What Should an Educated Person Know about Computers?

Brian W. Kernighan, Department of Computer Science, Princeton University, bwk@cs.princeton.edu

Introduction

Since the fall of 1999, I have been teaching a course at Princeton called "Computers in Our World"[1] that attempts to convey the important ideas of computers and communications, which are such a pervasive part of our lives. Some computing is highly visible: every student has a computer (each of which is far more powerful than the one that served the whole campus when I was a graduate student in 1964). Everyone has high speed Internet access, everyone uses email to keep in touch with friends and family, and we all shop and search online.

But this is a tiny part of a computing iceberg, most of which lies hidden below the surface. We don't see, and usually don't think about, the computers that lurk within appliances and cars and airplanes and the ubiquitous electronic gadgets -- cell phones, cameras, music players, games -- that we take for granted.

Nor do we think much about the degree to which infrastructure like the telephone system, air traffic control and the power grid depends on computing. Although we are from time to time reminded of the growth of surveillance systems, invasions of our privacy, and the perils of electronic voting, we perhaps do not realize the extent to which those are enabled by computing and communications.

Most people will not be directly involved in creating such systems, but everyone is strongly affected by them, and some people will be required to make important decisions about them. The students in my class are for the most part not technical; their primary interests are in the humanities and social sciences. But I believe that any educated person ought to know at least the rudiments of what computers can do and how they do it; what they can't do at all, and what's merely hard; and how they relate to the world around us. An educated person should be able to read and understand a newspaper article about computing, to learn more from it, and perhaps to spot places where it is not accurate. More broadly, I want my students to be intelligently skeptical about technology, and able to reason about its good and bad aspects. Realistically, many of the important decisions in our world are made by people who are not particularly technical in background. Surely it would be good if everyone had a decent understanding of crucial technologies like computing.

Three Topics, Three Big Ideas

What should an educated person know about computing? Everyone will have their own idea; my view focuses on three core pieces: hardware, software, and communications.

[1]www.cs.princeton.edu/courses/archive/fall07/cos109

Hardware is the tangible part, the computers that we can see and touch in our homes and offices. The main idea here is that computers are general purpose devices that process a universal digital representation of information: everything they work with, even the instructions that tell them what to do, is ultimately just numbers, usually expressed as bits -- zeros and ones.

Software -- the instructions that programmers write to control computers -- is by contrast hardly tangible at all, but it's what makes computers do things for us. Here, the idea is that programs describe computation in excruciatingly detailed steps, each one of which must be perfect for correct operation. But programmers are not perfect, so all software has errors, even software that must work perfectly.

Communications means computers talking to each other on our behalf: the Internet and the Web and email and chat and file-sharing. All of our computers, and increasingly our other gadgets, are connected by a universal digital network that moves the universal digital representation of information among universal digital processors. The network hides the differences among myriad different kinds of equipment at all levels, so almost anything can be connected, and ultimately everything will be.

Hardware

The basic structure of computers was more or less understood by Charles Babbage in the 1830s, though he was never able to complete one of his mechanical engines (see Figure 1). ([Swore] is a very good treatment of Babbage's work.) Figure 2 is a picture of Babbage.

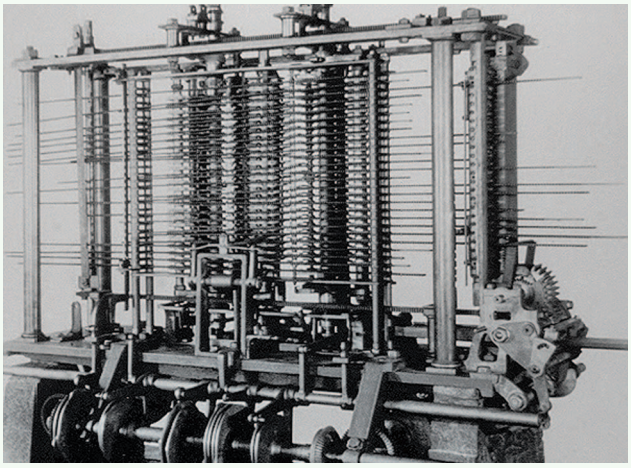


Figure 1. Babbage's Differential Engine (Source: www.msu.edu/course/lbs/126/lectures/images/babbage.jpg)

The clearest modern statement of how a computer works is found in the classic paper "Preliminary discussion of the logical design of an electronic computing

Figure 2. Charles Babbage (Source://commons.wikimedia.org/wiki/Image:Charles_Babbage.jpg)

instrument" by Burks, Goldstine, and von Neumann [BGvN, 1946], which is well worth reading even today.

The so-called von Neumann architecture has a processor (the CPU in today's terminology) that does logic and computation and controls the rest of the machine, a memory for instructions and data, and other devices for storing or communicating information. The crucial idea is that instructions that tell the computer what to do are encoded as numbers and stored in the same memory as the data being processed. This is why the computer is a general purpose device: change the numbers that tell it what to do and it does something different.

Computers are digital devices: they deal with numbers, and nothing else. Though we live in an analog world, digital is much simpler to work with -- it's just plain easier to make devices that only have two states: voltage high or low, current flowing or not, charged or uncharged, magnetized up or down, reflectance high or low, and so on. The abstraction of all these two-state physical systems is captured in the two binary digits 0 and 1. No matter what the particular physical representation, we can encode and process information in combinations of bits. The many different kinds of analog devices of earlier times -- long playing records, photographic film, VCR tapes, and so on -- have converged on numeric representations as a common denominator. The modern trend is strongly towards converting information from analog to digital as early as possible in any system, and converting back to analog as late as possible.

How can information be reduced to numbers? Consider music: If we sample a sound waveform often enough and accurately enough, we will generate a sequence of numbers (a voltage level, perhaps) that can be used to reproduce the original accurately. Nyquist's theorem tells us that sampling a waveform (see Figure 3) at twice the highest frequency it contains is enough to capture all the information in the waveform; thus the sampling rate of 44,100 samples per second used in audio CD's is adequate to capture the roughly 20 KHz range of human hearing. The samples are usually measured to 16 bits of accuracy, that is, 65,536 distinct levels.

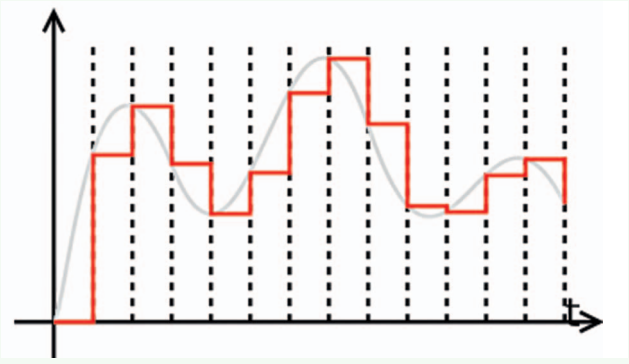


Figure 3. Sampling a Waveform (Source: upload.wikimedia.org/wikipedia/commons/thumb/1/15/Zeroorderhold.signal.svg/400px- Zeroorderhold.signal.svg.png)

Similarly, if an array of closely spaced photocells samples the intensity of light at different wavelengths, that too produces numbers that capture an image and that can be used to reproduce it later; this is the basis of digital cameras. Movies and TV are just a sequence of pictures with sound, so they too are readily represented as numbers. Text of any kind is straightforward: assign a different number to each letter or character. And so on: ultimately a digital representation is easy.

The logical structure (the architecture) of a computer as set forth in the von Neumann paper has not changed significantly since 1946, but the physical forms have evolved amazingly, from mercury delay lines and vacuum tubes to integrated circuits with billions of transistors on a chip. Most of the progress has come about because components are so much smaller, cheaper and faster. In 1956, Gordon Moore observed that the number of devices that could be placed on an integrated circuit was doubling rapidly and predicted that this growth would continue (see Figure 4.) Moore's Law, by now a sort of self-fulfilling prophecy, says that every year or two, things will be twice as good. If computing power doubles every 18 months, that is a factor of a thousand (since 2^{10} is 1024) in 15 years, and a million (2^{20}) in 30 years. Moore's Law has held for 45 years now, so we are indeed a billion times better off computationally than we were in 1960.

In 1967, Gene Amdahl (Figure 5) [Amdahl] explored the relationship between performance and (among other things) multiple processors. One of the most recent advances in computer hardware, the development of processors with multiple "cores" or CPUs on a single chip, brings Amdahl's work back as a central concern. Multiple cores are standard in consumer laptops today, but it is an open problem how to make the best use of this architecture, both from the hardware standpoint and from a software perspective.

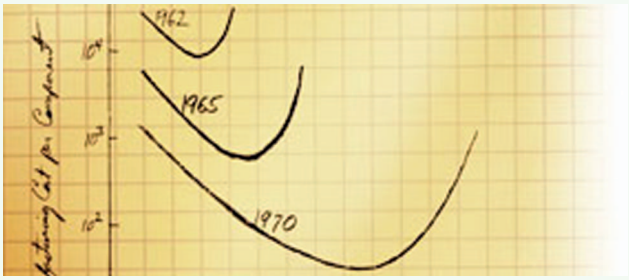


Figure 4. Moore's original graph and Intel's site about Moore's Law (Source: www.intel.com/pressroom/kits/ events/ moores_law_40th/index.htm)

Figure 5. Gene Amdahl (Courtesy of Dr. Amdahl)

Of course computers can't do everything, nor are they arbitrarily fast. There are practical and theoretical limits to how fast we can compute. Alan Turing (see Figure 6) showed in the 1930s [Turing] that all computers have the same computational power, in the sense that they can all compute the same functions (arguing by simulation: a "universal Turing machine" that could simulate any other computer) and he also demonstrated classes of computations that could not be performed in any reasonable amount of time. And naturally digital computers can't help if the problem can't be expressed in terms of numeric computations.

Software

By itself, computer hardware doesn't do much; it needs something to tell it what to do. "Software" is the general term for sets of instructions that make a computer do something useful. It's "soft" by comparison with "hard" hardware because it's intangible, not easy to put your hands on. Hardware is quite tangible: if you drop a computer on your foot, you'll notice. Not true for software.

There is a strong tendency today to use general purpose hardware wherever possible -- a processor, a memory, and suitable peripherals -- and create specific behaviors by means of software. The conventional wisdom is that software is more flexible, easier to change (especially once some device has left the factory), and cheaper. In fact, all of these presumed advantages are somewhat debatable, but the trend is there anyway. For example, if a computer program controls the way that power and brakes are applied to the drive wheels of a car, then apparently different features like anti-lock braking and electronic stability control can be implemented in software, since they are just different ways of controlling the power to the wheels.

Figure 6. Alan Turing (Source: www.bletchleypark.org.uk/edu/ lectures/turing.rhtm)

A popular metaphor for explaining software compares it to recipes for cooking. A recipe spells out the ingredients needed to make some dish and the sequence of operations that the cook has to perform. By analogy, a program needs certain data to operate on and it spells out what to do to the data. Real recipes are much more vague and ambiguous than programs could ever be, however, so the analogy is not good. Tax forms are better: they spell out in painful detail what to do ("Subtract line 30 from line 29. If zero or less, enter 0. Multiply line 31 by 25%, ...") The analogy is still imperfect, but tax forms better capture the computational aspects -- performing arithmetic operations, copying data from one place to another, and having values and computational steps depend on earlier ones -- and show more of the need to be precise and cover all possible cases.

An algorithm is the computer science version of a careful, precise, unambiguous recipe or tax form, a sequence of steps that is guaranteed to perform some computation correctly. Each step is expressed in terms of basic operations whose meaning is completely specified, for example "add two numbers". There's no ambiguity about what any operation means. The input data is unambiguous. And all possible situations are covered; the algorithm never encounters a situation where it doesn't know what to

do next. (Computer scientists add one more condition: the algorithm has to stop eventually, so the classic shampoo instruction to "Lather, Rinse, Repeat" is not an algorithm.)

One crucial aspect of both algorithms and programs is how efficiently they operate -- how long they are likely to run in proportion to the amount of data to be processed. Algorithm analysis is an active research area in computer science, but most algorithms in day to day life are "linear" -- the amount of work is directly or linearly proportional to the amount of data. Others are faster. For instance, searching for a specific item in a sorted list of n items can be done in time proportional to log n, by a divide and conquer strategy that mimics how we look up names in a phone book. Sorting itself takes n log n time to sort n items into order.

Although in general the goal is to find the fastest algorithm, some crucial processes depend on the apparent impossibility of finding a fast algorithm. For example, public-key cryptography, which is the basis of the security of electronic commerce, digital signatures, and the like, is based most often on the difficulty of factoring large (hundreds of digits) composite integers. So far as we know, the time to factor such numbers grows exponentially with their lengths, thus making it computationally infeasible to crack encryption schemes by brute force computation. But if some advance in mathematics or quantum computing renders factoring easy, this whole edifice will collapse.

Algorithms capture the abstract notion of how to perform a task, but real computers are not abstract, and they need detailed concrete instructions to proceed. Programming languages are artificial languages that (to varying degrees) make it easy to express the steps of a computation in a way that people can understand that can also be converted into a form that computers understand.

Early programming was done in so-called assembly languages. Assembly languages are closely tied to specific machines: each language expresses computation in terms of the instruction repertoire that the machine itself understands. For example, in one kind of machine, incrementing the value stored in a memory location M might be accomplished by three assembly language instructions like this:

```
LOAD   M
ADD    1
STORE  M
```

It is very hard to write programs at this level, and the programs are tied forever to the specific architectures. Moving to another machine means rewriting the programs; the same increment operation on a different machine might be expressed as

```
ADD   M, 1, M
```

and as

```
INCR   M
```

on a third.

Arguably the most important step in software was taken during the late 1950s and early 1960s, with the development of "high level" programming languages like Fortran ("Formula Translation", [Backus]), which were independent of any specific CPU type. Fortran was developed at IBM by a team led by John Backus (see Figure 7). Such languages made it easier for programmers to describe a computation, and, once written, the program could be translated (by a program!) into specific instructions for a target machine. This made programming accessible to a much wider population, and also greatly reduced the need to rewrite code to make it work on different kinds of machines. All of the sequences above would be written in Fortran as the single statement

```
M = M + 1
```

which the Fortran compiler would translate into the right instructions for whatever machine was being used.

Figure 7. John Backus (Source: //blog.hundhausen.com/files/johnbackus.jpg)

Programming languages continue to evolve; today's languages are more expressive and closer to the way that people think about computing processes compared to early Fortran, though they are still not "natural" in any sense. (Steve Lohr's Go To:... [Lohr] is an excellent discussion of programming languages.) The higher level the language, that is, the closer to our level, the more translation is needed, and perhaps the more machine resources are "wasted," but as computers have gotten faster thanks to Moore's Law, this overhead has become less and less relevant. By today's standards, programmers are very expensive, but computers are free.

What do we build with programming languages? Operating systems like Windows or Mac OS X or Unix/Linux control the hardware, managing its resources and providing a platform on which application programs like browsers and office suites and games and music players can run. Operating systems are complex and expensive to produce; Vista, Microsoft's most recent version, took at least five years and many thousands of people to create. Previous versions of Windows have been tens of millions of lines of source code; Vista is presumably bigger.

Other systems are large as well; for instance, recent distributions of the Linux kernel of the Linux are well over 7 million lines. Such systems are among the most complicated artifacts that we create.

Applications need some kind of platform that provides services, like a file system and network connections, and coordinates activities so that independent processes do not interfere with each other. Historically, that platform has been a conventional operating system, but one of the most interesting recent trends has been towards "middleware," most commonly a browser, that acts as a platform for applications while insulating them from the details of a specific operating system. Google Maps is a good example, as are web-based mail systems and the nascent area of web-based office tools. Naturally Microsoft is concerned about this trend, which is a threat to its commanding presence as the operating system supplier for the vast majority of home and office computers.

Sadly, no program works the first time, so a big part of the job of real-world programming is to test code as it's being written and certainly before it is shipped to its users, with the hope of getting rid of as many bugs as possible. A rule of thumb says that there is at least one bug for every thousand lines of code, so even if this is too pessimistic by an order of magnitude, large systems have thousands of residual bugs.

Another complexity in real-world software is that things change continuously, and programs have to be adapted. New hardware is developed; it needs new drivers and those may require changes in systems before they work properly. New laws and other requirements change the logic of programs -- think about what has to happen every time the tax code changes, for example. Machines and tools and languages become obsolete and have to be replaced. Expertise disappears too, as people retire, lose interest, or get fired in some corporate down-sizing. (Student-created systems at universities suffer in the same way when the expertise graduates.)

No matter what, software is hard to write, at least to the standards of correctness and reliability necessary for critical systems like avionics, medical equipment, military systems, automobile control, and so on. It is possible to create reliable software, but only at very high cost, and even then no system is perfect. How to write robust software economically is the biggest open problem in computing.

Software also raises some interesting legal issues. Historically, patents could be obtained only for mechanical devices and processes, but in the 1970s it became possible to obtain patent protection for software, and in the 1990s this was pushed much further by "business method" patents like Amazon's One-click technique for making an online purchase. To programmers, such "inventions" often seem utterly obvious, but that has not slowed the rate of patents or patent litigation. Liability for defective software is another area that is likely to become more important, though so far most software vendors have managed to sidestep this issue.

Communications

Communications means computers talking to each other, usually to do something useful for us, but sometimes up to no good at all. Most interesting systems now combine hardware, software, and communications, with the Internet serving as a universal "common carrier" that conveys the universal digital representation of information among universal digital processors. Communicating systems also give rise to most of computing's societal issues: difficult problems of privacy, security, and the conflicting rights of individuals, businesses and governments.

The Internet began [ISOC] with research into survivable networks in the 1960s, sponsored by the US Department of Defense; arguably this was one of the most productive uses of military money ever. The Internet remained the province of scientists and engineers at universities and research labs until the combination of ubiquitous personal computers, decent bandwidth, and the World Wide Web invented by Tim Berners-Lee (see Figure 8) in the early 1990s caused an explosion of use.

Figure 8. Sir Tim Berners-Lee, inventor of the World Wide Web (Source://blogs.zdnet.com/images/bernerslee400.jpg)

The role of the Internet is to connect a large number of local area networks, so that information originating on one network can find its way to any other local network no matter where it is. The genius of the Internet is that a comparative handful of protocols -- rules for how systems interact -- developed in the early 1970s have made it possible to connect a wide variety of different networking technologies, from phone lines to fiber optic cables, while hiding the specific properties of individual devices and networks.

There are only a handful of basic ideas behind the Internet. First, it is a packet network: information is sent in individual independent packets that are routed through a large and changing collection of networks. Each packet consists of a header that contains, in addition to the data itself, information like the source and destination, the packet length, the protocol version, and a very limited amount of checking. This is a different model from the telephone system's

circuit network, where each conversation has a dedicated circuit, conceptually a private wire, between the two talking parties.

Each packet travels through multiple routers that connect networks; each router passes the packet to a network that is closer to the packet's ultimate destination (see Figure 9). Routers continuously exchange routing information, so they always know how to move a packet closer to its destination, even as topology changes and network connections come and go. As a packet travels from here to there, it might easily pass through 20 routers, owned and operated by a dozen different companies or institutions.

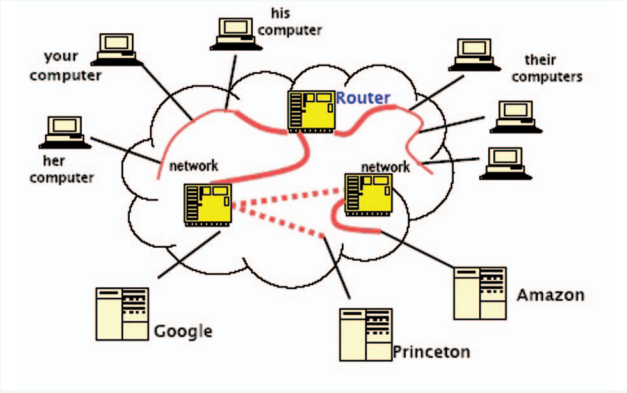


Figure 9. Internet Cloud (Source: www.cs.princeton.edu/~bwk/cloud.jpg)

Each computer currently connected to the Internet is assigned a unique 32-bit Internet Protocol ("IP") address; hosts on the same network share a common IP address prefix. The Domain Name System is a large distributed data base that converts names like google.com or ieee.org to IP addresses. A central authority (ICANN, the Internet Corporation for Assigned Names and Numbers) allocates a block of IP addresses to a network managed by some organization. Each host address on that network is then assigned locally by the organization. Thus, for example, IEEE has been allocated blocks of IP addresses that it can in turn allocate to subnetworks and computers within IEEE. ICANN is also ultimately responsible for allocating other resources that must necessarily be unique, like top-level domain names themselves.

The IP packet mechanism is an unreliable "best effort" network. The Transmission Control Protocol (TCP) uses redundancy, sequence numbers, acknowledgements and timeouts to synthesize a reliable two-way stream from the unreliable IP packets: TCP packets are wrapped up in a sequence of IP packets that can be used to achieve very high reliability. Most of the higher-level services that we associate with the Internet -- the Web itself, email, chat, file-sharing, telephony, and so on -- use TCP.

IP itself uses whatever networking technology gets the information from the current router to the next one on the path. Specific hardware technologies like Ethernet encapsulate IP packets as they move around, but the details of how any particular piece of hardware works, or even that such hardware is involved, are not visible at the IP level or above.

The protocols divide the software into layers, each of which provides services to the next higher level while calling on the services of the next lower level (see Figure 10). At each level of the protocol hierarchy, software behaves as if it is talking to a peer at the same level at the other end, independent of lower layers. This strict layering is fundamental to the operation of the Internet, a way to organize and control complexity and hide irrelevant details of implementation.

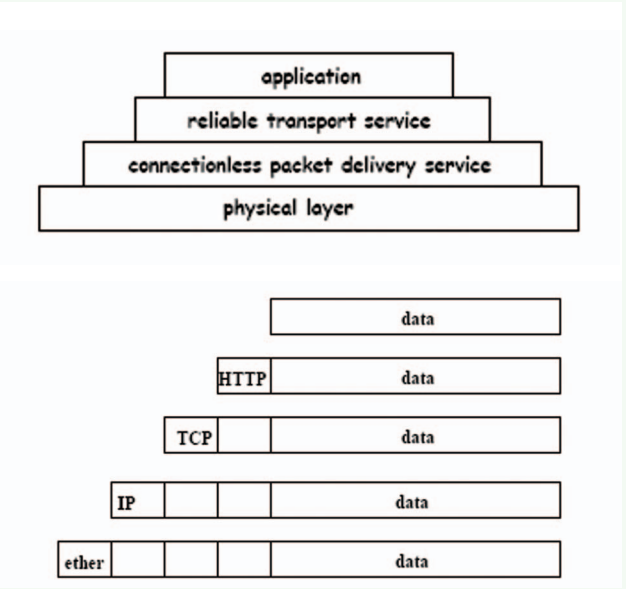


Figure 10. Protocol Hierarchy Diagrams (Source: www.cs.princeton.edu/~bwk)

The basic TCP/IP mechanism is an amazingly robust design; although it was developed in the early 1970s, it has stood up to many orders of magnitude of growth in computers, networks and traffic, with only minor tweaking.

Figure 11. Vint Cerf (left) and Bob Kahn, inventors of TCP/IP. (Source: www.google.nl/intl/nl/press/images/vint_cerf_lg.jpg) and (//isandtcolloq.gsfc.nasa.gov/spring2006/images/kahn.jpg)

The Internet presents some very difficult social, political and legal issues. Privacy and security are hard. Data passes through shared, unregulated, and diverse media and sites scattered over the whole world. It's

hard to control access and to protect information along the way. Many networking technologies use broadcast media, which are vulnerable to eavesdropping. Although attacks on Ethernets are now much reduced, attacks on wireless are on the rise since many wireless networks do not enable encryption.

The Internet was not designed with security in mind, so it is not hard to lie about identity and location; this makes it possible to mount a variety of attacks on the unsuspecting. People are remarkably naive and all too willing to trust a web page that claims to come from their bank or an online merchant, so phishing attacks that attempt to steal identities are more successful than one could believe.

The Internet has no geography and carries bits everywhere almost independent of national boundaries. Some countries have tried to limit Internet access by their citizens by forcing all Internet traffic to pass through a small set of routers that filter content. Others have claimed legal rights over Internet activities that occur largely or entirely outside their boundaries, for instance violation of laws on libel or gambling or pornography.

Of course, the Internet has enabled dissemination of copyrighted material, whether legally or not, at an unimaginable scale, largely through peer to peer networks, and it seems likely that this will continue regardless of attempts by content providers to restrict it with ever more Draconian laws and ever more onerous so-called digital rights management systems.

The Internet has only been in existence since about 1969. The core TCP/IP protocols date from about 1973, and have remained largely the same since then, in the face of exponential growth of size and traffic, a remarkable achievement. We are running low on 32-bit IP addresses, since 32 bits allows for at most 2^{32}, or about 4.3 billion, IP addresses. Mechanisms like network address translation and dynamic host configuration have pushed this off for a while and eventually version 6 of the IP protocol with its 128-bit addresses will eliminate the problem.

Conclusions

Although there are of course many, many technical details, and everything related to computing and communications is evolving rapidly, there are some fundamental notions that will remain central and that should be understood by any educated person, whether of a technical bent or not.

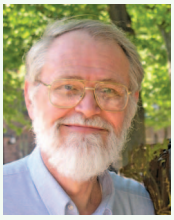
First, information is universally represented in digital form. Second, information is universally processed in digital form. Third, information is stored and transmitted in digital form. Finally, technology has advanced so far that these digital mechanisms are universally available for very little cost. Taken together, these explain the pervasive nature of computers and computing in our world.

All of these are changing rapidly; we are in a time of accelerating change. Change is always disruptive, and we are clearly in for much disruption as far as we can extrapolate current trends. We are totally dependent on digital technology, and there is no way to slow its evolution while we figure out how to handle the problems it presents. Although in almost every way, computing and communications technologies have greatly improved our lives, they will continue to present difficult challenges along with great rewards.

References

- [BGvN] "Preliminary discussion of the logical design of an electronic computing instrument", 1946 [available online at research.microsoft.com/~gbell/computer _structures__ readings_and_examples/00000112.htm].
- [Turing] "On computable numbers with an application to the Entscheidnungsproblem", Proc. London Math Soc. ser. 2, 42 (1936-7), 230-265. [available online at /www.abelard.org/turpap2/tp2-ie.asp]
- [Amdahl] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.
- [Backus] "The FORTRAN Automatic Coding System", J. W. Backus, et al, Proc. Western Joint Computing Conference, Feb 1957, 188-198. [available online at /web.mit.edu/6.035/www/papers/Backus EtAl-FortranAutomaticCodingSystem-1957.pdf]
- [Lohr] *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Scientists and Iconoclasts who were the Hero Programmers of the Software Revolution*, Basic Books, 2001.
- [ISOC] "A brief history of the Internet", Vint Cerf, et al, Internet Society, Dec 2003. [available online at www.isoc.org/internet/history/brief.shtml]
- [Swore] *Charles Babbage and the Quest to Build the First Computer*, Doron Swore, Penguin USA, 2002

About the Author



Brian Kernighan received his BASc from the University of Toronto in 1964 and a Ph.D. in electrical engineering from Princeton in 1969. He was in the Computing Science Research center at Bell Labs until 2000, and is now in the Computer Science Department at Princeton.

He is the author of 8 books and some technical papers, and holds 4 patents. He was elected to the National Academy of Engineering in 2002. His research areas include programming languages, tools and interfaces that make computers easier to use, often for non-specialist users. He is also interested in technology education for non-technical audiences.