Computation

- **Motivation**

The course on ***Regular Languages*** answers:

"What is the power of ***deterministic finite automata*** as sequence recognizers?"

This course has the same general thrust:

"What sums can be done, what issues decided, by ***mechanical process***?"

"What is the power of ***algorithms***?"

Saying "*the power of algorithms*" is all very well, but it's very imprecise (like saying "*power of programs*" without specifying the programming language!) . . .

It turns out that the exact details of the definition of “algorithm” don't matter, within bounds that most people find reasonable . . . ***BUT***

- differences of efficiency (***unary*** vs. ***binary***)
- a single thread of control

Further, these “acceptable” definitions not only are equivalent, but their properties fit with our intuitive experience of the process of computation.

- self-embedding / self-reference (treat ***program*** as ***data***)
- existence of ***universal*** algorithms (***interpretation*** vs. ***compilation***)
- there are ***undecidable*** problems! (the ***Halting*** problem)

Examples of Models of Computation

- Turing machines
- register machines (Minsky/Conway)
- general recursive functions (Gödel/Church)
- λ-calculus (Church)
- Type 0 grammars (Chomsky)
- Markov algorithms

Note that restriction to ***fixed, finite memory*** is no longer a reasonable assumption – do you believe that there is an algorithm for ***integer multiplication*** ?

How much paper do you need?

Finite automata vs. Algorithms

	Finite state automaton	General algorithm
Logic	**FINITE**	**FINITE**
Backing store	**none**	**unbounded**
Action	**SEQUENTIAL**	**SEQUENTIAL**
Input specification	**via an alphabet of stimuli**	**on backing store, in given alphabet**
Output observation	**via an alphabet of responses**	**from backing store, in same alphabet**
Time of output	**at each time step**	**specific event on termination**
Causality *	**deterministic *or* non-deterministic**	**deterministic *or* non-deterministic**

*** deterministic**	the input data determines a unique computation
non-deterministic	many possible execution paths from one problem statement – ***favourable guess*** or ***parallel search***

Logic, Formal Systems and Automated Reasoning

In the latter part of the 19th century and the early part of the 20th there was a strong movement to automate the process of mathematical proof and mathematical discovery. The intention was to establish axiom systems for the different branches of mathematics, formalise the process of logical inference, and develop theorems on an essentially solid foundation. Ideally the inference process would be mechanical, thus gaining the benefit of the reliability of machines, as opposed to the human weaknesses of fallacy in logic and error in calculation (the ***Analytical Engine*** of Charles Babbage had already been designed).

Leading names in this activity were Frege, Whitehead & Russell, and Hilbert. In 1910 David Hilbert proposed 23 problems as a challenge to succeeding generations of mathematicians. We shall meet the ***Tenth*** at the end of this course.

Gödel's Incompleteness Theorem

In the 1930s the move towards the axiomatisation of mathematics got a nasty shock with the publication of Kurt Gödel's Incompleteness Theorem. What Gödel showed was that no consistent finite set of axioms could be sufficient to establish all of the intuitive truths of whole number arithmetic. The idea of the proof is deceptively simple; the set of axioms, the rules of inference, chains of inference and so on, are assigned unique natural numbers (***Gödel numbers***) in a consistent way. For each statement that can be expressed within the formal system, there is a corresponding polynomial that has a solution if and only if the statement is true. Consider the statement "***this statement cannot be proved within the formal system***". The statement **cannot** be proved in a consistent system, but it is evidently true.

No solution of the corresponding polynomial can be found within the system!

Church's λ-calculus and the Recursive Functions

At the same time that Gödel was proving the Incompleteness Theorem in Vienna, Alonzo Church at Princeton together with his students (notably Stephen Kleene and Barkley Rosser) were investigating the precise definition of the process of calculation. They introduced the ***λ-calculus***, asserting that it captured the natural intuition. In his proof Gödel used ***recursive functions*** to construct the numbering of statements in a formal system, and the team at Princeton were able to show the consistency of the λ-calculus, and that the two definitions had equivalent power.

Church proposed in 1936 as a working hypothesis that ***A function of positive integers is effectively calculable (if and) only if recursive***. This has become known as ***Church's thesis***. For a full discussion see Church-Turing Thesis in Wikipedia

or http://www.seop.leeds.ac.uk/archives/fall2002/entries/church-turing/

Turing machines and human computation

At the same time, Alan Turing in Cambridge (King's) was tackling essentially the same problem from another angle. What Turing did was to analyse the actions of a human being when calculating, and to define an idealized computer, the ***Turing machine***. ***Turing's Thesis*** is that ***whenever there is an effective method for obtaining the values of a mathematical function, the function can be computed by a Turing machine***. In ***Proc LMS***, series 2, 42 (1936-37), 230-265, he obtained the same essential results as Church had done in his slightly earlier paper, indeed the final few pages 263ff. establish the formal equivalence of the two definitions.

Church in his review of Turing's paper said that ***computability by a Turing machine ... has the advantage of making the identification with effectiveness in the ordinary ... sense evident immediately***. We first investigate Turing's definition.

Turing machines with higher dimensional data storage

Turing's motivation was that of human calculation, and his starting point was that of a mathematician writing symbols at formal positions on a piece of paper. He investigated the power of machines that had a 2-dimensional or higher data array instead of a linear tape. On the analogy with a piece of paper, the action would be the same as for the linear machine that we are going to define, except that the movement of the head on the paper $D(q_{i,}\ s_j)$ could be $\Uparrow$ or $\Downarrow$ in addition to L or R.

Turing was able to show that the extra dimension did not lead to greater computing power, under reasonable assumptions about what it meant to extract the answer to a calculation from a finite array of characters on an n-dimensional grid.

Turing machines with more than one linear tape

Often calculations require more than one argument, and one way to handle this is to have a machine with more than one tape. It may also be convenient to use different tapes for different purposes during a calculation. At any stage of the calculation there is a current tape, as well as a current symbol for each tape. Change of state and change of symbol are treated in the usual way, but the *movement* can also be to resume execution with a change of current tape.

Turing was able to show that a multi-tape machine can be simulated by a machine having a single tape.

[It is however possible that using a machine with more than one tape can lead to a more efficient solution. There is a graph traversal algorithm for which the space complexity is reduced from K to $\log K$ by the introduction of an auxiliary tape – such issues are the business of the **Complexity** course.]

Turing machines

A ***Turing machine*** (T.m.) is a Deterministic Finite Automaton that has R/W access to an unbounded infinite tape, on each square of which the characters of a finite alphabet $\boldsymbol{S}$ can be written. The set of states $\boldsymbol{Q}$ contains an ***initial*** state $\boldsymbol{q}_0$, and the alphabet $\boldsymbol{S}$ a preferred character $\boldsymbol{s}_0$, the ***blank*** or ***filler*** symbol.

At the start of a computation all but a finite number of squares of the tape are blank. The DFA is placed on a particular square in state $\boldsymbol{q}_0$, and that square of the tape becomes current. The machine will subsequently undergo transitions which depend only on the current state and the symbol on the current square.

The transitions are a ***graphic response***, which replaces the symbol on the current square, a ***change of state***, and a ***movement*** of the head on the tape, ***either*** to the left (L), ***or*** to the right (R).

We represent the transitions as follows:

1. a graphic response, $R(t+1) = F(Q(t), S(t))$;

2. a change of state, $Q(t+1) = G(Q(t), S(t))$;

3. a head movement, $D(t+1) = D(Q(t), S(t))$

where we use L=0 to represent movement of the head to the ***left*** on the tape, and R=1 to represent movement to the ***right***.

We can handle termination in a number of ways; ***either*** there is a particular state that HALTs the machine, ***or*** we may just write

$$Q(t+1) = G(Q(t), S(t)) = \text{HALT}.$$

If the computation HALTs we may replace the symbol on the current square before HALTing; the result is the final state of the tape. If it does **not** HALT, the result is ***undefined***.

Specifying Turing machine logic

A Turing machine will be defined by giving the transitions for each (state, symbol) pair: thus $(q, s, G(q,s), F(q,s), D(q,s))$. Such an entry is often written $q_i \;\; s_j \;\; q_{ij} \;\; s_{ij} \;\; d_{ij}$, and is called a ***quintuplet***. The set of all such entries for $q_i \in Q$ and $s_j \in S$ is the ***full quintuplet description*** of the Turing machine.

For HALTing combinations

either	write	$q_i \;\; s_j$ HALT $[s_{ij}]$ –
or	omit	the quintuplet for the pair (q_i, s_j).

Conventions

- numbers will often be represented in unary, i.e. 0 will be represented by the null string, and the natural number n by 1111 ... 1 (n times).
- blank or 0 will be used as the filler; Y, X as replacement symbols for 0, 1; A, B as argument delimiters.

Action of a Turing machine

When designing a Turing machine for a specific problem we shall almost always use searching states (the only exception is the reduction via a 2-register machine to a 2-state Turing machine).

A typical transition may therefore be written:

$$((q_i, d_i),\ \ s_j) \ \rightarrow \ ((q_{ij}, d_{ij}),\ \ s_{ij})$$

Let's suppose that the machine has just entered state (q_i, d_i), on a square of the tape. This state is *searching*, in direction d_i . The action is:

1) ***move*** to the next square in direction d_i ;

2) ***replace*** the symbol found, s_j , by s_{ij} ;

3) ***enter*** the new state, (q_{ij}, d_{ij}) ;

4) ***repeat from* 1)** with state (q_{ij}, d_{ij}) .