PGCE Secondary Computing Assignment 1
2019-20
Mark Gadsby

What is computational thinking? How should it be taught and assessed in secondary schools?

Introduction

The case for computer science being taught in schools is strong. The world has experienced increasing digitalisation over recent decades with profound implications for the world of work. Jobs unimaginable a few years ago are now common place and this trend is set to continue with as many as 90% of newly created jobs requiring some level of digital literacy (Blackwood, 2016). Across the globe, it is being realised that equipping young people with computer science skills is as vital as equipping them with those in literacy and numeracy. Here in the UK, computing is a foundation subject in the national curriculum. The Department of Education has stated that 'A high-quality computing education equips pupils to use computational thinking and creativity to understand and change the world' (DfE, 2013). The term computational thinking is not defined within the national curriculum.

Computational Thinking

According to Grady Booch (2016) we are living through a computational revolution which represents a paradigm shift as profound as the shift to scientific thinking half a millennium ago. Thinking scientifically then helped us to understand our world and now computational thinking is our conduit to controlling it. He points to sub-second financial trades, smart phone apps for ordering travel or food and the prevalence of screen based entertainment as examples of this new paradigm. The computation that underpins much of our modern world leads Booch to conclude that we are therefore computational humans, but I am far less sure that simply engaging with the digital world meets the aspirations we have for our pupils to understand and change it.

Computational thinking (CT) was first coined as a phase by Seymour Papert in his 1980 book 'Mindstorms', to refer to the skills children develop when learning how to solve problems on a computer. Papert took a practical, constructionist approach that championed learning by making with programming at its heart. He was continuing a chain of thought that had been developing within computer science as the field evolved after the second world war. In his 1945 book 'How to Solve It', George Polya described the thought processes required for solving mathematical problems involving planning, calculation and reflection. By the late 1950's the term 'algorithmizing' was being used by Alan Perlis to describe the thought patterns required to understand the increasing computer automation across the culture (Katz, 1960). In the 1970’s, Donald Knuth noted how breaking a solution down to the tiny composite steps required by a computer engendered a truly deep understanding of the subject (Knuth, 1974). So some serious thought had been given to these 'computational habits of mind', as Edsger Dijkstra called CT in 1979; the term was widely popularised by Jennette Wing in 2006 when she wrote 'CT involves solving problems, designing systems, and understanding human behaviour by drawing on the concepts fundamental to computer science.'

Wing's contribution to the debate has been influential. Her goal of teaching all young people to think like computer scientists caught the collective imagination and triggered computing-for-all movements around the world, including Computing At School in this country. The problem with her 2006 definition is that it focuses on the concepts at the expense of the computation. This has led to two on-going misconceptions; that an algorithm can be any sequence of steps regardless of their dependence on technology, and that the thought process itself is useful in fields beyond computing.

I have been employed as a software engineer since the 1990's and, based on my experience in the field, I have arrived at my own short definition: CT is being able to recognise the elements of a problem that can be most effectively solved by a computer.

My definition clearly requires the presence of a computer and by 2010 Wing's definition had evolved to be closer to mine as, with input from Cuny & Snyder, she had included reference to the solution being carried out by an 'information-processing agent'. Here we arrive at the most common general definition cited over 5000 times in the literature: 'The thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information processing agent' (Cuny et al, 2010).

For me this definition of CT is still too ambiguous, I agree with Tedre and Denning (2016) that Alfred Aho has produced the clearest and most concise version when he defined CT as ‘The thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms' (Aho, 2012).

Literature review

A broad definition of CT, such as Aho's, is helpful, but so too is an operational definition of the key components of CT so that they might be better understood and ultimately taught and assessed effectively throughout the key stages of education. Since 2006 much work has been completed in this area (National Research Council, 2010; Barr and Stephenson, 2011). In 2013 Selby and Woollard reviewed this work with a view of establishing a common vocabulary for describing CT. They found consensus around three terms that had each developed after being included by Wing in her 2006 article. These are that CT is a 'thought process' that involves 'abstraction' and 'decomposition'. The list was expanded to include six terms. 'Generalization' was added as an adjunct to decomposition to describe the skill of recognising distinct repeating computational patterns. Although not used by Wing in her original essay (2006) 'algorithmic thinking' was also included to describe the uniquely sequential way a computer completes sets of step-by-step instructions. 'Evaluation' rounded off the list, it is the ability to review the strengths and weaknesses of a solution and is well established in many other related engineering disciplines.

The literature on the teaching and assessing CT reflects the debate that exists around a clear single operational definition of CT. For Lee et al (2011) the three key elements are 'abstraction', 'automation' and 'analysis'. Other researchers see CT in terms of 'concepts', 'practises' and 'perspectives' (Brennan & Resnick, 2012; Lye & Koh, 2014). What unites these authors with others such as Grover et al (2014) and Barr and Stephenson (2011) is that they all stress the importance of programming in their work. The supposition that, through programming, students can develop a working understanding of CT by exposure to the concepts, has a long and well established theoretical base. Piaget developed theories of how knowledge is actively constructed by the individual, it was his view that learning is best achieved as a by-product of engagement in, and experience of, the world first hand (Piaget, 1964). Piaget's student, Seymour Papert expanded upon these ideas with his insight that the development of a student's internal construction could be supported and encouraged by making real world artefacts (Papert, 1991). He invented the Logo programming language with Wally Feurzeig and Cynthia Solomon at MIT. Children programmed a small robotic turtle, through Logo's body syntonic commands, to draw on a computer screen. This approach taps into students' innate pleasure in making real things for other people to respond to. Turtle graphics is currently implemented in many programming languages for use as a classroom tool. ‘Perhaps if we wrote programs from childhood on, as adults we'd be able to read them’ (Perlis, 1982).

By the practise of computer programming students are developing tacit skills, akin to riding a bicycle, and the assessment of these skills remains a challenge to educators. In keeping with the spirit of Papert's work, Brennan and Resnick (2012) share digital artefacts, produced by children using the Scratch programming language, via YouTube and the Scratch on-line community. They go on to examine these projects for evidence of sequences, loops, parallelism, events, conditionals, operators, and data. A similar approach was attempted by Werner et al (2012) with their Fairy Assessment, written in the Alice programming language. Working with a key stage 3 aged cohort this study asked students to attempt solutions to a set of part-solved problems in order to identify students’ understanding and use of CT concepts.

Both Bocconi et al (2016) and Grover et al (2014) identify the need to develop objective assessment instruments. Brennan and Resnick (2012) concur and add that multiple modes of assessment are required and that ongoing formative assessment is preferable to one-off summative assessment. These authors highlight the high demands on teachers' time and skills required, and the overarching need for high quality professional development and support, to achieve this.

Repeatedly the literature reiterates the importance of constructionist-based problem solving in a 'rich computational environment' (Lee et al, 2011). How does this transfer to the advice offered to secondary computing teachers?

The aim is for students to gain practical experience of designing, writing and debugging programs (OCR computer science GCSE subject content, 2015). This is supported by the literature, but practical constraints have resulted in a situation where little of the evidence on effective assessment has been implemented. It is required that twenty hours be timetabled for the completion of a programming project, which can be chosen from a list of three. However, the code is not formally assessed. Candidates are commonly asked to produce pseudocode to express their understanding of CT. Pseudocode has no concrete rules or syntax and learning it appears to have little educational benefit beyond as a tool for summative assessment.

Also at odds with the evidence found in the literature is the way the course units are structured. The current GCSE specifications lists CT as a stand-alone topic consisting of decomposition, abstraction and algorithmic thinking (OCR, 2016; AQA, 2016). The associated resources include, as examples to illustrate these concepts, labelling the parts of a bicycle (decomposition), the London Underground map (abstraction) and making a cup of tea (algorithmic thinking). This approach makes two assumptions for which there is little evidence in the research literature; that CT can be taught by mastering blocks of knowledge rather than by developing competencies and sensibilities, and that examples do not necessarily have to be related to computational solutions. As a result this part of the computer science course seems unlikely to inspire students to 'design, create and experiment in areas they care about' (Bocconi et al, 2016) and is therefore a missed opportunity.

Reflection

I have been employed as a software engineer over the last twenty-five years. I have been able to use the practical experience of programming that I gained during my professional life to reflect on CT.

A typical problem that my recent employer ThermoFisher Scientific might set the development team would be to reduce the size of an x-ray spot on a sample of material from 30µm to 10µm in diameter, to improve sample resolution and so gain competitive advantage. The chief scientist would gather a multidisciplinary team including physicists, electrical engineers, mechanical engineers, project managers and software engineers. Various ideas for possible solutions will be suggested and evaluated. All of the contributors will have a broad understanding of CT concepts but it is the software engineer who is able to offer a detailed understanding of the viability of the computational innovations proposed. An example of this would be the application of a proportional–integral–derivative control loop in the firmware to support the fine control of the small spot. CT is the tool used to recognize applications such as these but also the tool used to express solutions, by the programming process, in working code. The application of CT is not explicit and the terms abstraction, decomposition and algorithmic thinking are not commonly used. At no point are either flow-charts or pseudocode employed as part of the CT process. However, when coupled with a deep understanding of the capabilities and potential of the technology available, CT is a useful tool for the design and development of practical solutions.

My early exposure to CT was when I was taught object-orientation in the early 90's from object first principles. I can recall the feeling of being ungrounded as a problem domain was broken down into objects with little reference to how these objects were to be implemented in actual code, something Ben-Ari calls the object-oriented paradox (2001). It asks the question, how do you use abstraction to manage complexity without first gaining a thorough appreciation of the details? It was only by establishing a firm grounding in the nature of the computer by studying the C programming language, and then by developing a working knowledge of how that can be applied to object-orientation by learning C++, did I truly become an effective and confident object designer.

I volunteer at Brighton CoderDojo, which is an informal programming club for young people. Here CT is encouraged by mentoring young people with their self-selected projects in the fields of robotics, android apps, game development and general programming in a variety of computer languages. Much of the project work is self-guided by attendees; the mentors find that some common questions concerning variables, data types, functions, file handling, binary and hexadecimal manipulation and boolean logic do keep recurring. This does seem to confirm the findings of Grover et al (2014) that the programming experience isn't necessarily educative without support and guidance that encourages thinking, planning and reflection as part the of the making process.

Analysis

Two of the themes that emerge from the literature are that computer programming is hugely beneficial to the development of CT skills and that the learning of it is constructivist in nature. It is my proposition that the C programming language is the most appropriate tool available to teach programming due to its synergy with the construction of an accurate mental model of the underlying computer architecture. The OCR Computer Science Programming Languages Guide (2015) concurs with me that 'in the ideal world, C would be the language to teach'.

The reality, which I have observed during my secondary school placement, is that Python is the pre-eminent text-based programming language employed. Python omits some important features that are included in exam board specifications such as support for switch statements, arrays and do-while loops. Python doesn't support structured programming as well as C, where the listing of function prototypes in header files creates clear interfaces that, in turn, support abstractions. Python's use of indentation to identify code blocks causes some confusion to beginners. Most concerning to me are the constraints Python has built in, in the name of simplicity, that have the potential to seriously obfuscate an accurate mental model of a computer.

For example, a feature of the way the CPU performs the decode section of the fetch, decode, execute cycle can involve indirect memory addressing. The application of this principle can be clearly demonstrated in C by passing a variable to a function, by reference rather than by copy, and changing that variable within the function call. That change will then be reflected back in the calling code too. Madison (1985) has completed work on how these concepts are fraught with the potential for an inaccurate mental representation, of how parameters are passed, to be constructed in the student's mind. This should highlight the need for the development of pedagogical techniques and resources to help teachers with this task. In contrast to C, Python takes the unusual approach of passing all parameters by reference, but then adding a constraint. Should the parameter's value be altered within a function, a copy of that parameter is made so that the calling code remains unaffected. For me this represents a lowering of the standards we as teachers should expect of our pupils. Superficially the teaching is simplified but at the expense of a clear demonstration of indirect memory addressing.

Even though bold claims are still being made such as; ‘Computational thinking also gives a new paradigm for thinking about and understanding the world’ (Csizmadia et al, 2015), there is little evidence in the literature to support the notion that you can uproot the principles of CT from the computer and apply them to other areas successfully (Guzdial, 2015). However, CT still has a contribution to make to other subjects; it is just that claims must be substantiated by linking solutions to computation. Consider the effect digital sequencers have had on music since their invention in the early 1970's.

As the field of computer science matures, so too does the complexity of problems that can be solved. In the early days computers were commonly stand-alone single-core machines; since then we have witnessed a revolution in the amount of computing power available. When Deep Blue defeated the world chess champion Garry Kasparov in 1997, it used a brute force approach whereby all possible candidates for the solution were systematically evaluated. Almost two decades later Google's DeepMind AlphaGo program used machine learning to defeat the Go world champion Lee Se-dol in 2016. AlphaGo runs on Google's cloud computer network, using 1,920 processors and a further 260 graphics processing units.

I have observed algorithms being illustrated in the classroom by exploring the steps involved in preparing a breakfast, getting dressed in morning, or selecting a film to watch that evening. None of these examples have a ready computational solution. Without the ability to follow an example through to working code I don't think any insight can be conveyed, and quite possibly an incorrect model of how computers work is constructed in the students' minds. When it comes to assessment I have seen 'A' level papers requiring candidates to explain the concept of abstraction (for 4 marks), I ascertain from the mark scheme that what's required includes ignoring the unnecessary and using symbolic representation to produce effective and focused programs. All of which can be learnt by rote and says little about a candidate's ability to apply the skill.

Conclusion

Almost as important as understanding what CT is, is being able to recognise what is it not.

Most importantly, CT must be grounded in the computer programming process. Algorithms are detailed sequences of steps designed to control a computation without recourse to human judgement. Algorithms are not arbitrary recipes for carrying out tasks.

CT can be usefully applied to any problem domain where an automated solution is sought, an example would be modelling the effect of a vaccination programme on a disease outbreak. CT is not a superior problem solving skill, it is analogous to other kinds of thinking such as scientific thinking or critical thinking. Claims that it brings extra benefits to other academic disciplines remain unsubstantiated.

CT can empower children to understand and change the world. Michael Young (2007) names new ways of thinking about the world as ‘powerful knowledge’. The modern world is increasingly a digital landscape and equipping students with the wherewithal to navigate and creatively contribute to that landscape is vitally important. Against this backdrop powerful knowledge is increasingly specialist knowledge; computer programming is an excellent example of such specialist knowledge.

Using computational tools is not CT. Most young people can be described as 'digital natives', owning digital devices that they are adept at using effectively, but it is only by learning CT do they gain a thorough understanding and appreciation of the technology they use. Moreover CT can offer the skill set necessary to confidently and creativity reconfigure and make additions to the digital landscape.

Competency-based skills assessment should be encouraged as a means of monitoring the development of CT. Assessing knowledge of CT by testing for facts and information may not measure CT skill.

Educators should be wary of teaching tools that prize simplicity above an authentic representation of the underlying computer architecture. Efforts should be directed at developing resources that will help teachers to unpick complex area in the curriculum in a coherent and iterative manner.

Computational models are evolving with increases in processing power. Aho (2012) considers the implementations of modern supercomputers and how they have transformed computational biology and helped in the quest to model the behaviour of cells and their DNA. Being able to envisage the solutions becoming reachable as the technology develops and the future unfolds is a key skill. I am reminded of a prescient observation from Ada Lovelace in 1842. When writing notes on Babbage's (theoretical) Analytical Engine, she said 'Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.' She had identified an area of application that neither Babbage nor anyone else had foreseen based on a clever and imaginative, full and confident understanding of the nature of the machine in question. It is by equipping our students with something of this confidence, born of understanding, that is the real prize CT can bring to the curriculum.

Bibliography

Aho, A.V., 2012. Computation and computational thinking. The Computer Journal, 55(7), pp.832-835.

AQA, 2016. GCSE GCSE Computer Science (9-1) - 8520, section 3.1.1

Barr, V. and Stephenson, C., 2011. Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community?. Inroads, 2(1), pp.48-54.

Ben-Ari, M., 2001. Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching, 20(1), pp.45-73.

Bers, M.U., Flannery, L., Kazakoff, E.R. and Sullivan, A., 2014. Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. Computers & Education, 72, pp.145-157.

Blackwood, N., 2016. Digital skills crisis: second report of Session 2016–17: report, together with formal minutes relating to the report: ordered by the House of Commons to be printed 7 Jun 2016.

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., Engelhardt, K., Kampylis, P. and Punie, Y., 2016. Developing computational thinking in compulsory education. European Commission, JRC Science for Policy Report.

Booch, G., 2016. The Computational Human. IEEE Software, 33(2), pp.8-10.

Brennan, K. and Resnick, M., 2012, April. New frameworks for studying and assessing the development of computational thinking. In Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada (Vol. 1, p. 25).

Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C. and Woollard, J., 2015. Computational thinking-A guide for teachers.

Cuny, J., Snyder, L. and Wing, J.M., 2010. Demystifying computational thinking for non-computer scientists. Unpublished manuscript in progress, referenced in http://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf

DfE, 2013. Computing programmes of study: Key stages 3 and 4.

Dijkstra, E.W., 1979, September. My hopes of computing science (EWD709). In Proceedings of the 4th international conference on Software engineering (pp. 442-448). IEEE Press.

Grover, S., Cooper, S. and Pea, R., 2014, June. Assessing computational learning in K-12. In Proceedings of the 2014 conference on Innovation & technology in computer science education (pp. 57-62). ACM.

Guzdial, M., 2015. Learner-centered design of computing education: Research on computing for everyone. Synthesis Lectures on Human-Centered Informatics, 8(6), pp.1-165.

Katz, D.L., 1960. Conference report on the use of computers in engineering classroom instruction. Communications of the ACM, 3(10), pp.522-527.

Knuth, D.E., 1974. Computer programming as an art. Communications of the ACM, 17(12), pp.667-673.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J. and Werner, L., 2011. Computational thinking for youth in practice. Acm Inroads, 2(1), pp.32-37.

Lye, S.Y. and Koh, J.H.L., 2014. Review on teaching and learning of computational thinking through programming: What is next for K-12?. Computers in Human Behavior, 41, pp.51-61.

Madison, S.K., 1995. A Study of College Students' Construct of Parameter Passing Implications for Instruction.

National Research Council, 2010. Report of a workshop on the scope and nature of computational thinking. National Academies Press.

OCR, 2016. GCSE Computer Science (9-1) - J276, section 2.1

OCR, 2015. Computer Science GCSE subject content.

OCR, 2015. Computer Science Programming Languages Guide, Version 2

Papert, S., 1980. Mindstorms: Children, computers, and powerful ideas. Basic Books, Inc..

Papert, S. and Harel, I., 1991. Situating constructionism. Constructionism, 36(2), pp.1-11.

Perlis, A.J., 1982. Special feature: Epigrams on programming. ACM Sigplan Notices, 17(9), pp.7-13.

Piaget, J., 1964. Part I: Cognitive development in children: Piaget development and learning. Journal of research in science teaching, 2(3), pp.176-186.

Polya, G., 2004. How to solve it: A new aspect of mathematical method (No. 246). Princeton university press.

Selby, C. and Woollard, J., 2013. Computational thinking: the developing definition.

Tedre, M. and Denning, P.J., 2016, November. The long quest for computational thinking. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research (pp. 120-129). ACM.

Werner, L., Denner, J., Campe, S. and Kawamoto, D.C., 2012, February. The fairy performance assessment: measuring computational thinking in middle school. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 215-220). ACM.

Wing, J.M., 2006. Computational thinking. Communications of the ACM, 49(3), pp.33-35.

Young, Michael. 2007. What are schools for? Educação & Sociedade. 28. 1287-1302.