What should Computer Science students learn from Mathematics?

Y.C. Tay
`http://www.math.nus.edu.sg/∼mattyc`
Department of Mathematics and Department of Computer Science
National University of Singapore

Abstract

There is an important difference between the title and “What Mathematics should Computer Science students learn?” This talk addresses both questions. To do so, it brings together some examples that illustrate the current state of computer science and information technology.

1 Prologue

The two major professional societies for computer science and information technology are Association for Computing Machinery and IEEE Computer Society. Their answer to “What Mathematics should Computer Science students learn?” is given in a recent document [1]. In making their recommendation, the authors assert that “computer science students must have a certain level of mathematical sophistication”.

I take “mathematical sophistication” (or “mathematical maturity”) to mean “the ability to formalize concepts, work from definitions, think rigorously, reason concisely, and construct a theory”, and this ability is an important part of my answer to “What should Computer Science students learn from Mathematics?”

To explain my view, let me first present a survey of the current state of computer science and information technology, by stringing together some illustrative examples. The survey is very brief, but you may still find it rambling; if you get impatient, you can skip directly to Section 3.

Space constraint prevents me from defining some terminology and providing relevant details but, where possible, I will provide Web addresses so you can quickly find more information. My focus is on university students; hopefully, the points I make are also relevant to pre-university education.

2 Survey

Outside of universities, computer science is often confused with computer programming and computer technology. I hope this section can help to clear the confusion and highlight the scientific aspects of computation.

Let us begin with the most famous problem in computer science: “Is P=NP?”.

2.1 A question on computational complexity — P vs NP

We can think of P and NP as sets of functions that are defined on nonnegative integers and take values 0 or 1. Two such functions are

$$f_{\text{EVEN}}(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd} \end{cases} \qquad f_{\text{GREATER}}(m,n) = \begin{cases} 1 & \text{if } m > n \\ 0 & \text{if } m \leq n \end{cases}$$

Computing f_{EVEN} and f_{GREATER} is easy. However, it is not obvious that this is so for the following two functions:

$$f_{\text{PRIME}}(n) = \begin{cases} 1 & \text{if } n \text{ is prime} \\ 0 & \text{if } n \text{ is not prime} \end{cases} \qquad f_{\text{FACTOR}}(m,n) = \begin{cases} 1 & \text{if } n \text{ has a factor } d \text{ such that } 1 < d < m \\ 0 & \text{otherwise} \end{cases}$$

In a sense, f_{PRIME} and f_{FACTOR} are also "easy" to compute: we can try division by all possible factors. However, such brute-force factorization of a thousand-digit integer would take millions of years with current technology, so factorization is hardly "easy". (Try factoring 114381625757888867669235779976146 612010218296721242362562561842935706935245733897830597123563958705058989075147599290 026879543541, which has 129 digits.) The issue raised by computer scientists is: what is the **computational complexity** of f_{PRIME} and f_{FACTOR}? In other words, how difficult is it to compute them?

One possible definition for "easy to compute" is "membership in P" where, for any $f \in$ P, $f(x)$ can be computed in a number of steps that is a polynomial in the length of x (e.g. the length of 2005 is 4). Thus $f_{\text{EVEN}} \in$P, $f_{\text{GREATER}} \in$P and it was only recently proven, by three computer scientists [2], that $f_{\text{PRIME}} \in$P.

It was long known that f_{FACTOR} is in NP, the class of functions that can be "verified" efficiently; in the case of f_{FACTOR}, this means that if $f_{\text{FACTOR}}(m,n) = 1$, then we can "guess" some d, $1 < d < m$, and verify efficiently that d is a factor of n. (NP stands for "nondeterministic polynomial", and this is a hand-waving explanation of what "nondeterministic" means.)

Although P is a subset of NP, it is not known if P is a proper subset, and the Clay Mathematical Institute lists "Is P=NP?" as one of the seven Millennium Problems [3] (with a prize of US$1million each) — a recognition from mathematicians that this problem from computer science is important.

2.2 Using computational complexity to secure electronic commerce

One way to understand the importance of P vs NP is to consider the impact of their difference on electronic commerce. There is already much buying and selling, as well as information exchange among businesses, over the Internet, a medium that is well known for its insecurity (with viruses, hacking, etc.). In this environment, how can we transmit information securely, verify the authenticity of a message and sign documents electronically?

The obvious way of ensuring secure transmission is to use a *key* (like a password) to encrypt a message before transmission, and for the receiver to decrypt the message using the same, shared key. But how can the receiver get the key from the sender in the first place, especially if the two have not corresponded before?

An ingenious solution to this problem is to use a **public key cryptosystem**, which replaces shared keys with different keys for encryption and decryption. Specifically, every party X has a *secret* decryption key D_X, and publicizes (on a website, say) a corresponding encryption key E_X; anyone — including strangers — can then use the *public* encryption key to securely send messages to X. The pair (D_X, E_X) can also be used for authentication and signatures [4].

2.3 Primality testing in public key generation

In the best-known public key cryptosystem [5], (D_X, E_X) is generated from two primes p_X and q_X. Only X knows p_X and q_X, but their product n_X is part of the public key E_X.

Obviously, different parties X and Y should have $n_X \neq n_Y$. Moreover, for the system to be secure, n_X must be so large that it would take too long to factorize n_X. (Note: the factors are the secret p_X and q_X, from which one can deduce the secret decryption key D_X). We therefore need the primes to be large; unfortunately, larger integers are less likely to be prime.

Since no formula is known to generate primes, we need to be able to quickly generate large integers and verify whether they are in fact primes; i.e. we need an efficient algorithm for computing f_{PRIME}. Such an algorithm was given by Agarwal et al. in their proof that f_{PRIME} $\in$P.

2.4 Breaking public key cryptosystems by factorization

As mentioned, one way of breaking the above public key cryptosystem is to factorize the publicly known n_X. We can factor n_X by repeatedly adjusting m and computing $f_{\mathrm{FACTOR}}(m, n_X)$. It follows that, if f_{FACTOR} can be efficiently computed, then it can be used to efficiently search for a factor of n_X.

We now see the relevance of P vs NP to electronic commerce: Recall that f_{FACTOR} is in NP; if P=NP, then n_X can be factored efficiently, thus breaking the public key cryptosystem.

2.5 Using quantum physics to factorize integers

If P$\neq$NP, we can still break the cryptosystem if we can, somehow, factor n_X efficiently. I mentioned that the brute-force way of guessing a factor d is computationally infeasible, but that is assuming we guess one d after another. If we can guess all possible factors and check them "simultaneously", then we can factor n_X efficiently.

It turns out that such a simultaneous computation is possible with **quantum physics**. In quantum mechanics, an observable quantity is a probabilistic combination of multiple quantum states; computation with such quantities then becomes a simultaneous computation with the constituent states.

Indeed, Shor has shown that such quantum computation can efficiently factorize integers [6]. This has inspired much research into building a computer that can store, add and multiply integers that are combinations of multiple integers.

2.6 The fractal nature of Internet traffic

Besides security, electronic commerce depends crucially on prompt message delivery over the Internet. Customers and businesses will not want to be part of a World Wide Wait.

Much of Internet traffic is carried by networks that are managed by telephone companies who, through decades of engineering experience, know how to model, predict and manage voice traffic. Unfortunately, Internet traffic does not behave like telephone calls.

Specifically, measurements have shown that Internet traffic is bursty, and remains so over several time scales: If we divide an hour of traffic into 60 minutes and plot the traffic volume in each minute, we will see big minute-by-minute oscillations in the traffic volume. If we divide one of those minutes into 60 seconds and plot the traffic volume in each second, we will see second-by-second bursty traffic. If we further divide one of those seconds into 100 ten-millisecond segments, we will again see the jumpy variation over those 100 segments. This **self-similar** behavior is like the fractals on a Mandelbrot set [7] — when we zoom in on part of the curve, we eventually see a repetition of the curve.

In contrast to voice traffic that smooths out over larger time scales, the fractal nature of Internet traffic means that you see burstiness at all time scales (10-msecs, seconds or minutes). With smoothed-out traffic, an engineer knows how much capacity to provide to comfortably accommodate that traffic. With bursty traffic, however, a wide and expensive margin of error must be provided; otherwise, a burst of traffic can cause messages to be lost, which is perceived by users as delays.

Traffic self-similarity thus poses a challenge that has stimulated much research into data traffic modeling and control.

2.7 Playing games with Internet traffic control and pricing

Some research has shown that traffic self-similarity may be partly caused by TCP [8], the principal Internet software that regulates traffic flow. When TCP detects network congestion, it slows down the rate at which a user can send data packets into the Internet. This is one reason the Internet has managed to cope with the exponential growth in Web traffic over the last decade.

Nonetheless, TCP is imperfect, and its interaction with queueing delays, packet losses, etc., can sometimes cause undesirable behavior. Internet engineers therefore continue to tweak it. Also, there is now an inexorable growth in file sharing traffic, generated by users of peer-to-peer systems (Napster, Gnutella, KaZaA, etc.). One download of a movie can take hours, so there is also a temptation for expert users to hack TCP and thus get a larger share of bandwidth for their traffic.

Neither tweaking nor hacking is guaranteed to have the desired effect, since a user's TCP connection could be interacting with thousands of other connections, each exercising some form of control. We can thus view the Internet as a huge game in which players try to capture bandwidth. Indeed, researchers are using game theory to study the appropriate TCP **control mechanism** for ensuring that the Internet remains efficient and stable, despite selfish user behavior.

There is in fact another game going on, with less players but real money. The Internet is divided into smaller networks, each controlled by an ISP (Internet Service Provider). Your traffic to an European destination, say, is passed from ISP to ISP along the way. When an ISP X sends traffic to ISP Y for it to be forwarded to ISP Z, Y will charge for routing the transit traffic. When there is a choice, X will want to pick some Y that minimizes X's cost.

How much should Y charge? Aside from the cost of routing the transit traffic, Y's price also depends on strategy: It may want to charge a high price (and accept less transit traffic), or charge a low price (and attract more traffic, but possibly causing congestion in its own network).

The price game that ISPs play can lead traffic to flow along circuitous routes, thus reducing network efficiency and possibly causing instability. Some computer scientists are now studying **pricing schemes** that balance the ISPs' profit motive and the Internet's performance goals.

2.8 Modeling business data with relations

The examples I have given so far involve difficult mathematics from number theory, quantum mechanics, statistical analysis and game theory. This is not typical; more often, computer scientists build their own theories, from scratch. Consider the data that is the basis for electronic transactions. Here is a toy example:

QS123	Aini	88235235	B747	200	$399
NA55	Shanti	11220000	A033	16	$500
NA11	Shanti	11220000	B747	200	$211
NA11	Aini	88235235	B747	200	$188

Figure 1: **The columns are, from left to right:**
Flight Number, Agent, Telephone, Aircraft, Number of Seats and Airfare.

We can represent this table mathematically as

{ (QS123, Aini, 88235235, B747, 200, $399), (NA55, Shanti, 11220000, A033, 16, $500), (NA11, Shanti, 11220000, B747, 200, $211), (NA11, Aini, 88235235, B747, 200, $188) }

In other words, the table is just a **relation**, with each row represented as a tuple. An enterprise's data can thus be represented as a database of relations. Almost all database systems now use this representation (including those at schools and universities for students and employees).

One reason for this dominance in database management is that the relational abstraction makes it possible to develop a theory for databases [9]. What theory could there be for such a mathematically trivial object? To illustrate, consider the following tables:

QS123	Aini	$399
NA55	Shanti	$500
NA11	Shanti	$211
NA11	Aini	$188

QS123	B747
NA55	A033
NA11	B747

Aini	88235235
Shanti	11220000

B747	200
A033	16

Figure 2: **Are these tables "equivalent" to the one in Figure 1? Which representation is "better"?**

How do we determine whether the relation in Figure 1 is "equivalent" to the four in Figure 2? How do you change the telephone number of an agent for each representation? In each case, how would you go about calculating the revenue that an agent can generate from selling all seats on all the agent's flights? As in the case of computing f_{PRIME} and f_{FACTOR}, the answers are "easy" to see; the issue is how to compute them efficiently. Banks, for instance, routinely handle hundreds of transactions per second.

When computer scientists first considered using the relational model for databases, they found very little theory in mathematics to fall back on. They therefore developed the theory for relational databases from first principle.

2.9 Structuring Web documents with XML

Another case of computer scientists starting from scratch can be found in formal languages and automata theory. One recent example of an application of this theory is in the design of XML [10].

Most businesses now have websites where they provide information for their customers and other businesses. The original language for describing webpages is HTML, but this language only specifies the layout of the page, without specifying the *meaning* of what appears on the page. (For example, without the caption in Figure 1, it would be hard for you to guess what "Aini" or "200" means.)

Increasingly, the information and data that businesses provide on their websites are now processed, not by humans, but by programs. There are programs, for example, that search through websites to compare prices for products (e.g. cars) or services (e.g. airfares). The aim of XML is to structure the Web documents with annotations, so that programs can use those annotations as guides to processing the information.

XML uses a *grammar* to generate “sentences” that are, in this instance, Web documents. The grammar is constrained to make it possible for a program to process the “sentence” unambiguously and efficiently. The theory of grammars comes from **formal languages**, while **automata theory** studies how to process what is generated by grammars.

Although grammars and automata are mathematically defined, their theory was developed by computer scientists. Such a theory is necessary because there is no end to the need for new languages (Fortran, Pascal, Java, XML, etc.).

2.10 Invaluable proofs for hardware verification

The theories for relational databases and formal languages were developed like any mathematical theories, proceeding from definitions to theorems and proofs. For example, one might prove that a particular combination of relational operators suffices to answer a given class of database queries, or that there is no algorithm to decide whether two arbitrarily chosen grammars generate the same set of sentences.

Besides theory construction, proofs also have another role to play. One holy grail in computer science is to have software that can verify (i.e. prove) that a given program has some desired property — that it computes f_{PRIME} correctly, or does not contain certain bugs, etc. Software verification is far from achieving this goal; otherwise, the operating systems on our computers would not crash, or succumb to viruses.

However, there is considerable progress in **hardware verification**. Hardware can have bugs too. Ten years ago, one version of the Pentium processor had a bug that was seemingly obscure, but it cost Intel considerable loss in revenue and damage in reputation [11].

That bug could have been detected and prevented if a verifier had first checked the processor’s implementation against its specification; the software technology for doing so already existed then. Learning from that experience, chip manufacturers nowadays have large teams of engineers who use verification software to check the correctness of their hardware before mass production begins.

Each verification is a proof that the gates and circuits have properties required of them. Such proofs are long and tedious, and not meant for human comprehension. Certainly, they do not contribute to any theory. Unlike the usual proofs in mathematics, however, these are literally worth millions of dollars.

2.11 Impossibility proofs for fault-tolerant distributed consensus

A software or hardware failure at one computer can adversely affect other computers if they are connected into a network. Such failures can, for example, prevent them from achieving some common objective.

Consider a power blackout. Power restoration can cause a sudden surge in demand for electricity, possibly tripping another failure. The stricken generators in the power grid therefore need to power up in a proper sequence. Rather than do this manually, why not have the computers that control the generators communicate with each other (over the Internet, say) and agree on the power up sequence?

We can generalize the question with the following abstract model: Suppose we have a set of geographically distributed computers that are connected to each other by a network; they exchange messages over the network, to arrive at an agreement; they want to reach such an agreement in finite time, even if some messages may be delayed indefinitely and some computers may fail during the message exchange. This agreement problem is called **distributed consensus**.

It has been proven that, unfortunately, it is impossible to guarantee such an agreement [12], i.e. no one, however smart, can design a message exchange that will ensure agreement under all failure scenarios. The proof shows that every computation will have some vulnerable point, such that a single failure at that point will hold up agreement indefinitely.

This theorem points out a fundamental limit on the ability of a distributed computation to tolerate faults. This and other, similar results have stimulated much research into the theory for distributed computing.

2.12 The power of randomization

Fault-tolerance is a standard requirement in engineering, so computer scientists, naturally, look for ways of circumventing the above limitation on distributed consensus. One possibility is randomization, which is a powerful tool that computer scientists often find useful for attacking difficult problems.

In the case of distributed consensus, a computer can be prevented from making progress towards an agreement because it is waiting to get enough messages to decide on the next move. If it is stuck in such a situation, it can get out by randomly picking a move. Thus, the algorithm is no longer deterministic — it is **randomized** [13].

It has been proven that, with randomization, it is possible to reach distributed consensus despite not just one failure, but up to $N/2$ failures; as tradeoff, the running time becomes random, and has no finite bound.

This example demonstrates how randomization can break a limit on computability. Randomization also has the power to increase computational efficiency. For example, despite the recently discovered algorithm for computing f_{PRIME}, the fastest primality test is a randomized algorithm that has the following property [14]: If n is prime, then the algorithm will correctly output $f_{\mathrm{PRIME}}(n) = 1$; but if n is not prime, then the algorithm may incorrectly output $f_{\mathrm{PRIME}}(n) = 1$ too.

The probability that such an error occurs can be made arbitrarily small, smaller than the probability of a hardware error, or the probability that some cosmic particle hits the electronic circuit and changes a crucial value during the computation. In other words, randomized primality testing is, practically speaking, as good as deterministic.

2.13 Pseudo-random number generators and one-way functions

You may be puzzled: Are computers not deterministic? How can they make random moves? In practice, this is done through a generator that produces a "random" number when called upon. An example is

$$x_{i+1} = \mathrm{remainder}((1317x_i + 27697)/131072).$$

Given some initial value (say, $x_0 = 2004$), this calculation produces a sequence $x_1, x_2, \ldots$ that many tests would find is "indistinguishable" from a random sequence. Strictly speaking, however, the sequence is not random, since it is generated deterministically by the equation. (In particular, the sequence eventually repeats itself.) The generator is therefore called a **pseudo-random number generator**.

Instead of sequences of integers, we can consider binary sequences, like

$$101010101010\ldots, \qquad 1011001110001111000\ldots, \qquad 0100011011000001010011100101110111\ldots$$

How can we test whether they are random? For a sequence s to be considered random, the computer scientists' criterion is that all efficient algorithms must fail to distinguish s from a (truly) random sequence.

No one knows whether there exists a pseudo-number generator that satisfies this criterion. In fact, this open question is closely related to whether P=NP.

Recall that no one has an efficient way of factorizing an integer, hence the interest in quantum factorization (Section 2.5). On the other hand, multiplication is easy: given integers c and d, one can quickly compute their product $n = cd$. Multiplication is thus a possible example of a **one-way function**. A one-way function f has two properties: first, for every x, $f(x)$ is easy to compute; second, for most y, it is hard to find an x such that $f(x) = y$.

It has been proven that pseudo-random generators satisfying the above-mentioned indistinguishability criterion exist if and only if one-way functions exist [15]. Also, their existence would imply P$\neq$NP.

2.14 Proofs and relativity

To prove some property (e.g. impossibility of fault-tolerant consensus) in a distributed computation, one could let $s_i(t)$ represent the state of computer i at time t; if there are N computers, then the vector $\mathbf{S}(t) = (s_1(t), s_2(t), \ldots, s_N(t))$ represents the **global state** of the computation at time t. By considering how an action (e.g. receiving a message) changes some $s_i(t)$, one can analyze the evolution of $\mathbf{S}(t)$ and prove its properties.

But what is t? We are accustomed to assuming the existence of some universal time, and using it to identify simultaneous events across space. Thus, we imagine $s_1(t), \ldots, s_N(t)$ to be the simultaneous states of the computers at **universal time** t. But Einstein's relativity has told us that what is "simultaneous" for one observer may not be so for another, that there is no universal time. Clearly, a proof should not be based on an assumption that is known to be false!

It is not nitpicking to suggest that a proof (be it impossibility, correctness, etc.) about distributed computation should take relativity into account. There is currently a project to build an Interplanetary Internet [16], i.e. a network that includes the computers in mission control, launchers, orbiters, landers, etc. The computers in this network are traveling at very different speeds and experiencing very different gravitational pull; the relativistic effects are therefore not negligible.

To illustrate, consider a network closer to Earth: The Global Positioning System (GPS) has 24 satellites in orbit, sending out signals that a receiver can use to locate its terrestrial position. GPS software can now be found in aircrafts (e.g. for navigation), on trucks (e.g. for tracking) and personal digital assistants (e.g. for trekking). The effects of special and general relativity experienced by the satellites are so large that, if not corrected, the increase in positioning error would exceed 10km per day! [17].

One way to factor in relativity would be to give up the Newtonian t and, instead, analyze a global state that is $\mathbf{S}(t_1, \ldots, t_N) = (s_1(t_1), \ldots, s_N(t_N))$, where $t_1, \ldots, t_N$ are local times on the computers' own timelines; the "simultaneity" is then replaced by some message-induced constraint on $t_1, \ldots, t_N$ (other than $t_1 = \cdots = t_N$).

2.15 The universality of Computer Science

One defining characteristic of a mathematical proof is that it is **universal** — $\sqrt{2}$ is irrational anywhere in the universe. Similarly, one expects that fault-tolerant distributed consensus is also impossible in another galaxy.

One of the earliest impossibility proofs about computation was for the **Halting Problem**, which asks: Is there a program to compute the function f_{HALT}, where

$$f_{\text{HALT}}(P, D) = \begin{cases} 1 & \text{if program } P \text{ halts when run on input } D \\ 0 & \text{otherwise} \end{cases}$$

Turing proved that the answer is “No”. This means that, although you can tell that *certain* programs will not halt (e.g. a loop that alternately doubles and halves a number), there is no algorithm that can determine, for *all* P and D, whether P will halt on D [18].

Many people do not consider computer science to be a science. They argue that it is about man-made devices, rather than nature; or, they contend that it is about technology, so computer science is really engineering. However, suppose some aliens from another galaxy were to land on Earth. Since they are able to navigate such vast distances, I would bet that they have some kind of computing device on board. Would their computer contain a Pentium chip? Use Microsoft Windows? Run Java programs? Surely not; but I am confident that it cannot solve the Halting Problem.

Everyone accepts that biology is a science, yet how relevant is our biology to alien life forms? In contrast, the mathematical proofs in computer science point out some truths about computation that are true throughout the universe.

2.16 The Church-Turing Law in Computer Science

But wait, there’s some magic here: How is it possible to conclude that aliens cannot compute f_{HALT}, without knowing how they do their computation? (Indeed, Turing wrote his proof in 1936, before the invention of modern computers.)

The magic is in the Church-Turing Thesis [19], which says that

f is computable (by any means) if and only if it is computable with a Turing machine.

For our purpose, a Turing machine is a 4-tuple (K, Σ, δ, s), where K and Σ are finite sets, δ is a transition function from $K \times \Sigma$ to $K \times (\Sigma \cup \{L, R\})$ — L and R denote “move left” and “move right”— and $s \in K$ [20]. What Turing proved was: f_{HALT} cannot be computed by any Turing machine. It follows from the Church-Turing Thesis that f_{HALT} cannot be computed by any means. But is the thesis true?

The thesis cannot be proven, for to do so requires that we first have a definition of “computable”. Many attempts have been made to formalize the intuitive notion of **computability**, but all generally accepted definitions have turned out to be equivalent to computability with a Turing machine. That led computer scientists to believe that the thesis is true.

Theoretical computer science is full of proofs (membership proofs for P and NP, equivalence proofs, impossibility proofs, computability proofs, etc.), so much so that it resembles mathematics. Yet, in accepting the Church-Turing thesis as true (without a proof!), computer scientists have clearly parted ways with mathematicians.

Physicists and chemists have long accepted various “Laws” (Conservation of Energy, Thermodynamics, etc.) as true, on the basis of accumulated experimental evidence. In similar fashion, computer scientists believe the thesis is true because various alternative models of computation have turned out to be equivalent to the Turing machine.

For this reason, I think the thesis should be called the **Church-Turing Law**, as a reminder that the basis of computability theory is closer to science than to mathematics.

3 What Mathematics should Computer Science students learn?

The computer science community has frequent debates on how much — and what sort of — mathematics their students need [21]. Some argue that traditional mathematics courses (linear algebra, calculus, etc.) should be replaced by discrete mathematics; the latter is not well-defined, but usually includes logic, probability, automata and formal languages, combinatorics and graphs.

The survey above illustrates why these topics are considered more relevant: software and hardware verification (Section 2.10) uses logic, probability is needed for analyzing randomized algorithms (Section 2.12), XML is a formal language (Section 2.9), one example of automata is the Turing machine (Section 2.16), and combinatorics is extensively used in the growing area of bioinformatics. Although I used f_{PRIME} and f_{FACTOR} to introduce P and NP, most of the problems that motivated the study of P vs NP (Section 2.1) concern graphs (e.g. the famous Traveling Salesman Problem [22]).

However, it would be overstating the case to claim that linear algebra and calculus are dispensable. The search engine Google is enormously successful partly because it uses linear algebra [23], and a student cannot learn quantum mechanics (Section 2.5), fractals (Section 2.6), economics (Section 2.7) and relativity (Section 2.14) without knowing calculus. Even partial differential equations are necessary, if a student is interested in computer animation [24] (think *Finding Nemo* [25]). And research that grew from the study of distributed consensus (Section 2.11) has recently led to a realization that algebraic topology is a powerful tool for such studies [26].

4 What should Computer Science students learn from Mathematics?

I am not suggesting that computer science undergraduates should learn all the mathematics mentioned in Section 3. The average computer science student will not want to take so many courses in mathematics, while those preparing for a research career may prefer to narrow their focus. In any case, I claim that what they should learn from mathematics goes beyond their choice of courses.

We teach two things in a mathematics course: **content** and **discipline**. Content refers to the conjunctive normal form, induction principle, Stirling numbers, Central Limit Theorem, Lorentz transformation, Nash equilibrium, etc., that are explicitly taught in these courses. By discipline, I mean **rigor** and **method**. While content varies from one course to another, the discipline remains the same.

4.1 Rigor

When teaching mathematics, we like to use examples and counter-examples to probe the students' understanding of

- definitions (does 2 divide $2\sqrt{2}$?),
- axioms (does every lower-bounded set of rational numbers have a minimum element?),
- theorems (does $d = bx + cy$ imply $\gcd(b, c) = d$?) and
- proofs (does an inductive proof that any finite union of countable sets is countable also prove that their countable union is countable?)

(A colleague once commented that we sometimes overdo it, giving students the impression that mathematics is a minefield littered with pathological examples — like everywhere continuous functions that are nowhere differentiable — leaving them frozen, in fear of making a misstep.)

Even in well-established engineering disciplines, the students can harmlessly forget, when they graduate, most of the content that they learnt. Similarly, for the average computer science student who graduates to become a programmer, I believe that what matters most is not the specific content, but the rigor that is instilled in mathematics courses.

In programming, the analogs of definitions, axioms, theorems and proofs are data structures, assumptions, procedures and algorithms. Algorithms that are careless about the data structures they use (which is one reason for the ubiquitous buffer overflow bugs exploited by viruses [27]) are like proofs that are careless about definitions.

In 1996, the rocket Ariane 5 encountered a software error that caused it to self-destruct; the error arose because Ariane 5 was using a navigation package inherited from the slower Ariane 4, whose programmers assumed that a certain velocity would not exceed a particular value [28]. This (very expensive) mistake is like a student using a theorem without first checking if the assumptions hold.

A student who is confused about the difference between necessary and sufficient conditions is likely to misunderstand an "`if` ... `then`" statement in a program. If students cannot write a one-page proof, how can they digest, analyze and debug the huge programs in current systems, which often have millions of lines of code?

I reckon most computer science students can forget all the definitions, theorems and proofs in their mathematics courses — it is the rigor they learned that will help their careers and last a lifetime.

4.2 Method

Besides content and rigor, a mathematics course also teaches (indirectly, by osmosis) the mathematical method: how to formalize concepts with definitions, work with small examples to gain some intuition, formulate conjectures to capture the intuition, build up to a theorem with useful lemmas, derive interesting special cases as corollaries, explore the limits of the theorem with generalizations and counterexamples, and thus construct a theory. (Recall my definition of "mathematical maturity" in the Prologue.)

For example, a first course in Algebra or Number Theory would cover the Fundamental Theorem of Arithmetic (on the prime factorization of integers). We might motivate the theorem with the factorization of some small integers; introduce the concept of a prime number; isolate the special cases of 0 and 1; extend the notion of "divides" with quotient and remainder; further their understanding of primes by proving there are infinitely many of them; confirm the intuition that if m and n are integers and p is a prime dividing mn, then p divides either m or n; generalize this lemma by introducing the greatest common divisor (gcd); illustrate the Euclidean algorithm for computing gcd; derive the corollary that $\text{gcd}(b, c) = bx + cy$ for some integers x and y; then, finally, inductively prove the Fundamental Theorem with the machinery. We might then examine the uniqueness of the factorization, discuss other "factorizations" (e.g. matrices), and use the theory to guide a study of polynomials.

We go through a similar process when proving, say, an infinite set has uncountably many subsets, or the size of a subgroup divides the size of a group, or the dimension of a vector space equals rank+nullity, etc.

Imbibing the mathematical method is important for the computer science student who wants to do more than just write programs, who aspires to becoming a researcher. There are areas in computer science that requires deep mathematics (in algorithmic analysis [29], cryptography [30], graphics [31], etc.), but recall that large parts of computer science are developed from scratch: relational databases (Section 2.8), formal languages (Section 2.9), software/hardware verification (Section 2.10), distributed computing (Section 2.11), automata theory (Section 2.16).

Unlike mathematics and the natural sciences, which (arguably) aim to discover some unchanging truth, computer science is driven by technology. Its researchers have a constant need to develop new theories for understanding nascent technologies, and lay the groundwork for inventing future ones.

The contribution of Mathematics in this endeavor is to play the role of a tutor, bestowing on the computer science student a method honed through the centuries since Euclid.

5 Epilogue: P vs NP – a question of mathematical modeling?

Among all that a computer science student learns from mathematics, I believe the training that is of particular importance to the researcher is the ability to craft an appropriate model. Consider: The relational model for data is the basis for a billion-dollar industry, the model of time lies at the heart of the impossibility proof for fault-tolerant distributed consensus, and the Turing model has the robustness enshrined in the Church-Turing Law that gives us confidence in plotting the limits of computability.

Modeling is a nontrivial exercise. Cryptographic algorithms may be provably secure, yet indirect attacks can break into the systems that contain them; this calls into question accepted models of cryptographic security [32]. Similarly, Willinger (one of the discoverers of traffic self-similarity) recently criticized current models of fractal traffic and Internet topology [33]. In database management, Codd's formulation of the relational model was motivated by the undesirable properties inherent in the hierarchical and network data models used by the industry then; it is not clear that one can get a comparatively clean data model for XML, given that it appears to be a regression to the structural models of pre-relation days [34].

Mathematics has modeling difficulties too, although these are usually hidden from students, who see only a polished version of mathematics. It took mathematicians some 2000 years, from Archimedes to Cauchy, to model the notion of a limit [35]. Some great mathematicians did not believe in negative numbers, and Descartes dismissed complex numbers as imaginary [36]. The ambitious attempt by Hilbert to model mathematics itself ended in failure [37].

Recently, there was a poll that asked computer scientists whether they believed P=NP [38]. My own view is that the difficulty in resolving the question may lie in the complexity model.

"Is P=NP?" is but one of very many such open questions in computer science: "Is NP=co-NP?", "Is NP=PSPACE?", "Is BPP=PSPACE?", etc. This reminds me of a similar situation in the 1950s, when physicists were struggling to explain the proliferation of subatomic particles. That confusion was resolved through the theory of quarks [39]. P, NP, co-NP, PSPACE, etc. are part of the complexity zoo [40], which now has more animals than the particle zoo [41], with no clarifying theory in sight.

The techniques in complexity theory (reductions, diagonalization, relativization, padding, etc.) are modeled after those in mathematical logic (specifically, recursion theory [42]). Perhaps, we are waiting for someone to formulate a new complexity model and develop a different theory that will, in one fell swoop, resolve many of the questions, including "Do one-way functions exist?" (Section 2.13).

If I could ask the visiting aliens (mentioned in Section 2.15), "Is P=NP?", their answer might be: "No, but that's not the way to formulate the question, dude!"

Acknowledgment

Many thanks to Helmer Aslaksen, Y.K. Leong, Peter Pang, Wei-Lung Wang and Hoeteck Wee for their helpful comments on the draft.

References

[1] `www.acm.org/sigcse/cc2001` (Computing Curricula 2001).

[2] `www.cse.iitk.ac.in/news/primality.html` (PRIMES is in P).

[3] `www.claymath.org/millennium/` (Millennium Problems).

[4] `developer.netscape.com/docs/manuals/security/pkin/contents.htm` (Introduction to Public-Key Cryptography).

[5] `world.std.com/~franl/crypto/rsa-guts.html` (The Mathematical Guts of RSA Encryption).

[6] `www.qubit.org/library/intros/cryptana.html` (Quantum Cryptoanalysis).

[7] `www-unix.oit.umass.edu/~dtillber/mandelbrotzoom1.html` (Mandelbrot Zoom Animation).

[8] `www.itprc.com/tcpipfaq/default.htm` (TCP/IP FAQ).

[9] `www2.cs.uregina.ca/~tang112x/research/papers/2003s/DavidMaier.htm` (The Theory of Relational Databases).

[10] `www.xml.com` (What is XML?).

[11] `kuhttp.cc.ukans.edu/cwis/units/IPPBR/pentium_fdiv/pentgrph.html` (The Pentium FDIV Bug).

[12] `www.teamten.com/lawrence/290.paper/node3.html` (Impossibility of Distributed Consensus with One Faulty Process).

[13] `encyclopedia.thefreedictionary.com/Randomized+algorithm` (Randomized Algorithm).

[14] `www.cryptomathic.com/labs/rabinprimalitytest.html` (Miller-Rabin Primality Test).

[15] `epubs.siam.org/sam-bin/dbq/article/24470` (A Pseudorandom Generator from any One-way Function).

[16] `www.ipnsig.org` (Interplanetary Internet Project).

[17] `www.physicscentral.com/writers/writers-00-2.html` (Einstein's Relativity and Everyday Life).

[18] `en.wikipedia.org/wiki/Halting_problem` (Halting Problem).

[19] `mathworld.wolfram.com/Church-TuringThesis.html` (Church-Turing Thesis).

[20] `en.wikipedia.org/wiki/Turing_machine` (Turing Machine).

[21] `www.computer.org/software/articles/se-math.htm` (How Important is Mathematics to the Software Practitioner?).

[22] `www.tsp.gatech.edu` (Traveling Salesman Problem).

[23] `dbpubs.stanford.edu:8090/pub/2003-20` (The Second Eigenvalue of the Google Matrix).

[24] `www.siggraph.org/s2002/conference/courses/crs10.html` (Level Set and PDE Methods for Computer Graphics).

[25] `www.researchmagazine.uga.edu/spring2004/math.htm` (Wind, Water & —Math?).

[26] `http://research.sun.com/spotlight/2004-05-10.godel.html` (The Topological Structure of Asynchronous Computability).

[27] `www.idinews.com/bufOvfl.html` (Buffers Don't Overfbw on Their Own!).

[28] `www.around.com/ariane.html` (A Bug and a Crash).

[29] `www.cs.huji.ac.il/~doria/markovchains.huji2002.html` (Markov Chains in Theoretical Computer Science).

[30] `world.std.com/~dpj/elliptic.html` (Elliptic Curve Cryptography).

[31] `www.gvu.gatech.edu/people/faculty/greg.turk/math_gr.html` (Mathematics for Computer Graphics).

[32] `www.financialcryptography.com/mt/archives/000147.html` (Turing Lecture by Adi Shamir).

[33] `icnp03.cc.gatech.edu/icnp03-panel-sollins.ppt#1` (Report on NREDS at SIGCOMM 2003: Where is the science in network research?).

[34] `www-106.ibm.com/developerworks/xml/library/x-matters8` (XML Matters: Putting XML in context with hierarchical, relational, and object-oriented models).

[35] `pages.towson.edu/ghan/Teaching/Summer2004/M273/history_of_limits.htm` (History of Limits).

[36] `members.fortunecity.com/kokhuitan/complexno.html` (The Story of Complex Numbers).

[37] `www.faragher.freeserve.co.uk/godeldef2.htm` (Godel's Incompleteness Theorem).

[38] `www.win.tue.nl/~gwoegi/P-versus-NP.htm` (P-versus-NP).

[39] `www.pbs.org/wgbh/aso/databank/entries/bpgell.html` (Murray Gell-Mann).

[40] `www.complexityzoo.com` (Complexity Zoo).

[41] `math.ucr.edu/home/baez/physics/ParticleAndNuclear/particle_zoo.html` (Particle Zoo).

[42] `www.worldhistory.com/wiki/l/list-of-mathematical-logic-topics.htm` (List of Mathematical Logic Topics).