Olympiads in Informatics, 2018, Vol. 12, 3–11

DOI: 10.15388/ioi.2018.01

Computer Science in K-12 Education: The Big Picture

Tim BELL
University of Canterbury, Christchurch, New Zealand
e-mail: tim.bell@canterbury.ac.nz

Abstract. As topics from computer science are increasingly being taught in K-12 schools, it is valuable for those teaching within new curricula to be aware of the purpose of the various components that students are expected to learn. We explore the main purposes of having computer science in curricula in the first place, and then use examples to show how particular topics that might be regarded by some as esoteric can be related to the bigger picture of what is trying to be achieved. The model used is to relate curriculum content to how it affects people, both those who are learning the subject, and those who will be using digital technologies developed by those who have just learned to develop them. This provides a framework to help teachers to motivate themselves, their students, and other stakeholders to engage with new curriculum content.

Keywords: CS education, curriculum, teaching programming.

1. Introduction

Until recently it has been rare for K-12 students to have topics relating to computer science as part of a formal national or state curriculum. However, recently many countries have been introducing topics such as programming and computational thinking (Hubwieser, Giannakos, Berges, *et al.*, 2015; Duncan and Bell, 2015; Heintz *et al.*, 2016). There are several motivations for this, which can loosely be divided into growing students' interest to increase the uptake of the subject by students, and building up knowledge and skills to support students' careers.

As can happen with any curriculum, there is a risk that students see multiple topics spread over a number of years as a disjoint set of independent academic ideas and practical skills that don't necessarily have an obvious purpose. Teachers themselves also need to recognise why a topic is important for their students to understand, and others in the community (including parents and school officials) will also need help to form views on the relevance and importance of new curriculum content.

In this paper we will explore the bigger picture of what curricula should be trying to achieve, and the purpose of including various components in typical curricula, so that all of those involved in it can keep a vision of the broader purpose.

2. The Purpose of K-12 Curricula

As topics relating to computer science enter international curricula, the public perception of the content is often simplified to the term "coding", or sometimes referred to as "computational thinking" to show that there is more depth to the subject. In reality, most countries are adopting broader curricula that cover issues such as the performance of algorithms, data representation and data structures, software engineering, networks, security, and more (Hubwieser, Giannakos, Berges, *et al.* 2015; Duncan and Bell, 2015; Heintz *et al.*, 2016).

Even if only focussing on programming, the goal should be considerably more than learning to code in some particular language(s). A key goal should be to help students find out if it is something that they might be passionate about, and to give them a vision of what they might do with the wide possibilities that skill in computer science and programming can open up. This is particularly important because stereotypes of who might be good at "coding" can deter those who would enjoy it from even trying (Overdorf and Lang, 2011). Helping students to explore the subject means that it is important that it is presented in a way that they can see the purpose, and appreciate that programming is a skill that supports problem solving and creativity, and can be applied to many areas.

Making programming accessible to those who might not think they are interested involves a delicate balance between supporting students to have early success, while being careful to help them see that it is a skill that will take many of hours of experience to build competence. This relates to Papert's "low floor" and "high ceiling" (Resnik, 2009). There are wide ranging views on how long it takes to learn programming, and in fact, there are many definitions of what it means to be a competent programmer; in a commercial environment it might mean knowing how to solve clients' problems using a particular language or software stack, while in a research context or programming competition it could involve having a wide range of skills around implementing sophisticated and novel algorithms. Students who are new to programming need to maintain confidence as they learn, but both under- and over-confidence can lead to them being disheartened and giving up. Peter Norvig (director of research at Google) wrote an article titled "Teach Yourself Programming in Ten Years",[1] which tries to provide a balance to books that give the impression that programming can be learned very quickly, with titles such as "Teach Yourself Java in 24 Hours". Of course, the message isn't that one should plan to spend 10 years learning before becoming a software developer, but that, like any skill, there is always more to learn, and mastery can involve a lot of experience and learning – the more we learn, the more we realise that we don't know! Having the

[1] http://norvig.com/21-days.html

topic in the school curriculum gives students more opportunity to develop skills gradually over time, and also to find out the accompanying skills that are needed to support their programming (such as math for analysing algorithms, communication skills for working with other programmers, and logic for reasoning about program correctness).

The success of helping students discover their passion for computer science can depend on the way it is taught, as can happen with any other subject. This means that having well-prepared teachers with a good understanding of the purpose of the curriculum is key to its success.

For those who have already discovered their passion, there is a need to support them through clubs and competitions that allow them to extend their skill beyond the basics covered in school curricula. In the past, such organisations have provided a key opportunity for students passionate about computing to build relationships with like-minded friends and experience success in a context where they are understood. Such organisations are likely to become *more* important as the subject becomes formalised in K-12, since there will always be those who may become bored with the basic curriculum that is designed for typical students, and need opportunities to stretch themselves. Giving students a vision of what they can do is fundamental to education, and clubs and competitions give students a chance to expand their vision by providing the opportunity to interact with a range of people, including industry professionals, experienced academics, and club/competition alumni.

As well as helping students discover their passion, having computer science as a subject in schools also gives students more time to develop their skills over a period of several years. As with other subjects, knowledge in this area doesn't necessarily have the goal of preparing students for a career in computing, as a general understanding of how digital systems work and what programming is can help students to function in an informed way in an increasingly digital society.

For example, many employees work with spreadsheets as part of their day to day work, yet it is well established that may of the spreadsheets used in practice contain substantive errors (Powell, 2008). This is not surprising when spreadsheet design is seen as having a significant overlap with the skills required to do programming, including proper testing, use of logic in selection (if) commands, and understanding the difference between constants and variables including using effective naming.

Broader topics such as Artificial Intelligence give students a chance to understand a little of the mechanisms behind these apparently mysterious technologies, particularly in the light of the ethical and moral decisions that society is increasingly likely to have to make as the capabilities of such systems grow.

In addition to benefiting students by helping them find out if computing is an area that they have a passion for, new curricula are also intended to benefit broader society, including industry and national interests, by addressing talent shortages and encouraging creativity and entrepreneurship, as well as increasing national digital capabilities that are increasingly needed to maintain security and financial stability for a country.

Whether the motivation is to benefit students or society, the reason for offering a computing curriculum is ultimately to benefit people, and ideally society is better off for having increased capability in this area. Of course, digital systems don't always

have a positive benefit, and students should also be made aware of the ethics surrounding the development of new technologies so that they can contribute positively to society.

3. Seeing How Curriculum Elements Fit Together

We now consider how to see the bigger picture of the curriculum when classes are necessarily focussed on specific topics. A range of curricula are being developed around the world in the area of computing; some refer to the topics as computational thinking, digital technologies, and computer science. All such curricula that we are aware of include programming as a key subject, and most include a range of related topics. For example, an analysis of curricula by Duncan and Bell (2015) classified other common components as:

- Algorithms (designing programs and/or understanding computational complexity).
- Data representation (representation of various data types, and encoding through encryption, compression and error corrections).
- Devices and infrastructure (including device architecture, networks and cloud computing).
- Digital applications (such as simulation, modelling, and creating digital content).
- Humans and computers (including cybersafety, ethics, careers, and human-computer interaction).

A related project has identified ten "big ideas" in computer science that were collated by seeking input from a range of coputer science education researchers and professionals from around the world (Bell *et al.*, 2018). The central big idea identified was that "digital systems are designed by humans to serve human needs"; other ideas covered topics such as data representation, algorithms, complexity, computability, virtual representations, time dependent operations, and communication protocols. These aren't intended to be exhaustive list of what there is to know, but were considered by professionals to be key ideas that students would benefit from being aware of.

With such a diverse range of topics, it's easy to miss seeing the forest for the trees – learning can become a lot of small topics that can loose connection to the big picture. In the following sections we consider three topics, and relate them to the bigger picture to illustrate these important connections. We build on the central ideas of the discipline being human-centric to identify the value or purpose of the topics.

4. Example: Programming

Computer programming is often characterised as "coding", but it is a very broad range of skills, and developing software can be broken into elements such as analysis, design, coding, testing, and debugging. Each of these is a discipline in itself; analysis of a situ-

ation for which software is to be developed requires very good communication with the people requiring the program; design involves creativity; testing requires rigour; and debugging requires persistence. Characterising programming as simply "coding" can hide these important human-centric qualities.

Likewise, seeing computer science education as teaching a particular language also risks missing the point. Instead of, say, "teaching Scratch[2]," we should think of it as "teaching programming, using Scratch"; or instead of "teaching C++", we might be "teaching OO programming using C++". This helps to focus on the broader skills such as testing and debugging, rather than seeing a programming language as a piece of software with some set of features to be learned.

Another important attitude to develop is that we don't write computer programs for computers; we write them for people! This puts the focus on getting the interface right; this isn't just about GUI and mobile interfaces, but even for a simple text or dialogue box interface it could involve making sure that it is fast enough to have a good response time for a user – 0.1 seconds is ideal as it will appear instantaneous, otherwise 1 second is a good target to keep interactions at a conversational pace (Johnson, 2013). The interface design could also take into consideration how input and output are presented, such as having simple messages to the user that are unambiguous.

In addition to writing programs for the end user, we also write them for the next programmer who may need to use or modify it (and the next programmer might be the original author in a year's time, or even the next day). For example, Fig. 1 shows a short Python function; the reader is encouraged to work out what it does before reading on.

Fig. 2 shows the same function with a meaningful name and comment (which makes the purpose of the function explicit), as well as a variable name that has a role in explaining the algorithm. In fact, the name "highest_so_far" could be the basis of an invariant for a proof of correctness for the algorithm, and helps the reader to see what the intended invariant is. Thus by using carefully chosen language in variable and function names, the programmer can help to organise their own thoughts as well as capture the logical intent of the program. Using clearer language also makes it easier to see a slight inefficiency – this implementation makes n key comparisons when only $n - 1$ are required.

Counter to common stereotypes, being an effective programmer requires good social and communication skills, whether the relationship is with the user, or with other

```
def hs(s):
    xxy = s[0]
    for xxx in s:
        if xxx > xxy:
            xxy = xxx
    return xxy
```

Fig. 1. A Python function.

[2] Scratch is a popular language for teaching programming to beginners.

```
def highest_score(scores):
    """takes a list of scores and returns the highest one"""
    highest_so_far = scores[0]
    for score in scores:
        if score > highest_so_far:
            highest_so_far = score
    return highest_so_far
```

Fig. 2. The Python function of Fig. 1 using better style.

programmers. Blackwell points out that "many skills of a professional programmer are related to social context rather than the technical one" (Blackwell, 2002). The point of learning programming is to develop something to help people, and the rules and conventions that we teach (such as commenting code) are there to support the people who may need to work with the program in the future.

5. Example: Formal Languages

Formal languages appear in some school curricula, and can sometimes be seen as a theoretical topic that is difficult to understand and doesn't have practical relevance. Here we consider one of the basic ideas of formal languages that often appear in introductory material: regular languages, which can be characterised by regular expressions and parsed by finite state automata (FSA).

Understanding regular languages through a FSA has been introduced at primary school level through a CS Unplugged activity called "Treasure Hunt",[3] where students create a map of ship routes between islands to try to find their way to Treasure Island. The CS Unplugged activity involves several students, and running around a playground; a similar activity that can be used by individuals is provided in the Computer Science Field Guide,[4] where a student explores an interactive map of commuter train stations.

Follow-up activities enable students to explore conventional FSAs and the regular languages that they represent. But the bigger picture can be found by thinking about how these affect people. We find regular languages being used for checking user input (such as the format of an email address or URL), and most programming languages offer support for regular expressions. They are also fundamental to implementing programming languages by providing lexical analysis based on a human view of the rules (through a regular expression) rather than a computer-centric version (which is the equivalent FSA).

Regular languages are also a gateway to the Chomsky hierarchy, which in turn addresses philosophical questions of what computing is and what can and cannot be computed (the Turing Machine is based on a FSA, and is regarded as a useful abstract model

[3] https://classic.csunplugged.org/finite-state-automata/

[4] http://csfieldguide.org.nz/en/chapters/formal-languages.html#finite-state-automata

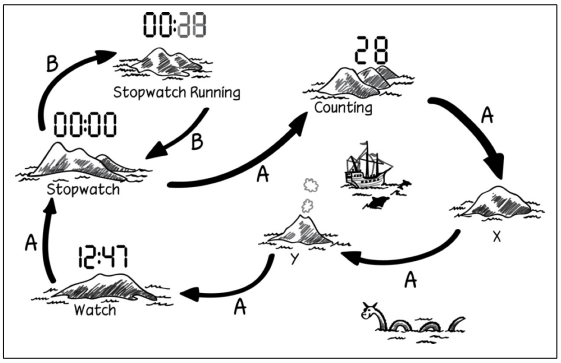

Fig. 3. A "treasure map" (FSA) that represents a digital watch interface.

of computation). This raises very human questions about what these devices are capable of and how the power and limitations of computation may affect us in the future.

An FSA can also be used to model interfaces; for example, Fig. 3 shows a whimsical "map" of the interface for a simple digital watch with two buttons (A and B) that change the state of the watch. This helps us to see an interface as a machine, rather than just a static layout of GUI elements, and can also identify unsafe transitions in more critical interfaces such as medical devices (Thimbleby, 2009). The impact on humans here ranges from mild frustration with a poor interface, to life-and-death situations.

6. Example: Error Detection and Correction

The final example we give is exploring algorithms for error detection and correction. An initial experience with error detection can be given in the classroom through the CS Unplugged Magic Parity Trick.[5] This involves a grid of 6 by 6 or more cards that can be flipped to show black on one side and white on the other (each card represents one bit). The "magic" trick uses parity forward error correction to determine which card has been flipped while the presenter was looking away. A related "trick" is to work out the check digit at the end of a product code; the presenter can pretend to "mind read" what it is, but of course it can be calculated from the other digits[6].

Error correction can be demonstrated by "damaging" a QR code. For example, the QR code in Fig. 4 contains the text of the first official Morse code message sent in 1844. To demonstrate error correction at work, the demonstrator can change some of the white "bits" to black with a pen. The second QR code in Fig. 4 has had quite a few of the bits changed, and yet can still be scanned correctly. The third one is probably too damaged to scan; finding the amount of "damage" that can be done while still having the code

[5] https://classic.csunplugged.org/error-detection/

[6] https://csunplugged.org/en/topics/error-detection-and-correction/unit-plan/product-code-check-digits/

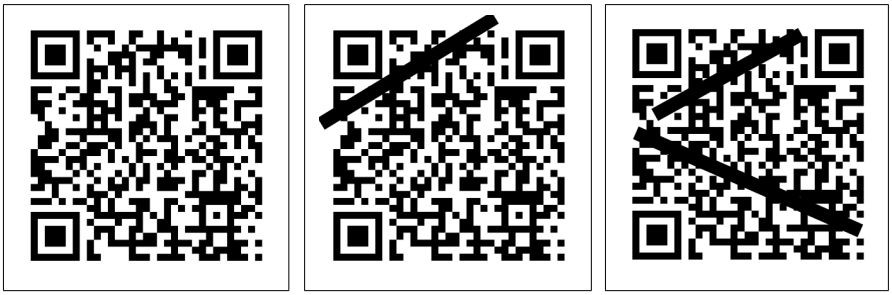

Fig. 4. A QR code with various amounts of "damage".

readable is an interesting exercise for students. The important point is that changing the binary representation doesn't change the message; without error correction, changing just one bit would change a character in the message.

The bigger picture of error control is again determined by thinking about how it affects people. Binary digits are stored and transmitted in very vulnerable situations, and small physical issues can change single bits easily. If one bit in a file was changed without the user knowing, then the data would be incorrect; a one-bit error could have serious consequences for the people concerned, whether it is a financial amount, exam result or medical data. We take it for granted that data will remain intact, or at worst, the computer will refuse to read a whole file even if only part of it has an error. Another important aspect is that the cost of error correction is considerably less than making full backups; if error correction techniques weren't used, multiple backups would be needed to have any certainty around the accuracy of data, and we would be constantly comparing copies of the data to check its integrity. The human costs here are the financial cost of extra storage and the delays needed for checking, or if errors are ignored, the impact on the user would be that they are working with incorrect data.

7. Conclusion

A central idea of the "big picture" of computer science is that it is about *people*: "digital systems are designed by humans to serve human needs" (Bell *et al.*, 2018). Programs are written for people, both the users of the program, and the next person who must maintain it. While there are some circumstances where an individual might be solely responsible for an entire program, such as programming exercises and challenges, or encapsulated components of a larger system, the reality is that programming generally leads to an outcome that will affect people. The same is true of all other topics in computer science, whether it is finding an algorithm with better time complexity (so that people can have their problems solved faster and with less demand on computing resources), or implementing checking so that data collection, storage and transmission happen accurately. And at an even broader level, the reason that we teach these subjects is to help students develop a vision for their own future, and to empower them to help others using skills

acquired that enable them to work effectively with digital systems, whether evaluating them critically or creating new systems.

In a subject where the focus is inevitably digital devices, it is good to stand back regularly and see the big picture. For any topic being taught, we want to be able to answer the question "How will someone potentionally be better off as a result of knowing this?" with something other than just getting good exam grades or a qualification; the topics in curricula have been chosen because they are tools to benefit people, so the challenge is to be aware of what those benefits are.

References

Bell, T., Tymann, P., Yehudai, A. (2018). The Big Ideas in Computer Science for K-12 Curricula. *Bulletin of EATCS*, 1(124). Retrieved from http://bulletin.eatcs.org/index.php/beatcs/article/view/521

Blackwell, A. (2002). What is programming? In: *14th workshop of the Psychology of Programming Interest Group*. 204–218.

Duncan, C., Bell, T. (2015). A Pilot Computer Science and Programming Course for Primary School students. In: *WIPSCE,* 39–48. http://doi.org/10.1145/2818314.2818328

Heintz, F., Mannila, L., Färnqvist, T. (2016). A review of models for introducing computational thinking, computer science and computing in K-12 education. In: *Proceedings Frontiers in Education Conference (FIE)*. 1–9. http://doi.org/10.1109/FIE.2016.7757410

Johnson, J. (2013). *Designing with the mind in mind: simple guide to understanding user interface design guidelines*. Elsevier.

Overdorf, R., Lang, M. (2011). Reaching out to aid in retention. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education – SIGCSE '11*. 583. http://doi.org/10.1145/1953163.1953325

Powell, S. G., Baker, K. R., Lawson, B. (2008). A critical review of the literature on spreadsheet errors. *Decision Support Systems*, 46, 128–138. http://doi.org/10.1016/j.dss.2008.06.001

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., *et al.* (2009). Scratch: programming for all. *Com. of the ACM*, 52(11), 60–67.

Thimbleby, H. (2009). Contributing to safety and due diligence in safety-critical interactive systems development by generating and analyzing finite state models. In: *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. 221–230.

T. Bell is a professor in the Department of Computer Science and Software Engineering at the University of Canterbury, where he leads the Computer Science Education Research Group. His "Computer Science Unplugged" project is being widely used internationally with the supporting materials (books and videos) having been translated into over 20 languages. Tim has received awards for his work in computing education including the 2018 ACM SIGCSE Outstanding Contribution to Computer Science Education award. He has been actively involved in the design and deployment of the new digital technologies curriculum in New Zealand schools.