Queen Mary
University of London

So what is Computational Thinking?

Paul Curzon

Queen Mary University of London

Version 0.2, 1 June 2014

So what is Computational Thinking?

Computational thinking is an important skill set that computer scientists learn and use to solve problems. Its so important school kids are now expected to learn the skills! What on earth is it, though?

Computational thinking is a set of skills and techniques for solving problems – a really powerful way of solving problems in fact. It is the way of thinking that has led to computers taking over large parts of our lives and changing everything we do from listening to music to trading stocks and shares. There are many definitions and descriptions by different people. One way to think about it is based on the following 5 themes[1]:

- algorithmic thinking,
- evaluation,
- generalisation,
- abstraction and
- decomposition.

Let's look at each area in turn and at the sub-skills and techniques that they comprise of.

Algorithmic thinking

Algorithmic thinking is the core idea behind computational thinking. It is a set of approaches based around the idea that the solutions to problems are not just answers (like '42'). Nor are they the achievement of something (like 'finished the crossword'). Solutions are *algorithms: which just means they* are a set of instructions to follow. If you follow the instructions in the algorithm exactly you should get an actual answer to the problem or achieve whatever you were trying to achieve. Once you have an algorithmic solution you can get the answer to that problem without thinking at all: just follow the instructions 'blindly'. The power of this idea is that following the algorithm should actually give solutions for a whole group of problems not just a single instance. An algorithm for solving crosswords would be able to solve lots of crosswords. An algorithm for doing arithmetic should be able to do any calculation. Thinking in this way about problems and solutions is called **algorithmic thinking**.

Once you have an algorithm that solves a problem, anyone can solve it without thinking. They don't even have to know or understand what the algorithm ultimately does. That also means a dumb machine – a computer – can also just mechanically follow the instructions and solve any instance of the problem too. That is all computers do – follow algorithms written by people.

For example, knowing that 20 + 22 is 42 isn't good enough. A computer scientist wants an algorithm that can add any two numbers – in fact everyone

[1] By:
C.C. Selby and J. Woollard (2014) Computational Thinking: the developing definition, Southampton University eprints.

learns an algorithm to do this in primary school! Similarly, all computers have the instructions of how to do addition built in, it is so important. Computers can only act as a calculator by following their instructions telling them how to do calculations. A computer program is just an algorithms or set of algorithms written in a language a computer can follow – a programming language.

It isn't just calculations though – algorithms can be used to do all sorts of things. If we write out an algorithm in the form of a program, which just means writing the instructions in a language a computer understands, then we can get them to blindly do all sorts of things. Banks use algorithms instead of humans to do trading: to buy and sell so as to make millions of pounds of profit. NASA uses them to fly space ships to Mars. You use algorithms to play music and videos. They fly aeroplanes, help surgeons and allow us to shop. They are even creating art and music. Algorithms are now involved in all aspects of our lives. That is why understanding algorithmic thinking is so important for everyone.

Thinking algorithmically involves **logical thinking**: being very careful and precise about the details. The instructions, for example, have to cover every eventuality. Did you think to include in your instructions for adding numbers what to do with negative numbers as well as positive numbers? If you didn't then a computer might either give the wrong answer or crash when confronted with the problem. As you develop an algorithm you need to think very logically about how it works. In your head at least, if not on paper, you have to have a logical argument of why it always works. You don't want your Mars lander to crash, because you forgot a detail months after it set off just as it is finally landing on Mars. **Logical thinking** is a part of **evaluation** too as we will see.

Technology is ultimately there to be used by and support people and that means computational thinking is really about problem solving for people. Therefore, **algorithmic thinking** has to include **understanding people** and especially understanding their strengths and weaknesses. For example, suppose you are designing a security algorithm to keep your online banking system secure. You might come up with an algorithm that requires people to enter a password that is 1000 letters long and must be made of random letters – no recognisable words within it. That would be really secure! It would also be very silly. Apart from an odd genius or two, no human could reliably remember a password like that. The algorithm would be useless.

A further skill involved in **algorithmic design** is that of **creativity**. **Algorithmic design** is a very creative process. You can do it in a very plodding way of course – turning the handle on some basic techniques and that is the way most people start. Brilliant computer scientists though come up with completely new algorithms either for old problems or completely new ones. Coming up with the ideas of how to do it is obviously creative.

Creativity needs the right conditions – the people involved need a **playful state of mind** and situations that promote it. It helps to want to have fun, so it's lucky that computing is so much fun. You need to have the time and space to let your mind wander. You need to be free from too much stress, from deadlines that must be met. Of course the most creative ideas don't come from individuals but from groups of people bouncing ideas about, feeding off

each other's creativity. Companies (and countries!) that can promote that sort of work place are really going to change the world!

That's because it's not just coming up with the algorithm that takes creativity. Sometimes the creativity is also in coming up with the problem to solve. It is about coming up with a transformational idea – the idea that once we have the algorithms to do it really does change the way the world works. Devising an algorithm unlike any that has been seen before is the way to completely transform a problem, and that can transform the way we live our lives. That's especially so if you are creative enough to see a problem to solve that no one else has even noticed before ... and then solve it. Out of creativity comes **innovation** – and that needs people who actually have the drive and skill to push an idea through to the end. All those big computer-based innovations like the web, social networking, online shopping, needed people with lots of creativity to start with and then a mix of people with different computational thinking skills to make them a reality,

Sometimes it's not actually possible to create an algorithm that guarantees to get the best solution for a task either at all or within the time available (and we really do mean impossible not just hard). In those situations a heuristic algorithm is used instead. **Heuristics** don't guarantee the best solution but just gives a reasonable solution in a reasonable amount of time. **Heuristic problem solving** is needed to come up with algorithms like that. Take an in-car satnav. Suppose it was set the task by a salesman of plotting the shortest route to visit a series of clients in different towns just once ending up back at the office and without doubling back on oneself. Even with 20 clients to visit this is totally impractical to find the best solution. Make the number of locations to visit big enough (which is actually not that big) and it would take more time than that since the universe was created to get a perfect solution! Instead you could use a heuristic called a greedy algorithm. The basic idea here is that at each step you just choose the town that is nearest to where you are now to visit next. That won't always get the best answer but it will usually get a good answer.

A massively important part of algorithmic thinking is the idea of **computational modelling**. This is the idea that you can take some thing in the real world (say the weather) that you want to understand better and create an algorithm that does the same thing in a virtual world – that **simulates** it. By running the algorithm you can make predictions about what the real thing will do (predict whether it will rain tomorrow or not, for example).

Computational modelling is the main way that computational thinking is transforming all the other subjects. For example, computational modelling is used in biology where biologists create algorithmic models of the heart or of cancer cells so that they can then do virtual experiments. It is a way to reduce the number of animals used in experiments too – run the experiment on a virtual animal instead. In economics, computational models of the economy can be used to predict what changes the politicians are planning might actually have. Climate scientists use models to predict the range of possible consequences of global warming. Some artists use computational models as a new way to create art – paintings painted by algorithms! Computational

modelling is being used in physics, in chemistry, in geography, in archaeology and lots of other subjects too as a new way of doing the subject.

Once you have a good computational model you can run lots of experiments on the simulations. It is more than that though. Computational modelling has even changed the way we play games: games like World of Warcraft are just a computational model of a fantasy world, for example; sports games are computational models of the game. In both cases a model of the laws of physics is built in to the program, so that, for example, what goes up comes down again! The rise of computational modelling is the reason why nowadays whatever subject you do, having computational thinking skills, not just IT skills, is important.

So algorithmic thinking is made up of a whole bunch of sub-skills including:

- logical thinking
- understanding people
- computational modelling
- heuristic problem solving
- creativity and a playful state of mind

Good algorithmic design also involves the use of abstraction, decomposition, and generalisation and their sub-skills. Because they are so important we look at them separately below.

Evaluation

Evaluation is about checking that your solution is a good solution. There are several different kinds of things that you need to evaluate. The most basic is **functional correctness** – does your algorithm actually work? Always! Whatever happens will it always do the right thing and give the right answer? You need to be sure it does.

Another thing that you might evaluate is **performance**. How fast is your algorithm? Are there other algorithms that would do the job more quickly? Are there particular situations where your algorithm is slow? Do those situations matter? For example, one algorithm used to sort things into order (called quicksort) is generally really fast. However if you give it a series of things to sort that just happen to be in the right order already, it is ridiculously slow. It takes longer to sort things that are already sorted than things that are completely mixed up. It is a great algorithm, but it would be silly to use if you knew you had a pile of things that were almost in the right order already. There's rarely a single best algorithm for a task. It depends on the situation, and you need to evaluate how well an algorithm fits that situation.

A third really important part of evaluation is about whether the solution you have come up with is actually **fit for purpose**. Algorithms are there to solve problems for people. They must work in a way that allows people to use them as we saw when talking about algorithmic design. You therefore have to evaluate them for how easy they are to use and how good an experience it is for people who are using them. You do not want them to lead to people making mistakes, and you don't want them to lead to people being frustrated or angry. That in particular involves **understanding people**. What you are really asking in this kind of evaluation is "Does it work well given the way

people are? Does it play to our strengths and limit the problems caused by our weaknesses?

Suppose you are designing a medical device to deliver a pain relief drug to patients. The nurse sets up the dose, hits go and it then pumps drug through a tube into their arm over several hours. Now obviously you want it to be functionally correct. If the nurse programs it to deliver 15.5 milligrams per hour for 6 hours then that is what it should do. It also has to work suitably quickly. It's no good if the nurse has to wait several minutes after typing in the dose before it will start because it is taking a long time to set itself up. Even more importantly it should be easy to use. It should help prevent the nurse making mistakes and help recover if a mistake is made. If the nurse enters 155 instead of 15.5 by mistake and that is a dangerous amount of that drug then the machine should at least warn the nurse, and give them a chance to undo it.

There are very many techniques and so skills used in evaluation. It involves **rigorous testing** – being very, very organised about the way you check that an algorithm or program implementing it actually works. That involves doing lots of testing, not just trying the program once or twice and deciding it always works. It also involves being clever in the way you pick what situations to test to increase the chances that there are no surprises. That in itself takes some logical thinking about what you need to test to be sure you have good coverage of the possibilities.

A complimentary approach to **testing** is **rigorous argument**. Rather than running a program to see if it works we can use the power of argument. Using logical reasoning we can come up with an argument as to why certain tests are enough to guarantee the whole is right. Taking this to an extreme our arguments can be about why the algorithm or program as a whole always works using **logical proof**, a variation on the kind of proofs that mathematicians do. When you create an algorithm or program you have in your head reasons why you think it works. At the evaluation stage you are checking those reasons, and making sure you haven't missed any detail. Often such proofs are done on an **abstraction** of the system (see below). That just means irrelevant details are ignored to make the proof easier to complete.

You can also evaluate different parts of a solution separately. This is a use of **decomposition** where we think of a problem or a system as a lot of simpler parts that can be worked on separately. As those parts are smaller, that is simpler. Evaluation is not something you just do at the end when you have a solution – you do it as you come up with solutions – as you develop the algorithms, the programs and their interfaces. You have to do it repeatedly as you develop a solution, creating early prototypes and evaluating them in different ways, solving problems that arise. **Decomposition** helps as it allows you to evaluate each smaller part separately as you complete it. You can check each part is ok and fix any mistakes you do find before you worry about whether the whole thing works.

When it comes to evaluating whether our solutions are fit for purpose we can also use methods a bit like testing where we try out the system:

observational methods. The difference is that this kind of evaluation involves real people using the system we are evaluating. One way is to set up experiments where we watch people using our system in lab conditions – essentially running scientific experiments. Another is to go out 'into the wild' and watch the system being used for real. In both cases we are looking for things going wrong, or things that people have difficulties with, asking ourselves all the time: "could we change the system to make things easier for people".

We can also use **analytical methods.** This essentially involves getting experts who **understand people,** and also understand what makes good or bad design, to look at a system in a very organised way. Their aim is to predict potential problems – things about the system that people are likely to struggle with. They might for example step through a particular task, and ask at each step "How might a person misunderstand what to do here?" The experts might use specific principles as guidance like: "It should always be possible to undo the last step if a mistake is made". If they find a situation where undo isn't available, then that can be reported as a problem to fix.

Evaluation thus makes use of another set of computational thinking skills including:

- Rigorous testing,
- Rigorous argument,
- Logical proof,
- Analytical methods,
- Observational methods,
- Understanding people.

It also draws on decomposition and abstraction.

Abstraction

Abstraction is the **hiding of details** in some way to make a problem easier to deal with. There are lots of different ways you can hide details. It can be done when designing algorithms and when evaluating them.

For example, one very important use of **abstraction,** called **control abstraction,** is in grouping instructions together so that you have instructions that do bigger steps. The idea here is that you are hiding the details of the individual steps needed. Recipe books do this all the time. They say things like "boil the potatoes". That involves lots of steps: filling a pan with water, turning the heat on, bringing it to the boil, adding the potatoes, and so on. All those steps are brought together into a simple command "boil the potatoes". To follow the instructions you need all the detail, but it helps in writing down instructions and when thinking about the algorithm (or recipe) as a whole to work with the big steps. All that detail is too much to think about. This form of abstraction is linked very closely to **decomposition** – see below.

Another kind of **abstraction,** called **data abstraction.** is where you hide the details of how data is stored. For example, numbers are actually stored in a computer in binary – as a sequence of 0s and 1s. The number 16 might be stored actually as 00010000. We ignore that fact when thinking about numbers though. We just think of them as the decimal numbers like 16 that

we know and are used to using. When we write programs we use decimal numbers in the instructions, not binary. Our programs ask the people using them to enter numbers in decimal instead of binary too. Ultimately though the computer works with binary numbers. No one using the program has to know that the numbers are actually stored like that though: that detail is hidden!

We don't just use **abstraction** when writing programs. We can also use it when evaluating our programs. For example if we want to decide which of two different algorithms that do the same thing is the quickest, we don't need to think about time itself. We can hide the detail of actual time and think about work done instead – how many operations do you need to follow with each algorithm to get the job done. If one algorithm involves following 100 instructions, and the other only 10, then the second is going to be the quickest. We can work that out just by counting the operations, without timing anything at all. We hide the detail of how much time operations take to make the problem easier.

Abstraction in itself requires a certain amount of **creativity** in working out the best details to hide to make the job as easy as possible.

Abstraction skills include:

- Control abstraction
- Data abstraction
- Creativity

Generalisation

Generalisation is the idea of taking a problem we have solved and adapting the solution (the algorithm) so that it solves other similar problems. For example, suppose we have to find our name in a seating plan – a list of names that tells us where to sit. Rather than looking randomly, we might start at the top of the list and check each name in turn until we come to ours. On another day we have to find a CD on a shelf. We might recognise that it is the same problem. In doing so we are **pattern matching**: matching one problem to another. Once we have realised two problems are the same in this way, we can use the same solution for both. We don't need to come up with an algorithm from scratch. We start at one end and run our finger along the shelf, checking each CD in turn until we find the one we want (or get to the end meaning it isn't there). We have **generalised** or **transformed** the algorithm – the solution to the first problem – so that it can be used to solve the new problem.

We can take this a step further if we realise that any time we have a sequence of things lined up in some way and have the problem of finding something in it we can use this solution. We have now **generalised** the problem of finding a name to finding anything, and **generalised** the list of names to any sequence of things lined up. We have transformed our algorithm in to a general search algorithm – it is an algorithm that doesn't just solve one problem but any problem of that kind – it can be used any time we are searching for something. We have **generalised** a solution that was invented to solve just one problem (finding our name) to be solutions to a whole class of problem.

Notice that to do the **generalisation** we also needed to hide some of the detail. We don't want to think about the detail of names or CDs, so we generalise that in to being 'a thing'. This is actually using a form of **abstraction** to do the generalisation.

Sometimes we do **generalisation** in this way to create very general algorithms that we can use in lots of situations as above. At other times we just realise that a new problem in a completely new area is similar to something we have done before, so we do a one-off generalisation of it – **transforming the problem** (and so solution) from one domain to the other. For example, phones use predictive texting – as you type a word they guess from the letters typed so far what the whole word will be. People who are totally paralysed and cannot speak communicate by spelling out words a letter at a time by blinking their eye. The same predictive texting algorithm can be used in this situation too – the people working out what is being said can guess the whole word early too. The same algorithm can be used for two apparently different problems once you realise there are similarities by **pattern matching**.

Pattern matching and **generalisation** can be used at all sorts of levels, from spotting that a whole problem is the same as one we've solved before to realising that a small fragment is similar. Often as we build up a program, individual parts of it are similar to things we've come across before – perhaps we need the program to repeatedly ask if we want to do some thing again (like play the game again when we win). If we have written code for that before then realising this is the same by pattern matching, we can just copy that bit of code, integrating it into our new program, without having to think through the commands needed from scratch.

Generalisation can also be helpful for **evaluation**. Suppose we have created some general algorithm. We can evaluate it once and everything we learn about it applies to every new use of it. For example, once we know how fast our algorithm is in different situations, and how it compares to other algorithms that do the same thing, we can use that in deciding whether to choose it for some new job.

Generalisation thus consists of sub-skills including:

- Pattern matching,
- Generalising algorithms from individual problems to a class of problem,
- Transforming problems to new domains,
- Abstraction.

Decomposition

Decomposition involves breaking a big problem into smaller problems that are easier to solve. We can solve the big problem by solving each of the small problems individually. This is a really powerful way to think about solving problems. It is **decomposition** that has allowed us to write complicated programs that are millions of instructions long. Without that we would not have programs doing all the things computers are now used for.

Decomposition is used in writing programs in a way that links closely to **control abstraction**. The idea is to break a program you are writing into lots of separate tasks. You then write separate little programs for each of those smaller tasks. Each of those small programs is easy to write. The bigger program that combines them is easier to write too as you don't have to think about all the details at once. Once the parts are complete you only have to think about what they do, not how they do it. To make that easier you give each a name that makes clear *what* it does but hides the detail of how it does it (and naming things like that is another kind of **abstraction**). Then when you put the small programs together to make the big one, you don't have to think about all the fiddly detail anymore.

Decomposition in this way gives another way of using generalisation. If the smaller programs are written in a suitably general way then you may be able to use them again in other big programs. You may even be able to just take existing programs to solve some of the sub-problems if you can **pattern match** them to existing things.

There are some special techniques used to do decomposition in a way that makes it easier to come up with solutions that work really quickly. One is **Divide and conquer problem solving**. The idea here is to solve a problem by finding a way to break it in to smaller but otherwise identical problems. To solve the problem of searching through a telephone directory, we can open it in the middle and see if the names on that page are before or after the one we are looking for. We then know which half to search ignoring the other half. We now have a similar but smaller problem – searching half a telephone directory. We solve that in the same way, going to the middle of the remaining half and so on, till we find the name we are looking for. That gives a far faster solution. Thinking in this way is an example of **recursive problem solving**: a special form of **algorithmic thinking**. It is the basic idea of writing algorithms that break problems into similar smaller problems. The special thing about **divide and conquer** over recursion is the idea of splitting the problem into halves (or thirds, quarters, etc) so that each new problem is roughly the same size and a lot smaller and so easier to solve than the original.

Decomposition thus consists of sub-skills including:

- Pattern matching
- Abstraction
- Divide and conquer problem solving
- Recursive problem solving

Summary

Computational thinking is a set of problem solving skills that focus around creating algorithms, and ultimately programs, to do things. It involves algorithmic thinking, evaluation, abstraction, generalisation and decomposition. Above all it is a very creative activity.

It's important to realize though that these areas are not really distinct activities. Computer scientists actually use them all together in a rich and

linked way when solving problems using computational thinking[2]. Many of the skills also overlap with skills used by mathematicians, design specialists and engineers. Computational thinking as practiced by computer scientists is the skill set when it is all pulled together in this rich way, and that gives a different way of thinking about both problems and systems. It is a way of thinking that has changed the way we all live, work and play and will continue to do so in the future.

Live demonstration of computational thinking activities

Teaching London Computing give live sessions for teachers demonstrating this and our other activities. See www.teachinglondoncomputing.org/ for details. Videos of some activities are also available or in preparation.

For courses and resources developed for teachers in London, including our classroom activities and slides directly linked to this booklet visit teachinglondoncomputing.org a joint project between Queen Mary University of London and King's College London funded by the Greater London Assembly. This booklet also drew on ideas from the CHI+MED project funded by EPSRC (www.chi-med.ac.uk).

For more cs4fn resources for schools, visit www.cs4fn.org/teachers/

We welcome feedback on this and our other resources. Please email Paul Curzon (p.curzon@qmul.ac.uk) with any thoughts or constructive comments you have.

Department for Education
SUPPORTED BY MAYOR OF LONDON
COMPUTING AT SCHOOL EDUCATE · ENGAGE · ENCOURAGE

[2] Our contextually rich problem solving 'stories' about computational thinking that show the richness of this are available from Teaching London Computing eg
P. Curzon (2014) Computational Thinking: searching to speak, Queen Mary University of London. Available from: http://teachinglondoncomputing.org/resources/inspiring-computing-stories/