory

A grasp of theory is a prerequisite to research in graduate school.

Theory is invaluable when it provides hard boundaries on a problem (or when it provides a means of circumventing what initially appear to be hard boundaries).

Computational complexity can legitimately claim to be one of the few truly predictive theories in all of computer "science."

A computer scientist **must** know where the boundaries of tractability and computability lie. To ignore these limits invites frustration in the best case, and failure in the worst.

Specific recommendations

At the undergraduate level, theory should cover at least models of computation and computational complexity.

Models of computation should cover finite-state automata, regular languages (and regular expressions), pushdown automata, context-free languages, formal grammars, Turing machines, the lambda calculus, and undecidability.

At the undergraduate level, students should learn at least enough complexity to understand the difference between P, NP, NP-Hard and NP-Complete.

To avoid leaving the wrong impression, students should solve a few large problems in NP by reduction to SAT and the use of modern SAT solvers.

Recommended reading

- Introduction to the Theory of Computation by Sipser.
- Computational Complexity by Papadimitriou.
- Algorithms by Sedgewick and Wayne.
- Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

Architecture

There is no substitute for a solid understanding of computer architecture.

Computer scientists should understand a computer from the transistors up.

The understanding of architecture should encompass the standard levels of abstraction: transistors, gates, adders, muxes, flip flops, ALUs, control units, caches and RAM.

An understanding of the GPU model of high-performance computing will be important for the foreseeable future.

Specific recommendations

A good understanding of caches, buses and hardware memory management is essential to achieving good performance on modern systems.

To get a good grasp of machine architecture, students should design and simulate a small CPU.

Recommended reading

- nand2tetris, which constructs a computer from the ground up.
- Computer Organization and Design by Patterson and Hennessy.
- "What every programmer should know about memory" by Drepper.

Operating systems

Any sufficiently large program eventually becomes an operating system.

As such, computer scientists should be aware of how kernels handle system calls, paging, scheduling, context-switching, filesystems and internal resource management.

A good understanding of operating systems is secondary only to an understanding of compilers and architecture for achieving performance.

Understanding operating systems (which I would interpret liberally to include runtime systems) becomes especially important when programming an embedded system without one.

Specific recommendations

It's important for students to get their hands dirty on a real operating system. With Linux and virtualization, this is easier than ever before.

To get a better understanding of the kernel, students could:

- print "hello world" during the boot process;
- design their own scheduler;
- modify the page-handling policy; and
- create their own filesystem.

Recommended reading

- Linux Kernel Development by Love.

Networking

Given the ubiquity of networks, computer scientists should have a firm understanding of the network stack and routing protocols within a network.

The mechanics of building an efficient, reliable transmission protocol (like TCP) on top of an unreliable transmission protocol (like IP) should not be magic to a computer scientist. It should be core knowledge.

Computer scientists must understand the trade-offs involved in protocol design--for example, when to choose TCP and when to choose UDP. (Programmers need to understand the larger social implications for congestion should they use UDP at large scales as well.)

Specific recommendations

Given the frequency with which the modern programmer encounters network programming, it's helpful to know the protocols for existing standards, such as:

- 802.3 and 802.11;
- IPv4 and IPv6; and
- DNS, SMTP and HTTP.

Computer scientists should understand exponential back off in packet collision resolution and the additive-increase multiplicative-decrease mechanism involved in congestion control.

Every computer scientist should implement the following:

- an HTTP client and daemon;
- a DNS resolver and server; and
- a command-line SMTP mailer.

No student should ever pass an intro neworking class without sniffing their instructor's Google query off wireshark.

It's probably going too far to require all students to implement a reliable transmission protocol from scratch atop IP, but I can say that it was a personally transformative experience for me as a student.

Recommended reading

- Unix Network Programming by Stevens, Fenner and Rudoff.

Security

The sad truth of security is that the majority of security vulnerabilities come from sloppy programming. The sadder truth is that many schools do a poor job of training programmers to secure their code.

Computer scientists must be aware of the means by which a program can be compromised.

They need to develop a sense of defensive programming--a mind for thinking about how their own code might be attacked.

Security is the kind of training that is best distributed throughout the entire curriculum: each discipline should warn students of its native vulnerabilities.

Specific recommendations

At a minimum, every computer scientist needs to understand:

- social engineering;
- buffer overflows;
- integer overflow;
- code injection vulnerabilities;
- race conditions; and
- privilege confusion.

A few readers have pointed out that computer scientists also need to be aware of basic IT security measures, such how to choose legitimately good passwords and how to properly configure a firewall with iptables.

Recommended reading

- Metasploit: The Penetration Tester's Guide by Kennedy, O'Gorman, Kearns and Aharoni.
- Security Engineering by Anderson.

Cryptography

Cryptography is what makes much of our digital lives possible.

Computer scientists should understand and be able to implement the following concepts, as well as the common pitfalls in doing so:

- symmetric-key cryptosystems;
- public-key cryptosystems;
- secure hash functions;
- challenge-response authentication;
- digital signature algorithms; and
- threshold cryptosystems.

Since it's a common fault in implementations of cryptosystems, every computer scientist should know how to acquire a **sufficiently** random number for the task at hand.

At the very least, as nearly every data breach has shown, computer scientists need to know how to salt and hash passwords for storage.

Specific recommendations

Every computer scientist should have the pleasure of breaking ciphertext using pre-modern cryptosystems with hand-rolled statistical tools.

RSA is easy enough to implement that everyone should do it.

Every student should create their own digital certificate and set up https in apache. (It's surprisingly arduous to do this.)

Student should also write a console web client that connects over SSL.

As strictly practical matters, computer scientists should know how to use GPG; how to use public-key authentication for ssh; and how to encrypt a directory or a hard disk.

Recommended reading

- Cryptography Engineering by Ferguson, Schneier and Kohno.

Software testing

Software testing must be distributed throughout the entire curriculum.

A course on software engineering can cover the basic styles of testing, but there's no substitute for practicing the art.

Students should be graded on the test cases they turn in.

I use test cases turned in by students against all other students.

Students don't seem to care much about developing defensive test cases, but they unleash hell when it comes to sandbagging their classmates.

User experience design

Programmers too often write software for other programmers, or worse, for themselves.

User interface design (or more broadly, user experience design) might be the most underappreciated aspect of computer science.

There's a misconception, even among professors, that user experience is a "soft" skill that can't be taught.

In reality, modern user experience design is anchored in empirically-wrought principles from human factors engineering and industrial design.

If nothing else, computer scientists should know that interfaces need to make the ease of executing any task proportional to the frequency of the task multiplied by its importance.

As a practicality, every programmer should be comfortable with designing usable web interfaces in HTML, CSS and JavaScript.

Recommended reading

- Paul Graham's essay on Web 2.0.
- "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets" by Spolsky.
- HTML and CSS: Design and Build Websites by Duckett.
- JavaScript: The Definitive Guide by Flanagan.

Visualization

Good visualization is about rendering data in such a fashion that humans perceive it as information. This is not an easy thing to do.

The modern world is a sea of data, and exploiting the local maxima of human perception is key to making sense of it.

Recommended reading

- The Visual Display of Quantitative Information by Tufte.

Parallelism

Parallelism is back, and uglier than ever.

The unfortunate truth is that harnessing parallelism requires deep knowledge of architecture: multicore, caches, buses, GPUs, etc.

And, practice. Lots of practice.

Specific recommendations

It is not at all clear what the "final" answer on parallel programming is, but a few domain-specific solutions have emerged.

For now, students should learn CUDA and OpenCL.

Threads are a flimsy abstraction for parallelism, particularly when caches and cache coherency are involved. But, threads are popular and tricky, so worth learning. Pthreads is a reasonably portable threads library to learn.

For anyone interested in large-scale parallelism, MPI is a prerequisite.

On the principles side, it does seem that map-reduce is enduring.

Software engineering

The principles in software engineering change about as fast as the programming languages do.

A good, hands-on course in the practice of team software construction provides a working knowledge of the pitfalls inherent in the endeavor.

It's been recommended by several readers that students break up into teams of three, with the role of leader rotating through three different projects.

Learning how to attack and maneuver through a large existing codebase is a skill most programmers will have to master, and it's one best learned in school instead of on the job.

Specific recommendations

All students need to understand centralized version control systems like svn and distributed version control systems like git.

A working knowlege of debugging tools like gdb and valgrind goes a long way when they finally become necessary.

Recommended reading

- Version Control by Example by Sink.

Formal methods

As the demands on secure, reliable software increase, formal methods may one day end up as the only means for delivering it.

At present, formal modeling and verification of software remains challenging, but progress in the field is steady: it gets easier every year.

There may even come a day within the lifetime of today's computer science majors where formal software construction is an expected skill.

Every computer scientist should be at least moderately comfortable using one theorem prover. (I don't think it matters which one.)

Learning to use a theorem prover immediately impacts coding style.

For example, one feels instinctively allergic to writing a `match` or `switch` statement that doesn't cover all possibilities.

And, when writing recursive functions, users of theorem provers have a strong urge to eliminate ill-foundedness.

Recommended reading

- Software Foundations.

Graphics and simulation

There is no discipline more dominated by "clever" than graphics.

The field is driven toward, even defined by, the "good enough."

As such, there is no better way to teach clever programming or a solid appreciation of optimizing effort than graphics and simulation.

Over half of the coding hacks I've learned came from my study of graphics.

Specific recommendations

Simple ray tracers can be constructed in under 100 lines of code.

It's good mental hygiene to work out the transformations necessary to perform a perspective 3D projection in a wireframe 3D engine.

Data structures like BSP trees and algorithms like z-buffer rendering are great examples of clever design.

In graphics and simulation, there are many more.

Recommended reading

- Mathematics for 3D Game Programming and Computer Graphics by Lengyel.

Robotics

Robotics may be one of the most engaging ways to teach introductory programming.

Moreover, as the cost of robotics continues to fall, thresholds are being passed which will enable a personal robotics revolution.

For those that can program, unimaginable degrees of personal physical automation are on the horizon.

Related posts

- Multitouch gesture control for a robot.

Artificial intelligence

If for no other reason than its outsized impact on the early history of computing, computer scientists should study artificial intelligence.

While the original dream of intelligent machines seems far off, artificial intelligence spurred a number of practical fields, such as machine learning, data mining and natural language processing.

Recommended reading

- Artificial Intelligence by Russell and Norvig.

Machine learning

Aside from its outstanding technical merits, the sheer number of job openings for "relevance engineer," indicates that every computer scientist should grasp the fundamentals of machine learning.

Machine learning doubly emphasizes the need for an understanding of probability and statistics.

Specific recommendations

At the undergraduate level, core concepts should include Bayesian networks, clustering and decision-tree learning.

Recommended reading

- Machine Learning by Mitchell.

Databases

Databases are too common and too useful to ignore.

It's useful to understand the fundamental data structures and algorithms that power a database engine, since programmers often enough reimplement a database system within a larger software system.

Relational algebra and relational calculus stand out as exceptional success stories in sub-Turing models of computation.

Unlike UML modeling, ER modeling seems to be a reasonable mechanism for visualing encoding the design of and constraints upon a software artifact.

Specific recommendations

A computer scientist that can set up and operate a LAMP stack is one good idea and a lot of hard work away from running their own company.

Recommended reading

- SQL and Relational Theory by Date.

Non-specific reading recommendations

- Gödel, Escher, Bach by Hofstadter.
- Nick Black's advice for MS students.

What else?

My suggestions are limited by blind spots in my own knowledge.

What have I not listed here that should be included?

Related posts

- HOWTO: Get tenure
- Parsing BibTeX into S-Expressions, JSON, XML and BibTeX
- PAANDA: An NDA for academics
- College tips, tricks and hacks
- Tips for defending a Ph.D.
- HOWTO: Respond to peer reviews
- 12 resolutions for grad students
- HOWTO: Peer review scientific work
- Electric meat
- Peer fortress: The scientific battlefield
- The shape of your problem
- 6 tips for low-cost academic blogging
- The illustrated guide to a Ph.D.
- The CRAPL: An open source license for academia
- Why peer reviewers should use TOR
- Academic job hunt advice

[article index] [email me] [@mattmight] [rss]

Latest: HOWTO: Get tenure
Next: Writing CEK-style interpreters in Haskell
Prev: Boost productivity: Cripple your technology
Rand: Tips for defending a Ph.D.