FIT1058 Foundations of Computing

Course Notes for Weeks 1–4

Graham Farr, Alexey Ignatiev, and Rebecca Robinson

Faculty of Information Technology

Monash University

March 28, 2025

Monash University
Faculty of Information Technology

PREFACE

Welcome to FIT1058 Foundations of Computing!

Computation uses abstract formal models of real objects and systems. This unit lays the theoretical foundations for working with the most fundamental abstract models used in computer science. It will develop skills in abstract modelling, logical reasoning, rigorous proof, and mathematical analysis of computational methods and structures. These skills will be used throughout your degree in both theoretical and practical settings. The material we cover is considered core knowledge for computer science students and graduates by the major national and international professional associations.

This document contains the course notes for the unit. It will be released in instalments through the semester.

PREREQUISITES

This unit assumes successful prior study in mathematics at least to the standard of *Mathematical Methods units 3 & 4* in the Victorian Certificate of Education (VCE). That VCE subject in turn builds on mathematics studied earlier in high school.

The specific topics from school mathematics that we make most use of in this unit are:

- sets (including Venn diagrams, subsets, supersets, complement, union, intersection, counting subsets);
- functions (incl. their domain, codomain, rule), inverse functions, relations;
- sequences and series (arithmetic and geometric, finite and infinite);
- numbers (natural numbers, integers, rational numbers, real numbers, representations in a base, divisors, factorisation, composite numbers, primes, arithmetic, remainders);
- counting (incl. using addition and multiplication, permutations, combinations, binomial coefficients);
- probability (incl. using counting, combinations of events, conditional probability, random variables, probability distributions, binomial distribution, normal distribution, probability density functions);
- exponentiation and logarithms (which are ubiquitous in computer science);

- calculus (not a lot, but we make some use of limits and integrals, and derivatives can give useful insight even in situations where we don't make specific use of them).

We also make pervasive use of standard high-school algebra when working with polynomials and other functions.

We review parts of some of these topics in the pre-reading (see next section), but the pace and level still assumes prior knowledge. So it is important that you work to fill in any gaps or hazy areas in your knowledge of these topics.

PRE-READING

Sections whose numbers have superscript α contain *pre-reading*. For example:

§ 2.5$^{\alpha}$ Functions with multiple arguments and values

These pre-reading sections should be read and studied before the seminars on that topic. Some of them review work you did in school, but you should still read them, for several reasons: they establish the notation, terminology and other conventions we will use, which sometimes differ from those used in schools; they give some important computer science context to the concepts discussed, which you may not be aware of even if you have studied the concepts themselves before; and our experience is that most students benefit from some reminders and revision of this school material anyway. Other sections marked α may cover new material that is so fundamental to the topic to be discussed that reading about it before the seminar will significantly increase your ability to learn the material and master it.

The amount of pre-reading varies, depending on the topic. Some chapters spend a lot of their early sections reviewing concepts and topics covered in school (e.g., the chapters on sets and functions). These have more pages of pre-reading, but reading them should not take as long as reading completely new material. Other chapters contain material that is almost entirely new. These have fewer pages of pre-reading, but those pages will need to be read more slowly and carefully.

EXTRA MATERIAL

Sections whose numbers have superscript ω contain extra material that is beyond the FIT1058 curriculum. For example:

§ 2.11$^{\omega}$ Loose composition

This may include discussion of alternative approaches that we don't study, or more advanced aspects of the topic, or some historical background. The specific content introduced in these extra sections won't be on tests or exams in this unit. But reading them may still be of indirect benefit, by consolidating the material studied in the other sections.

ACKNOWLEDGEMENTS

Thanks to those who have given feedback on earlier versions of these Course Notes, including: David Albrecht, Mathew Baker, Annalisa Calvi, Michael Gill, Thomas Hendrey, James Sherwood, Joel Wills, and some anonymous student reviewers.

CONTENTS

1

SETS

The raw material of computation and communication is information. This takes many different forms, due to the great variety of things we might want to represent and the many different ways of representing them.

In this unit and in other units in your degree, you will learn about many different structures that are used to represent information in order to store it, communicate it or compute with it.

We will start with sets because these are among the simplest possible information structures. Most other structures can be defined in terms of sets, so sets are a foundational topic.

Sets are used extensively to define the *types* of objects that we compute with. When we work with a whole number, we say it is of Integer *type* because it belongs to the set of integers and satisfies the various laws and properties of integers. When we work with a string of characters, we might say that it is of String *type* because it belongs to the set of all strings over a suitable alphabet and satisfies the properties expected of strings. Many programming languages take careful account of the *types* of the objects they work with, and sets always underlie any notion of type.

1.1$^{\alpha}$ SETS AND ELEMENTS

A **set** is just a collection of objects, without order or repetition. The objects in a set are called its **elements** or **members**. To specify a set, we can just give a comma-separated list of them between curly braces. So the following are all sets:

$$\{\text{Harry, Ginny, Hermione, Ron, Hagrid}\}$$
$$\{42,\ -273.15,\ 1729,\ 10^{100}\}$$
$$\{\text{CSIRAC, Manchester Baby, EDSAC}\}$$
$$\{\}$$

In the last example, the set is empty. This set, simple as it is, is so fundamental that it has its own symbol, $\emptyset$, not to be confused with zero, 0 (which, in the early days of the computer industry, was often written with a slash through it to distinguish it from the letter O).

When we write a set by listing its elements in the above way, we will inevitably list the elements in some specific order. But different orders of writing do not affect the identity of the set. Our third set above contains three of the earliest computers ever built, but they are not listed there in chronological order. If we wrote them in chronological order, the set would be written

$$\{\text{Manchester Baby, EDSAC, CSIRAC}\}.$$

But it would still be *the same set*:

$$\{\text{CSIRAC, Manchester Baby, EDSAC}\} = \{\text{Manchester Baby, EDSAC, CSIRAC}\}.$$

We state that an object is an element of a set using the symbol $\in$:

$$\textit{object} \in \textit{set}.$$

For example,

$$\text{CSIRAC} \in \{\text{Manchester Baby, EDSAC, CSIRAC}\}.$$

To state that an object does not belong to a set, we use $\notin$. For example,

$$\text{SILLIAC} \notin \{\text{Manchester Baby, EDSAC, CSIRAC}\}.$$

1.2$^{\alpha}$ SPECIFYING SETS

We will be working with many sets that are far larger than these examples, and many will be infinite. So it is often not practical to write out all the elements. So we need a succinct way of specifying precisely the elements of a set. One way is to give a condition that, in general, is either true or false, with the members of the set being precisely those objects for which the condition is true. For example,

$$\{x : x \text{ is even}\}$$

is the set of all even numbers. The variable x here is simply a name for elements of this set, so that we can talk about them. The colon, ":", separates the name x from the condition on x that must be satisfied in order for it to be an element of this set. We read this as "the set of all x such that x is even". The choice of name, x, is not important; we could equally well write the set as

$$\{n : n \text{ is even}\}$$

In this definition, the reader will naturally infer that the variable (x or n) represents a whole number, since the concept of a number being even or not only applies to whole

numbers; it makes no sense, in general, for rational numbers or real numbers. But it is often preferable to spell out the kind of numbers we are talking about, so that the reader does not have to fill in any gaps in our description. In this example, we might also want to remove any doubt in the reader's mind as to whether we are working with integers in general or just natural numbers. So we might rewrite our definition as

$$\{x : x \in \mathbb{Z} \text{ and } x \text{ is even}\}$$

Set definitions of this type have the general form

$$\{name \ : \ condition\},$$

where *name* is a convenient name for an arbitrary member of the set and *condition* is a statement involving *name* which is true precisely when the object called *name* belongs to the set and is false otherwise.

It is a common convention to include, in our statement of the name (before the colon), a specification of a larger set that the object must belong to. For example, the set of even integers could be written

$$\{x \in \mathbb{Z} : x \text{ is even}\}.$$

This can be read as "the set of x in $\mathbb{Z}$ such that x is even" or "the set of integers x such that x is even". In general we can write

$$\{name \in larger\ set \ : \ condition\}.$$

It is necessary that the *condition* be precise and clear. To ensure this, it will often be specified in a formal symbolic way. It is ok to use English text in the condition *provided* it is used clearly and precisely. It is also important for the text to be succinct, subject to ensuring precision and clarity.

Another way to specify a set is to give a rule by which each member is constructed. For example, the set of even integers could be written

$$\{2n : n \in \mathbb{Z}\}$$

We read this as "the set of $2n$ such that n belongs to $\mathbb{Z}$" or "the set of $2n$ such that n is an integer". The rule is a formula for converting a named object into a member of the set, and after the colon we give a condition that the named object must satisfy in order for the formula to be used. Taking all objects that satisfy this condition, and applying the formula to each one of them, must give *all* members of the set. In general, we can write

$$\{rule \text{ expressed in terms of } name \ : \ condition \text{ on } name\}.$$

Since the curly braces are read as "the set of", it's ok to write, for example, {even integers} for the set of even integers or {people on Earth} for the set of all people on

Earth. This way of defining sets — using just English text between the braces — is fine when the English is completely precise and not too long. But it should be used with care, because of the risk of imprecision, and only works well for sets that can be described very simply.

People sometimes describe large sets by listing a few of their elements and expecting readers to spot the pattern and infer what the entire set is. For example, the set of even integers might sometimes be written as

$$\{0, 2, -2, 4, -4, 6, -6, \ldots\}$$

or

$$\{\ldots, -6, -4, -2, 0, 2, 4, 6, \ldots\}.$$

While this sort of description might help communicate ideas in an informal conversation, it is *not* a definition of the set, since it does not precisely specify which elements are in the set, but rather turns that task over to the reader by the use of "...". Informal descriptions have their place, and we will use them sometimes, but they are not formal definitions.

1.3$^{\alpha}$ CARDINALITY

The **size** or **cardinality** of a set is just the number of elements it has. If A is a set, then its size is denoted by $|A|$ or sometimes $\#A$. When a set is specified by just listing its elements, we can determine its size by just counting those elements, which can be done manually if the set is small enough. For the above examples, we have

$$\begin{aligned} |\{\text{Harry, Ginny, Hermione, Ron, Hagrid}\}| &= 5, \\ |\{42,\ -273.15,\ 1729,\ 10^{100}\}| &= 4, \\ |\{\text{CSIRAC, Manchester Baby, EDSAC}\}| &= 3, \\ |\emptyset| = |\{\}| &= 0. \end{aligned}$$

Determining the cardinality of a set is a fundamental skill in computer science. For example, if a set represents all the objects that an algorithm must examine in order to find one that is best in some sense, then determining the size of that set helps determine how long the algorithm will take. If another set represents all the data items that must be stored in some memory in a device, then determining its size helps determine how much storage will be used by the data. If yet another set represents all the possible outcomes of some computation, then its size is an indicator of how uncertain you are about that outcome before you do the computation.

We will often have to deal with very large sets, due to the huge amounts of data that computers work with. It is sometimes useful to focus on the *logarithm* of the size of a set.

1.4 SETS OF NUMBERS

Some sets are so commonly used that they have special names. We have already met $\emptyset$ which denotes the empty set. There are names for some fundamental sets of numbers:

$\mathbb{N}$	the set of positive integers
$\mathbb{N}_0$	the set of nonnegative integers
$\mathbb{Z}$	the set of all integers
$\mathbb{Q}$	the set of rational numbers
$\mathbb{R}$	the set of real numbers

Usually, when we work with these fundamental number sets, we are not *only* interested in them as plain *sets*: we may also be interested in the natural *order* they have (with $\leq$), and in some *operations* we can do with their elements (like $+$, $-$, $\times$ and more). So, the symbol $\mathbb{Z}$ stands for the *set* of integers (as above), but it is also used to represent that same set *together with* some selection of operations that we are interested in at the time. We will not dwell on this point further; it would be too fussy to start using different names for a number set depending on what operations on it were being used at the time.

To restrict any of these sets to only its positive or negative members, we can use superscript $+$ or $-$. So $\mathbb{Z}^+$ is another way of denoting $\mathbb{N}$, and $\mathbb{R}^-$ is the set of negative real numbers. To denote the set of nonnegative members of one of these sets of numbers, we combine superscript $+$ with subscript 0, as in $\mathbb{R}_0^+$ for the set of nonnegative real numbers (since the nonnegative real numbers are just the positive real numbers together with zero). Similarly, $\mathbb{Q}_0^-$ is the set of nonpositive rational numbers.

For intervals of real numbers, there is some standard notation to indicate which, if any, of the two endpoints of the interval are included:

notation	definition	terminology
$[a,b]$	$\{x \in \mathbb{R} : a \leq x \leq b\}$	closed interval
$[a,b)$	$\{x \in \mathbb{R} : a \leq x < b\}$	half-open half-closed interval
$(a,b]$	$\{x \in \mathbb{R} : a < x \leq b\}$	
(a,b)	$\{x \in \mathbb{R} : a < x < b\}$	open interval

Sometimes we want to restrict the contents of the interval to one of our other special sets of numbers. We will indicate this using a subscript on the interval notation. For example, if we only want integers within the interval $[a,b]$, we write $[a,b]_{\mathbb{Z}}$, which is an abbreviation for $[a,b] \cap \mathbb{Z}$.

1.5 SETS OF STRINGS

For data to be stored, processed, or communicated, it first needs to be encoded in some symbolic form. So we start by specifying the symbols we are allowed to use, which we

often call **characters** or **letters**. The set of allowed characters is called the **alphabet**. We only consider finite alphabets.

For example, depending on the context, we could use the 26-letter English alphabet $\{\mathsf{a},\mathsf{b},\mathsf{c},\ldots,\mathsf{y},\mathsf{z}\}$ (restricting here to words consisting entirely of lower-case letters, and ignoring accents and apostrophes), or the alphabet of the ten decimal **digits** $\{0,1,2,\ldots,9\}$, or the set of two **bits** $\{0,1\}$.

To represent data symbolically, we use *strings* of characters, where the characters all belong to some alphabet. If A is an alphabet, a **string over** A is a finite sequence of characters, each of which is drawn from that alphabet. In other words, a string is anything you can get by taking a character from your alphabet, then taking another (which might be the same one, or might not be) and putting it after the first, and then taking a third (which may or may not be one of the characters you have already used) and putting it after the first two, and so on, for as long as you like (but it must be finite). The **length** of a string is its number of characters. So, for example, the length of the string "babbage" is 7.

We allow the **empty string**, which has no characters. Because the empty string is, by its nature, impossible to see, there is a special symbol for it to help us write about it: this is ε, the Greek letter epsilon. The length of the empty string is 0.

So, every English word (written in lower case and without accents) is a string over the 26-letter alphabet $\{\mathsf{a},\mathsf{b},\mathsf{c},\ldots,\mathsf{y},\mathsf{z}\}$. Every positive integer can be represented in decimal notation as a string over the ten-digit alphabet $\{0,1,2,\ldots,9\}$, or in binary notation as a string over the two-bit alphabet $\{0,1\}$. Every file on your computer may be regarded as a string over an appropriate alphabet, which might be the latest Unicode alphabet of about 155,000 characters. Strands of DNA are modelled as strings over the alphabet $\{\mathrm{C},\mathrm{G},\mathrm{A},\mathrm{T}\}$.

If A is an alphabet and $k \in \mathbb{N}_0$, then A^k denotes all the strings of exactly k characters in which each character belongs to A. So A^1 is just the alphabet itself, and A^2 is the set of two-characters strings from that alphabet, and so on. For example, if $A = \{0,1\}$ then

$$A^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}.$$

For any alphabet A, the set A^0 is the set containing just the empty string: $A^0 = \{\varepsilon\}$. This is not to be confused with the empty set!

How many strings of length k over the alphabet A are there? In other words, what is $|A^k|$? If $k = 1$, then we are just counting strings of length 1, which is the same as the number of letters in the alphabet, so $|A^1| = |A|$. If $k = 2$, then we are counting strings of length 2. For each possible first letter, we have $|A|$ choices for the second letter, since there is no restriction on the second letter. Since each choice of first letter gives the *same* number of choices for the second letter, and since there are $|A|$ choices for the first letter, we find that the number of strings of length 2 is $|A| \times |A| = |A|^2$. If $k = 3$, then we have $|A|^2$ choices for the first two letters (as we just saw), with each such choice followed by $|A|$ choices for the third letter, with this number being independent of the

choice we made for the first two letters. So the total number of strings of length 3 is $|A|^2 \times |A| = |A|^3$. This reasoning extends to any value of k. So the number of strings over A of length k is given by

$$|A^k| \;=\; |A|^k.$$

We also write A^* for the set of all finite strings (of all possible lengths) over the alphabet A. This is always an infinite set (provided $A \neq \emptyset$). For $A = \{0,1\}$, we give a few of its smallest members:

$$A^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}.$$

1.6 SUBSETS AND SUPERSETS

A **subset** A of a set B is a set A with the property that every element of A is also an element of B. We write $A \subseteq B$. For example,

$$\begin{aligned}
\{\text{Manchester Baby}, \text{CSIRAC}\} &\subseteq \{\text{Manchester Baby, EDSAC, CSIRAC}\},\\
\mathbb{N} &\subseteq \mathbb{Z},\\
\mathbb{Q} &\subseteq \mathbb{Q},\\
\{0,1\}^3 &\subseteq \{0,1\}^*,\\
\emptyset &\subseteq [1,3).
\end{aligned}$$

We can also write $A \not\subseteq B$ to mean that A is not a subset of B, as in

$$\begin{aligned}
\{\pi, i, e\} &\not\subseteq \{\pi, i, \phi\},\\
[0,1) &\not\subseteq (0,1],\\
\mathbb{Q}^+ &\not\subseteq \mathbb{Z},\\
\{0,1,2\} &\not\subseteq \{0,1\}^*,\\
\{\emptyset\} &\not\subseteq \emptyset.
\end{aligned}$$

On a Venn diagram, we illustrate $A \subseteq B$ by drawing the set A entirely within the set B. See Figure 1.1.

We can think of the subset relation as specifying a logical implication: $A \subseteq B$ means that membership of A implies membership of B. We can write this using the implication symbol $\Rightarrow$:

$$\text{membership of } A \quad \Rightarrow \quad \text{membership of } B$$

In other words, for all x, if $x \in A$ then $x \in B$:

$$x \in A \quad \Rightarrow \quad x \in B.$$

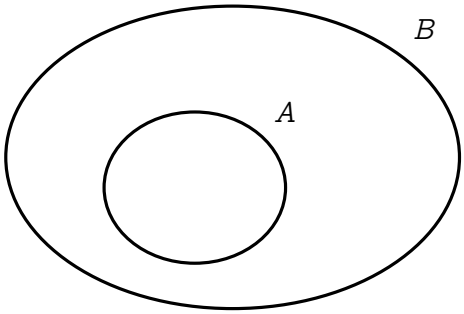


Figure 1.1: $A \subseteq B$.

This is sometimes read as "$x \in A$ only if $x \in B$".

Suppose we need to *prove* that $A \subseteq B$. This means we must prove that *every* member of A is also a member of B. In other words, we prove that every object that satisfies the condition for membership of A must also satisfy the condition for membership of B.

Every set is a subset of itself. Sometimes, we want to talk about subsets that are not the entire set. We say A is a **proper subset** of B if $A \subseteq B$ and $A \neq B$. So, if A is a proper subset of B, then there exists at least one element of B that is not also an element of A. We write $A \subset B$ to mean that A is a proper subset of B.

The empty set is a subset of every set (including itself). So, if B is any set, we can write $\emptyset \subseteq B$, and if B is nonempty then $\emptyset$ is a proper subset of B and we can write $\emptyset \subset B$.

If A and B are finite, then $A \subseteq B$ implies $|A| \leq |B|$. This explains the form of the subset symbol. But note that the converse does not hold in general: if $|A| \leq |B|$, it does not follow that $A \subseteq B$.

We say A is a **superset** of B, and write $A \supseteq B$, to mean that B is a subset of A. If in addition $A \neq B$ then A is a **proper superset** of B, written $A \supset B$.

All we are doing here is writing the subset relation in reverse. Similarly, we can write the implication in reverse too. So $A \supseteq B$ means that membership of A is implied by membership of B, which we can write using the reversed implication symbol:

$$\text{membership of } A \quad \Leftarrow \quad \text{membership of } B$$

In other words, for all x,

$$x \in A \quad \Leftarrow \quad x \in B.$$

This is sometimes read as "$x \in A$ if $x \in B$".

If we need to *prove* that $A \supseteq B$, then we just need to prove that $B \subseteq A$. We discussed proving subset relations above.

If we have *both* $A \subseteq B$ *and* $A \supseteq B$, then the two sets must actually be identical: $A = B$. The converse certainly holds too: if $A = B$ then $A \subseteq B$ and $A \supseteq B$. This suggests a way of proving that two sets A and B are equal: prove that each is a subset of the other. So the task of proving set equality is broken down into two subtasks, each requiring proof of a subset relationship, which is usually easier to prove than equality. In fact, this is a very common strategy for proving set equality.

We can think of $A \subseteq B$ and $A \supseteq B$ as giving logical implication in both directions: membership of A implies membership of B (because $A \subseteq B$), *and* is implied by membership of B (because $A \supseteq B$). More succinctly: membership of A is equivalent to membership of B, just as we would expect because $A = B$ here. Because we have implication in both directions, $\Rightarrow$ and $\Leftarrow$, it is convenient to put them together in a single symbol, $\Leftrightarrow$, which means that implication goes in both directions:

$$\text{membership of } A \quad \Leftrightarrow \quad \text{membership of } B.$$

In other words, for all x,

$$x \in A \quad \Leftrightarrow \quad x \in B.$$

This is often read as "$x \in A$ if and only if $x \in B$".

We will come across this "if and only if" wording often, so it is worth reflecting on its meaning. We think of it as stating two conditions that either both hold or both do *not* hold. In other words, either they both hold or neither of them holds. So they two conditions mean the same thing; they are *equivalent*.

In the case of "$x \in A$ if and only if $x \in B$", we have:

- the "**if** part", saying that "$x \in A$ if $x \in B$", which means (writing the membership statements the other way round) that "if $x \in B$ then $x \in A$", or equivalently, "$x \in B \ \Rightarrow \ x \in A$", or equivalently, "$x \in A \ \Leftarrow \ x \in B$";

- the "**only if** part", saying that "$x \in A$ only if $x \in B$", which means that "if $x \in A$ then $x \in B$", or equivalently, "$x \in A \ \Rightarrow \ x \in B$".

1.7 MINIMAL, MINIMUM, MAXIMAL, MAXIMUM

Suppose we have a set and we are interested in those subsets of it that have some specific property. For example, let B be a set of people. A **clique** in B is a set of people who all know each other. In other words, it's a subset $A \subseteq B$ such that, for each $x, y \in A$, person x and person y know each other. Given a set of people and their social links, we may wonder how "cliquey" they can be. To help us describe "peak cliques", we make a precise distinction between the adjectives "maximum" and "maximal".

- A **maximum clique** is a clique of maximum size; there is no larger clique.

- A **maximal clique** is a clique that is not a *proper* subset of any other clique.

Observe that these are *different concepts*, although they are related. Consider carefully the second of these, the concept of a maximal clique. Such a clique is not necessarily as large as the largest possible clique in B (although it might be). If A is a "maximal clique", then it's a clique with the extra property that, if we add any other person in B to the set, it's no longer a clique: that new person will be a stranger to at least one person already in A. So A cannot be enlarged while preserving the clique property. But that does not mean it is as large as any clique in B can be. There may be other quite different cliques that are even larger than A. So a maximal clique may be smaller in size than a maximum clique.

On the other hand, a maximum clique is also a maximal clique. A clique that is largest, in size, among all possible cliques in B cannot possibly be enlarged; it cannot possibly be a proper subset of another clique, because then the latter clique would be larger in size than the former one.

So,

$$\text{maximum} \implies \text{maximal}.$$

The reverse implication does not hold in general. (Typically, there are maximal cliques that are not maximum cliques. See if you can construct an example social network where this happens. But there do exist unusual situations where every maximal clique is maximum; can you construct one?)

We make this distinction between the meanings of "maximum" and "maximal" whenever we are talking about subsets with some property.

- A **maximum subset** with the property has largest size among all subsets with the property.
- A **maximal subset** is a subset with the property that is not a *proper* subset of any other subset with the property. In other words, it cannot be enlarged while still maintaining the property.

We make a similar distinction between "minimum" and "minimal".

- A **minimum subset** with some property has smallest size among all subsets with the property.
- A **minimal subset** with some property is a subset with the property that is not a *proper* superset of any other subset with the property. So, no proper subset has the property. In other words, if we remove anything from it, the property no longer holds.

In many situations in life, and especially if we are just talking about real numbers (rather than sets), this distinction between "maximum" and "maximal" is unnecessary (and likewise for "minimum" and "minimal"), and the terms are often treated as synonyms. What is the maximum numerical score you have ever made in your favourite game? You could replace "maximum" by "maximal" in this sentence, with no ambiguity (though it

would be less common wording in practice).[1] This is because real numbers are *totally ordered*; for every pair $x, y \in \mathbb{R}$, if $x \neq y$ then either $x < y$ or $y < x$. So, if a number has some property and cannot be increased while maintaining that property (i.e., it's maximal), then it's also the largest number with that property (i.e., it's maximum).

But the subset relation is different to the kind of order relation we are used to for real numbers. The subset relation does not give a total ordering; you can have two different sets A and B that are *incomparable* in the sense that $A \not\subseteq B$ and $B \not\subseteq A$, i.e., neither is a subset of the other. Such incomparability cannot occur among real numbers. But now that we are working with subsets, the terms "maximal" and "minimal" must be used with care, both in reading and writing. Unfortunately, they are often confused, even in technical publications in situations where the distinction matters.

From now on, we will mostly drop the underlining when using "maximum", "maximal", "minimum" and "minimal". But be observant about which suffix, -um or -al, is being used, and what the usage implies.

1.8 COUNTING ALL SUBSETS

If B is a finite set, with $|B| = n$ say, how many subsets does it have? A subset of B is determined by a choice, for each element of B, of whether or not to include it in the subset. Now, B has n elements, and for each of these we have two choices. These choices are independent, in the sense that making a choice for one element puts no restrictions whatsoever on the choices we may make for other elements. So the total number of choices we make is

$$\underbrace{2 \times 2 \times 2 \times \cdots\cdots\cdots \times 2 \times 2}_{\substack{\text{for each element of } B, \\ \text{choose between two options}}}$$

which is just $2^{|B|} = 2^n$. This tells us that the number of subsets of a set grows very quickly — in fact, grows *exponentially* — as the size of the set increases.

1.9 THE POWER SET OF A SET

The **power set** of a set B is the set of all subsets of B. We denote it by $\mathcal{P}(B)$. The observations of the previous paragraph tell us that

$$|\mathcal{P}(B)| = 2^n. \tag{1.1}$$

1 So, although a maximal clique is not necessarily a maximum clique, a maximal *size* clique is indeed just a maximum *size* clique. This is because, in "maximal/maximum size clique", the adjective "maximal/maximum" is applied to the *size*, which is a number (and therefore part of a total order), rather than to the set itself. Nonetheless, we will avoid applying the term "maximal" to sizes and other numbers, since there we can use "maximum" which is more common.

This is true even if B is empty, when $n = 0$ and $2^n = 2^0 = 1$, in keeping with the fact that $\emptyset$ has one subset, namely itself. This expression for $|\mathcal{P}(B)|$ explains the term "power set".

In algorithm design, we often need to find the "best" among all subsets of a set. Consider, for example, some social network analysis tasks, where we have a set of people and a set of pairs that know each other. Questions we might ask include: What is the largest clique, i.e., the largest set of people who all know each other? What is the largest set of mutual strangers? What is the smallest set of people who collectively know everyone? We could, in principle, solve these problems by examining all subsets of the set of people, or in other words, all members of its power set, provided we can easily determine, for each subset, whether or not it has the property we are interested in (being a clique, etc.). However, for reasonably large n, the number of sets to examine is prohibitive and the search would take too long. So we need to find smarter methods where we use the properties of networks and of the structures we are interested in to solve the problem without examining every single subset.

The power set of B is also often denoted by 2^B.

1.10 COUNTING SUBSETS BY SIZE USING BINOMIAL COEFFICIENTS

Sometimes we are focused on subsets of a specific size k. How many subsets of size k does a set B of size n have? This quantity is denoted by a **binomial coefficient**, written

$$\binom{n}{k}$$

and read as "n choose k" because we are interested in *choosing* k elements from n available elements. Between them, the binomial coefficients (taken over the full range of subset sizes, $k = 0, 1, 2, \ldots, n$) count every subset of B exactly once, so we already have

$$\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n-1} + \binom{n}{n} = 2^n.$$

This is an important and useful fact, but it does not yet give us a method for working out $\binom{n}{k}$. We now consider how to work this out.

We start with some simple cases. If $k = 0$, then we are choosing no elements at all, and this can be done in just one way, by doing nothing. (In this context, there's only one way to do nothing!) So, for all n,

$$\binom{n}{0} = 1.$$

At the other extreme, if $k = n$, then we choose *all* elements. Again, this can be done in only one way, because for each element of our set, we have no choice but to take it. So

$$\binom{n}{n} = 1.$$

Now suppose $k = 1$. We choose just one element from n elements, so we have n options:

$$\binom{n}{1} = n.$$

What about $k = n - 1$? This time, we are choosing one element *not* to include in our subset; once that choice is made, everything else is determined. So, again, we have n options:

$$\binom{n}{n-1} = n.$$

The symmetry we have seen here — firstly between $k = 0$ and $k = n$, and then between $k = 1$ and $k = n-1$ — is more general. To see this, observe that deciding which elements are included also determines which elements are excluded, and vice versa. The number of ways of choosing k elements to include in our subset is the same as the number of ways of choosing k elements to exclude from our subset, which in turn is just the number of ways of choosing $n - k$ elements to include. Therefore we have

$$\binom{n}{k} = \binom{n}{n-k}. \tag{1.2}$$

We now turn to methods for counting k-element subsets in general.

Although order does not matter within a set, sometimes it is easier to count them as if order did matter, and then correct for the overcounting. It's a bit like counting pairs of socks in your drawer by counting all the socks and then dividing by two.

So let's first count all ways of choosing k different elements, *in order*, from a set of n elements. Our first element can be any one of the n elements, so we have n choices. For our second element, we must not choose the first, but our choice is otherwise unrestricted, so we have $n-1$ choices. Our third element can be anything except the first two already-chosen ones, so we have $n-2$ choices. This process continues, so that when we come to choose the i-th element (where $1 \le i \le k$), we have $n - i + 1$ choices. At the very end, for our k-th element, we have $n - k + 1$ choices. So, altogether we have

$$\#\text{ ways to choose } k \text{ elements in order } = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1). \tag{1.3}$$

When $k = n$ we are just asking for the number of ways in which n elements can *all* be chosen in order, and that is just the **factorial** of n, written $n!$ and defined by

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1.$$

In fact, factorials allow a different way of writing (1.3):

$$\#\text{ ways to choose } k \text{ elements in order } = \frac{n!}{(n-k)!}. \tag{1.4}$$

Compare the number of arithmetic operations in each of these expressions, (1.3) and (1.4). It will be evident that the first expression, (1.3), is more efficient. The second is still important in understanding and using these counting problems, though.

We now return to our main aim of counting the *unordered* choices of k elements from n elements. Our ordered counting above will count each subset of size k some number of times. In fact, our sequence of choices was designed to count every possible ordering of the k elements exactly once. How many orderings are there? Since we drew these elements from a set (namely B), and each element of B is chosen at most once, all these chosen elements must be distinct. So there is no possibility of any of them looking identical to each other. So there are $k!$ ways to order the k elements, and therefore each subset of k elements gets counted $k!$ times by this process. Since this overcounting factor $k!$ is the same for all subsets of size k, we have

$$\#\text{ ways to choose } k \text{ elements } \underline{\text{in order}} = k! \cdot (\#\text{ ways to choose a } \underline{\text{subset}} \text{ of } k \text{ elements}).$$

It follows that

$$\begin{aligned}
&\#\text{ ways to choose a subset of } k \text{ elements} \\
&\qquad = \frac{1}{k!} \cdot (\#\text{ ways to choose } k \text{ elements in order}) \\
&\qquad = \frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)}{k!} \qquad \text{(using (1.3))} \qquad (1.5) \\
&\qquad = \frac{n!}{(n-k)!\,k!} \qquad \text{(using (1.4))}. \qquad (1.6)
\end{aligned}$$

Again, compare the number of arithmetic operations in the expressions (1.5) and (1.6), and consider which would be more efficient for computation. It is also worth thinking about the order in which the various multiplications and divisions are done. It makes no difference mathematically, but on a computer the order of operations can affect the accuracy of the result, because of limitations on the sizes and precision of numbers stored in the computer. In particular, the calculation works better, in general, if intermediate numbers used during the computation are not too large or small in magnitude. So, how can the computation be organised to best keep the sizes of those intermediate numbers under control?

A couple of special cases deserve special treatment because of their ubiquity in the analysis of algorithms and data structures.

$$\begin{aligned}\binom{n}{2} &= \frac{n(n-1)}{2},\\ \binom{n}{3} &= \frac{n(n-1)(n-2)}{6}.\end{aligned}$$

Counting subsets of a given size can also be done *recursively*. A **recursive** method for doing a task is one based on breaking the task down into simpler tasks *of the same type*. In this case, our task is to count the subsets, of a given size, in a given set. How can we reduce this to simpler subset-counting tasks?

Consider again our set B of size n and suppose we want to determine the number $\binom{n}{k}$ of k-element subsets of B. Let $b \in B$. We divide the k-element subsets of B into those that include b and those that do not. How many of each kind do we have?

Let's work through an example. Suppose $B = \{1,2,3,4,5\}$, so $n = 5$, and $k = 3$. So we want the number $\binom{5}{3}$ of 3-element subsets of B. (This example is small enough that you can just list these by hand, so please do so! It will be a handy check on what we are about to do.) Pick $b \in B$, say $b = 1$. Some 3-element subsets of B include 1, others do not. The point is that

$$\begin{aligned}\text{total \# 3-element subsets} \;=\;& \text{\# 3-element subsets that include 1}\\ &+\; \text{\# 3-element subsets that do not include 1.}\end{aligned}$$

So we have expressed the answer to our subset-counting problem in terms of answers to *simpler* subset-counting problems. Furthermore, these simpler subset-counting problems are of *the same type*:

- Observe that choosing a 3-element subset that includes 1 is really just choosing the rest of the subset that isn't 1, and we need exactly two of those non-1 elements to make up three elements altogether. So, counting 3-element subsets that include 1 is the same as counting 2-element subsets of the four-element set $\{2,3,4,5\}$. So

$$\text{\# 3-element subsets that include 1} \;=\; \binom{4}{2}.$$

- Observe that choosing a 3-element subset that does not include 1 is really just choosing three elements from among the non-1 elements. So, counting 3-element subsets that don't include 1 is the same as counting 3-element subsets of the four-element set $\{2,3,4,5\}$. So

$$\text{\# 3-element subsets that don't include 1} \;=\; \binom{4}{3}.$$

So

$$\begin{aligned}\binom{5}{3} &= \text{total \# 3-element subsets of } B \\ &= \quad \text{\# 3-element subsets that include 1} \\ &\quad + \text{\# 3-element subsets that do not include 1} \\ &= \binom{4}{2} + \binom{4}{3}.\end{aligned}$$

Now let's look at how it works in general.

- For those k-element subsets that include b, we choose any $k-1$ elements from among all elements of B other than b. So we must choose $k-1$ elements from $n-1$ available elements. This can be done in $\binom{n-1}{k-1}$ ways. (We also choose b, to complete our k-element subset, but there's only one way to do that!)
- For those k-element subsets that do not include b, we choose all k elements for our subset from among all elements of B other than b. So we now choose k elements from $n-1$ available elements. This can be done in $\binom{n-1}{k}$ ways.

The total number of k-element subsets is obtained by adding these two quantities together. So we have

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \tag{1.7}$$

So we can compute $\binom{n}{k}$ by doing two simpler computations of the same type (each with $n-1$ instead of n) and adding the results. Those two simpler computations can, in turn, be done in terms of other even simpler computations (with $n-2$), and so on. Eventually, the numbers get so small that we can use the simple cases $k=0$ and $k=n$, which are so simple that they can be solved without reducing them any further. We call these the **base cases**: they sit at the "base" of the whole reduction process, ensuring that the process does stop eventually, instead of just "descending forever".

It is worth comparing this method of computing $\binom{n}{k}$ with direct computation using (1.5) or (1.6).

This recursive method is especially useful when you want to compute $\binom{n}{k}$ for all n and k up to some limits. We start with base cases $\binom{n}{0} = \binom{n}{n} = 1$. The simplest case not covered by these is $\binom{2}{1}$, and applying (1.7) gives $\binom{2}{1} = \binom{1}{0} + \binom{1}{1} = 1+1 = 2$. The next simplest cases are $\binom{3}{1}$ and $\binom{3}{2}$. For the first of these, (1.7) gives $\binom{3}{1} = \binom{2}{0} + \binom{2}{1} = 1+2 = 3$; the second can be computed similarly, or even better, we can use symmetry: $\binom{3}{2} = \binom{3}{3-2}$, by (1.2), which we just calculated to be 3. Similar calculations for $n=4$, using the values we have just worked out, give $\binom{4}{1} = 4$, $\binom{4}{2} = 6$, and $\binom{4}{3} = 4$. And so on.

We can visualise the relation (1.7) using **Pascal's triangle**, shown symbolically in Figure 1.2a and with some actual values in Figure 1.2b. The binomial coefficients $\binom{n}{k}$ are arranged so that each one is the sum of the two immediately above it. In general, $\binom{n}{k}$ has $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$ just above it, in that order, with $\binom{n-1}{k-1}$ to its upper left and

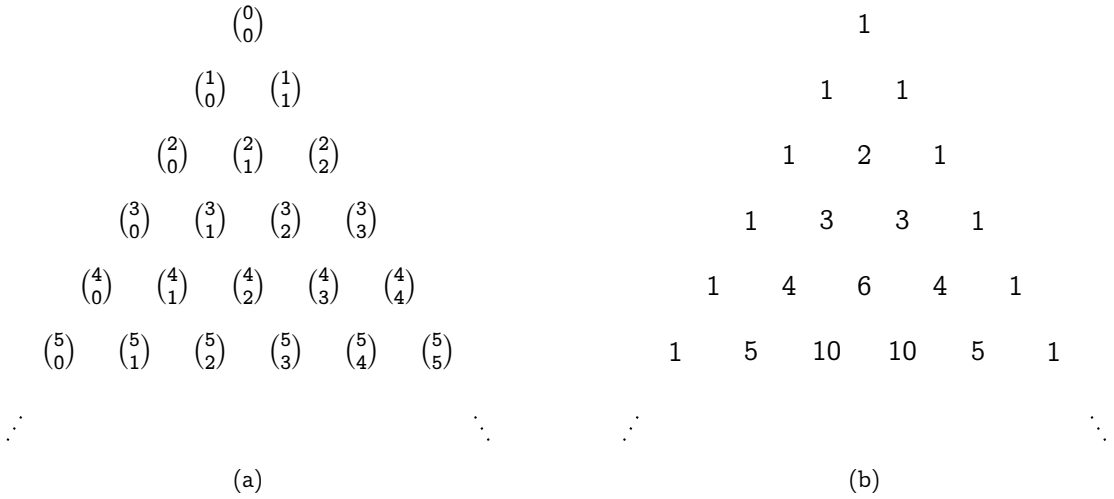


Figure 1.2: Pascal's triangle.

$\binom{n-1}{k}$ to upper right, and we saw in (1.7) that adding these two gives $\binom{n}{k}$. To take a specific example, consider $\binom{5}{2}$ in Figure 1.2a. We know from (1.7) that $\binom{5}{2} = \binom{4}{1} + \binom{4}{2}$, and we see in the triangular array in Figure 1.2a that $\binom{4}{1}$ and $\binom{4}{2}$ sit just above $\binom{5}{2}$. The actual values $\binom{5}{2} = 10$, $\binom{4}{1} = 4$ and $\binom{4}{2} = 6$ are shown in the corresponding positions in the triangular array in Figure 1.2b. The equation $\binom{5}{2} = \binom{4}{1} + \binom{4}{2}$ becomes $10 = 4 + 6$.

1.11 COMPLEMENT AND SET DIFFERENCE

Often, the sets we are discussing may all be subsets of some **universal set**, also called the **universe of discourse** or simply the **universe**.

For example, if we are working with various sets of integers (such as the even integers, or the odd integers, or the negative integers, or the primes), then the set $\mathbb{Z}$ of all integers can be used as the universal set. If we are working with sets of strings over the English alphabet A (such as the set of nouns, or the set of three-letter strings, or the set of names in the FIT1058 class list), then the set A^* of all strings over that alphabet may be a suitable universal set.

Suppose A is any set and U is some universal set, so that $A \subseteq U$. Then the **complement** of A, denoted by $\overline{A}$, is the set of all elements of U that are not in A. See Figure 1.3.

The notation $\overline{A}$ has the shortcoming that it does not include the universal set U, even though the definition depends on U. This is ok if the universal set has been clearly stated earlier or is clear from the context. But there is alternative notation that makes the dependence on U clear. We write $U \setminus A$ for everything (in the universal set) that is not in A. So $\overline{A} = U \setminus A$.

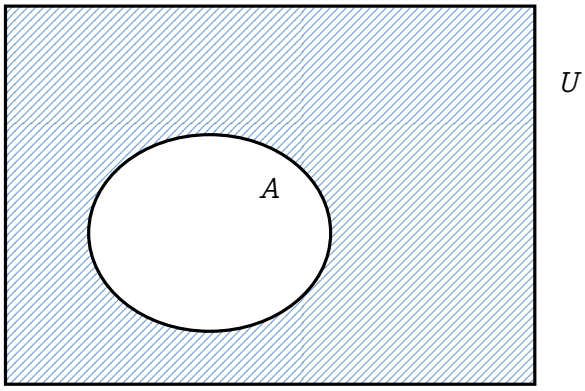


Figure 1.3: The complement $\overline{A}$, shaded.

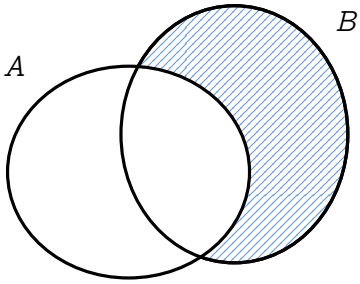


Figure 1.4: The set difference $B \setminus A$, shaded.

When A and U are finite sets, the size of $\overline{A}$ is given by

$$|\overline{A}| = |U| - |A|. \tag{1.8}$$

Taking the complement of the complement gives the original set:

$$\begin{aligned} \overline{\overline{A}} &= A, \\ U \setminus (U \setminus A) &= A. \end{aligned}$$

The operation $\setminus$ is called **set difference** and can be used between any two sets. So, if A and B are any sets, then $B \setminus A$ is the set of elements of B that are not in A:

$$B \setminus A = \{x \in B : x \notin A\}.$$

See Figure 1.4.

If $A \subseteq B$, then

$$|B \setminus A| = |B| - |A|. \tag{1.9}$$

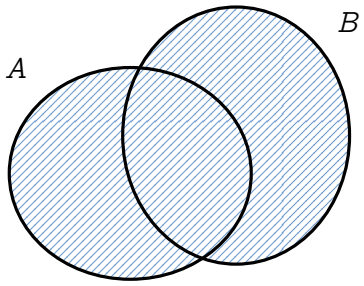


Figure 1.5: The union $A \cup B$, shaded.

In the special case when U is the universal set, this equation is just (1.8).

The size of the set difference does not satisfy (1.9) unless $A \subseteq B$. Why is this? How would you modify eq:size-of-set-difference-when-A-subset-B so that it covers any set difference $B \setminus A$? What extra information about the sets would you need, in order to determine $|B \setminus A|$?

The subset and superset relations are complementary in a precise sense:

$$A \subseteq B \iff \overline{A} \supseteq \overline{B}. \tag{1.10}$$

This gives us another approach to proving that $A \subseteq B$ (as well as the approach described on p. 8 in § 1.6). Instead of taking a general member x of A and proving that it also belongs to B, we could take a general nonmember of B and prove that it also does not belong to A. In other words, we show that, every time the condition for membership of B is violated, then the condition for membership of A must be violated too.

1.12 UNION AND INTERSECTION

The **union** $A \cup B$ of two sets A and B is the set of all elements that belong to at least one of the two sets:

$$A \cup B = \{x : x \in A \text{ or } x \in B\}. \tag{1.11}$$

The "or" here is inclusive in the sense that it includes the possibility that $x \in A$ and $x \in B$ are *both* true. This is how we will use the word "or" in set definitions and logical statements, unless stated otherwise at the time.

The union is illustrated in Figure 1.5.

The **intersection** $A \cap B$ of two sets A and B is the set of all elements that belong to both the two sets:

$$A \cap B = \{x : x \in A \text{ and } x \in B\}. \tag{1.12}$$

See Figure 1.6.

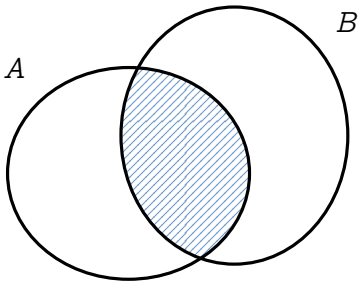


Figure 1.6: The intersection $A \cap B$, shaded.

Intersection gives us another way of writing set difference.

$$B \setminus A = \overline{A} \cap B.$$

When we count all the elements of A and all the elements of B, we are counting everything in either set except that everything in both sets is counted twice. Therefore

$$|A| + |B| = |A \cup B| + |A \cap B|. \tag{1.13}$$

This means that, if we know $|A|$ and $|B|$, then knowing either one of $|A \cup B|$ and $|A \cap B|$ will enable us to determine the other.

An important special case is when two sets A and B are **disjoint**, meaning that $A \cap B = \emptyset$. In that case, the size of the union is just the sum of the sizes of the members. The notation $\sqcup$ is sometimes used for the **disjoint union** of A and B, which is the ordinary union when the sets are disjoint, but is undefined otherwise:

$$A \sqcup B = \begin{cases} A \cup B, & \text{if } A \cap B = \emptyset; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

See Figure 1.7. There are some alternative symbols for disjoint union, the most common being obtained from the ordinary union symbol by placing a dot over it or $+$ inside it: $\dot{\cup}$ and $\uplus$.

When the disjoint union is defined, its size is just the sum of the sizes of the sets:

$$|A \sqcup B| = |A| + |B|. \tag{1.14}$$

We will use disjoint union occasionally, but mostly will focus on the normal, and more general, union.

The complement of the union of two sets is the intersection of their complements.

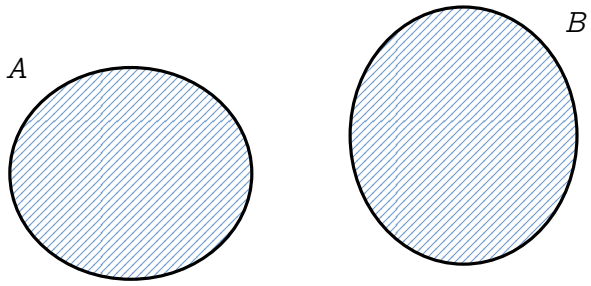


Figure 1.7: The disjoint union $A \sqcup B$, shaded. It is only defined when A and B are disjoint.

Theorem 1.

$$\overline{A \cup B} = \overline{A} \cap \overline{B}.$$

Proof.

$$\begin{aligned} x \in \overline{A \cup B} &\iff x \notin A \cup B \\ &\iff x \notin A \text{ and } x \notin B \\ &\iff x \in \overline{A} \text{ and } x \in \overline{B} \\ &\iff x \in \overline{A} \cap \overline{B} \end{aligned}$$

□

Similarly, the complement of the intersection of two sets is the union of their complements. We could prove this in a similar way, but we can prove it even more easily using Theorem 1.

Corollary 2.

$$\overline{A \cap B} = \overline{A} \cup \overline{B}.$$

Proof.

$$\begin{aligned} \overline{A \cap B} &= \overline{\overline{\overline{A} \cap \overline{B}}} \\ &= \overline{\overline{\overline{A} \cup \overline{B}}} \qquad \text{(by Theorem 1)} \\ &= \overline{A} \cup \overline{B}. \end{aligned}$$

□

Theorem 1 and Corollary 2 are known as **De Morgan's Laws for Sets**. They describe a *duality* between union and intersection. We will meet a similar duality later, when studying logic.

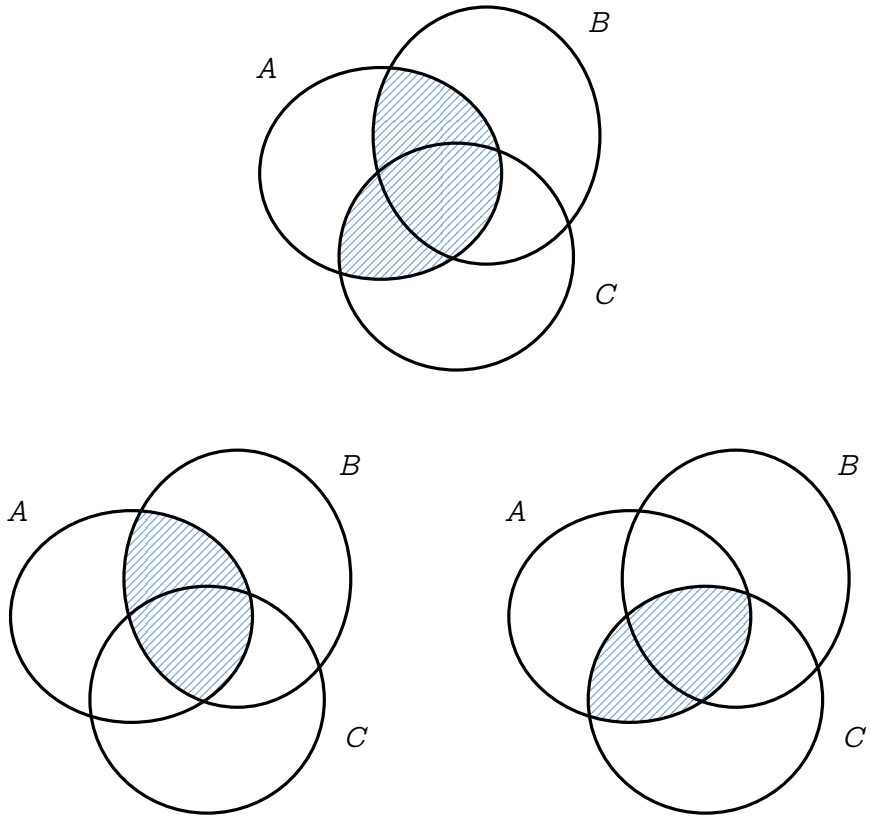


Figure 1.8: $A \cap (B \cup C)$ (top); compare with $A \cap B$ (left) and $A \cap C$ (right), and observe that $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

How do union and intersection interact *with each other*? If we take the union of two sets, and then the intersection with a third, what happens? What about taking an intersection first, then a union?

Consider $A \cap (B \cup C)$, shown in Figure 1.8. It is evident from the Venn diagrams that this is the same as $(A \cap B) \cup (A \cap C)$.

Now consider $A \cup (B \cap C)$. It is a good exercise to draw Venn diagrams to show how this relates to $(A \cap B) \cup (A \cap C)$.

In summary, we have the following.

Theorem 3. For any sets A, B and C,

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C), \tag{1.15}$$

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C). \tag{1.16}$$

□

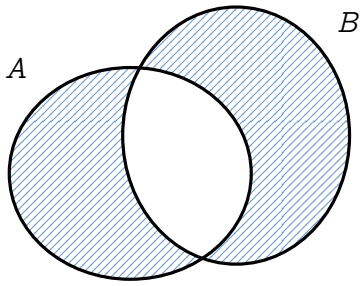


Figure 1.9: The symmetric difference $A \triangle B$, shaded.

Equations (1.15) and (1.16) are known as **the Distributive Laws for sets**. The first law, (1.15), is sometimes described as saying that "intersection distributes over union". This means that, when taking the intersection of A with a union of several other sets, we can "distribute" the intersection among those other sets, taking all the intersections separately, and then take the union. Similarly, the second law, (1.16), is sometimes described as saying that "union distributes over intersection". We will meet very similar Distributive Laws later, in logic. As for De Morgan's Laws, the algebra of sets will be seen to mirror the algebra of logic.

Although this may be a new Distribute Law for you, the notion of a Distributive Law should be familiar. You already know a Distributive Law for *numbers*. For any real numbers a, b, c, we have

$$a \times (b + c) = (a \times b) + (a \times c).$$

So multiplication distributes over addition. But, for numbers, *addition does not distribute over multiplication*: in general,

$$a + (b \times c) \neq (a + b) \times (a + c).$$

(There are some cases where equality just happens to hold here, but they are atypical and very rare.) So it is refreshing to work with sets, where the two operations are distributive in all possible ways!

1.13 SYMMETRIC DIFFERENCE

The **symmetric difference** $A \triangle B$ of A and B is the set of elements that are in *exactly one* of A and B.

$$A \triangle B = \{x : x \in A \text{ or } x \in B \text{ but not both}\}.$$

The "or" here is now exclusive in the sense that the possibility of belonging to both sets is excluded. See Figure 1.9.

There are other ways of writing the symmetric difference in terms of our other operations.

$$\begin{aligned} A \triangle B &= (A \setminus B) \cup (B \setminus A) && (1.17) \\ &= (A \cap \overline{B}) \cup (\overline{A} \cap B), \\ A \triangle B &= (A \cup B) \setminus (A \cap B). && (1.18) \end{aligned}$$

The symmetric difference of a set with itself is the empty set,

$$A \triangle A = \emptyset.$$

This is the only situation where the symmetric difference of two sets is empty. So the symmetric difference enables a neat characterisation of when two sets are identical.

Theorem 4. For any two sets A and B, they are identical if and only if their symmetric difference is empty.

Proof.

$$\begin{aligned} A = B &\iff A \subseteq B \text{ and } B \subseteq A \\ &\iff A \setminus B = \emptyset \text{ and } B \setminus A = \emptyset \\ &\iff (A \setminus B) \cup (B \setminus A) = \emptyset \\ &\iff A \triangle B = \emptyset \end{aligned}$$

□

The symmetric difference of two sets is the same as the symmetric difference of the complements.

Theorem 5.

$$A \triangle B = \overline{A} \triangle \overline{B}.$$

Proof.

$$\begin{aligned} A \triangle B &= (\overline{A} \cap B) \cup (A \cap \overline{B}) \\ &= (\overline{A} \cap \overline{\overline{B}}) \cup (\overline{\overline{A}} \cap \overline{B}) \\ &= \overline{A} \triangle \overline{B} \end{aligned}$$

□

1.14 CARTESIAN PRODUCT

If we have two objects a and b, then the **ordered pair** (a, b) consists of both of them together, in that order. You have used ordered pairs many times, for example as co-ordinates of points in the x, y-plane, or as rows in a table with two columns.

The **Cartesian product** $A \times B$ of two sets A and B is the set of all ordered pairs consisting of an element of A followed by an element of B:

$$A \times B = \{(a,b) : a \in A, b \in B\}.$$

So, if A is the set of all possible values of x, and B is the set of all possible values of y, then $A \times B$ is the set of all possible ordered pairs (x,y) of these values.

For example, if $A = \{\text{King}, \text{Queen}, \text{Jack}\}$ and $B = \{\clubsuit, \heartsuit\}$, then

$$A \times B = \{ (\text{King},\clubsuit), (\text{King},\heartsuit), (\text{Queen},\clubsuit), (\text{Queen},\heartsuit), (\text{Jack},\clubsuit), (\text{Jack},\heartsuit) \}.$$

The Cartesian product $\mathbb{R} \times \mathbb{R}$ is the set of all coordinates of points in the plane. If, for a given community of people, P is the set of all first (or personal) names and F is the set of all family names, then $P \times F$ is the set of all pairs (*first name*, *family name*). This would cover all pairings of names actually used by people in that community, but would typically include many unused pairings of names too.

If A and B are both finite sets, then the size of the Cartesian product is just the product of the sizes of the two sets:

$$|A \times B| = |A| \cdot |B|. \tag{1.19}$$

This is because we have $|A|$ possibilities for the first member of a pair, and $|B|$ possibilities for the second member, and these choices are made independently of each other. In more detail, each possibility for the first member gives $|B|$ possibilities for the second member, so the total number of pairs is

$$\underbrace{|B| + |B| + \cdots + |B|}_{|A| \text{ copies}}$$

which is just $|A| \times |B|$.

The Cartesian product of three sets gives the set of all triples of the appropriate kind:

$$A \times B \times C = \{(a,b,c) : a \in A, b \in B, c \in C\}$$

For example, $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ is the set of all coordinates of points in three-dimensional space.

More generally, if we have sets $A_1, A_2, \ldots, A_n$, then their Cartesian product is the set of n-tuples, or sequences of length n, in which the first member (or co-ordinate) is in A_1, the second is in A_2, and so on, with the i-th member belonging to A_i for all $i \in \{1, 2, \ldots, n\}$:

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n\}.$$

Again, if all the sets are finite then the size of the Cartesian product is the product of the sizes of all the sets:

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdot \cdots \cdot |A_n|.$$

If the sets $A_1, \ldots, A_n$ are all the same, then we can use an exponent to indicate how many of them are in the product:

$$\begin{aligned} A^n &= \underbrace{A \times A \times \cdots \times A}_{n \text{ factors}} \\ &= \{(a_1, a_2, \ldots, a_n) : a_i \in A \text{ for all } i \in \{1, 2, \ldots, i\}\}. \end{aligned}$$

For example,

$$\{0,1\}^3 = \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}.$$

In the special case when A is an alphabet (i.e., a finite set of characters), we often write n-tuples in A^n as strings of length n. So, for the binary alphabet $\{0,1\}$, we can write

$$\{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\},$$

as we did in § 1.5.

For another example, the sets of coordinates of points in two- and three-dimensional space are $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$ and $\mathbb{R}^3 = \mathbb{R} \times \mathbb{R} \times \mathbb{R}$, respectively. These are used extensively to model physical spaces, since the space around us is three-dimensional and we often deal with surfaces (terrain, paper, screens) that are two-dimensional. Higher-dimensional spaces are also useful. The set of coordinates in n-dimensional space is

$$\mathbb{R}^n = \underbrace{\mathbb{R} \times \mathbb{R} \times \cdots \times \mathbb{R}}_{n \text{ axes}}.$$

Spaces of more than three dimensions are hard to visualise, since the physical space we live in is only three-dimensional. But they are very useful and powerful. Models developed by machine learning programs can require millions or even billions of dimensions.

For the two smallest exponents, we have, for any set A,

$$A^0 = \emptyset, \qquad A^1 = A.$$

If A is finite, the size of A^n is just $|A|^n$.

1.15 PARTITIONS

A **partition** of a set A is a set of disjoint nonempty subsets of A whose union is A. These subsets are called the **parts** (or **blocks** or **cells**) of the partition.

Equivalently, a set of nonempty subsets of A is a partition of A if and only if every member of A belongs to exactly one of the parts.

Suppose $A = \{a, b, c\}$. One possible partition of A is

$$\{\ \{a, c\}, \{b\}\ \}.$$

This partition has two parts: $\{a, c\}$ and $\{b\}$. These parts are each nonempty, and disjoint, and their union is A, so the definition is satisfied. Another partition of A is

$$\{\ \{a\}, \{b\}, \{c\}\ \}.$$

This partition has three parts, namely the three sets $\{a\}$, $\{b\}$, $\{c\}$. At the other extreme, we have a partition of A with just one part:

$$\{\ \{a, b, c\}\ \}.$$

Our set A is small enough that we can list all its five partitions:

partition	# parts
{ {a, b, c} }	1
{ {a,b}, {c} }	2
{ {a,c}, {b} }	2
{ {b,c}, {a} }	2
{ {a}, {b}, {c} }	3

There are several ways in which a collection of subsets of a set can fail to be a partition. For our set $A = \{a, b, c\}$, the collection $\{\{a, b\}, \{c\}, \emptyset\}$ fails because one of its members is empty. The collection $\{\{a, b\}, \{b, c\}\}$ fails because its members are not all disjoint, in particular $\{a, b\} \cap \{b, c\} = \{b\} \neq \emptyset$, so b belongs to two members of the collection instead of just one. The collection $\{\{a\}, \{b\}\}$ fails because the union of the collection's members is not the entire set A, i.e., $\{a\} \cup \{b\} = \{a, b\} \neq A$; in particular, c does not belong to any members of this collection.

Partitions have many applications. Consider, for example, classification. Suppose that A is a collection of plant specimens. We would like to classify the specimens according to their species: specimens from the same species are grouped together, while those from different species are kept in separate groups. (Some groups may have just one specimen, if there is no other specimen of the same species.) These groups form the parts of a partition of A, with each part corresponding to one of the species represented in the collection. Finding such classifications, from data obtained from specimens, is a major topic in machine learning.

The number of partitions of a finite set grows rapidly as the size of the set increases.

set size, n	1	2	3	4	5	6	7	8	9	10
# partitions	1	2	5	15	52	203	877	4140	21147	115975

We can also talk about partitions of infinite sets. For example, $\{\{\text{even numbers}\}, \{\text{odd numbers}\}\}$ is a partition of the set of nonnegative integers; this partition has two parts. Consider also the following partition of the set A^* of all strings over a finite alphabet A:

$$\{A^n : n \in \mathbb{N}_0\}.$$

This partition has infinitely many parts, one for each $n \in \mathbb{N}_0$. The parts are the sets of all strings of a given length.

Every set A has two partitions that might be thought of as "extreme", but in opposite directions.

- The **coarsest partition** of A is the partition $\{A\}$ which has just one part, namely the entire set A itself. In effect, everything in A is "lumped together".
- The **finest partition** of A is the partition $\{\{a\} : a \in A\}$ which has one part for each element of A, and each part contains only that element. If A is finite, then this partition has $|A|$ parts; if A is infinite, then this partition has infinitely many parts. Every part of this partition is as small as a part of a partition can be. In effect, all the elements of A are "kept apart" from each other.

If A has just one element, then the coarsest and finest partitions are the same, but if A is larger then they are different. If A has just two elements, then these are the only partitions of A, but if A is larger, then it has many other partitions too, with all the others being in a sense intermediate between these two extreme partitions.

1.16 EXERCISES

1. Why does the set difference only satisfy (1.9) when $A \subseteq B$? How would you modify (1.9) so that it covers any set difference $B \setminus A$? What extra information about the sets would you need, in order to determine $|B \setminus A|$?

2. We mentioned at the start of this chapter that sets are used to define *types* of objects in many programming languages. For example, in C, the statement

```
int  monthNumber;
```

declares that the variable `monthNumber` has type `int`. The declaration also assigns a piece of memory to the variable, to contain the values that the variable has during the computation. Similarly, the statement

```
char  monthName[10];
```

is C's way of declaring that the variable `monthName` is a *string* of at most 9 characters; again, a piece of memory is allocated to it as well.

Let Int be the set of possible values for a variable of type `int`. Similarly, let String be the set of possible values for a variable that is declared to be a string of at most 9 letters.

In C, we can combine declarations in order to create more complex types.

(a) The following statement creates a new type, called `aNewType`, for representing objects consisting of any int followed by any string; it also sets aside consecutive pieces of memory, so that the int is followed by the string in memory. It also declares the variable `monthBothWays` to be of this type.

```
struct  aNewType  {
    int  year;
    char  monthName[10];
}  monthBothWays;
```

Using the sets Int and String, together with a standard set operation, what set is represented by the type `aNewType`?

(b) The following statement creates another new type, called `anotherNewType`, for representing objects that can be either an int or a string. It sets aside a piece of memory that is large enough to contain either an int or a string; at any one time, it will contain just one of these. It also declares the variable `monthEitherWay` to be of this type.

```
union  anotherNewType  {
    int  year;
    char  monthName[10];
}  monthEitherWay;
```

Using the sets Int and String, together with a standard set operation, what set is represented by the type `anotherNewType`?

3. Suppose A and B are subsets of some universal set U.

(a) If $A \subseteq B$, what is $\overline{A} \cup B$?

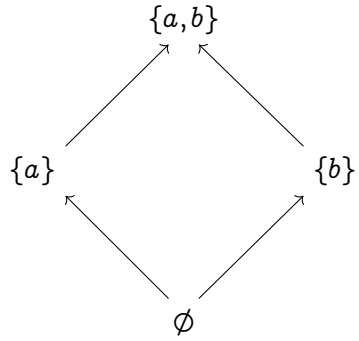
(b) If $A \not\subseteq B$, what can you say about $\overline{A} \cup B$?

(c) Complete the following: $A \subseteq B$ if and only if $\overline{A} \cup B =$ ___.

(d) Devise another equivalent condition for $A \subseteq B$ involving intersection instead of union.

(e) Devise another equivalent condition for $A \subseteq B$ involving set difference.

4. Consider the following diagrams. The one on the left shows the set $\{a\}$ and its sole subset, $\emptyset$. The one on the right shows $\{a,b\}$ and all its subsets.



The requirements of this diagram are:

- Sets are just represented by writing them (in their usual textual form, with elements listed between curly braces). These are not Venn diagrams.
- Smaller sets are lower, on your page, than larger sets.
- Sets of the same size are shown on the same horizontal level.
- The arrows indicate when a lower set is a subset of another set that has just one extra element. If $X \subset Y$ and $|Y| = |X| + 1$ then there is an arrow from X to Y.
- The diagram is as neat and symmetric as possible.

(a) Draw a diagram of this type that shows all subsets of $\{a, b, c\}$ and the subset relation among them.

(b) For every pair of sets X, Y such that $X \subset Y$ and $|Y| = |X| + 1$, label the corresponding arrow in your diagram by the sole member of $Y \setminus X$.

(c) Suppose you are now liberated from the requirement to draw your diagram on a medium of only two dimensions such as paper or a computer screen. How could you draw this diagram in three dimensions in a natural way?

(d) For a set of n elements, how many sets and how many arrows does a diagram of this type have?

(e) For each element of the n-element set considered in (d), how many arrows are labelled by that element (if we label them as in (c))?

(f) In such a diagram, suppose we have two sets X, Y that satisfy $X \subseteq Y$. How many *directed paths* are there from X to Y? Give an expression for this.

- A **directed path** is a path along arrows in which all arrows are directed forwards; you can't go backwards along an arrow. Paths are counted as different as long as they are not identical; they are allowed to have some overlap.

(g) With X, Y as in (f), what is the maximum number of *mutually internally disjoint* paths from X to Y? Describe this

- An **internal set** on a path is a set on the path that is not the start or end of the path, i.e., it's not X or Y. Two paths are **internally disjoint** if no internal set on either path appears anywhere on the other path. If we have a collection of some number of paths (possibly more than two), then the paths are **mutually internally disjoint** if every pair of paths in the collection are internally disjoint.

(h) Explain how to use the diagram to find, for any two sets in it, their union and intersection.

5. Given $U = \{e_1, e_2, \ldots, e_n\}$, the **characteristic string** of a subset A of U is the string of n bits $b_1 b_2 \cdots b_n$ where the i-th bit indicates whether or not e_i belongs to A:

$$b_i = \begin{cases} 1, & \text{if } e_i \in A; \\ 0, & \text{if } e_i \notin A. \end{cases}$$

Write down the characteristic string of each of the subsets of a set of three elements. List them, one above the other, so that each differs from the one above it in just one bit.

See if you can extend this to subsets of a set of n elements.

This has algorithmic applications. Suppose we want to search through all subsets of a set, by looking at each of their characteristic strings in turn. If each characteristic string differs from its predecessor in only one bit, then moving from one characteristic string to the next requires fewer changes than may be required otherwise, which saves time.

6. In a Venn diagram, the closed curves representing the sets together divide the plane into regions. A single set divides the plane — or the portion of the plane within the rectangular box representing the universal set, if that is shown in the diagram — into two regions, its interior and exterior. See Figure 1.3, where the regions correspond to A and $\overline{A}$ (with the latter shaded in that particular diagram, as it was being used to explain the complement, but that does not have to be done in general). Two intersecting sets, represented by two closed curves, divide the plane into four *basic regions*. (See Figure 1.4, Figure 1.5, and Figure 1.6.) If the sets are A and B, then the basic regions correspond to $\overline{A \cup B}$, $A \cap B$, $A \setminus B$ and $B \setminus A$. Three sets can be drawn to divide the plane into eight regions.

(a) Label each basic region of a Venn diagram for three sets A, B, C with appropriate intersections of sets.

A **general Venn diagram** for n sets $A_1, A_2, \ldots, A_n$ is one in which every possible intersection of the form

$$(A_1 \text{ or } \overline{A_1}) \cap (A_2 \text{ or } \overline{A_2}) \cap \cdots \cap (A_n \text{ or } \overline{A_n})$$

corresponds to one of the basic regions of the diagram. This is intended to be an intersection of i sets, the i-th of which is either A_i or $\overline{A_i}$. For each i, there is a choice of two options, so the total number of basic regions is 2^n.

For example, the Venn diagrams in each of these figures are general Venn diagrams: Figure 1.3, Figure 1.4, Figure 1.5, Figure 1.6, Figure 1.8, Figure 1.9.

But the Venn diagram in Figure 1.1 is not a *general* Venn diagram, because there is no region for $A \cap \overline{B}$ (which is fine in that case, because it is illustrating $A \subseteq B$). The Venn diagram in Figure 1.7 is not a *general* Venn diagram, because there is no region for $A \cap B$ (which is fine in that case, because it is illustrating the disjoint union $A \sqcup B$).

(b) What is the maximum number of sets for which a general Venn diagram can be drawn in which all the sets are circles of the same size?

(c) Draw a general Venn diagram in the plane for four sets. You can use closed curves other than circles.

(d) How could you draw a *three-dimensional* general Venn diagram using four sets, each represented as a sphere?

Drawing Venn diagrams (not necessarily general ones) to illustrate relationships among a collection of sets is one of many topics studied in *information visualisation*. There are many different criteria to aim for, including using only simple curves, not having too much variability in sizes of the basic regions, and symmetry. These criteria can conflict with each other. Algorithms have been developed and theorems have been proved.

7. Prove that if $A \subseteq B$ then $\mathcal{P}(A) \subseteq \mathcal{P}(B)$.

8. Consider the sequence of binomial coefficients $\binom{n}{r}$ with n fixed and r going from 0 to n:

$$\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots\ldots, \binom{n}{n-1}, \binom{n}{n}.$$

(a) Using one of the formulas for $\binom{n}{r}$, prove that these binomial coefficients increase as r goes from 0 to $\lfloor n/2 \rfloor$ and then decrease as r goes from $\lceil n/2 \rceil$ to n.

- Here, $\lfloor n/2 \rfloor$ is the "floor" of $n/2$, which is the greatest integer $\leq n/2$. If n is even, this is just $n/2$ itself, but if n is odd (which means $n/2$ is not an integer), its floor is the integer $(n-1)/2$.

  Similarly, $\lceil n/2 \rceil$ is the "ceiling" of $n/2$, which is the least integer $\geq n/2$. If n is even, this is just $n/2$ again, but if n is odd, its ceiling is the integer $(n+1)/2$.

- So, what we are saying here is that:
  - If n is even, $\binom{n}{r}$ increases as r goes from 0 to $n/2$, then it decreases as r goes from $n/2$ to n.
  - If n is odd, $\binom{n}{r}$ increases as r goes from 0 to $(n-1)/2$, then it decreases as r goes from $(n+1)/2$ to n.
  - In the odd case, we haven't yet said anything about what happens when r goes from $(n-1)/2$ to $(n+1)/2$. See if you can work that out too.

(b) Now prove that, for every positive integer r in the range $1 \leq r \leq n-1$,

$$\binom{n}{r}^2 > \binom{n}{r-1}\binom{n}{r+1}.$$

This is an important property and means that the sequence is said to be *strictly log-concave*. (If the inequality is just $\geq$ instead of $>$, then it's *log-concave*.)

(c) Suppose you and a friend are considering a fixed set of size n. Suppose you get to choose an ordered pair of subsets of size r of the set of size n, with no restriction at all (so your two sets are allowed to overlap, or be disjoint, or be identical, or whatever; the only rule is that they both have to have size r. Suppose also that your friend gets to choose one subset of size $r-1$ and another of size $r+1$, with again no restriction on the sets apart from these size requirements. Who has more options, you or your friend? Does the answer depend on n and r in any way? If so, how? If not, why not? Is there any situation where the numbers of choices that you and your friend have are the same?

9. Express $|A \cup B|$ in terms of $|A|$, $|B|$, $|A \cap B|$.

10. Express $|A \cap B|$ in terms of $|A|$, $|B|$, $|A \cup B|$.

11. Express $|A \cup B \cup C|$ in terms of $|A|$, $|B|$, $|C|$, $|A \cap B|$, $|A \cap C|$, $|B \cap C|$, $|A \cap B \cap C|$.

12. Express $|A \cap B \cap C|$ in terms of $|A|$, $|B|$, $|C|$, $|A \cup B|$, $|A \cup C|$, $|B \cup C|$, $|A \cup B \cup C|$.

13. Suppose we have sets $A_1, A_2, \ldots, A_n$. For each k, let

$$i_k := \text{sum of all sizes of } \underline{\text{intersections}} \text{ of } k \text{ of the sets.}$$

So i_1 gives just the sum of the sizes of the n sets, and i_2 gives the sum of the sizes of each intersection of $\underline{\text{two}}$ of the n sets, and so on.

For example, if $n=3$, our sets are A_1, A_2, A_3, and

$$\begin{aligned} i_1 &= |A_1|+|A_2|+|A_3|, \\ i_2 &= |A_1\cap A_2|+|A_1\cap A_3|+|A_2\cap A_3|, \\ i_3 &= |A_1\cap A_2\cap A_3|. \end{aligned}$$

Rewrite your expressions for $|A\cup B|$ (for which $n=2$) and $|A\cup B\cup C|$ (for which $n=3$) in terms of $i_1, i_2, \ldots$.

Then write out a general expression for $|A_1\cup A_2\cup A_3\cup\cdots\cup A_n|$. Pay careful attention to the sign of the coefficient of the last term in the sum.

For this exercise, you don't need to *prove* that your expression is correct. You will learn techniques for doing that later, in Chapter 3. But, if you do try to do a proof (even if only briefly), you will be even better prepared for that later material.

14. Now let

$$u_k := \text{sum of all sizes of unions of } k \text{ of the sets.}$$

So, for example, if $n=3$, with the same sets as before, we have

$$\begin{aligned} u_1 &= |A_1|+|A_2|+|A_3|, \\ u_2 &= |A_1\cup A_2|+|A_1\cup A_3|+|A_2\cup A_3|, \\ u_3 &= |A_1\cup A_2\cup A_3|. \end{aligned}$$

Rewrite your expressions for $|A\cap B|$ and $|A\cap B\cap C|$ in terms of $u_1, u_2, \ldots$.

Then write out a general expression for $|A_1\cap A_2\cap A_3\cap\cdots\cap A_n|$. Pay careful attention to the sign of the coefficient of the last term in the sum.

For this exercise, you don't need to prove that your expression is correct.

15. Express $|A\triangle B|$ in terms of the sizes of some other sets in three different ways.

16. Draw a Venn diagram for general sets A, B, C and shade the region(s) that form $A\triangle B\triangle C$.

17. Prove that, for any sets A and B,

$$(A\cup B)\triangle(A\cap B) = A\triangle B.$$

18. Suppose $A_1, A_2, \ldots, A_n$ are sets. How would you characterise $A_1\triangle A_2\triangle\cdots\triangle A_n$? More specifically, the members of $A_1\triangle A_2\triangle\cdots\triangle A_n$ are those that satisfy some specific condition on how many of the sets $A_1, A_2, \ldots, A_n$ they belong to. What is that condition?

For this exercise, you don't need to construct a formal proof that your condition is correct.

19. If A and B are sets, does $\overline{A} \times \overline{B}$ equal $\overline{A \times B}$? If so, prove it. If not: give an example when it does not hold; characterise those cases when it does hold; and determine what can be said about the relationship between them.

20. Write down all partitions of $\{1,2,3,4\}$ into three parts.

21. Suppose $A, B, C \subseteq U$ are nonempty sets such that all basic regions in their general Venn diagram are nonempty (see **??**). Then the three sets A, B, C define a partition of U into eight parts. What are those parts? Express each part in terms of one or more of A, B, C using set operations.

22. Does $\mathbb{R}$ have a partition in which all parts are

(a) open intervals?

(b) half-open half-closed intervals?

(c) closed intervals?

2

FUNCTIONS

Computation helps get things done. By "things", we mean tasks where we take some information and transform it somehow. To specify such a task, we must specify:

- what we start with;
- what we end up with;
- what needs to be done.

For example, suppose we want to sort a list of names into alphabetical order. We first specify exactly what kinds of lists of names we are dealing with (which alphabet? any requirements or restrictions on the names to be considered? how long can the names be? how many names are we allowed to have in the list? etc.). Then we specify that we'll end up with another list. Then we specify what we want to do to our first list, which is to sort it. So the list we end up with has all the same names as the list we started with, but now they are in alphabetical order.

Or suppose you want to determine the most goals kicked by any team in any season of your favourite football competition, from records of games. We first specify exactly what kinds of records we are working with: the information in them and how it is arranged. Then we specify that we'll end up with a number (to be precise, a non-negative integer). Then we specify that this number must be the maximum, over all seasons and all teams, of the number of goals kicked by that team in that season.

To model such tasks precisely, we use *functions*.

2.1$^{\alpha}$ DEFINITIONS

A **function** f consists of

- a set called the **domain**,
- a set called the **codomain**,
- a specification, for each element x of the domain, of a unique member $f(x)$ of the codomain. This member, $f(x)$, is the **value** of the function for the **argument** x.

Informally, the argument x "goes in" and the value $f(x)$ "comes out". These terms have some common synonyms.

- The *argument* x is also called the *parameter* or the *input*.
- The *value* $f(x)$ is also called the *output*.

But some care is needed with the use of "input" and "output" in this context, since in programming "input" is often used for extra information that a program reads from another source (such as a file), while "output" is often used for information a program writes to a file or screen, and these may be quite different to the argument and value.

Functions are ubiquitous in computer science, as well as in science, engineering, economics, finance, and any other field where symbolic objects of some type need to be transformed into something else in a precisely specified way.

We need to make some key points about the three parts of the definition of a function f.

2.1.1$^{\alpha}$ *Domain*

The **domain** of a function f can be any set, finite or infinite, provided that:

- Every member of the domain is considered to be a valid argument for the function. So, for every member x of the domain, its corresponding value $f(x)$ must be properly defined and "make sense".
- For everything that does *not* belong to the domain, the function is considered to be undefined and $f(x)$ has no meaning.

For our Sorting example, the domain is the set of all possible lists of names of the required type.

For the Maximum Goals example, it is the set of all possible records of all the games in a single season. Note that the domain is not just the set of all past seasons' records, since we want our function to be able to determine the required information for any possible future season as well.

Suppose now that we have a function called SumOfFourIntCubes which takes any four integers and finds the sum of their cubes, according to the rule

$$\mathsf{SumOfFourIntCubes}(w,x,y,z) = w^3 + x^3 + y^3 + z^3.$$

Then its domain is the set of all quadruples of integers, which we can write as $\mathbb{Z}\times\mathbb{Z}\times\mathbb{Z}\times\mathbb{Z}$. We could also envisage a function SumOfFourRealCubes that takes any four real numbers and finds the sum of their cubes. The rule looks the same:

$$\mathsf{SumOfFourRealCubes}(w,x,y,z) = w^3 + x^3 + y^3 + z^3.$$

But its domain is now $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$. Since this function has a different domain, it is considered to be a different function, even though the rule looks the same.

Why are functions with the same rule considered to be different if their domains are different? Why would we even need the function SumOfFourIntCubes when we can use the function SumOfFourRealCubes to do everything it does and more?

There are several reasons for this.

- Firstly, rules may *look* the same without actually *being* the same, because of details of how the operations in the rule depend on the types of objects being used. A multiplication symbol might be used to multiply numbers and also to multiply matrices, but they are very different operations.
- Secondly, the details of implementing functions in code will depend crucially on the specific types of objects they work with. In computing, real numbers are represented very differently to integers, even though we think of integers mathematically as a special case of real numbers. So some of the technical details of writing a program will be different for SumOfFourIntCubes and SumOfFourRealCubes. So, when we think of functions as specifications of tasks to be programmed, the specification of the domain is a crucial part of the definition.
- Thirdly, the purpose of the domain is not merely to help explain the rule; it is also a promise to the function's user that the function will work for every member of the domain. Sometimes, you may want a larger domain so that the function works for as many cases as possible. But bigger promises require more work to keep! Sometimes, it is better to use a more modest domain, provided it still captures everything that your function is supposed to work with; a more modest promise is easier to keep!

If f is a function, then we write $\mathrm{dom}(f)$ for the domain of f. So

$$\begin{aligned} \mathrm{dom}(\mathsf{SumOfFourIntCubes}) &= \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}, \\ \mathrm{dom}(\mathsf{SumOfFourRealCubes}) &= \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}. \end{aligned}$$

2.1.2$^{\alpha}$ *Codomain & co.*

The **codomain** of a function f must include every possible value of $f(x)$ for every member x of the domain. But the codomain is allowed to be "loose", in the sense that it is allowed to include other stuff too. We do not need to ensure that the *only* things in the codomain are things we can get by applying our function f to some member of its domain.

So, for the codomain, instead of specifying the possible function values *exactly*, we specify some *superset* of the set of possible function values.

The exact set of possible values of a function is called the **image** of the function. This is a subset of the codomain, but not necessarily a proper subset.

You may wonder at this point, why do we allow such "looseness" in the codomain when we were so insistent that the domain be the exact set of allowed arguments of the function? Why shouldn't we use the image, instead of the codomain, when defining a function?

The reason for this is practical. It is often harder to know the image than it is to specify a natural codomain. Sometimes it's impossible.

In our Maximum Goals example above, the codomain is the set $\mathbb{N} \cup \{0\}$. This does not mean that *every* nonnegative integer *must* arise as a possible maximum number of goals kicked by any team in a season. Indeed, some numbers (e.g., 10^{100}) could never arise in this way in practice. But it may be hard or impossible to know exactly which numbers are feasible values and which are not. So it is more practical to give a codomain, in the form of a simple, easily-described set which we know includes all possible function values (even if it has other things too). The set $\mathbb{N} \cup \{0\}$ works well.

In some cases, the difficulty may be computational. For example, consider the notorious **Pompous Python** function which takes any positive integer n and gives, as its value, the length (in characters) of the longest output that can be printed by *any* Python program which reads no input, eventually stops, and whose source code file has at most n characters.[1] On the face of it, this seems painful to compute, because there are so many programs that could be considered (once n is large enough to allow interesting programs of at most n characters to be written). In fact, it's worse than that; it can be shown that this function is *impossible* to compute perfectly. (This fact is not obvious, but is a consequence of some famous results on uncomputability from the 1930s. Uncomputability is covered in detail in the unit FIT2014 Theory of Computation.) So it is impossible, in a precise sense, to know exactly which numbers can be possible values of this function. Therefore, in specifying the function, we would prefer not to have to specify, in advance, which numbers to allow for its possible values! So, instead, we specify a suitable codomain, and in this case $\mathbb{N} \cup \{0\}$ will do fine, even though only very few numbers can actually be values of the Pompous Python function.

In other cases, the difficulty may be the limits of our current knowledge. Consider again SumOfFourIntCubes. It is not yet known whether or not every integer can be written as the sum of four cubes, so we do not know if the values taken by this function are all integers or only some subset of them. But we can specify the codomain to be $\mathbb{Z}$ and know that this is a superset (but not necessarily a proper superset, as far as we currently know) of the actual set of possible values.

The codomain, then, represents a *promise* to users of the function that all the values they get from it will be in that set, but it gives no guarantee that *every* one of its members is an actual function value.

---

1 Here we envisage an ideal computing environment where arbitrarily long programs can be run, arbitrarily long outputs can be printed, and programs can take an arbitrarily long time to run before stopping. If a program crashes or prints no output, then we define the length of its output to be 0. If a program runs forever, in an infinite loop, then we exclude it from consideration, regardless of how much output it produces.

It is good to know the image too, when that's possible. But we do not want our definition of the function concept to be hampered by the difficulties that are often associated with specifying the image. So we do not include a specification of the image in our specification of the function. Rather, we use the codomain as the "next best thing".

Of course, if we do know the image, there is nothing to stop us from stating it as our codomain. But, even then, it is often neater to specify a simple codomain than it is to specify the image. For example, a function with domain $\mathbb{N}$ whose value is the n-th Fibonacci number has, as its image, the set of all Fibonacci numbers, and it's easy enough to write that down.[2] But it's even easier to specify $\mathbb{N}$ as a codomain.

A function in which the codomain is the same as the image of the function is said to map its domain **onto** its codomain, and may be said to be **onto**. Such a function is also said to be **surjective** and is called a **surjection**.

Finally, a word of warning! We have studiously avoided using the word "range". This is because the term is, confusingly, used in two different ways: sometimes, it means the image, while at other times, it means the codomain. We will dodge this issue by not using "range" at all.

2.1.3$^{\alpha}$ *Rule*

The rule of a function must specify the relationship between each member of its domain and the corresponding value of the function. But it does not, in general, give an *algorithm* for computing the function.

In other words, the rule specifies *what* must be done, but it does not need to specify *how* it is to be done.

In our Sorting example, the rule is that the function's value is the sorted list of names. This rule does not specify *how* the sorting is to be done. As computer science students, you will meet many different sorting algorithms, including Bucket Sort, Merge Sort, Insertion Sort, and Quick Sort. They all have their strengths and all could be used to compute our sorting function. But the *function itself* does not include a choice of which algorithm we will use to compute it; that choice is a separate issue to the specification of the function.

Sometimes, a function's rule does give some information on how to compute it. For example, consider a function that squares integers. It has domain $\mathbb{Z}$, and we'll use the (loose) codomain $\mathbb{Z}$ too. Its rule is just that any integer argument is squared. This may be thought of as a small algorithm: it tells you what the value is in such a way that you also know how to work it out. Or do you? There is more than one algorithm for squaring an integer! To specify the rule, we don't need to specify which algorithm is to

2 The **Fibonacci numbers** are the numbers you get by starting with two consecutive 1s and then repeatedly adding the two most recent numbers together to get the next number. So the **Fibonacci sequence** is: 1,1,2,3,5,8,13,21,34,55,89,144,233,....

be used; we only need to give enough information so that the reader can know, for each argument, what its corresponding value is.

A function's rule associates, to each argument in its domain, a unique value in its codomain. One way to specify this information is to give the set of all possible ordered pairs $(x, f(x))$. For example, consider a function Employer, defined as follows. Its domain is the set

{Annie Jump Cannon, Henrietta Swan Leavitt, Muriel Heagney, Winsome Bellamy},

which consists of four human computers who worked at various astronomical observatories. For its codomain, we use the set of *all* astronomical observatories over the last two centuries. The rule gives values to arguments as follows.

$$\begin{aligned} \text{Employer(Annie Jump Cannon)} &= \text{Harvard College Observatory,} \\ \text{Employer(Henrietta Swan Leavitt)} &= \text{Harvard College Observatory,} \\ \text{Employer(Muriel Heagney)} &= \text{Melbourne Observatory,} \\ \text{Employer(Winsome Bellamy)} &= \text{Sydney Observatory.} \end{aligned}$$

We can specify this rule by giving the following set of ordered pairs ($computer$, Employer($computer$)).

{ (Annie Jump Cannon, Harvard College Observatory),
(Henrietta Swan Leavitt, Harvard College Observatory),
(Muriel Heagney, Melbourne Observatory),
(Winsome Bellamy, Sydney Observatory) }

Note that this function is not a surjection, since its image is only

{ Harvard College Observatory, Melbourne Observatory, Sydney Observatory }

which is a (very) proper subset of the codomain.

To do the same with our squaring function would require an infinite set of ordered pairs,

$$\{(0,0),(1,1),(-1,1),(2,4),(-2,4),\ldots\},$$

which we can write succinctly, using our standard conventions for writing sets, as

$$\{(x, x^2) : x \in \mathbb{Z}\}.$$

The set of all ordered pairs $(x, f(x))$ of a function f is called the **graph** of the function. This term reminds us that we often illustrate a function by drawing a plot of all points $(x, f(x))$, with the horizontal axis containing the domain and the vertical axis containing the codomain. We are used to referring to such a plot as a "graph" of the function, but the term *graph* as just defined is more abstract: it just refers to the set of pairs, without regard to how they might be displayed to a reader.

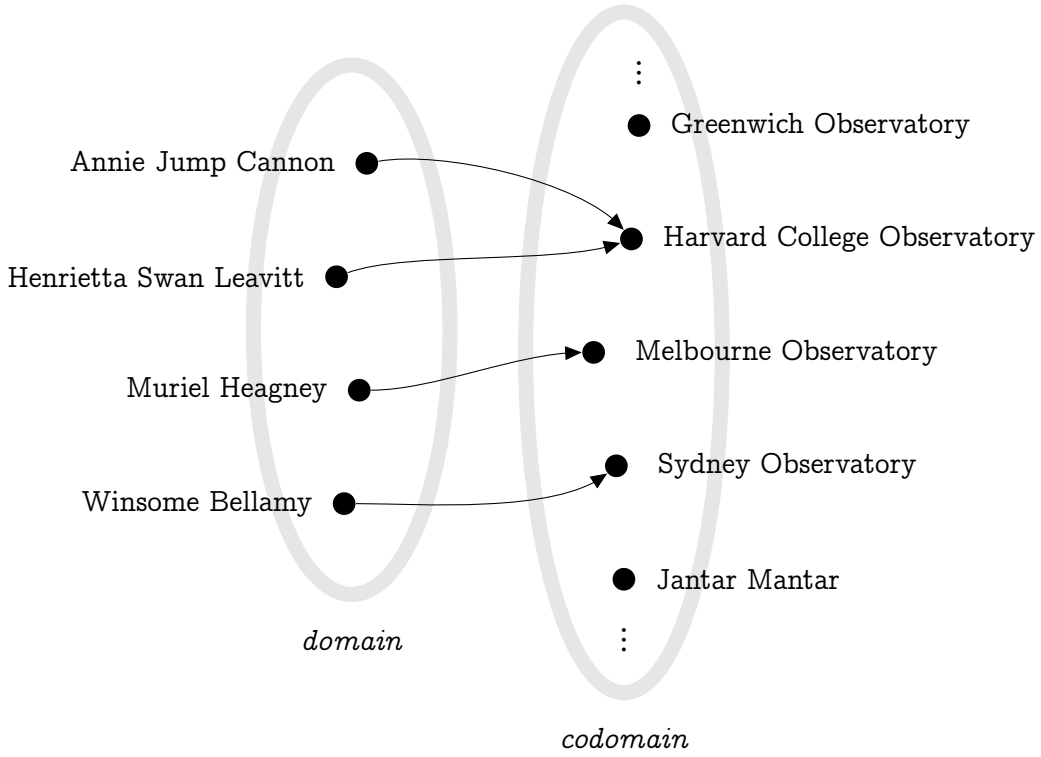


Figure 2.1: The Employer function.

Plots of functions, using horizontal and vertical axes, are convenient visual ways to display information about the function, but they also have their limitations. Many domains and codomains do not have an inherently one-dimensional structure, and increasing the number of dimensions — say, by using a 3D plot — does not always help. Some domains and codomains are not geometric in character at all. For example, the domain of Employer is a set of four people, and the domain of our Sorting function is the set of all possible lists of names, neither of which are defined in numerical or geometric terms.

There are other ways to depict functions. For example, we could start with a Venn diagram of the domain and codomain, draw points within the domain representing its members, and then, for every pair $(x, f(x))$, draw an arrow from x to $f(x)$ to indicate that the function sends x to $f(x)$. Because f is a function, every point in the domain has exactly one arrow going out of it. Our Employer function is depicted in this way in Figure 2.1.

2.2$^{\alpha}$ FUNCTIONS IN COMPUTER SCIENCE

2.2.1$^{\alpha}$ *Functions from analysis*

In software development, the first task is to work out what must be done. This process is traditionally called **analysis** and usually involves extensive communication with the owner of a problem (e.g., a client) in order to come up with a precise description of the task at hand. One possible outcome of this analysis process is a function.

At this stage, we have not yet worked out how to solve the problem at hand. But at least we have a precise statement of the task to be done (the "what"). With this, we can then try to **design** a method for doing this task (the "how"). If our task is specified by a function, then we will design an algorithm for the function.

Once we have an algorithm, we proceed to **implementation**, or *programming* the algorithm using a programming language such as Python.

Of course, this is a very simplified and incomplete view of software development. The process is seldom purely linear; each step often involves going back and re-doing parts of a previous stage. The design process might highlight problems, or gaps, in the specification, which may require further communication with the problem owners to sort out, leading to changes to the specification. Or the clients may simply change their minds about some aspect of the task! The implementation process may bring to light some problems with the design which must then be fixed. There are later stages we haven't mentioned, notably **maintenance**. And not all problem analysis leads to a function specification. For example, some may lead to a specification of how the various components of some system must interact; some may lead to a specification of a database. But functions remain a very important product of analysis, partly because more complicated systems often contain functions as components.

2.2.2$^{\alpha}$ *Functions in mathematics and programming*

Our view of functions, as specifications of what rather than how, originated in mathematics although is now widespread in computer science and other disciplines. But, as you study programming, you will find that the term is used in another way too.

In many programming languages, and even in many pseudocode conventions for writing algorithms, a definition of a "function" has — in addition to the parts discussed here — some code in the programming language, or an algorithm, that specifies how the function is to be computed. In fact, the very word "function" is a reserved word in some programming languages and has this meaning, possibly with additional technical details.

By default, we use the term "function" in the mathematical sense, where there is no code or algorithm given. Occasionally we may use the term "mathematical function" to emphasise this, but even without that adjective, the term "function" will be used in this way.

If we wish to use the programming sense of the term, we will specify that explicitly, as in "Python function" or "programming function" or "algorithmic function".

In *functional programming* languages, algorithmic functions are the most fundamental objects used, and all computation is done by manipulating them. They can be represented by variables and treated as both arguments and values of other functions. We do not consider the functional programming paradigm in this unit or in FIT1045. You can learn more about functional programming in FIT2102 Programming Paradigms.

2.3$^{\alpha}$ NOTATION

A function definition has the form

$$\begin{aligned} \textit{name} : \textit{domain} &\longrightarrow \textit{codomain}, \\ \textit{name}(x) &= \textit{precise description of the function value, in terms of } x. \end{aligned}$$

For example, a function f that gives squares of real numbers can be defined by

$$\begin{aligned} f : \mathbb{R} &\longrightarrow \mathbb{R}, \\ f(x) &= x^2. \end{aligned}$$

Here, the domain is $\mathbb{R}$, and the codomain is $\mathbb{R}$ too. Since we stated that our function would give the squares of *real numbers*, we really have no alternative but to specify $\mathbb{R}$ as the domain. But we have some more flexibility with the codomain. We could have used $\mathbb{R}_0^+$, since that is the image of this function: the squares of real numbers are precisely the nonnegative real numbers. We could, instead, have used $\mathbb{R} \cup \{0, -\sqrt{2}, -42\}$ as the codomain, which is perfectly valid mathematically, although this codomain has some extra detail that is irrelevant, useless, distracting, and shows poor user interface design!

The first line of these function definitions, such as $f : \mathbb{R} \longrightarrow \mathbb{R}$, is like a declaration in a program. It announces the name of the function and specifies the types of objects that it can take as its arguments and give as its values. The second line, such as $f(x) = x^2$, completes the definition by specifying the rule. A common form of wording is to say something like, "The function $f : \mathbb{R} \to \mathbb{R}$ is defined by $f(x) = x^2$."

There is another common convention for specifying the rule of a function, where we just write

$$x \;\mapsto\; \textit{precise description of the function value, in terms of } x.$$

Note how the "mapping arrow" $\mapsto$ in the rule differs from the ordinary arrow $\to$ going from domain to codomain. It is important not to mix the two arrow types up.

If we use this second convention, then our squaring function would be defined by

$$\begin{aligned} f : \mathbb{R} &\longrightarrow \mathbb{R}, \\ x &\mapsto x^2. \end{aligned}$$

2.4$^{\alpha}$ SOME SPECIAL FUNCTIONS

The most trivial, vacuous, degenerate function of all is the **empty function**, denoted $\emptyset$. Its domain and codomain are each empty, and it has no rule because there is nothing in the domain for any rule to apply to. It can be defined simply as $\emptyset : \emptyset \to \emptyset$. It's pretty useless; we may hope never to see it again! Let's move on.

For any set A, the **identity function** i_A on A is defined by

$$\begin{aligned} i_A : A &\longrightarrow A, \\ i_A(x) &= x. \end{aligned}$$

This function just maps each member of A to itself.[3]

For any domain D, we can define for every subset $A \subseteq D$ the **indicator function** χ_A, which uses 1 and 0 to indicate, for each member of the domain, whether or not it is in A:

$$\begin{aligned} \chi_A : D &\longrightarrow \{0,1\}, \\ \chi_A(x) &= \begin{cases} 1, & \text{if } x \in A; \\ 0, & \text{if } x \notin A; \end{cases} \end{aligned}$$

Although indicator function notation χ_A only mentions A, it must be kept in mind that a function's definition always includes a specification of its domain, which in this case is D. Different domains give rise to different indicator functions.

We can express the indicator functions of sets obtained from set operations on A and B using the indicator functions of A and B. For example, for all x we have

$$\chi_{A \cup B}(x) = \max\{\chi_A(x), \chi_B(x)\}.$$

See Exercise 2.

For any domain D and any object a, the **constant function** c_a just maps everything to a:

$$\begin{aligned} c_a : D &\longrightarrow \{a\}, \\ c_a(x) &= a. \end{aligned}$$

2.5$^{\alpha}$ FUNCTIONS WITH MULTIPLE ARGUMENTS AND VALUES

Many functions you will meet have multiple arguments. If f is a function of two arguments x and y, then we write its value as $f(x,y)$. Suppose $x \in X$ and $y \in Y$, and that the value of the function belongs to a codomain C. Then the function definition would start by stating $f : X \times Y \to C$.

3 So, in a sense, it does nothing. But at least it does *nothing* to *something*, whereas the empty function does nothing to nothing!

It may seem that we are extending our definition of functions here, since we seem to have two domains, X and Y, for the first and second argument respectively. And no real harm can come from this view. But functions of two arguments may also be viewed functions of a *single* argument where that one argument happens to be a pair (x, y) and its domain is the Cartesian product $X \times Y$. So, when we start a function definition with $f : X \times Y \to C$, we are still just using our usual way of defining functions. When we write $f(x, y)$, indicating two arguments, we are using a shorthand for $f((x, y))$, where the argument that we put inside $f(\cdots)$ is the ordered pair (x, y). In accordance with usual practice, we will drop the second pair of parentheses from $f((x, y))$, writing $f(x, y)$ instead, and we will happily speak of its first argument x, its second argument y, and so on. But keep in the back of your mind that we can also view this as a function of just one argument whose sole argument happens to be the ordered pair (x, y).

When we write $f(x, y)$ for applying function f to arguments x and y, we are using **prefix** notation, because we put the name of the function *before* the arguments. This is common practice and is the one we use when defining new functions. But there are also many well-known functions that use **infix** notation, where the function name is put *between* its arguments. Familiar examples include ordinary arithmetic functions $+$, $-$, $\times$, $/$ and many built-in operations in many programming languages. Far less common is **postfix** notation, where the name is placed *after* the arguments.

All these remarks extend readily to functions of three or more arguments. For example, a function of three arguments can also be regarded as a function of a single argument which happens to be a triple.

We also sometimes want the value of a function to be a tuple. For example, suppose the value of a function is to be a pair (y, z) where $y \in Y$ and $z \in Z$. If the domain of the function is X, then we may write $f : X \to Y \times Z$. This function may be regarded as giving two values, which we find it convenient to put in an ordered pair. We also view it as giving a single value which happens to be the ordered pair $(y, z) \in Y \times Z$.

Again, these comments extend readily to functions that return tuples of three or more values.

2.6 RESTRICTIONS

Sometimes we want to restrict a function to some subset of its domain, and to treat this restricted version of the function as a new function in its own right. For example, consider a function that assigns ID numbers to Monash students. Its domain is the set of all Monash students. If we want to use a function which only considers FIT1058 students, and assigns ID numbers to them, then this function is a restriction of the previous function just to the set of all FIT1058 students.

There are several reasons why we might want to focus just on the restriction of some function.

- If the domain is significantly smaller than the original function, then storing the restricted function as a list of ordered pairs takes up less space.

- As we discussed on page 39 in § 2.1.1$^{\alpha}$, the domain of a function serves as a promise to users of the function that anything in the domain is valid, for that function, and is assigned a value by the function. As we noted, a bigger domain amounts to a bigger promise which may require more work to keep. If a function is given an algorithm and implemented as a program, then testing is required in order to be sure that it works for all members of its domain. The bigger the domain is, the more testing is required. So you may prefer to restrict the function to a subset of its domain, focusing on the arguments you *really* care about and reducing the work of testing and the size of the promise implied by the domain.

- Sometimes a function is simpler to describe, analyse or compute on some particular subset of its domain that we most care about.

- Sometimes a restricted version of a function may have stronger properties that enable it to be used in situations where the original function cannot be used.

Suppose we have a function $f : A \to B$ and that $X \subseteq A$. The **restriction** of f to X, denoted by $f|_X$ or $f \restriction_X$, is the function with domain X, codomain B, and which agrees with f on X:

$$\begin{aligned} f|_X : X &\to B, \\ f|_X(x) &= f(x) \quad \text{for all } x \in X. \end{aligned}$$

For example, suppose f is our squaring function from § 2.3$^{\alpha}$:

$$\begin{aligned} f : \mathbb{R} &\longrightarrow \mathbb{R}, \\ f(x) &= x^2. \end{aligned}$$

Then its restriction $f|_{\mathbb{R}_0^+}$ to the nonnegative real numbers is given by

$$\begin{aligned} f|_{\mathbb{R}_0^+} : \mathbb{R}_0^+ &\longrightarrow \mathbb{R}, \\ f(x) &= x^2. \end{aligned}$$

This has some properties that the original function f does not have. For example, $f|_{\mathbb{R}_0^+}$ is continually increasing as x increases across its domain, whereas $f(x)$ is decreasing along some of its domain (specifically, along $\mathbb{R}_0^-$) and increases elsewhere (along $\mathbb{R}_0^+$), so its behaviour is a bit more complicated. Each member y in the image of $f|_{\mathbb{R}_0^+}$ comes from a *unique* x in the domain $\mathbb{R}_0^+$, namely the positive square root $\sqrt{y}$ of y (or 0, in the case $y = 0$). By contrast, each nonzero member y of the image of f comes from *two different* values of x, namely the two square roots $\pm\sqrt{y}$ of y. This illustrates the point mentioned above that a restriction can be simpler and have stronger properties, which can make it more useful in some situations. (Later, in § 2.8, we discuss inverse functions. Then, we can say that $f|_{\mathbb{R}_0^+}$ is invertible but f is not.)

2.7 INJECTIONS, SURJECTIONS, BIJECTIONS

As we have seen from some of our examples, it is perfectly ok for different function arguments to give the same value. So it is ok for both Annie Jump Cannon and Henrietta Swan Leavitt to have the value Harvard College Observatory under the Employer function (p. 42 in § 2.1.3$^{\alpha}$). There is no requirement for there to be a unique argument for each value. This contrasts with the requirement that there be a unique value for each argument, which is an essential property of *any* function. This specific Employer function only assigns one employer to each human computer.

Although it's ok in general for different arguments to be mapped to the same value, there are situations where we do not want that to happen. For example, a function that assigns an ID number to each student must ensure that different students get different ID numbers. A function that encrypts files must ensure that different files are encrypted differently, else the contents of a file cannot be recovered from its encrypted form.

A function with this property, that different arguments are always mapped to different values, is said to be **injective** and is called an **injection**. Mathematically, this property of a function $f : A \longrightarrow B$ can be expressed as follows: for any two distinct $x_1, x_2 \in A$, we have $f(x_1) \neq f(x_2)$. Such a function gives a **one-to-one correspondence** between the domain and the image, but *not* between the domain and the codomain in general.

Injections have the virtue of *preserving information*: for every member y in the *image* of an injection f, there is a unique x in its domain such that $f(x) = y$. In every case, knowing y is logically sufficient for determining x (although we are not saying anything here about how much *work* it might be to recover x; that depends on the details of the function). If a function is not an injection, then there must be at least one member y of its image such that there are two or more members x_1, x_2 of its domain which map to that image: $f(x_1) = f(x_2) = y$. So, in that case, knowing y still leaves you in doubt as to how it could have been produced by f.

Functions that aren't injections *lose information*. We might call them **lossy**. In fact, this term is used for data compression functions that are not injections. By contrast, an injective data compression function is called **lossless**.

Similarly, we can express mathematically the **onto** property of a function, which we defined in § 2.1.2$^{\alpha}$. A function $f : A \longrightarrow B$ is **surjective**, and is said to be a **surjection**, if for any value $y \in B$ there must be a value of the function's argument $x \in A$ such that $y = f(x)$.

A function that is both an injection and a surjection is said to be **bijective** and is called a **bijection**. Such a function is a **one-to-one correspondence** between the domain and the codomain (which is also the image in this case).

A bijection whose domain and codomain are the same set is also called a **permutation**, though the latter term is usually used only for bijections on finite sets.

Bijections preserve information, since they are injections. Furthermore, since they are also surjections, each member of the codomain may be thought of as encoding a

unique member of the domain. So a bijection establishes that the domain and codomain contain the same information, although it may be represented in different ways.

If a function has *finite* domain and codomain of the *same size*, and is an injection, then it must also be a surjection, and hence also a bijection. This is because there is no room in the codomain for the injection to avoid mapping to all its members. Similarly, a surjection with *finite* domain and codomain of the *same size* must also be an injection, and hence a bijection, since the need to map to all members of the codomain prevents any repetition of codomain elements. Both these assertions fail if the domain and codomain are infinite. Can you find examples to illustrate the failure in each case?

2.8 INVERSE FUNCTIONS

We think of a function as going *from* any member of its domain *to* its corresponding value in the codomain. But there are times when we may want to go backwards: given a value in the codomain, what argument in the domain does it come from? For example, which computer worked at Harvard College Observatory? Which number, when squared, gives 4? Which file corresponds to a particular encrypted file?

For functions in general, the answer may not be unique. We have just mentioned some cases of this: our Employer function mapped two different computers to Harvard College Observatory, and $2^2 = (-2)^2 = 4$.

This failure of uniqueness can happen in either of two different ways. Let $f : A \to B$ be a function. For a given value y in the codomain:

- There may be more than one argument that gives the value y under the function. So we may have $x_1, x_2 \in A$ such that $x_1 \neq x_2$ but $f(x_1) = f(x_2) = y$.

- There may be no argument that gives the value y. This happens when the image is a proper subset of the codomain and y lies in the codomain but not in the image.

But if neither of these occurs, then every $y \in B$ has a unique $x \in A$ such that $f(x) = y$. This means that, in giving each $y \in B$ a corresponding $x \in A$, we are actually defining a function from B to A, with domain B and codomain A. So the roles played by A and B are reversed, in keeping with the reversed "direction" of this new function. We call this new function the **inverse function** of f and denote it by f^{-1}. We can write its definition as follows.

$$\begin{aligned} f^{-1} : B &\longrightarrow A, \\ f^{-1}(y) &= \text{the unique } x \text{ such that } f(x) = y. \end{aligned}$$

If we want to write the rule of f^{-1} as a set of ordered pairs, then we just take all the ordered pairs in f and reverse them:

$$\{\ (y, x) : (x, y) \text{ belongs to } f\ \}$$

or, to put it slightly differently,

$$\{\ (f(x), x) : x \in A\ \}.$$

For a function to have an inverse function, it must be an injection (so that no value has two corresponding arguments) and it must also be a surjection (so that every value in the codomain is also in the image, i.e., has a corresponding argument). So, in fact, a function has an inverse function if and only if it is a bijection.

Our Employer function is not a bijection and therefore does not have an inverse function. The squaring function also does not have an inverse function, for the same reason. But we would want an encryption function to have an inverse function, so that an intended user of an encrypted file has no doubt about its contents. (In that context, there is the separate issue of how easy or hard it should be to actually compute the inverse. We would like that to be easy for intended users and hard for others. Achieving these competing aims is the fundamental challenge of cryptography.)

The role of the inverse function is to "undo" the function and get back what you started with. So, if you have $x \in A$ and apply f to get $y = f(x)$, then it does not matter much if you "lose" x, because you can recover it from y:

$$x = f^{-1}(y) = f^{-1}(f(x)).$$

Since the original function f is a bijection, its inverse f^{-1} is also a bijection. It follows that f^{-1} also has an inverse, which we could write as $(f^{-1})^{-1}$. But the inverse of the inverse is just the original function; if f^{-1} "undoes" f, then f also "undoes" f^{-1}. We have

$$y = f(x) = f(f^{-1}(y)).$$

2.9 COMPOSITION

It is common to use values obtained from one function as arguments to another function. For example, consider the functions Father and Mother which each have, as their domain and codomain, the set $\mathbb{P}$ of all people who have ever lived.

$$\begin{aligned} \mathsf{Father} : \mathbb{P} &\longrightarrow \mathbb{P}, \\ \mathsf{Father}(p) &= \text{the father of person } p. \\ \mathsf{Mother} : \mathbb{P} &\longrightarrow \mathbb{P}, \\ \mathsf{Mother}(p) &= \text{the mother of person } p. \end{aligned}$$

Starting with Alan Turing, the function Mother gives Alan's mother, Sara Turing. Applying the function Father to her gives Sara's father — Alan's maternal grandfather — Edward Stoney.

$$\begin{aligned} \mathsf{Mother}(\text{Alan Turing}) &= \text{Sara Turing}. \\ \mathsf{Father}(\text{Sara Turing}) &= \text{Edward Stoney}. \end{aligned}$$

This "chaining" of the two functions is called *composition*, and is denoted by stating the second function, followed by the composition symbol $\circ$, followed by the first function. (Note the order there.) In this case, the function is $\mathsf{Father} \circ \mathsf{Mother}$ and it is defined as follows.

$$\begin{aligned} \mathsf{Father} \circ \mathsf{Mother} : \mathbb{P} &\longrightarrow \mathbb{P}, \\ \mathsf{Father} \circ \mathsf{Mother}(p) &= \mathsf{Father}(\mathsf{Mother}(p)). \end{aligned}$$

Note now the order in which the functions are written in $\mathsf{Father} \circ \mathsf{Mother}$ is the same as the order in which they are written when we write one function as an argument of the other, i.e., in $\mathsf{Father}(\mathsf{Mother}(p))$. But our usual *order of reading and writing* (left to right) is the reverse of the *order of application* (right to left): we *apply* the function Mother *first*, and *then* we apply the function Father.

In this example, the codomain of the first function applied (Mother) equals the domain of the second function applied (Father); both are $\mathbb{P}$. This ensures that our first function application, using Mother, always produces something (or someone) that our second function, Father, can deal with. This is a general requirement for function composition.

In general, the **composition** $g \circ f$ of two functions $f : A \to B$ and $g : B \to C$ is defined by

$$\begin{aligned} g \circ f : A &\longrightarrow C, \\ g \circ f(x) &= g(f(x)). \end{aligned}$$

Note that, in this composition, f is applied *first*, and then the result is given to g. The order of doing things does matter; in general, function composition is not commutative, meaning that $g \circ f$ and $f \circ g$ are not the same. It is, however, associative: if $f : A \to B$, $g : B \to C$ and $h : C \to D$, then

$$h \circ (g \circ f) = (h \circ g) \circ f,$$

so we can write $h \circ g \circ f$ unambiguously.

Function composition is undefined if the codomain of the first function applied does not equal the domain of the second function appied. For example, suppose $\mathbb{S}$ is the set

of all students, $\mathbb{U}$ is the set of all Monash units, and the function FavouriteUnit is defined as follows.

$$\begin{array}{rcl} \mathsf{FavouriteUnit} : \mathbb{S} & \longrightarrow & \mathbb{U}, \\ \mathsf{FavouriteUnit}(p) & = & \text{the favourite Monash unit of person } p. \end{array}$$

If you want to know your mother's favourite unit at Monash, you might be tempted to use the function FavouriteUnit $\circ$ Mother. But not all mothers are students, and not all students are mothers. Formally, the codomain of Mother is $\mathbb{P}$, which does not equal the domain of FavouriteUnit, namely $\mathbb{S}$. So this function composition is undefined.

The requirement that the codomain of the first function f equals the domain of the second function g amounts to insisting that our guarantee about what f can produce (expressed in the form of its codomain) is the same as our guarantee about what g can handle (expressed in the form of its domain). So the two functions are compatible, in a precise sense.

You have seen composition before for mathematical functions. For example, the expression $(x-1)^2$ may be regarded as the rule for the composition $g \circ f$ of the two functions

$$\begin{array}{rcl} f : \mathbb{R} & \longrightarrow & \mathbb{R}, \\ f(x) & = & x-1. \\ g : \mathbb{R} & \longrightarrow & \mathbb{R}_0^+, \\ g(x) & = & x^2. \end{array}$$

We saw a very special case of composition in § 2.8. If $f : A \to A$ is a bijection, then

$$\begin{array}{rcl} f \circ f^{-1} & = & i_A, \\ f^{-1} \circ f & = & i_A. \end{array}$$

If $g : C \to D$ is any function, then composing it with the identity function makes no difference, provided we pick the correct identity:

$$\begin{array}{rcl} g \circ i_C & = & g, \\ i_D \circ g & = & g. \end{array}$$

We can compose a function *with itself* provided its domain and codomain are the same. If $f : A \to A$ then the definition of composition tells us that $f \circ f : A \to A$ is defined for all $x \in A$ by $f \circ f(x) = f(f(x))$. We can then do iterated composition of f with itself, if we wish. We write $f^{(n)}$ for the composition of n copies of f:

$$f^{(n)} = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ copies of } f}$$

We introduce three iterated compositions: one from a practical application, one party trick, and one profound unsolved theoretical question.

- Consider the function $\mathsf{LCG} : [0, 2^{31}-1]_{\mathbb{Z}} \to [0, 2^{31}-1]_{\mathbb{Z}}$ defined for all 31-bit nonnegative binary integers x by

$$\mathsf{LCG}(x) \;=\; \text{the last 31 bits of } \;\; 1103515245x + 12345.$$

  We always keep only the last 31 bits, to ensure that the numbers generated stay within our fixed interval. This function has been used to generate sequences of numbers that are *pseudorandom* in the sense that, superficially, they look random if you don't look too closely. Starting with some initial "seed" number s, the function LCG is applied repeatedly, and the successive numbers $\mathsf{LCG}^{(n)}(s)$ should behave in a way that looks statistically random in some sense. (We have used this example as it is one of the simpler pseudorandom number generators that have been used in practice, but its randomness properties are imperfect and it should not be used by itself in this naive way. It is usually used in conjunction with other methods in order to increase the randomness.) The name LCG comes from the term *Linear Congruential Generator*, which is a type of pseudorandom number generator of which this is one example.

- Consider the function $K : [0, 9999]_{\mathbb{Z}} \to [0, 9999]_{\mathbb{Z}}$ defined for any nonnegative integer x with at most four (decimal) digits as follows. First, form a four-digit number by writing the four digits of x (using leading zeros if necessary) from smallest to largest. Then reverse that number, so that the digits now go from largest to smallest. Then $K(x)$ is defined to be the difference between these two numbers. For example,

$$K(1729) = 9721 - 1279 = 8442.$$

  This function is not an injection (why?). It is clear that, if all the digits in x are the same, then $K(x) = 0$, and therefore $K^{(n)}(x) = 0$ for all $n \geq 1$. More surprisingly, in *all other cases* (i.e., when x has at least two different digits), iterated composition of this function with itself eventually reaches 6174, and does so after at most seven iterations. For example, starting with 1729 as above, we have

$$1729 \overset{K}{\longmapsto} 8442 \overset{K}{\longmapsto} 5994 \overset{K}{\longmapsto} 5355 \overset{K}{\longmapsto} 1998 \overset{K}{\longmapsto} 8082 \overset{K}{\longmapsto} 8532 \overset{K}{\longmapsto} 6174 \overset{K}{\longmapsto} 6174.$$

  So $K^{(7)}(x)$ is 0 if all digits are the same and 6174 otherwise. This was discovered by the Indian mathematician D. R. Kaprekar in 1946 and published in 1955.[4]

4 D. R. Kaprekar, An interesting property of the number 6174, *Scripta Mathematica* **21** (1955) 304. Martin Gardner, Mathematical Games, *Scientific American* (March 1975).

- Consider the function $\mathsf{Collatz} : \mathbb{N} \to \mathbb{N}$ defined for all $x \in \mathbb{N}$ by

$$\mathsf{Collatz}(x) = \begin{cases} 3x+1, & \text{if } x \text{ is odd;} \\ x/2, & \text{if } x \text{ is even.} \end{cases}$$

  For example, if we start with 7 and keep iterating, we obtain

$$7 \mapsto 22 \mapsto 11 \mapsto 34 \mapsto 17 \mapsto 52 \mapsto 26 \mapsto 13 \mapsto 40 \mapsto 20 \mapsto 10 \mapsto 5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1 \mapsto 4 \mapsto 2 \mapsto 1 \mapsto \cdots$$

  Note how it eventually gets stuck in a loop, going from 4 to 2 to 1, then back to 4, and so on.

  Iterated composition of this function is mysterious. It is conjectured that, for every x, there exists n such that $\mathsf{Collatz}^{(n)}(x) = 1$. This has become known as **Collatz's Conjecture** or the $3x+1$ **problem**. Currently it is unsolved. It is a remarkable illustration that even simple questions about very simple algorithms can be very deep and hard to answer.

We now consider how the injective, surjective and bijective properties are affected by composition.

We start with injection.

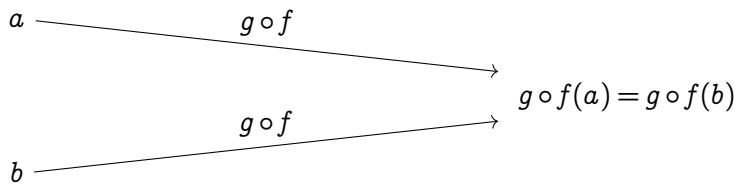
Theorem 6. If $f : A \to B$ and $g : B \to C$ are injections then $g \circ f$ is also an injection.

Proof. We prove this by proving the equivalent statement that

> if $g \circ f$ is *not* an injection, then f and g are *not* both injections.

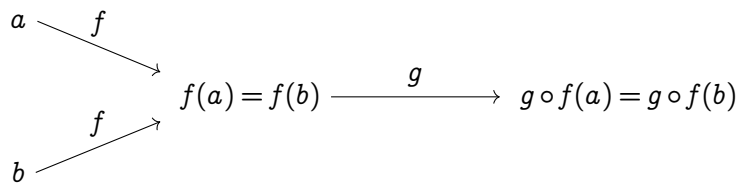
This is an example of the general logical principle that "A implies B" is equivalent to "not-B" implies not-A". For example, the statement that "every rectangle has four sides" is equivalent to the statement that "everything that does not have four sides is not a rectangle". See (1.10).

Suppose that $g \circ f$ is *not* an injection. Then there must exist $a, b \in A$ such that $a \neq b$ and $g \circ f(a) = g \circ f(b)$.

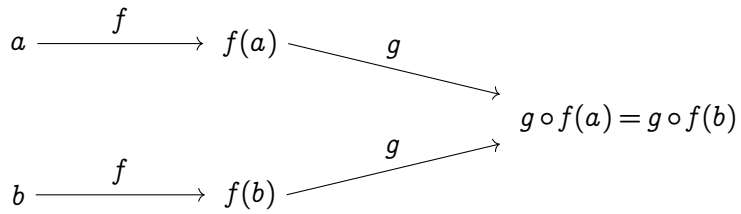


Consider $f(a)$ and $f(b)$. Either they are equal or unequal.

- If $f(a) = f(b)$, then we have $a \neq b$ and $f(a) = f(b)$, so f is not an injection.



- If $f(a) \neq f(b)$, then we have $f(a) \neq f(b)$ and $g(f(a)) = g(f(b))$, so g is not an injection (since we have two *distinct* members of its domain, namely $f(a)$ and $f(b)$, that are mapped by g to the same value).



So we see that, whatever happens with $f(a)$ and $f(b)$, at least one of f and g is not an injection. □

The converse of this theorem does not hold: $g \circ f$ being an injection does *not* imply that *both* f and g are injections. For example, define $f : \{1,2\} \to \{1,2,3\}$ by

$$\begin{aligned} f(1) &= 1, \\ f(2) &= 2, \end{aligned}$$

and define $g : \{1,2,3\} \to \{1,2\}$ by

$$\begin{aligned} g(1) &= 1, \\ g(2) &= 2, \\ g(3) &= 2. \end{aligned}$$

Then their composition $g \circ f : \{1,2\} \to \{1,2\}$ is defined by

$$\begin{aligned} g \circ f(1) &= g(f(1)) = g(1) = 1, \\ g \circ f(2) &= g(f(2)) = g(2) = 2. \end{aligned}$$

So $g \circ f$ is an injection, but it is not the case that both f and g are injections. In fact, f is an injection, but g is not.

Now let's consider surjections.

Theorem 7. If $f : A \to B$ and $g : B \to C$ are surjections then $g \circ f$ is also a surjection.

Proof. Suppose f and g are surjections. Consider $g \circ f : A \to C$.

Let $c \in C$ be any member of its codomain C.

Since g is a surjection, there must exist $b \in B$ such that $g(b) = c$.

Since f is a surjection, there must exist $a \in A$ such that $f(a) = b$.

But then we have

$$g \circ f(a) \;=\; g(f(a)) \;=\; g(b) \;=\; c.$$

So, every member of the codomain of $g \circ f$ also belongs to its image. Therefore $g \circ f$ is a surjection. □

For Theorem 7, too, the converse does not hold. In fact, the same f and g we gave above, after the proof of Theorem 6 and before stating Theorem 7, shows this here too. In that example, $g \circ f$ is a surjection, but f is not a surjection (although g is a surjection).

Theorem 6 and Theorem 7 together give a similar statement for bijections.

Theorem 8. If $f : A \to B$ and $g : B \to C$ are bijections then $g \circ f$ is also a bijection.

Proof. Suppose $f : A \to B$ and $g : B \to C$ are bijections. Then, by definition, they are both injections and they are both surjections.

Since they are both injections, their composition is also an injection by Theorem 6.

Since f and g are both surjections, their composition is also a surjection by Theorem 6.

So $g \circ f$ is both an injection and a surjection. Therefore, by definition, it is a bijection. □

Once again, the converse does not hold in general, and once again our little functions f and g show this, since $g \circ f$ is a bijection but neither f nor g is a bijection.

There is one important situation where the converse does hold as well.

Theorem 9. Let $f : A \to B$ and $g : B \to C$ be functions where A, B, C are finite sets of the same size. Then $g \circ f$ is a bijection if and only if both f and g are bijections.

Proof. The domains and codomains of f, g and $g \circ f$ are finite and of the same size, as stated. So each of them is a bijection if and only if it is an injection, by our remarks at the end of § 2.7. So it is enough to prove that

$$g \circ f \text{ is an injection} \quad \text{if and only if} \quad \text{both } f \text{ and } g \text{ are injections.}$$

We have already seen that, if f and g are injections, then so is $g \circ f$ (Theorem 6). So it remains to prove that

$$\underline{\text{if}} \quad g \circ f \text{ is an injection} \quad \underline{\text{then}} \quad \text{both } f \text{ and } g \text{ are injections.}$$

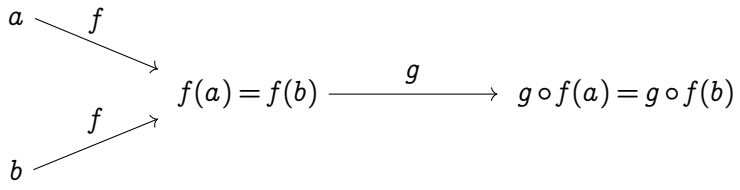
This we do now, by proving the equivalent statement that

$$\underline{\text{if}} \quad \text{at least one of } f \text{ and } g \text{ is } \textit{not} \text{ an injection} \quad \underline{\text{then}} \quad g \circ f \text{ is } \textit{not} \text{ an injection.}$$

Our starting assumption here, that at least one of f and g is not an injection, divides naturally into two cases: (i) f is not an injection, and (ii) g is not an injection. These two cases overlap, which is ok.

Case (i):

If f is not an injection, then by definition there exist distinct $a, b \in A$ such that $f(a) = f(b)$. Then $g(f(a)) = g(f(b))$. So in fact our distinct a, b also give $g \circ f(a) = g \circ f(b)$, so $g \circ f$ is not an injection.



Case (ii):

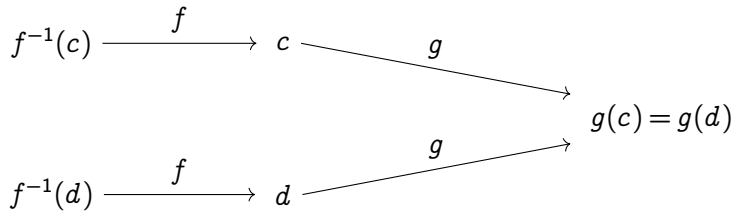
It remains to consider the possibility that g is not an injection. Within this case, we can restrict to cases where f is an injection, since we have just dealt with the possibility that f is not an injection. (Effectively, we are ignoring the overlap between the two cases, since that overlap is covered by Case (i).)

Suppose then that f is an injection. Since its domain and codomain are finite and have the same size, this means it is also a bijection, and therefore has an inverse.

If g is not an injection, then by definition there exist distinct $c, d \in B$ such that $g(c) = g(d)$. Now because f is a bijection, its inverse f^{-1} is defined, has the same domain B, and is also a bijection. So $f^{-1}(c)$ and $f^{-1}(d)$ are both defined and must be distinct since $c \neq d$. Furthermore, $c = f(f^{-1}(c))$ and $d = f(f^{-1}(d))$. So $g(c) = g(d)$ implies $g(f(f^{-1}(c))) = g(f(f^{-1}(d)))$, which may be rewritten

$$g \circ f(f^{-1}(c)) = g \circ f(f^{-1}(d)).$$

But $f^{-1}(c) \neq f^{-1}(d)$, so we have two distinct members of A which are mapped to the same thing by $g \circ f$. So $g \circ f$ is not an injection.



Summarising, we have found that if either of f, g is not an injection then $g \circ f$ is not an injection either. Therefore $g \circ f$ is not a bijection. □

The restriction to finite sets is essential. Consider the following example.

Define $f : \mathbb{N} \to \mathbb{N}$ by $f(n) = 2n$, and define $g : \mathbb{N} \to \mathbb{N}$ by $f(n) = \lfloor n/2 \rfloor$. Neither of these is a bijection: f is an injection but not a surjection, and g is a surjection but not an injection. Yet their composition $g \circ f$ is a bijection, and in fact is the identity function on $\mathbb{N}$:

$$g \circ f(n) \;=\; g(f(n)) \;=\; \lfloor (2n)/2 \rfloor \;=\; \lfloor n \rfloor \;=\; n.$$

Now that we can compose two functions, it is natural to ask about the inverse of the composition. This turns out to be the *reverse* composition of their inverses. This aligns with everyday experience of doing and undoing sequences of tasks: if we wrap up a gift in multiple layers of wrapping paper, then the recipient unwraps the layers in reverse order.

Theorem 10. If $f : A \to B$ and $g : B \to C$ are bijections, then $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$

Proof. Let $f : A \to B$ and $g : B \to C$ be bijections. By Theorem 8, their composition $g \circ f : A \to C$ is also a bijection, so its inverse $(g \circ f)^{-1} : C \to A$ exists. Now consider $f^{-1} \circ g^{-1}$. This exists, since the inverses f^{-1} and g^{-1} exist, since f and g are bijections. We check the two conditions that $f^{-1} \circ g^{-1}$ must satisfy in order to be the inverse of $g \circ f$.

$$\begin{aligned}
(g \circ f) \circ (f^{-1} \circ g^{-1}) &= g \circ (f \circ f^{-1}) \circ g^{-1} && \text{(since function composition is associative)} \\
&= g \circ i_B \circ g^{-1} \\
&= g \circ g^{-1} \\
&= i_C.
\end{aligned}$$

Similarly,

$$\begin{aligned}
(f^{-1} \circ g^{-1}) \circ (g \circ f) &= f^{-1} \circ (g^{-1} \circ g) \circ f && \text{(again using associativity)} \\
&= f^{-1} \circ i_B \circ f \\
&= f^{-1} \circ f \\
&= i_A.
\end{aligned}$$

So $f^{-1} \circ g^{-1}$ is indeed the inverse of $g \circ f$. □

2.10 CRYPTOSYSTEMS

One important application of function composition is to cryptosystems.

Suppose we have

- a **message space**, which is a finite set M of possible **messages** (in the form of strings over some alphabet),

- a **cypher space**, which is a finite set C of strings, which we call **cyphertexts**, and
- a **keyspace**, which is a finite set K whose members we call **keys**.

An **encryption function** is a function $e : M \times K \to C$, and a **decryption function** is a function $d : C \times K \to M$. These are functions of two arguments. Each also gives us a family of single-argument functions. For each key $k \in K$, define the function $e_k : M \to C$ by $e_k(m) = e(m,k)$, and define the function $d_k : C \to M$ by $d_k(c) = d(c,k)$. These are the functions of one argument that you get from e and d by fixing their second argument. So, for a given key $k \in K$, the function e_k encrypts messages just with that one key, while the function d_k decrypts just with that one key. We want decryption to undo encryption, provided the same key is used by each.

We say that (M, C, K, e, d) is a **cryptosystem** if, for all k,

(i) e_k and d_k are bijections, and

(ii) $d_k = e_k^{-1}$.

In practice we will also want some conditions on how easy or hard it is to compute these functions or even to obtain partial information from them.

For convenience, we restrict ourselves to cryptosystems where the cypherspace and message space are the same, i.e., $M = C$. (Most real cryptosystems either have this property or can easily be modified so that they do.)

Suppose we have two cryptosystems with the same message spaces but with keyspaces and encryption/decryption maps that may be different. Call them $\mathcal{C} = (M, M, K, e, d)$ and $\mathcal{C}' = (M, M, K', e', d')$. We would like to *compose* them to make a more complex cryptosystem. For encryption, we want to first encrypt with e and then encrypt further with e'. This is shown in Figure 2.2. For decryption, we want to do the reverse: decrypt using d', then decrypt further with d.

But our definition of function composition (§ 2.9) only applies to functions of one argument. So we need to extend this definition for our encryption and decryption functions.

The **keyed composition** of encryption functions $e : M \times K \to M$ and $e' : M \times K' \to M$ is the function $e' \bullet e : M \times (K \times K') \to M$ defined for all $m \in M$ and $(k, k') \in K \times K'$ by

$$(e' \bullet e)(m, (k, k')) \;=\; e'(e(m,k), k'). \tag{2.1}$$

Similarly, the **keyed composition** of decryption functions $d' : M \times K' \to M$ and $d : M \times K \to M$ is the function $d \bullet d' : M \times (K \times K') \to M$ defined for all $m \in M$ and $(k, k') \in K \times K'$ by

$$(d \bullet d')(m, (k, k')) \;=\; d(d'(m, k'), k). \tag{2.2}$$

Then the **composition** $\mathcal{C}' \circ \mathcal{C}$ of the cryptosystems $\mathcal{C}$ and $\mathcal{C}'$ is $(M, M, K \times K', e' \bullet e, d \bullet d')$.

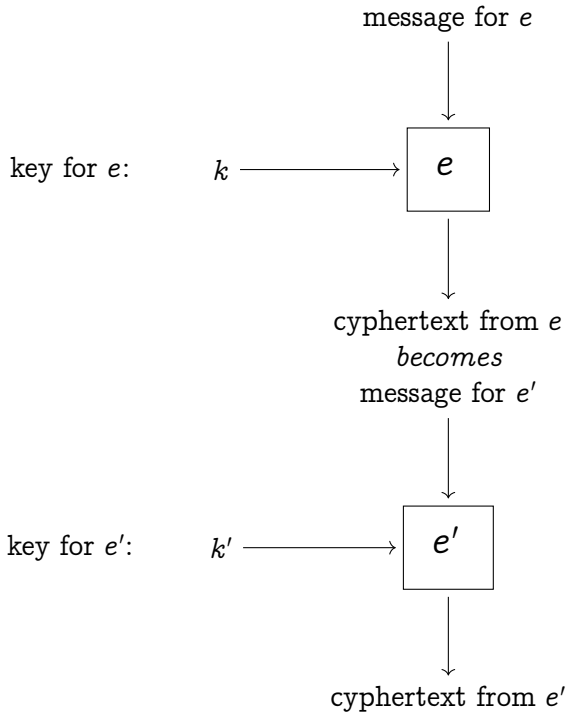


Figure 2.2: Composition of encryption functions

Observe that, in the composition $\mathcal{C}' \circ \mathcal{C}$, the encryption function $e' \bullet e$ gives rise to a family of single-argument functions, as follows. For each key pair (k, k'), the function $e_{(k,k')} : M \to M$ is defined for each $m \in M$ by

$$e_{(k,k')}(m) \;=\; (e' \bullet e)(m, (k, k')).$$

But, by (2.1), this is just $e'(e(m,k), k')$. And we can express this in terms of composition of our single-argument encryption functions:

$$e'(e(m,k), k') \;=\; e'_{k'}(e_k(m))$$

Therefore each single-argument encryption function $e_{(k,k')}$ in the composition $\mathcal{C}' \circ \mathcal{C}$ of the two cryptosystems is itself just the composition of the two single-argument encryption functions:

$$e_{(k,k')} \;=\; e' \circ e.$$

Similarly, the decryption function $d \bullet d'$ yields the family of single-argument functions $d_{(k,k')} : M \to M$, defined for each $m \in M$ by

$$d_{(k,k')}(m) \;=\; (d \bullet d')(m, (k, k')) \;=\; d(d'(m, k'), k) \;=\; d_k(d'_{k'}(m)),$$

and

$$d_{(k,k')} = d \circ d'.$$

Theorem 11. The composition of two cryptosystems is also a cryptosystem.

Proof. We need to prove that, for each key pair (k, k'), the encryption function $e_{(k,k')}$ and the decryption function $d_{(k,k')}$ are both bijections, and that the latter is the inverse of the former.

The fact that they are bijections follows from the fact that $e_k, e'_{k'}, d_k, d'_{k'}$ are all bijections for all k and k' (because C and C' are both cryptosystems) and Theorem 8.

The fact that $d_{(k,k')} = (e_{(k,k')})^{-1}$ follows from Theorem 10. □

In a good cryptosystem, e_k and d_k should be easy to compute if the key k is known, otherwise the intended users will find it hard to use. But, in order for it to be secure, the encryption function should be hard to invert *without* the key: if all an eavesdropper knows is the encrypted text $c \in C$, it should be hard for them to recover either the original message $m \in M$ or the secret key $k \in K$.

Composition can be used to make decryption even harder for an eavesdropper. If each of e and e' is hard to invert, it is reasonable to hope that $e' \circ e$ might be even harder to invert, since an eavesdropper now has to undo the work of both encryption functions, not just one of them. There are many cryptosystems where this is indeed the case. But care is needed, because there are also systems where composition does not give any extra security.

2.11$^{\omega}$ LOOSE COMPOSITION

We might call the above type of function composition **tight composition**, because it requires an "exact fit" between the codomain of the first function to be applied and the domain of the second. There is also a looser form of function composition, which we'll call "loose composition", which is *always defined*, regardless of the domains and codomains of the functions, although in the worst case it might turn out to be the empty function.

Let us try again to define a function based on applying Mother followed by FavouriteUnit, which we considered in § 2.9. We'd like a function that equals FavouriteUnit(Mother(p)) whenever this makes sense. We saw in § 2.9 that the composition FavouriteUnit ∘ Mother is undefined. But we will define a new form of composition, called **loose composition** and denoted by ∘ (a larger, "looser" version of the symbol ∘), under which FavouriteUnit∘ Mother is defined.

To properly define this loose composition, we must specify its domain and codomain. The codomain is easy, since any student's favourite unit will always belong to $\mathbb{U}$ (although the image would be more complicated). But what should we use for this function's domain? Not all mothers are students, so not every person $p \in \mathbb{P}$ gives a valid FavouriteUnit(Mother(p)).

The domain of a *loose composition* is the subset of the domain of the first function (i.e., the one that is applied first) containing everything for which the successive function applications are possible. In other words, it's everything that the first function maps into the domain of the second function. For FavouriteUnit ∘ Mother, this means every person whose mother is a student, since for any such person, the function Mother produces a member of $.

In general, the **loose composition** $g \circ f$ of two functions $f : A \to B$ and $g : C \to D$ is defined by

$$\begin{aligned} g \circ f \ : \ \{x \in A : f(x) \in C\} &\longrightarrow D, \\ g \circ f(x) &= g(f(x)). \end{aligned}$$

Note that, as usual for our composition notation, f is applied *first*, and then the result is given to g.

The loose composition of two functions is *always* defined. It might sometimes be useless: if $B \cap C = \emptyset$, then nothing that f produces is in the domain of g, so the composition $g \circ f$ is just the empty function.

It will be seen from the definition of loose composition that it is harder to work out the domain of loose composition than it is to work out the domain of tight composition. This makes it a bit harder to use in practice. Tight composition has the advantage that the question of whether the composition is defined can be answered solely by looking at the appropriate codomain and domain; you do not need to study the rule at all. This makes it much easier to work with, and from a computing perspective, much easier to use as a specification of a task based on combining two tasks.

In this unit, we will use tight composition rather than loose composition.

2.12 COUNTING FUNCTIONS

We often want to count functions of various types. You might want to determine the amount of time an algorithm takes, if the algorithm has to search through all functions of some type. You might want to determine the amount of space that a collection of data requires, if the data items correspond to functions. You might want to determine the probability that a random function has some particular property.

Suppose $f : A \to B$, where the sets have sizes $|A| = m$ and $|B| = n$. How many functions of this type are there? The domain has m elements, and each of them is mapped to one, and only one, member of the codomain B. So there are n possibilities for $f(x)$ for each element $x \in A$. These choices are independent; there is no requirement for the various values of x to differ from each other or to be related in any other way. So we have m independent choices, each being among n possibilities. This means we have n^m functions.

Suppose now we require f to be an injection. We still have m elements in the domain, and for each of these, we must still choose exactly one member of the codomain.

But now these choices are no longer independent, since as soon as one member of the codomain is chosen, that member is no longer available for any other member of the domain. Suppose we make these choices in order, and to help describe this, we suppose that the elements of A are enumerated as $a_1, a_2, \ldots, a_m$. Now, a_1 can be mapped to any of the n elements of B. Then, a_2 can be mapped to any element of B except $f(a_1)$, so it has $n-1$ choices. Then, a_3 can be mapped to any element of B except $f(a_1)$ and $f(a_2)$, so it has $n-2$ choices. And so on. Finally, a_m can be mapped to any element of B except $f(a_1), f(a_2), \ldots, f(a_{m-1})$, so it has $n-(m-1)$ choices, which is $n-m+1$ choices. So the total number of injections is

$$n(n-1)(n-2)\cdots(n-m+1).$$

This formula also copes nicely with the possibility that $m > n$. In that case, we know that the codomain is too small to allow any injections from the domain at all, so the answer should be 0, and that is indeed what the formula gives, since one of the factors will be 0.

Requiring f to be a surjection needs some more care and will be considered later.

If we require f to be a bijection, then that's the same as requiring f to be an injection whose codomain and domain are the same size, since both sets are finite. (See the end of § 2.7 on p. 50.) So this is the same as the injective case when $m = n$. So the number of bijections is just $n(n-1)(n-2)\cdots(n-n+1)$, which is $n!$.

Recall that, when a bijection maps a finite set to itself, it is called a *permutation* of that set (§ 2.7). So the number of permutations of an n-element set is $n!$.

2.13 BINARY RELATIONS

We often want to know when objects of one type are related in some specific way to objects of another type. For example, in considering sets of people, we might like to know who is a parent of whom. In mobile communications, systems that manage calls would use information on which smartphones contacted which cellphone towers. Some ecologists monitor predator-prey relationships among species in a geographic area. In timetabling Monash classes in a given semester, we want to know which pairs of units have at least one student in common, so that we can try to schedule classes in those units at different times.

These situations, and very many others, can be modelled by *binary relations*.

A **binary relation** consists of two sets A and B and a set of ordered pairs (a, b) where $a \in A$ and $b \in B$. The ordered pairs are used to state which members of A and B are related to each other in the required way.

Recall that the Cartesian product $A \times B$ is the set of *all* ordered pairs in which the first and second members of the pair belong to A and B respectively. This gives us a very succinct way to restate our definition. A **binary relation** consists of two sets A

and B and a subset of $A \times B$. We sometimes say that the binary relation is **from** A **to** B, and we may still call A the **domain** and B the **codomain**.

The two sets might be the same. A **binary relation on** a set A is a binary relation from A to itself. Each of the two sets is A, so that the relation is a subset of $A \times A$.

A binary relation is also called a **binary predicate** or a **predicate with two arguments**.

If R is the name of a binary relation, then we write xRy or $R(x, y)$ or $(x, y) \in R$ to mean that (x, y) is one of the ordered pairs in R. The notation xRy is an example of *infix* notation, where the name of the operation/function/relation is placed *between* the two things it links. The notation $R(x, y)$ is a further example of prefix notation. (Recall the discussion of prefix, infix and postfix notation on p. 47.)

For example, the Parent relation is a relation on the set $\mathbb{P}$ of all people (so the two sets are the same in this case), and the pair of people (p, q) belongs to this relation if q is a parent of p. Members of the Parent relation include:

$$\begin{array}{rcl}
(\text{Ada Lovelace} & , & \text{George Gordon Byron}), \\
(\text{Annie Jump Cannon} & , & \text{Wilson Cannon}), \\
(\text{John Ferrier Turing} & , & \text{Sara Turing}), \\
(\text{John Ferrier Turing} & , & \text{Julius Turing}), \\
(\text{Alan Mathison Turing} & , & \text{Sara Turing}), \\
(\text{Alan Mathison Turing} & , & \text{Julius Turing}), \\
\vdots & & \vdots
\end{array}$$

So we may write, for example,

$$(\text{Ada Lovelace} , \text{George Gordon Byron}) \in \mathsf{Parent},$$

or using infix notation,

$$\text{Ada Lovelace} \quad \mathsf{Parent} \quad \text{George Gordon Byron},$$

or using prefix notation,

$$\mathsf{Parent}(\text{Ada Lovelace}, \text{George Gordon Byron}).$$

For the mobile communications example, the two sets are different, namely a set of smartphones and a set of cellphone towers, and the relation includes pairs like

$$(\text{Catherine Deneuve's phone}, \text{the Eiffel Tower}).$$

It may be tempting to use binary relation names in the same way we use function names. Recall the functions Mother and Father, which enable us to write statements like

$$\begin{aligned} \text{Mother}(\text{Alan Turing}) &= \text{Sara Turing}, \\ \text{Father}(\text{Alan Turing}) &= \text{Julius Turing}. \end{aligned}$$

But we will not write "Parent(Alan Turing)"; such notation would treat Parent as a function of people, when in fact "the parent" of a person is not, in general, uniquely defined.

We saw in § 2.1.3$^{\alpha}$ that one way to specify the rule of a function $f : A \to B$ is to give its graph, i.e., to give all its ordered pairs $(x, f(x))$, which all belong to $A \times B$. So a function is a type of binary relation. To be precise, a function with domain A and codomain B is a binary relation on A and B in which, for every $a \in A$, there is a unique $b \in B$ such that (a, b) belongs to the relation. To put it the other way round (and less formally), a binary relation is a "function" where we drop the requirement that each member of the domain gives exactly one member of the codomain. With this latter viewpoint, a binary relation is sometimes called a "many-valued function" because a single member of the domain can yield many different values in the codomain (whereas normal *functions* are just single-valued). But we will not refer to "many-valued functions" because it is a contradiction in terms: a function, *by definition*, cannot be many-valued in that sense.

Examples of binary relations on sets of numbers include $=$, $\leq$, $\geq$. Examples of binary relations on sets of *sets* include $=$, $\subseteq$, $\supseteq$.

One common source of binary relations is network structures. Networks consist of nodes with links between some pairs of nodes. For example:

- In a social network, the nodes are people and the links represent which people know which other people.
- In a communications network, nodes represent the communicating devices and the links may represent messages sent from one device to another.
- In an ecological network, nodes represent species and links represent predator-prey relations among the species.
- In Monash timetabling, nodes represent the units running in a given semester, and links represent pairs of units that have at least one student in common for that semester. A small fragment of this network is shown in Figure 2.3. In this example, MTH1030 and MTH1035 have no students in common (because it is prohibited to enrol in both of them), so their classes may be at the same time (or at different times), and we represent this lack of restriction by the absence of a link between the corresponding nodes. But every other pair of these units has some students that do both of them, so every other pair of nodes has a link between them.

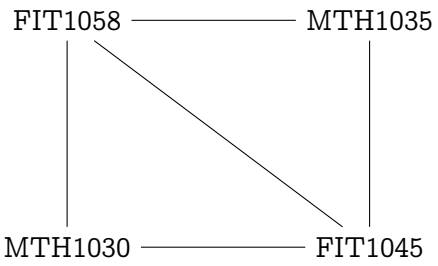


Figure 2.3: A fragment of the Monash timetabling network

- In a software system, nodes represent software components of some kind (modules, programs, ...), and the links represent some mechanism for passing information from one component to another.

In each of these cases, we have a set of nodes, and the set of links between them can be represented by a binary relation.

Another common source of binary relations is tables of data. Whenever you write a table with two columns, you are specifying a binary relation whose ordered pairs (x, y) correspond to the rows of the table, with x and y being the entries in the first and second column respectively. If you have a table with more than two columns, then often taking some (or perhaps any) pairs of columns will give you a binary relation in the same way. Similar remarks apply to data stored in other ways that may be thought of as tables, such as in spreadsheets and databases.

Important operations which you might want to do, when using a binary relation R on sets A and B, include:

- Determine the set of all members of A that actually appear as the first member of a pair in R. This is $\{x \in A : (x, y) \in R \text{ for some } y \in B\}$.

- Determine the set of all members of B that actually appear as the second member of a pair in R. This is $\{y \in B : (x, y) \in R \text{ for some } x \in A\}$.

- Given x, determine everything that is related to it by R. This is $\{y \in B : xRy\}$. This set is often denoted by $R(x)$, but we must not confuse this with the notation for the value of a function, since $R(x)$ is typically not just one single value in B, but rather a set of values, and it might even be empty.

- Given y, determine everything that is related to it by R. This is $\{x \in A : xRy\}$. This is often denoted by $R^{-1}(y)$. Again, take care to avoid confusion with the value of an inverse function, since $R^{-1}(y)$ is in general some set of values that are each related to y rather than just one value.

- For each of these sets, we may want to determine its size.

Let R be any binary relation from A to B. Its **inverse** R^{-1} is the binary relation from B to A defined by

$$yR^{-1}x \iff xRy.$$

So the inverse is constructed by just swapping the roles of the domain and codomain and reversing all the pairs. In the special case when R is actually a function, this is just the definition of the inverse of a function (see § 2.8). We can now talk of the inverse of *any* function, not just of bijections, but we have to remember that, if f is not a bijection, then f^{-1} is not a function (although it is a valid binary relation).

2.14 PROPERTIES OF BINARY RELATIONS

We now meet some important types of binary relations.

Let R be a binary relation on a set A.

We say R is **reflexive** if, for all $x \in A$, we have aRa. In other words, everything is related to itself.

Examples of reflexive binary relations include equality, $\leq$, $\geq$, $\subseteq$, $\supseteq$ (these latter two because these versions of the subset relation allow improper subsets).

Using the set of all people again, let knows be the binary relation that holds when one person knows another. It contains an ordered pair (p, q) of people precisely when person p knows person q. It seems reasonable to describe knows as reflexive, as everyone knows themselves (more or less).

Binary relations that are not reflexive include $<$, $>$, $\subset$, $\supset$. The relation Parent is also not reflexive.

We say R is **symmetric** if, for all $x, y \in A$, xRy implies yRx.

The relation $=$ is symmetric, but the relations $<$, $\leq$, $>$, $\geq$, $\subseteq$, $\subset$, $\supset$, $\supseteq$ are not symmetric. The relation knows is symmetric, assuming that we are using the word "knows" for relationships where each knows the other. The relation Parent is not symmetric. The hyperlink relation on the set of all webpages — consisting of all pairs (p, q) where webpage p has a hyperlink to webpage q — is also not symmetric.

We say R is **antisymmetric** if, for all $x, y \in A$, xRy and yRx together imply $x = y$. This is not the same as just being *not symmetric*.

The relations $=$, $\leq$, $\geq$, $\subseteq$, $\supseteq$ are antisymmetric. This may seem slightly surprising for $=$, since it is also symmetric and the two terms sound opposite. But this example illustrates that it is indeed possible for a binary relation to be both symmetric and antisymmetric. Are there any other cases where this happens?

The hyperlink relation on webpages is not antisymmetric. This illustrates the fact that it is possible for a binary relation to be neither symmetric nor antisymmetric.

The relation knows is also not antisymmetric.

We say R is **transitive** if, for all $x, y, z \in A$, xRy and yRz together imply xRz.

The relations $=$, $<$, $\leq$, $>$, $\geq$, $\subset$, $\subseteq$, $\supset$, $\supseteq$ are all transitive, but $\neq$ is not. The relation knows is not transitive: a "friend of a friend" is not necessarily your friend! The relation Parent is not transitive either.

2.15 COMBINING BINARY RELATIONS

Viewing a binary relation as a set of ordered pairs enables us to apply ordinary set operations on them. Suppose R and S are both binary relations from A to B. Then their **union** $R \cup S$ is the binary relation from A to B containing every pair that belongs to at least one of R and S.

$$R \cup S = \{(x, y) : xRy \text{ or } xSy\}.$$

For example,

$$\mathsf{Parent} = \mathsf{Mother} \cup \mathsf{Father}.$$

The **intersection** $R \cap S$ is the binary relation from A to B containing every pair that belongs to both R and S.

$$R \cap S = \{(x, y) : xRy \text{ and } xSy\}.$$

If R is a binary relation from A to B, and S is a binary relation from B to C, then the **composition** $S \circ R$ is the binary relation from A to C whose set of ordered pairs is

$$\{(x, z) : \text{there exists } y \in B \text{ such that } xRy \text{ and } yRz\}.$$

In the special case when R and S are both functions, their composition is just their composition as functions, using the definition of function composition given in § 2.9.

Let's investigate the composition of any relation R from A to B with its inverse relation R^{-1}. The composition $R^{-1} \circ R$, which goes from A to itself, satisfies

$$\begin{aligned} (x, z) \in R^{-1} \circ R &\iff \text{there exits } y \text{ such that } xRy \text{ and } yR^{-1}z \\ &\iff \text{there exits } y \text{ such that } xRy \text{ and } zRy \\ &\qquad\qquad (\text{using the definition of } R^{-1}). \end{aligned}$$

So, in the composition $R^{-1} \circ R$, two elements of A are related by $R^{-1} \circ R$ precisely when there is an element of B that they are both related to by R.

Similarly, the composition $R \circ R^{-1}$, which goes from B to itself, relates two elements of B precisely when there is an element of A that is related to both those elements of B.

Binary relations, like functions, can be composed with themselves. Consider again the binary relation knows on the set of all people. In the composition knows $\circ$ knows, two people are related if they have a mutual acquaintance, i.e., they each know someone who knows the other. We can extend this. In the composition knows $\circ$ knows $\circ$ knows, one

person is related to another if they know someone who knows someone who knows the other. The five-fold composition

$$\mathsf{knows} \circ \mathsf{knows} \circ \mathsf{knows} \circ \mathsf{knows} \circ \mathsf{knows} \circ \mathsf{knows}$$

is said to relate every pair of people on Earth. This is the principle known as "six degrees of separation"; it uses the knows relation *six* times, with *five* compositions.

As for function composition, we can use exponents in parentheses to denote composition of relations with themselves: $R^{(n)}$ is the composition of n copies of R. So the composition we wrote above, for six degrees of separation, could be written $\mathsf{knows}^{(6)}$.

Sometimes, for a binary relation R on a set A, we may want to go further and identify every pair of elements of A that are linked by *any* chain, no matter how long, of pairs in R.

The **transitive closure** of a binary relation R on a set A is the unique binary relation R^+ on A such that, for all $x, y, z \in A$,

(i) if xRy then xR^+y;

(ii) if xR^+y and yRz then xR^+z;

(iii) if xRy and yR^+z then xR^+z.

Actually, you can drop either the second or third (but not both) of these conditions from the definition. You cannot drop the first condition, which gets the transitive closure process started.

The transitive closure R^+ certainly contains all pairs in R, by condition (i) in its definition. It also contains all pairs in $R^{(2)}$, and all pairs in $R^{(3)}$, and so on. So the set of pairs in the transitive closure is given by

$$R^+ = R \cup R^{(2)} \cup R^{(3)} \cup R^{(4)} \cup \cdots,$$

which may be written more succinctly as

$$R^+ = \bigcup_{i=1}^{\infty} R^{(i)}.$$

The transitive closure of knows identifies whenever two people can be linked by an arbitrarily long chain of social connections. Under the hypothesis of six degrees of separation,

$$\mathsf{knows}^+ = \mathsf{knows}^{(6)}$$

and every pair of people on Earth are linked in this way.

The transitive closure of a binary relation is always transitive, hence the name.

2.16 EQUIVALENCE RELATIONS

It seems fair to regard *equality* as the most fundamental binary relation. No matter what kind of objects we are working with, we want to be able to say whether or not two of them are really the same. If we cannot do even that, it is hard to imagine having any useful discussion about such objects at all. So, even if there is no other relationship to speak of among the objects we are discussing, there must be an equality relation. We will always feel free to use equality on any set at all, without announcing its existence beforehand.

In many situations, different objects may be treated as being equivalent for *some* purposes even though they must be treated differently for other purposes. For example, if two different students are in the same weekly tutorial class, then they can be treated as equivalent for FIT1058 timetabling purposes, even though their other subjects may be different so that they cannot be treated as equivalent for other timetabling purposes. Define the binary relation $\overset{1058}{=\!=}$ on the set $\mathbb{S}$ of all students by

$$p \overset{1058}{=\!=} q \quad \Longleftrightarrow \quad p \text{ and } q \text{ are in the same FIT1058 tutorial,}$$

for any $p, q \in \mathbb{S}$.

If a binary relation is to capture some notion of "equivalence", what properties should it have? We can use equality as a guide, since "equivalence" should be like equality but a bit "looser" in that two things can be equivalent without being identical. We have just seen that the equality relation is reflexive, symmetric, and transitive. This is also what we would expect of a binary relation that tells us when two things are equivalent. Every object, of any kind, is certainly equivalent to itself; if one object is equivalent to another, then the latter object must also be equivalent to the former; and if an object is equivalent to another, which in turn is equivalent to a third object, then the first object must also be equivalent to the third object.

Equality is also antisymmetric. But we will not add this to our requirements of equivalence in general, since antisymmetry requires that if an object is equivalent to another, which in turn is equivalent to our first object, then the two objects must actually be equal. This would be tantamount to saying that equivalence implies equality, which would mean that we have no form of equivalence other than equality itself, which is too narrow.

With these thoughts in mind, we make the following definition.

An **equivalence relation** is a binary relation that is reflexive, symmetric, and transitive.

It is routine to check that the binary relation $\overset{1058}{=\!=}$ is an equivalence relation.

Other examples of equivalence relations:

- Let $m \in \mathbb{N}$. Two integers x, y are **congruent modulo** m if $x - y$ is a multiple of m (often written $m \mid (x-y)$). We write this equivalence as $x \equiv y \pmod{m}$.
- Two real numbers x, y are **equivalent in integer part** if, when each is rounded to the nearest integer, they become equal. We write this as $x \overset{\text{int}}{=\!=} y$.
- Two triangles in the plane are **congruent** if one can be moved onto the other by some sequence of translations, rotations and reflections.
- Every function $f : A \to B$ defines an equivalence relation as follows. If $x, y \in A$, we write $x \sim_f y$ if $f(x) = f(y)$. This relation $\sim_f$ is an equivalence relation on A.
- The transitive closure of any reflexive symmetric binary relation is an equivalence relation.
- For any relation R, the relations $R \circ R^{-1}$ and $R^{-1} \circ R$ are equivalence relations.

Let R be an equivalence relation on a set A. An **equivalence class** of A under R is a nonempty subset $X \subseteq A$ in which

(i) whenever $x, y \in X$, we have xRy;

(ii) there are no $x \in X$, $z \notin X$ such that xRz.

So all members of X are equivalent to each other, but no member of X is equivalent to anything outside X. This condition may be rephrased as saying that X is a maximal subset of A in which all elements are equivalent. (Reminder: the word "maximal" does not mean that X is largest in size among all those with this property. Rather, it means that X cannot be enlarged while maintaining this property; in other words, no proper superset of X has the property. See § 1.7.)

Recalling our earlier examples of equivalence relations:

- For the binary relation $\overset{1058}{=\!=}$, the equivalence classes are the FIT1058 tutorials (or, more precisely, the sets of students allocated to each tutorial).
- For congruence modulo m, the equivalence classes are the sets $\{km + r : k \in \mathbb{Z}\}$, for each $r \in \{0, 1, \ldots, m-1\}$. These sets are called the **residue classes modulo** m. There is one such set for each $r \in \{0, 1, \ldots, m-1\}$. For example, if $m = 2$, then $r \in \{0, 1\}$ and we have two equivalence classes: the set of all even integers (for $r = 0$), and the set of all odd integers (for $r = 1$).
- For the relation $\overset{\text{int}}{=\!=}$, the equivalence classes are the intervals $[n - \frac{1}{2}, n + \frac{1}{2})$ for all $n \in \mathbb{Z}$.

- For congruent triangles, the equivalence classes each contain all triangles of one particular shape and size (so they all have the same angles and side lengths, but otherwise can be in any location and orientation in the plane).
- For $\sim_f$, the equivalence classes are the preimages $f^{-1}(y) = \{x : f(x) = y\}$ of each $y \in \operatorname{im} f$.

In each of these cases, the equivalence classes divide up the set A neatly, so that each element of A belongs to exactly one equivalence class. These are manifestations of a general phenomenon.

Theorem 12. The equivalence classes of an equivalence relation R on a set A form a partition of A.

Proof. To show that the equivalence classes form a partition of A, we need to show that (a) every member of A belongs to an equivalence class, and (b) no two different equivalence classes overlap. (Note that equivalence classes are nonempty by definition.)

(a)

Let $x \in A$. Define X to be the set of all members of A that are equivalent to x under our equivalence relation R:

$$X = \{y \in A : xRy\}.$$

We claim that this is an equivalence class of R that contains x. We prove parts (i) and (ii) of the definition of equivalence relation, in turn.

Firstly, suppose $y, z \in X$. By definition of X, this implies that xRy and xRz. By symmetry and transitivity of R, it follows that yRz. So every pair of members of X are equivalent, satisfying the first condition for an equivalence class.

Now suppose that $v \in X$, $w \notin X$ such that vRw. The fact that $v \in X$ implies that xRv. This, combined with vRw and transitivity, gives xRw. Therefore $w \in X$. This contradicts the assumption that $w \notin X$. So there cannot be any $v \in X$, $w \notin X$ such that vRw. So the second condition for an equivalence class is satisfied.

We conclude that X is an equivalence class.

Furthermore, X clearly contains x, since xRx by reflexivity.

So every member of A belongs to an equivalence class.

(b)

Suppose X and Y are overlapping equivalence classes of R. So $X \cap Y \neq \emptyset$. Let $x \in X \cap Y$.

We claim that $X = Y$.

To do this, we consider $Y \setminus X$ and $X \setminus Y$ and show that they are both empty, which implies that $X = Y$. (See Theorem 4 and (1.17).)

Consider first the possibility that $Y \setminus X \neq \emptyset$. Suppose $y \in Y \setminus X$. Because X is an equivalence class and $y \notin X$, it follows that $(x, y) \notin R$. But because Y is an equivalence

class and $x, y \in Y$, it follows that $(x, y) \in R$. So we have a contradiction in this case. Therefore $Y \setminus X \neq \emptyset$ is impossible, so $Y \setminus X = \emptyset$.

It remains to consider the possibility that $X \setminus Y \neq \emptyset$. But the argument here is the same as that of the previous paragraph, with X and Y interchanged. So $X \setminus Y = \emptyset$.

So we have $Y \setminus X = X \setminus Y = \emptyset$. This implies that $X = Y$.

So the only way two equivalence classes can overlap is if they are identical.

So we have shown that every member of A belongs to exactly one equivalence class. So the equivalence classes of R form a partition of A. □

2.17 RELATIONS

Binary relations are *binary* in the sense that they consists of ordered *pairs*. It is often useful to consider relations whose members have more objects.

A **ternary relation** on sets A, B, C is a set of triples (x, y, z) such that $x \in A$, $y \in B$ and $z \in C$. (Triples written in parentheses are always considered to be ordered.) In other words, it is a subset of $A \times B \times C$. For example, a set of points in real three-dimensional space is a ternary relation on $\mathbb{R}$. A list of names in which each is listed as

first name *middle name* *surname*

could be represented as a ternary relation in which each triple has the form

(*first name*, *middle name*, *surname*).

Many Chinese names could be represented in a ternary relation with triples

(*family name*, *personal name*, *generation name*).

Dates can be represented by (day, month, year) triples, so they too can be represented naturally in a ternary relation.

More generally, an **n-ary relation** on sets $A_1, A_2, \ldots, A_n$ consists of n-tuples $(x_1, x_2, \ldots, x_n)$ where $x_i \in A_i$ for all $i \in \{1, \ldots, n\}$. In other words, it is a subset of $A_1 \times A_2 \times \cdots \times A_n$. The set A_i is called the **i-th domain**.

When you use a table with n columns, its rows represent the n-tuples in an n-ary relation. For each i, the set A_i is a set that contains all objects appearing in the i-th column (and is allowed to contain more, much as a codomain of a function is allowed to contain more than just the image). So, when you work with spreadsheets, you are usually working with n-ary relations whose members represent the rows of the spreadsheet. Relations are fundamental in databases, so much so that the term *relational database* is used for one of the most widely used types of database.

An n-ary relation is also called an **n-ary predicate** or a **predicate with n arguments**.

2.18 COUNTING RELATIONS

How many binary relations from A to B are there, where A and B are finite sets?

Put $m := |A|$ and $n := |B|$.

A binary relation is just a subset of $A \times B$, so the number of binary relations from A to B is just the number of subsets of $A \times B$. This is just the size of the power set $\mathcal{P}(A \times B)$, which is

$$2^{|A \times B|},$$

by (1.1). But $|A \times B| = mn$, by (1.19). So

$$\#\text{ binary relations from } A \text{ to } B \;=\; 2^{mn}.$$

In the special case when $A{=}B$, with $|A| = m$, we have

$$\#\text{ binary relations on } A \;=\; 2^{m^2}.$$

Now let's look at more general relations. How many n-ary relations are there on sets $A_1, A_2, \ldots, A_n$?

Put $m_i := |A_i|$ for each $i = 1, 2, \ldots, n$.

An n-ary relation is just a set of n-tuples from $A_1 \times A_2 \times \cdots \times A_n$. So the number of n-ary relations is just the size of the power set of $A_1 \times A_2 \times \cdots \times A_n$:

$$\#\ n\text{-ary relations on } A_1 \ldots, A_n \;=\; 2^{|A_1 \times A_2 \times \cdots \times A_n|} \;=\; 2^{m_1 m_2 \cdots m_n}.$$

In the special case when all sets are the same (say, $A_i = A$ for all i) and have size m, we have

$$\#\ n\text{-ary relations on } A \;=\; 2^{m^n}.$$

2.19 EXERCISES

1. Let A be a finite set. How many indicator functions with domain A are there?

2. Let A and B be any sets. Express the indicator functions of each of the following in terms of the indicator functions χ_A and χ_B.

(a) $\overline{A}$

(b) $A \cap B$

(c) $A \setminus B$

(d) $A \triangle B$

3. If two functions $f : A \to \mathbb{R}$ and $g : A \to \mathbb{R}$ have the same domain A and give real numbers as their values, then their **sum** $f + g : A \to \mathbb{R}$ is defined for all $x \in A$ by

$$(f+g)(x) \;=\; f(x)+g(x).$$

What is the sum of all the indicator functions of all the subsets of a finite set?

4. Draw a Venn diagram showing each of the following sets of functions and the relationships between them: functions, injections, surjections, bijections, identity functions, binary relations, ternary relations, relations.

5. Functions can be viewed as sets of ordered pairs, so we can combine functions using set operations. The result will be a binary relation but, depending on the operation, it may or may not be another function.

Suppose $f : A \to B$ and $g : C \to D$ are functions. Which of the following is always a function?

$$f \cap g; \quad f \cup g; \quad f \setminus g; \quad g \setminus f; \quad f \triangle g; \quad f \times g\,.$$

For each that is always a function, give its definition in the usual form, including showing how the rule depends on f and g. For each that is not necessarily a function, explain why this is the case (e.g., with the help of examples for f and g under which the operation does not give a function).

6. Suppose that $f : A \to B$ and $g : C \to D$ are functions with disjoint domains: $A \cap C = \emptyset$. Is the disjoint union $f \sqcup g$ always a function? If so, give its definition in terms of f and g. If not, why not?

7. Give an example of a function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which is an injection. Even better, give one which is a bijection.

8. A **bitstring** is a string over the binary alphabet {0,1}; in other words, it is a member of $\{0,1\}^*$.

(a) Give a bijection from $\{0,1\}^*$ to $\mathbb{N}$.

Now let $\{0,1\}^{*+}$ denote the set of all finite nonempty tuples (or sequences) of bitstrings. Examples of members of $\{0,1\}^{*+}$ include:

$$(011, 1, 10001), \quad (10, 10, 10, 10), \quad (\epsilon, 1011), \quad (1010010001).$$

(b) (Challenge) Give a bijection from $\{0,1\}^{*+}$ to $\{0,1\}^*$.

9. Let A be a set of n elements and let $x \in A$. We say that a function $f : A \to A$ **fixes** x if $f(x) = x$. We also say in this case that x is a **fixed point** of f.

This says nothing about what f does to other elements of A; it is possible that some other elements are fixed by f too, or that none are. All this definition requires is that f sends x to itself.

Now let $X \subseteq A$. The function f **fixes** X if it fixes all elements of x. So, for all $x \in X$, we have $f(x) = x$. In other words, the restriction of f to X is just the identity function on X:

$$f|_X = i_X.$$

Members of $A \setminus X$ may or may not be fixed by f.

Let $\mathrm{Fix}(X)$ be the set of bijections on A that fix X, and let F be the set of all bijections on A that fix at least one element of A.

Using Exercise 1.13 (i.e., Exercise 13 in Chapter 1), express $|F|$ in terms of the set sizes $|\mathrm{Fix}(X)|$ using all $X \subseteq A$.

10. For each $k \in \{1, \ldots, n\}$ and any subset $X \subseteq A$ with $|X| = k$, write f_k for the number of bijections on A that fix X.

(a) Why don't we include X in the notation f_k, given that its definition refers to X?

(b) Give an expression for f_k.

(c) Express $|F|$ in terms of $f_1, f_2, \ldots, f_n$, and then use your expression for f_k to give an expression for $|F|$.

(d) Hence express the number of fixed-point-free bijections on A in terms of $f_1, f_2, \ldots, f_n$. A function is **fixed-point-free** if it has no fixed points.

(e) For each of the following sets, what *proportion* of all bijections on the set are fixed-point-free? $\{0, 1\}$; $\{\spadesuit, \clubsuit, \diamondsuit, \heartsuit\}$; the set of ten decimal digits; the 26-letter English alphabet.

You may need to use a program like Wolfram Alpha, or a spreadsheet, to help calculate the latter two.

(f) What is the largest set for which you can determine this proportion, and what is the value of the proportion for that set? (Use a spreadsheet or a program if you wish.)

11.

(a) Write down all surjections from $\{1, 2, 3, 4\}$ to $\{a, b, c\}$. Compare your answer with Exercise 1.20.

(b) Define a surjection from

$$\{\text{surjections from } \{1, 2, 3, 4\} \text{ to } \{a, b, c\}\}$$

to

$$\{\text{partitions of } \{1,2,3,4\} \text{ into three parts}\}.$$

(c) What sizes can the preimages of members of its image be?

12. Suppose $f : A \to B$ and $g : C \to D$ are bijections, $A \cap C = \emptyset$ and $B \cap D = \emptyset$. Since $A \cap C = \emptyset$, the disjoint union $f \sqcup g$ exists. Show that $f \sqcup g$ is a bijection and express the inverse of $f \sqcup g$ in terms of the inverses of f and g.

13. If the function f has an inverse function, which of its iterated compositions $f^{(k)}$ have inverse functions (where $k \in \mathbb{N}$)?

14. Determine all functions from $\{1,2,3\}$ to itself whose indefinitely iterated composition is a constant function (i.e., $f^{(k)}$ is a constant function if k is large enough).

For a challenge: investigate when this happens in general. Try to characterise those functions $f : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$ such that, for large enough k, the iterated composition $f^{(k)}$ is constant.

15. This exercise is about the **Caesar slide** cryptosystem, one of the oldest and simplest cryptosystems. It is not secure, but its core operation is used in many stronger and more complex cryptosystems.

Caesar slide encrypts a message by sliding each letter along the alphabet, rightwards, by some fixed number of steps. This fixed number is the key, k, and the same amount of sliding is done to each letter of the message. The sliding is done with wrap-around, so if sliding ever takes you beyond the end of the alphabet, then you resume sliding at the start of the alphabet. In effect, we treat the alphabet as a circle rather than a straight line.

For example, if the message is[5]

```
thefamilyofdashwoodhadlongbeen
```

and the key is $k = 3$ (Julius Caesar's favourite), then the cyphertext is

```
wkhidplobrigdvkzrrgkdgorqjehhq
```

Here, sliding `t` along the alphabet by 3 steps gives `w`. We see the wrap-around when encrypting the letter `y`, since sliding it step-by-step gives `z`, then (wrapping around) `a`, then `b`.

Decryption is the reverse of encryption, meaning that we slide k steps leftwards instead of rightwards, again with wrap-around.

We now give a formal definition of this cryptosystem, using the definition of cryptosystems given in §2.10.

5 from the first sentence of *Sense and Sensibility* by Jane Austen, first published by Thomas Egerton in London in 1811.

- The message space M is the set of all strings of English lower case letters (without blanks).
- The key space K is the English lower-case alphabet, $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots, \mathsf{y}, \mathsf{z}\}$.
- The cypher space C is the same as the message space: $C = M$.
- The encryption and decryption functions, to be defined below, need to use the **letter addition** operation defined by the following table.

+	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
b	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a
c	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b
d	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c
e	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d
f	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e
g	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f
h	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g
i	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h
j	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i
k	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j
l	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k
m	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l
n	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m
o	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n
p	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
q	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
r	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
s	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
t	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
u	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
v	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
w	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
x	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
y	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
z	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y

In effect, letters correspond to numbers in $\{0, 1, 2, \ldots, 24, 25\}$, and when we do the letter addition $\alpha + \beta$, we start at α and slide to the right (with wrap-around as needed) by a number of steps given by β. Letter addition is commutative and associative. We can also define letter subtraction, where we slide to the left instead of the right.

- The Caesar slide encryption function $e : M \times K \to M$ is defined for any message $m \in M$ and key $k \in K$ as follows. Let n be the length of m and let its letters be $m_1, m_2, \ldots, m_n$, so we can write $m = m_1 m_2 \cdots m_n$. Then $e(m,k)$ is the string $c = c_1 c_2 \cdots c_n$ of n letters where, for each i, the i-th letter c_i is obtained from the i-th letter of the message by adding the key letter to it, using the wrap-around method described above. Formally,

$$\begin{aligned} e(m,k) &= c_1 c_2 \cdots c_n, \\ c_i &= m_i + k. \end{aligned}$$

  The addition here is *letter addition*. Note that, for each i, the *same* key letter is used. Although message and cypher letters may (and usually do) change as you go along the message, the key letter used for encryption does not change.

- The Caesar slide decryption function $d : M \times K \to M$ is defined for any cyphertext $c = c_1 c_2 \cdots c_n \in M$ and key $k \in K$ as follows. Its value $d(c,k)$ is the string $m = m_1 m_2 \cdots m_n$ where, for each i, the i-th letter m_i is obtained from the i-th letter of the cyphertext by subtracting the key letter to it, again using wrap-around. Formally,

$$\begin{aligned} d(c,k) &= m_1 m_2 \cdots m_n, \\ m_i &= c_i - k. \end{aligned}$$

  Again, we use the same key letter at all positions.

Now refer to the definitions of *embeddable*, *equivalent* and *idempotent* given in Assignment 1.

Prove that the Caesar slide cryptosystem is idempotent.

16. If f is an injection and f^{-1} is its inverse *relation*, what can you say about $f \circ f^{-1}$ and $f^{-1} \circ f$? What kinds of functions or relations are they, and what are their domains and codomains?

17. If a binary relation is symmetric, what can you say about its inverse relation?

18. Recall the Parent relation from § 2.13.

Suggest a good name for its transitive closure.

Then use that relation (i.e., the transitive closure) to define formally the relation that links two people that are related to each other (however distantly).

19. Rephrase the Collatz Conjecture as a statement about the transitive closure of the Collatz function.

20. Below we give some binary relations on the set of all Python programs. The Python programs are denoted by P and Q, and the binary relations are denoted by $\simeq_1, \simeq_2, \ldots, \simeq_{10}$. For each relation, determine whether or not it is an equivalence relation, and give reasons.

notation	definition of when the relation holds
$P \simeq_1 Q$	P and Q have the same number of characters
$P \simeq_2 Q$	P and Q compute the same function
$P \simeq_3 Q$	P and Q have the same set of variable names
$P \simeq_4 Q$	P and Q have at least one variable name in common
$P \simeq_5 Q$	P and Q have no variable names in common
$P \simeq_6 Q$	P and Q were written by the same set of programmers
$P \simeq_7 Q$	P and Q have at least one coauthor in common
$P \simeq_8 Q$	P was completed before Q
$P \simeq_9 Q$	P was completed on the same day as Q
$P \simeq_{10} Q$	P was completed after Q

21. What can you say about the transitive closure of an equivalence relation?

22. The anagram relation on the set of all English words is defined as follows. Let $x_1x_2\cdots x_m$ and $y_1y_2\cdots y_n$ be two English words of lengths m and n respectively, where the x_i and y_j are their letters ($1 \le i \le m$, $1 \le j \le n$). The ordered pair of words

$$(x_1x_2\cdots x_m,\ y_1y_2\cdots y_n)$$

belongs to the anagram relation if and only if there exists a bijection $f : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, n\}$ such that, for all $i \in \{1, 2, \ldots, m\}$, we have $x_i = y_{f(i)}$.

We assume all letters belong to the usual lower-case English alphabet $\{\mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}\}$. The exact choice of dictionary does not matter for this exercise.

(a) Find the largest set of three-letter words you can that are all related to each other by the anagram relation.

(b) For each $k = 1, 2, 3, 4$, find an English word of length k that is not related to any other word by anagram.

(c) Prove that anagram is an equivalence relation.

(d) $^{\omega}$ (Programming challenge) Using a standard open-access list of English words, determine the number of equivalence classes of anagram on the set of words in that list.

23. How many non-reflexive binary relations are there on a set of size n?

24. Recall that when you have a table with n columns, its rows represent the n-tuples in an n-ary relation. For $1 \leq i \leq n$, let A_i be a set containing all the entries in column i (and possibly more elements), so that the columns of the table define an n-ary relation on $A_1 \times A_2 \times \cdots \times A_n$. Assuming the columns of this table are all distinct, how many ternary relations can you construct by choosing columns from this table of n columns?

3

PROOFS

Why should a programmer learn to write mathematical proofs? The most obvious answer is that proofs give rigorous justification for properties of your programs or of the structures that they work with. A computer *scientist* is not just a computer *user* or *hobbyist* or *fan*. A computer scientist provides rational support for their claims. Sometimes, this can be done by computational experiments, where programs are run on a large number of different inputs that hopefully form a representative sample of the situations of interest, and the outputs are studied carefully and perhaps analysed statistically. But a proof gives a more fundamental kind of support for a claim. It is independent of the specific technology which is used to develop and run the program. It applies to a far wider range of scenarios than can ever actually be run in a set of computational experiments.

In this chapter, we learn about the nature of mathematical proofs and the art of writing them. We treat the main types of proof, with emphasis on proof by induction. We conclude in § 3.14 by reflecting further on the role of proofs in computer science. Proofs are not only a tool for proving statements about programs; proofs are, themselves, like programs in many ways, and writing them is like programming, and developing skill in proof-writing will make you a better programmer.

3.1$^{\alpha}$ THEOREMS AND PROOFS

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

You may also come across the terms "proposition", "lemma" and "corollary". 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.[1,2] 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

1 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, which you'll meet later in this unit, and Burnside's Lemma.

2 These days, the plural of "lemma" is "lemmas", as you would expect. But, traditionally, the plural was "lemmata", which you may encounter in old books or papers. Are there other English words where the plural ending is -ta?

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 next week onwards. A **corollary** is a theorem that follows almost immediately from another theorem that has just been stated.

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

A proof consists of a finite sequence of statements, culminating in the claim that is being proved. These statements are often called the **steps** of the proof. Each statement in the 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

- an assumption, as a start towards proving something that is true under that assumption;

  or

- a logical consequence of some of the previous statements. In other words, there must be some previous statements which, together, imply the current statement.

- The last statement in the sequence should establish that the claim follows from some previous statements in the proof. If the last statement is, by that stage, an obvious consequence of the statements before it, it is often omitted.

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 □ indicates the end of the proof.[3]

3 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 //.

We illustrate these concepts with the following theorem and proof, which you have seen before in Exercise 7. We number the steps of the proof, and give each its own line, to help with later discussion. But we won't normally do this in proofs.

Theorem 13. For any sets A and B, if $A \subseteq B$ then $\mathcal{P}(A) \subseteq \mathcal{P}(B)$.

Proof.
(1) Assume $A \subseteq B$.
(2) Let $X \in \mathcal{P}(A)$.
(3) Therefore $X \subseteq A$, by definition of $\mathcal{P}(A)$.
(4) Therefore $X \subseteq B$.
(5) Therefore $X \in \mathcal{P}(B)$.
(6) So we have shown that $X \in \mathcal{P}(A)$ implies $X \in \mathcal{P}(B)$.
(7) Therefore $\mathcal{P}(A) \subseteq \mathcal{P}(B)$.

□

Think about the role of each step in this proof.

- What type of proof step is it? (See our listing of types of proof steps above.)

- Does it make use of any *previous* steps? If not, why not? If so, how?

- Does it use any *subsequent* steps? If not, good! If so, we have a problem!

The following table repeats all the proof steps, slightly expanding some of them, and annotating each step to show how it relates to our discussion of proof steps above.

	proof step	comment
(1)	Assume $A \subseteq B$.	The theorem statement is about what happens under the assumption that $A \subseteq B$, so we start by making this **assumption**.
(2)	Let $X \in \mathcal{P}(A)$.	This is a **definition**. Its purpose is to give a name to a general member of $\mathcal{P}(A)$, so we can talk about it.
(3)	Therefore $X \subseteq A$, by definition of $\mathcal{P}(A)$.	$X \subseteq A$ is a **logical consequence** of (2) and the definition of power set.
(4)	Therefore $X \subseteq B$, by (1) and (3).	$X \subseteq B$ is a **logical consequence** of (3) $X \subseteq A$ and (1) $A \subseteq B$.
(5)	Therefore $X \in \mathcal{P}(B)$, by (4) and the definition of $\mathcal{P}(B)$.	This is a **logical consequence** of (4) $X \subseteq B$ and the definition of power set.
(6)	So $X \in \mathcal{P}(A)$ implies $X \in \mathcal{P}(B)$.	This really just summarises what we've done over steps (2)–(5).
(7)	Therefore $\mathcal{P}(A) \subseteq \mathcal{P}(B)$.	This is a **logical consequence** of (6) and the definition of the subset relation.

3.2 LOGICAL DEDUCTION

The backbone of any proof consists of its logical deductions. These are the steps where we deduce a **logical consequence** of previous steps. If a proof does not have logical deductions, then it's just a collection of known facts and assumptions, and we get nothing new. It's a kind of "logical jelly" without structure or substance.

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 implies 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).

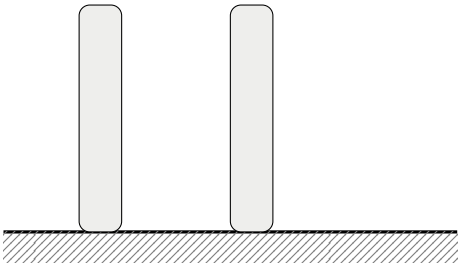


Figure 3.1: Two dominos which could fall to the right

For example, consider the deduction we made at step (3) of the proof of Theorem 13. At that stage, we know from earlier steps that

- $X \in \mathcal{P}(A)$, which we can treat as true simply because it's the definition of X (step (2));
- if $X \in \mathcal{P}(A)$ then $X \subseteq A$, which comes from the definition of $\mathcal{P}(A)$.[4]

If we let P stand for $X \in \mathcal{P}(A)$, and Q stand for $X \subseteq A$, then $P \Rightarrow Q$ represents the assertion that

$$\text{if } X \in \mathcal{P}(A) \text{ then } X \subseteq A.$$

So, step (2) is P, the definition of power set gives $P \Rightarrow Q$, and logical deduction (or *modus ponens*, if we want to practise our Latin) then gives Q.

The role of *implication* in logical deduction is crucial. An implication $P \Rightarrow Q$ is the link that translates the truth of P into the truth of Q. So let us consider it further.

Suppose we have two dominos standing on their ends, a short distance apart and with their faces parallel, as shown side-on in Figure 3.1.

Let P be the statement that the left domino falls to the right, and let Q be the statement that the right domino falls to the right. Each of these statements could be true or false. There is no requirement here for either domino to fall; it's fine for them both to remain standing. It's also fine for the left domino to remain standing but for the right one to fall (by whatever means). But if the left domino falls, then the right domino must fall too. It is impossible to have the left domino fall with the right domino remaining standing. So we have three possible situations, which we'll represent as ordered pairs:

$$\begin{aligned} &(\text{left stands}, \quad \text{right stands}) \\ &(\text{left stands}, \quad \text{right falls}) \\ &(\text{left falls}, \quad \text{right falls}) \end{aligned}$$

4 That definition actually gives "if and only if" here: $X \in \mathcal{P}(A) \Leftrightarrow X \subseteq A$. But we do not need the reverse implication right now.

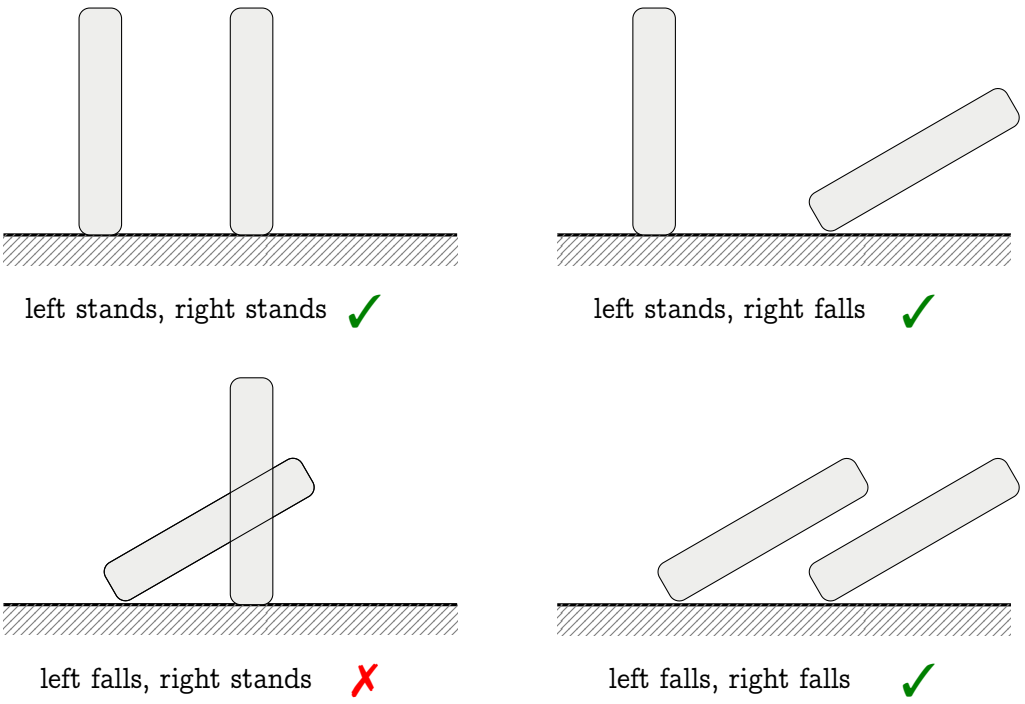


Figure 3.2: Two dominos, standing or falling: three possible situations, one impossible one.

We can depict these various possibilities using sets. Let P be the set of those situations where the left domino falls, and let Q be the set of those situations where the right domino falls. Then

$$\begin{aligned} P &= \{(\text{left falls, right falls})\}, \\ Q &= \{(\text{left falls, right falls}),\ (\text{left stands, right falls})\}. \end{aligned}$$

The various situations and the sets P and Q are shown in a Venn diagram in Figure 3.3.

Observe that the impossible situation

$$(\,\text{left falls, right stands}\,)$$

is not shown on the Venn diagram. If it were possible, then it would belong to $P \setminus Q$ and then P would not be a subset of Q. But its impossibility means that $P \setminus Q = \emptyset$ and $P \subseteq Q$.

This example illustrates the general principle that the logical implication $P \Rightarrow Q$ between the statements P and Q corresponds to the subset relation $P \subseteq Q$ between the sets of situations they represent.

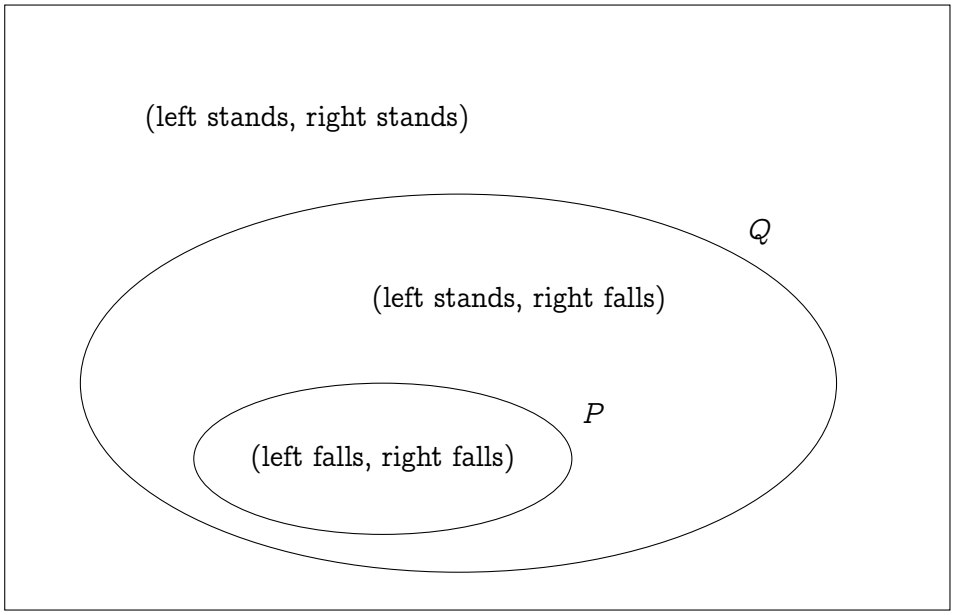


Figure 3.3: The sets P and Q.

We have seen this principle in action before, when the statements are framed as statements about set membership, on p. 7 in § 1.6. For any two sets A and B,

$$A \subseteq B \text{ if and only if for all } x \text{ we have } x \in A \Rightarrow x \in B.$$

Our domino example illustrates that the link between the subset relation and logical implication is more general, and applies even where the logical implication is not stated in terms of set membership.

An important special case of implication is when the starting condition (on the left) is false. If we have an implication $X \Rightarrow Y$ when X is false, then the implication $X \Rightarrow Y$ is true regardless of what Y is. This corresponds to the fact that, if X is the empty set and Y is *any* set, then $X \subseteq Y$, since $\emptyset \subseteq Y$; the empty set is a subset of every set.

Keep in mind that the truth of an implication does not mean that either of its two parts is true. In our domino example, we know that $P \Rightarrow Q$, but this does not mean that P falls or that Q falls. It just gives a logical relationship between these two events, namely that if P falls then Q falls.

Note also that this is a purely logical relationship. In the domino scenario, there is also the ingredient of time. This plays a role in the physical mechanism by which the falling of P (if it happens) causes the falling of Q, and it follows from that mechanism that the fall of Q happens later, in time, than the fall of P. But this is a detail of the actual physical setting. Logical implication itself is not an assertion about time, but merely an assertion about the truth or falsehood of the two parts, in this case P and Q.

Indeed, it is entirely possible that logical implication can "go backwards" in time. Suppose X is the statement that you can see stars in the sky and Y is the statement that the sun has set. The sun setting does not itself mean that you can see stars, since it might be too cloudy. But if you can see stars, then you know the sun has set. So $X \Rightarrow Y$ holds, even though X happened after Y, and there is no suggestion that X *causes* Y.

Always keep in mind that implication is *not symmetric*. In the domino example, we have $P \Rightarrow Q$, but we do not have $Q \Rightarrow P$, because Q falling does not have to be because P falls (as we have supposed throughout that Q could be made to fall, by some external force, even if P remains standing). In the sunset example just given, we have $X \Rightarrow Y$, but we do not have $Y \Rightarrow X$, because the sun setting does not imply that you can see stars (as it might be cloudy).

The **converse** of an implication is the reverse implication, i.e., the implication you get by swapping the order of the two parts *or* by reversing the direction of the implication arrow symbol.[5] So the converse of $P \Rightarrow Q$ is $Q \Rightarrow P$, which can also be written $P \Leftarrow Q$. When an implication holds, we cannot assume that the converse also holds. In the examples of the previous paragraph, we showed that, for the two example implications we have been discussing, the converse does not hold.

Sometimes, an implication and its converse *both* hold. For example, suppose you are holding a ball above the ground. Suppose R means that you release the ball and S means that the ball falls to the ground. Then $R \Rightarrow S$, and its converse $S \Rightarrow R$ holds too. In these situations, we can put the two implications together to make a two-way implication, written $R \Leftrightarrow S$, which means that R and S are logically equivalent, and we often say that R holds if and only if S holds. We have seen statements of this type before, in some of our Theorems.

3.3 PROOFS OF EXISTENTIAL AND UNIVERSAL STATEMENTS

We now give more examples of theorems and proofs, highlighting the relationship between the kind of statement you are trying to prove and the way you need to prove it.

Theorem 14. English has a palindrome.

Proof. 'rotator' is an English word and also a palindrome. □

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. This one happens to be true, and we have given a proof of it, so it is a theorem.

Proving an existential statement, such as ...

There exists a palindrome in English

5 But don't do *both* of those, or you'll get the implication you started with, just written differently! $P \Rightarrow Q$ and $Q \Leftarrow P$ are just different ways of writing the same thing.

...just requires one suitable example.

Most proofs are not this short ...

Theorem 15. Every English word has a vowel or a 'y'.

Proof. This can be shown by listing all English words:
'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.

3.4 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.

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 symbolic manipulation using algebraic rules can transform expression A to expression B, then that's good;
but if not:

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

3.5 TYPES OF PROOFS

We now consider five types of proof, namely

- Proof by symbolic manipulation
- 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.

3.6 PROOF BY SYMBOLIC MANIPULATION

Some proofs proceed by a sequence of equations, where each equation uses some basic law (an axiom or some fundamental theorem) about the objects in question.

Many proofs you did in secondary school mathematics would have been of this type. For example, consider the difference of two squares:

$$x^2 - y^2 = (x+y)(x-y). \tag{3.1}$$

This is typically proved along the following lines.

$$\begin{aligned}
(x-y)(x+y) &= x^2 + xy - yx - y^2 && \text{(expanding, using the distributive law for numbers)} \\
&= x^2 + xy - xy - y^2 && \text{(since multiplication is commutative)} \\
&= x^2 - y^2 && \text{(by additive cancellation).}
\end{aligned}$$

Incidentally, this proof also illustrates that, if you want to prove an equation, you can start with *either side* of the equation and work towards the other side. You don't have to start with the left side, just because you read it first! In the above proof, we started with the right side. Any proof of this type can be done in either direction, but sometimes one direction seems more intuitive than the other.

Similarly, the basic laws of sets can be used to prove equality between some expressions involving sets. We have already seen some proofs of this type, in Theorem 1, Corollary 2, and Theorem 5,

We can also use symbolic manipulation in parts of other proofs, as we did for example in the proof of Theorem 10, where we used basic properties of function composition and inverse functions to do chains of equalities that establish the desired claims about inverses of compositions of functions.

3.7 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 Theorem 14.

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.

3.8 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 Theorem 15. 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.

3.9 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 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 16. Every natural number is interesting.[6]

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. □

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 an ordered 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?

We now give a proof by contradiction of a fundamental theorem about prime numbers.

Theorem 17 (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. □

3.10 PROOF BY MATHEMATICAL INDUCTION

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

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

One powerful technique for proving theorems of this type is the **Mathematical Induction**. It is widely used across computer science and is particularly useful in proving theorems about the behaviour of algorithms and the discrete structures they work with including those considered in this unit. You will also use it extensively in later Computer Science units: FIT2004, FIT2014 and MTH3170/3175.

The **Principle of Mathematical Induction** is:

IF	$S(1)$ is true	*(inductive basis)*
AND	for all k: **if** $\underbrace{S(k) \text{ is true}}_{\textit{inductive hypothesis}}$ **then** $S(k+1)$ is true	*(inductive step)*
THEN	for all n: $S(n)$ 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;
- $\vdots$
  $\vdots$
- 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".

Our first use of Mathematical Induction is to prove an extension of De Morgan's Law for Sets, Theorem 1, to arbitrary numbers of sets. It is illustrated for three sets in Figure 3.4.

Theorem 18. For all $n \in \mathbb{N}$,

$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}. \tag{3.2}$$

Proof. Let $S(n)$ be the statement that (3.2) holds for n. We must prove that, for all $n \in \mathbb{N}$, the statement $S(n)$ is true.

We prove it by induction on the number n of sets.

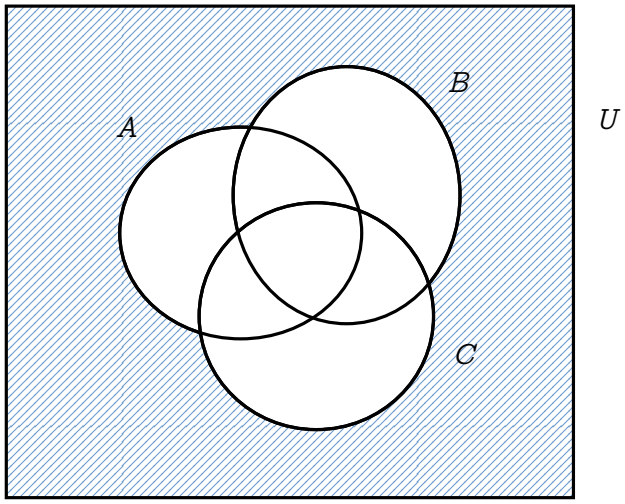


Figure 3.4: $\overline{A_1 \cup A_2 \cup A_3} = \overline{A_1} \cap \overline{A_2} \cap \overline{A_3}$, shaded.

Inductive basis:

$S(1)$ is trivially true. In that case, each side of (3.2) is just $\overline{A_1}$, so the equation holds.

Inductive step:

Let $k \geq 1$.

Suppose $S(k)$ is true:

$$\overline{A_1 \cup A_2 \cup \cdots \cup A_k} \;=\; \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_k}.$$

This our Inductive Hypothesis. We will use it later.

We have:

$$\begin{aligned}
&\overline{A_1 \cup A_2 \cup \cdots \cup A_{k+1}} \\
&= \overline{(A_1 \cup A_2 \cup \cdots \cup A_k) \cup A_{k+1}} \quad \text{(just grouping ...)} && (3.3)\\
&= \overline{A_1 \cup A_2 \cup \cdots \cup A_k} \cap \overline{A_{k+1}} \quad \text{(by De Morgan's Law for two sets, Theorem 1)} && (3.4)\\
&= \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_k} \cap \overline{A_{k+1}} \quad \text{(by Inductive Hypothesis)}
\end{aligned}$$

Therefore $S(k+1)$ is true.

Conclusion:

So, by the Principle of Mathematical Induction, it's true for any number of sets. □

Because induction helps prove theorems about statements that hold for all positive integers, it is a natural tool for proving statements about infinite sequences. In the next section (§ 3.11), we will use it to prove some statements about infinite sequences of

numbers. Because of this, it is a useful tool for proving statements about the behaviour of loops in programs; we will see this in the next section too.

A key thought process in the inductive step is to construct, from the "$(k+1)$-object", a smaller object to which you can apply the inductive hypothesis. In the previous proof, our "$(k+1)$-object" is

$$\overline{A_1 \cup A_2 \cup \cdots \cup A_{k+1}}.$$

Our aim is to show that this satisfies (3.2) (with $n = k+1$). In order to do this, we try to construct, from it, a "k-object", in this case

$$\overline{A_1 \cup A_2 \cup \cdots \cup A_k}.$$

We first group the first k sets, in (3.3), as a step in this direction. Then applying De Morgan's Law for two sets, in (3.4), gives us what we are aiming for: $\overline{A_1 \cup A_2 \cup \cdots \cup A_k}$, our "$k$-object", shown in green in (3.4). Once we have the "k-object", we can apply the Inductive Hypothesis to it.

This technique, of reducing an object to a simpler object (or objects) of the same type, is known as **recursion**. It is one of the most fundamental problem solving strategies in computer science; you will encounter it and use it again and again and again. It is also provided for in most programming languages, when functions or methods can call themselves. By using this skill in proofs by induction, you are practising a core skill of your discipline. Moreover, proof by induction is the go-to proof technique for proving claims about the behaviour of recursive functions in programs.

3.11 INDUCTION: MORE EXAMPLES

For another use of induction, consider the following equation, which gives a formula for obtaining the sum of the first n positive integers, so that you can compute this sum without having to add up all those numbers.

$$1+2+3+\cdots+(n-1)+n \;=\; \frac{n(n+1)}{2}. \tag{3.5}$$

This is very useful in computer science and in fact throughout science, engineering and in many other fields. We illustrate this with an application.

Suppose you are trying to construct the Monash timetabling network described in § 2.13. You have access to a function which tells you, for any pair of units, how many students are doing both those units. You need to search all pairs of units, check how many students are doing both units in a pair, and if that number is nonzero, add a link between those units, to record the fact that their classes must be at different times.

A natural approach is to use two nested loops. The outer loop iterates over all units in some order (e.g., lexicographic order). For each unit considered in the outer loop, the inner loop iterates over all units that come later in the order. This saves duplication: if

the inner loop also iterated over *all* units, then each pair of units would be considered twice, once for each of the two possible ordered pairs based on those units.

If there are N units altogether, then these nested loops have the structure

for each $i = 1, 2, \ldots, N$:
    for each $j = i+1, i+2, \ldots, N$:
        if at least one student is doing both the i-th unit and the j-th unit,
        then make a link between these two units.

There are N iterations of the outer loop, but these have varying numbers of inner loop iterations. If we want to work out how long this computation takes, we need to know how many inner loop iterations are done altogether.

- For the first iteration of the outer loop, $i = 1$, and the inner loop starts at $i+1 = 1+1 = 2$ and considers $j = 2, 3, \ldots, N$, so it does $N-1$ iterations.
- For the second iteration of the outer loop, $i = 2$, and the inner loop considers $j = 3, 4, \ldots, N$, so it does $N-2$ iterations.
- For the third iteration of the outer loop, the inner loop does $N-3$ iterations.
- ...and so on ...
- For the $(N-1)$-th iteration of the outer loop, we have $i = N-1$, so there is only one iteration of the inner loop, with $j = i+1 = (N-1)+1 = N$.
- The N-th iteraton of the outer loop actually has no inner loop iterations at all, because the range $i+1 \le j \le N$ is empty, because $i+1 > N$.

So the total number of inner loop iterations is

$$(N-1) + (N-2) + \cdots + 2 + 1.$$

This expression is of the same type as the left-hand side of (3.5). We've written it in a different order, but that doesn't matter at all. The only other difference is that we're adding up the first $N-1$ positive integers instead of the first n, but we can still use the right-hand side of (3.5) to give us the answer, with an appropriate substitution.

Nested loop structures along these lines are very common in programming, and equations like (3.5) enable us to give good estimates of how long they will take even before we run the program.

So, how do we *prove* (3.5)? There are many beautiful proofs of this fact, and we will see at least two proofs in this unit. The first we give now, using induction.

Theorem 19. 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 $n=k+1$.

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

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. □

Notice, in the inductive step, how we construct, from the "$(k+1)$-object" $1+2+\cdots+(k+1)$, a "k-object" $1+2+\cdots+k$. In this case, it's just a matter of grouping, since the k-object just sits within the $(k+1)$-object. As soon as we construct the k-object, we can apply the Inductive Hypothesis to it.

Often, as in our proof of Theorem 18, we need to do some more work to construct the k-object from the larger $(k+1)$-object.

There is a slightly different way of writing inductive proofs that you are likely to come across. We could make the inductive step go from $n=k-1$ to $n=k$, instead of from $n=k$ to $n=k+1$. Let us re-do the previous proof in this way.

Slightly different proof: We prove Theorem 19 by induction on n.

Inductive basis:

When $n = 1$, LHS $= 1$ and RHS $= 1(1+1)/2 = 1$.

Inductive step:

Let $k \geq 2$. *[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 $n = k$.

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

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.

□

We will mostly frame our inductive steps as going from $n = k$ (where we *assume* the statement is true) to $n = k+1$ (where we *deduce* it's true, using the Inductive Hypothesis), as we did in our first proof of Theorem 19. But there is nothing wrong with doing it from $n = k-1$ to $n = k$, as in the second proof above. If you read other books and resources, you will find that some authors do it one way, while others do it the other way. If you do it the second way (from $n = k-1$ to $n = k$), then you need to take care that the inductive step is grounded in the inductive basis. In the first proof, the inductive step starts with "Let $k \geq 1$"; in the second proof, the inductive step starts with "Let $k \geq 2$" which ensures that $k - 1 \geq 1$ so that the inductive step is, indeed, grounded in the inductive basis.

Even though we have now proved Theorem 19, you are entitled to wonder where the expression, $n(n+1)/2$, came from in the first place. There are many ways to derive this expression from scratch. We will discuss this in more detail later, in **??**.

When you first try to work out what the expression should be, you might *explore* the first few cases and try to discern a pattern. This may lead you to *conjecture* that the expression is $n(n+1)/2$.

A pattern that works for small cases is not, in itself, a proof. But it might still help you come up with a proof, since now you have something to aim for: a conjecture that you can try to prove. And, in this case, mathematical induction can be used to *prove* the conjecture.

We will discuss this explore-conjecture-prove methodology, for discovering and proving formulae for mathematical expressions, in ??.

Mathematical induction is a proof technique, and as such can only be used once you have a statement — in this case, an equation — to prove. It won't help you *discover* what the right equation should be; that's where exploration and conjecture come in. The upside is that you get a rigorous proof for a conjecture which you might otherwise have remained unsure about or been unable to justify fully.

Having used induction to prove that the sum of the first n positive integers is indeed $n(n+1)/2$, it is natural to ask, what about sums of higher powers of positive integers? To start with, what about the sum of the squares of the first n positive integers, i.e., $1^2+2^2+\cdots+n^2$? This gives the number of iterations in the following triply-nested loops:

for each $i = 1, 2, \ldots, n$:
    for each $j = i, i+1, \ldots, n$:
        for each $k = i, i+1, \ldots, n$:
            [some action]

If you'd like a challenge, *explore* $1^2+2^2+\cdots+n^2$ by computing it for several small values of n and trying to understand its behaviour. How does it grow as n increases? Then try to *conjecture* a possible formula for it, and then try to *prove* it by induction.

3.12$^{\omega}$ INDUCTION: EXTENDED EXAMPLE

We now give another detailed example of proof by induction. This is quite involved but it does give extended practice at reasoning about functions and relations as well as induction.

Consider the following problem. Suppose there are m job vacancies to be filled from a pool of n applicants. For each vacant job, the employer has constructed a shortlist of applicants. Is it possible for each position to be filled by a different applicant, so that no position remains unfilled?

If $m > n$, then this is impossible: there are simply too many positions for the available applicants, and some number of positions must remain unfilled. If $m \leq n$, then it may or may not be possible, depending on the shortlists.[7]

We model this problem as follows. Let A be the set of positions, with $|A| = m$, and let B be the set of applicants, with $|B| = n$. Let $S \subseteq A \times B$ be the *shortlist relation*, a binary relation consisting of all pairs (a, b) such that the shortlist for position a includes

7 If $m < n$, then some applicants will not get a position, but that is permitted in this problem. One could also imagine a scenario where it is the applicants who have shortlists and the employers who have no choice.

applicant b. The full shortlist for position a is the set $S(a)$, which is a subset of B; here, we use the notation introduced on p. 67 in § 2.13. If we have a set of positions $X \subseteq A$, then we can, if we wish, form a combined shortlist for the set of positions by taking the union of all the individual shortlists:

$$S(X) = \{b \in B : aSb\} = \bigcup_{a \in X} S(a).$$

We want each position to be filled by exactly one applicant, so the set of pairs (a,b) that specify allocation of applicants to positions must be a *function*. Furthermore, we want each position to be filled by an applicant who does not also fill any other position; no applicant fills two positions. So we want, for each $a \in A$, a unique $b \in B$ such that $(a,b) \in S$. These pairs (a,b) that specify a valid allocation of applicants to positions, filling each position with a unique applicant, must therefore be an *injection*. So we are asking: does the shortlist relation S contain an injection?

Some situations can be dealt with easily.

If $m > n$, then B is too small to enable an injection to it from A, so S contains no injection. More generally, suppose there is a set $X \subseteq A$ of jobs for which the union of their shortlists $S(X)$ is smaller than X, i.e., $|X| > |S(X)|$. Then S cannot contain an injection, since an injection would ensure that the shortlists for X together include at least $|X|$ applicants. We have now proved:

Theorem 20. If a shortlist relation from A to B contains an injection with domain A then, for all $X \subseteq A$, we have $|X| \leq |S(X)|$.

(The condition here means that, for every set X of jobs, the union of their shortlists contains at least as many applicants as there are jobs in X.) □

More surprisingly, the converse is true as well. We will prove this by induction.

Theorem 21. Let $S \subseteq A \times B$ be a binary relation. If, for all $X \subseteq A$ we have $|X| \leq |S(X)|$, then S contains an injection with domain A.

Proof. We prove this by induction on $|A|$.

Inductive basis:

When $|A| = 1$, the set A contains a single element a. Using $X = \{a\}$, if the shortlist for this one job has at least one applicant, then there exists $b \in B$ such that $(a,b) \in S$. So S contains the simple function whose sole ordered pair is (a,b). This function is an injection with domain A. So the claim holds for $|A| = 1$.

Inductive step:

Let $m \geq 1$.

Assume that the following holds for every binary relation $T \subseteq C \times D$ in which $|C| \leq m$:

> If, for each $X \subseteq C$ we have $|X| \leq |T(X)|$, then T contains an injection with domain C.

This is our Inductive Hypothesis. Note that this is an *implication*. Like any implication, it has a *condition* (namely, "for each $X \subseteq A$ we have $|X| \leq |T(X)|$") and a *consequence* (namely, "T contains an injection with domain C"). In assuming this Inductive Hypothesis, we are not assuming that the condition is true, and we are not assuming that the consequence is true. We are merely assuming that, *if* the condition is true, *then* the consequence is true.

Let $S \subseteq A \times B$ where $|A| = m+1$. Suppose that the following condition holds:

$$\text{for each } X \subseteq A \text{ we have } |X| \leq |S(X)| \tag{3.6}$$

We need to prove that S contains an injection with domain A.

Case 1: for each nonempty $X \subset A$ we have $|X| \leq |S(X)| - 1$. (Note here that X is a proper subset of A.)

Let (a,b) be any pair in S. Then consider the restriction of S to $(A \setminus \{a\}) \times (B \setminus \{b\})$, i.e., the subrelation consisting of all pairs of S that do not meet either a OR b. Call this smaller relation T, and put $C := A \setminus \{a\}$ and $D := B \setminus \{b\}$. Let $X \subseteq C$. Then $|S(X)| \geq |X|+1$, by the assumption on which this case is based. But, in T, the applicant b is excluded. Nevertheless, we can still say that, in T, the union $T(X)$ of X's shortlists has at least $|X|$ applicants (obtained from the $k+1$ applicants by excluding b if necessary): $|T(X)| \geq |X|$. Now, T has one fewer position than S, since $|C| = |A|-1$. So the Inductive Hypothesis applies to it. Therefore T contains an injection with domain C. Call that injection g. This injection, together with the pair (a,b), gives a new binary relation $f := g \cup \{(a,b)\}$ which is a subset of S:

$$g \cup \{(a,b)\} \subseteq S,$$

since $g \subseteq T$ and $T \subseteq S$ and $(a,b) \in S$. Furthermore, this new relation f is actually an injection with domain A, since

- g is an injection, *and*
- a appears in no other pair of g (so there is no "one-to-many" violation of the definition of a function), *and*
- b appears in no other pair of g (so there is no "many-to-one" violation of the surjection property), *and*
- the domain of f is A (since it is just the domain of g, namely $A \setminus \{a\}$, augmented by a which is mapped to b by f).

So S does indeed contain an injection with domain A in this case.

Case 2: for some nonempty $X \subset A$ we have $|X| = |S(X)|$.

Let $X \subset A$ be such a set of positions.

Consider the restriction of S to pairs (a,b) such that $a \in X$. Call this binary relation T. It is a subset of $X \times S(X)$. Since X is a proper subset of A, we have $|X| < |A|$, so

$|X| \leq m$. So we can apply the Inductive Hypothesis, with $C = X$ and $|C| \leq m$. Now, the Inductive Hypothesis is an implication, with condition and consequence as discussed above. We first establish that the condition holds in our current situation. Recall that every $Y \subseteq A$ satisfies $|Y| \geq |S(Y)|$, by (3.6). So, certainly every $Y \subseteq C$ satisfies $|Y| \geq |S(Y)|$ (since $C \subseteq A$). Also, T is just the restriction of S to $C \times B$, so $T(Y) = S(Y)$ (because $Y \subseteq C$). Therefore every $Y \subseteq C$ satisfies $|Y| \geq |T(Y)|$. Therefore T satisfies the condition of the Inductive Hypothesis. Therefore, by the Inductive Hypothesis, its consequence also holds, namely that T contains an injection with domain X, which we call g. It has domain $X = C$, and one suitable codomain is $T(C)$ which is the same as $S(C)$. So we can write $g : C \to S(C)$.

This injection g is a step towards constructing an injection in S. Its domain is X, so it fills the jobs in that set. But X is a proper subset of A, so there are other jobs what g does not fill.

So we need to find a way to fill jobs in $A \setminus X$. Again, we work towards applying the Inductive Hypothesis. Observe that $A \setminus X$ is a nonempty proper subset of A, since X is too.

Let $Z \subseteq A \setminus X$. We have

$$\begin{aligned}
|X| + |Z| &= |Z \cup X| && \text{(since the union is disjoint)} \\
&\leq |S(Z \cup X)| && \text{(by (3.6))} \\
&= |S(X) \cup (S(Z) \setminus S(X))| && \text{(since } S(Z \cup X) = S(X) \cup (S(Z) \setminus S(X))\text{)} \\
&= |S(X)| + |S(Z) \setminus S(X)| && \text{(since the union is disjoint)} \\
&= |X| + |S(Z) \setminus S(X)| && \text{(since } |S(X)| = |X|\text{, by the assumption on which this case is based).}
\end{aligned}$$

Subtracting X from the first and last expressions, we have

$$|Z| \leq |S(Z) \setminus S(X)|. \tag{3.7}$$

Let U be the binary relation obtained from X by restricting to those pairs (a, b) such that $a \notin X$ and $b \notin S(X)$. In other words, U is the restriction of S to $(A \setminus X) \times (B \setminus S(X))$. If $Z \subseteq A \setminus X$, then $U(Z)$ consists of those pairs (a, b) such that $a \in Z$, $b \in S(Z)$ and $b \notin S(X)$. So

$$U(Z) = S(Z) \setminus S(X).$$

This, together with (3.7), gives

$$|Z| \leq |U(Z)|.$$

We are now in a position to apply the Inductive Hypothesis. It can be used on U, because $|A \setminus X| \leq m$ (which follows from the fact that $A \setminus X$ is a proper subset of A). Its condition also holds, because, as we have just seen, $|Z| \leq |U(Z)|$ for all $Z \subseteq A \setminus X$. Therefore the consequence follows, namely that U contains an injection with domain $A \setminus X$. Call this injection h. Its domain is $A \setminus X$ and its codomain can be taken to be $B \setminus X$. So the domains of g and h are disjoint, and their codomains are disjoint too.

Since g and h are functions on disjoint domains and the union of their domains is A, the union $g \cup h$ is also a function and its domain is A. Also, since g and h are both injections and their codomains are disjoint, their union $g \cup h$ is also an injection.

We have constructed an injection $g \cup h$ with domain A. Since $g \subseteq T$ and $T \subseteq S$, and since also $h \subseteq U$ and $U \subseteq S$, we have $g \cup h \subseteq S$. So S contains an injection with domain A.

This completes Case 2. Since Cases 1 and 2 cover all possibilities, the Inductive Step is now complete.

Conclusion:

Therefore, by the Principle of Mathematical Induction, the theorem holds. □

Putting Theorem 20 and Theorem 21 together we have

Corollary 22. Let $S \subseteq A \times B$ be a binary relation. The relation S contains an injection if and only if for all $X \subseteq A$ we have $|X| \leq |S(X)|$. □

This gives us an important and useful characterisation of situations when there is an injection. It does not give us an algorithm, though it does contain ideas that are useful in the development of algorithms for this and related problems. What it does give us is succinct, easily-verified evidence for both positive and negative cases. We explain this now, for each case in turn.

- Positive case: if S does contain an injection, then the evidence of this is the injection itself. Once we have an injection f, we can easily check that it is an injection. First, check that every job $a \in A$ belongs to some pair $(a,b) \in f$, so that the domain of f is indeed A. Then check, for each $a \in A$, that it belongs to no more than one pair in f, so that f is indeed a function. Then check, for each $b \in B$, that it belongs to no more than one pair in f, so that f is indeed an injection.
- Negative case: if S does not contain an injection, then the evidence of this is some set $X \subseteq A$ such that $|X| > |S(X)|$. Once we have such a subset X, we can easily check that $|X|$ and $|S(X)|$ satisfy this inequality, as follows. For each $a \in X$ in turn, we find all pairs $(a,b) \in S$, and for each such pair, we add b to our set $S(X)$ if it is not already there. Once we have finished compiling $S(X)$, we check that its size is $< |X|$.

The details of these checking methods would depend on the specific data structure used to store the binary relation S. It would be a good programming exercise to write programs to do each of these two checking methods and to study how long they take to run, as a function of the sizes of A and B and the number of ordered pairs in S.

3.13 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.

3.14 PROGRAMS AND PROOFS

At the start of this chapter, we discussed the role of proofs in computer science, in particular their importance in providing rigorous, rational support for general claims about programs and the structures they work with.

The different types of proofs we have introduced can each be used to prove statements about particular programming language constructs. If we want to prove something about an if-statement, then we would normally use proof by cases. If we want to prove something about a function that calls itself recursively, then we would normally use proof by induction. Proving statements about loops in programs is also often done by induction.

But there is a deeper reason for programmers to develop skills in writing proofs.

Programming is often thought of as a completely different activity to writing mathematical proofs. In fact, the two activities are surprisingly similar, and developing the skill of writing mathematical proofs will make you a better programmer.

Let's compare *programs* and *proofs*.

Each consists of a sequence of precise statements. Each of these statements obeys the rules of some language: for programs, this is the programming language; for proofs, this is the language of mathematics augmented by precise use of a natural language such as English.

Each uses variables, which are names that can refer to any object of some type. Variables can be combined into expressions using the operations that are available for objects of that type. One difference between programs and proofs is that, in programs, variables have memory set aside for them, but this does not happen in proofs.

Each statement must depend on previous statements in a precise way. In a program, when an operation is applied to a variable, it uses the most recently computed value of that variable; it does not "look ahead" and use some other value of the variable that will be computed later. In a proof, each statement is a logical consequence of *previous* statements (not of later statements), or it might be just an already-known fact or axiom.

In a program, we can use an if-statement to decide, according to some logical condition, which set of statements to execute next. In a proof, we can use a finite number of cases, which together cover all possible situations (§ 3.8). Each case pertains to situations that satisfy some logical condition.

Programs can call other programs, which may have been written by you or by other people. Proofs can use other theorems, which may have been written by you or by other

people. So building a proof, using theorems that have been proved previously, is like writing a function in your program that calls other functions.

In a program, a block of statements can be executed repeatedly, for as long as some logical condition is satisfied. Another way to do repetition is to use recursion, in which a function or method calls itself. The analogue in proofs is mathematical induction.

A program crashes if it encounters a situation that its statements cannot deal with. We say it has a bug, and this should be fixed. Ideally, the program never crashes. If a proof cannot deal with some situation it is supposed to cover, then we might also call this a bug, or a "hole" or a "gap". This is more serious for proofs than for programs. A program with a bug might still be of some use, provided it crashes rarely and can correctly deal with the inputs that don't cause it to crash (even though it *should* be fixed). But a "proof" with a hole is actually *not* a proof at all. It might be able to be repaired, to turn it into a proof, or it might not; in the latter case, the "theorem" being "proved" might actually be false, in which case it has no proof, and is therefore not a theorem after all.

Programming and proof-writing have differences as well as similarities, but the analogy is close. Writing mathematical proofs is like programming in a different "paradigm". A *programming paradigm* is, loosely speaking, an approach to programming and a way of thinking about it. Important programming paradigms include procedural programming (historically the first), object-oriented programming, logic programming and functional programming. Learning to program in more than one *paradigm* (as well as in more than one language) improves your thinking about programming and makes you a better programmer in any language. Learning to write proofs — in the "programming paradigm" of mathematical reasoning — yields similar benefits to a programmer.

3.15 EXERCISES

1. Recall the Collatz function from p. 55 in § 2.9.

Prove by cases that, for all positive integers n,

$$\text{Collatz}^{(2)}(n) \leq \tfrac{3}{2}n + 1.$$

2.

A set of strings L is called **hereditary** if it has the following property:

> For every nonempty string x in L, there is a character in x which can be deleted from x to give another string in L.

Prove by contradiction that every nonempty hereditary set of strings contains the empty string.

3. Review the *statements* of all the theorems (including corollaries) in the previous chapters.

(a) Which ones are implications?

(b) Which ones are equivalences?

(c) Which ones are existential statements?

(d) Which ones are universal statements?

4. Review the *proofs* of all the theorems in the previous chapters.

(a) Which ones use proof by construction?

(b) Which ones prove an implication by proving its contrapositive?

(c) Which ones use proof by cases?

5. Let E be a finite set and let $\mathcal{I}$ be a collection of subsets of E that satisfies the following three properties:

(I1) $\emptyset \in \mathcal{I}$.

(I2) Whenever $X \subseteq Y$ and $Y \in \mathcal{I}$, we also have $X \in \mathcal{I}$. This is called the **hereditary property**.

(I3) Whenever $X, Y \in \mathcal{I}$ and $|X| < |Y|$, there exists $y \in Y \setminus X$ such that $X \cup \{y\} \in \mathcal{I}$. In other words, given two different-sized sets in $\mathcal{I}$, you can always enlarge the smaller set, using some new member from the larger set, to get another set in $\mathcal{I}$. This is called the **independence augmentation property**.

The sets in $\mathcal{I}$ are called **independent**.

For example, if $E = \{a, b, c\}$, then each of the following collections satisfies all three properties (I1)–(I3):

- $\mathcal{I} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
- $\mathcal{I} = \{\emptyset, \{a\}, \{b\}, \{c\}\}$
- $\mathcal{I} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, c\}, \{b, c\}\}$
- $\mathcal{I} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}\}$
- $\mathcal{I} = \{\emptyset\}$

By way of counterexample, using the same E, each of the following collections *fails* to satisfy all three properties (I1)–(I3):

- $\mathcal{I} = \{\{a\},\{b\},\{a,b\}\}$. This fails (I1) and (I2).
- $\mathcal{I} = \{\emptyset,\{a\},\{a,b\}\}$. This fails (I2).
- $\mathcal{I} = \{\emptyset,\{a\},\{b\},\{c\},\{a,b\}\}$. This satisfies (I1) and (I2) but fails (I3).
- $\mathcal{I} = \emptyset$. This fails (I1)

(a) For each counterexample above that fails (I2) and/or (I3), give specific sets X and Y for which the property fails.

(b) Prove that, for a social network (see § 1.7, § 2.14, § 2.13), the set of cliques satisfies (I1) and (I2) but not, in general, (I3).

(c) Prove by contradiction that all maximal independent sets have the same size.

Suppose that each member $e \in E$ has a nonnegative real **weight** $w(e)$. So $w : E \to \mathbb{R}_0^+$. We use this weight function to also give weights to *subsets* of E, by defining the weight of a set to be the sum of the weights of its elements: for all $X \subseteq E$, define

$$w(X) := \sum_{e \in X} w(e).$$

Suppose we seek an independent set of maximum weight.

Consider the following **greedy algorithm**, so called because it always makes the choice that gives biggest immediate gain:

Input: a finite set E, a function $w : E \to \mathbb{R}_0^+$, a collection $\mathcal{I}$ of subsets of E.
Initialisation:
    $X := \emptyset$
    $i := 1$.
If there exists $y \in E \setminus X$ such that $X \cup \{y\} \in \mathcal{I}$:
    From all these y, choose the one with largest weight $w(y)$. (This is where greed comes in.) Cal
    Add e_i to X.
else [we cannot enlarge X further]
    Stop and output X.

Greedy algorithms are important because they are simple, easy to program, and efficient. In general, they do not give best-possible solutions. But it is important to recognise situations when they do give best-possible solutions, since then we attain a rare state of algorithmic bliss: simplicity, speed, optimality.

(d) Construct a simple social network, and give a nonnegative real weight to each person, such that the greedy algorithm, with $\mathcal{I}$ being the set of all cliques, does not find

the maximum weight clique.

Now suppose $\mathcal{I}$ satisfies (I1)–(I3). The greedy algorithm chooses elements of E in a specific order, which we represent as

$$e_1, e_2, \ldots, e_r.$$

(e) Prove by contradiction that the weights of the chosen elements are decreasing, i.e.,

$$w(e_1) \geq w(e_2) \geq \cdots \geq w(e_r).$$

By "decreasing" we allow the possibility that some weights stay the same as we go along the list; we don't require that they be *strictly* decreasing (which is a stronger property and would entail replacing each $\geq$ by $>$).

(f) Prove that the greedy algorithm finds a maximum weight independent set.

So, whenever (I1)–(I3) hold, we can confidently use the greedy algorithm, knowing that it will give us the best possible solution.

Less obviously, it turns out that properties (I1)–(I3) give a precise characterisation of situations where greedy algorithms give optimum solutions! To be specific, if $\mathcal{I}$ is a collection of sets satisfying (I1) and (I2), and the greedy algorithm *always* gives the maximum weight member of $\mathcal{I}$ (regardless of the weights of the elements of E), then $\mathcal{I}$ must satisfy (I3) too. It is an interesting but challenging exercise to try to prove this.

So, it is useful to be able to recognise structures where properties (I1)–(I3) hold, because they are precisely the situations where you can't do better than a greedy algorithm. These structures are known as **matroids** and their role in optimisation goes beyond their connection with greedy algorithms. They also have many applications in the study of networks, vectors, matrices, and geometry.

6.

Prove the following statement, by mathematical induction:

For all k,

(*) the sum of the first k odd numbers equals k^2.

(a) First, give a simple expression for the k-th odd number.

(b) Inductive basis: now prove the statement $(*)$ for $k = 1$.

Inductive Step:
Let $k \geq 1$.

Assume the statement $(*)$ true for k. This is our **Inductive Hypothesis**.

(c) Express the sum of the first $k+1$ odd numbers ...

$$1+3+\cdots+((k+1)\text{-th odd number})$$

...in terms of the sum of the first k odd numbers, plus something else.

(d) Use the inductive hypothesis to replace the sum of the first k odd numbers by something else.

(e) Now simplify your expression. What do you notice?

(f) When drawing your final conclusion, don't forget to briefly state that you are using the Principle of Mathematical Induction!

7. Prove by induction on n that, for all $n \in \mathbb{N}_0$,

$$1+2+2^2+2^3+\cdots+2^n \;=\; 2^{n+1}-1.$$

8. Prove by induction that, for all n:

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

9. Prove by induction that, for all n:

$$1^3+2^3+\cdots+n^3 = (1+2+\cdots+n)^2.$$

10.
Consider the following algorithm:

```
for each i = 1, 2, ..., n:
        for each j = 1, 2, ..., i:
                for each k = 1, 2, ..., j:
                        [some action]
```

(a) For each pair i, j, how many times is the action inside the innermost loop performed? Express this in terms of i or j or both of them.

(b) For each i, how many times is the action inside the innermost loop performed? Write this as both a sum of some number of terms, and (using a theorem in this chapter) a simple algebraic expression. Both these expression should be in terms of i.

(c) Now write a sum of some number of terms, expressed in terms of n, for the total number of times the action is performed.

(d) Now work out this expression for $n = 1, 2, 3, 4, 5$, and for as many further values of n as you like.

(e) *Explore* how these numbers behave.

(f) *Conjecture* a simple algebraic expression, in terms of n, for the total number of times the action is performed.

(g) *Prove*, by induction on n, that your expression is correct.

11.

The n-th *harmonic number* H_n is defined by

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

These numbers have many applications in computer science. We will meet at least one later in this unit.

In this exercise, we prove by induction that $H_n \geq \log_e(n+1)$. (It follows from this that the harmonic numbers increase without bound, even though the differences between them get vanishingly small so that H_n grows more and more slowly as n increases.)

(i) Inductive basis: prove that $H_1 \geq \log_e(1+1)$.

(ii) Let $n \geq 1$. *Assume* that $H_n \geq \log_e(n+1)$; this is our *inductive hypothesis*. Now, consider H_{n+1}. Write it recursively, using H_n. Then *use the inductive hypothesis* to obtain $H_{n+1} \geq \ldots\ldots$ (where you fill in the gap). Then use the fact that $\log_e(1+x) \leq x$, and an elementary property of logarithms, to show that $H_{n+1} \geq \log_e(n+2)$.

(iii) In (i) you showed that $H_1 \geq \log_e(1+1)$, and in (ii) you showed that **if** $H_n \geq \log_e(n+1)$ **then** $H_{n+1} \geq \log_e((n+1)+1)$. What can you now conclude, and why?

Advanced afterthoughts:

- The above inequality implies that $H_n \geq \log_e n$, since $\log_e(n+1) \geq \log_e n$. It is instructive to try to prove directly, by induction, that $H_n \geq \log_e n$. You will probably run into a snag. This illustrates that for induction to succeed, you sometimes need to prove something that is *stronger* than what you set out to prove.

- Would your proof work for logarithms to other bases, apart from e? Where in the proof do you use the base e?

- It is known that $H_n \leq (\log_e n) + 1$. Can you prove this?

12. (Challenge)
Let E be a finite set, and let $\mathcal{S}$ be a set of subsets of E. We say $\mathcal{S}$ is **linear under** $\triangle$ if, for every two sets $X, Y \in \mathcal{S}$, we have $X \triangle Y \in \mathcal{S}$. So, applying symmetric difference to sets in $\mathcal{S}$ always gives sets in $\mathcal{S}$; the operation never takes you out of $\mathcal{S}$.

The **span** of $\mathcal{S}$, written $\text{span}(\mathcal{S})$, is the set of all symmetric differences of any number of members of $\mathcal{S}$. In other words,

$$\text{span}(\mathcal{S}) := \{X_1 \triangle X_2 \triangle \cdots \triangle X_k : X_1, X_2, \ldots, X_k \in \mathcal{X}, k \in \mathbb{N}\}.$$

By convention, the symmetric difference of zero sets is the empty set. Therefore, the span of any set $\mathcal{S}$ always contains the empty set.

$\mathcal{S}$ is **dependent under** $\triangle$ if, for some $X_1, X_2, \ldots, X_k \in \mathcal{S}$ which are all distinct; i.e., $X_i \neq X_j$ whenever $1 \leq i < j \leq k$, we have

$$X_1 \triangle X_2 \triangle \cdots \triangle X_k = \emptyset.$$

If $\mathcal{S}$ is not dependent under $\triangle$ then we say it is **independent under** $\triangle$.

$\mathcal{S}$ is a **circuit under** $\triangle$ if it is dependent under $\triangle$ and also *minimal* with respect to that property.

$\mathcal{S}$ is a **base under** $\triangle$ if it is independent under $\triangle$ and also *maximal* with respect to that property.

(Recall the precise usage of "minimal" and "maximal" from § 1.7.)

(a) Let $\mathcal{S}$ be linear under $\triangle$. Prove, by induction on k, that for all k and all $X_1, X_2, \ldots, X_k \in \mathcal{S}$,

$$X_1 \triangle X_2 \triangle \cdots \triangle X_k \in \mathcal{S}.$$

(b) Prove that, for every set $\mathcal{S}$ of subsets of E, the set $\mathcal{S}$ is linear under $\triangle$ if and only if it equals its own span, i.e.,

$$\mathcal{S} = \text{span}(\mathcal{S}).$$

(c) Prove by induction on k, that for all k and all $X_1, X_2, \ldots, X_k \in S$, either

$$X_1 \triangle X_2 \triangle \cdots \triangle X_k = \emptyset,$$

or there exist $Y_1, Y_2, \ldots, Y_n \in S$ with $n \leq k$, such that $Y_i \neq Y_j$ for all $i \neq j$ and

$$X_1 \triangle X_2 \triangle \cdots \triangle X_k = Y_1 \triangle Y_2 \triangle \cdots \triangle Y_n.$$

(d) Prove that $\mathcal{S}$ is dependent under $\triangle$ if and only if there exists $X \in \mathcal{S}$ such that

$$X \in \text{span}(\mathcal{S} \setminus \{X\}).$$

(e) Prove that if $\mathcal{S}$ is a minimal dependent set under $\triangle$, then for every $X \in \mathcal{S}$ we have

$$X \in \text{span}(\mathcal{S} \setminus \{X\}).$$

(f) Prove that every minimal spanning set under $\triangle$ is independent under $\triangle$.

(g) Now prove that every minimal spanning set under $\triangle$ is a *maximal* independent set under $\triangle$.

(h) Prove that every maximal independent set under $\triangle$ is also a spanning set under $\triangle$.

(i) Now prove that every maximal independent set under $\triangle$ is also a *minimal* spanning set under $\triangle$.

(j) Prove that $\mathcal{S}$ is a minimal spanning set under $\triangle$ if and only if it is a maximal independent set under $\triangle$.

(k) Prove that, if $\mathcal{S}$ is independent under $\triangle$ then

$$|\text{span}(\mathcal{S})| = 2^{|\mathcal{S}|}.$$

(l) Prove the *independence augmentation property*: if $\mathcal{S}$ and $\mathcal{T}$ are independent under $\triangle$ and $|\mathcal{T}| = |\mathcal{S}| + 1$, then there exits $X \in \mathcal{T} \setminus \mathcal{S}$ such that $\mathcal{S} \cup \{X\}$ is independent under $\triangle$.

(m) Prove that all bases under $\triangle$ have the same size.

(n) Do all minimal dependent sets under $\triangle$ have the same size? Give a proof of your claim.

(o) Prove that, for every base $\mathcal{B}$ of $\mathcal{S}$ under $\triangle$, and every $X \in \mathcal{S}$, there is a unique way to write X as a symmetric difference of members of $\mathcal{B}$, using each element of $\mathcal{B}$ at most once.

(p) For each previous part of this question, (a)–(o), state what type of proof you have used.

(q) Design a simple algorithm for the following problem.

> INPUT: a collection $\mathcal{S}$ of subsets of E, together with a nonnegative real weight for each member of $\mathcal{S}$.
> OUTPUT: a subset $\mathcal{X}$ of $\mathcal{S}$ that is independent under $\triangle$ such that the sum of the weights of all members of $\mathcal{X}$ is maximum.

Does this algorithm always finds the best possible solution? Justify your answer, using a previous exercise.

13. Identify the errors in the following "theorem" and "proof", which are both incorrect.

The steps in the "proof" are numbered for convenience.

"Theorem." For all $n \in \mathbb{N}$, a circular disc can be cut into 2^n pieces by n straight lines.

"Proof."

1. Consider $n = 0$. The disc is not cut at all, so it is still in one piece. So the number of pieces is $2^0 = 1$. So the claim is true for $n = 0$.
2. Consider $n = 1$. A single line across a disc cuts it into two pieces, so the number of pieces is 2^1. So the claim is true for $n = 1$.
3. Now consider $n = 2$. If the circle is cut by one line into two pieces, then we can cut it again by a line that crosses the first line. Such a line must cut right across both of the pieces created by the first cut. Therefore each of those pieces is divided in two, so the total number of pieces is $2 \times 2 = 2^2$, so the claim is true for $n = 2$.
4. It is clear from the cases considered so far that, once the circle is cut by some number of lines, then another line can be used to divide every piece into two pieces, thereby doubling the number of pieces.
5. So, if we use n lines, then the repeated doubling gives 2^n pieces. Therefore the claim is true for all n.

"□"

14. Identify the errors in the following "theorem" and "proof", which are both incorrect.

"Theorem." Every rational number is an integer.

"Proof."

1. We need to show that the sets $\mathbb{Z}$ and $\mathbb{Q}$ are equal.
2. To prove that two sets are equal, we can prove the appropriate subset and superset relations between the two sets.
3. We first prove the subset relation between these two sets, $\mathbb{Z} \subseteq \mathbb{Q}$.
4. Let $x \in \mathbb{Z}$.
5. Since x is an integer, we have $x = \frac{x}{1}$.
6. So x is a quotient of integers.
7. Therefore x is rational, i.e., $x \in \mathbb{Q}$.
8. So we have proved that $\mathbb{Z} \subseteq \mathbb{Q}$.
9. Now we need to prove the superset relation, which is the reverse of subset.
10. So we will prove that $\mathbb{Q} \supseteq \mathbb{Z}$.
11. Let $x \notin \mathbb{Q}$.
12. Then x cannot be written in the form $\frac{p}{q}$ where p and q are integers.
13. So it certainly cannot be written in this form with $q = 1$.
14. Therefore x is not an integer, i.e., $x \notin \mathbb{Z}$.
15. So we have proved that $\mathbb{Q} \supseteq \mathbb{Z}$.
16. It follows from our subset and superset relations between the two sets that $\mathbb{Z} = \mathbb{Q}$.

"□"

> "God made the integers; all else is the work of man."
>
> Leopold Kronecker (1823–1891)

15. Let G be the cryptosystem obtained from Caesar slide by restricting the keyspace to the first half of the alphabet. (See Exercise 2.15 for definitions relating to the Caesar slide cryptosystem.) So the keyspace K_G is

$$\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}\}.$$

Prove that G is not idempotent.

4

PROPOSITIONAL LOGIC

Computers are logical machines. This is not just a description of their behaviour, but also of their nature. Their hardware consists of huge numbers of tiny components that each do simple operations based on logic, and that do them very quickly. In their memory, information is stored as sequences of basic pieces which can each be True or False. The instructions used to get computers to do things — their programs — are expressed in a language that specifies what to do using formal, precise rules that use logic. Even before a program is written, a formal specification of the task to be done will make essential use of logic.

But logic is not just for machines. It is fundamental to clear thinking and precise communication. By learning about logic, you will improve your own thinking and communication, and therefore your prospects in life.

Logic plays a fundamental role in rigorous reasoning, especially in proofs. It will pervade everything we do in this unit and is fundamental to your future studies, not only in computer science but in mathematics, other sciences, engineering, economics, philosophy, and indeed any field of human activity where precise reasoning is important.

We will therefore now study the most fundamental types of logic. This week we study Propositional Logic, and next week we study Predicate Logic.

4.1$^{\alpha}$ TRUTH VALUES

Logic is built on just two values, called **truth values**: False and True.[1] These are abbreviated as F and T respectively.

Physically, truth values may correspond to the absence or presence of electrical current in a wire, or the absence or presence of magnetisation, or a switch being off or on, and so on. We use the abstraction of truth values in order to be able to discuss the logical workings of computers in a way that is independent of the particular technology used.

1 There are forms of logic that use other values too, including three-valued logics which use the extra truth value Unknown. But we will focus entirely on classical two-valued logic, firstly because it is what computers are based on, and secondly because it is embedded within most other logics anyway, so that understanding two-valued logic is necessary in order to understand those other logics.

Another common abstraction of the notion of two possible states for something is the **bit**, which can be 0 or 1. This view is intimately related to truth values: in many situations, we can represent False by 0 and True by 1. It is worth exploring how the mathematical behaviour of 0 and 1 under basic arithmetic operations compares with that of False and True under logical operations. We will develop this link further in **??**.

4.2$^{\alpha}$ BOOLEAN VARIABLES

You are familiar with variables for numerical quantities, whose value can be any number from some set. We have already been using variables for other objects too. So it is natural to use variables whose values are truth values.

A **Boolean variable**, also called a **propositional variable**, is a variable whose value can be True or False.

Often, Boolean variables are used as names for statements which are either True or False.

4.3$^{\alpha}$ PROPOSITIONS

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

Examples

$1+1=2$ — a proposition which is true.

Xià Péisù designed the first computer in China — a proposition which is true.

The earth is flat. — a proposition which is false.

It will rain tomorrow. — a proposition.

The left domino fell. — a proposition.

See § 3.2.

'Twas brillig, and the slithy toves
did gyre and gimble in the wabe. — *not* a proposition.

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

Come and work for us! — *not* a proposition.

This statement is false. — *not* a proposition.

For brevity, a proposition may be given a name, which is a Boolean variable. For example, let X be the proposition $1+1=2$. Then the truth value of X is True.

4.4$^{\alpha}$ 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**. Inside computers, they are implemented in electronic circuits called **logic gates**.

4.5 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, $\neg\text{True} = \text{False}$ and $\neg\text{False} = \text{True}$. 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 $\neg P$ that you may come across: $\sim P$, $\overline{P}$, $-P$, $!P$

We can specify how $\neg$ works using a **truth table**, which lists all possible values of the arguments and, for each, states 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 $\neg P$.

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

Logical negation is reminiscent of set complementation (§ 1.11). In each case, you get something completely opposite, and doing it twice gets you back where you started. For logical negation, we have

$$\neg\neg P \;=\; P.$$

If a proposition asserts membership of a set, then its logical negation asserts membership of the complement. For example, consider the proposition $\sqrt{2} \in \mathbb{Q}$ (which is False) and

suppose our universal set is $\mathbb{R}$. Its logical negation $\neg(\sqrt{2} \in \mathbb{Q})$ may be written $\sqrt{2} \notin \mathbb{Q}$ or $\sqrt{2} \in \mathbb{R} \setminus \mathbb{Q}$ (which is True).

4.6 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**

Conjunction is closely related to set intersection. If x is an object and A and B are sets, then the conjunction of the propositions $x \in A$ and $x \in B$ is the proposition $x \in A \cap B$:

$$(x \in A) \wedge (x \in B) \;=\; (x \in A \cap B).$$

Restating (1.12) using conjunction, we have

$$A \cap B \;=\; \{x : x \in A \wedge x \in B\}.$$

4.7 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". The disjunction of P and Q is written $P \vee Q$ and read as "P or Q". Again, our use of English words is unusually specific: "or" is being used here in a strict, precise, logical sense, 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; we discuss it in § 4.11.)

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**

Disjunction is closely related to set union. If x is an object and A and B are sets, then the disjunction of the propositions $x \in A$ and $x \in B$ is the proposition $x \in A \cup B$:

$$(x \in A) \vee (x \in B) \;=\; (x \in A \cup B).$$

Restating (1.11) using disjunction, we have

$$A \cup B \;=\; \{x : x \in A \vee x \in B\}.$$

At this point, it is worth reflecting on other set operations we discussed in Chapter 1 and thinking about what logical operations they might correspond to. Feel free to invent your own logical operations if that helps.

4.8 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**.

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

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

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, which 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? In other words, *assuming* that $\neg(P \vee Q) = \neg P \wedge \neg Q$ holds for all P and Q, can you *prove* that $\neg(P \wedge Q) = \neg P \vee \neg Q$ holds for all P and Q without starting from scratch again and going through the same kind of detailed truth table argument? (Exercise 4.3)

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 we'd need a separate truth table for each n, which means we'd need infinitely many truth tables.

Theorem 23. For all n:

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

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?

There is a clear correspondence between De Morgan's Laws for Sets, in Theorem 1 and Corollary 2, and De Morgan's Laws for Logic.

4.9 IMPLICATION

Our next logical operation is **implication**, also called **conditional**, from § 3.2. Recall that, 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.

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.

Here is its truth table.

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

As we discussed in § 3.2, implication is closely related to the subset relation.

4.10 EQUIVALENCE

Our next binary logical operation is **equivalence**, also called **biimplication** or **biconditional**. We discussed this previously towards the end of § 1.6 on p. 9, and in the last paragraph of § 3.2 on p. 90.

For arguments P and Q, equivalence 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.

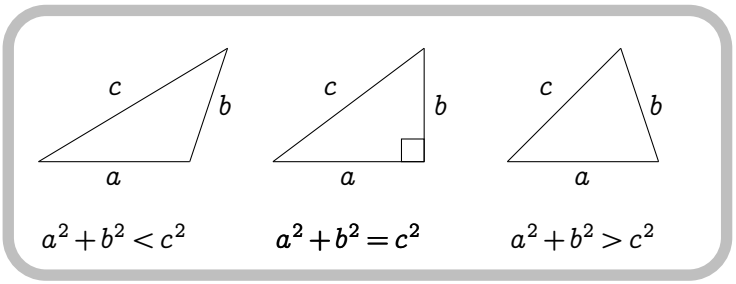
You can view $P \Leftrightarrow Q$ as the conjunction of $P \Rightarrow Q$ and $Q \Rightarrow P$:

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

$P \Leftrightarrow Q$ 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**

Equivalence is closely related to set equality. If x is an object and A and B are sets, then the propositions $x \in A$ and $x \in B$ are equivalent if x belongs to both sets or neither of them. If this holds for all x then the two sets are identical. Conversely, if two sets are identical then the propositions $x \in A$ and $x \in B$ are always equivalent.

4.11 EXCLUSIVE-OR

Our final logical operation is **exclusive-or**. For arguments P and Q, exclusive-or is written $P \oplus Q$ (or sometimes P XOR Q), and read "P exclusive-or Q" or "P ex-or Q". It is true precisely when either P is True or Q is True, *but not both*.

Exclusive-or differs from inclusive-or (disjunction, § 4.7) in its value when both P and Q are True. In that case, exclusive-or is False but inclusive-or is True.

Here is its truth table.

P	Q	$P \oplus Q$
F	F	**F**
F	T	**T**
T	F	**T**
T	T	**F**

It is evident from their truth tables that exclusive-or is actually the logical negation of equivalence:

$$P \oplus Q \quad = \quad \neg(P \Leftrightarrow Q).$$

The exclusive-or of two propositions is True if and only if *exactly one* of them is True. What happens if we combine three propositions with exclusive-or, as in $P_1 \oplus P_2 \oplus P_3$? The exclusive-or of the first two, $P_1 \oplus P_2$, is True precisely when exactly one of P_1 and P_2 is True, and in that case $P_1 \oplus P_2 \oplus P_3$ can only be True if P_3 is False, so we still

have exactly one of the propositions P_1, P_2, P_3 being True. But there is another way for $P_1 \oplus P_2 \oplus P_3$ to be True, namely if $P_1 \oplus P_2$ is False and P_3 is True. For $P_1 \oplus P_2$ to be False, *neither* of P_1 and P_2 is True, or they *both* are. So we might again have exactly one of P_1, P_2, P_3 being True, or we might in fact have *all three* of P_1, P_2, P_3 being True.

We have covered all the possible things that can happen, if $P_1 \oplus P_2 \oplus P_3$ is to be True, and we have found that the number of P_1, P_2, P_3 that are True must be 1 or 3. So it needn't be *exactly* one; fortunately, we have avoided a common mistake there. In fact, what we can say is that the number of P_1, P_2, P_3 that are True must be odd.

This generalises to arbitrary numbers of propositions. You should play with some examples (say, with four propositions P_1, P_2, P_3, P_4), satisfy yourself that this does indeed happen in general, and try to understand why. Then you will get more out of reading the formal proof which we now present.

Theorem 24. For all $n \in \mathbb{N}$, and for any propositions $P_1, P_2, \ldots, P_n$,

$$P_1 \oplus P_2 \oplus \cdots \oplus P_n \quad = \quad \begin{cases} \text{True}, & \text{if an odd number of } P_1, P_2, \ldots, P_n \text{ is True;} \\ \text{False}, & \text{if an even number of } P_1, P_2, \ldots, P_n \text{ is True.} \end{cases}$$

Proof. We prove this by induction on n.

Inductive Basis:

When $n = 1$, the expression on the left of (24) is just P_1, and this is True if if just one of P_1 is True, and False otherwise! So (24) holds in this case.

Inductive Step:

Let $k \geq 1$. Assume (24) holds for $n = k$; this is the Inductive Hypothesis.

Now consider $P_1 \oplus P_2 \oplus \cdots \oplus P_{k+1}$.

$$
\begin{aligned}
& P_1 \oplus P_2 \oplus \cdots \oplus P_{k+1} \\
&= \left(P_1 \oplus P_2 \oplus \cdots \oplus P_k\right) \oplus P_{k+1} \\
&\qquad \text{(identifying a smaller expression of the same type, within this expression)} \\
&= \left(\begin{cases} \text{True}, & \text{if an odd number of } P_1, P_2, \ldots, P_n \text{ is True;} \\ \text{False}, & \text{if an even number of } P_1, P_2, \ldots, P_n \text{ is True.} \end{cases} \right) \oplus P_{k+1} \\
&\qquad \text{(by the Inductive Hypothesis)} \\
&= \begin{cases} \text{True} \oplus P_{k+1}, & \text{if an odd number of } P_1, P_2, \ldots, P_n \text{ is True;} \\ \text{False} \oplus P_{k+1}, & \text{if an even number of } P_1, P_2, \ldots, P_n \text{ is True.} \end{cases} \\
&= \begin{cases} \text{True} \oplus \text{False}, & \text{if an odd number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is False;} \\ \text{True} \oplus \text{True}, & \text{if an odd number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is True;} \\ \text{False} \oplus \text{False}, & \text{if an even number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is False;} \\ \text{False} \oplus \text{True}, & \text{if an even number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is True.} \end{cases} \\
&\qquad \text{(breaking each option down according to the value of } P_{k+1}\text{)} \\
&= \begin{cases} \text{True}, & \text{if an odd number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is False;} \\ \text{False}, & \text{if an odd number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is True;} \\ \text{False}, & \text{if an even number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is False;} \\ \text{True}, & \text{if an even number of } P_1, P_2, \ldots, P_n \text{ is True and } P_{k+1} \text{ is True.} \end{cases} \\
&\qquad \text{(evaluating the exclusive-ors)} \\
&= \begin{cases} \text{True}, & \text{if an odd number of } P_1, P_2, \ldots, P_k, P_{k+1} \text{ is True;} \\ \text{False}, & \text{if an even number of } P_1, P_2, \ldots, P_k, P_{k+1} \text{ is True.} \end{cases} \\
&\qquad \text{(combining the True cases and combining the False cases, and noting that their conditions also combine to capture precisely the evenness or oddness of the number of the } P_i \text{ that are True).}
\end{aligned}
$$

So (24) holds for $n = k+1$ too. This completes the Inductive Step.

Conclusion:

Therefore, by Mathematical Induction, (24) holds for all $n \in \mathbb{N}$. □

Exclusive-or is closely related to the symmetric difference of sets. If x is an object and A and B are sets, then

$$x \in A \triangle B \quad \text{if and only if} \quad (x \in A) \oplus (x \in B).$$

More generally, if $A_1, A_2, \ldots, A_n$ are sets, then

$$x \in A_1 \triangle A_2 \triangle \cdots \triangle A_n \quad \text{if and only if} \quad (x \in A_1) \oplus (x \in A_2) \oplus \cdots \oplus (x \in A_n).$$

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

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.

4.13$^{\omega}$ HISTORY

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

4.14 DISTRIBUTIVE LAWS

$$P \wedge (Q \vee R) \quad = \quad (P \wedge Q) \vee (P \wedge R) \tag{4.1}$$

$$P \vee (Q \wedge R) \quad = \quad (P \vee Q) \wedge (P \vee R) \tag{4.2}$$

It is notable that we have two distributive laws in propositional logic. Conjunction is distributive over disjunction (first law above, (4.1)), and disjunction is distributive over conjunction (second law above, (4.2)).

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

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

Just as for De Morgan's Laws, we see a correspondence between the algebra of sets and the algebra of logic. The Distributive Laws for sets, Theorem 3, correspond to the Distributive Laws for Logic.

4.15 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.

$$\neg\neg P \quad = \quad P$$

$\neg$True	=	False	$\neg$False	=	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
$P \wedge \neg P$	=	False	$P \vee \neg P$	=	True
$P \wedge$ True	=	P	$P \vee$ False	=	P
$P \wedge$ False	=	False	$P \vee$ True	=	True

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

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

4.16 DISJUNCTIVE NORMAL FORM

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

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.

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 expression 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). We do not consider $\neg\neg X$ to be a literal, as it is, but it is equivalent to X, which is a literal. Similarly, $\neg\neg\neg X$ is not a literal but is equivalent to the literal $\neg X$.

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 conjunction 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.
$P \vee Q \vee \neg R$	a disjunction of three parts, each containing a single literal.
$P \wedge Q \wedge R$	a disjunction of one part, which contains three literals.
Z	a disjunction of one part, which contains one literal.

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.

$$P = \underbrace{\overbrace{(\neg X \wedge \neg Y)}^{\text{conjunction}} \vee \overbrace{(\neg X \wedge Y)}^{\text{conjunction}} \vee \overbrace{(X \wedge Y)}^{\text{conjunction}}}_{\textbf{dis}\text{junction}}$$

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, unless the number of variables is very small. (If a Boolean expression is actually provided in the form of its entire truth table, then constructing its equivalent DNF expression by the above method is ok, since the expression you get won't be any larger than the truth table. But if the Boolean expression is provided as a compact formula, then the size of its equivalent DNF expression may be exponential in the size of the formula you started with.)

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.

On the other hand, it does not seem so easy to tell if a DNF expression is a tautology. For a DNF expression to *not* be a tautology, there would have to be some truth assignment to its variables that makes it False. There is no known way to do test for this efficiently.

The big 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.

4.17 CONJUNCTIVE NORMAL FORM

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:

$$\overbrace{\overbrace{(\neg P \vee Q)}^{\text{disjunction}} \wedge \overbrace{(P \vee \neg Q)}^{\text{disjunction}}}^{\textbf{con}\text{junction}}$$

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.

4.18 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 followed, 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 can be tackled later, and is a different skill.)

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:

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

So our expression is now:

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

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 $\neg\text{Hagrid} \vee \text{Norberta}$, 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 $\neg\text{Voldemort} \vee \neg\text{Bellatrix}$. 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?

4.19 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 at least two of them are True, or that at most two of them are True, or exactly two of them are True. We might want to make similar statements with "two" replaced by some other number. Conditions of this kind are examples of *cardinality contraints*, since they are only about the number of variables with a given value.

We saw examples of this in the previous section. We wanted at least *one* of Harry, Ron, Hermione and Ginny to be True. We also wanted at most *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 exactly k variables are True, we can express this as a conjunction:

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

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. The number of disjunctions we use is just the number of subsets of $k+1$ variables chosen from our n variables, which is $\binom{n}{k+1}$.

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 $n=4$). This means that, for every *three* of the variables, at least one of them is False. So, at least one of w, x, y is False, and at least one of w, x, z is False, and so on. But saying that at least one of a set of variables is False is the same as saying that at least one of their *negations* is True. For example, at least one of w, x, y is False if and only if at least one of $\neg w, \neg x, \neg y$ is True. This is now a job for disjunction: at least one of $\neg w, \neg x, \neg y$ is True if and only if $\neg w \vee \neg x \vee \neg y$ is True. So, we create all disjunctions of *triples* ($k+1=3$) of *negated* literals, which gives the disjunctions

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

The number of these disjunctions is $\binom{n}{k+1} = \binom{4}{3} = 4$. These disjunctions 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 *triples* ($n-k+1=4-2+1=3$) 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

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

4.20 UNIVERSAL SETS OF OPERATIONS

We saw in § 4.16 that every logical expression is equivalent to a DNF expression. Later, in § 4.17, we observed that every logical expression is also equivalent to a CNF expression.

One consequence of this is that every logical expression is equivalent to an expression that only uses the operations from the operation set $\{\wedge, \vee, \neg\}$.

We say that a set of operations X is **universal** if every logical expression P is equivalent to an expression Q that only uses operations in that set (together with variables from the original expression). So the set of operations that actually appear in Q must be a subset of the operation set X.

Our observations above, about DNF and CNF, demonstrate that the operation set

$$\{\wedge, \vee, \neg\}$$

is universal.

One reason for considering universal sets of operations relates to the construction of electronic circuits that compute the values of logical expressions. It is easier to make these circuits if we can put them together from simple components of only a few different types. Simple components are easier to construct, and the complexity of manufacturing is reduced if we don't have to make too many different types of components. We also gain economies of scale from making large numbers of these simple components instead of smaller numbers of more complex components.

The operation set $\{\wedge, \vee, \neg\}$ is not the smallest possible universal set of operations. De Morgan's Laws (§ 4.8) show how to express $\wedge$ in terms of $\vee$ and $\neg$, and dually how to express $\vee$ in terms of $\wedge$ and $\neg$. So either $\wedge$ or $\vee$ could be dropped from our universal set, as long as we retain the other as well as $\neg$. So the operation set

$$\{\wedge, \neg\}$$

is also universal, and so is

$$\{\vee, \neg\}.$$

Having just seen that we can drop either $\wedge$ or $\vee$ from our universal set $\{\wedge, \vee, \neg\}$, while still retaining the universal property, it is natural to ask if we can drop $\neg$ and keep the other two. In other words, is the set

$$\{\wedge, \vee\}$$

universal?

Having found universal sets of just two operations, we can ask if this is the smallest possible. Does there exist a universal set of just one operation?

Clearly the set $\{\neg\}$ is not universal, since $\neg$ is a unary operation that cannot be used to combine logical variables together. So, if we seek a single universal operation, we should start by looking at binary operations.

4.21 EXERCISES

1. Suppose we have the following propositions B, M, S regarding an overseas trip you are doing.

B: you visit Brazil
M: you visit Malaysia
S: you visit Singapore

Use B, M and S to write a Boolean expression which is true if and only if you visit Brazil or both the other countries (but not all three).

2. Suppose we have the following propositions P, S about a certain unknown integer.

N: the integer is negative
P: the integer is prime
S: the integer is a square

Use P, S to write a Boolean expression with the following meaning:

If the integer is a square, then it is neither negative nor prime.

3. Assuming the first of De Morgan's Laws,

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

prove the second of De Morgan's Laws,

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

as simply as possible.

4. Prove the two Distributive Laws ((4.1) and (4.2) in § 4.14).

5. Do you need parentheses in the expression $a \wedge b \vee c$? Investigate the two expressions

$$a \wedge (b \vee c) \qquad \text{and} \qquad (a \wedge b) \vee c.$$

Are they logically equivalent? If not, what can you say about the relationship between them? Does either imply the other?

6.
Prove that

$$(P_1 \wedge \cdots \wedge P_n) \Rightarrow C \quad \text{is logically equivalent to} \quad \neg P_1 \vee \cdots \vee \neg P_n \vee C$$

A disjunction of the form $\neg P_1 \vee \cdots \vee \neg P_n \vee C$ is called a *Horn clause*. These play a big role in the theory of logic programming.

7. Simplify the following expression as much as possible, using Boolean algebra:

$$(\neg X \wedge \neg Y) \vee (\neg X \wedge Y) \vee (X \wedge Y).$$

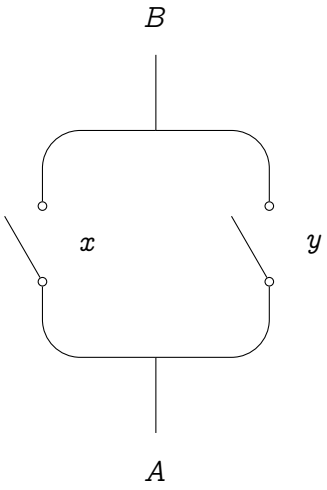
8. This question is about using Boolean algebra to describe the algebra of switching. An electrical switch is represented diagrammatically as follows.

The state of a switch can be represented by a Boolean variable, with the states Off and On being represented by False and True respectively. Let x be a Boolean variable representing the state of a switch. Then $x =$ True represents the switch being On, so that electrical current can pass through it; $x =$ False represents the switch being Off, so that there is no electrical current through the switch.

We can put switches together to make more complicated circuits. Those circuits can then be described by Boolean expressions in the variables that represent the switches.

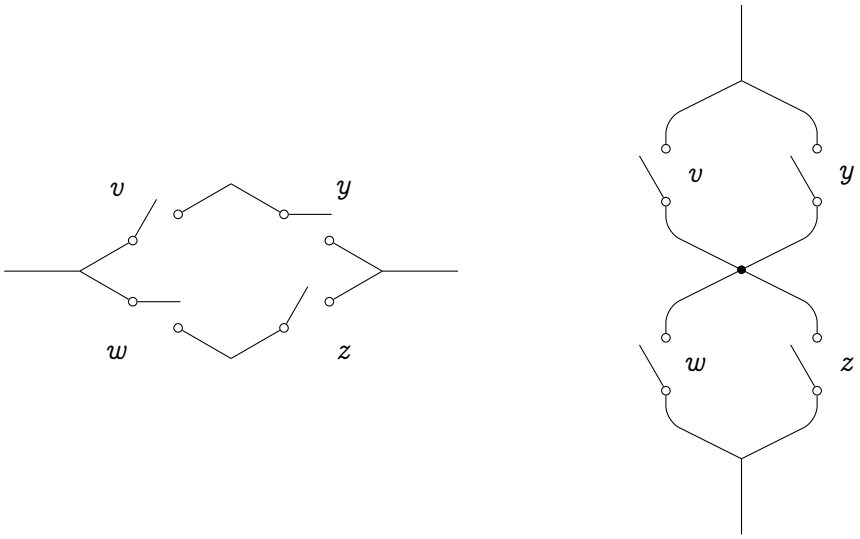
In the following circuits, v, w, x, y, z are Boolean variables representing the indicated switches.

(a) For each of the two switching circuits below, write a Boolean expression in x and y for the proposition that current flows between the two ends of the circuit. (The ends are shown as A and B in the diagram. In effect, the diagrams only show part of a complete circuit. The rest of the circuit would contain a power supply, such as a battery, and an electrical device that operates when the current flows, such as a light or an appliance.)



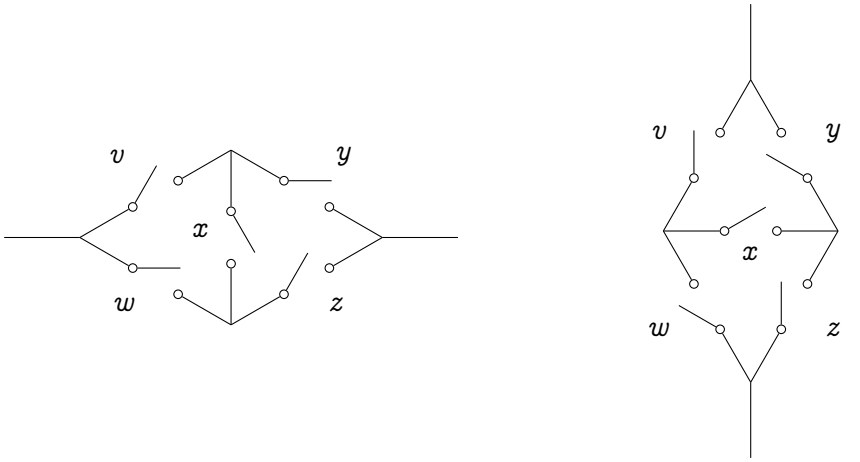
(b) For each of the next two circuits, again construct a Boolean expression to represent the proposition that current flows between the two ends of the circuit.

Compare the two expressions you construct. What can you say about the relationship between them?



(c) Similarly, construct Boolean expressions for the next two circuits.

Compare the two expressions you construct. What can you say about the relationship between them?



(d) For each of our circuit pairs in (a)–(c), discuss how they are related to each other, and what this means for how the Boolean expressions derived from them are related to each other.

9. Does logically negating an implication reverse it? In other words, are the expressions

$$\neg(P \Rightarrow Q) \qquad \text{and} \qquad P \Leftarrow Q$$

logically equivalent? If so, prove it; if not, determine (with proof) if either implies the other.

10. Prove that

$$(P \wedge (P \Rightarrow Q)) \Rightarrow Q$$

is a tautology. Comment on what this means for logical deduction.

11. Prove that

(a) $P \oplus Q \;=\; (P \wedge \neg Q) \vee (\neg P \wedge Q)$.

(b) $P \oplus Q \;=\; (P \vee Q) \wedge (\neg P \wedge \neg Q)$.

12.

(a) Express $P \vee Q$ using just the operations $\oplus$ and $\wedge$, and no negation.

(b) Express $P \wedge Q$ using just the operations $\oplus$ and $\vee$, and no negation.

13. We saw two distributive laws for $\vee$ and $\wedge$ in § 4.14, and contrasted that with the situation for ordinary numbers under $+$ and $\times$, when only one distributive law holds.

Investigate the situation for $\oplus$ and $\wedge$. Do you have two distributive laws, or one (and which one?), or none? Give explanations for your answers.

14. Consider the following truth table for a proposition P in terms of X, Y and Z.

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

(a) Express P in Disjunctive Normal Form (DNF).

(b) Express $\neg P$ in DNF.

(c) Using (b) as your starting point, express P in Conjunctive Normal Form (CNF).

(d) An alternative way to try to express P in CNF might be to start from the DNF for P you found in (a) and then expand the whole thing using the Distributive Laws. If you start with your expression from (a) and do all this expansion without yet doing any further simplification, how many parts are combined together in the large conjunction

you construct? How many literals does it have? How does this approach compare with (c), with respect to ease and efficiency?

15.
A meeting about moon mission software is held at NASA in 1969. Participants may include Judith Cohen (electrical engineer), Margaret Hamilton (computer scientist), and Katherine Johnson (mathematician). Let Judith, Margaret and Katherine be propositions with the following meanings.

Judith	Judith Cohen is in the meeting.
Margaret	Margaret Hamilton is in the meeting.
Katherine	Katherine Johnson is in the meeting.

For each of the following statements, write a proposition in Conjunctive Normal Form with the same meaning.

(a) Judith and Margaret are not both in the meeting.
(b) Either Judith or Margaret, but not both of them, is in the meeting. (This is the "exclusive-OR".)
(c) At least one of Judith, Margaret and Katherine is in the meeting.
(d) At most one of Judith, Margaret and Katherine is in the meeting.
(e) Exactly one of Judith, Margaret and Katherine is in the meeting.
(f) At least two of Judith, Margaret and Katherine are in the meeting.
(g) At most two of Judith, Margaret and Katherine are in the meeting.
(h) Exactly two of Judith, Margaret and Katherine are in the meeting.
(i) Exactly three of Judith, Margaret and Katherine are in the meeting.
(j) None of Judith, Margaret and Katherine is in the meeting.

16.
Recall the logical expression given on p. 139 in § 4.18 for your dinner party guest list:

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

How long would an equivalent DNF expression be? Specifically, how many disjuncts — smaller expressions combined using $\vee$ to make the whole expression — would it have?

17. A Boolean function is in **algebraic normal form** if it is an exclusive-or of some number of parts, where each part is a conjunction of unnegated variables or the logical constant True, and we also allow the entire expression to be just the logical constant False (but otherwise False is not allowed to appear in the expression).

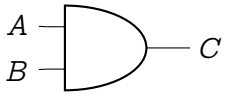


Figure 4.1: An AND gate. A and B are the inputs and C is the output (which is therefore true if and only if both A and B are true).

So, for example, the following expressions are in algebraic normal form:

$$x \oplus y \oplus (x \wedge y), \qquad \mathsf{True} \oplus y \oplus (x \wedge y \wedge z), \qquad x, \qquad \mathsf{True} \oplus x, \qquad x \wedge z, \qquad \mathsf{True}, \qquad \mathsf{False}.$$

But the following expressions are not in algebraic normal form:

$$x \oplus y \oplus (\neg x \wedge y), \qquad \mathsf{False} \oplus y \oplus (x \wedge y \wedge z), \qquad \neg x, \qquad x \vee z, \qquad \mathsf{True} \oplus \mathsf{True}.$$

In Exercise 12(a), you expressed the disjunction of two variables in algebraic normal form.

Prove that every Boolean function can be written in algebraic normal form. (Use Exercise 12(a) and what you know about DNF.)

18. Suppose you have an unlimited supply of AND gates, which are logic gates for the binary operation $\wedge$. Each gate has two inputs, for the two arguments, and one output, which gives the conjunction of the inputs. The output of one gate can be used for the input of another gate.

To represent an AND gate in a circuit diagram we will use the symbol in Figure 4.1.

Show how to put AND gates together to compute the conjunction of eight logical arguments $x_1, x_2, \ldots, x_8$.

How many AND gates do you need?

Suppose each AND gate computes its output as soon as both its inputs are available, and that it takes time t to compute the output. Assume that all the initial inputs $x_1, x_2, \ldots, x_8$ are available simultaneously, at time 0. How long does your combination of AND gates take to compute its output?

Try and put your AND gates together to minimise the total time taken to compute the final output.

19. Prove, by induction on n, that the conjunction of 2^n inputs can be computed in time nt by a suitable combination of AND gates.

Hence prove that the conjunction of m arguments (where m is not necessarily a power of 2) can be computed by a suitable combination of AND gates in time $\lceil \log_2 m \rceil$.

20. A Boolean expression is called **affine** if it is of one of the following forms:

$$x_1 \oplus x_2 \oplus \cdots \oplus x_n$$
or
$$x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus \text{True}.$$

Prove, by induction on n, that for all $n \in \mathbb{N}$, the number of satisfying truth assignments of an affine Boolean expression with n variables is 2^{n-1}.

It follows that half of all truth assignments are satisfying for the expression, and half are not.

Here, a **truth assignment** is just an assignment of a truth value to each variable, and it is **satisfying** if the assignment makes the whole expression True.

21. Let x_1x_0 be the bits of a two-bit binary number x, representing an integer in $\{0,1,2,3\}$. So $x_1, x_0 \in \{0,1\}$ and we have $x = 2x_1 + x_0$.

Let $y_2y_1y_0$ be the three-bit binary representation of the number $x+1$ obtained from x by incrementing it. So $y \in \{1,2,3,4\}$, and its leading bit is 0 except if $x = 3$, in which case $y = 4$ which in binary is `100`.

In this exercise we use 0 and 1 for the truth values False and True (as mentioned on p. 120 in § 4.1$^{\alpha}$).

Give Boolean expressions for each of the three bits of y, in terms of the two bits x_1 and x_0 of x.

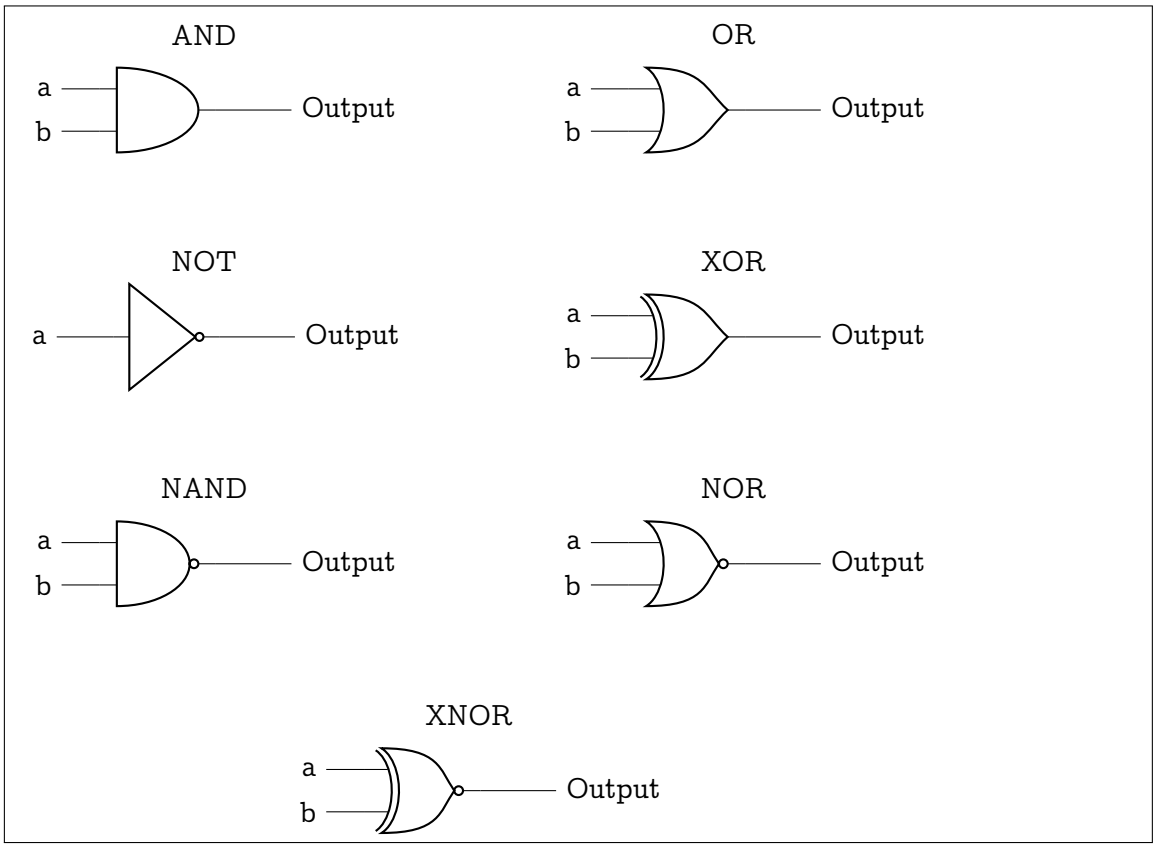
22. Is the operation set $\{\oplus, \neg\}$ universal? Justify your answer.

23. Prove or disprove the claim that there exists a universal operation set containing just a single operation.

You need not restrict consideration to the three operations $\wedge, \vee, \neg$. You can use another operation of at most two arguments (in other words, a binary operation) if you wish.

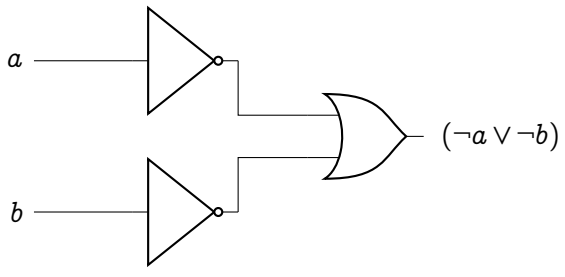
What type of proof did you use? (Recall the proof types discussed in Chapter 3.)

24. Computer circuits actually perform the very same Boolean logic that we have studied using *logic gates* (§ 4.4$^{\alpha}$). The idea is that you have one or more wires coming into a gate, and usually one output wire. If an input wire has current flowing through, the variable it represents is set to True, and if not, then False. The logic gate then takes those signals and gives off an output signal if the result should be True, and no signal if the result is False. Below are the most common logic gates seen in circuits.



The AND, OR, and NOT gates function the same as what we have seen previously. If both signals of an AND gate are on, it will output a signal. For OR, only one input signal is required to be on, and for NOT, there will only be an output signal if the input signal is off (False).

XOR is "EXCLUSIVE OR", which outputs a signal only if a or b are on, but not both. NAND, NOR, and XNOR are simply the inversions of AND, OR, and XOR respectively (i.e. NAND is "not both", NOR is "neither", and XNOR is "both or neither"). These gates can also be connected in series, like in the example below, which is equivalent to the logical expression $(\neg a \vee \neg b)$.



(a) What single logic gate is the above circuit equivalent to?

(b) We saw in § 4.20, that $\{\vee, \wedge, \neg\}$ is a *universal set of operations*, meaning that any logical expression can be expressed with only these operations. How would you express the function of an XOR gate as a Boolean expression using only these operations?

(c) Draw a circuit diagram which performs the same functionality as an XOR gate using only AND, OR, and NOT gates.

(d) Do we even need three types of gates? Given that $\{\wedge, \vee, \neg\}$ is a universal set of operations, use De Morgan's Law to prove that $\{\wedge, \neg\}$ and $\{\vee, \neg\}$ are also a universal sets of operations. *(Hint: Can you express $\vee$ with the other two operators?)*

(e) Draw a circuit diagram that performs an OR operation using only AND and NOT gates, and also one which performs AND using only OR and NOT gates.

(f) Can you perform the function of a NOT gate, or an AND gate, with only NAND gates? What does this tell you about NAND gates?