# CA320 - Computability & Complexity

## Introduction

Dr. David Sinclair





## Overview

In this module we are going to answer 2 important questions:

- Can all problems be solved by a computer?
- What problems be efficiently solved by a computer?

**Computability** The study of computable functions. In other words, the study of problems that are computable and hence have an algorithm.

**Complexity** The classification of computable problems by their inherent difficulty.

# Overview (2)

- Introduction
  - Sets, functions and languages
- Introduction of Haskell
- Regular languages and finite automata
  - Regular grammars, regular expressions and finite state automata
- Context-free languages and pushdown automata
  - Context-free grammars, derivations, ambiguity and pushdown automata
- Context-sensitive languages and linear bounded automata
  - Context-sensitive grammars and linear bounded automata
- Models of Computation
  - Turing machines, partial recursive functions, lambda calculus, Church-Turing thesis
- Computability
  - Halting problem and reducibility
- Complexity
  - Asymptotic notation, time complexity and space complexity





# Texts

Recommended:

- *Introduction to the Theory of Computation*, Sipser, PWS, ISBN 053494728X, 1996
- *Haskell: The Craft of Functional Programming*, Thompson, Addison-Wesley, ISBN 0-201-34275-8, 1999

Supplementary:

- *Elements of the Theory of Computation*, Lewis and Papadimitriou, Prentice Hall, ISBM 0-13-272741-2, 1998
- *Introduction to Languages and the Theory of Computation*, Martin, McGraw-Hill, ISBN 0-07-115468-X, 1997
- *Programming in Haskell*, Hutton, Cambridge University Press, ISBN 9780521692694, 2007

# Contact Details

**Lecturer:** Dr. David Sinclair
**Office:** L253
**Phone:** 5510
**Email:** david.sinclair@dcu.ie
**WWW:** http://www.computing.dcu.ie/~davids
**Course web page:**
http://www.computing.dcu.ie/~davids/CA320/CA320.html





# How do I successfully complete this module?

The module mark is a straight weighted average of:

- 25% continuous assessment
  - 2 assignments
    - first assignment (10%)
    - second assignment (15%)
- 75% end-of-semester examination
  - 5 questions. Do 4 questions.
  - 2 sections, at least 1 from section A
    - Section A: Haskell
    - Section B: Computability & Complexity

# What if I don't successfully complete this module?

If you fail the module in the end-of-semester exams then you will need to repeat some elements of the assessment.

- If you just failed the exam you can resit the exam in the Autumn.
- If you just failed the continuous assessment then you must complete a resit assignment.
  - Contact me after the results are published for the resit assignment.
- If you failed both the exam and the continuous assessment, you must repeat both.

If you fail the module after the resit examination and continuous assessment you must repeat all aspects of the module in the following year.





# Some Mathematical Revision: Logic

| Connective | Symbol | Use | English equivalent |
|---|---|---|---|
| conjunction | $\wedge$ | $p \wedge q$ | $p$ and $q$ |
| disjunction | $\vee$ | $p \vee q$ | $p$ or $q$ |
| negation | $\neg$ | $\neg p$ | not $p$ |
| conditional | $\rightarrow$ | $p \rightarrow q$ | if $p$ then $q$ |
| | | | $p$ only if $q$ |
| biconditional | $\leftrightarrow$ | $p \leftrightarrow q$ | $p$ if and only if $q$ |

| $p$ | $q$ | $p \wedge q$ | $p \vee q$ | $p \rightarrow q$ | $p \leftrightarrow q$ |
|---|---|---|---|---|---|
| T | T | T | T | T | T |
| T | F | F | T | F | F |
| F | T | F | T | T | F |
| F | F | F | F | T | T |

# Logic (2)

A *compound proposition* is a proposition constructed from an arbitrary combination of the 5 basic operations.

$$(p \vee q) \wedge \neg(p \rightarrow q)$$

| $p$ | $q$ | $p \vee q$ | $p \rightarrow q$ | $\neg(p \rightarrow q)$ | $(p \vee q) \wedge \neg(p \rightarrow q)$ |
|---|---|---|---|---|---|
| T | T | T | T | F | F |
| T | F | T | F | T | T |
| F | T | T | T | F | F |
| F | F | F | T | F | F |

A *tautology* is a compound proposition that is **true** for every possible truth value combination of its constituent propositions.

A *contradiction* is a compound proposition that is **false** for every possible truth value combination of its constituent propositions.





# Logic (3)

Two propositions $P$ and $Q$ are *logically equivalent*, $P \Leftrightarrow Q$, if they have the same truth value for every possible combination of base propositions. Hence, in any expression where $P$ is used we can substitute $Q$ and the entire expression remains unchanged.

A proposition $P$ *logically implies* a proposition $Q$, $P \Rightarrow Q$, if in every case $P$ is true then $Q$ is also true.

Beware of the subtle difference between $P \rightarrow Q$ and $P \Rightarrow Q$!
$P \rightarrow Q$ is a proposition, just like $P$ and $Q$, whereas $P \Rightarrow Q$, is a *meta-statement*. It is an assertion about the relationship between the propositions $P$ and $Q$. If $P \Rightarrow Q$, then $P \rightarrow Q$ is a *tautology*.

# Logic (4)

There is a large set of logical identities that we can use when manipulating compound propositions. Some of the more useful ones are:

| | |
|---|---|
| commutative laws | $p \vee q \Leftrightarrow q \vee p$<br>$p \wedge q \Leftrightarrow q \wedge p$ |
| associative laws | $p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$<br>$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$ |
| distributive laws | $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$<br>$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$ |
| De Morgan's laws | $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$<br>$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$ |
| | $p \rightarrow q \Leftrightarrow \neg p \vee q$<br>$p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$ |
| contrapositive | $p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$ |





# Logic (5)

**Quantifiers**

$\forall x(P(x))$ states that the proposition $P$, which depends on the truth value of $x$, is true for all values of $x$.
$\exists x(P(x))$states that the proposition $P$, which depends on the truth value of $x$, is true for some value of $x$.

Quantifiers can combined in the same expression but great care is needed. The following 2 expressions which are very similar mean 2 different things (in fact the second is not true if $x$ and $y$ are from the domain of natural numbers, $\mathcal{N}$).

$$\forall x(\exists y(x < y))$$
$$\exists y(\forall x(x < y))$$

The logical identifiers for quantifiers are:

$$\forall x(P(x)) \Leftrightarrow \neg(\exists x(\neg P(x)))$$
$$\exists x(P(x)) \Leftrightarrow \neg(\forall x(\neg P(x)))$$

# Sets

Sets are unordered collections of distinct elements. A set can be described by listing its elements.
$A = \{1, 2, 4, 8\}$
In the case of large (and infinite) sets we can use ellipses (...)
$B = \{2, 4, 6, 8, \ldots, 100\}$
$C = \{0, 5, 10, 15, 20, 25, \ldots\}$
However, a much nicer way of specifying a set is to use a property that all the elements satisfy.
$C = \{x | x = 5i, i \in \mathcal{N}\}$

For any set $A$, $x \in A$ means that $x$ is an element of $A$. $x \notin A$ means that $x$ is not an element of $A$. $A \subseteq B$ means that every element of $A$ is also an element of $B$. $A \nsubseteq B$ means that at least one element of $A$ is not and element of $B$.
The *empty set*, the set with no elements, is denoted $\emptyset$.





# Sets (2)

Given 2 sets $A$ and $B$, we can define their *union*, $A \cup B$, their *intersection*, $A \cap B$ and *difference*, $A - B$, as:
$A \cup B = \{x | x \in A \vee x \in B\}$
$A \cap B = \{x | x \in A \wedge x \in B\}$
$A - B = \{x | x \in A \wedge x \notin B\}$

If $A \cap B = \emptyset$ then $A$ and $B$ are *disjoint*.
A collection of sets are *pairwise disjoint* if distinct pairs of sets $A$ and $B$ from the collection are disjoint.

# Sets (3)

The *complement* of a set $A$, denoted $A'$ is everything not in $A$. If $U$ is the universal set then

$$A' = \{x | x \in U \wedge x \notin A\}$$

The De Morgan laws for sets are:

$$(A \cup B)' = A' \cap B'$$
$$(A \cap B)' = A' \cup B'$$

The *Cartesian product* $A \times B$ of two sets $A$ and $B$ is:

$$A \times B = \{(a, b) | a \in A \wedge b \in B\}$$

The elements of $A \times B$ are called *ordered pairs* because $(a, b) = (c, d)$ only if $a = c$ and $b = d$.





# Relations and Functions

A *relation* $R$ on two sets $A$ and $B$ is a set of ordered pairs, $A \times B$, where $A$ is the *domain* of $R$ and $B$ is the *codomain* of $R$.
If $x \in A$ and $y \in B$, then $xRy$ is true if $(x, y) \in R$.

A *function* is a special kind of relationship in which an element of the domain to related to just one element of the codomain.

A function $f : A \to B$ relates an element $x \in A$ to an element $y \in B$ where $y = f(x)$. If $f(x)$ is defined for all $x \in A$ then the function is said to be *total*. If $f(x)$ is not defined for some $x \in A$ then the function is said to be *partial*. The set of values $f(x)$ is called the *range*, which is a subset of the codomain.

# Relations and Functions (2)

A function $f : A \to B$ is said to be *one-to-one* if it never assigns the same value to two different elements of its domain. A function $f : A \to B$ is said to be *onto* if it its range is the same as its codomain. A function that is *one-to-one* and *onto* is called a *bijection*.

We will be interested in a special kind of relation, called an *equivalence relation*. A relation $R$ on a set $A$ is an *equivalence relation* if it satisfies the following conditions:

1. $R$ is *reflexive*, i.e. $\forall x \in A, xRx$
2. $R$ is *symmetric*, i.e. $\forall x, y \in A$, if $xRy$, then $yRx$
3. $R$ is *transitive*, i.e. $\forall x, y, z \in A$, if $xRy$ and $yRz$, then $xRz$

For an equivalence relation $R$ on a set $A$, and an element $x \in A$, the *equivalent class containing* $x$ is

$$[x]_R = \{y \in A | yRx\}$$





# Proofs

During this module we will use a few different proof techniques. All proofs use reasoning based on logic to derive some statement from initial facts, assumptions, hypotheses or statements that have been previously proven.

**Proof by Contrapostive**
Remember that $p \to q \equiv \neg q \to \neg p$.

Example: $\forall i, j$ and $n$, if $ij = n$ then $i \leq \sqrt{n}$ or $j \leq \sqrt{n}$.

Rather than trying to prove that $ij = n \to (i \leq \sqrt{n} \vee j \leq \sqrt{n})$ we will prove $\neg(i \leq \sqrt{n} \vee j \leq \sqrt{n}) \to ij \neq n$.

$\neg(i \leq \sqrt{n} \vee j \leq \sqrt{n}) \equiv \neg(i \leq \sqrt{n}) \wedge \neg(j \leq \sqrt{n})$ by De Morgans law
$\equiv (i > \sqrt{n}) \wedge (j > \sqrt{n})$
Therefore $ij > \sqrt{n}\sqrt{n}$
$\equiv ij > n \equiv ij \neq n$

# Proofs (2)

**Proof by Contradiction**

A variant of *proof by contrapositive* is *proof by contradiction*. Every proposition $p$ is equivalent to the proposition $true \rightarrow p$. Its contrapositive is $\neg p \rightarrow false$. *Proof by contradiction* works by assuming $p$ is false and deriving the statement *false* (i.e. deriving its contradiction).

Example: $\forall m, n \in \mathcal{N},\ m/n \neq \sqrt{2}$.

Assume the contradiction of the proposition, i.e. $\exists m, n \in \mathcal{N}$ such that $m/n = \sqrt{2}$.
Dividing out the common factors of $m$ and $n$ yields $p/q = \sqrt{2}$ where $p$ and $q$ have no common factors.
$p/q = \sqrt{2} \equiv p = q\sqrt{2} \equiv p^2 = 2q^2$.
Since $p^2$ is even, then $p$ must be even (aside: can you prove that the product of two even numbers must be even?), so $p = 2r, r \in \mathcal{N}$





# Proofs (3)

Since $p = 2r$ then $p^2 = 4r^2$ and
$p^2 = 2q^2 \equiv 4r^2 = 2q^2 \equiv 2r^2 = q^2$.

Hence $q^2$ and $q$ are even.If $p$ and $q$ are even they must have a common factor, 2, but based on the contradiction of the proposition we derived that $p$ and $q$ had no common factors. The contradiction of the proposition has given rise to a *false* statement, and hence the proposition $\forall m, n \in \mathcal{N},\ m/n \neq \sqrt{2}$ is *true*.

**Proof by Cases**

If we want to prove $P$ and $P_1, P_2, \ldots, P_n$ are propositions of which at least one must be *true* then we can prove $P$ by proving that $P_i \rightarrow P, \forall i$.

$$
\begin{aligned}
(P_1 \rightarrow P) \wedge (P_2 \rightarrow P) \wedge \ldots (P_n \rightarrow P) &\Leftrightarrow (P_1 \vee P_2 \vee \ldots \vee P_n) \rightarrow P \\
&\Leftrightarrow true \rightarrow P \\
&\Leftrightarrow P
\end{aligned}
$$

# Proofs (4)

**Proof by Structural Induction**

When dealing with recursive definitions *proof by structural induction* is a common approach. Given an object $O$ which comprises base elements, $a_1, a_2, \ldots, a_n$, and operations $o_1, o_2, \ldots, o_m$ such that

- $a_i \in O, 1 \leq i \leq n$
- $\forall x_1, x_2, \ldots, x_p \in O, op_j(x_1, x_2, \ldots, x_p) \in O, 1 \leq j \leq m$

the principle of *structural induction* states that to prove $P(x)$ is *true* for every $x$, it is sufficient to prove

- $P(a_i)$ is *true*, $1 \leq i \leq n$ (*basis statement*)
- $\forall x_1, x_2, \ldots, x_p \in O$, if $P(x_j)$ is *true*, $1 \leq k \leq p$ then $P(op_j(x_1, x_2, \ldots, x_p)$ is *true*)(*induction step*)





# Proofs (5)

Example:

The language *Expr* is defined as:

- $a \in Expr$
- $\forall x, y \in Expr, x + y, x * y \in Expr$

Prove every element of *Expr* has odd length.

Using *structural induction* this translates into

- $|a|$ is odd
- $\forall x, y \in Expr$ if $|x|$ and $|y|$ are odd, then $|x + y|$ and $|x * y|$ are odd

A number $n$ is odd if $\exists j \in \mathcal{Z}$ such that $n = 2j + 1$

*Base step*: $|a|$ is odd since $|a| = 1$.

*Induction step*: Assuming $|x| = 2g + 1$ and $|y| = 2h + 1$ (*induction hypothesis*, $|x|$ and $|y|$ are odd) then $|x + y|$ and $|x * y|$ are $(2g + 1) + 1 + (2h + 1) = 2(g + h + 1) + 1 = 2k + 1$ are odd, where $k = g + h + 1$.

# Languages

This module addresses the 2 central questions in this module, namely,

1. What problems are computable?
2. Which problems have efficient algorithmic solution?

To do this we will need:

1. A way to describe problems without specifying how they are solved.
2. An abstract model of a "computer" that is independent of any technological implementation.

The most fundamental type of problem is a *decision problem*, a problem to which the answer is either **yes** or **no**.
Other types of problems, such as calculating a function or manipulating a data structure, have a *decision problem* at their core. So how do we represent a *decision problem* without specifying how the problem is solved?





# Languages (2)

For any given *decision problem*,$\Pi$, we can partition the set of inputs into 2 sets:

1. Input values that result in the answer **yes**, $Y_\Pi$.
2. Input values that result in the answer **no**, $N_\Pi$.

Let's focus on the set of input values that result in the answer **yes**, $Y_\Pi$. We can think of each element of $Y_\Pi$ as a word in a language, $L_\Pi$, and the complete set $Y_\Pi$ as the complete language, $L_\Pi$.

A machine that then determines if a word belongs to the language $L_\Pi$ is then said to *accept* the language and hence solve the related decision problem, $\Pi$.