Olympiads in Informatics, 2019, Vol. 13, 157–170
© 2019 *IOI, Vilnius University*
DOI: 10.15388/ioi.2019.10

Programming, Software Development, and Computer Science – The Golden Triangle

Tom VERHOEFF
Department of Mathematics and Computer Science
Eindhoven University of Technology
e-mail: t.verhoeff@tue.nl

Abstract. I present my thoughts on programming, software development, and computer science (CS), and their inevitable relationship. Originally this was intended to help prepare some CS courses aimed (also) at non-CS university students. But it is also relevant for students in secondary education, especially if they have an interest in participating in the International Olympiad in Informatics.

Keywords: Computer science, programming, software engineering, education, competition.

1. Introduction

The reason for writing this note is my involvement in defining some computer science (CS) courses at Eindhoven University of Technology in the Netherlands, especially for non-CS students. Here are some initial questions to set the scene.

- What should everyone know about computer science?
- What should every university student know about computer science?
- What should every engineering student know about computer science?

With “know about” I do not just mean superficial meta-knowledge, where someone knows of the existence of some CS topics and the related jargon without knowing any actual content. That is, the questions could be rephrased as

What CS knowledge and skills should every . . . acquire?

Compare this to similar questions for mathematics, physics, chemistry, biology, etc. The founders of the International Olympiad in Informatics (IOI) must have asked such a question as well. Note that the three questions are related but do not have the same answers.

Related to this question, one should also ask *why* such knowledge and skills are relevant. Subsequent questions are:

- When to teach CS? Earlier or later?
- How to teach CS? Integrated in the student’s primary domain of interest, or more purely? According to what didactic principles?

An important reference document is ACM/IEEE CS Curriculum (2013), which covers most, if not all, of the topics that I touch here. But it weighs in at over 500 pages, and in many cases it only presents alternatives and tradeoffs, without making specific choices or recommendations. We will not answer all questions; neither will we make definite choices or recommendation. But we will delve a bit deeper into these issues. (Short answer: my opinions come close to Kernighan (2017); see below for details.)

2. What and Why

Let's start with the why. Why would someone need to have CS knowledge and skills? Here are some answers.

1. **Because our world has become so much more computational in recent decades**.

   It is important to know about CS simply in order to "survive"; without that knowledge, life will be more difficult. All kinds of decisions that people need to make involve computers, automation, and cyberspace. We can communicate data to any place on earth, store all data that we collect, and process data anywhere we like (also see Verhoeff (2013)). There are many digital dangers nowadays, not in the least due to the rise of artificial intelligence (AI) driven by big data (I recommend du Sautoy (2019) for an interesting exploration).

2. **Because in both professional and personal life, people (scientists and engineers, but also entrepreneurs, and anyone involved with information and its processing) will be required to apply some CS knowledge and skills**.

   Creating a spreadsheet with formulae, developing software tools to help create (non-CS) products, leading a multidisciplinary team to design and develop products that include domain-specific software, running virtual experiments and analyzing the results, formulating computational models, communicating domain knowledge to software developers. Here are some diverse ways in which program code plays an important role:

   - To create products (such as models for 3D printing and pictures using computer graphics).
   - To operate devices (such as cars, drones, and robots).
   - To provide services on the web (such as interactive maps and secure email).
   - To solve computational problems (such as optimal routing of packages and aircraft, weather forecasting).
   - To create software tools that help develop software (such as compilers, interpreters).
   - To analyze massive amounts of data (so as to rank web pages by relevance, and discover new medicines).

3. **Because it is interesting**,
   Just as any other science can be interesting, CS is a very interesting discipline, with important relationships to mathematics, physics, chemistry, biology, psychology, economics, etc.

When addressing the what, it is useful to make the following distinctions. We follow Computing at School Working Group (2012).

Digital Literacy (DL) "the ability to use computer systems confidently and effectively, including:

- Basic keyboard and mouse skills.
- Simple use of 'office applications' such as word processing, presentations and spreadsheets.
- Use of the Internet, including browsing, searching and creating content for the Web, communication and collaboration via e-mail, social networks, collaborative workspace and discussion forums."

Information Technology (IT) "the creative and productive use and application of computer systems, especially in organisations, including considerations of e-safety, privacy, ethics, and intellectual property."

Computer Science (CS) "the study of the foundational principles and practices of computation and computational thinking, and their application in the design and development of computer systems."

We will presume that our students are digitally literate, and that DL is not a goal of our courses (nor of the IOI). Although IT is important, we should not include it as goal of our courses, because IT is focused more on short-term technological issues. The principles that underly IT systems are long lasting, and belong to CS.

Peter Denning provides a broad classification of CS principles in Denning (2003):

Computation "meaning and limits of computation"
Communication "reliable data transmission"
Coordination "cooperation among networked entities"
Recollection "storage and retrieval of information"
Automation "meaning and limits of automation"[1]
Evaluation "performance prediction and capacity planning"
Design "building reliable software systems"

Recently, several books have appeared that put computer science in a broader perspective: Rosenbloom (2013); St. Amant (2012); Tedre (2014). Also the Advanced Placement (AP) Computer Science course is turning to an approach through principles.[2]

Brian Kernighan (2017) (original article Kernighan (2008); first edition Kernighan (2011)) has the subtitle 'What you need to know about computers, the Internet, privacy, and security', and is summarized on Amazon.com as follows.

[1] In Denning and Martell (2015), automation is dropped as a separate category.

[2] `http://apcsprinciples.org`

> “[This book] explains how computer hardware, software, networks, and systems work. Topics include how computers are built and how they compute; what programming is and why it is difficult; how the Internet and the web operate; and how all of these affect our security, privacy, property, and other important social, political, and economic issues. This book also touches on fundamental ideas from computer science and some of the inherent limitations of computers.”

2.1. *Programming*

According to the DL-IT-CS definitions above, *programming* can belong both to IT (“application of computer[ized] systems”) and CS (“development of computer[ized] systems”). The former concerns the more concrete side of programming, whereas the latter focuses more on abstract aspects, such as design.

As argued by Denning in Denning (2004), “The persistent public image of computing as a field of programmers has become a costly myth. Reversing it is possible but not easy.” It is also instructive to consult the online FAQ of Denning (2003), and read the questions and answers about programming.

In Denning’s classification, programming does not appear as a principle; instead, he treats it as a practice. Denning and Martell Denning and Martell (2015) have this to say about programming.

- “What is the paradigm of computing? . . . There were three waves of attempts to unify views. . . . The first . . . argued that computing was unique among all the sciences in its study of information processes. A catchphrase of this wave was that “computing is the study of phenomena surrounding computers.” . . . The second wave focused on programming, the art of designing algorithms that produced useful information processes. . . . A catchphrase of this wave was “computer science equals programming.” In recent times, this view has foundered because the field has expanded well beyond programming and because public understanding of a programmer became so narrow (a coder). . . . The third wave . . . defined computation as the automation of information processes in engineering, science, and business. Its catchphrase was “computing is the automation of information processes.”” Denning and Martell (2015, Ch. 1 (Computing))
- “A *program* is a set of instructions arranged in a pattern that causes the desired function to be calculated. *Programming* is the art of designing a program and providing convincing evidence that the program computes its function correctly. A *computing system* is a combination of program and machine.” Denning and Martell (2015, Ch. 4 (Machines))
- “A *program* is an expression of an algorithm, encoded for execution on a machine. . . . Programming is the practice of encoding algorithms for execution on a machine.” Denning and Martell (2015, Ch. 5 (Programming))

- "[I]t appears to many that algorithm analysis and programming are the heart of computer science. This conclusion does not seem right to us. . . . [I]t appears to us that the architecture of computers is as important as the algorithms they run. This is abundantly evident in the principles of computing. Many principles are about the systems on which computations run. We cannot give a complete picture of computing if we limit our principles to algorithms and ignore the principles of architecture." Denning and Martell (2015, Ch. 12 (Afterword))

By the way, their book does not seem to contain an explicit definition of *algorithm*.

Robert St. Amant has this to say in St. Amant (2012, Ch. 5 (Programming: Putting Plans into Action)):

> "Algorithms and collections of information, organized by abstract data types, need to be translated into a form that a computer can process. This is what programs are for: they translate between the abstract and the concrete.
>
> "*Programming* means expressing abstractions in a language that a computer can deal with. Given what we know about computer architecture . . . I suspect programming may sound a bit daunting. And it can be, . . . "

Kernighan (2017) devotes

- Chapter 4 to algorithms (a dozen pages):

  > ". . . algorithms, which are abstract or idealized descriptions of processes that ignore practicalities. An algorithm is a precise and unambiguous recipe. It's expressed in terms of a fixed set of basic operations whose meanings are completely known and specified; it spells out a sequence of steps using those operations, with all possible situations covered; and it's guaranteed to stop eventually." (First paragraph of Ch.5)

- Chapter 5 to programming and programming languages (twenty pages):

  > ". . . , a *program* is anything but abstract – it's a concrete statement of every step that a real computer must perform to accomplish a task." (second paragraph of Ch.5)

- Chapter 7 to learning to program (in JavaScript, a dozen pages), including loops, conditionals, libraries and interfaces:

  > "I think it's important for a well-informed person to know something about programming, perhaps only that it can be surprisingly difficult to get very simple programs working properly." (First paragraph of Ch.7)

In Barr *et al.* (2010), the issue of 'What everyone needs to know about computation' is discussed by four panelists. One conclusion appears to be that some form of programming (in a language with a well-defined semantics; hence, executable) is necessary, if only to keep people from becoming sloppy in expressing their computational ideas. By the way, this has also been the motivation to include actual programming in the International Olympiad in Informatics (IOI, 2019). The IOI is an algorithmic problem solving contest for high school students, aimed at identifying, encouraging, and challenging students with a talent for CS. The contestants are required to solve *algorithmic problems*, and code their solutions in one of the supported programming languages. These programs are then checked by execution.

Chris Granger argues in Granger (2015) that 'coding is not the new literacy', but that modeling is. Others have countered that modeling must be done in some language, and that in the end this comes pretty close to programming.

2.2. *Software Development*

When programming is done professionally,

- with the goal of developing complex software products,
- often as part of even more complex systems,
- that are maintainable over many years,
- intended for external non-CS customers and users,
- involving multidisciplinary development teams,
- under economic resource constraints,

a whole set of new problems arises. The field of *Software Engineering* (SE) attempts to address these problems. It goes well beyond the basics of programming, including

- domain engineering and requirements engineering;
- modeling;
- architecture;
- evolution, maintenance;
- dealing with errors, quality control, validation & verification, reviewing & testing;
- configuration management, revision control;
- project management;
- specialized software tools for these.

However, anyone doing serious programming, even if only for personal use, can benefit from the key insights of software engineering. In fact, as a teacher you do someone a disservice by not explaining these insights, because without them, programming can a frustrating experience. In particular, the topics of (1) dealing with errors (in a broad sense, including unit testing), (2) configuration management (e.g., using Git), and (3) coding idioms, design patterns, and architecture are essential. It surprises me that the IOI environment still does not offer standard tools for (unit) testing and configuration management, given that the contestants must develop code that works.

3. Challenges

The Asian board game *go* has very simple rules, yet it is a notoriously deep game. Only recently[3] have we succeeded in letting computers play above the mere beginner's level (ComputerGo – Wikipedia, 2019). In chess, the world champion has been beaten by a computer already back in 1996 (DeepBlue – Wikipedia, 2019).

Programming is like go: the basics are very simple, but it is notoriously hard to write good programs. Michael Jackson, the British computer scientist, captures this well in his essay "Brilliance" (Jackson, 1995), which I quote here in full.

> *Some years ago I spent a week giving an in-house program design course at a manufacturing company in the mid-west of the United States. On the Friday afternoon it was all over. The DP Manager, who had arranged the course and was paying for it out of his budget, asked me into his office.*
>
> *'What do you think?' he asked. He was asking me to tell him my impressions of his operation and his staff. 'Pretty good,' I said. 'You've got some good people there.' Program design courses are hard work; I was very tired; and staff evaluation consultancy is charged extra. Anyway, I knew he really wanted to tell me his own thoughts.*
>
> *'What did you think of Fred?' he asked. 'We all think Fred is brilliant.' 'He's very clever,' I said. 'He's not very enthusiastic about methods, but he knows a lot about programming.' 'Yes,' said the DP Manager. He swiveled round in his chair to face a huge flowchart stuck to the wall: about five large sheets of line printer paper, maybe two hundred symbols, hundreds of connecting lines. 'Fred did that. It's the build-up of gross pay for our weekly payroll. No one else except Fred understands it.' His voice dropped to a reverent hush. 'Fred tells me that he's not sure he understands it himself.'*
>
> *'Terrific,' I mumbled respectfully. I got the picture clearly. Fred as Frankenstein, Fred the brilliant creator of the uncontrollable monster flowchart. 'But what about Jane?' I said. 'I thought Jane was very good. She picked up the program design ideas very fast.'*
>
> *'Yes,' said the DP Manager. 'Jane came to us with a great reputation. We thought she was going to be as brilliant as Fred. But she hasn't really proved herself yet. We've given her a few problems that we thought were going to be really tough, but when she finished it turned out they weren't really difficult at all. Most of them turned out pretty simple. She hasn't really proved herself yet – if you see what I mean?'*
>
> *I saw what he meant.*

[3] I wrote this sentence in 2015. In the meantime, *AlphaGo* convincingly beat the world champion go. And not much later *AlphaZero* thrashed *AlphaGo*.

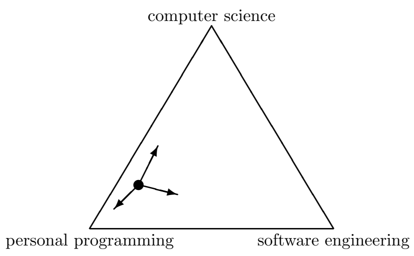

Fig. 1. The force field in which to place a CS/'programming' course.

Managers and directors (educational and industrial), administrators, politicians, they all often still hold similar misunderstandings and misconceptions.

Fig. 1 shows the three forces that need to be balanced in a computer science course on/using programming, both in setting the goals and choosing the means. The same holds for a competition like the IOI. These forces pull towards the three 'pure' topics:

Computer science teach general concepts and insights from computing.

Personal programming teach a programming language for personal use.

Software engineering teach how to develop software beyond personal use.

I call it 'personal programming' here to avoid confusion with 'professional programming' as applied in software engineering.

Note that these three goals are overlapping but quite distinct. A course and contest must be positioned in this force field, especially when no prior knowledge is presumed. Paying more attention to one aspect will detract attention from other aspects. It is true that through (personal) programming, most of Denning's great principles of computing can be visited, provided the trip is carefully planned. It is also the case that many lessons from software engineering are valuable (some would even say indispensable) when doing personal programming.

Should these topics be addressed in some particular order? It seems to make little sense to start on software engineering without a background in computer science and programming. On the other hand, one can start to address software engineering principles and practices early on. For instance, to write program code that is readable and understandable, through proper indentation, spacing, comments, naming, and structuring. One can write comments that document the interfaces of functions and classes. One can think about testing, and write unit tests.

4. Choice of Programming Language and Tools

Programming languages come in many different flavors. A programming language typically supports one or more programming *styles* or *paradigms*: structured, imperative (procedural, object-oriented), declarative (functional, logical), concurrent, parallel.
Properties of programming languages that I consider relevant:

- Degree of formality (of syntax and semantics)
  - Informally described algorithms, such as in Cormen (2013);
  - More formal pseudo code, such as in Cormen *et al.* (2009);
  - Real' (machine executable) languages.
- standardized by international organization (ISO, IEEE, ECMA, ANSI),
- vendor-specific and commercial, versus open-source,
- having up-to-date and supported implementations,
- widely used in industry, versus academic,
- usable by beginners,
- with a large user community,
- for a broad range of applications (including support for GUI, graphics, database connection, internet protocols),
- with good execution performance (speed, memory),
- availability of user-friendly tools (IDE, profiling, testing, documentation generation, integrated version control),
- availability of courses and textbooks.

Here is the list of programming languages that I considered (all are higher level, general purpose, structured). They are split into two groups, and each group is roughly in chronological order. The first group (above the line), I consider serious candidates, and the second group is there mostly because others wanted me to consider them.

(Object) Pascal, Delphi From 1970s, imperative object-oriented (not pure), compiled or interpreted, ISO standard (but that is dated), simple readable syntax, strong static typing, reasonably good compilers and IDE (also open source); used to be the foundation of MacOS; kept alive by Embarcadero (Delphi is vendor specific)
C From 1970s, imperative, ISO standard (current: C18), imperative (not object-oriented), weak static typing, efficient, enforces thinking about low-level optimizations (adorned assembly language), quirky syntax, good (open-source and commercial) compilers and IDEs, favorite for embedded systems (control) not so good for beginners (memory management, robustness); is the basis of the Unix (and nowadays, Linux) operating system

Scheme From 1970s, functional (not pure), IEEE standard (current: 2008?); Lisp dialect; based on lambda calculus

Effel From 1980s, imperative object-oriented, with support for Design-by-Contract, ECMA-ISO standard (current: ECMA-367 from 2006)

Erlang From 1980s, functional and concurrent, aimed at critical high-reliability high-availability systems (hot swapping; think of telephone exchange systems that need to run without downtime)

Java From 1990s, imperative object-oriented (not pure), strong static typing, open standard, commercial owner (Oracle), via interpreter (JVM), C-like syntax, garbage collected, exception mechanism, good compilers and IDEs, favorite for Android mobile platforms, lots of textbooks

Python From 1990s, imperative object-oriented with functional features (not pure), dynamic typing (supports explicit type annotations since v3.5), open source (Python Software Foundation), interpreted, clean syntax[4], extensive libraries, favorite for scripting and coordination

ECMAScript (better known as **JavaScript**) From 1990s, imperative object-oriented with functional features (not pure), interpreted, ECMA standard (current: ES2018); C-like syntax, dynamic typing, good interpreters (e.g., in web browsers), favorite for web programming (client side, nowadays also server side, and for mobile apps); also see Kernighan (2017); Verhoeff (2010)

C++ From 1990s, modernization of C, imperative object-oriented with functional features (not pure), ISO standard (current: C++17), efficient, good (open-source and commercial) compilers, favorite for embedded systems and high-performance computing, not so easy for beginners

Haskell From 1990s, pure functional, lazy, based on category theory

Scala From 2000s, object-oriented and functional (not pure), runs on JVM

Fortran From late 1950s, imperative with object-oriented extensions, ANSI-ISO-IEC standard (current: Fortran 2018), favorite for high-performance computing

Mathematica, Wolfram Language From late 1980s, commercial (Wolfram), focused on applying mathematics

MATLAB From 1980s, commercial (MathWorks), favorite for scientific/engineering modeling and simulation

R From late 1990s, a language and environment for statistical computing and graphics. Open source; based on S from the late 1970s.

Scratch From 2000s, block-structured visual, aimed at children

Go From late 2000s, imperative, C-like, commercial (Google), aimed at server-side networked applications

Swift From 2010s, commercial (Apple), aimed at mobile platform

Dart From 2010s, commercial (Google), ECMA standard, aimed at web, mobile, internet-of-things (IoT)

SageMath open source, alternative for Mathematica

Octave open source, alternative for MATLAB

The choice is not easy, and it is surprising that the CS community has not (yet?) been able to come up with a lingua franca (compare this to mathematics, where the situation is considerably better). This is also known in the IOI community. There are many trade-offs. If one wants to take popularity into account, then also consult TIOBE Index (2019).

[4] My main gripe is that the symmetric = is used for assignment, a bad heritage from C.

Keep in mind that students (in engineering) should learn (about) multiple programming languages and paradigms.

Nowadays, I lean more towards a functional language. This is well explained by Simon Peyton Jones in Heath (2017): "If you want to see which features will be in mainstream programming languages tomorrow, then take a look at functional programming languages today." The tendency is towards side-effectfree functions and immutable data, because these are so much easier to reason about and parallelize.

Since Python introduced type annotations that are supported by some of the tools (notably PyCharm[5]), it has moved up considerably in my list. When I don't need the highest performance and don't need 3D graphical output, Python is my preferred language. Together with the Jupyter notebook technology[6], it provides a very productive interactive experience. For high performance programs, I prefer to generate C or C++ code from higher-level models through some higher-level domain-specific language (DSL).

5. Role of Didactics and Problem Domain

It is very important to motivate students, so that they take a deeper interest in computer science and programming, and will not consider it a triviality. Therefore, the application domain must be attractive. This is not easy in a course or competition with a diverse audience.

When teaching a course on programming, it is important to be aware of didactic issues. For instance, the looping construct in most programming languages is syntactically not so difficult, and even semantically it may be simple. But loops serve many purposes, and each requires further insights. Hence, it is important to teach the design of loops in a systematic way. Similarly, recursion is almost trivial from the language perspective, but didactically it needs a careful approach (Verhoeff, 2018).

Another essential topic in programming is that of abstraction (Verhoeff, 2011). Once the primitive building blocks of a programming language have been treated, it turns out that in real programs it is important to capture all kinds of abstractions (both on the level of data and actions). Defining and designing such abstractions is at the heart of computer science and programming.

Good study material should include:

- A textbook (preferably in interactive digital form, that can be searched).
- Exercises, with feedback; possibly on-line and automated, such as Codingbat (2019); Datacamp (2019).
- Web lectures (cf. Khan Academy, EdX, Lynda, etc.).

Certain choices (language, tools, problem domain, and didactic approach) will require an investment that makes it harder to change these choices later on. For instance, development of exercises and assignments is costly, but their details will critically depend on these choices.

[5] `https://www.jetbrains.com/pycharm/`

[6] `https://jupyter.org/` supporting over 40 programming languages

6. Conclusion

It is legitimate to expect that non-CS students will need some programming skills. It is also legitimate to expect that they need at least some software engineering skills. But they also need more fundamental insights in computing concepts that transcend programming and software development.

Whether this can be combined in a single course (a single competition) is debatable, and would certainly pose an extreme challenge. When covering this material in multiple courses, these courses must be carefully coordinated.

We should avoid pretending that a 'personal programming' course will make you a computer scientist, or a software engineer. Learning a programming language is relatively easy, writing good software is hard.

We identified three forces (see Fig. 1), and observed the following dependencies.

- Even if a course is intended solely as an introduction to CS, it is a good idea to include some programming involving an executable programming language.
  Motivation: Using informal notation or pseudo code to express algorithms leaves the door open for sloppiness and misunderstanding.
  Note: In this case, only a minimum of software engineering issues need to be addressed.
- If programming needs to be applied in practice, no matter on what scale, then computer science and software engineering knowledge is needed.
  Motivation: Without CS and SE knowledge, it is too easy to create low-quality software. This is frustrating, time consuming, and costly.
  Note: The scale of application will determine the amount of CS and SE to include.
- To understand the lessons of software engineering, it is necessary to have some programming experience.

The appendices list, what I consider, the top-10 most relevant topics from each of the three corners. Compare this to IOI Syllabus (2019).

References

ACM/IEEE-CS Joint Task Force on Computing Curricula (2013). *Computer Science Curricula 2013*. ACM Press and IEEE Computer Society Press. DOI: `dx.doi.org/10.1145/2534860`

Barr, J. *et al.* (2010). What everyone needs to know about computation. *SIGCSE'10*, pp.127–128, 10–13.

Codingbat, `codingbat.com`

Cormen, Th.H. (2013). *Algorithms Unlocked*. MIT Press.

Cormen, Th.H. *et al.* (2009). *Introduction to Algorithms* (3rd Ed.). MIT Press.

Computing at School Working Group (2012). *A Curriculum Framework for Computer Science and Information Technology*. `https://www.computingatschool.org.uk`

Datacamp, `https://datacamp.com`

Denning, P.J. (2003). Great principles of computing. *CACM*, 46(11),15–20. Also see: `http://greatprinciples.org` with online FAQ

Denning, P.J. (2004). The field of programmers myth, *CACM*, 47(7),15–20.

Denning, P.J., C.H. Martell (2015). *Great Principles of Computing*. MIT Press.

Granger, C. (2015). *Coding Is not the new literacy*. Blog post, 26 Jan. 2015. `https://www.chris-granger.com/2015/01/26/coding-is-not-the-new-literacy/`

Heath, N. (2017). What's the future of programming? The answer lies in functional languages (an interview with Simon Peyton Jones). *TechRepublic*, 23 Oct 2017.
`https://www.techrepublic.com/article/whats-the-future-of-programming-the-answer-lies-in-functional-languages/`

International Olympiad in Informatics. `ioinformatics.org`

IOI Syllabus `https://ioinformatics.org/page/syllabus/12`

Jackson, M. (1995). *Software Specifications and Requirements: A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley.

Kernighan, B.W. (2008). What should an educated person know about computers? *IEEE Solid-State Circuits Society Newsletter*, 13(2), 5–11. `https://doi.org/10.1109/N-SSC.2008.4785733`

Kernighan, B.W. (2011). *D is for Digital: What a Well-informed Person Should Know about Computers and Communications*. CreateSpace Independent Publishing Platform.

Kernighan, B.W. (2017). *Understanding the Digital World: What you Need to Know about Computers, the Internet, Privacy, and Security*. Princeton Univ. Press.

Rosenbloom, P. (2013). *On Computing: The Fourth Great Scientific Domain*. The MIT Press.

du Sautoy, M. (2019). *The Creativity Code: How AI is Learning to Write, Paint, and Think*. Fourth Estate.

St. Amant, R. (2012). *Computing for Ordinary Mortals*. Oxford University Press.

Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*. Chapman and Hall/CRC.

TIOBE (2019). *Programming Community Index*. `https://www.tiobe.com/tiobe-index/`

Verhoeff, T. (2010). An Enticing Environment for Programming. *Olympiads in Informatics* 4:134–141.

Verhoeff, T. (2011). On Abstraction and Informatics, presented at *ISSEP 2011*, Bratislava, Slovakia.

Verhoeff, T. (2013). Informatics everywhere: Information and computation in society, science, and technology, *Olympiads in Informatics*.
`https://www.win.tue.nl/~wstomv/publications/issep-2011-on-abstraction.pdf`
errata and addenda `https://www.win.tue.nl/~wstomv/publications/abstraction-extra.pdf`

Verhoeff, T. (2018). A master class on recursion. In: *Adventures Between Lower Bounds and Higher Altitudes* (Lecture Notes in Computer Science Vol.11011). Springer. 610–633.
DOI: `https://doi.org/10.1007/978-3-319-98355-4_35`

Wikipedia Contributors (2019). Computer Go. *Wikipedia, The Free Encyclopedia*.
`en.wikipedia.org/wiki/Computer_Go`
Also see: `www.youtube.com/watch?v=OnBpkpOFAug`

Wikipedia Contributors (2019). Deep Blue, *Wikipedia, The Free Encyclopedia*.
`en.wikipedia.org/wiki/Deep_Blue_(chess_computer)`

A Computer Science Topics

1. Computational problems, decision problems, reduction
2. Automata (finite state, Turing machine)
3. Decidability, computability
4. Machine architecture, memory hierarchy, communication networks
5. Order analysis of algorithms, complexity (runtime, memory usage)
6. Tractability, P vs NP
7. Data encoding, compression, error detection and correction, information security
8. Reasoning about computations, formal methods
9. Data organization, databases
10. Numerical computations, floating-point arithmetic

B Programming Topics

1. Programming language, machine, operating system, interpreter, compiler
2. Imperative, object-oriented, functional, and (constraint) logic programming
3. Values, literals, types, expressions, named constants, variables, assignment

4. Input and output, formatting
5. Control ow, sequencing, conditional execution, iteration, loop invariants
6. Goal-directed algorithmic problem solving, coding idiom
7. Procedural abstraction, functions, parameters, local versus global variables, side effects
8. Modularization, data abstraction, abstract data types, recursion
9. Reuse, standard libraries, standard algorithms, idiom, design patterns
10. Event handling, (graphical) user interface, concurrency

C Software Engineering Topics

1. Coding standards, writing understandable code
2. Documentation, two-party contracts with assumptions and obligations
3. Requirements engineering, quality criteria, modeling
4. Dealing with errors, robustness, exceptions, fault tolerance
5. Decomposition (functional, data), refactoring
6. Architecture, (de)coupling, cohesion
7. Reviewing, including code review
8. Testing, unit testing, integration testing
9. Revision control, configuration management, issue tracking
10. Continuous integration, metrics, code generators, evolution, maintenance

Tom Verhoeff is Assistant Professor in Computer Science at Eindhoven University of Technology, where he works in the group Software Engineering & Technology. His research interests are support tools for verified software development and model driven engineering. He received the IOI Distinguished Service Award at IOI 2007 in Zagreb, Croatia, in particular for his role in setting up and maintaining a web archive of IOI-related material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI Scientific Committee from 1999 until 2007.