The Complete Computer Engineer

The title is a bit audacious. *But there are some essential ideas and skills that every computer engineer should possess.*

Spend the next two to four years working on these ideas.

Do not be limited by your courses.

"I never let my schooling interfere with my education."

— Mark Twain

Four Intersecting Questions

- 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?

Portfolio vs. Resume

A resumé says nothing of a programmer's ability.

Every computer engineering student should build a portfolio.

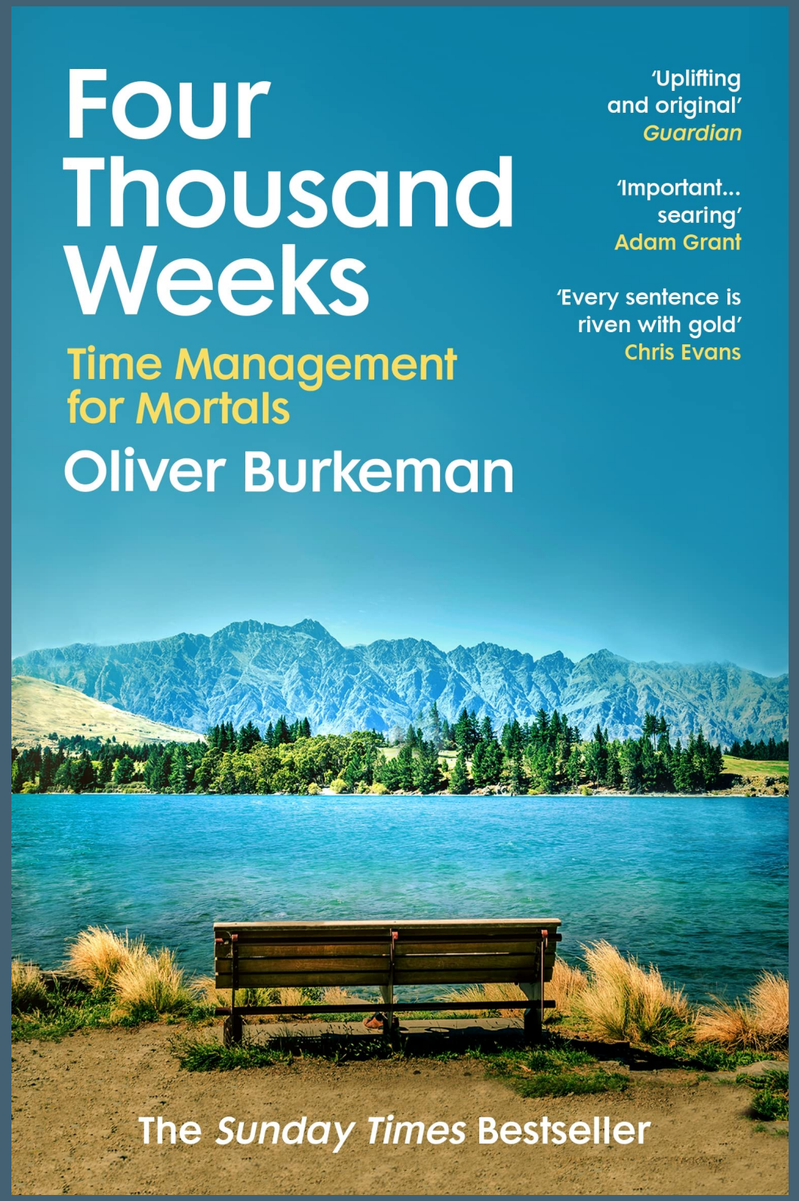
Examples

- Muchen He's web site
- Github is my resume

Time Management

» You have a limited amount of time. Use it well.

» Think in terms of weekly goals over your life.


Four Thousand Weeks
Time Management for Mortals
Oliver Burkeman
'Uplifting and original'
Guardian
'Important... searing'
Adam Grant
'Every sentence is riven with gold'
Chris Evans
The Sunday Times Bestseller

Quality … Quantity?

Sometimes, you have to do a thing many times over.

A story from Art & Fear by Bayles and Orland:

The ceramics teacher announced on opening day that he was dividing the class into two groups. All those on the left side of the studio, he said, would be graded solely on the quantity of work they produced, all those on the right solely on its quality.

His procedure was simple: on the final day of class he would bring in his bathroom scales and weigh the work of the "quantity" group: fifty pound of pots rated an "A", forty pounds a "B", and so on. Those being graded on "quality", however, needed to produce only one pot — albeit a perfect one — to get an "A".

Well, came grading time and a curious fact emerged: the works of highest quality were all produced by the group being graded for quantity. It seems that while the "quantity" group was busily churning out piles of work — and learning from their mistakes — the "quality" group had sat theorizing about perfection, and in the end had little more to show for their efforts than grandiose theories and a pile of dead clay.

Communication Skills

Lone wolves in engineering are an endangered species.

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

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

Recommendations for Communication

I would recommend that students master a presentation tool like PowerPoint or Keynote. (Sorry, as much as I love LaTeX, presentation tools based on LaTeX are static.)

For producing beautiful mathematical documentation, LaTeX has no equal. All written assignments in technical courses should be submitted in LaTeX. These days, Markdown + `pandoc` does come close.

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.

An Engineering Core

- Calculus
- Linear Algebra
- Probability
- Physics all the way up to some electromagnetics

Recommended Reading

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

The Unix philosophy

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` or `vim`;

» create, modify and execute a Makefile for a software project;

» write simple shell scripts.

Some Command Line Challenges

- 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

Computer engineers 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

- 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.

Specific Languages

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

Racket; C; JavaScript/TypeScript; Squeak; Java;
Standard ML; Prolog; Scala; Haskell;
C++; Assembly Language.

Racket

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

ANSI C

ANSI C by Kernighan and Ritchie.

JavaScript

- 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

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. 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.

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.

Java

Effective Java by Bloch.

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.

Discrete Mathematics

Computer engineers 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 engineers 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

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.

How to Solve It: Modern Heuristics by Michalewicz and Fogel.

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 engineers 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 engineers seeking stable long-run employment should know all of the following:

» hash tables;

» linked lists;

» trees;

» binary search trees; and

» directed and undirected graphs.

Computational 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."

Computer Architecture

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

Computer engineers should understand a computer from the transistors up.

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.

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.

Computer Networks

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

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.

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

Every student should implement the following:

» an HTTP client and daemon;

» a DNS resolver and server; and

» a command-line SMTP mailer.

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.

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.

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.

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

Computer engineers 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.

Specific Recommendations

Every computer engineer 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.)

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

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.

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 systems design.

Visualization

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.

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.

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.

Other Topics

Graphics and Simulation
Robotics
Artificial Intelligence
Machine Learning
Databases

Ethics
Philosophy

Non-specific Recommendations

Gödel, Escher, Bach by Hofstadter.

The Sciences of the Artificial by Simon.

Technology and Courage by Sutherland.

Caveat

My suggestions are limited by my own blindspots.