# Outline of the course

We will look at the following topics

## Behaviour specification

We will look at the following questions

- How do we describe a problem to be solved? (specification)
- Given a program, exactly what problem does it solve? (documentation)
- Given a problem, what is a program to solve it? (design or derivation)
- Given a problem and a program, does the program solve the problem? (verification)

Why?

- Designing correct programs is difficult.
- It is more difficult when you don't have a clear idea of what the specification is.
- Programs that are not correct are expensive and dangerous.

(Note: The word "program" suggests "software". However the same considerations apply to any kind of engineered system. In this part of the course we will concentrate on so-called "transformational systems" which are systems that transform their input data into some output data. Later in the course we will look at "interactive systems" — systems where part of the output may be required prior to all of the input being available— and "reactive systems", which are interactive systems with time constraints. All these kinds of systems can be realized in either hardware or software.)

## Formal language theory and interactive systems

We will look at the following questions

- How can we mathematically model a state-based computer system?
- How does the finiteness of memory limit the problems systems can solve?
- Are there problems that computer systems can not solve?

By looking at state-based systems, we can expand our vision to include interactive and reactive systems.

Why?

- Interactive, reactive, and parallel systems are quite important.
- Given a problem, what model of computation is required to solve it? E.g. does your problem require a fixed or unlimited amount of memory.

We can apply these ideas to some interesting problems

- How can we formally describe languages (set of strings)
- How can we analyse strings using finite state systems

- How can we analyse strings using infinite state systems

## Efficiency (Computational Complexity)

We will look at the theory of “computational complexity”.

More efficient things do more with fewer resources.

In computing we are interested in solving bigger problems with less time (or space or sometimes something else).

We look at the following questions

- Given an algorithm, how efficient is it? (Algorithm complexity)
- Given a problem, what is the efficiency of the most efficient algorithm? (Problem complexity)
- How are problems linked together in terms of efficiency? (Complexity classes)

In this part of the course we will explore the idea of “complexity classes”. This will give us a way of talking about how fast (or space efficient) an algorithm is in a way that is sufficiently vague that what we can prove won’t be contradicted by minor changes to how the algorithm is implemented, or by running the algorithm on a faster computer.

Why?

- Before tackling a problem, it is a good idea to know how hard it will be.
- Sometimes our solutions to problems turn out to be

impractically inefficient. Is this because the problem was inherently hard, or because the solution is poor?