Two Lectures on Computational Thinking: a brief essay.

~~Draft: please do not distribute.~~ This version dated 8 November 2012 released to support CSC9A1 teaching at University of Stirling.

Introduction

It's not immediately obvious what Computational Thinking (CT) is. There's little agreement about it. The Wikipedia page, http://en.wikipedia.org/wiki/Computational_thinking, reckons it's a problem solving technique, but is otherwise vague. The Apollo Research Institute reckons it's the "Ability to translate vast amounts of data into abstract concepts and to understand data-based reasoning" (http://apolloresearchinstitute.com/sites/default/files/future-work-skills-2020-computational-thinking-02-14-12.pdf). And NSF and others in the US reckon it's a problem solving process (see the operational definition of Computational thinking for K12 education http://www.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf).

One thing that CT certainly is not is *learning to think like a computer*. Indeed, it's almost the opposite, because it's about learning to think in a way that enables you to make sensible use of the capabilities that computers offer you. And if the computer could do that for itself, there'd be no need for it. But what does that actually imply?

I suggest it's about being able to implement solutions to problems. That is to say, going all the way from the problem statement (which is probably not a clear statement of the problem) though design, to some kind of solution, in this case using, in some sense, a computer. Of course, the best solution might not include a computer at all, or might use it in some unexpected fashion (like maintaining air-flow in a room by holding a door open with an old desktop machine!). More realistically, the solution might altering changing a paper system, … or something else which solves the problem without using a computer. But I digress.

Computers offer large number of facilities now: there is a vast array of ready-to-use software available, as well as powerful programming systems, whether based round a core of routines like MATLAB or Mathematica, or round a scripting language, like R or Perl or Python, or round a full-scale programming language like Java or Objective C or C++. Clearly, if what you want to do is to enhance a photograph, you use Photoshop (or Gimp is money is short!), or if you want to create a simple invoice you use a spreadsheet: but often the problem you want to solve doesn't quite fit in with exactly what the available software does. And, of course, you may not know which piece of stftware would really suit the problem you are trying to solve – the range of available software is vast, and its title might not immediately make it clear that this is what you really want to use.

Then you have a problem to solve. And that means *thinking*, rather than following a set of prescribed steps. And it's that *thinking* that lies at the root of computational thinking. What sort of *thinking* is going to enable the solution of the problem?

So how does one teach problem-solving? How does one teach thinking (and computational thinking in particular)?

Should I start with the Hitch-Hikers Guide to the Galaxy quote: *Don't Panic*!

Certainly panicking in the face of a problem is very unlikely ever to help. Better to think about the problem, breaking it up, trying to identify what can be done, and what the really (apparently) tricky parts are. Better to stand back just a little, and try to see the problem for what it is: to understand the problem and in particular, to understand what is needing to be done, and then to be able to think of ways to tackle it. (See *Zen and the art of motorcycle maintenance* by Robert Pirsig: another useful book related to problem-solving).

The most famous book in this area is Polya's "How to solve it": it was written in 1944, and published in 1945, and I don't think it's ever been out of print. According to Wikipedia it has sold over 1 million copies. It's not a long book, nor a difficult book, and though a mathematician wrote it, it's not just about mathematics. Polya was, clearly, a great teacher. Quite apart from this pair of

lectures, the book is well worth reading. (For a very brief introduction you could look at the Wikipedia page on this book.)

So, following some of his ideas, instead of thinking about all the sage advice that I might attempt to impart, I will consider in what way I want to influence the student's behaviour by means of my pair of lectures. That's not unreasonable: my aim is to improve their overall problem-solving skills and to enable them to be able to solve the kinds of problems that programming computers will throw at them. I know that there's no rote method of solving such problems, and I also know some of the wrong ways of trying to make programs work (for example, changing lines at random, moving variable declarations around from method to object to see if you can get rid of annoying compilation errors). And taking advice from solving Mathematics problems, I know that it is often the case that one should try to see what the given premises imply, even if one cannot directly see how such attempts might help to solve the stated problem. Sometimes simply blindly building on them can lead to insights that might not appear if one just read and re-read the premises!

Lectures

Here's how I will divide up the two lectures?

- Lecture one will be about problem-solving as such, with some examples from Polya, and from Michalewicz & Fogel (How to Solve it: Modern Heuristics, Springer 2000)[1]. This is perhaps more about problems in general, and less about computational problems in particular. And here's another easier problem[2]
- Lecture two will be about using the different types of constructions available in imperative & OO languages (if, while/repeat, for; functional abstractions, objects & inheritance (might not mention)): how and when to use these in terms of solving problems.

Essentially, I'm thinking of there being two different aspects of CT that I want to talk about, one of which is about generic problem solving – perhaps it's rather more mathematical – and the other being more tightly bound to Computing – and programming in particular.

Whichever way around, I need to use examples. In lecture 1, the example on page 9-10 of M&F (footnote 1) might be good to start with: it illustrates using **all** the information, even when it's really not very clear that some of the statements represent useful information at all. It's perhaps a little tricky, and the problem in footnote 2 is easier. Certainly, the idea that one has a problem to solve, and no initial guidance on how to solve it is good and relevant, though connecting the

---

[1] Two men meet on the street. They haven't seen each other for many years. They talk about various things, and then after some time one of them says: "Since you're a professor in mathematics, I'd like to give you a problem to solve. You know, today's a very special day for me. All three of my sons celebrate their birthday this very day! So, can you tell me how old each of them is?" "Sure," answers the mathematician, "but you'll have to tell me something about them." "OK, I'll give you some hints," replies the father of the three sons, "The product of the ages of my sons is 36." "That's fine," says the mathematician, "but I'll need more than just this." "The sum of their ages is equal to the number of windows in that building," says the father pointing at a structure next to them. The mathematician thinks for some time and replies, "Still I need an additional hint to solve your puzzle." "My oldest son has blue eyes," says the other man. "Oh, this is sufficient!" exclaims the mathematician, and he gives the father the correct answer: the ages of his three sons. (Michalewicz and Fogel, p9-10)

[2] Two bicycles are approaching each other at a constant speed of ten miles an hour. When they are two miles apart, a bird leaves one bicycle and flies toward the other at a speed of fifty miles an hour. Upon reaching that bicycle it immediately reverses direction. This continues until the two bicycles meet. How far does the bird fly altogether?
At first glance, it appears that the only way to solve this problem is to add up the progressively smaller distances that the bird covers as the two bicycles approach each other. But there's an easier way to solve the problem. Simply figure out how long it is before the two bicycles reach each other (six minutes) and calculate how much distances the bird covers in that time (five miles).

more mathematical problem solving concept on to the issues that arise in problem solving in Computing problems will require thought!

What other, more computational, examples might I use in lecture 1?

Consider trying to find the peaks in a 1-dimensional sampled signal. (For example, looking for action potentials in low-pass filtered electrophysiological signals, but that's probably too obscure for these students: is there a simpler problem with the same types of issues? Perhaps looking for the peaks in the valuation of some stock, quoted daily, or of the value of the £ against the $, where the values are sampled, say, hourly during trading periods.) How can one find the peaks? What *is* a peak? [a value with a smaller one before and after it? But what if there's a plateau? And what about very small changes, that might be caused by noise we'd like to ignore?]

From there, I will consider the more general issues. Here's the generic concepts copied from the beginning of Polya's book:

UNDERSTANDING THE PROBLEM

First.
You have to *understand* the problem.

What is the unknown? What are the data? What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?

Draw a figure. Introduce suitable notation.

Separate the various parts of the condition. Can you write them down?

DEVISING A PLAN

Second.
Find the connection between the data and the unknown. You may be obliged to consider auxiliary problems if an immediate connection cannot be found. You should obtain eventually a *plan* of the solution.

Have you seen it before? Or have you seen the same problem in a slightly different form?

Do you know a related problem? Do you know a theorem that could be useful?

Look at the unknown! And try to think of a familiar problem having the same or a similar unknown.

Here is a problem related to yours and solved before. Could you use it? Could you use its result? Could you use its method? Should you introduce some auxiliary element in order to make its use possible?

Could you restate the problem? Could you restate it still differently? Go back to definitions.

If you cannot solve the proposed problem try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? Keep only a part of the condition, drop the other part; how far is the unknown then determined, how can it vary? Could you derive something useful from the data? Could you think of other data appropriate to determine the unknown? Could you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other?

Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

CARRYING OUT THE PLAN

Third.
Carry out your plan.

Carrying out your plan of the solution, *check each step*. Can you see clearly that the step is correct? Can you prove that it is correct?

LOOKING BACK

Fourth.
Examine the solution obtained.

Can you *check the result?* Can you check the argument?

Can you derive the result differently? Can you see it at a glance?

Can you use the result, or the method, for some other problem?

And here they are recast from http://web.media.mit.edu/~walter/MAS-A12/week1notes.html:

> Stage 1: Preparation. You go over the elements of the problem and study their relationships.
>
> Stage 2: Incubation. Unless you've been able to solve the problem quickly, you sleep on it. You may be frustrated at this stage because you haven't been able to find an answer and don't see how you're possibly going to.
>
> Stage 3: Inspiration. You feel a spark of excitement as a solution (or a possible path to one) suddenly appears.
>
> Stage 4: Verification. You check the solution to see if it really works.

Both the above are directed primarily at mathematical problems. Yet these attitudes to problem solving are of much wider application (see also the coda at the very end of this document). Here's my attempt at rewriting them from a more computational viewpoint:

> Stage 1: Understanding the problem
>
> > What are the data supplied? What are you trying to produce? Can you see how they are connected? What other data might you need to produce the output you require? Can you draw a diagram that connects what you have to what you want to produce? Can you annotate the diagram?
>
> Stage 2: Devise a plan
>
> > Have you seen a similar problem before?
> >
> > Does the problem break down into subtasks, and have you seen, or can you find solutions to these subtasks? Or solutions to related subtasks that might be re-used?
> >
> > Re-state the problem: break it down into a different set/sequence of subtasks. Is there a solution that works in parallel? Or is the problem inherently sequential (or some mixture of the two?). Draw diagrams of how the subtasks fit together.
> >
> > If you cannot make a plan at this point (because you really cannot see how to solve the problem) what can you do? Can you try out some ideas that solve little parts of the problem? What more (information, perhaps) do you need to be able to make a plan? Are you using all the information that's available to you?
>
> Stage 3: Carry out the plan
>
> > Design the implementation. You might use pseudocode, for example, perhaps on each subtask. Draw more diagrams, illustrating (as relevant) the flow of control, and/or the flow of data within the solution.
> >
> > Check that your proposed solution really does what you want it to do. Test it on both expected data and unexpected (but possible) data. Be prepared to go back to stage 2 (or even stage 1!) if it doesn't work quite as you thought it would.
> >
> > Code up functions, and/or objects and and/or methods, as appropriate for the implementation substrate. Test them as you go along: testing the components individually is much easier than delaying testing until later, and much more likely to result in systems that behave as you expect them to. Build the solution and test it some more. Test it on the "edge" cases.
>
> Stage 4: Looking back

Does the system behave as you meant it to? Test it more thoroughly: get other users to test it as well.

Would there have been a better solution that you can now see, because the process of developing a solution has made you understand the problem much better? Be prepared to go back to stage 2 and start again. Many badly designed systems have been deployed and are difficult to use, or not fit for purpose because of an unwillingness to tear the original, poor solution up and start again.

For lecture 2, a more directly computational approach seems appropriate:

We can introduce the pseudo-code ideas of

- if…then…else
- repeat…until…
- while … do …
- for (iteration)
- functional abstraction
- object concepts.

although one they will have met these in considering the Java programming they have already covered. But using them in pseudocode can still be informative.

Even something as simple as making a pot of tea?

```
Make tea ::=
        Boil kettle (see later)
        Put tea in teapot
        Pour kettle into teapot (see later)
        Repeat
                Wait
        Until tea is correct strength.

Boil kettle ::=
        Put water in kettle
        Switch kettle on
        Repeat
                Wait (a short time)
        Until kettle has boiled

Pour kettle into teapot ::=
        While teapot in not full
        do
                Pour water into teapot
```

Now let's consider making *N* cups of tea. Easy:

```
Make cups of tea(N) ::=
        Make tea
        For i = 1  to N
                Pour cup(i) of tea.
```

But what happens when *N* is too big? What happens when the teapot is empty, but there's still more cups of tea to be poured?

This sort of computational thinking is really about thinking clearly about what steps are involved in a particular task. We also note that we can avoid being caught up in the intricacies of some tricky part of the task by using functional abstraction - for example, we just wrote "Boil Kettle" at one point, and then later considered what this actually involves. This is a little like not getting stuck, as discussed in Pirsig's book. But real problems always have more complexity, as illustrated in making *N* cups of tea (and what might we do if *N* was 0?).

Consider some sort of data processing task.

> For example, student records (Each student has a name, registration number, list of classes they are taking, address, courses taken, etc.) We might want to find (e.g.) all students taking some specific class. And we might have a list of the courses available, with their prerequisites, and we might want to find the classes that a student has the prerequisites for. Or we might want to check that for a particular class that a student wants to take, the student has the appropriate prerequisites. One might put together a nice solution for this (not particularly easy, since there's issues about updating the list after exams, or for new students starting, or for students leaving, and in general we want to have some capabilities for dealing with exceptions as well) - but then in addition, the course list changes, and/or the prerequisites change, and whatever solution we might want to make for this needs to have the capability to be able to cope with all these expected changes.

But what about unexpected changes? Renumbering classes? Changing prerequisities? Bending rules?

Consider how the way in which we represent the problem influences how we might write the program. We want to be able to ensure that

- We could have as many students as we want
- We can introduce new classes
- We can introduce new prerequisites
- Students can leave, new students can arrive, etc.

In general, computational thinking has to consider both what needs to be done, and how the problem will be represented. Often these are not dissociable: they are strongly tied together, and the way in which the problem (or more precisely the data within the problem) will be represented influences what can be done with the data. It's important not to be too tied down to initial choices: again, a willingness to tear up what has been started, and start again when one realises that one's initial concept of a solution will not work (or will not work well, or will be clumsy, or will be difficult to modify, or …) is often important.

Note: there should be something about events as well: conceptually, events lead to parallel asynchronous processing. In older machines, this was implemented using a simple interrupt mechanism, that is, the "main" stream of processing was suspended while the event was dealt with, and (generally) some of the data structures in the main stream of processing were updated. But conceptually (and in reality in modern multicore or multiprocessor machines), the processing of both the "main stream" and the event(s) may be happening in parallel (as well as asynchronously), and this means that updating shared variables requires care to avoid unexpected results caused by parallel updating of the same entity.

Coda

This document is itself the solution (well, part of the solution) to the problem of giving two lectures of Computational Intelligence. To what extent did I follow my own techniques for solving problems?

> Stage 1: Understanding the problem.
>
> > What is CT? What do different sets of people think CT is? How might I go from the different views of CT to a set of lecture notes, or if lecture notes were not the

appropriate solution, to a point where I could stand up in front of the class of students in CSC9A1, and give two lectures. Understanding the problem also included reading more of the reference in both the books, and on the internet, and trying to understand where these agreed, and disagreed.

Stage 2: Devise a plan (and also incubation).

The issues had been incubating quite a long time before I started producing this document. The critical element of the plan was deciding how to divide up the two lectures: one on problem solving in general, and the other on a more computational view of problem solving.

Stage 3: Carry out the plan:

This involved writing initially an earlier version of this document, as well as extracting the critical aspects from the books and other materials that could be used in the lectures. There was a considerable amount of rearrangement of the material between earlier drafts of this document, and the current version of the document. The last part of carrying out the plan, is, of course, actually delivering the lectures, and that has not yet happened.

Stage 4: Looking back

Obviously, this implies post-lecture delivery. But this will certainly happen: in the light of my own experience of giving the lectures, in the light of any feedback from the students and perhaps colleagues as well. The material may be rearranged, revised, revamped, or completely deleted, depending on how matters actually run!

Collated references

The Wikipedia page, http://en.wikipedia.org/wiki/Computational_thinking,

The Apollo Research Institute page
http://apolloresearchinstitute.com/sites/default/files/future-work-skills-2020-computational-thinking-02-14-12.pdf

NSF (and others in the US) operational definition of Computational thinking for K12 education
http://www.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf

G. Polya, *How to solve it*, originally published 1945 by Princeton Press, and reprinted/republished by many other publishers

Robert Pirsig, *Zen and the art of motorcycle maintenance* 1974. Also republished by Vintage Press.

Z. Michalewicz & D.B. Fogel *How to Solve it: Modern Heuristics*, Springer 2000.