FIT2014 Theory of Computation
Lectures 1–5
Languages, logic, proofs

Graham Farr
Faculty of Information Technology
Monash University

July 2023

Monash University
Faculty of Information Technology

Preliminaries

0.1 Why study the Theory of Computation?

"There is nothing so practical as a good theory." (Kurt Lewin, 1943)

If you are studying computer science, then — as for any other science — it is important to understand the fundamental principles of the field.

Studying the Theory of Computation will enable you to:

- understand properly the power and limits of computation;
- identify whether a problem is tractable or intractable;
- understand why a particular problem seems hard: is it because of the limitations of your computer, or because of some intrinsic feature of the problem?
- identify problems from different fields that have the same underlying structure.

A recurring theme in this subject is the use of abstract models. There is a common misconception that if something is "abstract" then it is far from reality and not very useful. In fact, proper abstraction leads to models and methods that are far more widely applicable than they could otherwise have been.

0.2 Some applications

The Theory of Computation is useful in any domain where computation is done. It helps us to design algorithms and to analyse and classify them in terms of their performance and their usage of resources. It helps us make computational links between different-looking problems, so we can use methods developed for one problem on other problems.

Here is a tiny sample of the domains where the theory of computation is used.

- pattern recognition (in text, DNA, proteins, financial data, ...)
- modelling of natural languages
- compilers and interpreters for programming languages
- network design and analysis

- information security
- communications: codes, protocols
- verification of complex systems

0.3 Mathematics

This subject makes extensive use of mathematics, since mathematics is the language of abstract modelling and logical reasoning. It uses the mathematics covered in the prerequisite units. We won't summarise that material here, but we mention just a few points that may be helpful to computer science students using mathematics.

When defining a function in mathematics, we start by writing something like $f : A \to B$, where A is the domain and B is the codomain. We can think of this as *declaring* the function, because we are naming the function and announcing the types of things it works with. Our declaration $f : A \to B$ says that the function f can take any member of the domain A as its argument, and the value returned by f must be in B (though not everything in B has to be a possible return value). So, it's a lot like *declaring* something when writing a program. Having declared our function, we complete the definition of the function by specifying what value we get for each possible argument. For example, a function f from integers to integers that gives the square of each integer might be defined by writing $f : \mathbb{Z} \to \mathbb{Z},\ x \mapsto x^2$. The last part here, $x \mapsto x^2$, tells us what value the function produces for a given argument x. It is conventional to use the normal arrow $\to$ between the domain and the codomain and to use $\mapsto$ when specifying what each element of the domain is transformed to by the function.

Note that, in mathematics, a function only specifies <u>what</u> value is returned for each possible value of its argument. It does not say <u>how to compute</u> that returned value. In some cases, the definition of the function includes a mathematical description that is easily turned into an algorithm. For example, in our squaring function above, the algebraic specification $x \mapsto x^2$ is easily turned into a simple algorithm. But, in other cases, functions can be defined mathematically without giving much idea of how to compute them, and there are even some perfectly well-defined mathematical functions which cannot be computed at all, as we will see later in this unit.

In this respect, functions in mathematics are different to functions in programming. Many programming languages have functions, and writing a function in a program involves writing code to compute the function. A function in programming may be thought of as a mathematical function plus some code for computing the function.

Lecture 1

Languages

One of the themes of this unit is the study of *formal languages*. This means sets of strings of symbols that satisfy some formal rules. They are used to model some aspects of natural languages, such as the following (which include many of the native languages of students of this unit):

- English, Australian, Woiwurrung, Mandarin, Auslan, . . . , Malayalam, Japanese, Scottish Gaelic, Irish Gaelic, Javanese, Cantonese, Punjabi, Bengali, Singhala, Marathi, Hebrew, Indonesian, Kannada, Hokkien, Croatian, Romanian, Russian, Tamil, Klingon, Spanish, Turkish, Tagalog, Vietnamese, French, Hungarian, Swedish, Elvish, Esperanto, Greek, Latin, Korean, Urdu, Konkani, Singlish, Dutch, Ukrainian, Persian, Malay, German, American Sign Language (ASL), Hindi, . . .

An even more important motivation for us, in computer science, is the use of formal language theory to define programming languages and to write compilers and interpreters for them. The following list of programming languages includes many suggested by students of this unit.[1]

- Python, Java, Haskell, awk, C, MIPS Assembler, Smalltalk, Prolog, Simula67, Interprogram, Algol-60, Cobol, Fortran, . . . , Rust, C++, MATLAB, Racket, R, Lisp, UwU, Go, Lua, C#, APL, Malbolge, Scratch, Verilog, Phi, Julia, Rockstar, GLSL, Scratch, . . .

There are many other areas where information is recorded or communicated in a way that can be thought of as a language. For example, each of the following involves one or more languages:

- mathematics, music, knitting, games, . . .

The study of formal languages will give you tools that can be used to work with all these different kinds of languages.

In this chapter, we define languages formally, beginning with their basic components.

[1]Puzzles for fun: try and put them in the order in which they were invented; try and arrange them in a family tree (or forest); find the one invented in Australia.

1.1 Alphabets

At its lowest level, computation works with **symbols**, also called **letters** or **characters**. So we need to specify what symbols are allowed.

An **alphabet** is a finite set of symbols.

For example:

- the English alphabet:
  $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$
- $\{\mathtt{a}, \mathtt{b}\}$
- the set of ten decimal digits: $\{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$
- the set of two bits: $\{\mathtt{0}, \mathtt{1}\}$
- the Korean alphabet:
  {ㄱ,ㄲ,ㄴ,ㄷ,ㄸ,ㄹ,ㅁ,ㅂ,ㅃ,ㅅ,ㅆ,ㅇ,ㅈ,ㅉ,ㅊ,ㅋ,ㅌ,ㅍ,ㅎ, ㅏ, ㅐ, ㅑ, ㅒ, ㅓ, ㅔ, ㅕ, ㅖ,ㅗ,
  ㅘ,ㅙ,ㅚ,ㅛ,ㅜ,ㅝ,ㅞ,ㅟ,ㅠ,ㅡ,ㅢ, ㅣ}
- the set of the four symbols that represent the four suits used in card games:
  {♠, ♣, ♦, ♥}

We often use Σ to denote an alphabet. In this unit, we will usually use $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

1.2 Words

A **word** is a finite string of symbols.

A **word over alphabet** Σ is a finite string of symbols all taken from Σ.

The **empty word** is the word of length 0, denoted by ε (or sometimes Λ).

Examples:

- some words over the English alphabet: `book`, `zhongguo`, `xjktzzqmbf`.
  - Only the first of these is a valid *word in the English language*. But they are all finite strings of letters from the English alphabet, so they are valid *words over the English alphabet*.
- Some words over the alphabet $\{\mathtt{a}, \mathtt{b}\}$: ε, `a`, `b`, `aa`, `ab`, …, `abba`, `baba`, …

Some standard notation for repetition of letters and words:

$$\mathtt{a}^2 = \mathtt{aa}, \qquad \mathtt{ab}^3 = \mathtt{abbb}, \qquad (\mathtt{ab})^3 = \mathtt{ababab}$$

If x is a word, then x^k is the string obtained by concatenating k copies of x together:

$$x^k = \underbrace{xxx \cdots\cdots xx}_{k \text{ copies of } x}.$$

$(\mathtt{baa})^0 = \ldots ?$

Question: How do you denote the empty word over the Greek alphabet?

1.3 Languages

A **language over alphabet** Σ is a set of words over Σ.

This is much broader than the usual way in which the term "language" is used. Normally, when we use the term "language", we mean more than just an arbitrary set of words; we would envisage a set of rules for combining the words, and perhaps some meanings for the words and for combinations of them. The term "language" as defined above was inspired by the more traditional use of the term, but is deliberately broader.

Some simple languages have special names and notation.

- **empty language**: $\emptyset = \{\,\}$
  - not to be confused with the language containing just the *empty word*: $\{\varepsilon\}$
- $\Sigma^k := \{$ all words over Σ of length $k\,\}$
  - E.g.: $\{\mathtt{a},\mathtt{b}\}^2 = \{\mathtt{aa},\mathtt{ab},\mathtt{ba},\mathtt{bb}\}$
- **universal language**: $\Sigma^* := \{$ all finite words over $\Sigma\,\}$

Here are some basic languages that will be used frequently in examples and exercises in this unit.

$$\begin{aligned}\text{EVEN-EVEN} &:= \{\text{all strings in which } \textit{each} \text{ of } \mathtt{a},\mathtt{b} \text{ occurs an } \underline{\text{even}} \text{ number of times}\}\\ &= \{\varepsilon, \mathtt{aa}, \mathtt{bb}, \mathtt{aaaa}, \mathtt{aabb}, \mathtt{abab}, \mathtt{abba}, \mathtt{baab}, \ldots\}\end{aligned}$$

Remember, 0 is even!

$$\begin{aligned}\text{DOUBLEWORD} &:= \{xx : x \in \Sigma^*\}\\ &= \{\text{all strings obtained by concatenating some string with itself}\}\\ &= \{\varepsilon, \mathtt{aa}, \mathtt{bb}, \mathtt{aaaa}, , \mathtt{abab}, \mathtt{baba}, \mathtt{bbbb}, \mathtt{aaaaaa}, \mathtt{aabaab}, \ldots\}\end{aligned}$$

Note, $\varepsilon\varepsilon = \varepsilon$.

$$\begin{aligned}\text{PALINDROMES} &:= \{\text{all strings that are the same forwards and backwards}\}\\ &= \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{bb}, \mathtt{aaa}, \mathtt{aba}, \mathtt{bab}, \mathtt{bbb}, \mathtt{aaaa}, , \mathtt{abba}, \mathtt{baab}, \ldots\}\end{aligned}$$

$$\begin{aligned}\text{HALF-AND-HALF} &:= \{\mathtt{a}^n\mathtt{b}^n : n \in \mathbb{N} \cup \{0\}\}\\ &= \{\varepsilon, \mathtt{ab}, \mathtt{aabb}, \mathtt{aaabbb}, \ldots\}\end{aligned}$$

1.4 Definitions, Theorems, Proofs

A **definition** specifies the precise meaning of something.

We will use bold type for the term being defined.

A **theorem** is a mathematical statement that has been proved to be true.

You may also come across the terms “proposition”, “lemma” and “corollary”, although we don’t use them in this unit. They are also theorems, but the different terms tell us something about how the theorem is used. A **lemma** is a theorem whose sole purpose is to be used in the proof of a more significant theorem later.[2][3] A **proposition** is a theorem that is (unlike lemmas) of interest in its own right, but the term is typically used for theorems that are easy to prove and are of less significance than other theorems being proved in the same article/chapter/book. We also use the term “proposition” in a more specific sense from the next lecture onwards. A **corollary** is a theorem that follows almost immediately from another theorem that has just been stated.

A **proof** is a step-by-step argument that establishes, logically and with certainty, that something is true.

A proof must be verifiable, so it must be written clearly. It must be able to be read sequentially, with each step depending only on what comes before it; it should not be necessary to read ahead to determine if a step is correct (although it’s ok to read ahead to help your understanding).

We announce the beginning of a proof with the heading **Proof.** The end-of-proof symbol $\square$ indicates the end of the proof.[4]

We now give some examples of theorems and proofs.

Theorem 1
English has a palindrome.

Proof. ‘rotator’ is an English word and also a palindrome. $\square$

An **existential** statement is a statement that asserts that something with a specified property exists. It may or may not be true.

The above theorem is an existential statement. Since it’s a theorem, it is true.

Proving an *existential statement*, such as ...

There exists a palindrome in English

...just requires one suitable example.

[2]Lemmas can tend to be highly technical and are often forgotten even in cases where the theorem they help prove becomes famous. But some lemmas have found fame in their own right, e.g., the Handshaking Lemma, Burnside’s Lemma.

[3]These days, the plural of “lemma” is “lemmas”, as you would expect. But, traditionally, the plural was “lemmata”, which you may encounter in old papers. Are there other English words where the plural ending is -ta?

[4]Another traditional way to indicate the end of a proof is using the acronym “QED”, which stands for the Latin phrase “quod erat demonstrandum”, meaning “which was to be proved”. Occasionally, the end of a proof is indicated by //.

Most proofs are not this short . . .

Theorem 2
Every English word has a vowel or a 'y'.

Proof.
'aardvark' has a vowel.
'aardwolf' has a vowel.
'aasvogel' has a vowel.
. . .
. . .
'syzygy' has a 'y'.
. . .
'zygote' has a vowel. □

We have only shown a few lines of this proof. The number of lines of the full proof equals the number of English words, which is several tens of thousands.

A **universal** statement is a statement that asserts that everything within some set has a specified property.

To prove a *universal* statement, such as . . .

For every English word, it has a vowel or a 'y'

. . . you need to cover every possible case.

One way is to go through all possibilities, in turn, and check each one. But the number of things to check may be huge, or infinite. So usually we want to reason in a way that can apply to many different possibilities at once.

Our next theorem gives a simple subset relationship between two of the languages we introduced earlier. We use this to make some important points about proofs.

Theorem 3
DOUBLEWORD $\subseteq$ EVEN-EVEN

This theorem is true, but the following proof is incorrect. As you read it, think about why it is incorrect.

Pr✗of:

See the example members of DOUBLEWORD listed above. Every one of them has an even number of `a`s and an even number of `b`s.

So every member of DOUBLEWORD is also a member of EVEN-EVEN.

✗

This "proof" is incorrect because the reasoning does not cover all possible cases. It's ok to use examples to *illustrate* concepts as part of *explaining* them. But a *proof* is not an *explanation*.[5] Using examples to do the work of the proof — sometimes called "proof by example" — is, in general, incorrect. There is only one type of situation where "proof by example" works, namely, when the Theorem just asserts the existence of a specific example, or a specific finite set of examples. Theorems 1 and 2 were of this type. But Theorem 3 is not of this type, since it is about *all* members of the *infinite* language DOUBLEWORD. We should never use a proof by example for a *universal* statement about an infinite set of objects.

Now consider Theorem 3 again. This theorem asserts that one set is a subset of another. Here is some general advice on how to prove such a subset relationship. (These three tips do not have to be done in the order given here.)

To prove a subset relationship, $A \subseteq B$:

- Prove that every element of A is also an element of B.
  - Start with a general member of A.
  - Work towards proving that it also belongs to B.
- Give names to things.
- Use the definitions of the sets.

With this advice in mind, let's now prove Theorem 3 properly. We give the proof on the left side. On the right, in *italics* within square brackets, we comment on how the above advice is used, but these comments are not part of the proof itself.

Proof.

Let $w \in$ DOUBLEWORD. *[starting with a general member of DOUBLEWORD, and giving it a name.]*

Then $w = xx$ for some word x. *[using the definition of DOUBLEWORD]*

So, $\#$ `a`'s in w $=$ $2 \times (\ \#$ `a`'s in x $)$, so it's even.

Also, $\#$ `b`'s in w $=$ $2 \times (\ \#$ `b`'s in x $)$, so it's even too.

[working towards proving that w also satisfies the definition of EVEN-EVEN]

So $w \in$ EVEN-EVEN.

$\square$

Reading

- Sipser, pp 13–14
  - strings and languages

[5]...although reading a proof may help explain to you why something is true, and conversely, an explanation may help you to read and understand a proof.

- Sipser, §0.3, pp 17–20
  - definitions, theorems, proofs

Lecture 2

Propositional logic

Logic is used to define many languages we are interested in, to express problems precisely, and for rigorous reasoning in proofs.

2.1 Propositions

A **proposition** is a statement which is either *true* or *false*.

Examples

$1+1=2$	— a proposition which is true.
The earth is flat.	— a proposition which is false.
It will rain tomorrow.	— a proposition.
'Twas brillig, and the slithy toves did gyre and gimble in the wabe.	— *not* a proposition.
Come and work for us!	— *not* a proposition.
This statement is false.	— *not* a proposition.

From: Lewis Carroll, *Through the Looking Glass, and What Alice Found There*, Macmillan, London, 1871.

For brevity, a proposition may be given a name. We think of the name as a variable, and the variable has a **truth value**, True or False.

For example, let X be the proposition $1+1=2$. Then the truth value of X is True.

A **Boolean variable** is a variable that can be True or False. So the name of any proposition is a Boolean variable.

2.2 Logical operations

Propositions may be combined to make other propositions using logical operations. Here are the basic logical operations we will use. We will define each of them shortly.

Not	$\neg$	$(\sim, \overline{\ }, -)$
And	$\wedge$	(&)
Or	$\vee$	
Implies	$\Rightarrow$	$(\rightarrow)$
Equivalence	$\Leftrightarrow$	$(\leftrightarrow)$

Logical operations are also called **connectives**.

2.2.1 Negation

Logical negation is a unary operation which changes True to False and False to True. It is denoted by $\neg$, which is placed before its argument. So, $\neg$True = False and $\neg$False = True. If a proposition P is True then $\neg P$ is False, and if P is False then $\neg P$ is True.

Example:

P:	You have prepared for next week's tutorial.
$\neg P$:	You have not prepared for next week's tutorial.

Other notation for $\neg P$: $\sim P$, $\overline{P}$, $-P$

We can specify how $\neg$ works using a **truth table**, which lists all possible values of the arguments and states, for each, what the resulting value is. In this case, we only have one argument (let's call it P again), which has two possible values, so the table just has two rows (excluding the headings). The left column gives the possible values of the argument P and the right column gives the corresponding values of $\neg P$. We follow the common convention of writing T for True and F for False.

P	$\neg P$
F	**T**
T	**F**

2.2.2 Conjunction

Conjunction is a binary logical operation, i.e., it has two Boolean arguments. The result is True if and only if both its arguments are True. So, if at least one of its arguments is False, then the result is False.

We denote conjunction by $\wedge$ and read it as "and". So the conjunction of P and Q is written $P \wedge Q$ and read as "P and Q". Note that we are using the English word "and" in a strict, precise, logical sense here, which is narrower than the full range of meanings this word can have in English.

Example:

P	Radhanath was a computer.
Q	Radhanath was a person.
$P \wedge Q$	Radhanath was a computer and a person.

Radhanath Sikdar (1813–1870)
http://news.bbc.co.uk/2/hi/south_asia/3193576.stm

We can define conjunction symbolically using its truth table. It has two arguments (let's call them P and Q again), each of which can have two possible values (True and False), so there are $2^2 = 4$ combinations of arguments, hence four rows of the truth table. In each row, the corresponding value of $P \wedge Q$ is given in the last column.

P	Q	$P \wedge Q$
F	F	**F**
F	T	**F**
T	F	**F**
T	T	**T**

2.2.3 Disjunction

Disjunction is another binary logical operation. Its result is True if and only if at least one of its arguments is True. So, if both of its arguments are False, then the result is False.

We denote disjunction by $\vee$ and read it as "or". So the disjunction of P and Q is written $P \vee Q$ and read as "P or Q". The English word "or" is being used here in a strict, precise, logical sense here, much narrower than its full range of English meanings; an analogous situation arose with "and" previously. Also, we are using the word "or" inclusively, so that a disjunction is True whenever *any one or more* of its arguments are True. For this reason, disjunction is sometimes called *inclusive-OR*. (This contrasts with the *exclusive-or* of two propositions, which is True precisely when *exactly one* of its two arguments is True.)

Example:

P	I will study FIT3155 Advanced Data Structures & Algorithms.
Q	I will study MTH3170 Network Mathematics.
$P \vee Q$	I'll study FIT3155 **or** I'll study MTH3170.
	I'll study *at least one of* FIT3155 and MTH3170.

Here is the truth table definition of disjunction.

P	Q	$P \vee Q$
F	F	**F**
F	T	**T**
T	F	**T**
T	T	**T**

2.2.4 De Morgan's Laws

Conjunction and disjunction seem somewhat opposite in character, but there is a close relationship between them, a kind of logical duality. This is captured by **De Morgan's Laws**.

$$\neg(P \vee Q) \quad = \quad \neg P \wedge \neg Q$$

$$\neg(P \wedge Q) \quad = \quad \neg P \vee \neg Q$$

Augustus De Morgan
(1806–1871)
https://mathshistory.st-andrews.ac.uk/Biographies/De_Morgan/

These laws can be proved using truth tables. Consider the table below, which proves the first of De Morgan's Laws, $\neg(P \vee Q) = \neg P \wedge \neg Q$. We start out with the usual two columns giving all combinations of truth values of our variables P and Q. The overall approach is to gradually work along to the right, adding new columns that give some part of one of the expressions we are interested in, always using the columns we've already constructed in order to construct new columns. We'll first work towards constructing a column giving truth values for the left-hand side of the equation, $\neg(P \vee Q)$. As a step towards this, we make a column for $P \vee Q$: this becomes our third column. In fact, our first three columns are just the truth table for the disjunction $P \vee Q$, which we have seen before. Then we negate each entry in the third column to give the entries of the fourth column, whch gives the truth table for $\neg(P \vee Q)$. So we've done the left-hand side of the equation. Then we start on the right-hand side of the equation, which is $\neg P \wedge \neg Q$. For this, we'll need $\neg P$ and $\neg Q$, which are obtained by negating the columns for P and Q respectively. This gives the fifth and sixth columns. Finally we form $\neg P \wedge \neg Q$ in the seventh column by just taking the conjunction of the corresponding entries in the fifth and sixth columns.

We now have columns giving the truth tables of both sides of the first of De Morgan's Laws: these are the fourth and seventh columns below, shown in green. These columns are identical! This shows that the two expressions $\neg(P \vee Q)$ and $\neg P \wedge \neg Q$ are logically equivalent, i.e., their truth values are the same for all possible assignments of truth values to their arguments P and Q. In other words, as Boolean expressions, they are equal. So $\neg(P \vee Q) = \neg P \wedge \neg Q$. This proves the first of De Morgan's Laws.

P	Q	$P \vee Q$	$\neg(P \vee Q)$	$\neg P$	$\neg Q$	$\neg P \wedge \neg Q$
F	F	F	T	T	T	T
F	T	T	F	T	F	F
T	F	T	F	F	T	F
T	T	T	F	F	F	F

We could prove the second of De Morgan's Laws by the same method. But, now that we know the first of De Morgan's Laws, it is natural to ask: can we use it to prove the second law more easily, so that we avoid doing the same amount of work all over again?

De Morgan's Laws can also be proved by reasoning in a way that covers all possible combinations of truth values, rather than working laboriously through each combination of truth values separately (which is what the truth table proof does). We give such a proof now. For good measure, we do it for a more general version of the Law which caters for arbitrarily long conjunctions and disjunctions. This would not be possible just using the truth table approach, since the length of the proof must be bounded by a finite constant.

Theorem 4
For all n:

$$\neg(P_1 \vee \cdots \vee P_n) = \neg P_1 \wedge \cdots \wedge \neg P_n$$

First proof:
Left-Hand Side is True
if and only if $P_1 \vee \cdots \vee P_n$ is False
if and only if $P_1, \ldots, P_n$ are all False
if and only if $\neg P_1, \ldots, \neg P_n$ are all True
if and only if Right-Hand Side is True. □

Again, we ask: having proved the first of De Morgan's Laws (in this more general form), can we use it to prove the second law more easily? How would you prove the second law?

2.2.5 Conditional

Our next logical operation is the **conditional**, also called **implication**. If P and Q are its arguments, it is written $P \Rightarrow Q$ and read "P implies Q" or "if P then Q". It means that if P is True then Q must also be True. The only way its result can be False is if P is True and Q is False.

It is occasionally convenient to write $P \Rightarrow Q$ in its equivalent reversed form, $Q \Leftarrow P$. Observe, though, that to do this, we have to reverse *everything* in the expression: the direction of the arrow *and* the order of the arguments. If we just do *one* of these reversals, then we get a different expression with different meaning. So $P \Rightarrow Q$ is different to both $Q \Rightarrow P$ and $P \Leftarrow Q$.

Example:

P	Stars are visible.
Q	The sun has set.

$P \Rightarrow Q$	**If** stars are visible **then** the sun has set.
	Stars being visible **implies** the sun has set.
	Stars are visible **only if** the sun has set.
	Stars are visible is **sufficient** for the sun to have set.

Another example:

P	Grace is a COBOL expert.
Q	Grace can program.

$P \Rightarrow Q$ **If** Grace is a COBOL expert **then** she can program.

Grace Hopper (1906–1992)
https://www.cs.vassar.edu/history/hopper

Here is its truth table.

P	Q	$P \Rightarrow Q$
F	F	**T**
F	T	**T**
T	F	**F**
T	T	**T**

2.2.6 Biconditional

Our last binary logical operation is the **biconditional**. For arguments P and Q, the biconditional is written $P \Leftrightarrow Q$ and read "P if and only if Q" or "P is equivalent to Q". It is true precisely when P and Q are either both True or both False. In other words, as Boolean variables, their values are always equal.

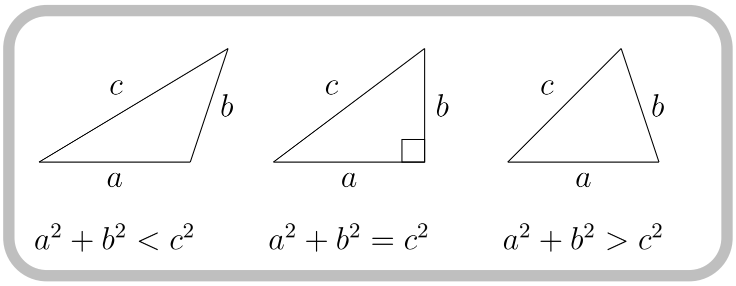
You can view $P \Leftrightarrow Q$ as the conjunction of $P \Rightarrow Q$ and $Q \Rightarrow P$:

$$P \Leftrightarrow Q \quad = \quad (P \Rightarrow Q) \wedge (Q \Rightarrow P).$$

$P \Leftrightarrow Q$ can be written the other way round, as $Q \Leftrightarrow P$. They have the same meaning.

Example:

P	The triangle is right-angled.
Q	The side lengths satisfy $a^2 + b^2 = c^2$.



$P \Leftrightarrow Q$ The triangle is right-angled **if and only if** $a^2 + b^2 = c^2$.

The triangle being right-angled is a **necessary and sufficient condition** for $a^2 + b^2 = c^2$.

$a^2 + b^2 = c^2$ is a **necessary and sufficient condition** for the triangle being right-angled.

Here is its truth table.

P	Q	$P \Leftrightarrow Q$
F	F	**T**
F	T	**F**
T	F	**F**
T	T	**T**

2.3 Tautologies and logical equivalence

A **tautology** is a statement that is always true. In other words, the right-hand column of its truth table has every entry True.

Two statements P and Q are **logically equivalent** if their truth tables are identical.

In other words, P and Q are equivalent if and only if $P \Leftrightarrow Q$ is a tautology.

Examples

$\neg\neg P$	is logically equivalent to	P
$\neg(P \vee Q)$	is logically equivalent to	$\neg P \wedge \neg Q$
$\neg(P \wedge Q)$	is logically equivalent to	$\neg P \vee \neg Q$
$P \Rightarrow Q$	is logically equivalent to	$\neg P \vee Q$
$P \Leftrightarrow Q$	is logically equivalent to	$(P \Rightarrow Q) \wedge (P \Leftarrow Q)$
	and to	$(\neg P \vee Q) \wedge (P \vee \neg Q)$

These can all be proved using truth tables.

We usually denote logical equivalence by "=". So we write $\neg\neg P = P$, etc.

We have already met logical equivalence. Each of De Morgan's Laws states that its left-hand side is logically equivalent to its right-hand side.

2.3.1 History

George Boole (1815–1864)
https://mathshistory.st-andrews.ac.uk/Biographies/Boole/

2.3.2 Distributive Laws

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$

$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

It is notable that we have <u>two</u> distributive laws in propositional logic. Conjunction is distributive over disjunction (first law above), and disjunction is distributive over conjunction (second law above).

This contrasts with the situation for ordinary algebra of numbers, when multiplication is distributive over addition, but addition is <u>not</u> distributive over multiplication.

$$\begin{array}{rcl} p \times (q + r) & = & (p \times q) + (p \times r) \\ & \textbf{\textit{but}} & \\ p + (q \times r) & \neq & (p + q) \times (p + r) \end{array}$$

2.4 Laws of Boolean algebra

Here is a full listing of the laws of Boolean algebra, which we may use to convert propositional expressions from one form to another. Reasons why we might do this include finding simpler forms for Boolean expressions and determining algebraically whether or not two given Boolean expressions are logically equivalent.

Because of the logical duality between conjunction and disjunction, these laws may be arranged in dual pairs. In the table below, each law involving conjunction or disjunction is written on the same line as its dual.

$$\neg\neg P = P$$

$$\begin{array}{rclcrcl} \neg\text{True} & = & \text{False} & & \neg\text{False} & = & \text{True} \\ P \wedge Q & = & Q \wedge P & & P \vee Q & = & Q \vee P \\ (P \wedge Q) \wedge R & = & P \wedge (Q \wedge R) & & (P \vee Q) \vee R & = & P \vee (Q \vee R) \\ P \wedge P & = & P & & P \vee P & = & P \\ P \wedge \neg P & = & \text{False} & & P \vee \neg P & = & \text{True} \\ P \wedge \text{True} & = & \text{P} & & P \vee \text{False} & = & \text{P} \\ P \wedge \text{False} & = & \text{False} & & P \vee \text{True} & = & \text{True} \end{array}$$

Distributive Laws

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R) \qquad P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

De Morgan's Laws

$$\neg(P \vee Q) = \neg P \wedge \neg Q \qquad \neg(P \wedge Q) = \neg P \vee \neg Q$$

2.5 Disjunctive Normal Form (DNF)

We will introduce two standard ways of writing logical expressions. The first of these is *Disjunctive Normal Form (DNF)*, treated in this subsection, The second is *Conjunctive Normal Form (CNF)*, treated in the next subsection.

We treat DNF first, but CNF will be much more important for us. In brief, that's because CNF is more natural for encoding logical problems and real-world conditions. You can always convert one to the other, but at considerable cost in both time and space as we will see.

The next definition is needed here and also later in the unit.

A **literal** is an appearance of a logical variable in which it is either unnegated or negated just once. So, if X is a logical variable, then its corresponding literals are X and $\neg X$. Separate appearances of a logical variable within a larger logical expresion are counted as separate literals. For example, the expression $(\neg X \wedge \neg Y) \vee (\neg X \wedge Y) \vee (X \wedge Y)$ has six literals (even though some are equivalent to each other).

A Boolean expression is in **Disjunctive Normal Form (DNF)** if

- it is written as a disjunction of some number of parts, *where*
- each part is a conjunctijon of some number of literals.

Examples:

$(\neg X \wedge \neg Y) \vee (\neg X \wedge Y) \vee (X \wedge Y)$	a disjunction of three parts, each of which is a conjunction of two literals
$(A \wedge \neg B) \vee (\neg A \wedge \neg B \wedge C \wedge D)$	a disjunction of two parts, where the first part is a conjunction of two literals and the second part is a conjunction of four literals.

You can convert any proposition into an equivalent one in DNF using its truth table.

Consider the proposition P given by the following truth table.

X	Y	P
F	F	**T**
F	T	**T**
T	F	**F**
T	T	**T**

As a step towards designing a logical expression for the entire proposition P, we will design one logical expression for each row where P is True.

Consider the first row. This is for when $X =$ False and $Y =$ False. We want the result to be True. We can do this using $\neg X \wedge \neg Y$. Satisfy yourself that this is True when $X =$ False and $Y =$ False, and also that it is False *for every other combination of truth values* for X and Y. So it is True *only* in this first row, as its own truth table shows:

X	Y	$\neg X \wedge \neg Y$
F	F	**T**
F	T	F
T	F	F
T	T	F

Now consider the second row of the truth table for P, which is for when $X =$ False and $Y =$ True. This time we will use $\neg X \wedge Y$, which is True for this row but False for all the other rows. We can add an extra column to include its truth table too.

X	Y	$\neg X \wedge \neg Y$	$\neg X \wedge Y$
F	F	**T**	F
F	T	F	**T**
T	F	F	F
T	T	F	F

The third row of the truth table for P has $P =$ False, so we will ignore that.

The fourth row of the truth table for P has $P =$ True again. We will now use $\neg X \wedge Y$, which is True for this row but for no other. We add a further column for its truth table.

X	Y	$\neg X \wedge \neg Y$	$\neg X \wedge Y$	$X \wedge Y$
F	F	**T**	F	F
F	T	F	**T**	F
T	F	F	F	F
T	T	F	F	**T**

Now look at what happens when we take the disjunction of the last three columns.

X	Y	$\neg X \wedge \neg Y$	$\neg X \wedge Y$	$X \wedge Y$	$(\neg X \wedge \neg Y) \vee (\neg X \wedge Y) \vee (X \wedge Y)$
F	F	**T**	F	F	**T**
F	T	F	**T**	F	**T**
T	F	F	F	F	F
T	T	F	F	**T**	**T**

This shows that the DNF expression $(\neg X \wedge \neg Y) \vee (\neg X \wedge Y) \vee (X \wedge Y)$ is equivalent to P.

$$P \quad = \quad \underbrace{\overbrace{(\neg X \wedge \neg Y)}^{\text{conjunction}} \vee \overbrace{(\neg X \wedge Y)}^{\text{conjunction}} \vee \overbrace{(X \wedge Y)}^{\text{conjunction}}}_{\text{disjunction}}$$

DNF expressions like this can be read easily from truth tables. You don't have to add extra columns as we did above. In each row, look at the pattern of truth values for the variables. Then write down a conjunction of literals where variables that are True are written normally and variables that are False are written in negated form. For example, in the second row of our truth table, the variable X is False so we negate it, whereas Y is True so we just use it unchanged. Taking the conjunction of these two literals gives $\neg X \wedge Y$, which is the part we want for that row. Do this for every row in which the proposition P is True. This is shown for our current example in the following table.

X	Y	P	
F	F	**T**	$\neg X \wedge \neg Y$
F	T	**T**	$\neg X \wedge Y$
T	F	**F**	
T	T	**T**	$X \wedge Y$

When this is done, just take the disjunction of all the parts in the final column and you have your DNF expression for P.

Exercise: simplify the above expression P as much as possible, using Boolean algebra.

Here is another example of this method. The columns X, Y, Z correspond to three variables X, Y, Z, and the fourth column gives the truth table for a new proposition P. In the fifth column, we look at each row where $P =$ True and convert the first three truth values in that row into a conjunction of literals that follows the required pattern. For this particular P, this gives four conjunctions.

X	Y	Z	P	
F	F	F	**T**	$\neg X \wedge \neg Y \wedge \neg Z$
F	F	T	**F**	
F	T	F	**T**	$\neg X \wedge Y \wedge \neg Z$
F	T	T	**F**	
T	F	F	**F**	
T	F	T	**T**	$X \wedge \neg Y \wedge Z$
T	T	F	**T**	$X \wedge Y \wedge \neg Z$
T	T	T	**F**	

Taking the disjunction of all the conjunctions in the last column gives a DNF expression for P.

$$P = (\neg X \wedge \neg Y \wedge \neg Z) \vee (\neg X \wedge Y \wedge \neg Z) \vee (X \wedge \neg Y \wedge Z) \vee (X \wedge Y \wedge \neg Z)$$

The importance of this method is that it shows that every logical expression is equivalent to one in DNF. So DNF can be viewed as a “standard form” into which any logical expression can be transformed.

BUT this transformation comes at a price. The DNF expression has as many parts as there are truth table rows where the expression is True. An expression with k variables has 2^k rows in its truth table, which is exponential in the number of variables. If the expression is True for a large proportion of its truth table rows, then the number of parts in the DNF expression may also be exponentially large in the number of variables. So it may be too large and unwieldy to be useful.

One apparent attraction of DNF is that it is easy to tell if a DNF expression is satisfiable, that is, if there is some assignment of truth values to its variables that makes the whole expression True. In fact, if you take any of the parts of a DNF expression, the pattern of the literals (i.e., whether each appears plainly or negated) tells you a truth assignment that makes that part True, and then the whole disjunction must also be True because a disjunction is True precisely when at least one of its parts is True. In effect, the parts of a DNF expression yield a kind of encoded listing of all the truth assignments that make the whole expression True.

The problem with DNF, though, is that, in real life, logical rules are not usually specified in a form that is amenable to DNF. They are typically described by listing conditions that must be satisfied together. In other words, they are described in a way that lends itself to expression as a conjunction rather than a disjunction.

2.6 Conjunctive Normal Form (CNF)

A Boolean expression is in **Conjunctive Normal Form (CNF)** if

- it is written as a conjunction of some number of parts (sometimes called **clauses**), *where*
- each part is a disjunctijon of some number of literals.

For example:

$$\underbrace{\overbrace{(\neg P \lor Q)}^{\text{disjunction}} \land \overbrace{(P \lor \neg Q)}^{\text{disjunction}}}_{\text{conjunction}}$$

There is a close relationship — a kind of logical duality — between CNF and DNF. Suppose you have an expression P in CNF, and suppose you negate it, giving $\neg P$. So $\neg P$ is a negation of a conjunction of a disjunction of literals. But, by De Morgan's Law, a negation of a conjunction is a disjunction of negations. So $\neg P$ will then be expressed as a disjunction of negations of disjunctions of literals. But, again by De Morgan's Law, each negation of a disjunction is equivalent to a conjunction of negations. So $\neg P$ is now a disjunction of conjunctions of negations of literals. But the negation of a literal is always equivalent to another literal (since $\neg\neg X = X$). So we see that $\neg P$ is a disjunction of conjunctions of literals. In other words, it's in DNF.

We now have a way to convert any logical expression P in truth table form to a CNF expression. To do so, we just

1. negate all the truth values in the output column (turning it into a truth table for $\neg P$),
2. use the method of the previous subsection to construct a DNF expression for $\neg P$,
3. then negate the expression (so that it will now be an expression for P again),
4. and use De Morgan's Laws to transform the negated DNF expression into CNF.

This establishes the important theoretical point that every expression is equivalent to a CNF expression. But this is usually not a good way to construct CNF expressions in practice, because:

- Truth tables are too large. As discussed earlier, their size is exponential in the number of variables. Complex logical conditions in real computational problems usually contain enough variables for truth tables to be unusable. Furthermore, even for more modest-sized problems, the truth table approach for CNF uses a lot of time and space and, for manual work, is quite error-prone.
- Logical conditions are usually expressed in a way that makes CNF a natural way to represent them.

2.7 Representing logical statements

Suppose we are given a set of rules or conditions that we want to model as a logical expression. These are often expressed as conditions that must be satisfied *together*. For example, the rules of a game must *all* be folllowed, to play the game correctly; you can't just ignore the rules you don't like on the grounds that you're following *some* of the rules! A software specification typically stipulates a set of conditions that must all be met, and similarly for legal contracts, acts of parliament, traffic regulations, itineraries, and so on. While some rules may offer choices as to how they can be satisfied, at the highest level a set of rules is usually best modelled as a conjunction.

So, if you are given some specifications and you want to model them by a logical expression, one first step you can take (working " top-down") is to identify how the rule is structured at the top level as a conjunction. What are the parts of this conjunction? You can then keep working top-down and try to decompose those parts as conjunctions too. A conjunction of conjunctions is just one larger conjunction.

Working from the other direction ("bottom-up"), you also have to think about what your most elementary logical "atoms" are. In other words, think about the simplest, most basic assertion that could hypothetically be made about this situation, without worrying about whether it might be True or False. In fact, in this kind of situation, you won't initially know what values your logical variables might have; you're merely encoding your problem in logical form, without solving it yet, so you avoid thinking about actual truth values for any of your variables. You are just trying to identify the kinds of "atomic assertions" that are needed to describe *any* hypothetical situation in your scenario.

Example:

You are planning a dinner party. Your guest list must have:

- at least one of: Harry, Ron, Hermione, Ginny
- Hagrid *only if* it also has Norberta
- none, or both, of Fred and George
- no more than one of: Voldemort, Bellatrix, Dolores.

At the top level, we can see that this is a conjunction:

(at least one of: Harry, Ron, Hermione, Ginny)
$\wedge$ (Hagrid *only if* it also has Norberta)
$\wedge$ (none, or both, of Fred and George)
$\wedge$ (no more than one of: Voldemort, Bellatrix, Dolores).

Note that we're not trying to convert everything to logic at once. The four parts of our conjunction are not yet expressed in logical form; they're still just written in English text. That's ok, in this intermediate stage.

Early in the process, we should think about what Boolean variables to use, and what they should represent. In this case, that is fairly straightforward. The simplest logical statement we can make in this situation is that a specific person is on your guest list. So, for each person, we'll introduce a Boolean variable with the intended interpretation that the person is on your guest list. So, the variable Harry is intended to mean that the person Harry is on your guest list, and so on. This gives us eleven variables, one for each of our eleven guests. As far as we know at the moment, each of the variables might be True or False; it is the role of the logical expression we are constructing to ensure that the combinations of truth values for these eleven variables must correspond to valid guest lists. We will do that by properly representing the rules in logic. We will not try, at this stage, to enumerate all possible guest lists, or even to find one valid guest list. Our current task is to encode the problem's rules in logic, not to *solve* the problem. (That comes later in the semester.)

Now, let's look at each of the four parts of our conjunction, in turn, and see how they may be logically expressed using our variables.

- at least one of: Harry, Ron, Hermione, Ginny.
  This is a *disjunction* of the four corresponding variables:

  $$\text{Harry} \vee \text{Ron} \vee \text{Hermione} \vee \text{Ginny}$$

- Hagrid *only if* it also has Norberta.
  This is modelled by *implication*:

  $$\text{Hagrid} \Rightarrow \text{Norberta}$$

- none, or both, of Fred and George.
  This is a job for the *biconditional*:

  $$\text{Fred} \Leftrightarrow \text{George}$$

- no more than one of: Voldemort, Bellatrix, Dolores.
  This requires some more thought! Unlike the previous parts, it doesn't correspond immediately to a logical operation we have already met. So we will try to see if it can be broken down further into a conjunction of simpler conditions. To do this, consider that "no more than one of ..." is the same as saying that every *pair* of them is forbidden. So, the pair Voldemort & Bellatrix is forbidden, and the pair Voldemort & Dolores is forbidden, and the pair Bellatrix & Dolores is forbidden. See the logical structure emerging: "... and ... and ...". So we have:

  $$\begin{gathered}(\textit{not both}\ \text{Voldemort \& Bellatrix}) \wedge \\ (\textit{not both}\ \text{Voldemort \& Dolores}) \wedge \\ (\textit{not both}\ \text{Bellatrix \& Dolores})\end{gathered}$$

So our expression is now:

$$\begin{array}{ll} \multicolumn{2}{l}{(\text{Harry} \vee \text{Ron} \vee \text{Hermione} \vee \text{Ginny})} \\ \wedge & (\text{Hagrid} \Rightarrow \text{Norberta}) \\ \wedge & (\text{Fred} \Leftrightarrow \text{George}) \\ \wedge & (\textit{not both}\ \text{Voldemort \& Bellatrix}) \\ \wedge & (\textit{not both}\ \text{Voldemort \& Dolores}) \\ \wedge & (\textit{not both}\ \text{Bellatrix \& Dolores}). \end{array}$$

We are now getting to the point where we can use logical manipulations (Boolean algebra) to transform each of the four parts into a disjunction of literals. The first part is already a disjunction. The second part can be written as the disjunction $\neg$Hagrid $\vee$ Norberta, as we have already seen. The third part can be written

$$(\text{Fred} \Rightarrow \text{George}) \wedge (\text{Fred} \Leftarrow \text{George})$$

and turning each implication into an appropriate disjunction gives

$$(\neg\text{Fred} \vee \text{George}) \wedge (\text{Fred} \vee \neg\text{George}).$$

For the fourth part, requiring that Voldemort and Bellatrix are *not both True* is the same as requiring at least one of them to be False, which is the same as requiring at least one of their negations to be True, which is captured by $\neg$Voldemort $\vee$ $\neg$Bellatrix. Treating the other two pairs the same way gives the conjunction of disjunctions

$$(\neg\text{Voldemort} \vee \neg\text{Bellatrix}) \wedge (\neg\text{Voldemort} \vee \neg\text{Dolores}) \wedge (\neg\text{Bellatrix} \vee \neg\text{Dolores})$$

Putting these altogether gives the expression

$$\begin{aligned}
&(\text{Harry} \vee \text{Ron} \vee \text{Hermione} \vee \text{Ginny})\\
&\wedge \ \ (\neg\text{Hagrid} \vee \text{Norberta})\\
&\wedge \ \ (\neg\text{Fred} \vee \text{George}) \ \wedge \ (\text{Fred} \vee \neg\text{George})\\
&\wedge \ \ (\neg\text{Voldemort} \vee \neg\text{Bellatrix}) \ \wedge \ (\neg\text{Voldemort} \vee \neg\text{Dolores}) \ \wedge \ (\neg\text{Bellatrix} \vee \neg\text{Dolores})
\end{aligned}$$

This is now in CNF.

Challenge: how long would an equivalent DNF expression be?

2.8 Statements about how many variables are True

Given a collection of Boolean variables, we are often interested in how many of them are True. We might want to state that <u>at least</u> two of them are True, or that <u>at most</u> two of them are True, or <u>exactly</u> two of them are True. We might want to make similar statements with "two" replaced by some other number.

We saw examples of this in the previous section. We wanted <u>at least</u> *one* of Harry, Ron, Hermione and Ginny to be True. We also wanted <u>at most</u> *one* of Voldemort, Bellatrix and Dolores to be True. So, to build your intuition about dealing with these situations, it would be worth pausing now and spending a few minutes reading that analysis again, and thinking through how the reasoning given there could be extended to situations where the number of True variables involved is some other number (i.e., not one).

If ever we want to specify that <u>exactly</u> k variables are True, we can express this as a conjunction:

$$(\underline{\text{at least}}\ k \text{ are True}) \ \wedge \ (\underline{\text{at most}}\ k \text{ are True}).$$

So we now focus our discussion on just the "at least" and "at most" cases.

Suppose we want to state that at most k of the n variables $x_1, x_2, \ldots, x_n$ are True. This means that, for every set of $k+1$ variables, at least one of them is False, or in other words, at least one of their negations is True. So, for every set of $k+1$ variables, we form a disjunction of their negations (to say that at least one of these negations is True), and then we combine all these disjunctions into a larger conjunction.

For example, suppose we want to say that at most *two* of the four variables w, x, y, z is True (i.e., $k = 2$ and $n = 4$). We create all disjunctions of *triples* $(k + 1 = 3)$ of *negated* literals, which gives the disjunctions

$$\neg w \vee \neg x \vee \neg y, \quad \neg w \vee \neg x \vee \neg z, \quad \neg w \vee \neg y \vee \neg z, \quad \neg x \vee \neg y \vee \neg z\,.$$

These are then combined by conjunction:

$$(\neg w \vee \neg x \vee \neg y) \;\wedge\; (\neg w \vee \neg x \vee \neg z) \;\wedge\; (\neg w \vee \neg y \vee \neg z) \;\wedge\; (\neg x \vee \neg y \vee \neg z).$$

Now suppose we want to state that at least k of the n variables $x_1, x_2, \ldots, x_n$ are True. This means that at most $n-k$ of them are False. This means that, for every set of $n-k+1$ variables, at least one of them is True. So, for every set of $n-k+1$ variables, we form a disjunction of them (to say that at least one of them is True), and then we combine all these disjunctions into a larger conjunction.

For example, suppose we want to say that at least *two* of the four variables w, x, y, z is True (i.e., $k = 2$ and $n = 4$). We create all disjunctions of textittriples $(n-k+1 = 4-2+1 = 3)$ of literals (unnegated, this time), which gives the disjunctions

$$w \vee x \vee y, \quad w \vee x \vee z, \quad w \vee y \vee z, \quad x \vee y \vee z\,.$$

These are then combined by conjunction:

$$(w \vee x \vee y) \;\wedge\; (w \vee x \vee z) \;\wedge\; (w \vee y \vee z) \;\wedge\; (x \vee y \vee z).$$

Finally, if we want to say that exactly two of the four variables are True, then we take the conjunction of the expressions for "at least" and "at most", giving

$$\begin{array}{l}(\neg w \vee \neg x \vee \neg y) \;\wedge\; (\neg w \vee \neg x \vee \neg z) \;\wedge\; (\neg w \vee \neg y \vee \neg z) \;\wedge\; (\neg x \vee \neg y \vee \neg z) \;\wedge \\ (w \vee x \vee y) \;\wedge\; (w \vee x \vee z) \;\wedge\; (w \vee y \vee z) \;\wedge\; (x \vee y \vee z).\end{array}$$

2.9 Reading

See Sipser, pp. 14–15, and top paragraph of p. 302.

Lecture 3

Predicate logic

3.1 Statements with variables

Consider these statements:

- W is negative.
- X passed this subject.
- $Y = Z$.

These statements do not yet have truth values, so they are not yet propositions.

The variables in these statements are **free**, in that no value is (yet) given to them.

You can, if you wish, assign values to these variables. Each set of values you give to the variables creates a different specific proposition.

For example, in the statement W is negative, the variable W is free. If we assign values to it, we can create specific propositions:

$\vdots$	$\vdots$
-2 is negative	True
-1 is negative	True
0 is negative	False
1 is negative	False
2 is negative	False
$\vdots$	$\vdots$

3.2 Predicates

A **predicate** is a statement with variables such that, for any values of the variables, it is either True or False, i.e., it becomes a proposition.

We treat each variable as ranging over some **domain**.

In the previous example, for the predicate W is negative, we've just been using the domain $\mathbb{Z}$.

The variables of a predicate are also called its **arguments**. A predicate is called k**-ary** if it has k arguments. Special terms exist for some small numbers of arguments: a predicate is **unary**, or **binary**, or **ternary**, according as it has one, two or three arguments, respectively.

A predicate with one argument (i.e., a unary predicate) is also called a **property**.

A predicate with at least two arguments is also called a **relation**.

Here are some examples of predicates, with the domains of their variables.

# args.	example	domain
1	isNegative(X)	$\mathbb{N}$
1	isNegative(X)	$\mathbb{Z}$
2	$=$ *[always available]*	objects
2	$X < Y$	numbers
2	isMotherOf(X, Y), meaning "X is the mother of Y"	people
3	gives(X, Y, Z), meaning "X gives Y to Z"	X, Z are people, Y is a gift
⋮	⋮	⋮

The equality predicate, $=$, is always considered to be available. You do not need to be told that you are allowed to use it; you always can, no matter what type of objects you are working with.

Predicates may be thought of as *truth-valued functions*, i.e., functions whose value is always in {True, False}.

Exercise: what could a predicate with no arguments be?

3.3 Functions

We'll also use functions whose values aren't necessarily just True or False.

For example:

# args.	example	domain	codomain
1	$\sqrt{X}$	nonnegative numbers	numbers
1	motherOf(X)	people	people
2	$X + Y$	numbers	numbers
⋮	⋮	⋮	

Functions with no arguments are called **constants**.

- Examples: 5, Annie, …

A function's arguments can be constants, or variables, or other functions (or even the same function). If an argument of a function is also a function, then that second function must also have arguments which can be constants, variables or functions. This can continue, but only to finite depth: you cannot have functions within functions within functions … and so on and on, forever.

3.4 Existential quantifier

The **existential quantifier** is written $\exists$ and read as "there exists". It is placed before a variable to mean that there exists some value of that variable, within the variable's domain, that makes the subsequent statement True.

For example, consider the statement

There's a fly in my soup.

This may be written

$$\exists X : \ (X \text{ is a fly}) \wedge (X \text{ is in my soup}).$$

Here, "X is a fly" is a unary predicate with variable X, and so is "X is in my soup". Their conjunction gives another predicate with the same variable. At the beginning, we have the quantifier $\exists$ applied to the variable X, so the entire statement is asserting that

There exists X such that X is a fly and X is in my soup.

This is just a rewording of the statement we started with.

We usually need to specify the domains of all the variables when writing any expression that uses them. It might be clear from the context, or it might be specified as part of the written expression by specifying domain membership immediately after the variable, as in, $\exists Y \in \mathbb{Q} \ \cdots$ (so in this case the domain of Y is $\mathbb{Q}$). The domain of a variable certainly affects the meaning of the expression it is part of, in general; changing the domain might change the truth value of the expression.

For example, consider the following expression, written as text on the left and as a logical expression on the right.

There exists W : W is negative. $\qquad \exists W : W < 0.$

If domain of W is $\mathbb{N}$, then we could write the expression as $\exists W \in \mathbb{N} : W < 0$, and it is False.

If domain of W is $\mathbb{Z}$, then we could write the expression as $\exists W \in \mathbb{Z} : W < 0$, and it is True.

There is an analogy between the existential quantifier and disjunction. In each case, the expression that uses them is True if and only if at least one of its "possibilities" is true. For a disjunction, we require that at least one of the parts of the disjunction is true; for an existential quantifier applied to some variable X, we require that at least one value of X makes the entire expression True. In other words, at least one member of the domain of X may be assigned to X to make the expression True.

For example, suppose we have the statement "Someone did it". We may write this more formally as

$$\exists X : X \text{ did it.} \tag{3.1}$$

Suppose the domain of X is a large set of people,

$$\{\ldots, \text{Annie}, \text{Edward}, \text{Henrietta}, \text{Radhanath}, \ldots\}.$$

Then the quantified expression (3.1) is *sort-of* like a disjunction ...

$$\cdots\cdots \vee (\text{Annie did it}) \vee (\text{Edward did it}) \vee (\text{Henrietta did it}) \vee (\text{Radhanath did it}) \vee \cdots\cdots$$

However, the existential quantifier is not just a shorthand notation for disjunction. Firstly, if the domain of a variable is infinite, then existential quantification over that variable cannot be replaced by a disjunction because a disjunction is only allowed to have finitely many parts (and, indeed, logical expressions in general must be of finite size). Secondly, variables and their quantifiers allow us to do some reasoning that cannot be done in propositional logic.

Once a variable in an expression has had a quantifier applied to it, so that all occurrences of the variable come after the quantifier and are subject to it, the variable is said to be **bound**. You can no longer give specific values to the variables to create specific propositions. The quantifier has turned the statement into a single proposition about the entire domains of the variables.

Note that quantifiers can only be used with *variables*. Using them with constant objects *makes no sense.* It is an error to write something like $\exists 5$, $\exists$Annie.

It is useful to consider how to incorporate a restriction on an existentially-quantified variable into a predicate logic statement.

Suppose we have the statement

Some computer is human. *i.e.,* There exists a human computer.

To begin with, suppose the domain of X is $\{\text{computers}\}$, and that we have the predicate human(X) which is intended to mean that X is human.

Then our statement may be written

$$\exists X : \text{human}(X).$$

But what if the domain of X is $\{\text{everything on Earth}\}$?

In this case, suppose we have another predicate, computer(X), which means that X is a computer.

The following table shows the correct expression on the left and an incorrect attempt at the expression on the right.

Correct: $\exists X : \text{computer}(X) \wedge \text{human}(X)$	*Incorrect:* $\exists X : \text{computer}(X) \Rightarrow \text{human}(X)$
• "There exists something that is both computer and human." • "There exists a human computer." • "Some computer is human."	• "There exists something which is not a computer or is human." • "There exists something which is not both a computer and non-human." • "Not everything is a nonhuman computer."

3.5 Universal quantifier

The **universal quantifier** is written $\forall$ and read as "for all" (or "for every" or "for each"[1]). It is placed before a variable to mean that for all values of that variable, within the variable's domain, the subsequent statement is True.

For example, consider the statement

Everyone can pass this subject.	For every X : X can pass this subject.

This may be written

$$\forall X : \text{canPass}(X).$$

This statement happens to be True, provided the domain of X is the set of students who have been allowed to enrol in the subject.

Here is a famous statement:

All numbers are interesting.	$\forall X$: X is interesting.	$\forall X : \text{isInteresting}(X)$.

This statement is also True, and we'll prove it!

The following statement is False:

For all W : W is negative	$\forall W : W < 0$.	False.

Now that we've applied quantifiers to the variables in these expressions, the variables are all **bound**.

We now consider how to incorporate a restriction on a universally-quantified variable into a predicate logic statement.

Again, we use our statement that "Every computer is human" and suppose we have the predicate human(X) with the same meaning as before.

If the domain of X is { computers }, then our statement may be written

$$\forall X : \text{human}(X).$$

But what if the domain of X is { everything on Earth } ?

As before, suppose we have the predicate computer(X).

The following table shows the correct expression on the *right* (*not* the left, this time) and an incorrect attempt at the expression on the *left*.

Incorrect: $\forall X : \text{computer}(X) \wedge \text{human}(X)$	*Correct:* $\forall X : \text{computer}(X) \Rightarrow \text{human}(X)$
• "Everything is both computer and human." • "Everything is a human computer."	• "For everything, if it's a computer, then it's human." • "Everything that's a computer is also human." • "Every computer is human."

[1]but we do *not* read it as "for some", because that means the same as "there exists".

3.6 Multiple quantifiers

It is permissible to have multiple quantifiers in an expression.

If there are two consecutive quantifiers at the start of an expression, then they must have different variables. (It makes no sense to write something like $\exists X \, \forall X \cdots$.)

We illustrate how quantifiers work together with an example from graph theory.

Suppose we have a predicate $\mathrm{adj}(X, Y)$ meaning that vertices X and Y are adjacent (in a particular graph). The following table shows how we might represent various statements, with the statements on the left and the predicate logic expressions on the right.

Some two vertices are not adjacent.	$\exists X \, \exists Y : \neg\mathrm{adj}(X, Y).$
Every pair of vertices is adjacent.	$\forall X \, \forall Y : \mathrm{adj}(X, Y).$
Some vertex is adjacent to all other vertices.	$\exists X \, \forall Y : \mathrm{adj}(X, Y).$
Every vertex has a neighbour.	$\forall X \, \exists Y : \mathrm{adj}(X, Y).$

The logical expressions in the above table have the correct quantifier structure, *but*, in some cases, the parts of the expressions after the quantifiers need something extra. This is because some of the text statements require X and Y to be distinct, but we have not enforced that yet in the logical expressions. For example, if our graph has no loops (which is a common restriction on graphs), then it is certainly the case that a vertex X cannot be adjacent to itself, so $\neg\mathrm{adj}(X, Y)$ is True, so $\exists X \, \exists Y : \neg\mathrm{adj}(X, Y)$ is always true in such cases since Y can just be the same as X. The English wording "Some two vertices ..." implies two *distinct* vertices, not just one vertex. So we have to do something to require X and Y to be different vertices. Similar issues arise in the second and third rows of the above table. We can correct these three expressions to give the following table.

Some two vertices are not adjacent.	$\exists X \, \exists Y : \neg(X = Y) \wedge \neg\mathrm{adj}(X, Y).$
Every pair of vertices is adjacent.	$\forall X \, \forall Y : \neg(X = Y) \Rightarrow \mathrm{adj}(X, Y).$
Some vertex is adjacent to all other vertices.	$\exists X \, \forall Y : \neg(X = Y) \Rightarrow \mathrm{adj}(X, Y).$
Every vertex has a neighbour.	$\forall X \, \exists Y : \mathrm{adj}(X, Y).$

When you have two variables in successive identical quantifiers, i.e., $\exists X \exists Y \cdots$ or $\forall X \forall Y \cdots$, they can be replaced by a *single* quantifier of the same type, applied to a *pair* of variables. So $\exists X \exists Y \cdots$ can be replaced by $\exists(X, Y) \cdots$, which is sometimes written $\exists X, Y \cdots$, and $\forall X \forall Y \cdots$ can be replaced by $\forall(X, Y) \cdots$, which is sometimes written $\forall X, Y \cdots$. So, in the above example, $\exists X \, \exists Y : \neg(X = Y) \wedge \neg\mathrm{adj}(X, Y)$ can be rewritten as $\exists(X, Y) : \neg(X = Y) \wedge \neg\mathrm{adj}(X, Y)$, and so on.

Exercise: Six degrees of separation

Suppose we have a predicate knows(X, Y) meaning that person X knows person Y.

It has been claimed that, in the human social network, the distance between any two people is at most 6.

Your challenge is to write this claim in predicate logic, using just the predicate knows.

3.7 Doing logic with quantifiers

If we know that

$$\forall X \ \text{blah}(X)$$

and obj is any specific object (in the domain of X),
then we can deduce that

$$\text{blah}(\text{obj}).$$

In other words, if the blah is True for all X, then it's certainly true for any specific value from the domain of X:

$$(\forall X \ \text{blah}(X)) \ \Rightarrow \ \text{blah}(\text{obj})$$

In similar vein, if it's True for a specific value from the domain of X, then its certainly True for *some* X.

$$\text{blah}(\text{obj}) \ \Rightarrow \ (\exists X \ \text{blah}(X))$$

Universal quantifiers and conjunction mix in a natural way.

$$\forall X \ (p(X) \wedge q(X)) \text{ is logically equivalent to } (\forall X \ p(X)) \ \wedge \ (\forall X \ q(X))$$

Similarly, existential quantifiers and disjunction mix naturally too.

$$\exists X \ (p(X) \vee q(X)) \text{ is logically equivalent to } (\exists X \ p(X)) \ \vee \ (\exists X \ q(X))$$

What about the logical relationship between . . .

$$\forall X \ (p(X) \vee q(X)) \ \text{ and } \ (\forall X \ p(X)) \ \vee \ (\forall X \ q(X))$$

. . . ? Are they equivalent, or is there an implication in one direction, or is there no direct logical relationship?

Similarly, what is the logical relationship between . . .

$$\exists X \ (p(X) \wedge q(X)) \ \text{ and } \ (\exists X \ p(X)) \ \wedge \ (\exists X \ q(X))$$

. . . ?

3.8 Relationship between quantifiers

If you have negation immediately to the left of a quantifier, then you may move it to the right of the quantifier (and its associated variable) provided you "flip" the quantifier as you do so ($\exists \longleftrightarrow \forall$).

$\neg\,\forall Y$ means the same as $\exists Y\,\neg$
$\neg\,\exists Y$ means the same as $\forall Y\,\neg$
So, for example, "Not all dogs are happy" is the same as "There exists an unhappy dog". To see this using logic:

$$\begin{array}{rll} & \neg\,\forall X\,(\text{dog}(X)\ \Rightarrow\ \text{happy}(X)) & \text{Not all dogs are happy} \\ = & \exists X\,\neg(\text{dog}(X)\ \Rightarrow\ \text{happy}(X)) & \text{(changing } \neg\,\forall \text{ to } \exists\,\neg\text{)} \\ = & \exists X\,\neg(\neg\text{dog}(X)\ \vee\ \text{happy}(X)) & \text{(see last lecture)} \\ = & \exists X\,(\neg\neg\text{dog}(X)\ \wedge\ \neg\text{happy}(X)) & \text{(by De Morgan)} \\ = & \exists X\,(\text{dog}(X)\ \wedge\ \neg\text{happy}(X)) & \text{There exists an unhappy dog} \end{array}$$

Similarly,

$\neg\,\exists Y$	means the same as	$\forall Y\,\neg$
$\neg\,\forall Y\,\neg$	means the same as	_____
$\neg\,\exists Y\,\neg$	means the same as	_____

Lecture 4

Proofs

4.1 Proof (recap)

A **proof** is a step-by-step argument that establishes, logically and with certainty, that something is true.

Every statement in a proof must be one of the following:

- something you already knew before the start of the proof, i.e.,
  - a definition,
  - an axiom (i.e., some fundamental property that is always taken to be true for the objects under discussion, such as the distributive law $x(y+z) = xy + xz$ for numbers),
  - a previously-proved theorem;

  or

- a logical consequence of some conjunction of previous statements.

When making a logical deduction from a previous statement in a proof, the fundamental principle is:

If you've previously established P
and also that $P \Rightarrow Q$
then you can deduce Q.

This principle is known as **modus ponens**. We usually don't bother to refer to it by that name when we're doing proofs, though, as we use it very often and is so natural (indeed, it's the very essence of logical deduction itself).

Exercise in Boolean algebra: Prove that $(P \wedge (P \Rightarrow Q)) \Rightarrow Q$ is a tautology.

It is often possible to deduce a statement Q from several previous statements, provided you have also already shown that the conjunction of those previous statements implies Q. This is an extension of modus ponens:

If you've previously established all of $P_1, P_2, \ldots, P_n$
and also that $(P_1 \wedge P_2 \wedge \cdots \wedge P_n) \Rightarrow Q$
then you can deduce Q.

4.2 Finding proofs

There is no systematic method for finding proofs for theorems. There are deep theoretical reasons for this, based on work in the 1930s (Gödel, 1931; Church, 1936; Turing, 1936).

Discovering proofs is an art as well as a science. It requires

- skill at logical thinking and reasoning
- understanding the objects you're working with
- practice and experience
- play and exploration
- creativity and imagination
- perseverence.

Although we can't give a recipe for discovering proofs, we will give some general advice on dealing with some common situations.

To prove subset relations, $A \subseteq B$ (where A and B are sets):

1. Take a general member of A, and give it a name. e.g., "Let $x \in A$"
2. Use the definition of A to say something about x.
3. Follow through the logical consequences of that,
4. ...aiming to prove that x also satisfies the definition of B.

See, e.g., Theorem 3. See also Exercise 1.2 (i.e., Exercise sheet 1, exercise 2).

To prove set equality, $A = B$ (where A and B are sets):

1. Prove $A \subseteq B$
2. Prove $A \supseteq B$

To prove numerical equality, $A = B$ (where A and B represent numbers):
If algebra can transform expression A to expression B, then that's good;
but if not:

1. Prove $A \leq B$
2. Prove $A \geq B$

4.3 Types of proofs

We now consider four types of proof, namely

- Proof by construction
- Proof by cases
- Proof by contradiction
- Proof by induction.

This list is not exhaustive.

Proofs can be quite individual in character and hard to classify, although many will follow one of the above patterns.

Many proofs are a mix of these types.

4.4 Proof by construction

...also known as **Proof by example**.

A **proof by construction** describes a specific object precisely and shows that it exists and that it satisfies the required conditions.

This can be used for some theorems that are existential statements, just asserting the existence of a certain object that satisfies some specified properties.

We saw a proof of this type in Lecture 1 (Theorem 1).

Mistakes to avoid:

- attempting a proof by construction for a *universal* statement.
  - If a theorem asserts that *every* object has some property, then it's not enough to just construct *one* object with the property.
- constructing an example that has the claimed property, thinking that a convincing example is enough to prove that the property holds for other objects too.
  - An example may be useful in illustrating a proof or explaining the key ideas of a proof. But it is not, of itself, a proof.

4.5 Proof by cases

...also known as **Proof by exhaustion** or (if lots of cases) "brute force".

To do a proof by cases,

- identify a finite number of different cases which cover all possibilities,
- prove the theorem for each of these cases.

  - These separate proofs of the cases may be thought of as "subproofs" of the proof of the theorem.

We saw an example in Lecture 1 (Theorem 2). That was not typical of proofs by cases, since the number of cases was so large (one case for each English dictionary word) and the number of possibilities to be covered was finite. More typically, a theorem asserts that every object from some infinite set has some property, and we divide the infinite set up into a small finite number of cases, and do a separate proof for each of the cases. In such situations, some of these cases must cover an infinite number of objects, and our reasoning in each case must be general enough to apply to all the objects covered by that case.

It's ok if cases overlap (although it might indicate that the proof includes some unnecessary duplication of effort and is inefficient). But they must be *exhaustive* in the sense that every object considered by the theorem must belong to (at least) one of the cases.

4.6 Proof by contradiction

. . . also known as "reductio ad absurdum".

A **proof by contradiction** works as follows.

- Start by assuming the negation of the statement you want to prove.
- Reason from this until you deduce a contradiction.
- This contradiction shows that the initial assumption was wrong.
- Therefore, the original statement must be true.

Our first example of a proof by contradiction sets the scene for some more important proofs later (Lecture 22).

Theorem 5
The statement "This statement is false" is not a proposition.

Proof.
Assume that it is a proposition.
Then it must be either true or false.
If it is true, then it is false.
If it is false, then it is true.
So, it is false if and only if it is true.
This is a contradiction.
So our assumption, that the statement is a proposition, must be false. □

Our second proof by contradiction is somewhat whimsical, but has the structure of many proofs of this type and illustrates some important points about such proofs.

Theorem 6
Every natural number is interesting.[1]

Proof.
Assume that not every natural number is interesting.
So, there exists at least one uninteresting number.
Therefore there exists a *smallest* uninteresting number.
But that number must be interesting, by virtue of having this special property of being the smallest of its type.
This is a contradiction, as this number is uninteresting.
Therefore our original assumption was wrong.
Therefore every natural number is interesting. □

"Every positive integer was one of his personal friends."
— J. E. Littlewood on Srinivasa Ramanujan, quoted by G. H. Hardy, *Srinivasa Ramanujan* (obituary), *Proceedings of the London Mathematical Society* **19** (1921) xl–lviii. See p. lvii.

Srinivasa Ramanujan
(1887–1920)
https://mathshistory.st-andrews.ac.uk/Biographies/Ramanujan/

Comments:
That "theorem" and "proof" is really just an informal argument, as the meaning of "interesting" is imprecise and subjective.
But it illustrates the structure of proof by contradiction.
It also illustrates the point that, if you know a set of objects is nonempty, then you can choose an element of *smallest* size in the set.
Often, the smallest object in a set may have special properties that can help you go further in the proof.

Can you always choose an object of *largest* size in a nonempty set?
Is every integer interesting?
Would the above proof still work, if applied to the set of all integers?

4.7 Prelude to induction

We will shortly be introducing Mathematical Induction, which is a proof technique that uses recursive decomposition of objects of some type (which might be strings, languages, expressions, ...) into a smaller object of the same type.

[1]See, e.g., Ch. 14 (Fallacies), in: Martin Gardner, *The Scientific American Book of Mathematical Puzzles and Diversions*, Simon & Schuster, New York, 1959.

To set the scene, consider De Morgan's Laws:

$$\neg(P \vee Q) = \neg P \wedge \neg Q \tag{4.1}$$
$$\neg(P \wedge Q) = \neg P \vee \neg Q \tag{4.2}$$

We proved these using truth tables in Lecture 2.
We also proved its extended form:

Theorem 7
For all n:

$$\neg(P_1 \vee \cdots \vee P_n) = \neg P_1 \wedge \cdots \wedge \neg P_n \tag{4.3}$$

We will now attempt another proof of the extended form. Our approach is to try to use the basic form of De Morgan's Law, (4.1), to prove its extended form, (4.3).

Theorem 8
For all n:

$$\neg(P_1 \vee \cdots \vee P_n) = \neg P_1 \wedge \cdots \wedge \neg P_n$$

***Attempted* "proof":**

$$\begin{aligned}
&\neg(P_1 \vee \cdots \vee P_n) \\
&\quad = \neg((P_1 \vee \cdots \vee P_{n-1}) \vee P_n) \qquad \text{(just grouping \ldots)} \\
&\quad = \neg(P_1 \vee \cdots \vee P_{n-1}) \wedge \neg P_n \qquad \text{(by De Morgan's Law, (4.1))} \\
&\quad = \ldots \text{ and so on and so on } \ldots \\
&\quad = \neg P_1 \wedge \cdots \wedge \neg P_n \qquad \text{Q.E.D.??}
\end{aligned}$$

This might be called a "nice try", but it is not a proof. The reader has to infer how to fill the gap. It's really shorthand for a "proof" whose length depends on n. But the length of a proof is not allowed to depend on one of the parameters in the statement being proved.

Although this is not a *proof* as it stands, it does give an intuitive *explanation* of why the theorem is true. Fortunately, we can turn its main idea into a proper proof. To do this, we introduce a new proof technique.

4.8 Proof by mathematical induction

Suppose you want to prove that a statement $S(n)$ holds for every natural number n.

One powerful technique for proving theorems of this type is the **Principle of Mathematical Induction**. It is widely used across computer science. It is used extensively to prove theorems about languages, models of computation, algorithms, graphs and many other types of structures. If you do FIT2004, you will also use it there.

Principle of Mathematical Induction

IF $\quad S(1)$ is true *(inductive basis)*

AND $\quad \forall k:$ **if** $\underbrace{S(k) \text{ is true}}_{\textit{inductive hypothesis}}$ **then** $S(k+1)$ is true *(inductive step)*

THEN

$\forall n: \quad S(n)$ is true.

The intuition behind Induction is that: by the inductive basis, we know $S(1)$ is true; since $S(1)$ is true and the inductive step tells us that $S(1) \Rightarrow S(2)$, we deduce that $S(2)$ is true; since $S(2)$ is true and the inductive step tells us that $S(2) \Rightarrow S(3)$, we deduce that $S(3)$ is true; and so on, forever. Mathematical Induction is a *single* logical principle that allows us to apply modus ponens repeatedly, forever, all the way along the number line, without "waving our hands" and saying "and so on".

We now use Induction to prove the extended form of De Morgan's Law, just using the basic form (4.1).

Theorem 9
For all n:

$$\neg(P_1 \vee \cdots \vee P_n) = \neg P_1 \wedge \cdots \wedge \neg P_n$$

Proof: We prove it by induction on the # of propositions.

Inductive basis:
It is trivially true when we have just one proposition:

$$\text{Left-Hand Side} = \neg P_1, \qquad \text{Right-Hand Side} = \neg P_1, \qquad \textit{so} \;\; \text{LHS} = \text{RHS!}$$

Inductive step:
Let $k \geq 1$.
Suppose it's true for k propositions:

$$\neg(P_1 \vee \cdots \vee P_k) = \neg P_1 \wedge \cdots \wedge \neg P_k$$

This our Inductive Hypothesis. We will use it later.

We have:

$$\begin{aligned}
&\neg(P_1 \vee \cdots \vee P_{k+1}) \\
&\quad = \neg((P_1 \vee \cdots \vee P_k) \vee P_{k+1}) && \text{(just grouping \dots)} \\
&\quad = \neg(P_1 \vee \cdots \vee P_k) \wedge \neg P_{k+1} && \text{(by De Morgan's Law, (4.1))} \\
&\quad = \neg P_1 \wedge \cdots \wedge \neg P_k \wedge \neg P_{k+1} && \text{(by Inductive Hypothesis)}
\end{aligned}$$

Conclusion:
So, by the Principle of Mathematical Induction, it's true for any number of propositions. $\square$

Here is another example, one that is used in very many introductions to Induction.

Theorem 10
For all n:

$$1 + \cdots + n = \frac{n(n+1)}{2}.$$

Proof: We prove it by induction on n.

Inductive basis:
When $n = 1$, LHS $=$ 1 and RHS $=$ 1(1+1)/2 $=$ 1. ✓

Inductive step:
Let $k \geq 1$.
Suppose it's true for $n = k$:

$$1 + \cdots + k = k(k+1)/2$$

We will deduce that it's true for $n = k + 1$.

$$\begin{aligned}
1 + \cdots + (k+1) &= (1 + \cdots + k) + (k+1) && \text{(preparing to use the inductive hypothesis)} \\
&= k(k+1)/2 + (k+1) && \text{(by the Inductive Hypothesis)} \\
&= (k+1)k/2 + (k+1) && \text{(algebra \ldots)} \\
&= (k+1)(k/2+1) \\
&= (k+1)(k+2)/2 \\
&= (k+1)((k+1)+1)/2
\end{aligned}$$

This is just the equation in the Theorem, for $n = k + 1$ instead of k.
So the inductive step is now complete. ✓

Conclusion:
Therefore, by the Principle of Mathematical Induction, the equation holds for all n. $\square$

Alternatively, we could make the inductive step go from $n = k - 1$ to $n = k$, instead of from $n = k$ to $n = k + 1$.

Slightly different proof: We prove it by induction on n.

Inductive basis:
When $n = 1$, LHS $=$ 1 and RHS $=$ 1(1+1)/2 $=$ 1. ✓

Inductive step:
Let $k \geq \mathbf{2}$. *[Note the change here!]*
Suppose it's true for $n = k-1$, where $k \geq 2$:

$$1 + \cdots + (k-1) = (k-1)k/2$$

We will deduce that it's true for $n = k$.

$$\begin{aligned} 1 + \cdots + k &= (1 + \cdots + (k-1)) + k \quad \text{(preparing to use the inductive hypothesis)} \\ &= (k-1)k/2 + k \quad \text{(by the Inductive Hypothesis)} \\ &= k(k-1)/2 + k \quad \text{(algebra \dots)} \\ &= k((k-1)/2 + 1) \\ &= k(k+1)/2 \end{aligned}$$

This is just the equation in the Theorem, for $n = k$ instead of $k-1$.
So the inductive step is now complete. ✓

Conclusion:
Therefore, by the Principle of Mathematical Induction, the equation holds for all n. □

Exercise:
Prove by induction that, for all n:

$$1^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

4.8.1 Induction and recursion

Proof by mathematical induction involves relating objects to simpler objects of the same type. This style of thinking also arises in recursion, which you have met in the context of algorithm design and programming. A recursive algorithm is one that calls itself, with the requirement that the recursive call must use arguments that are simpler, in some sense, than the original argument.

Induction and recursion are closely related. It is worth thinking about this relationship, to compare and contrast the two. One important aspect of their relationship is that induction is an especially appropriate tool for proving theorems about recursive algorithms.

4.8.2 Mathematical induction and statistical induction

The use of the term "Induction" here is different to the use of "induction" in statistics, which is the process of drawing general conclusions from data. Statistical induction is typically used in situations where there is some randomness in the data, and conclusions drawn can include some amount of error provided the conclusions drawn are significant enough that the error is unimportant. It is not a process of *logical deduction*; this, together with the presence of errors, means that it cannot be used as a step in a mathematical proof. By contrast, Mathematical induction is a rigorous and very powerful tool for logical reasoning in mathematical proofs.

Revision

Practise doing proofs!

- exercise sheets, textbooks, ...

Sipser, pp. 22–25.

For more about Srinivasa Ramanujan, see:

- https://mathshistory.st-andrews.ac.uk/Biographies/Ramanujan/
- R Kanigel, *The Man Who Knew Infinity: A Life of the Genius Ramanujan*, Washington Square Press, New York, 1991.
- *The Man Who Knew Infinity*, feature film, 2015.
- film review: G Farr, The Man Who Knew Infinity: inspiration, rigour and the art of mathematics, *The Conversation*, 24 May 2016.
https://theconversation.com/the-man-who-knew-infinity-inspiration-rigour-and-the-art-of-mathematics-59520

Lecture 5

Proofs: the Good, the Bad and the Ugly

In this lecture, we present a diverse collection of proofs (and even some "proofs") ranging from the divine down to the abysmal — some even plumbing the darkest depths of falsehood — and finishing on a few that are merely ugly.

The proofs in this lecture are not themselves examinable in this unit. But the *lessons about proofs* that they give are very relevant to the proofs that we will do.

5.1 Good proofs

The great Hungarian mathematician Paul Erdős used to say that the most beautiful proofs of theorems are recorded by God in *The Book*. We now give three proofs from *The Book*. These three proofs are all proofs by contradiction, although *The Book* contains many proofs of other types too.

Our first theorem is about prime numbers. Computer scientists should be familiar with the properties of prime numbers. They are of critical importance in information security, where they are used in public-key cryptosystems. They are also used in hash tables.

Theorem 11
(Euclid) There are infinitely many prime numbers.

Proof.

Suppose, by way of contradiction, that there are only finitely many primes.
Let n be the number of primes.
Let $p_1, p_2, \ldots, p_n$ be all the primes.
Define: $q := p_1 \cdot p_2 \cdot \cdots \cdot p_n + 1$.
This is bigger than every prime p_i. Therefore q must be composite.
Therefore q is a multiple of some prime.
But, for each prime p_i, if you divide q by p_i, you get a remainder of 1.
    So q cannot be a multiple of p_i.
So q cannot be a multiple of any prime. *This is a contradiction.*
So our initial assumption was wrong.
So there are infinitely many primes. $\square$

The proof of the next theorem is an excellent example of the structure of many of the kinds of proofs by contradiction that we will meet later in this unit.

Theorem 12
[(Pythagoras)] $\sqrt{2}$ is irrational.

Proof.
Suppose, by way of contradiction, that $\sqrt{2}$ is rational.
Then, by definition, there exist positive integers m, n such that $\sqrt{2} = \frac{m}{n}$.
Among all such pairs m, n, choose a pair that have no common factors.
Squaring each side of our equation gives: $2 = \frac{m^2}{n^2}$.
Rewrite slightly: $2n^2 = m^2$.
This tells us that m^2 is even. Therefore m is even. Therefore $m = 2k$ for some k.
Substituting this back in gives: $2n^2 = (2k)^2$, i.e., $2n^2 = 4k^2$, i.e., $n^2 = 2k^2$.
This tells us that n^2 is even. Therefore n is even.
Since m and n are both even, they both have a common factor, namely 2.
But we chose them so that they have no common factors. *This is a contradiction.*
Therefore our initial assumption, that $\sqrt{2}$ is rational, must be wrong.
Therefore $\sqrt{2}$ is *irrational.* $\square$

We need a definition for our next theorem.
Definition: A set is **countable** if *either*

- it is finite, *or*
- it can be put in one-to-one correspondence (i.e., bijection) with $\mathbb{N}$.

Some examples of countable sets:

- the set of FIT2014 students:

  { Dakṣiputra, Muḥammad, Ramon, Blaise, Gottfried, Nicole-Reine, Maria, Charles, Ada, George, Augustus, Annie, Henrietta, Radhanath, Williamina, Kurt, Rózsa, Alonzo, Alan, Konrad, Bill, Tommy, Hedy, Trevor, Maston, Noam, Winsome, Stephen, Grace, John, Péisù, Katherine, Margaret, Richard, . . . }

  This set is large and may take some time to count! But it is finite, therefore it is countable.

- the set of positive integers (natural numbers), $\mathbb{N}$.
  This set is countable, because there is a bijection between it and $\mathbb{N}$, i.e., between it

and itself.

$$\begin{array}{ccc}
 & & \mathbb{N} \\
 & & \\
1 & \longleftrightarrow & 1 \\
2 & \longleftrightarrow & 2 \\
3 & \longleftrightarrow & 3 \\
4 & \longleftrightarrow & 4 \\
5 & \longleftrightarrow & 5 \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots
\end{array}$$

- the set of integers, $\mathbb{Z}$.
  This may seem "bigger" than the set $\mathbb{N}$ of *positive* integers. It certainly *contains* $\mathbb{N}$, as a proper subset. But that does not prevent the existence of a bijection between them:

$$\begin{array}{ccr}
 & & \mathbb{Z} \\
 & & \\
1 & \longleftrightarrow & 0 \\
2 & \longleftrightarrow & 1 \\
3 & \longleftrightarrow & -1 \\
4 & \longleftrightarrow & 2 \\
5 & \longleftrightarrow & -2 \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots
\end{array}$$

  Note that there is no requirement for the bijection to pay attention to any of the internal mathematical structure of the sets. So it doesn't need to preserve the ordering of the integers when mapping them all across to the positive integers. The bijection pairs off the *elements* of the two sets, treating them purely as sets and nothing more.

  This example also demonstrates that you can have a bijection between two sets even though one is a proper subset of the other. This cannot happen for finite sets, but it can happen for infinite sets.

- the set Σ^* of all finite strings over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

$$\begin{array}{ccrcc}
 & & & & \Sigma^* \\
 & & & & \\
1 & \longleftrightarrow & \mathtt{1} & \longleftrightarrow & \varepsilon \\
2 & \longleftrightarrow & \mathtt{10} & \longleftrightarrow & \mathtt{a} \\
3 & \longleftrightarrow & \mathtt{11} & \longleftrightarrow & \mathtt{b} \\
4 & \longleftrightarrow & \mathtt{100} & \longleftrightarrow & \mathtt{aa} \\
5 & \longleftrightarrow & \mathtt{101} & \longleftrightarrow & \mathtt{ab} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots
\end{array}$$

> This particular bijection treats each string over $\{\texttt{a}, \texttt{b}\}$ as encoding a string of bits, using the correspondence $\texttt{a} \leftrightarrow 0, \texttt{b} \leftrightarrow 1$. This string of bits, with an extra leading `1`, becomes a binary number representing a positive integer.

Although the set of all *strings* over our alphabet is countable, the set of all *sets of strings* — i.e., the set of all *languages* — is not. We prove this in the next theorem.

We study this theorem for two reasons.

- As we look at various classes of languages in this unit, it will turn out that most (or all?) of the language classes we meet will actually be countable. For such a countable class of languages, this theorem shows that not all languages belong to that class. So it helps establish limits on what our computational models can do.
- The technique used in the proof of this theorem, called **diagonalisation**, is very important. It lies at the heart of the famous Proof of the Undecidability of the Halting Problem, which we will meet later in the semester.

Theorem 13
[(Cantor)] The set of *all languages* is *uncountable*.

Idea of proof: If the set of all languages were countable, there would be a bijection between natural numbers and languages. We illustrate how such a bijection might look in the table below, which you can think of as the top left corner of a table that extends infinitely far down and infinitely far to the right. The natural numbers are listed vertically on the left, and the arrows $\longleftrightarrow$ indicate the correspondence between them and some languages. We describe languages by using ticks ✓ and crosses ✗ to indicate whether each possible string does, or does not, belong to the language in that row. The headings along the top list all possible strings in order of increasing length (and in lexicographic order for strings of the same size). So you can determine whether a string is in a particular language by looking along the row for that language and down the column for that string, to see if the entry is ✓ or ✗.

So, for example, the language in the first row, which corresponds to 1 under our bijection, contains the strings ε, `aa`, `ba`, `bb`, `aaa`, and possibly others we have not shown; it does *not* contain the strings `a`, `b`, `ab`, `aab`, and possibly others too.

			ε	a	b	aa	ab	ba	bb	aaa	aab	…
1	$\longleftrightarrow$	L_1:	✓	✗	✗	✓	✗	✓	✓	✓	✗	…
2	$\longleftrightarrow$	L_2:	✗	✗	✓	✓	✗	✗	✗	✓	✓	…
3	$\longleftrightarrow$	L_3:	✓	✓	✗	✗	✗	✓	✗	✗	✓	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	…
m	$\longleftrightarrow$	L_m:	✓	✗	✗	✓	✗	✗	✓	✗	✗	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

We have only shown the initial portions of a few languages, namely L_1, L_2, L_3, and some later language L_m. Keep in mind that, *by assumption*, every language appears in the above table, somewhere in the infinitely long list $L_1, L_2, L_3, \ldots, L_m, L_{m+1}, \ldots$. So, for any language K, there is some row for which it is the language in that row; in other words, $\exists i : K = L_i$.

We now construct a new language $\hat{L}$ whose sole purpose is to be different to all the others! To begin with, we want $\hat{L}$ to be different to L_1. We see that the empty string ε belongs to L_1, so we insist now that it does *not* belong to $\hat{L}$. (See the circled entries in the ε-column in the next diagram.) We will not consider ε again in this process. We now move on to L_2 and look at the string `a`. We see that `a` does not belong to L_2, so we insist that it *does* belong to $\hat{L}$. (See the circled entries in the `a`-column in the diagram.) Next, we consider the third language L_3 and the third string `b`. Since `b` does not belong to L_3, we insist that it does belong to $\hat{L}$. And so we continue, working our way down the list of languages and along the list of strings, ensuring for each that $\hat{L}$ differs from the m-th language L_m by insisting that they disagree on the membership status of the m-th string. In effect, we are traversing the diagonal (from top left down towards bottom right) and ensuring that the membership status of strings in $\hat{L}$ is the opposite to what is indicated in the diagonal entries. This explains the name of this proof technique, "diagonalisation".

			ε	`a`	`b`	`aa`	`ab`	`ba`	`bb`	`aaa`	`aab`	$\ldots$
1	$\longleftrightarrow$	L_1:	✓	✗	✗	✓	✗	✓	✓	✓	✗	$\ldots$
2	$\longleftrightarrow$	L_2:	✗	✗	✓	✓	✗	✗	✗	✓	✓	$\ldots$
3	$\longleftrightarrow$	L_3:	✓	✓	✗	✗	✗	✓	✗	✗	✓	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ldots$
m	$\longleftrightarrow$	L_m:	✓	✗	✗	✓	✗	✗	✓	✗	✗	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
		$\hat{L}$:	✗	✓	✓	$\ldots$						

By construction, $\hat{L}$ differs from every language in the list. So there is no k such that $\hat{L} = L_k$. But, by our original assumption, every language must appear in our list of all languages. This gives our contradiction!

This account is intended to convey the main ideas in the proof of Theorem 13. But it is not, in itself, a *proof* of that Theorem. This is because we used a specific example to specify the construction of $\hat{L}$. Hopefully the example was helpful. Doing the construction for that example should enable you to see how to do it in general. But we did not specify — formally, precisely and rigorously — how to do that construction *in general*, in a way that covered all possibilities. We merely *illustrated* it, and hoped that this illustration got the main idea across. That's fine, for getting the main idea across. And hopefully our example-based explanation will make the proof we are about to give easier to read and understand. But the example-based explanation is not a proof.

We now give a proper proof, in written form.

Proof.

Suppose, by way of contradiction, that the set of all languages is countable.

Since we know it's not finite, there must be a bijection between $\mathbb{N}$ and $\{$all languages$\}$.
Let the members of the set of all languages be L_m, $m \in \mathbb{N}$.
Recall that the set of all finite strings is countable, so we can list them as x_n, $n \in \mathbb{N}$.
Define the language $\hat{L}$ as follows:

$$\forall n \in \mathbb{N} : x_n \in \hat{L} \Leftrightarrow x_n \notin L_n.$$

We have constructed $\hat{L}$ so that, for each n, it differs from L_n in whether or not it contains x_n.
So it differs from all languages. Yet it is a language! *This is a contradiction.*
So our initial assumption was wrong.
So the set of languages is uncountable. $\square$

It is worth satisfying yourself that this proof does indeed describe, in symbolic form, the same diagonalisation process we described earlier.

5.2 Bad proofs

Consider the truth table of $P \Rightarrow Q$: it is always True when P is false, regardless of Q. So the consequences of an error in a proof are drastic: from a falsehood, you can prove *anything*! This point was highlighted in a celebrated conversation involving the great mathematician G. H. Hardy.[1] The story goes that a colleague, John McTaggart (a noted philosopher), disagreed that you can prove anything at all from a falsehood. He defied Hardy to prove, from the false assertion "$2 + 2 = 5$", that he (McTaggart) was the Pope. Hardy immediately did so, using what amounted to the following argument.[2]

	$2+2 = 5$
Therefore	$4 = 5$
Therefore	$1 = 2$
Now,	$\lvert\{$ McTaggart, The Pope $\}\rvert = 2$.
Therefore	$\lvert\{$ McTaggart, The Pope $\}\rvert = 1$.
Therefore	McTaggart is the Pope.

We now give two incorrect proofs dressed up as correct proofs, and invite you to find the errors.

Firstly, we will try to liberate computer science students from the tyranny of trees, by "proving" that they do not actually exist.

"Theorem": Every graph has a cycle.

[1]Hardy's specialty was number theory, including the theory of prime numbers. He was the ultimate pure mathematician, taking delight in the lack of practical application of his research. Ironically, prime number theory is now very useful because of its applications including to information security and hash tables.

[2]recounted in: Harold Jeffreys, *Scientific Inference*, Cambridge University Press, 1931/1957/1973, Chapter I.

We restate the theorem as:
For all n: every graph on n vertices has a cycle.

“Proof”. We prove this by induction on the number of vertices.

1. Assume that **every graph on n vertices has a cycle**.
2. Let G be any graph on $n+1$ vertices.
3. Let v be a vertex of G. Obtain the graph $G - v$ by removing v, and all its incident edges, from G.
4. Now, the graph $G - v$ has n vertices.
5. By the **Inductive Hypothesis**, $G - v$ has a cycle.
6. But, since $G - v$ is a subgraph of G, any cycle in $G - v$ is also a cycle in G.
7. Therefore G has a cycle.
8. Therefore, by Mathematical Induction, the result is true for all n.
   So every graph has a cycle.

We now try to further liberate students from having to deal with *different* letters in strings by “proving” that, in fact, every string consists only of repetitions of a single letter.

Definition: A string is *uniform* if all its letters are identical.

- i.e., it consists entirely of `a`s or entirely of `b`s
- i.e., it's either $\mathtt{aa}\cdots\mathtt{a}$ or $\mathtt{bb}\cdots\mathtt{b}$

Now, it is commonly thought that not all strings are uniform. But we will now try to “prove”, by induction, that *all* strings are uniform!

“Theorem”: Every string over the alphabet $\{\mathtt{a},\mathtt{b}\}$ is uniform.

“Proof”. We prove this by induction on the string length n.

1. Inductive basis: when $n = 1$, the string can only be “`a`” or “`b`”, and these are each of the required form, so the “theorem” is true in this case.
2. Now assume $n \geq 2$, and suppose every string of length n is uniform.
3. Let w be any string of length $n+1$.
4. Let w_1 be the string obtained from w by deleting the *first* letter of w, and let w_2 be the string obtained from w by deleting the *last* letter of w.
5. Both w_1 and w_2 are of length n.

6. By the Inductive Hypothesis, both w_1 and w_2 must be uniform.

7. w_1 and w_2 overlap in $n-1$ letters. Since $n-1>0$, this means that the number of letters shared by w_1 and w_2 is nonzero. So w_1 and w_2 must each consist entirely of the *same letter*, i.e., either they both consist entirely of `a`s or they both consist entirely of `b`s.

8. It follows that w also consists entirely of `a`s or entirely of `b`s, so it is uniform too.

9. The result follows for all n, by Mathematical Induction.

5.3 Ugly proofs

We now come to proofs that are correct but ugly in some way.

Our first ugly proof is a variation on the proof of Theorem 3 in Lecture 1.

Theorem 3 again:
DOUBLEWORD ⊆ EVEN-EVEN.

Proof. Let $w \in$ DOUBLEWORD.
Assume w is not in EVEN-EVEN.
Then $w = xx$ for some word x.
So, $\#$ `a`'s in w $=$ $2 \times (\,\#$ `a`'s in $x\,)$, so it's even.
Also, $\#$ `b`'s in w $=$ $2 \times (\,\#$ `b`'s in $x\,)$, so it's even too.
This contradicts our assumption that w is not in EVEN-EVEN.
Therefore that assumption was wrong.
Therefore $w \in$ EVEN-EVEN. □

This proof is correct. You will recognise, within it, the proof given earlier. But there is some extra stuff too, which makes this new proof look like a proof by contradiction. However, the extra stuff serves no logical purpose; it is purely cosmetic, and makes the proof look longer and harder than it is. Let us remove the extra stuff, revealing again our original proof.

Proof. Let $w \in$ DOUBLEWORD.

Then $w = xx$ for some word x.
So, $\#$ `a`'s in w $=$ $2 \times (\,\#$ `a`'s in $x\,)$, so it's even.
Also, $\#$ `b`'s in w $=$ $2 \times (\,\#$ `b`'s in $x\,)$, so it's even too.

Therefore $w \in$ EVEN-EVEN. □

In fact, *any* proof can be dressed up as a proof by contradiction. Suppose you have a proof that some assertion A is true:

Proof.
[first line of your proof]
[second line of your proof]

⋮

⋮

[penultimate line of your proof]
Therefore A is true. $\square$

To make this into a proof by contradiction, we start out by assuming that A is false, then use the above proof to prove that it's true, and then observe that this contradicts our initial assumption that A is false, so our initial assumption is wrong and A must be true! Here is the proof with the extra stuff in grey.

Proof.
Assume, by way of contradiction, that A is false.
[first line of your proof]
[second line of your proof]

⋮

⋮

[penultimate line of your proof]
Therefore A is true.
This contradicts our initial assumption.
Therefore that initial assumption was wrong.
Therefore A is true. $\square$

This is still logically correct, but the extra stuff serves no purpose and makes the proof less clear. It also misleads the reader as to the true nature of the proof.

So, when you have a *direct* proof of your theorem, there's no need to dress it up as a proof by contradiction!

5.4 Ugly proofs?

We now look at proofs that are long and complicated and were later superseded by better proofs. Does this mean that the original proofs are ugly, or that they were not important?

A **colouring** of a graph G is a function that assigns a colour to each vertex of G such that *adjacent* vertices receive *different* colours.

- i.e., a function $f : V(G) \to \{\text{colours}\}$ such that $\forall u, v \in V(G) : u \sim v \Rightarrow f(u) \neq f(v)$.

A colouring is a k-**colouring** if the number of colours used is $\leq k$.

Graph colouring has many applications:

- scheduling (timetabling)

- compilers (register allocation)
- communications (frequency assignment)

A graph is **planar** if it can be drawn on the plane so that the curves representing the edges do not cross, except that their endpoints may coincide at a vertex if the curves represent edges that are incident at that vertex.

The question of whether or not every planar graph has a 4-colouring was originally proposed by Francis Guthrie, in 1852, inspired by his empirical observation that four colours were necessary and sufficient to colour a map of the counties of England. The question was communicated, via his brother Frederick, to Augustus De Morgan (who we met in Lecture 2), who promoted it in the mathematical community. For over a century it was one of the most famous unsolved problems in mathematics, defying serious attempts at proof by many great mathematicians. Then, in 1976, a proof was announced. It was unlike any major mathematical proof that had been published before. It has been accepted by the mathematical community but nonetheless generated considerable controversy.

Theorem 14
If G is planar then it has a 4-colouring.

A series of proofs of this theorem have appeared.

- very long proof using computer to check 1476 configurations spanning 400 pages.
  - K. Appel and W. Haken, Every planar map is four colorable. I. Discharging, *Illinois Journal of Mathematics* **21** (3) (1977) 429–490.
  - K. Appel, W. Haken and J. Koch, Every planar map is four colorable. II. Reducibility, *Illinois Journal of Mathematics* **21** (3) (1977) 491–567.
- long proof using computer to check 633 configurations
  - N. Robertson, D. Sanders, P. Seymour and R. Thomas, The four-colour theorem, *Journal of Combinatorial Theory, Series B* **70** (1997) 2–44.
- proof by Robertson *et al.* (1997) formalised and formally verified by computer
  - G. Gonthier, Formal proof — the Four Color Theorem, *Notices of the American Mathematical Society*, **55** (11) (Dec. 2008) 1382–1393.

Another example of an ugly proof (or partial proof) relates to solutions of polynomial equations.

Recall that, to solve quadratic equations $ax^2 + bx + c = 0$, you can use the formula $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Formulas also exist for cubic and quartic equations. *But* …

Abel-Ruffini Theorem
There is no general algebraic formula (using arithmetic operations, powers & roots) for the roots of polynomials of degree ≥ 5.

The first published purported proof took over 500 pages, and turned out to be incomplete.

- Paolo Ruffini, *Teoria generale delle equazioni, in cui si dimostra impossibile la soluzione algebraica delle equazioni generali di grado superiore al quarto*, Stamperia di S. Tommaso d'Aquino, Bologna, 1799.

A couple of decades or so later, a complete proof of just six pages was published.

- Niels Henrik Abel, *Mémoire sur les équations algébriques, ou l'on démontre l'impossibilité de la résolution de l'équation générale du cinquième degré*, Groendahl, Christiania (Oslo), 1824.

If a proof is incomplete, as well as long, then calling it "ugly" is understandable. Nonetheless, it may contain some important ideas that are used in later, complete proofs.

Is it fair to label a proof "ugly" just because it seems too long? Perhaps ugliness is in the eye of a beholder! It is often the case that the *first* proof of a theorem is longer and more complicated than necessary. At that stage, many of the required concepts and techniques have to be worked out from scratch. The person proving the theorem is moving into unknown territory and there is little by way of maps and landmarks to help. Later, they or others may revisit the proof and refine it, or they may come up with completely new proofs. Over time, an important theorem will attract enough attention that better proofs will probably be found. Eventually, maybe, a proof will be found that surely comes from The Book! The later proofs are much more likely to find their way into textbooks and be taught to students. But we should still give credit to the pioneers: their proofs may not last, but their contribution in showing what could be done and in opening up new territory for others to explore is crucial, and their achievement in constructing the *first* proof — ugly as it may seem to others later — is arguably greater than the later achievement of making a *better* proof.

G. H. Hardy famously wrote:[3]

> Beauty is the first test: there is no permanent place in the world for ugly mathematics.

But that does not mean that there is no role *at all* for "ugly" mathematics! It may play an important role in carving out a place to be permanently occupied later by work of greater beauty.

Martin Aigner and Günter M. Ziegler, *Proofs from THE BOOK*, Springer, Berlin, 1998 (1st edn.), 2010 (4th edn.).

[3] G. H. Hardy, *A Mathematician's Apology*, Cambridge University Press, 1940