COMP11012: Overview

Sean Bechhofer, Uli Sattler, Andrea Schalk

January 2022

This narrative is based loosely on the final chapter of *The Power Of Computational Thinking: Games, Magic And Puzzles To Help You Become A Computational Thinker* by Paul Curzon and Peter Willaim Mcowan.

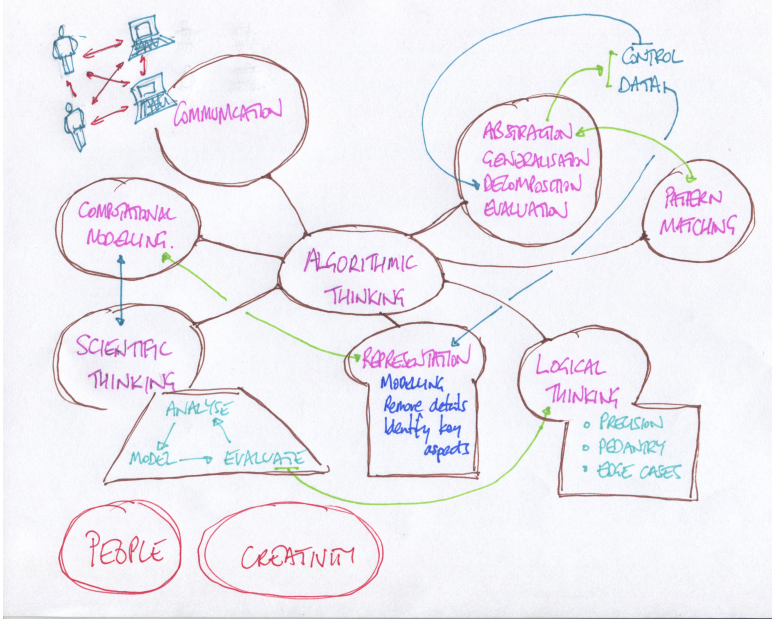

Figure 1: Mindmap of Concepts in COMP11012

Computational Thinking is a set of problem solving skills, based largely around the creation of algorithms.

During the course of the unit, we will go through a number of exercises and tasks that will be about trying to nurture these skills.

Algorithmic thinking: see the solution to problems as algorithms.

Writing an algorithm down allows us to

- check what it's doing
- perform it many times
- give it to another person
- give it to a machine

This is this all about **Communication**. How do we communicate our ideas to other people, to machines, or indeed communicate between machines?

An important thing here is **Computational Modelling**. We take something in the real world: weather, maps, tourists and try and create representations and algorithms that will simulate that thing. Then we can mimic its behavious, run experiments, reason about it.

Scientific Thinking is about approaching problems using a particular approach. We analyse the problems, and use the results of this to generate a model. This model is then evaluated or tested, with the results of that evaluation informing our analysis and model. We should not lose sight of that fact that we are operating on a model though, which will be an approximation of the real world phenomena we are interested in. **Evaluation** also requires us to engage in **Logical Thinking**: employing precision and pedantry in order to be very careful and precise about our details.

Pattern Matching is something we do all the time. It saves us time, allowing us to apply known solutions to a new problem. It will often involve a level of abstraction: we need to get past the details in order to determine that a known solution will work. So this is also closely linked to questions of representation.

Choosing appropriate **Representations** can make problems easier to solve: we'll see examples of this in the unit. A good representation will also make it much easier to code up a solution, once we get to the point of actually doing some programming.

Abstraction is something that we will discuss a lot in this unit, and it's a very important concept in Computer Science. One example of abstraction, which is very closely linked to the notion of **Decomposition** is to group lots of smaller instructions together into a bigger instruction that *hides* the detail of what's going on. This is sometimes referred to as **Control Abstraction**. **Data Abstraction** on the other hand tries to hide the details of how information or data is actually represented: we might not actually care how, for example numbers are represented. We just need to know that we can do it. **Abstraction** is also useful during **Evaluation**. When we are trying to understand how different algorithms performed, we can look at counting, for example, how many *set* operations or *comparisons* there were. Ignoring details make this an easier task – without compromising the results.

Generalisation takes a problem that we've solved and adapts it so that we can solve other problems. This may involve **Pattern Matching** and the use of a suitable **Representation**.

Decomposition involves breaking something down into smaller problems. It's related to the notion of **Control Abstraction** that we described above. **Pattern Matching** and **Abstraction** also play a part here. The idea of **Recursion** is a particular example of Decomposition, where the smaller problem is similar to the larger problem.

Evaluation is about checking that a solution is fit for purpose. Does it do the right thing: is it functionally correct? Does it do the right thing in a usable way: will it actually run in a sensible amount of time even if given large input? Testing our algorithms requires **Logical Thinking**: pedantry and precision. We can also evaluate through rigorous arguments. Can we reason about the performance or behaviour? We'll see that this isn't always possible in the general case. Such reasoning may take place on an **Abstraction** of the problem. **Decomposition** might allow us to evaluate sections of the solution independently.

Running through all this are two strands that you might not immediately think of as

relating to Computational Thinking: **Understanding People** and **Creativity**. Our solutions and algorithms need to be usable, so we need to understand how those solutions might be used and the problems that users will face. We need to be creative in coming up with solutions, and looking for the problems that need to be solved. But this requires the right environment and conditions: head space to work in, supportive colleagues and peers, and the willingness and opportunity to get things wrong.