Mathematics for Information Technology

Discrete Mathematics

2016

School of Mathematical Sciences

THE UNIVERSITY of ADELAIDE

Version of May 14, 2019.

Contents

Introduction

A famous quip has it that "Computer Science is merely the post-Turing decline of formal systems theory". To make sense of this, we need to know that *formal systems theory* refers to the algebraic manipulation of symbols and data, and it studies which problems can be solved in principle by a well-defined iterative procedure (an *algorithm*) on an idealised machine. The famous mathematician Alan Turing was one of the pioneers in developing abstract models of computability. *Computer science* evolved as the science of automating the algorithms developed by formal system theory. This added the need to consider technical issues on top of the purely mathematical ones, such as the construction of processing units or the physical representation of abstract data.

This course aims to convey a basic understanding of the mathematical tools and formalisms required by computer scientists to make sense of the theoretical framework of their profession. It begins in Chapters 1 and 2 with the concepts from set theory and logic, the language in which all of mathematics is expressed. Equipped with these fundamentals, we can already look at the practical application of logic to the design of digital circuits in Chapter 3. In Chapter 4 the proofs by induction are introduced as one of the main tools in the analysis of data structures and algorithm performance. Then modular arithmetic and some elementary number theory are discussed in Chapter 5. Modular arithmetic is the way the computations in a digital processor are performed, and with a little knowledge of it we can already understand some very effective problem solving methods. It is also the foundation of modern cryptography, which is introduced in Chapter 6 in the guise of the RSA encryption method. Finally, graphs and their basic properties are introduced in Chapter 7. Graphs are the most fundamental and most versatile data structures in computer science, and they are easily accessible for mathematical analysis.

1 Sets and relations

In this first chapter we get acquainted with the language of mathematics by studying its most basic concepts, *sets* and *functions*. It is a short account of the "naive" set theory developed by Georg Cantor in the late 19th century.

1.1 Notation and basic properties of sets

A **set** is a collection of distinct objects, called the **elements** of the set.

We specify a set by

- either listing the elements
- or describing the elements of the set according to some formula.

Example 1.1 There are two ways to describe the set of odd numbers between 1 and 11:

$$A = \{1, 3, 5, 7, 9, 11\}$$

or

$$A = \{x \mid x \text{ is an odd integer and } 1 \leq x \leq 11\}.$$

In the second notation, we can read the | as "such that".

Note: Since we require the objects of a set to be distinct, one particular object cannot appear more than once in a set. So $\{1, 2\}$ and $\{1, 1, 2, 2\}$ denote the same set. Also, we do not assume any order in a set, so that $\{1, 2\}$ and $\{2, 1\}$ again denote the same set.

Most of the sets we will encounter are sets of *numbers*, such as the following

$$\begin{aligned}
&\mathbb{N} = \{0, 1, 2, 3, \ldots\} = \text{non-negative integers} \\
&\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} = \text{integers} \\
&\mathbb{Q} = \left\{ \frac{p}{q} \;\middle|\; p, q \in \mathbb{Z}, q \neq 0 \right\} = \text{rational numbers} \\
&\mathbb{R} = \text{real numbers.}
\end{aligned}$$

However, sets can contain objects other than numbers as elements.

Example 1.2

$$A = \{x \mid x \text{ is green and eats stones}\}$$
$$B = \{\text{true}, \text{false}\}$$
$$C = \{\text{blue}, \text{white}, \text{red}\}$$
$$D = \{\clubsuit, \diamond, \heartsuit, \spadesuit\}.$$

If an object x is an element of a set A we write

$$x \in A.$$

Otherwise, we write $x \notin A$ to indicate that x is not an element of A.

Example 1.3 With A as in Example 1.1,

$$3 \in A, \quad \text{but } 15 \notin A, \quad 10 \notin A.$$

A **finite set** S is a set which contains a finite number of elements. The number of elements in a set S is called the **order** or **cardinality** of S and denoted by $|S|$.

Example 1.4 The set A has order 6, $|A| = 6$. The set of real numbers $\mathbb{R}$ is of course an *infinite* set.

Two sets A and B are **equal**,

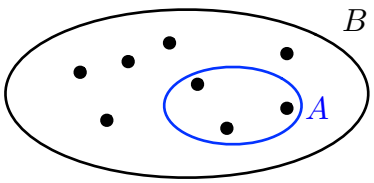
$$A = B,$$

if they have the same elements, that is, $x \in A$ if and only if $x \in B$. If this is not the case, then we write $A \neq B$.

A is a **subset** of B,

$$A \subseteq B,$$

if every element of A is also an element of B, that is, if $x \in A$ then $x \in B$.



If there exists an element $x \in A$ that is not an element of B, $x \notin B$, then we write $A \not\subseteq B$.

If both sets A and B are given by some formula, say $A = \{x \mid x \text{ satisfies } F_A(x)\}$ and $B = \{x \mid x \text{ satisfies } F_B(x)\}$, then we can prove that A is a subset of B by showing that formula $F_A(x)$ implies $F_B(x)$ for all x. To show $A = B$ we show $A \subseteq B$ and $B \subseteq A$.

Example 1.5 Consider the sets

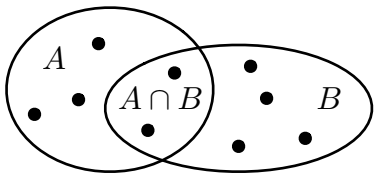
$$A = \{x \mid x \in \mathbb{N}, x > 1 \text{ and } x \text{ divides } 32\}$$
$$B = \{x \mid x \in \mathbb{N} \text{ is even}\}.$$

Then $A \subseteq B$: If $x \in A$, then x divides 32. But the only numbers $x > 1$ dividing 32 are 2, 4, 8, 16 and 32 itself. These are all even numbers. So $A \subseteq B$. However, $B \not\subseteq A$, because $34 \in B$, but $34 \notin A$.

The **intersection** of A and B is

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\},$$

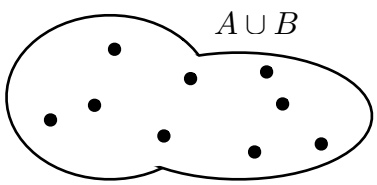
that is, the set of elements in both A and B.



The **union** of A and B is

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\},$$

that is, the set of elements in either A or B (or both).



Example 1.6

$$\begin{aligned} A &= \{1,3,5,7,9,11\} \\ B &= \{3,6,9\} \\ A \cup B &= \{1,3,5,6,7,9,11\} \\ A \cap B &= \{3,9\} \end{aligned}$$

Note that $A \nsubseteq B$ and also $B \nsubseteq A$; however if $D = \{1,3,5\}$ then $D \subseteq A$.

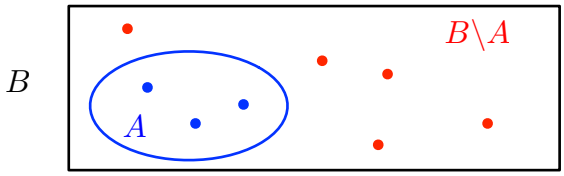
The **difference** of two sets A and B is

$$A \backslash B = \{x \mid x \in A \text{ and } x \notin B\}.$$

Suppose A is a subset of B. Then $B \backslash A$ is also a subset of B called the **complement of A in B**,

$$\begin{aligned} A^c &= \{x \mid x \notin A\} \\ &= \{x \mid x \in B \text{ and } x \notin A\}. \end{aligned}$$

Of course, there can be many sets containing A as a subset, so when using the notation A^c, it has to be clear from the context with respect to which set $B \supseteq A$ we take the complement.



We will often assume that all our sets and elements belong to some large set. We call this the **universal set**. Suppose we have a universal set U. By default, we can assume a complement A^c to be taken with respect to this universal set.

Example 1.7 When studying random experiments in probability theory, the universal set is the sample space Ω of all possible outcomes. Subsets $A \subseteq \Omega$ are called events, and the events A and A^c are mutually excluive outcomes.

Example 1.8 $U = \{1,2,3,4,5,6,7,8,9,10,11\}$ and A, B as in Example 1.6.

$$\begin{aligned} A^c &= \{2,4,6,8,10\}, \\ B^c &= \{1,2,4,5,7,8,10,11\}, \\ A\backslash B &= \{1,5,7,11\}. \end{aligned}$$

The **empty set** $\varnothing$ is the set which has no elements, so $\varnothing = \{\}$.

Example 1.9 Some appearances of the empty set:

(a) With a universal set $U = \{1,2,3,4,5,6,7,8,9,10,11\}$ and $A = \{1,3,5,7,9,11\}$, $B = \{2,4,6,8\}$ we have

$$A \cap B = \varnothing.$$

(b) The set $\{0\}$ has an element, namely 0, so it is *not* the empty set.

(c) Let U be any universal set and let $A \subseteq U$. Then

$$\begin{aligned} A \cap A^c &= \{x \mid x \in A \text{ and } x \in A^c\} \\ &= \{x \mid x \in A \text{ and } x \notin A\} \\ &= \varnothing. \end{aligned}$$

The elements of a set do not always have to be numbers. For example, sets can have other sets as elements: The **power set** $\mathcal{P}(A)$ of a set A is the set of all subsets of A.

Example 1.10 Some examples of power sets:

(a) $A = \{a,b,c\}$.

$$\mathcal{P}(A) = \Big\{\varnothing, \{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\} = A\Big\}$$

(b) Consider $A = \{a\}$. Then

$$\mathcal{P}(A) = \{\varnothing, \{a\}\}.$$

The power set itself is a set, so we can also take the power set of $\mathcal{P}(A)$:

$$\begin{aligned} \mathcal{P}(\mathcal{P}(A)) &= \text{set of all subsets of } \mathcal{P}(A) \\ &= \Big\{\varnothing, \{\varnothing\}, \{\{a\}\}, \{\varnothing, \{a\}\}\Big\} \end{aligned}$$

Note: $\varnothing$ and $\{\varnothing\}$ are different: $\varnothing$ does not contain any element, but $\{\varnothing\}$ has one element, namely $\varnothing$.

Exercise 1.1 If the finite set A has n elements, how many elements are there in $\mathcal{P}(A)$?

Sets satisfy many algebraic properties involving $\cap, \cup, {}^{c}$:

Theorem 1.11 *Let A, B, C be subsets of a universal set U.*

(a) *Involution:*

$$(A^{c})^{c} = A$$

(b) *Commutative laws:*

$$A \cup B = B \cup A$$
$$A \cap B = B \cap A$$

(c) *Associative laws:*

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

(d) *Distributive law:*

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

(e) *Idempotent laws:*

$$A \cup A = A$$
$$A \cap A = A$$

(f) *Identity laws:*

$$A \cap \varnothing = \varnothing$$
$$A \cup \varnothing = A$$

(g) *Identity law for a subset $B \subseteq A$:*

$$B \cap A = B$$
$$B \cup A = A$$

(h) *Inverse laws:*

$$A \cup A^c = U$$
$$A \cap A^c = \varnothing$$

(i) *De Morgan's laws:*

$$(A \cup B)^c = A^c \cap B^c$$
$$(A \cap B)^c = A^c \cup B^c$$

PROOF: We prove only the first distributive law:

$$\begin{aligned} A \cap (B \cup C) &= \{x \mid x \in A \text{ and } x \in B \cup C\} \\ &= \{x \mid x \in A \text{ and } (x \in B \text{ or } x \in C)\} \\ &= \{x \mid (x \in A \text{ and } x \in B) \text{ or } (x \in A \text{ and } x \in C)\} \\ &= \{x \mid x \in A \cap B \text{ or } x \in A \cap C\} \\ &= (A \cap B) \cup (A \cap C) \end{aligned}$$

as required. ■

Exercise 1.2 Prove the remaining laws in Theorem 1.11.

1.2 Relations

The **Cartesian product** of two sets S and T is the set

$$S \times T = \{(s,t) \mid s \in S \text{ and } t \in T\} = \begin{array}{l}\text{set of all ordered pairs with} \\ \text{the first element from } S \text{ and} \\ \text{the second element from } T.\end{array}$$

Note: $(s,t) = (u,v)$ if and only if

$$s = u \text{ and } t = v.$$

In particular, $(s,t) = (t,s)$ only if $s = t$.

Example 1.12 $S = \mathbb{R}$, $T = \mathbb{R}$ where $\mathbb{R}$ denotes the set of real numbers. The Cartesian product

$$S \times T = \mathbb{R} \times \mathbb{R} = \{(x,y) \mid x \in \mathbb{R} \text{ and } y \in \mathbb{R}\}$$

is also called the **Cartesian plane**, because we can think of the numbers x and y as the **Cartesian coordinates** describing the location of a point in the plane.

Example 1.13 $S = \{0, 1\}$, $T = \{1, 2, 3\}$.

$$S \times T = \{(0,1), (0,2), (0,3), (1,1), (1,2), (1,3)\}.$$

Note: In general $S \times T \neq T \times S$.

In this example, $(0,3) \in S \times T$ since $0 \in S$, $3 \in T$ but $(0,3) \notin T \times S$ since $0 \notin T$, $3 \notin S$.

A **relation** $\mathcal{R}$ from a set S to a set T is a subset of $S \times T$, that is,

$$\mathcal{R} \subseteq S \times T.$$

If $(s,t) \in \mathcal{R}$ we say s is related to t and we write $s\mathcal{R}t$.

Note: We say $\mathcal{R}$ is a **binary relation** as it is a subset of the Cartesian product of the *two* sets S and T.

Example 1.14 Let $S = T = \mathbb{R}$. Consider the relation

$$\mathcal{R} = \{(x,y) \mid x \leq y\} \subseteq \mathbb{R} \times \mathbb{R}.$$

That is $(x,y) \in \mathcal{R}$ if and only if $x \leq y$. So $(2,3) \in \mathcal{R}$ and $(3,3) \in \mathcal{R}$, but $(4,3) \notin \mathcal{R}$.

Example 1.15 Take $\mathtt{S}$ = set of character strings, where a *character string* is a string of symbols which can be entered from a keyboard, and $\mathbb{N} = \{0, 1, 2, \ldots\}$ the set of non-negative integers. Let

$$\mathcal{R} = \{(\mathtt{c}, n) \mid \text{the character string } \mathtt{c} \text{ has length } n\} \subseteq \mathtt{S} \times \mathbb{N}.$$

Thus $(\mathtt{b} \times \mathtt{2@+}, 5) \in \mathcal{R}$, but $(\mathtt{abc}, 4) \notin \mathcal{R}$.

Example 1.16 A *ternary* relation: Let

S = set of students at The University of Adelaide

T = set of subjects offered

U = set of tutorial times.

An interesting ternary relation is

$\mathcal{R} = \{(x,y,z) \mid \text{student } x \text{ takes subject } y \text{ and has a tutorial in } y \text{ at time } z\}$
$\subseteq S \times T \times U.$

1.3 Properties of relations

If $\mathcal{R} \subseteq S \times S$ then $\mathcal{R}$ is a **relation on S**. For such relations we define:

$\mathcal{R}$ is **reflexive** if $s\mathcal{R}s$ for all $s \in S$.

$\mathcal{R}$ is **symmetric** if for all $s, t \in S$: if $s\mathcal{R}t$, then $t\mathcal{R}s$.

$\mathcal{R}$ is **transitive** if for all $s, t, u \in S$: if $s\mathcal{R}t$ and $t\mathcal{R}u$, then $s\mathcal{R}u$.

A relation $\mathcal{R}$ on S which is *reflexive, symmetric and transitive* is called an **equivalence relation**.

For equivalence relations, we write $x \sim y$ instead of $x\mathcal{R}y$.

Example 1.17 $\mathcal{R} = \{(x, y) \mid x \leq y\} \subseteq \mathbb{R} \times \mathbb{R}$.

(a) $\mathcal{R}$ is reflexive since $x \leq x$.

(b) $\mathcal{R}$ is not symmetric, since $x \leq y$ does not imply $y \leq x$ (it follows that $\mathcal{R}$ is not an equivalence relation). In fact, $x \leq y$ and $y \leq x$ both hold if and only if $x = y$.

(c) $\mathcal{R}$ is transitive: if $x \leq y$ and $y \leq z$, then also $x \leq z$.

Example 1.18 Let A denote all people living in Australia and

$$\mathcal{R} = \{(x, y) \mid x \text{ lives close to } y\},$$

where we say "x lives close to y" if they live at most 50 kilometers apart.

$\mathcal{R}$ is reflexive and symmetric but *not* transitive.

Example 1.19 Consider $\mathcal{R} = \{(x, y) \mid x = y\} \subseteq \mathbb{R} \times \mathbb{R}$.

(a) $\mathcal{R}$ is reflexive, since $x = x$.

(b) $\mathcal{R}$ is symmetric: if $x = y$ then of course also $y = x$.

(c) $\mathcal{R}$ is transitive: if $x = y$ and $y = z$, then also $x = z$.

Hence $\mathcal{R}$ is an equivalence relation.

1.4 Congruence relations

We will now study a particularly important example of an equivalence relation. Let $S = \mathbb{Z}$ and let n be a fixed positive integer. Any integer $x \in \mathbb{Z}$ can represented in the form

$$x = nq + r, \qquad \text{where } 0 \leq r < n$$

by performing a *division by n with remainder*. The integer q is called the **quotient** and the integer r is called the **remainder**.

Note: For given n, the quotient q and the remainder r are uniquely determined for each integer x.

If two numbers $x, y \in \mathbb{Z}$ have the same remainder when divided by n, we write

$$x \equiv y \mod n.$$

Read this as "x and y are congruent modulo n".

The relation **congruence modulo n** on $\mathbb{Z}$ is

$$\mathcal{R} = \{(x, y) \mid x \equiv y \mod n\}.$$

Example 1.20 Let $n = 10$. Then

$$17 = 1 \cdot 10 + 7 \quad \text{and} \quad -113 = -12 \cdot 10 + 7$$

so that

$$17 \equiv -113 \mod 10.$$

But

$$-37 = -4 \cdot 10 + 3$$

so

$$-37 \not\equiv 17 \mod 10.$$

Suppose $x \equiv y \bmod n$ for some fixed n. Then

$$x = q \cdot n + r$$
$$y = q' \cdot n + r$$

where $0 \leq r < n$. Subtracting,

$$x - y = (q - q')n.$$

Thus $x - y$ is a multiple of n, or in other words, n divides $x - y$. We can therefore describe congruence mod n in another (equivalent) way:

Theorem 1.21 *$x \equiv y \bmod n$ if and only if $x - y$ is a multiple of n.*

PROOF: We saw above that $x \equiv y \bmod n$ implies that $x - y$ is a multiple of n. Now show that $x - y$ being a multiple of n also implies $x \equiv y \bmod n$: Suppose $x - y = kn$ for some $k \in \mathbb{Z}$. Division with remainder allows us to write $y = q'n + r$, where $0 \leq r < n$. But then

$$x = y + kn = q'n + r + kn = (q' + k)n + r,$$

and this just means x and y have the same remainder r on division by n, so $x \equiv y \bmod n$. ∎

Example 1.22 Let $n = 10$.

(a) $-113 \equiv 17 \bmod 10$, as $-113 - 17 = -130$ is divisible by 10.

(b) $-37 \not\equiv 17 \bmod 10$, as $-37 - 17 = -54$ is *not* divisible by 10.

Example 1.23 Let $n = 2$.

$x \equiv 0 \bmod 2$ if and only if x is an *even* integer,
$x \equiv 1 \bmod 2$ if and only if x is an *odd* integer.

Theorem 1.24 *Congruence modulo n is an equivalence relation.*

PROOF: Recall that $x \equiv y \bmod n$ if they have the same remainder when divided by n.

(a) $\equiv$ is *reflexive*: $x - x = 0$ is certainly divisible by n.

(b) $\equiv$ is *symmetric*: if x and y have the same remainder on division by n, then y and x have the same remainder on division by n. So $x \equiv y \bmod n$ implies $y \equiv x \bmod n$.

(c) $\equiv$ is *transitive*: suppose x and y both have remainder r on division by n, and y and z both have remainder r' on division by n. Then $r = r'$, since the remainder is unique, and both r and r' are remainders for y on divison by n. This means x and z both have remainder r on division by n. ∎

1.5 Partitions and equivalence classes

A collection of subsets $A_1, A_2, \ldots$ of a set A is called a **partition** of A if

(i) $A = A_1 \cup A_2 \cup \ldots$

(ii) $A_i \cap A_j = \varnothing$ for all $i, j = 1, \ldots, n$.

If A is finite, then a partition contains only finitely many subsets $A_1, \ldots, A_n$.

Example 1.25 Let $A = \{1, 2, 3, 4, 5, 6\}$. The sets $A_1 = \{3\}$, $A_2 = \{1, 2, 5, 6\}$ and $A_3 = \{4\}$ partition A.

Let $\mathcal{R}$ be an equivalence relation on S. For any $x \in S$, the subset

$$[x] = \{y \in S \mid x \sim y\} \subseteq S$$

is the **equivalence class** containing x.

The definition says that $[x] = [y]$ if only if $x \sim y$.

Exercise 1.3 Prove that the equivalence classes of $\mathcal{R}$ form a partition of S.

Example 1.26 Let $\mathcal{R}$ be congruence modulo 10 on $\mathbb{Z}$. The equivalence classes are:

$$\begin{aligned}
[0] &= \{0, \pm 10, \pm 20, \ldots\} &&= \{n \mid n \equiv 0 \mod 10\} \\
[1] &= \{1, 11, 21, \ldots, -9, -19, \ldots\} &&= \{n \mid n \equiv 1 \mod 10\} \\
[2] &= \{2, 12, 22, \ldots, -8, -18, \ldots\} &&= \{n \mid n \equiv 2 \mod 10\} \\
&\vdots \\
[9] &= \{9, 19, 29, \ldots, -1, -11, \ldots\} &&= \{n \mid n \equiv 9 \mod 10\} \\
[10] &= [0].
\end{aligned}$$

There are exactly 10 equivalence classes, $[0], [1], \ldots, [9]$.

Example 1.27 Let W denote the set of words in the Oxford Dictionary (or any other dictionary). The relation

$$\mathcal{R} = \{(w_1, w_2) \mid \text{words } w_1 \text{ and } w_2 \text{ begin with the same letter}\} \subseteq W \times W$$

is an equivalence relation on W. The equivalence classes consist of all words starting with the same letter. There are of course 26 classes.

Example 1.28 Let $S = \{0, 1, 2, 3, \ldots, 10\}$ and $\mathcal{R}$ the equivalence relation

$$\mathcal{R} = \{(x, y) \mid x \equiv y \mod 3\} \subseteq S \times S.$$

The equivalence classes are

$$[0] = \{0, 3, 6, 9\},\ [1] = \{1, 4, 7, 10\},\ [2] = \{2, 5, 8\}.$$

If S is a set with an equivalence relation $\mathcal{R}$ on it, the set of its equivalence classes is denoted by

$$S/\sim = \{[x] \mid x \in S\}.$$

Studying the set of equivalence classes $S/\sim$ of S rather than the elements of S itself can be used to discard irrelevant information.

1.6 Functions

Given two sets S and T, a **function** f from S to T assigns a unique element $f(s) \in T$ to each element $s \in S$. We write

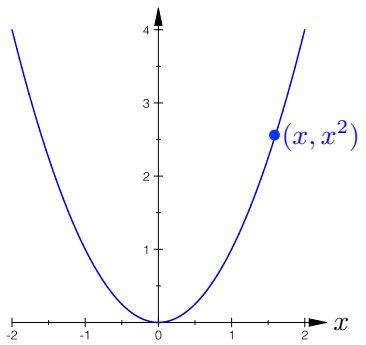
$$f : S \to T, \quad s \mapsto f(s).$$

The set S is the **domain** of the function f.

Example 1.29 Consider the function with domain $S = \mathbb{R}$, $T = \mathbb{R}_{\geq 0}$ and

$$f : \mathbb{R} \to \mathbb{R}_{\geq 0}, \quad x \mapsto x^2.$$

This is the familiar parabola function $f(x) = x^2$.



An very common type of functions are those defined on the non-negative integers: A **sequence** is a function with domain $\mathbb{N} = \{0, 1, 2, \ldots\}$ or $\{1, 2, \ldots\}$. For sequences, it is customary to write f_n instead $f(n)$.

Example 1.30 Some sequences:

(a) The **factorial** of $n \in \mathbb{N}$ is defined as

$$0! = 1 \quad \text{and} \quad n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n \text{ for } n > 0.$$

The function $f : \mathbb{N} \to \mathbb{N}$, $n \mapsto n!$ is a sequence.

$$f_0 = 1, \quad f_1 = 1, \quad f_2 = 2, \quad f_3 = 6, \quad f_4 = 24, \ldots$$

(b) The function $p : \mathbb{N} \to \mathbb{N}$, $n \mapsto 2^n$ is a sequence,

$$p_0 = 1, \quad p_1 = 2, \quad p_2 = 4, \quad p_3 = 8, \ldots$$

(c) The function $f : \mathbb{N}\backslash\{0\} \to \mathbb{Q}$, $n \mapsto \frac{1}{n}$ is a sequence.

$$f_1 = 1, \quad f_2 = \frac{1}{2}, \quad f_3 = \frac{1}{3}, \ldots$$

Suppose we have two function $f : S \to T$ and $g : T \to U$. Then we can **compose** f and g to obtain a new function from S to U, denoted by $g \circ f$:

$$g \circ f : S \to U, \quad x \mapsto g(f(x)).$$

Example 1.31 Let $f : \mathbb{N}\backslash\{0\} \to \mathbb{R}$, $n \mapsto \frac{1}{n}$ and $g : \mathbb{R} \to \mathbb{R}$, $x \mapsto x^2$. The composition $g \circ f$ of f and g is

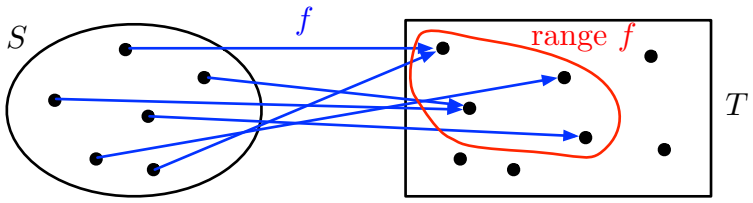
$$g \circ f : \mathbb{N} \to \mathbb{R}, \quad x \mapsto \frac{1}{n^2}.$$

1.7 Properties of functions

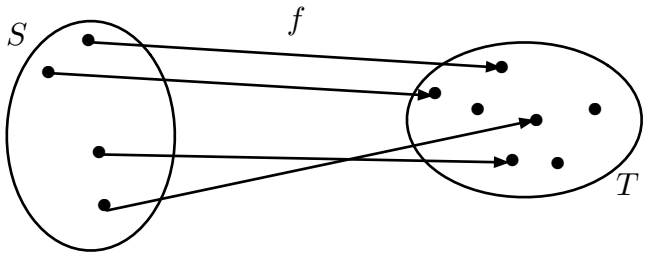
Let $f : S \to T$ be a function with domain S. The **range** of f is the set

$$\text{range } f = f(S) = \{t \in T \mid \text{there exists } s \in S \text{ such that } f(s) = t\} \subseteq T.$$

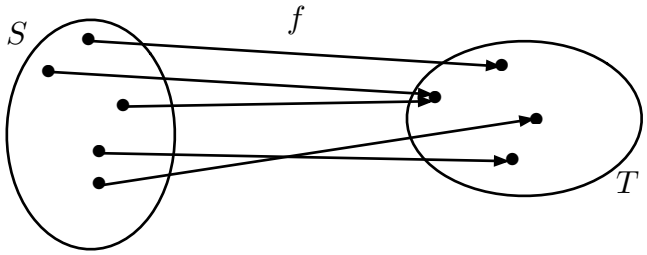
Sometimes the range is also called the **image** of f. In general, the range is not the whole set T.



We say that a function f is **one-to-one** (or **injective**) if $f(s_1) = f(s_2)$ implies $s_1 = s_2$ (or equivalently: if $s_1 \neq s_2$, then $f(s_1) \neq f(s_2)$).



We say that f is **onto** (or **surjective**) T if for every $t \in T$, there exists some $s \in S$ such that $f(s) = t$. In other words, range $f = T$.



Example 1.32 Onto and one-to-one functions.

(a) The function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$, $x \mapsto x^2$ is onto: Every positive real number y is the image of its square root $x = \sqrt{y}$ under f,

$$f(\sqrt{y}) = \sqrt{y}^2 = y.$$

However, if we choose all of $\mathbb{R}$ to be the set of values T for f, then f is not onto: For $x \in \mathbb{R}$, the values x^2 are all ≥ 0, so no negative real number can appear in the range of f.

f is not one-to-one: $f(-x) = (-x)^2 = x^2 = f(x)$, and $-x \neq x$ if $x \neq 0$.

(b) The factorial function $f : \mathbb{N} \to \mathbb{N}$, $n \mapsto n!$, is neither one-to-one nor onto: $f(0) = 1 = f(1)$, so f is not one-to-one. For $n > 1$, all factorials $n! = 1 \cdot 2 \cdots n$ are even since they contain a factor 2. So no odd number other than 1 is contained in the range of f.

(c) The function $p : \mathbb{N} \to \mathbb{N}$, $n \mapsto 2^n$ is one-to-one: For two distinct integers $n \neq k$, the numbers 2^n and 2^k are distinct, that is, $p(n) \neq p(k)$.

The function p is not onto: No odd number is contained in the range of p.

(d) For a real number x, the **floor function** $\lfloor x \rfloor$ denotes the greates integer n less than or equal to x. So $\lfloor\ \rfloor : \mathbb{R} \to \mathbb{Z}$. The floor function is not one-to-one. For example, the numbers 1 and $1 + \frac{1}{10}$ both satisfy $\lfloor 1 \rfloor = 1 = \lfloor 1 + \frac{1}{10} \rfloor$. Floor is onto: For every $n \in \mathbb{Z}$, the same number $n \in \mathbb{R}$ is mapped to n, $\lfloor n \rfloor = n$.

Theorem 1.33 *Suppose S and T are finite sets.*

(a) *If $f : S \to T$ is one-to-one, then $|S| \leq |T|$.*

(b) *If $f : S \to T$ is onto, then $|S| \geq |T|$.*

(c) *If $f : S \to T$ is one-to-one and onto, then $|S| = |T|$.*

PROOF: Let $|S| = n$, so that $S = \{s_1, \ldots, s_n\}$. Set $t_1 = f(s_1), \ldots, t_n = f(s_n)$.

(a) If f is one-to-one, the elements $t_1, \ldots, t_n$ are all distinct. Hence T contains at least n elements.

(b) Because f is onto, the $t_1, \ldots, t_n$ encompass all the elements of T. Hence T contains at most n elements.

(c) By part (a) and (b), $n \leq |T| \leq n$. So $|T| = n$ follows. ∎

Example 1.34 Let S be a finite set with power set $\mathcal{P}(S)$, and consider the function $c : \mathcal{P}(S) \to \mathcal{P}(S)$, $c(A) \mapsto A^c$ (every subset $A \subseteq S$ is mapped to its complement in S).

(a) c is one-to-one: if $A \neq B$, then $A^c \neq B^c$.

(b) c is onto $\mathcal{P}(S)$: for any set $A \subseteq S$,

$$c(A^c) = (A^c)^c = A$$

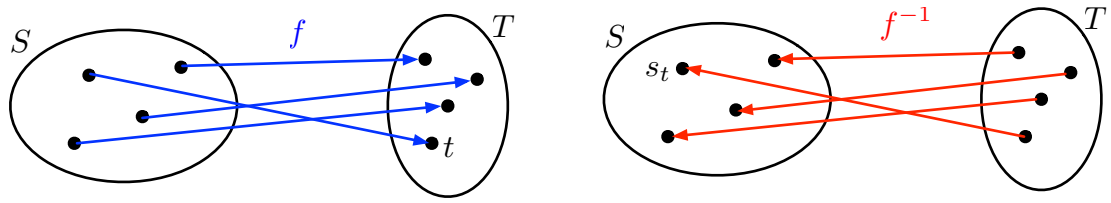
so every element A of $\mathcal{P}(S)$ is in the range of c.

1.8 Inverse functions and preimages

If a function $f : S \to T$ is one-to-one and onto, then we can define its **inverse function**

$$f^{-1} : T \to S, \quad t \mapsto s_t$$

where s_t is the unique element in S that is mapped to t by the function f (this element exists, because f is onto, and it is unique, because f is one-to-one).



Note: Do not confuse f^{-1} with $\frac{1}{f}$! For example, if $f : \mathbb{R} \to \mathbb{R}$, $x \mapsto x^3$, then $f^{-1}(x) = \sqrt[3]{x}$, but $\frac{1}{f(x)} = \frac{1}{x^3}$. These are two completely different functions.

Exercise 1.4 Assume that $f : S \to T$ has an inverse function f^{-1}. Show that $f^{-1} \circ f = \mathrm{id}_S$, where id_S denotes the identity function on S (that is, $\mathrm{id}_S(x) = x$ for all $x \in S$).

Example 1.35 Inverse functions.

(a) Recall from Example 1.32 that the function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$, $x \mapsto x^2$, is onto, but not one-to-one. So does f does not have an inverse function.

If we restrict the domain of f to $\mathbb{R}_{\geq 0}$, then $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is one-to-one and onto. The inverse function is given by

$$f^{-1} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, \quad x \mapsto \sqrt{x}.$$

(b) The function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$, $x \mapsto 2^x$, has an inverse, $f^{-1}(x) = \log_2(x)$.

(c) The complement function $c : \mathcal{P}(S) \to \mathcal{P}(S)$ from Example 1.34 is onto and one-to-one. Its inverse $c^{-1} : \mathcal{P}(S) \to \mathcal{P}(S)$ maps A^{c} to A. But $A = (A^{\mathrm{c}})^{\mathrm{c}}$, so c^{-1} maps any subset to its complement. So we find

$$c = c^{-1}.$$

If a function $f : S \to T$ is one-to-one, but not onto, then we can replace T by $\operatorname{range} f$ to obtain an inverse function. The function $f : S \to \operatorname{range} f$ is onto by the very definition of the range, so $f^{-1} : \operatorname{range} f \to S$ exists.

If a function $f : S \to T$ is *not* one-to-one, we cannot define an inverse function. But we use the notation $f^{-1}(y)$ to denote the set of those elements in S that f maps to y. More generally, for a subset $Y \subseteq T$, we write

$$f^{-1}(Y) = \{s \in S \mid f(s) \in Y\} \subseteq S.$$

This is called the **preimage of Y under the function f**. Technically, f^{-1} is a function $f^{-1} : \mathcal{P}(T) \to \mathcal{P}(S)$.

1.9 Addendum: Relational data bases

The relations we encountered in this chapter provide the theoretical foundations for one of the most common type of data management structures, the **relational databases**. Such a relational database is a collection of tables holding data, and each table can be formally defined as a relation $\mathcal{R} \subseteq D_1 \times \cdots \times D_n$, where $D_1, \ldots, D_n$ are the sets the table's entries assume their values in.

Suppose, for example, we want to establish a database to manage student data. This database comprises several different tables of data, say

$\mathcal{P}$ = Personal

Student ID	Name	Address	Semester
8391	Donald Knuth	Stanford avenue	3
6091	Grace Hopper	Harvard street	6
4191	Dennis Richie	Bell drive	5
9217	Dennis McArthy	Mit highway	5
5918	Shafi Goldwasser	Carnegie road	2

$\mathcal{C}$ = Courses

Subject	Lecture Time
Compiler Construction	Fri 16:00
Algorithms	Mo 8:00
Mathematics	Tue 10:00
Software Engineering	Tue 10:00

$\mathcal{A}$ = Allocation

Student ID	Subject	Grade
8391	Algorithms	A
6091	Compiler Construction	A
4191	Compiler Construction	B
9217	Compiler Construction	A
5918	Algorithms	B

The labels in the top row of a table are called the table's **attributes**. The attributes serve to identify the data sets $D_1, \ldots, D_n$.

Here, the table $\mathcal{P}$ is a relation in $I \times N \times A \times S$, where

$$\begin{aligned} I &= \text{set of student id numbers} \\ N &= \text{set of names} \\ A &= \text{set of addresses} \\ S &= \text{set positive integers.} \end{aligned}$$

Its attributes are a_1 = "Student ID", a_2 = "Name", a_3 = "Address" and a_4 = "Semester".

The purpose of a database is to allow **queries** to its tables in order to retrieve information and extract new information by relating the data in different tables. These queries can be modelled by set-theoretic operations on the relations (the tables) in the database.

The usual set operations $\cup$, $\cap$ and $\setminus$ can be applied to two relations $\mathcal{R}_1$ and $\mathcal{R}_2$, given that $\mathcal{R}_1$ and $\mathcal{R}_2$ are both relations in the same set, $\mathcal{R}_1, \mathcal{R}_2 \subseteq D_1 \times \cdots \times D_n$:

$$\begin{aligned} \mathcal{R}_1 \cup \mathcal{R}_2 &= \{d \mid d \in \mathcal{R}_1 \text{ or } \mathcal{R}_2\} \\ \mathcal{R}_1 \cap \mathcal{R}_2 &= \{d \mid d \in \mathcal{R}_1 \text{ and } \mathcal{R}_2\} \\ \mathcal{R}_1 \setminus \mathcal{R}_2 &= \{d \mid d \in \mathcal{R}_1 \text{ and } d \notin \mathcal{R}_2\}. \end{aligned}$$

Here, d is short for a tuple $d = (d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$.

Example 1.36 Given the tables

$\mathcal{P}_1$ =

Student ID	Name
8391	Donald Knuth
6091	Grace Hopper
4191	Dennis Richie

, $\mathcal{P}_2$ =

Student ID	Name
4191	Dennis Richie
9217	Dennis McArthy
5918	Shafi Goldwasser

Then:

$\mathcal{P}_1 \cup \mathcal{P}_2 =$

Student ID	Name
8391	Donald Knuth
6091	Grace Hopper
4191	Dennis Richie
9217	Dennis McArthy
5918	Shafi Goldwasser

, $\mathcal{P}_1 \backslash \mathcal{P}_2 =$

Student ID	Name
8391	Donald Knuth
6091	Grace Hopper

Some other set operations are defined specifically for queries in relational databases (for simplicity, we treat tables with identical attributes in different order as one and the same):

Given a relation $\mathcal{R} \subseteq D_1 \times \cdots \times D_n$ with attributes $a_1, \ldots, a_n$. The **projection** from $\mathcal{R}$ to a subset $\{a_{i_1}, \ldots, a_{i_k}\}$ of its attributes with $k \leq n$ is defined as

$$P_{a_{i_1},\ldots,a_{i_k}}(\mathcal{R}) = \{(e_1, \ldots, e_k) \mid \text{ there exists } (d_1, \ldots, d_n) \in \mathcal{R} \text{ with } d_{i_1} = e_1, \ldots, d_{i_k} = e_k\}.$$

Example 1.37 Consider the "Personal" table $\mathcal{P}$ and let $i_1 = 2, i_3 = 4$. The projection $P_{a_2,a_4}(\mathcal{P})$ is the table

$P_{\text{Name,Semester}}(\mathcal{P}) =$

Name	Semester
Donald Knuth	3
Grace Hopper	6
Dennis Richie	5
Dennis McArthy	5
Shafi Goldwasser	2

The **product** of two relations $\mathcal{R}_1$, $\mathcal{R}_2$ is defined in a similar way to the Cartesian product for sets:

$$\mathcal{R}_1 \times \mathcal{R}_2 = \{(d_1, \ldots, d_n, d_{n+1}, \ldots, d_m\} \mid (d_1, \ldots, d_n) \in \mathcal{R}_1, (d_{n+1}, \ldots, d_m) \in \mathcal{R}_2\}.$$

This means we combine every entry in the table $\mathcal{R}_1$ with every entry in the table $\mathcal{R}_2$.

Example 1.38 The product of the "Course" relation $\mathcal{C}$ and the "Allocation" relation $\mathcal{A}$ is

$$\mathcal{C} \times \mathcal{A} =$$

Subject	Lecture Time	Student ID	Subject	Grade
Compiler Construction	Fri 16:00	8391	Algorithms	A
Compiler Construction	Fri 16:00	6091	Compiler Construction	A
Compiler Construction	Fri 16:00	4191	Compiler Construction	B
Compiler Construction	Fri 16:00	9217	Compiler Construction	A
Compiler Construction	Fri 16:00	5918	Algorithms	B
Algorithms	Mo 8:00	8391	Algorithms	A
Algorithms	Mo 8:00	6091	Compiler Construction	A
Algorithms	Mo 8:00	4191	Compiler Construction	B
Algorithms	Mo 8:00	9217	Compiler Construction	A
Algorithms	Mo 8:00	5918	Algorithms	B
Mathematics	Tue 10:00	8391	Algorithms	A
Mathematics	Tue 10:00	6091	Compiler Construction	A
Mathematics	Tue 10:00	4191	Compiler Construction	B
Mathematics	Tue 10:00	9217	Compiler Construction	A
Mathematics	Tue 10:00	5918	Algorithms	B
Software Engineering	Tue 10:00	8391	Algorithms	A
Software Engineering	Tue 10:00	6091	Compiler Construction	A
Software Engineering	Tue 10:00	4191	Compiler Construction	B
Software Engineering	Tue 10:00	9217	Compiler Construction	A
Software Engineering	Tue 10:00	5918	Algorithms	B

Suppose $n \geq k$, and let $\mathcal{R} \subseteq D_1 \times \cdots \times D_n$ be a relation with attributes $a_1, \ldots, a_n$. Given a second relation $\mathcal{Q} \subseteq D_{i_1} \times \cdots \times D_{i_k}$, we define the **selection**

$$\mathrm{sel}_{\mathcal{Q}}(\mathcal{R}) = \{d \in \mathcal{R} \mid P_{a_{i_1},\ldots,a_{i_k}}(d) \in \mathcal{Q}\}.$$

This means $\mathrm{sel}_{\mathcal{Q}}(\mathcal{R})$ contains only those entries d from $\mathcal{R}$ whose projection on the attributes $a_{i_1}, \ldots, a_{i_k}$ also occurs in $\mathcal{Q}$.

Example 1.39 Consider the "Courses" and "Allocation" tables $\mathcal{C}$ and $\mathcal{A}$. We want to obtain a list of those courses in $\mathcal{C}$ which are currently taken by some student. First, we find out which courses are taken by the students:

$\mathcal{Q} = P_{\text{Subject}}(\mathcal{A}) =$

Subject
Algorithms
Compiler Construction
Compiler Construction
Compiler Construction
Algorithms

Then use the new relation $\mathcal{Q}$ to select the course in the relation $\mathcal{C}$:

$\mathrm{sel}_{\mathcal{Q}}(\mathcal{C}) =$

Subject	Lecture Time
Compiler Construction	Fri 16:00
Algorithms	Mo 8:00

Exercise 1.5 Let $m, k \leq n$. Given three relations $\mathcal{R} \subseteq D_1 \times \cdots \times D_n$, $\mathcal{Q} \subseteq D_1 \times \cdots \times D_k$ and $\mathcal{S} \subseteq D_1 \times \cdots \times D_m$, prove that $\mathrm{sel}_{\mathcal{S}}(\mathrm{sel}_{\mathcal{Q}}(\mathcal{R})) = \mathrm{sel}_{\mathcal{Q}}(\mathrm{sel}_{\mathcal{S}}(\mathcal{R}))$.

Instead of using a relation $\mathcal{Q}$ to define the selection, one can define by some logical condition on the attributes of $\mathcal{R}$. If a_1, a_2 are attributes of $\mathcal{R}$, then

$$\mathrm{sel}_{a_1=a_2}(\mathcal{R}) = \{d \in \mathcal{R} \mid d_1 = d_2\}$$

selects those entries in the table $\mathcal{R}$ whose values for the attributes a_1 and a_2 coincide. More generally, one can define $\mathrm{sel}_{a_1 \Theta a_2}$, where Θ is some comparison operator for the attributes a_1 and a_2. For example, Θ can be $=$, $<$, $\leq$, etc.

The most important operations on database tables are the **joins**. Suppose we have two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ with their respective sets of attributes A_1 and A_2. Let $A = \{a_1, \ldots, a_k\}$ be another set of attributes such that $A \subseteq A_1$ and $A \subseteq A_2$. Set $B_1 = A_1 \backslash A$, $B_2 = A_2 \backslash A$. The **natural join** of $\mathcal{R}_1$ and $\mathcal{R}_2$ is

$$\mathcal{R}_1 \bowtie_A \mathcal{R}_2 = \left\{d \in P_{B_1 \cup A \cup B_2}(\mathcal{R}_1 \times \mathcal{R}_2) \mid P_{A_i}(d) \in \mathcal{R}_i \text{ and } P_A(P_{A_1}(d)) = P_A(P_{A_2}(d))\right\}.$$

So the operation $\bowtie_A$ selects and combines those entries in $\mathcal{R}_1$ and $\mathcal{R}_2$ that coincide in the attributes $a_1, \ldots, a_k$. The projection $P_{B_1 \cup A \cup B_2}$ in the definition takes care that no attribute appears twice in the resulting table.

Example 1.40 Let $A = \{\text{Subject}\}$. The natural join $\mathcal{A} \bowtie_A \mathcal{C}$ of the tables $\mathcal{A}$ and $\mathcal{C}$ is

$\mathcal{A} \bowtie_{\text{Subject}} \mathcal{C} =$

Student ID	Grade	Subject	Lecture Time
8391	A	Algorithms	Mo 8:00
6091	A	Compiler Construction	Fri 16:00
4191	B	Compiler Construction	Fri 16:00
9217	A	Compiler Construction	Fri 16:00
5918	B	Algorithms	Mo 8:00

Exercise 1.6 If $A = \varnothing$, then $\mathcal{R}_1 \bowtie_\varnothing \mathcal{R}_2 = \mathcal{R}_1 \times \mathcal{R}_2$.

The last operation we study is the **Θ-join** of two relations $\mathcal{R}_1$ and $\mathcal{R}_2$. Let a_1 be an attribute of $\mathcal{R}_1$ and a_2 be an attribute of $\mathcal{R}_2$, such that a_1 and a_2 take values in a common set D, on which a binary comparison operator Θ is defined. Then the Θ-join of $\mathcal{R}_1$ and $\mathcal{R}_2$ is

$$\mathcal{R}_1[a_1 \Theta a_2]\mathcal{R}_2 = \mathrm{sel}_{a_1 \Theta a_2}(\mathcal{R}_1 \times \mathcal{R}_2).$$

In practice, the commands of query languages for relational databases (such as SQL) are loosely based on the relational operations introduced here. For technical reasons, it can be preferable to deviate from the formal definition or add additional functionality.

2 Formal logic

In logic we study propositions and the laws governing the relationships between the truth and falsity of different propositions. In *formal logic* we are concerned with the syntactic structure of propositions (meaning the way they are composed of more basic propositions) and which conclusions can be drawn from it. Because these conclusions are valid for all propositions of the same syntactic form, they are completely independent of the particular meaning of the propositions involved, but depend solely on their truth value.

2.1 Propositional calculus

A **proposition** is a statement which is either *true* or *false*, but not both. We use sans-serif font and quotation marks "..." to express statements and lower case letters $p, q, r, \ldots$ to denote proposition.

Example 2.1 Some proposition:

(a) "3 is an integer"

(b) "Adelaide is the capital of Australia"

(c) "$x + 3 = 10$ for all $x \in \mathbb{R}$"

(d) "There is life outside the solar system"

(e) "$2^{9586125} - 1$ is a prime number"

(f) "It is raining"

Note: For a statement p to be a valid proposition it is not required that we actually know whether p is true or not. But it has to make sense to say "p is true" or "p is false".

Example 2.2 Some statements that are not proposition:

(a) "Go home"

(b) "How old are you?"

(c) "It is colder at night than outside"

(d) "Logic is faster than light"

Exercise 2.1 Explain why (d) and (e) in Example 2.2 are not propositions.

Exercise 2.2 Is the statement "This statement is false" a proposition?

To each proposition p we can assign a **truth value** $\mathrm{val}(p)$, which is either

$$\mathsf{T}\ (\text{true}) \quad \text{or} \quad \mathsf{F}\ (\text{false}).$$

Example 2.3 Truth values:

(a) $p =$ "3 is prime", $\mathrm{val}(p) = \mathsf{T}$

(b) $q =$ "6 is prime", $\mathrm{val}(q) = \mathsf{F}$

(c) $r =$ "6 is not prime", $\mathrm{val}(r) = \mathsf{T}$

If p, q are a proposition and $f(p,q)$ is some other proposition depending on (the truth values of) p and q, then the **truth table** for f is

p	q	$f(p,q)$
T	T	$f(\mathsf{T},\mathsf{T})$
F	T	$f(\mathsf{F},\mathsf{T})$
T	F	$f(\mathsf{T},\mathsf{F})$
F	F	$f(\mathsf{F},\mathsf{F})$

which is to be read as "if p is true and q is true, then $f(p,q)$ has the truth value $f(\mathsf{T},\mathsf{T})$" etc.

Propositions can be combined using **connectives**:

$\neg$	not
$\wedge$	and
$\vee$	or
$\rightarrow$	implication (if ... then ...)
$\leftrightarrow$	equivalence (if and only if)

A precise definition of these connectives is given by the following truth tables:

p	$\neg p$
T	F
F	T

p	q	$p \wedge q$
T	T	T
F	T	F
T	F	F
F	F	F

p	q	$p \vee q$
T	T	T
F	T	T
T	F	T
F	F	F

p	q	$p \rightarrow q$
T	T	T
F	T	T
T	F	F
F	F	T

p	q	$p \leftrightarrow q$
T	T	T
F	T	F
T	F	F
F	F	T

Example 2.4 $\neg p$ ("not p") stands for the negation of a proposition.

The negation of the true proposition

$$p = \text{"3 is a prime number"}$$

is the false proposition

$$\neg p = \text{"3 is not a prime number"}.$$

Example 2.5 $p \wedge q$ ("p and q") is true only if both components are true.

(a) For

$$p = \text{"3 is a prime number"}$$
$$q = \text{"6 is a prime number"},$$

the composed proposition is

$$p \wedge q = \text{"3 is prime and 6 is prime"}.$$

Even though p is true, q is false, so the composed proposition $p \wedge q$ is false.

(b) The two propositions

$$p = \text{"the world is not flat"}$$
$$q = \text{"2 + 2 = 4"}$$

are both true, so the composed proposition is true:

$$p \wedge q = \text{“the world is not flat and } 2+2=4\text{”}.$$

Whereas their negations,

$$\neg p = \text{“the world is flat”}$$
$$\neg q = \text{“}2+2 \neq 4\text{”},$$

are both false. So

$$\neg p \wedge \neg q = \text{“the world is flat and } 2+2 \neq 4\text{”}$$

is false.

Example 2.6 $p \vee q$ (“p or q”) is true if at least one component is true.

(a) Let p, q be as in Example 2.5 (a). The proposition

$$p \vee q = \text{“}3 \text{ is prime and } 6 \text{ is prime”}$$

is true, since at least one of components is true (in this case p).

(b) Note that this “or” is an inclusive or, meaning a statement is true if one or both of its components is true. Consider

$$p \vee \neg q = \text{“}3 \text{ is prime and } 6 \text{ is not prime”}.$$

The propositions p and $\neq q$ are both true, and so is the statement $p \vee \neg q$.

(c) In fact, $p \vee q$ is false only when both p and q are false:

$$\text{“the world is flat or } 2+2 \neq 4\text{”}$$

is false, since both component statements are false.

Example 2.7 $p \to q$ (“if p, then q”) means the truth of p implies the truth of q (so $p \to q$ is false only when p is true whilst q is false).

(a) Let p, q be as in Example 2.5 (a). Then

$$p \to q = \text{“if } 3 \text{ is prime, then } 6 \text{ is prime”}$$

is false. After all, q is a false statement, so it cannot be implied by any other statement.

(b) Consider the propositions

$$p = \text{“It is raining”}$$
$$q = \text{“The ground is wet”}.$$

Look at the truth table for this particular example:

p = “It is raining”	q = “The ground is wet”	$p \to q$
“It is raining”	“The ground is wet”	T
“It is not raining”	“The ground is wet”	T
“It is raining”	“The ground is not wet”	F
“It is not raining”	“The ground is not wet”	T

Note in particular the plausibility of the second and third row of this table: In the second row, $p \to q$ is true, since the ground can be wet for reasons other than rain. However, since the ground is always wet when it rains, $p \to q$ must be false in the third row.

Note: One can be tempted to think that $p \to q$ implies a causal relationship between p and q (“p causes q to happen”), but this is not what formal logic is about. Our definition of $p \to q$ allows statements we would not make in an everyday conversation, where the respective meanings of two statements p and q is usually related by the context of the conversation. However, in formal logic we abstract the symbols from any context and are only concerned with the structure of the statements, not with their meaning. So we must allow for all possible propositions p and q (whether “related” or not).

Example 2.8 Consider

$$p = \text{“the world is flat”}$$
$$q = \text{“6 is prime”},$$

and

$$p \to q = \text{“If the world is flat, then 6 is prime”}$$

$p \to q$ is a true statement (refer to the truth table defining $\to$).

Example 2.9 $p \leftrightarrow q$ (“p if and only if q”) means the truth or falsity of p is equivalent to that of q. This agrees with our understanding of “if and only if”, namely $p \leftrightarrow q$ is true if either both p and q are true or both are false, and $p \leftrightarrow q$ is false if one of p, q is true and the other false.

(a) Let p, q be as in Example 2.5 (a).

$$p \leftrightarrow q = \text{“3 is prime if and only if 6 is prime”}$$

Since p is true and q is false, the two propositions cannot be equivalent. So $p \leftrightarrow q$ is false.

(b) Let

$$p = \text{“You earn a degree”}$$
$$q = \text{“You pass all of your exams”}.$$

Then

$$p \leftrightarrow q = \text{“You earn a degree if and only if you pass all of your exams”}$$

is true.

We combine propositions with connectives $(\neg, \wedge, \vee, \rightarrow, \leftrightarrow)$ to obtain **compound propositions**. A proposition that cannot be written as a compound proposition is called an **atom**.

Example 2.10 Compound propositions:

$$(p \wedge q) \rightarrow \neg q$$
$$(p \rightarrow q) \wedge (q \rightarrow p)$$
$$\big((p \rightarrow q) \wedge (q \rightarrow r)\big) \rightarrow (p \rightarrow r)$$

Example 2.11 A proposition such as “It is raining” is an atom, because it does not contain any connectives and cannot be further decomposed into simpler propositions. On the other hand, “It is raining and the ground is wet” is not an atom, because it contains the two propositions “It is raining” and “The ground is wet”, connected by “and”.

To find the truth value of a compound proposition p, we use the truth tables above for a “bottom up” evaluation. This means we start evaluating the component propositions of p beginning with the atoms, and then iteratively evaluate the components in order of the priority of the connective, which is: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

Example 2.12 Consider the compound statement with atoms p and q

$$r = (p \wedge q) \to \neg q.$$

We use the truth tables for $\neg$, $\to$ and $\wedge$.

$$\begin{aligned}
\mathrm{val}(r) = \mathrm{val}\big((p \wedge q) \to \neg q\big) &= \begin{cases} \mathsf{F} & \text{if } \mathrm{val}(p \wedge q) = \mathsf{T} \text{ and } \mathrm{val}(\neg q) = \mathsf{F} \\ \mathsf{T} & \text{else} \end{cases} \\
&= \begin{cases} \mathsf{F} & \text{if } \mathrm{val}(p) = \mathsf{T} \text{ and } \mathrm{val}(q) = \mathsf{T} \text{ and } \mathrm{val}(q) = \mathsf{T} \\ \mathsf{T} & \text{else} \end{cases} \\
&= \begin{cases} \mathsf{F} & \text{if } \mathrm{val}(p) = \mathsf{T} \text{ and } \mathrm{val}(q) = \mathsf{T} \\ \mathsf{T} & \text{else} \end{cases}
\end{aligned}$$

So the truth table for r is

p	q	$(p \wedge q) \to \neg q$
T	T	F
F	T	T
T	F	T
F	F	T

Example 2.13 Consider the compound statement with atoms p and q:

$$r = (p \to q) \wedge (q \to p).$$

The atoms are p and q. We use the truth tables for $\wedge$ and $\to$ to determine the tables for $p \to q$ and $q \to p$ first, and from these the table for r:

p	q	$p \to q$	$q \to p$	$(p \to q) \wedge (q \to p)$
T	T	T	T	T
F	T	T	F	F
T	F	F	T	F
F	F	T	T	T

Note that this is precisely the truth table for $p \leftrightarrow q$. This corresponds to our intuitive understanding that "p is equivalent to q" means the same as "p implies q and q implies p".

Example 2.14 Find the truth table for

$$s = \big((p \to q) \wedge (q \to r)\big) \to (p \to r).$$

The atoms are p, q, r. We use the truth tables for $\to$ and $\wedge$ to determine the tables for $p \to q$, $q \to r$, $p \to r$, then for $(p \to q) \to (q \to r)$, and finally for s:

p	q	r	$p \to q$	$q \to r$	$p \to r$	$(p \to q) \wedge (q \to r)$	$\big((p \to q) \wedge (q \to r)\big) \to (p \to r)$
T	T	T	T	T	T	T	T
F	T	T	T	T	T	T	T
T	F	T	F	T	T	F	T
F	F	T	T	T	T	T	T
T	T	F	T	F	F	F	T
F	T	F	T	F	T	F	T
T	F	F	F	T	F	F	T
F	F	F	T	T	T	T	T

Note the peculiar fact that in Example 2.14, the compound propostion s always has truth value T, irrespective of the truth values of p, q and r. Propositions such as this one have a name: A **tautology** is a proposition which always takes the truth value T (here, "always" means for all possible truth values of the atoms contained in p). On the other hand, a **contradiction** is a proposition which always takes the value false F.

Example 2.15 Tautologies:

(a) $p \to p$

p	$p \to p$
T	T
F	T

(b) $p \leftrightarrow \neg\neg p$

p	$\neg p$	$\neg\neg p$	$p \leftrightarrow \neg\neg p$
T	F	T	T
F	T	F	T

Example 2.16 The proposition $p \wedge \neg p$ is a contradiction:

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

A compound statement can be a tautology, a contradiction or neither.

Theorem 2.17 *p is a tautology if and only if $\neg p$ is a contradiction.*

PROOF: We always have $\mathrm{val}(p) \neq \mathrm{val}(\neg p)$. Because $\mathrm{val}(p) = \mathsf{T}$ for all possible truth values of the atoms it is composed of, it follows that $\mathrm{val}(\neg p) = \mathsf{F}$ for all possible truth values of the atoms it is composed of. The converse is proved in the same way. ■

If p and q are propositions and $p \leftrightarrow q$ is a tautology, then we say that p and q are **logically equivalent** and we write $p \Leftrightarrow q$.

Example 2.18 In Example 2.15 (b) we saw that for p and $\neg\neg p$ are logically equivalent for any proposition p.

Note: Do not confuse the meaning of the symbols $\leftrightarrow$ and $\Leftrightarrow$. The symbol $\leftrightarrow$ is part of our propositional calculus, whereas $\Leftrightarrow$ is not. The latter is used to make statements *about* propositions, and it is a shorthand notation for "propositon p and proposition $\neg\neg p$ are logically equivalent".

Example 2.19 Determine logical equivalence with the help of truth tables:

(a) $p \leftrightarrow q$ is logically equivalent to $(p \to q) \wedge (q \to p)$: Using Example 2.13, we determine the truth tables for the two propositions:

p	q	$p \leftrightarrow q$	$(p \to q) \wedge (q \to p)$
T	T	T	T
F	T	F	F
T	F	F	F
F	F	T	T

We see that $p \leftrightarrow q$ is T (or F, respectively) whenever $(p \to q) \wedge (q \to p)$ is. This means

$$(p \leftrightarrow q) \leftrightarrow \big((p \to q) \wedge (q \to p)\big)$$

is a tautology.

(b) $p \to q$ is logically equivalent to $\neg q \to \neg p$.

p	q	$\neg p$	$\neg q$	$p \to q$	$\neg q \to \neg p$
T	T	F	F	T	T
F	T	T	F	T	T
T	F	F	T	F	F
F	F	T	T	T	T

So we find

$$(p \leftrightarrow q) \quad \Leftrightarrow \quad (\neg q \rightarrow \neg p).$$

$\neg q \rightarrow \neg p$ is called the **contrapositive** of $p \rightarrow q$.

Exercise 2.3 Prove that $p \wedge (p \rightarrow q)$ is logically equivalent to $p \wedge q$.

We look at some concrete instances of the above examples:

Example 2.20 The definition of a one-to-one function: "f is one-to-one if different numbers take different values (under f)" can be stated in two ways: Define the propositions p, q as

$$p = \text{“}x \neq y\text{”}, \quad q = \text{“}f(x) \neq f(y)\text{”},$$

f being one-to-one is expressed by the proposition

$$p \rightarrow q = \text{“if } x \neq y \text{ then } f(x) \neq f(y)\text{”}.$$

By Example 2.19 (b), we can also state this as the contrapositive

$$\begin{gathered} \neg q \rightarrow \neg p \\ \neg(f(x) \neq f(y)) \rightarrow \neg(x \neq y) \\ f(x) = f(y) \rightarrow x = y. \end{gathered}$$

So

$$\begin{gathered} p \rightarrow q = \text{“if } x \neq y \text{ then } f(x) \neq f(y)\text{”} \\ \Leftrightarrow \\ \neg q \rightarrow \neg p = \text{“if } f(x) = f(y) \text{ then } x = y\text{”}. \end{gathered}$$

The contrapositive is sometimes easier to show when trying to prove functions are one-to-one.

Example 2.21 A well known theorem in Calculus is

$$\text{“If } f \text{ is differentiable then } f \text{ is continuous”}$$

The contrapositive, which is equivalent, is

$$\text{“If } f \text{ is not continuous then } f \text{ is not differentiable”}$$

This is useful when you know a function is not continuous at some point x, then you know it is not differentiable at x.

Example 2.22 A theorem from linear algebra is

$$\text{"If } \det(A) \neq 0 \text{ the } A \text{ is invertible"}$$

This is equivalent to

$$\text{"If } A \text{ is not invertible then } \det(A) = 0\text{"}.$$

Let p and q be two propositions. Suppose that $p \to q$ is a tautology. Then we say that p **logically implies** q and we write $p \Rightarrow q$.

Note: As before for the symbol $\Leftrightarrow$, note that $\Rightarrow$ is not part of any proposition, but a shorthand notation for a statement about propositions.

Example 2.23 $(p \to q) \wedge p \Rightarrow q$: Use truth tables to show that $(p \to q) \wedge p \to q$ is a tautology:

p	q	$p \to q$	$(p \to q) \wedge p$	$(p \to q) \wedge p \to q$
T	T	T	T	T
F	T	T	T	T
T	F	F	F	T
F	F	T	F	T

We can use these ideas to analyse arguments to see if they are valid.

Example 2.24 Consider

$$p = \text{"Bob reads the Advertiser"}$$
$$q = \text{"Bob is well informed"}.$$

Suppose that in some alternate reality the following holds:

$$p \to q = \text{"If Bob reads the Advertiser then Bob is well informed"}.$$

Now, someone might argue that this means if Bob does not read the Advertiser, he cannot be well informed. This amounts to claiming

$$((p \to q) \wedge \neg p) \Rightarrow \neg q,$$

or in other words, $((p \to q) \wedge \neg p) \to \neg q$ is a tautology. But checking the truth table reveals

p	q	$p \to q$	$(p \to q) \wedge \neg p$	$(p \to q) \wedge \neg p \to \neg q$
F	T	T	T	F

so the above claim is not true. This means it is possible for Bob to be well informed despite not reading the Advertiser.

Note that we do not need to write down the whole truth table, since it is enough to find one row in which $((p \to q) \wedge \neg p) \to \neg q$ does not have truth value T.

2.2 Arguments in propositional calculus

In Example 2.24 above we saw that formal logic can be used for reasoning. In this section, we want to explore how conclusions can be drawn systematically from a given set of assumptions.

An **argument** is a rule allowing us to derive the truth of a proposition c, the **conclusion**, given the truth of a certain set of propositions $p_1, \ldots, p_n$ which we call the **premises** of our argument. We use the notation

$$\begin{array}{c} \text{premise 1} \\ \vdots \\ \text{premise } n \\ \hline \therefore \text{ conclusion} \end{array} \qquad \text{or} \qquad \begin{array}{c} p_1 \\ \vdots \\ p_n \\ \hline \therefore c \end{array}$$

The symbol $\therefore$ is to read as "therefore". Such a rule is called **valid** if

$$p_1 \wedge \ldots \wedge p_n \quad \Rightarrow \quad c.$$

In other words, the argument is valid if there is no row in the truth table of this expression in which all premises have value T and the conclusion has value F.

Here, we will consider three common types of arguments. However, there are many more. The validity of these rules can be tested with truth tables.

The first rule is **direct reasoning**, also known as **modus ponens**[1)]. The premises are $p \to q$ and p. The conclusion is q. So the form of this argument is

$$\begin{array}{c} p \to q \\ p \\ \hline \therefore q \end{array}$$

[1)] Latin for "the way that affirms".

In plain English, this states "if p implies q, and if p is true, then it follows that q is true". This seems intuitively reasonable, and the validity of this rule follows from Example 2.23, where we used truth tables to show that

$$(p \to q) \wedge p \quad \Rightarrow \quad q.$$

We apply this rule to some concrete examples:

Example 2.25 Examples of direct reasoning:

(a) Let

$$p = \text{"It is raining"}$$
$$q = \text{"The ground is wet"}.$$

Then our argument by direct reasoning goes as follows:

$$\frac{\begin{array}{c}\text{"If it is raining, then the ground is wet"} \\ \text{"It is raining"}\end{array}}{\therefore \text{"The ground is wet"}}$$

(b) Let

$$p = \text{"Bob works hard"}$$
$$q = \text{"Bob passes his exam"}.$$

Direct reasoning yields:

$$\frac{\begin{array}{c}\text{"If Bob works hard, then Bob passes his exam"} \\ \text{"Bob works hard"}\end{array}}{\therefore \text{"Bob passes his exam"}}$$

The **contrapositive argument** is the rule

$$\frac{p \to q}{\therefore \neg q \to \neg p} \quad \text{or equivalently} \quad \frac{\neg q \to \neg p}{\therefore p \to q}$$

This argument is valid, since by Example 2.19 (b),

$$(p \to q) \quad \Leftrightarrow \quad (\neg q \to \neg p).$$

Example 2.26 We wish to prove

$$c = \text{“if } n^2 \text{ is an even integer, then } n \text{ is an even integer”}.$$

We prove the contrapositive assertion. Let

$$p = \text{“}n^2 \text{ is an even integer”}$$
$$q = \text{“}n \text{ is an even integer”}.$$

The contrapositive argument is then

$$\frac{\text{“if } n \text{ is an odd integer, then } n^2 \text{ is an odd integer”}}{\therefore\text{“if } n^2 \text{ is an even integer, then } n \text{ is an even integer”}}.$$

Now we need to prove that the premise is true: Suppose n is odd, so that $n = 2k + 1$ for some integer k. Then

$$\begin{aligned} n^2 &= (2k+1)^2 \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1 \end{aligned}$$

which is odd. So the premise is true, and by the contrapositive argument above, the conclusion c is true.

Example 2.27 Recall that a function $f : S \to T$ is one-to-one if it satisfies

$$\text{“if } f(s_1) = f(s_2) \text{ then } s_1 = s_2\text{”}.$$

Use the contrapositive argument

$$\frac{\text{“if } f(s_1) = f(s_2) \text{ then } s_1 = s_2\text{”}}{\therefore\text{“if } s_1 \neq s_2 \text{ then } f(s_1) \neq f(s_2)\text{”}.}$$

To we can use the statement in the conclusion to test if a given function f is one-to-one.

The third type of argument is the **proof by contradiction**,

$$\frac{p \to (q \wedge \neg q)}{\therefore \neg p}$$

Clearly, $q \wedge \neg q$ is a contradiction. If p implies a contradiction, then p has to be false.

We observe the validity of proof by contradiction by checking that

$$p \to (q \wedge \neg q) \quad \Rightarrow \quad \neg p$$

holds. Check that $\big(p \to (q \wedge \neg q)\big) \to \neg p$ is a tautology:

p	q	$q \wedge \neg q$	$\neg p$	$p \to (q \wedge \neg q)$	$\big(p \to (q \wedge \neg q)\big) \to \neg p$
T	T	F	F	F	T
F	T	F	T	T	T
T	F	F	F	F	T
F	F	F	T	T	T

Note: In the argument form for proof by contradiction, the contradiction $q \wedge \neg q$ can be replaced by any other contradiction, since all contradictions are logically equivalent.

Example 2.28 Prove that $\sqrt{2}$ is not rational (that is, $\sqrt{2} \notin \mathbb{Q}$).

Let

$$p = \text{“}\sqrt{2} \text{ is rational”}.$$

If $\sqrt{2} \in \mathbb{Q}$, then $\sqrt{2} = \frac{a}{b}$ for certain integers $a, b \in \mathbb{Z}$, $b \neq 0$. Since common factors of a and b cancel out, we can assume that a and b have no common prime factors. Let

$$q = \text{“}a \text{ and } b \text{ have no common prime factors”}.$$

So if p holds, then q follows. We will now show that $\neg q$ also follows if we assume p:

$$\begin{aligned} \sqrt{2} &= \frac{a}{b} \\ 2 &= \frac{a^2}{b^2} \\ b^2 &= 2a^2. \end{aligned}$$

This means 2 is prime factor of b^2. Because the of the unique decomposition of an integer into prime factors, the prime factors of b^2 are exactly the same as those of b, but appearing doubled multiplicity. Hence 2 is also a prime factor of b, that is, $b = 2k$ for some integer k. Then

$$\begin{aligned} b^2 = (2k)^2 &= 2a^2 \\ 4k^2 &= 2a^2 \\ 2k^2 &= a^2, \end{aligned}$$

which means that 2 is also prime factor of a^2 and hence of a. This means $\neg q =$ "a and b have a common prime factor" holds.

Our argument by contradiction is now

$$\frac{\text{"if } \sqrt{2} = \frac{a}{b} \in \mathbb{Q} \text{ then } a \text{ and } b \text{ have no common factor and } a \text{ and } b \text{ have a common factor"}}{\therefore \text{"}\sqrt{2} \notin \mathbb{Q}\text{"}}$$

and since we have just shown that the premise holds, it follows that $\neg p$,

$$\text{"}\sqrt{2} \notin \mathbb{Q}\text{"},$$

is true.

2.3 Predicate calculus

In mathematics we frequently consider statements such as "$x > 10$" or "if A is an invertible matrix". These statements are *not* propositions as they contain a variable (x or A in this case), and so they are true for some values of x (or A) and false for others. To discuss these statements we must extend our investigations to include statements like these; we call these statements **predicates** and use the notation $P(x)$ when they contain a variable x.

It is also important to specify the set of values the variables can take. This set is sometimes called the **universe**, so we will denote it by U. Often, it will be clear from the context what our universe is.

When allowing variables, we also need a way to indicate which values a variable can refer to. A predicate $P(x)$ could hold for a single element of the universe U, but also need a way to express that $P(x)$ should hold *for all* elements in U, or that *there exists* at least one element in U for which $P(x)$ should hold. We use **quantifiers** to express this:

The **universal quantifier** $\forall x \ldots$ means "for all x in $U \ldots$". So the statement "For all elements $x \in U$, $P(x)$ is true" is written as

$$\forall x : P(x).$$

The ":" does not have a meaning of its own here, it only serves to enhance readability. If we want to emphasise the universe U in which x can take its values, we write $\forall x \in U : P(x)$.

The universal quantifier $\forall$ is defined by its truth values:

$$\mathrm{val}(\forall x : P(x)) = \begin{cases} \mathsf{T} & \text{if } \mathrm{val}(P(x)) = \mathsf{T} \text{ for all } x \in U \\ \mathsf{F} & \text{else} \end{cases}.$$

Example 2.29 Universal quantifiers:

(a) Let $U = \mathbb{N}\backslash\{0\}$ and $P(x) =$ "$x \geq 1$". Then $\forall x : P(x)$ is true.

(b) Let $U = \mathbb{N}$ and $P(x) =$ "$x \geq 1$". Then $\forall x : P(x)$ is false, since the choice $x = 0 \in \mathbb{N}$ is allowed and $0 < 1$.

Note: As we see from the example above, the evaluation of the universal quantifier depends on the universal set U.

The **existential quantifier** $\exists x$ means "there exists x...", or more precisely "there exists at least one x in U...". So the statement "There exists an $x \in U$ such that $P(x)$ is true for this x" is written as

$$\exists x : P(x),$$

or as $\exists x \in U : P(x)$ if we want to emphasise the universe U.

The existential quantifier is defined by its truth values:

$$\mathrm{val}(\exists x : P(x)) = \begin{cases} \mathsf{T} & \text{if there exists } x \in U \text{ such that } \mathrm{val}(P(x)) = \mathsf{T} \\ \mathsf{F} & \text{else} \end{cases}.$$

Example 2.30 Existential quantifiers:

(a) Let $U = \mathbb{N}\backslash\{0\}$ and $P(x) =$ "$x < 1$". Then $\exists x : P(x)$ is false.

(b) Let $U = \mathbb{N}$ and $P(x) =$ "$x < 1$". Then $\exists x : P(x)$ is true, since $0 < 1$ and $0 \in U$.

Predicates can have more than one variable, and they can be combined using connectives.

Example 2.31 Let $U = \mathbb{R}$ and

$$Q(x, y) = \text{"}x < y\text{"}, \quad P(x, y) = \text{"}xy \leq 0\text{"}, \quad R(x, y) = \text{"}x^2 < y^2\text{"}.$$

These can be combined to the compound predicate

$$(P(x, y) \wedge Q(x, y)) \rightarrow R(x, y)),$$

which means "If $xy \leq 0$ and $x < y$ then $x^2 < y^2$".

Note: It makes only sense to talk about the truth value of a predicate if it is combined with one quantifier for each of its variables. Otherwise, the resulting statement contains an unspecified variable and hence its truth value cannot be determined.

In expressions with more than one quantifier, we omit brackets and write

$$\forall x \exists y : P(x, y) \quad \text{instead of} \quad \forall x : (\exists y : P(x, y)).$$

Example 2.32 Write

$$\text{"For each } n \in \mathbb{Z}\text{, there exists } m \in \mathbb{Z} \text{ such that } m < n\text{"}$$

using quantifiers and predicates. Solution:

variables:	m, n
universe:	$U = \mathbb{Z}$
predicate:	$P(m, n) =$ "$m < n$"
quantifiers:	$\exists m$, $\forall n$

Then the statement is

$$\forall n \exists m : m < n.$$

Is this a true statement?

How about

$$\exists m \forall n : P(m, n)?$$

Note: As the above example illustrates, the order of *distinct* quantifiers $\forall$, $\exists$ cannot be interchanged without changing the meaning of the statement. However, the order of several appearances of *the same* quantifier, say $\forall x \forall y$, can be interchanged without changing the meaning of the statement ($\forall x \forall y \ldots$ is the same as $\forall y \forall x \ldots$, and similarly for $\exists$).

Exercise 2.4 Given any predicate $P(x, y)$, show that $\exists y \forall x : P(x, y)$ logically implies $\forall x \exists y : P(x, y)$. Give a counterexample to show that they are not logically equivalent.

2.4 Arguments in predicate calculus

Just as in propositional calculus, we can use arguments in predicate calculus to draw conclusions from a given set of premises.

Direct reasoning can be extended to predicate calculus in the following way: Let $a \in U$, then the rule for direct reasoning is

$$\frac{\begin{array}{c}\forall x : (P(x) \to Q(x)) \\ P(a)\end{array}}{\therefore Q(a)}$$

The validity is verified by showing that for the chosen $a \in U$:

$$\big(\forall x : (P(x) \to Q(x)\big) \wedge P(a) \quad \Rightarrow \quad Q(a).$$

Note: In this type of argument, a is not a variable, but a fixed element of U. Then $P(a)$ is a proposition (not a predicate) which is obtained from the predicate $P(x)$ by replacing every occurrence of x in $P(x)$ by this particular choice a. So we get one rule of argument for each $a \in U$.

Example 2.33 Apply direct reasoning.

(a) Let the universe U be the set of all 2×2-matrices. Consider the predicates

$$P(X) = \text{“Matrix } X \text{ has determinant } \det(X) \neq 0\text{”}$$
$$Q(X) = \text{“Matrix } X \text{ is invertible”}.$$

A well-known theorem from linear algebra states

$$\forall X : Q(X) \to P(X) = \text{“If any matrix } X \text{ has determinant } \det(X) \neq 0 \text{ then } X \text{ is invertible”}.$$

For example, the matrix

$$A = \begin{pmatrix} 1 & 0 \\ 2 & 3 \end{pmatrix}$$

has determinant $3 \neq 0$. So direct reasoning allows us to conclude that A is invertible:

$$\frac{\begin{array}{c}\forall X : (P(X) \to Q(X)) \\ P(A)\end{array}}{\therefore Q(A)}$$

(b) Consider the following proof: Let $U = \mathbb{N}$ and

$$P(x) = \text{“}x \text{ is a prime number”}$$
$$Q(x) = \text{“}x \text{ is an odd integer”}.$$

Then we use direct reasoning

$$\frac{\begin{array}{c}\forall x : (P(x) \to Q(x)) \\ P(17)\end{array}}{\therefore Q(17)}$$

to conclude that 17 is a an odd integer. However, here we used the *false* proposition

"For all x it holds that if x is a prime number then x is an odd integer"

as a premise. Even though our direct reasoning was valid, using a false premise can lead to contradictory results: using the same reasoning for $x = 2$ yields $Q(2)$, but 2 is not odd, so $\neg Q(2)$ also holds. This is a contradiction.

We therefore cannot accept the reasoning in part (b) as a valid proof.

We have the following two rules for the **negation of quantifiers**

$$\begin{aligned} \neg\forall x : P(x) \quad &\Leftrightarrow \quad \exists x : \neg P(x), \\ \neg\exists x : P(x) \quad &\Leftrightarrow \quad \forall x : \neg P(x). \end{aligned}$$

Example 2.34 Consider universe $U = \mathbb{Z}$ and the predicates

$$\begin{aligned} P(x) &= \text{“}x > 0\text{”} \\ Q(x) &= \text{“}x^2 \leq 0\text{”}. \end{aligned}$$

(a) The proposition

$$\forall x : x > 0 = \text{“every integer } x \text{ is positive”}$$

is false. So its negation $\neg\forall x : P(x)$ is true. The negation states that not every integer is positive, which is equivalent to saying that there is at least one $x \in \mathbb{Z}$ that is not positive. This is consistent with the first rule:

$$\exists x : \neg(x > 0) = \exists x : x \leq 0 = \text{“there exists an integer } x \text{ that is not positive”}.$$

(b) The proposition

$$\exists x : x^2 \leq 0 = \text{“there exists an integer } x \text{ such that } x^2 \text{ is non-positive”}$$

is true (take $x = 0$). So its negation $\neg\exists x : Q(x)$ is false. The negation states that there exist no integers whose squares are non-positive, which is equivalent to saying that for all integers, their squares are positive. This is consistent with the second rule:

$$\forall x : \neg(x^2 \leq 0) = \forall x : x^2 > 0 = \text{“for all integers } x,\ x^2 \text{ is positive”}.$$

Suppose we face the task of deciding whether a given predicate $P(x)$ implies another predicate $Q(x)$ for every possible $x \in U$. That is, prove the truth or falsity of

$$\forall x : (P(x) \to Q(x)).$$

If we suspect that this might actually be false, then we will rather attempt to prove

$$\neg\forall x : (P(x) \to Q(x)).$$

The previous paragraph tells us that this is logically equivalent to proving

$$\exists x : \neg(P(x) \wedge Q(x)),$$

and by virtue of the logical equivalence $\neg(p \to q) \Leftrightarrow p \wedge \neg q$, this again is equivalent to proving

$$\exists x : (P(x) \wedge \neg Q(x)).$$

So, in order to disprove the original statement, we need to find *one* element $x \in U$ such that $P(x)$ is true, but $Q(x)$ is false. This approach is called **proof by counterexample**.

Example 2.35 Let $U = \{f \mid f : \mathbb{R} \to \mathbb{R} \text{ is a function}\}$, and consider the predicates

$$P(f) = \text{“the function } f \text{ is continuous”}$$
$$Q(f) = \text{“the function } f \text{ is differentiable”}.$$

Suppose someone claims that all continuous functions are differentiable,

$$\forall f : (P(f) \to Q(f)).$$

We can refute this claim by providing just one counterexample, such as the function $f(x) = |x|$, which is continuous ($P(f)$ is true) but not differentiable at the point $x = 0$ ($Q(f)$ is false). So there exist functions which are continuous but not differentiable,

$$\text{val}(\exists x : (P(x) \wedge \neg Q(x))) = \mathsf{T}.$$

3 Digital circuits

Formal logic is fundamental to the design of digital circuits used in computers and other electronic hardware.

Example 3.1 Consider a buzzer wired into a car, and look at the following five statements.

(a) The headlights are on.

(b) The key is in the ignition.

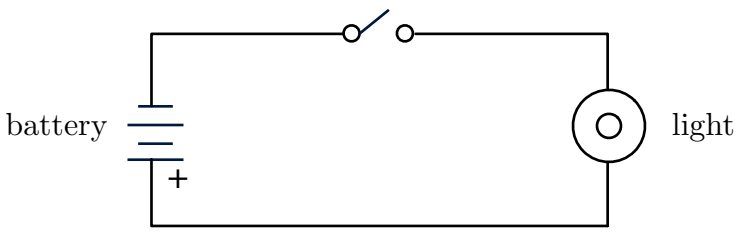
(c) The motor is running.

(d) The door is open.

(e) The seatbelt is fastened.

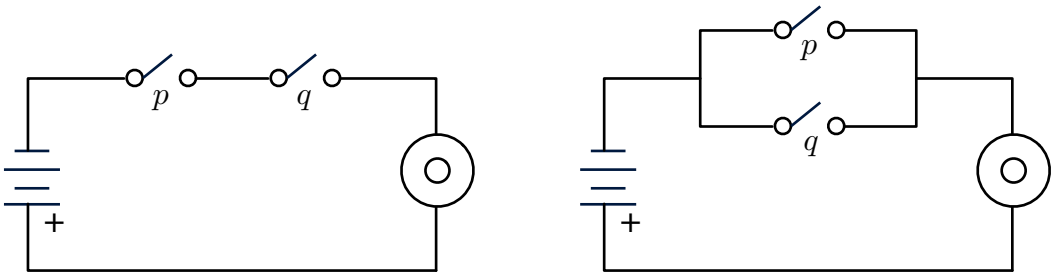
If (a) is true and (b) is false, then the buzzer should sound, since the driver has forgotten to turn off the lights. If (b), (c), and (d) are true, the buzzer should sound. If (b) is true, (d) is false, and (e) is false, the buzzer should sound to warn the driver to fasten the seatbelt. How do you design a circuit that will make the buzzer sound precisely when it should?

In the late 1930s, Claude Shannon (one of the pioneers in the study of Information Theory) observed the connection between logical connectives and electrical circuits. Consider a simple switch which allows current to flow (and light the globe) when the switch is closed:



If the switch is open, the current can not flow – the light globe is off.

Consider more complicated circuits:



switches in series switches in parallel

On the left, the current flows only when *both* switches are closed, and on the right it requires only one to be closed. The possible situations are completely described by the following tables:

	p	q	light
	closed	closed	on
series:	open	closed	off
	closed	open	off
	open	open	off

	p	q	light
	closed	closed	on
parallel:	open	closed	on
	closed	open	on
	open	open	off

If "closed" and "on" are represented by T (true) and "open" and "off" are represented by F (false), then the two tables become the truth tables for

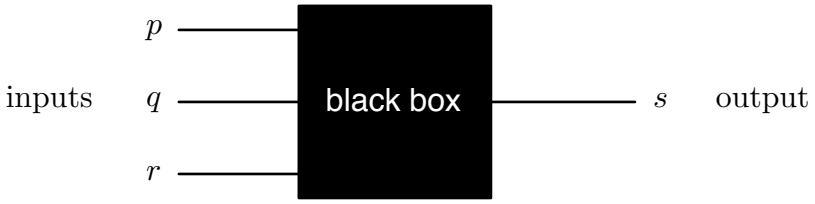
$$p \wedge q \quad \text{and} \quad p \vee q.$$

More complicated circuits can be analysed using truth tables.

In the 1940s and 1950s switches were replaced by electronic devices in which physical states of closed and open were replaced by electronic states of high and low voltage.

The basic electronic components of computers or other digital electronic systems are called digital circuits or switching circuits. The word *digital* indicates the circuits process discrete signals, as opposed to continuous ones. In discussing these circuits the symbols 1 and 0 are used instead of T and F. The symbols 0 and 1 are called **bits**, short for *binary digits*.

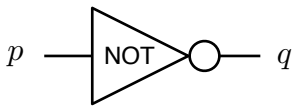
Combination of bits, 1s and 0s, are transformed into other 0s and 1s by means of various circuits. We can think of the basic circuits as "black boxes" producing one output bit out of a given input bits.



The inputs and the output all take values 0 or 1. The black box can be described completely by drawing up an **input/output table** to show the value of the output given the various values of the inputs.

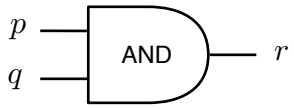
3.1 Basic gates

The design of complicated circuits is made up from more basic circuits. Three important basic circuits are known as **NOT-**, **AND-**, and **OR-gates**:



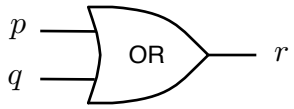
p	q
1	0
0	1

If we think of 1 as T and 0 as F, this is the truth table for $\neg p$.



p	q	r
1	1	1
0	1	0
1	0	0
0	0	0

This is the truth table for $p \wedge q$.



p	q	r
1	1	1
0	1	1
1	0	1
0	0	0

This is the truth table for $p \vee q$.

To construct more complicated circuits, we adopt the following rules:

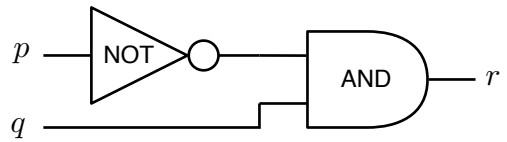
1. Two input wires cannot be combined, but an input wire can be split in two and used as inputs into two separate gates.

2. An output wire can be used as an input for a succeeding gate, but can not feed back into the gate from which it was output.

Circuits which follow these rules are called **combinational circuits**.

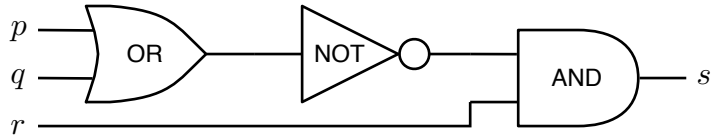
Example 3.2 Determine the input/output tables of the following circuits:

(a)



p	q	r
1	1	0
0	1	1
1	0	0
0	0	0

(b)



p	q	r	s
1	1	1	0
0	1	1	0
1	0	1	0
0	0	1	1
1	1	0	0
0	1	0	0
1	0	0	0
0	0	0	0

As we can think of the inputs 1 and 0 as T and F and the gates as the logic connectives, we can derive a **Boolean expression** (compound proposition) which represents the digital circuit.

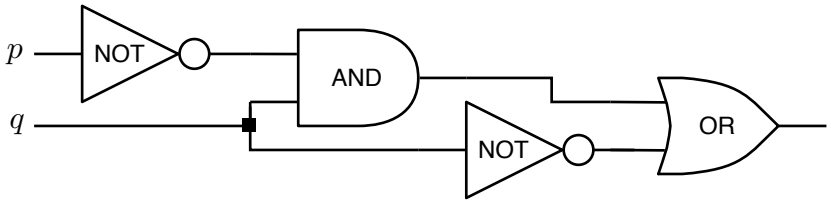
Example 3.3 The Boolean expressions for the circuits in Example 3.2:

(a) $\neg p \wedge q$

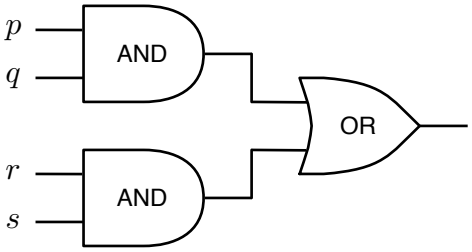
(b) $\neg(p \vee q) \wedge r$

Example 3.4 Let us work back the other way and construct a circuit to represent a given Boolean expression.

(a) The circuit for $(\neg p \wedge q) \vee \neg q$ is



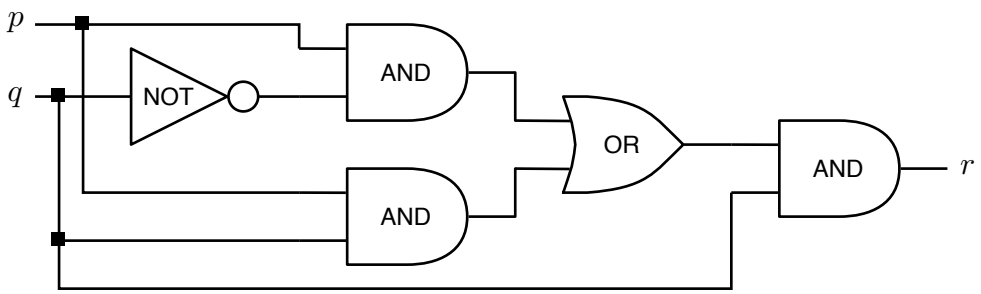
(b) The circuit for $(p \wedge q) \vee (r \wedge s)$ is



3.2 Equivalent circuits

If two circuits have the same input/output table then they do the same job. They are then said to be **equivalent**. If one is much simpler than the other, then there are cost advantages in using the simpler circuit. Two circuits will be equivalent if their corresponding Boolean expressions are equivalent, that is, they have the same truth tables.

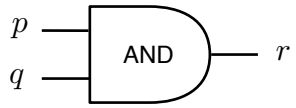
Example 3.5 Consider the circuit



The input/output table for this circuit is:

p	q	r
1	1	1
0	1	0
1	0	0
0	0	0

You should recognise this as the truth table for $p \wedge q$. Thus the above circuit is equivalent to



In terms of Boolean expressions or propositions:

$$(p \wedge \neg q) \vee (p \wedge q) \wedge q \quad \Leftrightarrow \quad p \wedge q.$$

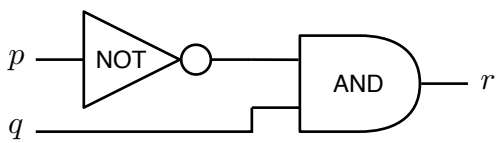
Example 3.6 Design a circuit with two inputs p, q such that the output is 1 only when $p = 0$ and $q = 1$ (in all other cases it is 0). Look at the truth table needed:

p	q	r
1	1	0
0	1	1
1	0	0
0	0	0

The output should remind you of the truth table for $p \wedge q$ except the 1 is not in the correct position. If we table $\neg p \wedge q$ we get the correct table:

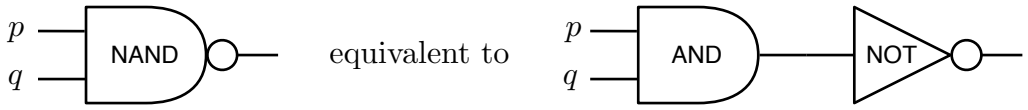
p	q	$\neg p$	$\neg p \wedge q$
1	1	0	0
0	1	1	1
1	0	0	0
0	0	1	0

The circuit is therefore:



Example 3.7 There are three further simple gates that are often used – the NAND-, the NOR-gate and the XOR-gate

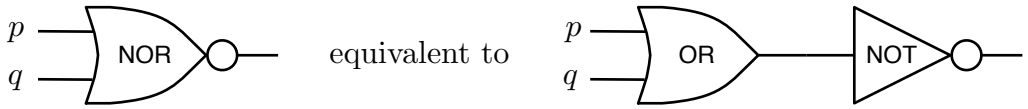
A **NAND-gate** is a single gate that acts like an AND-gate followed by a NOT-gate:



Its input/output table is

p	q	r
1	1	0
0	1	1
1	0	1
0	0	1

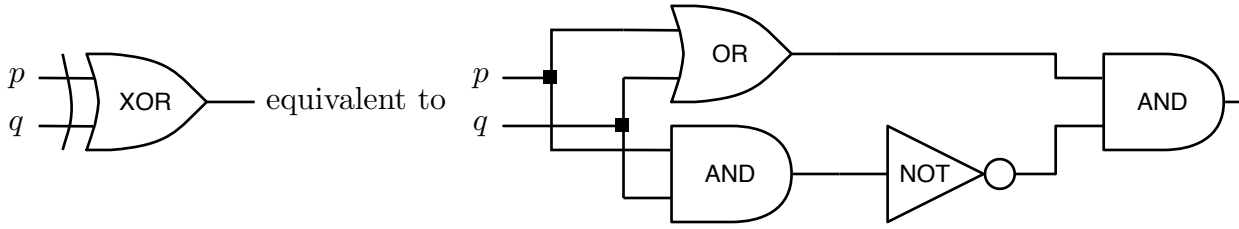
A **NOR-gate** is a single gate that acts like an OR-gate followed by a NOT-gate:



Its input/output table is

p	q	r
1	1	0
0	1	0
1	0	0
0	0	1

An **XOR-gate** produces output 1 if either p or q are 1 but not both:



Its input/output table is

p	q	r
1	1	0
0	1	1
1	0	1
0	0	0

3.3 Binary adders

Let us now move beyond mimicking Boolean expressions by circuits and implement a simple arithmetical operation as a combinational circuit.

Digital circuits can deal with inputs 0 and 1. So any computer based on digital circuits naturally represents numbers in the **binary numeral system**. To find the binary representation of an integer number b, write it as

$$b = b_{k-1}2^{k-1} + b_{k-2}2^{k-2} + \ldots + b_1 2 + b_0,$$

where k is chosen such that $2^k > b$ and $b_{k-1}, \ldots, b_0 \in \{0, 1\}$. The binary representation of b is then given by the string of bits

$$b_{k-1}b_{k-2}\ldots b_1 b_0.$$

This is analogous to digits in the decimal representation of b being given by the multiples of the powers of 10 occuring in b.

Example 3.8 The number $42 = 4 \cdot 10^1 + 2$ is expressed in the binary system as

$$101010.$$

This is because

$$42 = 32 + 8 + 2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 1.$$

Exercise 3.1 There are 10 kinds of people: those who can read binary and those who cannot. Explain.

We can add two numbers in binary representation by the following algorithm:

Algorithm 3.9 (Binary addition) Given the binary representations of two integers a and b,

$$a_{n-1}a_{n-2}\dots a_1a_0, \quad b_{m-1}b_{m-2}\dots b_1b_0.$$

Let $M = \max\{m, n\}$. Compute the binary representation of $c = a + b$:

Set

$$c_0 = \begin{cases} 1 & \text{if } a_0 + b_0 = 1 \\ 0 & \text{else} \end{cases}$$

$$\text{carry}_0 = \begin{cases} 1 & \text{of } a_0 + b_0 = 2 \\ 0 & \text{else} \end{cases}$$

For i from 1 to $M - 1$ compute:

$$c_i = \begin{cases} 1 & \text{if } a_i + b_i + \text{carry}_{i-1} \text{ is odd} \\ 0 & \text{else} \end{cases}$$

$$\text{carry}_i = \begin{cases} 1 & \text{if } a_0 + b_0 + \text{carry}_{i-1} \geq 2 \\ 0 & \text{else} \end{cases}$$

If $\text{carry}_{M-1} = 1$, set $c_M = 1$. Otherwise, set $c_M = 0$.

Then the binary representation of $c = a + b$ is

$$c_Mc_{M-1}\dots c_1c_0.$$

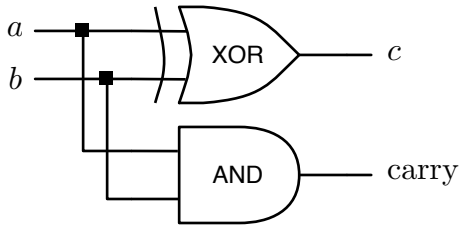
A string of 8 bits is called a **byte**. One byte can represent the positive integers from 0 to $2^8 - 1$. Most computers store data in multiples of bytes and usually process numbers in the size of 32 bits (4 bytes) or 64 bits (8 bytes). To keep things simple, we assume a processor handling 4-bit numbers and implement a circuit to add two positive 4-bit numbers in the following examples.

Example 3.10 The basic element is a circuit adding two bits a and b, with output being the sum $a + b$ according to Algorithm 3.9 and the carry of the addition. Such a circuit is called a **half adder**.

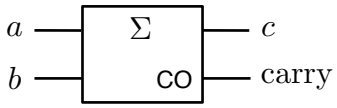
From the initial case in Algorithm 3.9 we can deduce what the input/output table has to look like:

a	b	c	carry
1	1	0	1
0	1	1	0
1	0	1	0
0	0	0	0

We use an AND- and an XOR-gate to implement a circuit for this input/output table:



We use the following symbol for the half adder:



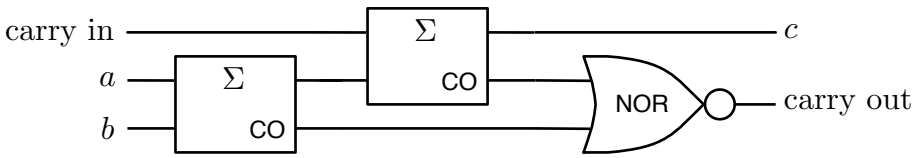
Here, "CO" stands for *carry out*.

Example 3.11 To add numbers given by a string of k bits, we need to extend the functionality of the half adder to deal with a carry coming from the addition of the previous bits, as in the loop of Algorithm 3.9.

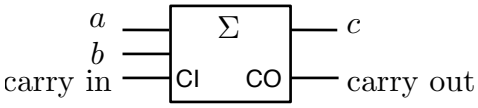
A **full adder** implements this functionality. Its truth table is

a	b	carry in	c	carry out
1	1	1	1	1
0	1	1	0	1
1	0	1	0	1
0	0	1	1	0
1	1	0	0	1
0	1	0	1	0
1	0	0	1	0
0	0	0	0	0

We can use half adders and a NOR-gate to implement a full adder:

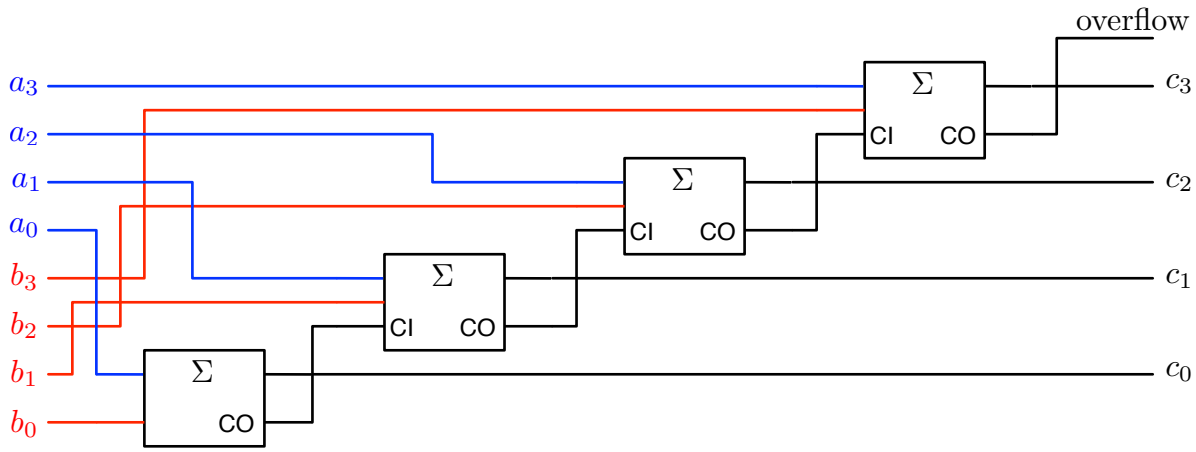


We use the following symbol for the full adder:



We are now ready to design a circuit adding two numbers $a, b \in \{0, \ldots, 2^4 - 1\}$ given in their 4-bit binary representations $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$.

Example 3.12 The digital circuit to add two 4-bit numbers resembles Algorithm 3.9 for $M = n = m = 4$, with the sole difference being that if $\text{carry}_3 = 1$ in the last step, we cannot introduce an additional digit, since we are limited to 4 bits. In this case, an overflow will be indicated by setting a carry out bit to 1. The circuit uses one half adder and three full adders:



4 Induction and recursion

Suppose $P(n)$ is some predicate depending on a variable n taking its in $\mathbb{N}$, and that we want to prove that, beginning with a certain inital value $n_0 \in \mathbb{N}$, the statement $P(n)$ holds for all integers $n \geq n_0$.

For example, suppose we want to prove that for all $n \geq 1$, the following formula holds:

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

Induction will provide us with a tool that allows us to prove this statement and most others of the type described above.

4.1 The principle of induction

The **principle of induction** was first formulated by the philosopher Blaise Pascal in 1665. We take it to be an *axiom*, meaning we postulate it to be true without needing to prove it. It is stated as follows:

Let $P(n)$ be an predicate for a variable n taking values in $\mathbb{N}$. If the following holds,

(1) $P(n_0)$ is true for some $n_0 \in \mathbb{N}$,

(2) $P(n)$ implies $P(n+1)$ for all $n \geq n_0$,

then $P(n)$ holds for all $n \geq n_0$.

We can replace (2) by the equivalent form

(2*) $P(n_0) \wedge \ldots \wedge P(n)$ implies $P(n+1)$ for all $n \geq n_0$.

In this form, we call it the **strong principle of induction**.

At first glance, the principle of induction may seem somewhat bewildering. It often helps to imagine domino stones set up in a long line, and compare the principle of induction with the toppling of one stone after another, once the first (or the n_0th) stone has been toppled. In this image, the inital case corresponds to the first stone being toppled, and the condition that $P(n)$ implies $P(n+1)$ corresponds to the fact once the nth stone falls, the $n+1$st stone will also fall.

The (strong) principle of inductions allows us to use the following three-step approach to proving a statement $P(n)$ for all $n \geq n_0$ as follows:

Initial case: Prove the statement $P(n_0)$ for the initial value n_0.

Induction hypothesis: Make the assumption that $P(n)$ holds for a certain $n \geq n_0$ (or in the strong form: $P(k)$ holds for $n \geq k \geq n_0$).

Induction step: Use the induction hypothesis to prove that $P(n+1)$ follows from $P(n)$ (or in the strong form: that it follows from $P(n_0), \ldots, P(n)$).

We can then apply the (strong) principle of induction to conclude that $P(n)$ holds for all $n \geq n_0$.

Exercise 4.1 Prove the example from the beginning of this section, namely, that

$$1 + 2 + \ldots + n = \frac{n(n-1)}{2}$$

holds for all $n \geq 1$.

Example 4.1 We use induction to prove the **Bernoulli inequality** for all $x \geq -1$ and for all integers $n \geq 0$:

$$(1+x)^n \geq 1 + nx.$$

PROOF: The statement $P(n)$ to prove is $(1+x)^n \geq 1 + nx$ all $x \geq -1$. The initial value is $n_0 = 0$.

Initial case: For $n = 0$,

$$(1+x)^0 = 1 \geq 1 + 0.$$

Induction hypothesis: For a certain $n \geq 0$, it holds that for all $x \geq -1$

$$(1+x)^n \geq 1 + nx.$$

Induction step: Consider

$$\begin{aligned}(1+x)^{n+1} &= (1+x)(1+x)^n \\ &\geq (1+x)(1+nx) \quad \text{use induction hypothesis} \\ &= 1 + nx + x + \underbrace{nx^2}_{\geq 0} \\ &\geq 1 + (n+1)x.\end{aligned}$$

By the principle of induction, $P(n)$ holds for all $n \geq 0$. ∎

Exercise 4.2 Prove $(1+x)^n > 1+nx$ for all $n \geq 2$.

Example 4.2 Any number $n \geq 12$ can be written as

$$n = 7a + 3b$$

for certain $a, b \in \mathbb{N}$.

PROOF: The initial value is $n_0 = 12$, the statement $P(n)$ to prove is that there exist $a, b \in \mathbb{N}$ such that $n = 7a + 3b$ for all $n \geq 12$.

Initial case: $n = 12$. Choose $a = 0$ and $b = 4$, then

$$12 = 3 \cdot 4.$$

Induction hypothesis: For a certain $n \geq 12$, there are $a, b \in \mathbb{N}$ such that

$$n = 7a + 3b.$$

Induction step: By the induction hypothesis, $n = 7a + 3b$. Then

$$n + 1 = 7a + 3b + 1.$$

If $a = 0$ or $a = 1$, then $b \geq 2$, since $n \geq 12$. Then

$$n + 1 = 7a + 3b + 1 = 7a + 3(b-2) + 6 + 1 = 7 + 3(b-2).$$

So for $n+1$, replace a by $a+1$ and b by $b-2$.

Otherwise, $a \geq 2$. Then

$$n + 1 = 7a + 3b + 1 = 7(a-2) + 15 + 3b = 7(a-2) + 3(b+5).$$

So for $n+1$, replace a by $a-2$ and b by $b+5$.

In either case, the statement is true for $n+1$. By the principle of induction, $P(n)$ holds for any $n \geq 12$. ■

The proof of the following theorem is an example where we need the strong form of the principle of induction.

Theorem 4.3 *Any integer $n > 1$ is a product of prime numbers.*

PROOF: *Initial case:* $n = 2$ is a prime number itself.

Induction hypothesis: For certain n all numbers $2 \le k \le n$ are products of prime numbers.

Induction step: If $n+1$ is a prime number itself, we are done.

Otherwise, $n+1 = ab$ for some integers $a, b \ge 2$. Also, both a and b are smaller than $n+1$. This means $2 \le a, b \le n$, and by the induction hypothesis both of them are products of prime numbers. But then $n+1$, being the product of a and b, is a product of prime numbers. ■

The representation of a number n as a product of prime numbers is called the **prime factorisation** of n.

Exercise 4.3 Prove that the prime factorisation in Theorem 4.3 is unique: Suppose

$$n = p_1 \cdots p_s = q_1 \cdots q_t$$

are two prime factorisations for n with prime numbers $p_1, \ldots, p_s, q_1, \ldots, q_t$. Prove by induction on s that $s = t$ and the p_i coincide with the q_j up to their order (use the following property of prime numbers: if a prime number p divides a product ab, then p must divide one of the factors a or b).

To prove a result by induction it is absolutely necessary to prove both, the initial case and the induction step. The following examples show how things can go horribly wrong if either step is not verified.

Example 4.4 Suppose the statement $P(n)$ we want to prove is $5^n > 5^{n+1}$ for all $n \ge 1$. If we conveniently forget to verify the initial case $n = 1$, then we can indeed use the induction hypothesis to conclude

$$5^{n+1} = 5 \cdot 5^n > 5 \cdot 5^{n+1} = 5^{n+2}.$$

So $P(n)$ does indeed imply $P(n+1)$.

Of course, this proof is invalid because we did not verify the inital case.

Example 4.5 Verifying that $P(n)$ is true for a few (even a large number of) initial cases is not enough to say that $P(n)$ is true for all $n \ge n_0$.

Consider the statement $P(n)$ saying that $n^2 + n + 41$ is prime for all positive integers n. We find that this is true for a surprisingly large number of initial

values:

n	$n^2 + n + 41$	
0	41	
1	43	
2	47	
3	53	all prime
4	61	
5	71	
⋮		
39	1601	

However, for $n = 40$ we find

$$40^2 + 40 + 41 = 41^2,$$

which is not a prime number.

Even though we find initial values satisfying $P(n)$, the induction step cannot be proved for any n, as the counterexample shows.

4.2 Recursion

Recall from Section 1.6 that a **sequence** is a function f which takes its value on (a subset of) the non-negative integers $\mathbb{N}$. We often write f_n for the function value at n instead of $f(n)$.

A sequence f is defined **recursively** if

(1) it is defined for some initial values,

(2) the value f_n is defined in terms of previously defined values $f_{n-1}, f_{n-2}, \ldots$.

A **recurrence relation** is a formula that expresses how f_n depends on the previously defined values $f_{n-1}, f_{n-2}, \ldots$

Note: The number of previously defined values that f_n depends on is fixed. In particular, f_n does not have to depend on all previously defined values.

Due to the dependence on previously defined values, it is natural to use induction as an approach to prove properties of recursively defined sequences.

Example 4.6 The famous **Fibonacci numbers** are defined recursively by the two inital values

$$f_0 = 0, \quad f_1 = 1$$

and the recurrence relation

$$f_n = f_{n-1} + f_{n-2} \quad \text{for } n \geq 2.$$

The Fibonacci numbers are known for their habit of appearing unexpectedly in many different places in computer science.

Exercise 4.4 The **golden ratio** is the number $\Phi = \frac{1}{2}(1 + \sqrt{5})$. Set $\Psi = 1 - \Phi$. Prove by induction on n that the nth Fibonacci number is given by

$$f_n = \frac{\Phi^n - \Psi^n}{\sqrt{5}}.$$

Hint: Prove $\Phi^n = \Phi^{n-1} + \Phi^{n-2}$ and $\Psi^n = \Psi^{n-1} + \Psi^{n-2}$ first.

Recursively defined sequences play an important role computer programming. Here, **recursion** refers to the principle of implementing an algorithm or a program by reducing the task at hand to a task of the same type, but with smaller input values. Initial values must be provided so that the program can at some point stop calling itself.

Example 4.7 The following recursive program computes the nth Fibonacci number:

```
int fib(n)
    if n = 0 then return 0
    if n = 1 then return 1
    else return fib(n-1) + fib(n-2)
```

If $n > 1$, then the program will call itself with the reduced input $n - 1$. By checking first if we are in one of the initial cases $n = 0$ or $n = 1$, we provide a condition to eventually stop the recursion and terminate the execution of the program.

One important task of mathematics in computer science is the analysis if algorithms and their implementations. This includes

(a) Verifying that a given program terminates eventually.

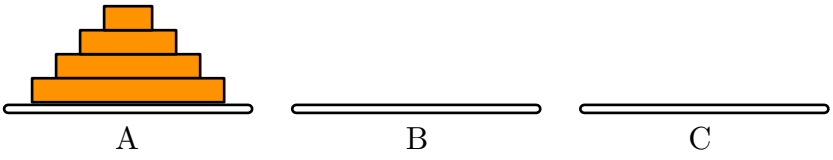
(b) Verifying that a program computes a correct result, provided the input data meet certain conditions.

(c) Determine the costs of the computation, for example the number of functions calls, loop iterations, comparisons or arithmetic operations.

Due to the close relation of recursion and mathematical induction, recursive programs are well-suited for such a mathematical analysis.[2)] However, there is a downside to recursive progamming: If the input parameter n is large and the recursive program calls itself very often, then a lot of computer memory has to be used to remember at each new call where the previous execution of the program needs to be continued. For this reason, recursive algorithms are often transformed into iterative procedures and implemented using loops rather than repeated program calls.

We will now take a look at two famous examples of recursive algorithms, and indicate how a mathematical analysis works.

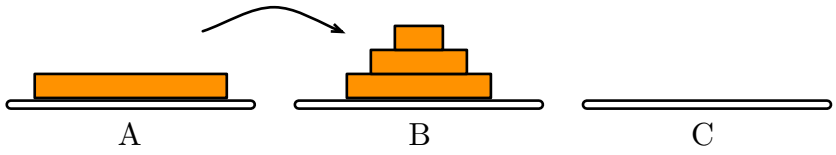
Algorithm 4.8 (Tower of Hanoi) In this classical puzzle, we are given three positions labelled A, B and C. On position A we have a pile of n disks, with the lowest disk being the largest and each of the other disks being smaller than the one below it.



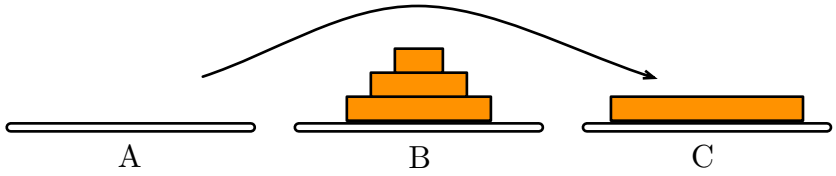
The task is to *move all disks from position A to position C*, subject to the rule that we may only place a smaller disk on top of a larger disk, but not the other way round. The position B can be used as a buffer.

The solution to this problem lies in the following observation: Assume we already know how to move a pile of $n-1$ disks. We can then move the topmost $n-1$ disks from our pile to position B (using C as the buffer), so that only disk n remains on position A.

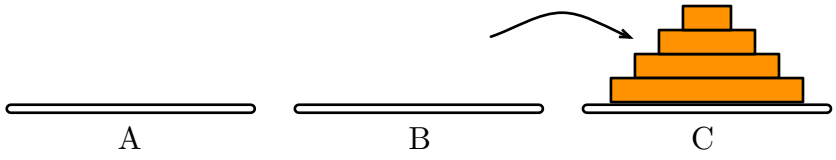
[2)] This is one of the reasons why functional programming languages such as Lisp are so popular for logical or mathematical applications.



Next, we move disk n to position C.



Because we already know how to move $n-1$ disks, all that is left to do now is to move the $n-1$ disks on position B to position C (using A as the buffer).



At first glance, it seems like we are cheating by assuming we already know how to solve the problem when we apply it to move $n-1$ disks. But the trick is that we may now apply the same idea *recursively* to the problem with $n-1$ disks by assuming we already know how to move $n-2$ disks, and so on. Eventually, we arrive at the problem with only 1 disk left. Since this disk is the smallest one in the pile, we can move it wherever we want.

The following program solves the Tower of Hanoi puzzle:

```
void hanoi(n,A,B,C)
    if n = 1 then move(A,C)
    else
        hanoi(n-1,A,C,B)
        move(A,C)
        hanoi(n-1,B,A,C)
```

Here, the function `move(A,C)` moves the topmost disk from A to C.

The proof of correctness is an easy exercise now.

Exercise 4.5 Under the assumption that $n \geq 1$ and the function `move(A,C)` correctly moves the topmost disk from position A to C, use induction on n to prove that the program given for Algorithm 4.8 is *correct*, meaning that it eventually terminates and it does indeed move the pile of n disks from A to C.

The computational costs of Algorithm 4.8 is determined by the number of move operations performed.

Theorem 4.9 *When solving the Towers of Hanoi puzzle with n disks using Algorithm 4.8, a total of $2^n - 1$ moves are required.*

PROOF: We prove this by induction on n.

Initial case: For $n = 1$, only one move is required, $1 = 2^1 - 1$.

Induction hypothesis: For a tower of n disks, $2^n - 1$ are required.

Induction step: Given a tower of $n+1$ disks, the algorithm performs one move and calls itself twice with reduced input size n. By the induction hypothesis, each of these calls lead to $2^n - 1$ moves. In total we have

$$1 + 2 \cdot (2^n - 1) = 1 + 2^{n+1} - 2 = 2^{n+1} - 2$$

moves. ■

Our next example is a widely used sorting algorithm.

Algorithm 4.10 (Quicksort) Given a list L containing n elements of a certain data type T that allows us to compare elements of T. For example, $T = \mathbb{R}$ and the comparison is the test if $x \leq y$ for $x, y \in \mathbb{R}$.

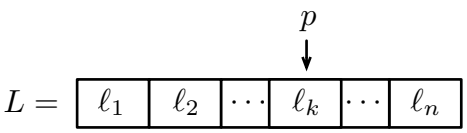
The n elements of the list L will be referred to as $\ell_1, \ell_2 \ldots, \ell_n$.

$$L = \boxed{\ell_1} \boxed{\ell_2} \boxed{\cdots} \boxed{\ell_n}$$

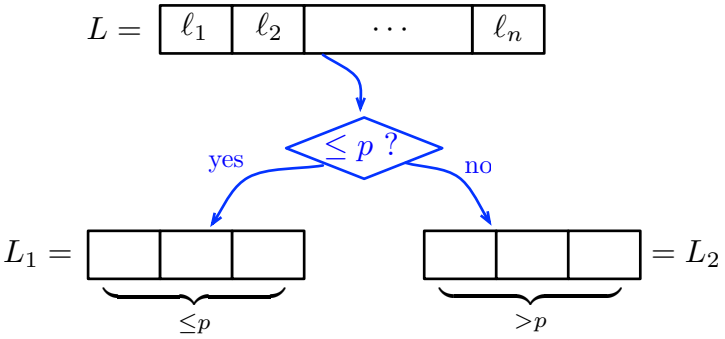
The task is to *sort the list in ascending order*, so that on termination of the algorithm, the sorted list L' satisfies

$$\ell'_1 \leq \ell'_2 \leq \ldots \leq \ell'_n.$$

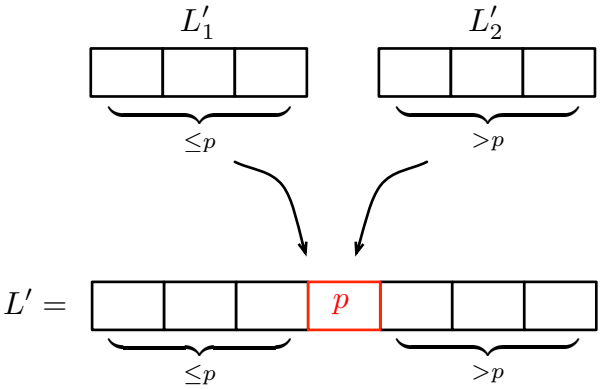
To achieve this, we proceed as follows: First, pick an element p from the list L. This element p is called the **pivot**. The pivot is often chosen at random.



Then remove p and iterate over the list L and move all elements $\ell_i \leq p$ into a new list L_1 and all elements $\ell_i > p$ into another new list L_2.



We apply the same procedure recursively to the lists L_1 and L_2, terminating when a list with zero or one element appears. Then we concatenate the now sorted lists L'_1, $[p]$ and L'_2 into the sorted list L'.



The following program implements the quicksort algorithm:

```
listType quicksort(L)
    if length(L) <= 1 then return L
    L1 := new listType []
    L2 := new listType []
    p := pick_and_remove_pivot(L)
    for k = 1 to length(L)
        if L[k] <= p then append(L1,L[k])
        else append(L2,L[k])
    L1 := quicksort(L1)
    L2 := quicksort(L2)
    return concatenate(L1,[p],L2)
```

Here, `listType` is a data type for lists, the constructor `new listType` creates a new empty list, `append` inserts an element at the end of a given list, and `concatenate` creates a new list by combining its arguments into one larger list.

Note: More sophisticated implementations of quicksort can do without using memory space for new lists, but rather using the space in the original list L.

Theorem 4.11 *Algorithm 4.10 correctly sorts a list L of length n containing elements of a data type which allows comparisons of its elements.*

PROOF: We need to prove that quicksort terminates and the list L' is sorted in ascending order on termination of the algorithm.

Initial case: $n = 0$. An empty list is always ordered.

Induction hypothesis: For a given n, assume that on termination, quicksort returns a correctly ordered list L' for any input list L of length between 1 and n.

Induction step: Given a list L of length $n + 1$, quicksort picks and removes a pivot p. The remaining n elements get sorted into either L_1 or L_2, which means that both L_1 and L_2 have length $\leq n$.

By the induction hypothesis, quicksort terminates for either argument L_1 or L_2 and returns the sorted list L_1' or L_2', respectively.

By construction of L_1, all elements in L_1' are $\leq p$. So appending p to L_1' yields again an ordered list L_1'' with maximal element p. Then all elements in L_2'

are larger than the elements in L_1'', and appending L_2' to L_1'' yields an ordered list L', containing all the elements from our original list L.

After returning from the recursive calls for L_1 and L_2, quicksort immediately terminates by returning the list L'. ■

The computational cost of quicksort is determined by the number of comparisons of list elements that need to be performed. In the worst case, it turns out that quicksort needs roughly n^2 comparisons for a list of length n. However, in the average case, it can be shown that quicksort performs much better, with roughly $n \log(n)$ comparisons only. We will not go into the details, since this analysis is quite involved and requires methods from probability theory.

5 Modular arithmetic

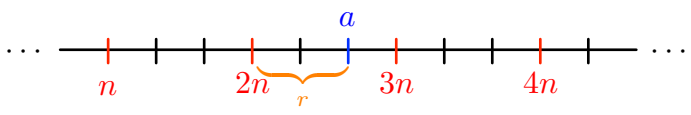
We are familiar with the arithmetic of the integers $\mathbb{Z}$ which allows us to add and multiply any two integer numbers. In modular arithmetic, these two operations are modified in such a way that they "wrap around" a certain value n, effectively producing a system of arithmetic on the finite set of the numbers $0, 1, \ldots, n-1$. Modular arithmetic was introduced by Carl Friedrich Gauß in his monumental work *Disquisitiones Arithmeticae* in 1801. Since then it has evolved into the lively research area of number theory. It abounds in computer science; in particular, the arithmetic performed by any digital computer is modular, and modern cryptographic procedures are based on modular arithmetic.

5.1 Division with remainder and congruences

Given two integers $a, n \in \mathbb{Z}$, we can only divide a by n if a is a multiple $a = qn$ of n. In this case, the quotient of the division is $q = \frac{a}{n} \in \mathbb{Z}$. Knowing q and n allows us to recover $a = qn$.

However, for most choices of integer numbers a, n, we cannot divide a by n in the integers, as the result would be a proper fraction. So we generalise the division to a **division with remainder**: If $n \neq 0$ and qn is the largest multiple of n that is less or equal to a, we define q to be the **quotient** of the division with remainder (if a is a multiple of n, then q coincides with the quotient of the usual division). In general, we cannot recover a by knowing q and n. For this, we need the **remainder** r of the division, which is the difference $r = a - qn$.

For example, in the figure below we have $a = 8$ and $n = 3$. The remainder is $r = 2$. The largest multiple of 3 smaller than 8 is $6 = 2n$, so the quotient is 2.



Theorem 5.1 *Given a number $n \in \mathbb{Z}\backslash\{0\}$, every other number $a \in \mathbb{Z}$ can be written in a unique way as*

$$a = qn + r \qquad \text{with } 0 \leq r < n.$$

PROOF: Define q to be the number such that qn is the largest multiple of n that is less or equal to a. Then q is unique by definition, and from this the uniqueness of the remainder $r = a - qn$ follows. Again by the definition of q,

$$qn \leq a < (q+1)n$$

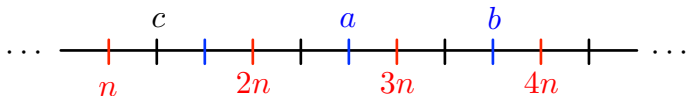
holds, which is equivalent to $0 \leq r < n$. ∎

Recall from Section 1.4 the equivalence relation "congruence modulo n",

$$a \equiv b \mod n,$$

defined by $a, b \in \mathbb{Z}$ having the same remainder on division by n. By Theorem 1.21, this is equivalent to $x - y$ being a multiple of n.

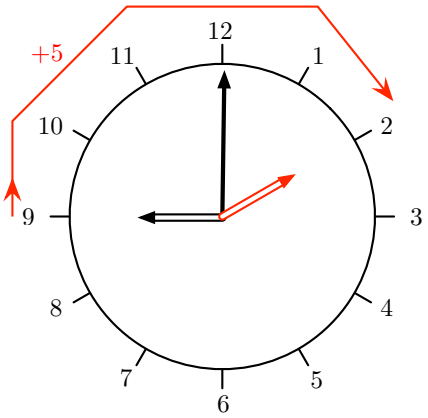
Below, the numbers a and b are congruent modulo n, where we chose $n = 3$. This is evident from the fact that both a and b have the same distance $r = 2$ from their respective next-smallest multiple of n. The number c is not congruent to a or b, since its remainder on division by $n = 3$ is only 1.



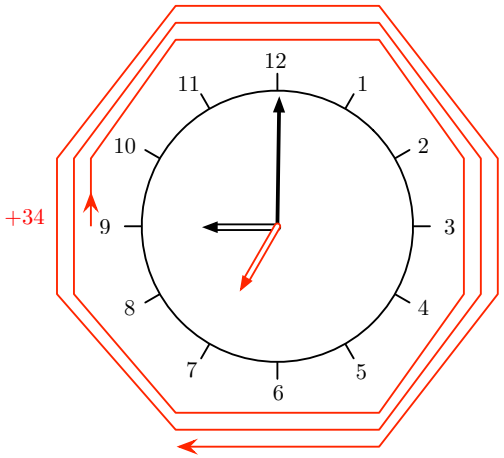
Note that the difference between a and b is precisely $n = 3$.

5.2 Arithmetic modulo n

We use modular arithmetic all the time, even if we are not aware of it. Think of the face of a clock, and the way we determine what the time will be in a few hours.



Suppose it is 9 o'clock, and we want to know what the time will be in 5 hours. We add 5 hours to 9 o'clock. However, we will not say that it will be $9 + 5 = 14$ o'clock, but rather subtract all multiples of 12 from the computed value, $14 - 12 = 2$, and say it is 2 o'clock. Similarly, in 34 hours from now, it will not be $9 + 34 = 43$ o'clock, but $43 - 3 \cdot 12 = 7$ o'clock:



Now observe that

$$14 \equiv 2 \mod 12,$$
$$43 \equiv 7 \mod 12.$$

When computing the time, we are not interested in any multiples of 12, but only in the remainder on division by 12. In particular, if we add 12 hours to

the current time, the clock will be in the same position. So we obtain our "clock arithmetic" by identifying the number 12 with the number 0. Another way to interprete this is as the introduction of the new law "$12 = 0$" into the arithmetic of the integers.

Now let n be any integer number, and suppose we introduce the new law "$n = 0$" into the arithmetic of the integers $\mathbb{Z}$. With this new law, we find that every multiple kn of n becomes a multiple of 0 and hence equals 0 itself. Also, we find that two numbers a and b that differ by a multiple of n, say $b = a + kn$, now only differ by a multiple of 0, which means they can no longer be distinguished. So the new law implies "$a = b$" if $a - b = kn$. But this is precisely the condition that a and b are congruent modulo n. Equality of two numbers under the new law is thus the same thing as congruence modulo n, so we speak of **modular arithmetic** and will write

$$a = b \mod n$$

when two integers a, b are identical in arithmetic modulo n.

Theorem 5.2 *Let $a, b, a', b' \in \mathbb{Z}$ and assume*

$$a = a' \mod n,$$
$$b = b' \mod n.$$

Then:

(a) $a + b = a' + b' \mod n$.

(b) $ab = a'b' \mod n$.

PROOF: Recall that the assumption is equivalent to $a - a'$ and $b - b'$ being multiples of n. Then $a + b - (a' + b') = (a - a') + (b - b')$ clearly is also a multiple of n, which proves (a). Also, $ab - a'b'$ is a multiple of n, which proves (b). ■

Example 5.3 Let $n = 7$ and $a = 5$, $b = 6$, $a' = 26$, $b' = -1$. Then $a - a' = -21 = (-3) \cdot 7$ and $b - b' = 7$, so indeed

$$a = a' \mod n, \quad b = b' \mod n.$$

Now compute $a + b = 5 + 6 = 11$ and $a' + b' = 26 - 1 = 25 = 11 + 2 \cdot 7$. So

$$a + b = a' + b' \mod 7.$$

Theorem 5.2 says that the operations addition and multiplication of the integers $\mathbb{Z}$ are inherited by modular arithmetic. There are only n equivalence classes $[0], [1], \ldots, [n-1]$ of numbers modulo n (compare Example 1.26), and so we write

$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$$

whenever we think of the numbers $0, 1, \ldots, n-1$ together with addition and multiplication inherited from the integers. These numbers are called the **residues** modulo n.

We will still allow to write $a + kn$ for a in modular arithmetic, as the two numbers now represent the same element in $\mathbb{Z}_n$.

Example 5.4 A particularly important case is the arithmetic modulo 2 in $\mathbb{Z}_2 = \{0, 1\}$. Here,

$$\begin{aligned} 0 + 0 &= 0 \mod 2 \\ 1 + 0 &= 1 \mod 2 \\ 0 + 1 &= 1 \mod 2 \\ 1 + 1 &= 0 \mod 2, \end{aligned}$$

This is precisely the arithmetic we used for the addition of single bits in Algorithm 3.9, and when we recall the implementation of binary addition in the half adder circuit, we see that this addition table resembles the input/output table of the XOR-gate.

Multiplication in $\mathbb{Z}_2$ is given by

$$\begin{aligned} 0 \cdot 0 &= 0 \mod 2 \\ 1 \cdot 0 &= 0 \mod 2 \\ 0 \cdot 1 &= 0 \mod 2 \\ 1 \cdot 1 &= 1 \mod 2. \end{aligned}$$

The multiplication table resembles the input/output table of the AND-gate.

Example 5.5 Equations that do not have solutions over the integers can sometimes have solutions in modular arithmetic. Consider the equation

$$x^2 = -1.$$

This equations clearly has no solution in the integers. However, for $n = 10$, the equation

$$x^2 = -1 \mod 10$$

has the solution $x = 3$. This is because $3^2 = 9 = 10 - 1 = -1 \mod 10$. Then again, for $n = 6$ the equation

$$x^2 = -1 \mod 6$$

has no solution, as one discovers by checking the squares of all elements $0, 1, 2, 3, 4, 5$ in $\mathbb{Z}_6$.

5.3 The Euclidean algorithm

Before, we saw that equations in modular arithmetic can have solutions even if they are not solvable over the integers. The equation

$$ax = 1 \mod n$$

is of particular importance. If it is solvable for a given element $a \in \mathbb{Z}_n$, we can find an **inverse** x of a in $\mathbb{Z}_n$. If this is the case, then the solution is unique and we write

$$x = a^{-1} \mod n.$$

Those elements of $\mathbb{Z}_n$ which have an inverse are called the **units** in $\mathbb{Z}_n$.

Example 5.6 In $\mathbb{Z}_4$, the units are 1 and 3:

$$1 \cdot 1 = 1 \mod 4 \quad \text{and} \quad 3 \cdot 3 = 8 + 1 = 1 \mod 4.$$

Another important equation is

$$ax = 0 \mod n.$$

If for a given $a \in \mathbb{Z}_n$ this equations has a solution $x \in \mathbb{Z}_n$ and $x \neq 0$, then a is a **zero-divisor**.

Example 5.7 In $\mathbb{Z}_4$, the zero-divisors are 0 and 2:

$$0 \cdot 0 = 0 \mod 4 \quad \text{and} \quad 2 \cdot 2 = 4 = 0 \mod 4.$$

As Examples 5.6 and 5.7 show, every element of $\mathbb{Z}_4$ is either a unit or a zero-divisor. More generally, this holds for all $\mathbb{Z}_n$.

Exercise 5.1 Prove that if a is a unit in $\mathbb{Z}_n$, then it is not a zero-divisor.

Exercise 5.2 Prove that if a is *not* a unit in $\mathbb{Z}_n$, then its multiples $0, a, 2a, \ldots, (n-1)a$ do not include every element of $\mathbb{Z}_n$. Show that this implies a is a zero-divisor.

We can determine whether an element a is a unit modulo n if we know the **greatest common divisor** $\gcd(a, n)$ of $a, n \in \mathbb{Z}$, which is defined as the largest number $g \in \mathbb{N}$ dividing both a and n.

The Euclidean algorithm below will allow us to compute the greatest common divisor of two integers a, b. But it can do even more than that; it allows us to compute integers s, t such that $\gcd(a, b) = as + bt$. This will come in handy when we wish to determine the inverse of an element modulo n.

In the algorithm, we use the following observation: Recall that division with remainder of two numbers $a_0, a_1 \in \mathbb{Z}$ yields $a_0 = qa_1 + r$ with $0 \le r < a_1$. Solving this for the remainder r gives

$$r = a_0 - qa_1.$$

Writing $a_2 = r$, we can reformulate this in matrix notation:

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}. \qquad (*)$$

Algorithm 5.8 (Euclidean algorithm) Set $a_0 = |a|$, $a_1 = |b|$ and let A_0 be the 2×2-identity matrix. If $a_0 < a_1$, then swap a_0 and a_1.

For $i = 1, \ldots, m$, divide with remainder,

$$a_{i-1} = q_i a_i + a_{i+1} \qquad \text{with } 0 \le a_{i+1} < a_i$$

and set

$$Q_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix}, \quad A_i = Q_i A_{i-1}$$

until the remainder $a_{m+1} = 0$ is obtained.

Then

$$a_m = \gcd(a_0, a_1) = sa_0 + ta_1,$$

where $A_m = \left(\begin{smallmatrix} s & t \\ s' & t' \end{smallmatrix}\right)$. If either of a or b is negative, replace the respective factor s or t by its negative.

Theorem 5.9 *The Euclidean algorithm is correct: it terminates for any input $a, b \in \mathbb{Z}$ and computes $\gcd(a, b)$ and $s, t \in \mathbb{Z}$ such that*

$$\gcd(a, b) = sa + tb.$$

PROOF: The Euclidean algorithm terminates because the remainders are ≥ 0 and decrease in each step, so that they will eventually reach 0.

First prove that for $i = 0, \ldots, m$,

$$\gcd(a_{i-1}, a_i) = \gcd(a_i, a_{i+1})$$

holds: Since a_{i+1} is defined by $a_{i+1} = a_{i-1} - q_i a_i$, every common divisor of a_{i-1} and a_i is also a common divisor of a_i and a_{i-1}, and vice versa. So a_{i-1}, a_i and a_i, a_{i+1} have the same common divisors, and in particular their greatest common divisor is the same.

It follows immediately that

$$\gcd(a_0, a_1) = \gcd(a_1, a_2) = \ldots = \gcd(a_m, a_{m+1}) = a_m,$$

since $a_{m+1} = 0$ (assuming the algorithm terminates in the mth step).

From equation $(*)$ on p. 79 it follows by induction on i that for $i \geq 0$

$$\begin{pmatrix} a_i \\ a_{i+1} \end{pmatrix} = A_i \cdot \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}.$$

In particular,

$$\begin{pmatrix} \gcd(a_0, a_1) \\ 0 \end{pmatrix} = \begin{pmatrix} a_m \\ a_{m+1} \end{pmatrix} = A_m \cdot \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}.$$

The first row of this product is given by $sa_0 + ta_1$, if s and t denote the first and second entry in the first row of A_m.

This proves the correctness of the computation for non-negative a, b. If one of a or b is negative, then clearly replacing the respective value s or t by its negative yields the correct result. ■

Example 5.10 Compute the greates common divisor of $a = 14$, $b = -10$. Set $a_0 = 12$ and $a_1 = |-10| = 10$ and use the Euclidean algorithm:

- $a_0 = 14, a_1 = 10$ gives $q_1 = 1$, $a_2 = 4$ and the matrix $Q_1 = \left(\begin{smallmatrix} 0 & 1 \\ 1 & -1 \end{smallmatrix}\right)$.

- $a_1 = 10, a_2 = 4$ gives $q_2 = 2$, $a_3 = 2$ and the matrix $Q_2 = \left(\begin{smallmatrix} 0 & 1 \\ 1 & -2 \end{smallmatrix}\right)$.
- $a_2 = 4, a_3 = 2$ gives $q_3 = 2$, $a_4 = 0$ and the matrix $Q_3 = \left(\begin{smallmatrix} 0 & 1 \\ 1 & -2 \end{smallmatrix}\right)$.
  The algorithm terminates in this step, since $a_4 = 0$.

So the greatest common divisor is

$$\gcd(14, 10) = a_3 = 2,$$

and

$$A_3 = Q_3 Q_2 Q_1 = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} -2 & 3 \\ 5 & -7 \end{pmatrix}.$$

The entries in the first row of A_3 satisfy

$$\gcd(14, 10) = 2 = (-2) \cdot 14 + 3 \cdot 10.$$

So set $s = -2$ and $t = -3$, then

$$\gcd(14, -10) = 2 = sa + tb = (-2) \cdot 14 + (-3) \cdot (-10).$$

Theorem 5.11 *A residue $a \in \mathbb{Z}_n$ is a unit if and only if* $\gcd(a, n) = 1$.

PROOF: If a is invertible modulo n, then there exist $s, t \in \mathbb{Z}$ such that $sa + tn = 1$. So every common divisor of a and n is a divisor of 1, including $\gcd(a, n)$. Since $\gcd(a, n)$ is positive, $\gcd(a, n) = 1$ follows.

Conversely, if $\gcd(a, n) = 1$, then we can use the Euclidean algorithm to compute $s, t \in \mathbb{Z}$ such that $sa + tn = 1$. But this means $sa = 1 \mod n$, or $s = a^{-1} \mod n$. Hence a is invertible. ■

Example 5.12 Let $a = 7$ and $n = 9$. Using the Euclidean algorithm, we find $\gcd(9, 7) = 1$, so 7 is invertible modulo 9. We also obtain the values $s = -3$ and $t = 4$ satisfying

$$1 = (-3) \cdot 9 + 4 \cdot 7.$$

So $4 = 7^{-1} \mod 9$.

Another application of the Euclidean algorithm is the solution of **linear Diophantine equations**: For given $a, b, c \in \mathbb{Z}$, find $x, y \in \mathbb{Z}$ solving

$$ax + by = c.$$

They are named after the Greek mathematician Diophantos of Alexandria who lived in the second century.

Theorem 5.13 *Let $a, b, c \in \mathbb{Z}$. The linear Diophantine equation*

$$ax + by = c$$

has solutions $x, y \in \mathbb{Z}$ if and only if $\gcd(a, b)$ divides c.

PROOF: If there are solutions $x, y \in \mathbb{Z}$, then $g = \gcd(a, b)$ divides $ax + by = c$ because it divides a and b.

Conversely, if c is a multiple of g, say $c = kg$, then we can use the Euclidean algorithm to find $s, t \in \mathbb{Z}$ such that $as + bt = g$. Then $x = sk$, $y = tk$ solves the original equation. ■

Once a solution $x_0, y_0 \in \mathbb{Z}$ is known, every other solution can be obtained by adding to x_0, y_0 a solution of the equation

$$ax' + by' = 0.$$

The integer solutions for the latter are given by

$$x' = \frac{kb}{\gcd(a, b)}, \quad y' = -\frac{ka}{\gcd(a, b)}, \quad \text{for any } k \in \mathbb{Z}.$$

So all solutions of the linear Diophantine equation $ax + by = c$ are given by

$$x = x_0 + \frac{kb}{\gcd(a, b)}, \quad y = y_0 - \frac{ka}{\gcd(a, b)}, \quad \text{for any } k \in \mathbb{Z}.$$

Example 5.14 Solve the linear Diophantine equation

$$39x + 69y = 1137.$$

Solution: Use the Euclidean algorithm to find $\gcd(39, 69) = 3$ and $s = -7$, $t = 4$ satisfying

$$3 = (-7) \cdot 39 + 4 \cdot 69.$$

Indeed, $3 \cdot 379 = 1137$, so by Theorem 5.14 a solution is given by

$$x_0 = 379 \cdot (-7) = -2281, \quad y_0 = 379 \cdot 4 = 1516.$$

Then

$$x = -2281 + 13k, \quad y = 1516 - 23k, \quad \text{for any } k \in \mathbb{Z}$$

are all solutions.

5.4 Simultaneous congruences

The Euclidean algorithm is the fundamental tool in modular arithmetic. It also proves helpful in solving a particularly interesting class of equations.

Let $q_1, \ldots, q_m \in \mathbb{Z}$ satisfying $\gcd(q_i, q_j) = 1$ for all $1 \leq i, j \leq m$. Numbers with this property are called **coprime** to each other. Given the set of congruences

$$\begin{aligned} x &\equiv a_1 \mod q_1 \\ &\vdots \\ x &\equiv a_m \mod q_m \end{aligned}$$

for some fixed values $a_1, \ldots, a_m \in \mathbb{Z}$, is there a solution $x \in \mathbb{Z}$ satisfying all these congruences? Equations of this type are called **simultaneous congruences**.

Before studying a procedure to solve this problem in general, let us look at a simple example:

Example 5.15 Find $x \in \mathbb{Z}$ satisfying the simultaneous congruences

$$\begin{aligned} x &\equiv 3 \mod 7, \\ x &\equiv 6 \mod 9. \end{aligned}$$

Try to find a solution by writing x in the following form:

$$x = 3 + 7y.$$

Then the first congruence is satisfied in any case. Now it remains to determine y such that the second congruence is satisfied as well:

$$3 + 7y \equiv 6 \mod 9 \quad \text{is equivalent to} \quad 7y \equiv 3 \mod 9.$$

As 7 and 9 are coprime, we can compute the inverse of 7 modulo 9 by means of the Euclidean algorithm. We already did this in Example 5.12: $1 = (-3) \cdot 9 + 4 \cdot 7$, so 4 is the inverse of 7 modulo 9. Multiplying both sides of the above congruence by 4, we obtain

$$y \equiv 4 \cdot 3 \equiv 3 \mod 9.$$

Hence

$$x = 3 + 3 \cdot 7 = 24$$

satisfies both congruences. It is not the only solution, though: Adding a multiple of $7 \cdot 9 = 63$ yields another solution, and the set of all solutions is

$$24 + 63\mathbb{Z} = \{24 + 63k \mid k \in \mathbb{Z}\}.$$

The procedure to solve a general system of simultaneous congruences follows the idea in the previous example.

Theorem 5.16 (Chinese Remainder Theorem) *Let $q_1, \ldots, q_m \in \mathbb{Z}$ be coprime to each other. Then there exist a solution to the simultaneous congruences*

$$\begin{aligned} x &\equiv a_1 \mod q_1 \\ &\vdots \\ x &\equiv a_m \mod q_m \end{aligned}$$

for given $a_1, \ldots, a_m \in \mathbb{Z}$. Set $q = q_1 \cdots q_m$. If x_0 is one solution, then the set of all solutions is given by

$$x_0 + q\mathbb{Z} = \{x_0 + kq \mid k \in \mathbb{Z}\}.$$

The problem of solving simultaneous congruences was first recorded by the Chinese mathematician Sunzi, probably in the 5th century, and the first general construction of a solution was given by Qin Jiushao in the 13th century. Hence the theorem's name.

The following algorithm constructs a solution for the simultaneous congruences and thereby proves the theorem. It generalises the approach used in Example 5.15.

Algorithm 5.17 (Solving simultaneous congruences) In the situation of Theorem 5.16, we are given the coprime numbers $q_1, \ldots, q_m \in \mathbb{Z}$ and the arbitrary values $a_1, \ldots, a_m \in \mathbb{Z}$.

We try to find a solution by writing x in the following form

$$x = x_1 + x_2 q_1 + x_3 q_1 q_2 + \ldots + x_k q_1 \cdots q_{k-1}$$

and determine the unknown x_i iteratively:

- In the first step,
$$x \equiv x_1 \equiv a_1 \mod q_1$$
must hold, so we may choose $x_1 = a_1$.

- Assume $x_1, \ldots, x_{i-1}$ have already been determined. Then
$$x \equiv x_1 + x_2 q_1 + x_3 q_1 q_2 + \ldots + x_i q_1 \cdots q_{i-1} \equiv a_i \mod q_i$$
must hold. We want to solve this for the unknown x_i. First,
$$x_i q_1 \cdots q_{i-1} \equiv a_i - x_1 - x_2 q_1 - x_3 q_1 q_2 - \ldots - x_{i-1} q_1 \cdots q_{i-2} \mod q_i.$$
All elements on the right-hand side are already known at this stage in the algorithm. We can therefore solve for x_i if the $q_1 \cdots q_{i-1}$ are invertible modulo q_i. As the $q_1, \ldots, q_k$ are coprime to each other, this is possible by Theorem 5.11, and the inverse can be computed with the Euclidean algorithm. Let s_i denote the inverse of $q_1 \cdots q_{i-1}$ modulo q_i. Set
$$x_i = (a_i - x_1 - x_2 q_1 - x_3 q_1 q_2 - \ldots - x_{i-1} q_1 \cdots q_{i-2}) s_i.$$

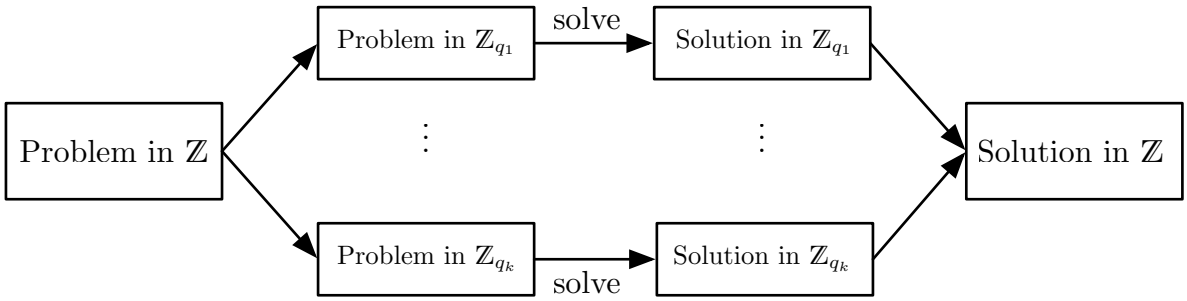
After k of these steps, the algorithm terminates and returns a solution x.

Proof of Theorem 5.16: The existence of a solution follows from Algorithm 5.17. If x, y are both solutions, then the difference $x - y$ satisfies
$$x - y \equiv 0 \mod q_j \quad \text{for } j = 1, \ldots, k.$$
So $x - y$ is a multiple of all the q_j and hence a multiple of their product $q = q_1 \cdots q_k$. Conversely, adding any multiple of q to x still gives a solution to the simultaneous congruences. Hence $x + q\mathbb{Z}$ is the set of solutions. ∎

The condition that the $q_1, \ldots, q_k$ are coprime is crucial. For if they are not, the system might not be solvable, as the following example shows.

Example 5.18 Let $q_1 = 14$, $q_2 = 10$ and consider the system of simultaneous congruences
$$x \equiv 2 \mod 14$$
$$x \equiv 5 \mod 10.$$
The first congruence requires $x = 2 + 14k$ for some $k \in \mathbb{Z}$. This means x must be an even number. But the second congruence requires $x = 5 + 10h$ for some $h \in \mathbb{Z}$. This means x must be an odd number. So there are no solutions for this system of congruences.

One application of the Chinese Remainder Theorem is **arbitrary-precision arithmetic**, which refers to the arithmetic of integers with hundreds of digits, as is often needed in cryptography or computer algebra. These numbers are much larger than those that the built-in arithmetic functions of a common processor can handle. Suppose we want to solve a computational problem involving large integers whose absolute value is bounded by some constant $q \in \mathbb{Z}$, which we assume to be much larger than what a processor can handle. We could solve the problem if we could reduce it to a problem with much smaller numbers. The Chinese Remainder Theorem allows us to do just that: If q (or number close to q) has a factorisation $q = q_1 \cdots q_k$ into mutually coprime factors which are much smaller than q (for example its prime factorisation), then we do our computations modulo q_1, modulo q_2,..., modulo q_k.



Once the solution modulo q_j has been found for all $q_1, \ldots, q_k$, we use Algorithm 5.17 to reconstruct the solution of the original problem. In addition, the "easy" computations modulo $q_1, \ldots, q_k$ can be done independently of each other, so we can perform them in parallel on different processors, thus gaining an additional speed-up.

5.5 Fermat's Theorem

Let p be a prime number. Since no number $a \in \{1, 2, \ldots, p-1\}$ can have p as a prime factor, $\gcd(a, p) = 1$ for any of these numbers.

Theorem 5.19 *For a prime number p, every non-zero element $1, 2, \ldots, p-1$ in $\mathbb{Z}_p$ is a unit. Moreover, the product of any two of these elements is again a unit in $\mathbb{Z}_p$.*

PROOF: The first statement follows immediately from Theorem 5.11 and the remark above. The product of any two non-zero numbers $k, h \in \mathbb{Z}_p$ has the inverse $h^{-1}k^{-1}$ and hence is a unit in $\mathbb{Z}_p$. ■

Theorem 5.20 (Fermat) *Let p be a prime number. Then for all $a \in \mathbb{Z}$ which are not multiples of p,*

$$a^{p-1} \equiv 1 \mod p.$$

PROOF: Since we assume that a is not a multiple of a and p is prime, $\gcd(a, p) = 1$. In particular, a is invertible modulo p (by Theorem 5.11).

Now compute modulo p: Consider the two lists of numbers

$$1, 2, 3, \ldots, p-1 \quad \text{and} \quad a, 2a, 3a, \ldots, (p-1)a.$$

The $p-1$ elements in the first list are precisely the units in $\mathbb{Z}_p$ (by Theorem 5.19). We show that the second list is identical to the first: Assume that $ka = ja \mod p$ for two numbers k, j from 1 to $p-1$. Because a is a unit, we can multiply both sides with a^{-1} and find $k = j$. This shows that all numbers in the second list are distinct, and by Theorem 5.19, they are all units. So the second list contains $p-1$ units. But there are $p-1$ units in $\mathbb{Z}_p$ in total, so the second list coincides with the first.

Since the two lists above coincide, multiplying their elements yields the same result:

$$1 \cdot 2 \cdots (p-1) = a \cdot 2a \cdots (p-1)a = a^{p-1} \cdot 1 \cdot 2 \cdots (p-1) \mod p.$$

Since all elements involved are units, we can cancel the product $1 \cdot 2 \cdots (p-1)$ on both sides and find

$$1 = a^{p-1} \mod p,$$

or expressed as congruences of integers: $a^{p-1} \equiv 1 \mod p$. ■

Fermat's Theorem allows us to test if a number q is *not* a prime number. If $a^{q-1} \equiv 1 \mod q$ is not satisfied for any $0 < a < q$, then q is not prime. Unfortunately, it is not valid to make the converse conclusion that q is prime if this holds for all $0 < a < q$: There are numbers satisfying this without being prime numbers, called **Carmichael numbers**. The three smallest Carmichael numbers are 561, 1105 and 1729.

Example 5.21 Let $q = 10$. We use Fermat's Theorem to prove that q is not a prime number: Test if there is some $a \in \{2, \dots 9\}$ with $a^{10-1} \neq 1 \mod 10$:

$$2^9 = 512 = 2 \neq 1 \mod 10$$

This would contradict Fermat's Theorem if 10 was a prime number.

6 Cryptography

Alice wants to send a secret message $\mathtt{m}$ to Bob. However, she cannot be sure that the channel on which $\mathtt{m}$ is to be sent is safe from eavesdropping. Therefore, she has to devise a method to encrypt her message in such a way that only Bob can decrypt and read it. The art of constructing and analysing such methods is known as **cryptography**. More precisely, one studies the following situation: Given is a set $\mathcal{M}$ of "plaintext" messages which are to be mapped to a set of "ciphertexts" $\mathcal{C}$ by means of an encryption function

$$\mathrm{enc} : \mathcal{M} \to \mathcal{C}.$$

The code $\mathcal{C}$ is supposed to unreadable for anyone without the proper authorisation. A legitimate recipient of a message on the other hand should be able to use a decryption function

$$\mathrm{dec} : \mathcal{C} \to \mathcal{M}$$

to obtain the original plaintext from the ciphertext. That is,

$$\mathrm{dec}(\mathrm{enc}(\mathtt{m})) = \mathtt{m}$$

holds.

Modern cryptography encompasses more than just secret communication. It deals with problems of message authentification, digital signatures, protocols for exchanging secrets and digital cash.

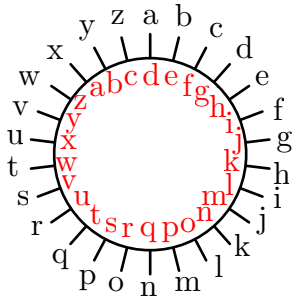
In this chapter we follow the convention that data which are to be kept secret are set in typewriter font $\mathtt{m}, \mathtt{k}, \ldots$.

6.1 Historical ciphers

Throughout history, cryptographic methods have been used mainly for military purposes. Before sophisticated mathematical methods and computational power became available in the 20th century, cryptographical methods were improvised and their effectiveness relied on keeping the method of encryption secret.

One of the earliest use of ciphers is attributed to Julius Caesar in the first century BC and is therefore known as **Caesar's cipher**. His method of

encrypting letters consists of replacing each letter in the alphabet by the third letter following it ("shifting by 3"), wrapping around at the end of the alphabet.



If we label the letters by numbers from 0 to 25, a shift by 3 is simply an addition of 3 modulo 26.

Example 6.1 A secret message encrypted by Caesar's cipher reads

wkhglhlvfdvw.

By shifting each letter backwards by 3, we decrypt it to

thedieiscast.

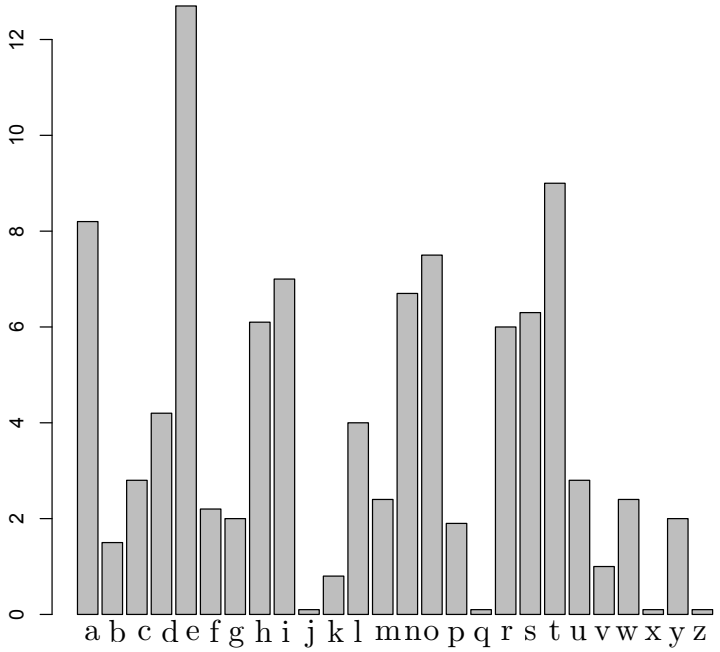
A generalisation of Caesar's cipher is the **shift cipher**. Here, we introduce an arbitrary number `k` as a secret key and shift the alphabet by `k` letters. Even if a spy learns that we use a shift cipher as encryption method, he still needs to know the secret key `k` to decrypt the message.

However, given that there are only 26 possible keys, the shift cipher is vulnerable to a **brute-force attack** which simply tries all possible keys until the right one is found.

Even if it is not known that we use a shift cipher, this method is still vulnerable to a **frequency-based attack**. Assuming that the encrypted message is a text in the English language, one can use a table of the letter frequency in English and compare the frequencies with those of the letters appearing in the encrypted text. For example, `e` is the most frequent letter in English, so if `h` is the most frequent letter in the encrypted text, it is reasonable to assume that `h` stands for `e` and so the shift is `k` = 3.

The relative frequency (in percent) of letters in the English language is given by the following table:

a	b	c	d	e	f	g	h	i	j	k	l	m
8.2	1.5	2.8	4.2	12.7	2.2	2.0	6.1	7.0	0.1	0.8	4.0	2.4
n	o	p	q	r	s	t	u	v	w	x	y	z
6.7	7.5	1.9	0.1	6.0	6.3	9.0	2.8	1.0	2.4	0.1	2.0	0.1



Exercise 6.1 Decrypt the following shift cipher-encrypted English text:

guerrcrbcyrpnaxrrcnfrpergvsgjbbsgurznerqrnq

More generally, a frequency-based attack works for all ciphers based on substituting every letter of the alphabet by a fixed other letter.

An improvement over the substitution ciphers was the use of multiple substitution rules for the same encryption. One such method was devised by Blaise de Vigenère in the 16th century and is named the **Vigenère cipher**

in his honour. Here, the secret key for the encryption is a finite sequence $\mathtt{k} = (\mathtt{k}_1, \ldots, \mathtt{k}_\mathtt{m})$ of $\mathtt{m}$ integer numbers. We then apply the shift cipher with key $\mathtt{k}_1$ to the first character of the secret message, the shift cipher with key $\mathtt{k}_2$ to the second character and so on, until we finally apply the shift cipher with key $\mathtt{k}_\mathtt{m}$ to the $\mathtt{m}$th character. For the $\mathtt{m}+1$st character, we use the key $\mathtt{k}_1$ again, and so on.

Example 6.2 Given the secret key $\mathtt{k} = (3, 2, 5, 8)$, the secret message

all glory to the hypnotoad

is encrypted as follows:

letter	a	l	l	g	l	o	r	y	t	o	t	h	e	h	y	p	n	o	t	o	a	d
shift	3	2	5	8	3	2	5	8	3	2	5	8	3	2	5	8	3	2	5	8	3	2
cipher	d	n	q	o	o	q	w	g	w	q	y	p	h	j	d	x	q	q	y	w	d	f

After removing whitespaces, the Vigenère cipher of with key $(3, 2, 5, 8)$ of the message is therefore

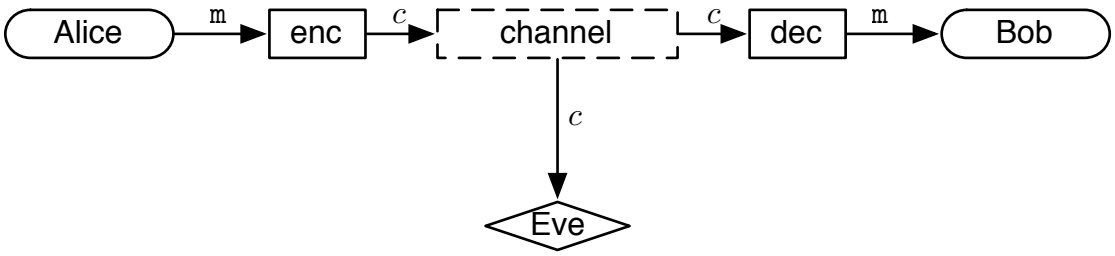
dnqooqwgwqyphjdxqqywdf.

Using multiple shifts and keeping the length of the key secret makes it hard to apply a frequency-based attack to the Vigenère cipher. Also, a brute-force attack is not feasible, since one would have to try all possible combinations of numbers $\mathtt{k}_i \in \{0, \ldots, 25\}$ for i from 1 to the unknown length $\mathtt{m}$. Instead of trying 26 possibilities as for the shift cipher, one would have to try 26^j possibilities for each number $j = 1, 2, \ldots, \mathtt{m}$.

The Vigenère cipher was a significant improvement over the shift ciphers, and it was not until the 19th century that an effective attack on it was devised. The idea is to look for repetitions of short combinations of letters. For example, using the key $\mathtt{k} = (3, 2, 5, 8)$ above, the word the will be encrypted by wkh, vjg, yph or bpm, depending on its relative position in the text. If the encrypted message is long enough, then it is likely that one of these combinations, say wkh, will appear quite often, since the is a very common word. So if two occurrences of wkh are found, the distance between these two occurrences is a multiple of the key length $\mathtt{m}$. The greates common divisor of all distances between repeated sequences yields the key length or a (in general much smaller) multiple of it. Once the key length $\mathtt{m}$ is known, the Vigenère cipher is once more vulnerable to a frequency-based attack on each of its shifts $\mathtt{k}_1, \ldots, \mathtt{k}_\mathtt{m}$.

6.2 Security and attacks

A fundamental assumption we make is that there is no such thing as a secure channel. This means we must always assume that data Alice sends to Bob and vice versa can be intercepted by an eavesdropper Eve. Any specification of a cryptographic system must take this into account.



All of the historical ciphers in Section 6.1 are **private-key encryptions**. This means Alice and Bob need to share a common secret, the **secret key**, in advance which they use for encrypting and decrypting the secret message. A problem is that they need to arrange for a meeting in person to share the secret key, because it cannot be transmitted via the insecure channel. The shortcomings of the historical methods teach us some of the things we need to consider for a systematic approach to cryptography.

Caesar's cipher is an example where the security of the encryption requires the method of encryption to remain secret (so the secret key is the encryption algorithm itself). However, the problems with this approach is obvious: Once the method becomes known via betrayel or negligence, it becomes worthless, and Alice and Bob need to devise a different method if they want to uphold their secret communication.

It is therefore wise to have the secret key `k` as an input parameter to the encryption algorithm rather than being the algorithm itself. Even if the algorithm becomes known, the Eve still needs to find `k` to make sense of an intercepted cipher, and even then Alice and Bob can use the same encryption method once they agree on a new secret key. This principle is known as **Kerckhoffs' principle**: *The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience.*

Even though the shift cipher with unknown shift length `k` as secret key obeys Kerckhoffs' principle, we have seen that this method is flawed because it is so easy to find the secret key by a brute-force attack. So Alice and Bob need a **key space** (the set from which they choose the secret key `k`) large enough to prevent Eve from finding `k` by an exhaustive search in a reasonable amount of time.

The Vigenére cipher obeys Kerckhoffs' principle and has a sufficiently large key space, given that the key is long enough. But the security of the Vigenére cipher was based only on the intuitive assumption and the experience that it was too hard to decrypt without knowing the secret key – until eventually a reliable method to break it was developed.

Modern cryptography is concerned with the systematic development of encryption schemes that are provably secure. This is guided by three principles:

1. The need for a *precise definition* of what "security" means. If we implement a security system, we need an understanding of what the system should achieve. Precise definitions are also important for comparing two different encryption schemes. Usually, the definition of security needs to specify against which type of attack (see below) the system should be secure, and what computational power we assume the adversary to have.

2. A *precise statement of the assumptions* that our scheme relies on. Modern cryptographic methods rely on assumptions that certain computational problems are very hard to solve, and breaking the encryption scheme at hand is at least as hard as one of these problems. Again, a precise statement of these assumptions is important for the comparison of different schemes, and also essential for proofs of security.

3. Using precise definitions and assumptions enables us to *rigorously prove the security* of an encryption scheme. If we have no proof that an adversary with a specified computational power can break the system, then we can only rely on our intuition that the system is secure. Very often a proof of security works by reducing the cryptographic problem to some computational problem, and the (assumed) hardness of the computational problem then proves the hardness of breaking the encryption scheme.

Note: A precise definition of security provides a *mathematical model* of the real world. If this model does not accurately reflect the real world, then even a rigorous proof that a system based on this model is secure (in the sense of the model) is useless. For example, the assumptions on the computational power of an adversary could be wrong (meaning the adversary has more computational power than expected), or there are unforseen types of attack on the encryption scheme.

The attacks on an encryption scheme usually fall into one of the following categories:

- In a **ciphertext-only attack** the eavesdropper intercepts one or more ciphertexts and tries to determine the plaintext message from them.
- In a **known-plaintext attack** the eavesdropper was able to learn the plaintext corresponding to a certain ciphertext and tries to determine the plaintext of some other ciphertext from it.
- In a **chosen-plaintext attack** the eavesdropper has the ability to obtain the encryption of a plaintext of its choice. It then tries to learn something about the encryption mechanisms from this in order to decrypt other ciphertexts.
- In a **chosen-ciphertext attack** the eavesdropper can get the decryption of any ciphertext it chooses. From this information it can attempt to recover the key used for encryption.

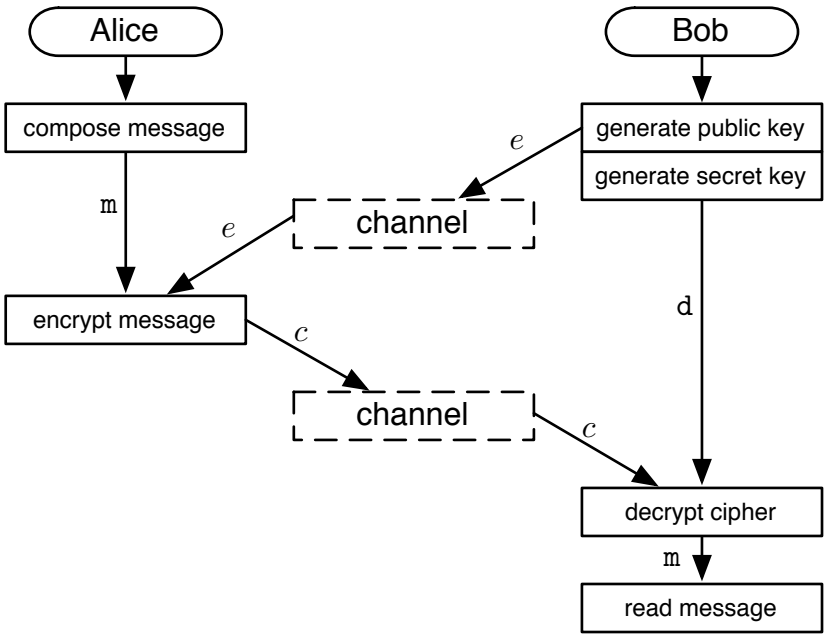
Example 6.3 The frequency-based attack on the shift cipher in Section 6.1 is a ciphertext-only attack. All of the historical ciphers in Section 6.1 can be immediately broken if the adversary is able to carry out a known-plaintext attack.

A different kind of attack is a **side channel attack**, which is directed at the physical implementation of a cryptographic system rather than its encryption method. For example, by observing the power usage of a smart-card during an encryption process, an adversary can try to read out the secret key or even the plaintext message directly, even though the encryption scheme has been proven to be secure.

6.3 Public-key cryptography

An obvious problem with an encryption scheme where Alice and Bob use the same secret key is that they need to find a way to generate this key without braodcasting it via the insecure channel. So they either must meet in person or use a trusted messenger service to share the key. This may be possible for military or intelligence organisations, but in modern every-day applications of cryptography such as online banking or email encryption there can be hundreds of participants, so this is simply not a realistic option.

It is therefore the philosophy of **public-key cryptography** to use a key consisting of two components, the **public key** e for encryption and the **secret key** `d` for decryption. This way, each Bob can share his public key e with Alice, who uses e to encrypt a message `m` to Bob. Then Alice sends the cipher c to Bob, who uses his secret key `d` to recover the message `m`.



The idea of public-key cryptography was first introduced by Whitfield Diffie and Martin Hellman in 1976. It was the first step of making high quality cryptography available to the public domain.

6.4 RSA

In 1977, the first public-key encryption algorithm was published by Ron Rivest, Adi Shamir and Leonard Adleman. It is called **RSA** in their honour and remains the most widely used encryption scheme to date.

Like most public-key encryption schemes, RSA uses the modular arithmetic from Chapter 5 to realise the encryption and decryption functions. Our assumption here is that we can encode the message $\mathtt{m}$ by a positive integer less or equal to a certain integer number n.

Algorithm 6.4 (RSA encryption) Alice wants to send a message $\mathtt{m}$ to Bob.

(i) Bob picks two secret large prime numbers $\mathtt{p}, \mathtt{q}$. Set $n = \mathtt{pq}$.

(ii) Bob picks a number e coprime to $(\mathtt{p}-1)(\mathtt{q}-1)$ and publishes the pair (n, e) as the public key. For the secret key, he uses the Euclidean algorithm to determine a number $\mathtt{d}$ satisfying

$$e\mathtt{d} \equiv 1 \mod (\mathtt{p}-1)(\mathtt{q}-1).$$

(iii) Alice encrypts the message $\mathtt{m} \in \mathbb{Z}_n$ by means of the public data as follows:

$$\mathrm{enc} : \mathbb{Z}_n \to \mathbb{Z}_n, \quad \mathtt{m} \mapsto \mathtt{m}^e \mod n.$$

She then sends the encrypted message $c = \mathtt{m}^e$ to Bob.

(iv) Bob decrypts the ciphertext c using his secret key $\mathtt{d}$:

$$\mathrm{dec} : \mathbb{Z}_n \to \mathbb{Z}_n, \quad c \mapsto c^{\mathtt{d}} \mod n.$$

Then

$$\mathrm{dec}(\mathtt{m}^e) = \mathtt{m}^{e\mathtt{d}} = \mathtt{m}.$$

To convice ourselves that RSA works, it remains to prove the correctness of the last statement.

Theorem 6.5 *In the situation of Algorithm 6.4,*

$$\mathrm{dec}(\mathrm{enc}(x)) = x$$

holds for all $x \in \mathbb{Z}_n$.

PROOF: Let $x \in \mathbb{Z}_n = \{0, \ldots, n-1\}$, where $n = \mathtt{pq}$. Set

$$x_1 = x \mod \mathtt{p}$$
$$x_2 = x \mod \mathtt{q}.$$

If $x_1 = 0$, then trivially $x_1^{e\mathtt{d}} = x_1 \mod \mathtt{p}$. If $x_1 \neq 0$, then x_1 is a unit in $\mathbb{Z}_\mathtt{p}$ by Theorem 5.19. By assumption, $e\mathtt{d} \equiv 1 \mod (\mathtt{p}-1)(\mathtt{q}-1)$, so in particular

$$e\mathtt{d} = 1 + k(\mathtt{p}-1) \quad \text{for some } k \in \mathbb{Z}.$$

Now it follows from Fermat's Theorem that

$$x_1^{e\mathtt{d}} = x_1^{1+k(\mathtt{p}-1)} = x_1(x_1^{\mathtt{p}-1})^k = x_1 \mod \mathtt{p}.$$

In the same way we see that $x_2^{e\mathtt{d}} = x_2 \mod \mathtt{q}$. It follows that $x^{e\mathtt{d}}$ solves the simultaneous congruences

$$x^{e\mathtt{d}} \equiv x_1^{e\mathtt{d}} \equiv x_1 \equiv x \mod \mathtt{p}$$
$$x^{e\mathtt{d}} \equiv x_2^{e\mathtt{d}} \equiv x_2 \equiv x \mod \mathtt{q}.$$

Obviously, x is another solution. According to the Chinese Remainder Theorem, all integer solutions differ only by a multiple of $\mathtt{pq} = n$, so

$$x = x^{e\mathtt{d}} \mod n.$$

This proves $\mathrm{dec}(\mathrm{enc}(x)) = x$ for all $x \in \mathbb{Z}_n$. ∎

Example 6.6 Alice wants to send the secret message

all glory to the hypnotoad

to Bob. She encodes the message by replacing each letter a by 01, b by 02, ..., z by 26 and whitespace by 27. Then encoded message m is

112122707121518252720152720080527082516141520150104

where the leading 0 was discarded.

(i) Bob picks the secret prime numbers

$$\mathtt{p} = 10000000000000000000000000000049,$$
$$\mathtt{q} = 10000000000000000000000000000061.$$

Then

$$n = \mathtt{pq} = 100000000000000000000000000000055100000000000000000000000000002989.$$

(ii) Bob picks a number e coprime to $(\mathtt{p}-1)(\mathtt{q}-1)$, say

$$e = 10000000019,$$

and then publishes the public key (n, e).

He then computes the inverse of 10000000019 modulo n,

$$\mathtt{d} = 25040872412422342416397549408858453643894398076600643654458777805.$$

as his secret key.

(iii) Alice uses Bob's public key (n, e) to encrypt the message The resulting cipher c is

$$c = 11490763157416587805346557907847713846134752980299732303983238094.$$

Alice then sends c to Bob.

(iv) Bob receives c and decrypts using the secret key $\mathtt{d}$:

$$c^{\mathtt{d}} = 11212270712151825272015272008052708251614152015010 4.$$

This way Bob recovers the message $\mathtt{m}$.

The security of RSA encryption depends on the difficulty of computing the eth root of an integer modulo n: If there exists an *efficient* algorithm $\text{modroot}_{e,n}$ to compute the eth root modulo n, then Eve can intercept the cipher c and use this algorithm to determine $\text{modroot}_{e,n}(c) = \mathtt{m}$. This problem is known as the **RSA problem**.

At present, the only known way to solve the RSA problem is to compute the prime factorisation $n = \mathtt{pq}$ and then use the Euclidean algorithm to determine Bob's secret key $\mathtt{d}$ and then decrypt c by raising to the power of $\mathtt{d}$. So *under the assumption* that computing the prime factorisation of n cannot be done in a reasonable amount of time (meaning less than several decades), RSA encryption is secure.

So how can we determine whether this assumption is justified? If b is the number of bits in the binary representation of n, then we say that an algorithm can factorise a number in **polynomial time** if the number of computational steps which this algorithm performs is a polynomial function of b.

Polynomial-time algorithm are considered efficient, since the time they need for a computation grows only moderately as the number b of bits gets larger and larger. For algorithms that factorise a number in **exponential time** the number of steps is an exponential function in b. For exponential-time algorithms the computational effort grows considerably for larger b. The currently known methods to compute prime factorisations of integers are all exponential time algorithms. With these algorithms, it would take a single core processor about 2000 years to compute the prime factorisation of a 768-bit integer. In view of this, RSA can be considered secure.

6.5 Attacks on RSA

As we have seen, RSA is secure in principle as long as the RSA problem is too hard to solve in practice. However, the security of the simple version of RSA in Algorithm 6.4 can be compromised if the implementation is careless. In this section we discuss some possible ciphertext-only attacks on flawed RSA implementations.

The first attack is possible if *the public key e is too small.*

Example 6.7 Suppose Bob uses the key $e = 3$ for encryption is $e = 3$, and the encryption modulus n is very large compared to e, say $n = 20000499997$. Suppose Alice sends the secret message $\mathtt{m} = 809$. Since this message is smaller than $\sqrt[3]{n} \approx 2714$, the encrypted message $c = \mathtt{m}^e = 809^3 = 529475129$ is smaller than n. So taking the third power of $\mathtt{m}$ modulo n is the same as taking $\mathtt{m}^3$ over the integers. Since n and e are public, Eve can expect this and intercept the message. She can then easily decrypt it by computing $\mathtt{m} = c^{\frac{1}{3}}$ *over the integers* rather than modulo n. This is a much easier computational task than the RSA problem.

A similar attack on RSA with a small public key e is possible when there are multiple recipients of the same message $\mathtt{m}$. This attack also works if $\mathtt{m}$ is not smaller than $\sqrt[3]{n}$.

Example 6.8 Again, suppose $e = 3$ and the message $\mathtt{m}$ is sent by Alice to at

least three different receivers with public keys

$$(n_1, e) = (1022117, 3)$$
$$(n_2, e) = (1040399, 3)$$
$$(n_3, e) = (1223227, 3).$$

Eve intercepts the three encryptions $c_1 = 18523$, $c_2 = 952437$ and $c_3 = 1041065$ of the same secret message $\mathtt{m}$. In particular, this means

$$\mathtt{m} < \min\{c_1, c_2, c_3\} = 18523.$$

Moreover, Eve can assume that n_1, n_2 and n_3 are coprime to each other (if not, she could compute a common prime factor $\mathtt{p}$ of, say, n_1, n_2 using the Euclidean algorithm and then easily factor $n_1 = \mathtt{pq}_1$). The three intercepted ciphers give rise to a system of simultaneous congruences

$$\hat{c} \equiv 18523 = \mathtt{m}^3 \mod 1022117$$
$$\hat{c} \equiv 952437 = \mathtt{m}^3 \mod 1040399$$
$$\hat{c} \equiv 1041065 = \mathtt{m}^3 \mod 1223227.$$

Set $\hat{n} = n_1 n_2 n_3 = 1300791218184872041$. Using Algorithm 5.17, Eve can solve this system and find a solution $\hat{c} = 529475129$ in $\mathbb{Z}$ (if she finds a solution $\hat{c}$ larger than $\hat{n}$, the Chinese Remainder Theorem allows her to replace it by a solution less than $\hat{n}$ by subtracting a suitable multiple of $\hat{n}$). This means

$$\hat{c} = 529475129 = \mathtt{m}^3 \mod 1300791218184872041.$$

Since $\mathtt{m} < \min\{n_1, n_2, n_3\}$, we have $\mathtt{m}^3 < \hat{n}$, so in fact

$$\hat{c} = \mathtt{m}^3$$

over the integers. Eve can easily take the third root and find the secret message

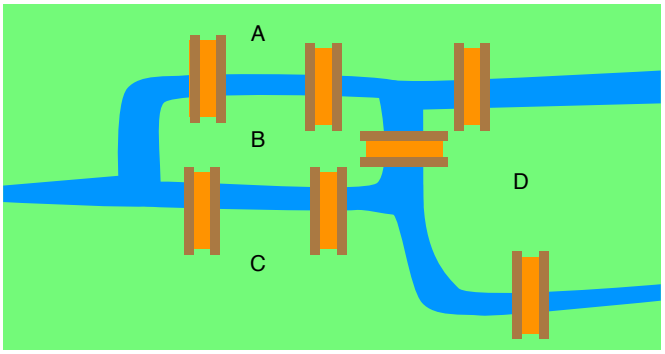
$$\mathtt{m} = \sqrt[3]{\hat{c}} = \sqrt[3]{529475129} = 809.$$

To prevent attacks of the types discussed above, a proper RSA implementation uses **padding**, which means the cipher c is inflated to a large enough size by augmenting it with random bits.

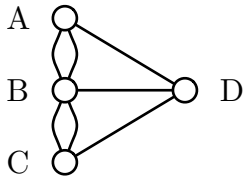
7 Graphs and trees

Graphs are among the most versatile data structures in discrete mathematics and computer science. Loosely speaking, they represent collections of objects of the same type with a certain relation between some of these objects. We often visualise graphs by drawing the objects as dots or circles and indicating the relations by lines between the related objects.

Example 7.1 A classical example from graph theory is the **Königsberg bridge problem** originally posed (and solved) by Leonhard Euler in 1735: The city of Königsberg is set on both sides of river Pregel. There are two large islands in the river which (in Euler's time) were connected to the mainland by seven bridges.



Euler asked if it is possible to find a walk through Königsberg, starting and finishing at the same place and crossing each bridge exactly once.



The essential information of the problem can be encoded in the above graph: There is one circle representing each part of the city and the lines represent the bridges connecting the different parts. We will return to this example and answer Euler's question later.

7.1 Properties and representations of graphs

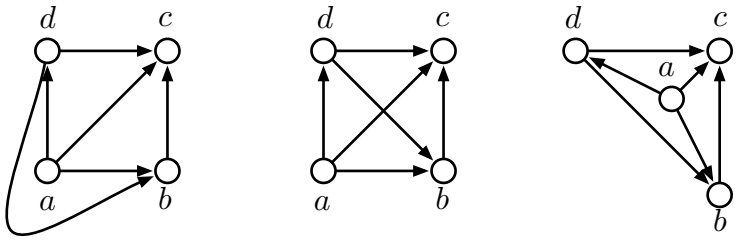
A **directed graph** (also called a **digraph**) is a pair $G = (V, E)$ consisting of a set V of **vertices** together with a set E of **edges** such that

$$E \subseteq V \times V.$$

We say G is **finite** if V is a finite set (then E automatically is also finite).

The intuition behind this definition is that the vertices $v \in V$ represent the objects (represented by dots or circles) in the graph, and each edge $(v, w) \in E$ (represented by an arrow from v to w) indicates that v is in some kind of relation to w.

Example 7.2 Consider the graph G with vertices $V = \{a, b, c, d\}$ and edges $E = \{(a, b), (a, c), (a, d), (b, c), (d, b), (d, c)\}$. We can represent G graphically in different ways:



Another way to represent a finite directed graph G with vertices $v_1, \ldots, v_n$ is by its **adjacency matrix**, which is the $n \times n$-matrix A with coefficients a_{ij} defined as

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}.$$

That is, A has an entry 1 in row i, column j precisely if there is an edge from v_i to v_j. Otherwise, the entry is 0.

Example 7.3 In Example 7.2, G has 4 vertices $v_1 = a$, $v_2 = b$, $v_3 = c$ and $v_4 = d$. The adjacency matrix A of G is the 4×4-matrix

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

Exercise 7.1 Suppose S is a set and $\mathcal{R} \subseteq S \times S$ is a binary relation on S. Show that to this relation there corresponds a unique graph G with vertex set $V = S$. Conversely, every graph G with vertex set V gives rise to a binary relation on V.

A **loop** is a edge (v, v) from a vertex v to itself. A graph without loops is called **loop-free**.

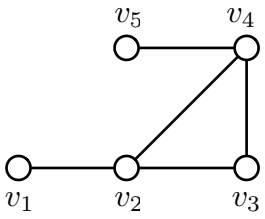
Let $G = (V, E)$ be a directed graph. The **out-degree** of a vertex $v \in V$ is the number of edges emerging from v (meaning those edges (v, w) for some $w \in V$), and the **in-degree** of $v \in V$ is the number of edges ending at v (meaning those edges (w, v) for some $w \in V$).

Often we do not want to distinguish between an edge from v to w and an edge from w to v. In this case we assume that for all vertices $v, w \in E$, $(v, w) \in E$ automatically implies $(w, v) \in E$. A graph with this property is called an **undirected graph**.

For undirected graphs, we consider (v, w) and (w, v) as one and the same edge and count it only once: Let $G = (V, E)$ be an undirected graph. The **degree** $\deg(v)$ of a vertex $v \in V$ is the number of edges emerging from v.

In an undirected graph we draw the edges as lines rather than arrows.

Example 7.4 Consider the following undirected graph:



The degrees of the vertices are: $\deg(v_1) = 1$, $\deg(v_2) = 3$, $\deg(v_3) = 2$, $\deg(v_4) = 3$, $\deg(v_5) = 1$.

Theorem 7.5 (Handshaking Lemma) *Let $G = (V, E)$ be an undirected loop-free finite graph with vertices $v_1, \ldots, v_n$. If $|E| = m$, then*

$$\sum_{i=1}^{n} \deg(v_i) = 2m.$$

PROOF: We count the elements of the set

$$X = \{(v, e) \mid v \in V, e \in E \text{ such that } e = (v, w) \text{ for some } w \in V\}$$

in two ways:

1. There are m choices to pick e, and each $e = (v, w)$ appears twice in the set X, first in (v, e) and then in (w, e). So there are $2m$ elements in the set X.

2. For every vertex v_i, there are $\deg(v_i)$ edges e emerging from v_i which appear in X. Taking the sum for all vertices, we find that X contains $\sum_{i=1}^{n} \deg(v_i)$ elements.

Hence $2m = |X| = \sum_{i=1}^{n} \deg(v_i)$. ∎

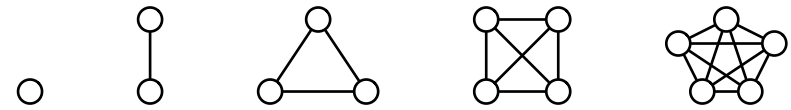
Convention: Unless stated otherwise, whenever we speak of a *graph*, we mean an undirected finite graph.

Exercise 7.2 Let G be a finite graph. Show that the following are equivalent:

(a) G is an undirected graph.
(b) The adjacency matrix of A is symmetric (meaning $A = A^\top$).
(c) The relation $\mathcal{R}$ corresponding to a graph G (see Exercise 7.1) is symmetric.

A graph $G = (V, E)$ is **complete** if $E = V \times V$, that is, E contains all possible edges between pairs of distinct vertices. The complete graph with n vertices is denoted by K_n.

Example 7.6 The complete graphs K_n for $n = 1, \ldots, 5$ are:

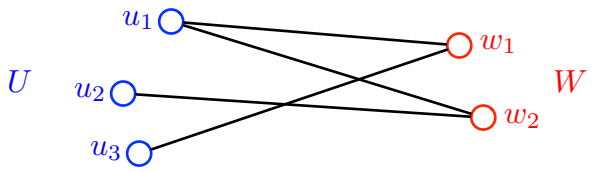


K_1 K_2 K_3 K_4 K_5

Exercise 7.3 Show that the complete graph K_n has $\binom{n}{2}$ edges.

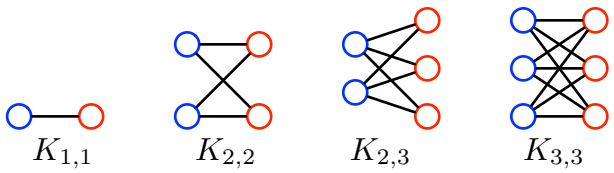
An undirected graph $G = (V, E)$ is called **bipartite** if we can partition V into two subsets U, W (meaning $V = U \cup W$ and $U \cap W = \varnothing$) such that any edge $(v_1, v_2) \in E$ has $v_1 \in U$ and $v_2 \in W$. Note that in a bipartite graph, there are no edges within U and no edges within W.

Example 7.7 The graph $G = (V, E)$ with vertex set $V = U \cup W$, where $U = \{u_1, u_2, u_3\}$ and $W = \{w_1, w_2\}$, and edges $(u_1, w_1), (u_1, w_2), (u_2, w_2), (u_3, w_1)$ is bipartite.



If G is a bipartite graph and $E = U \times W$, then G is a **complete bipartite graph** $K_{n,m}$ where $|U| = m$, $|W| = n$. This means all possible edges between U and W appear.

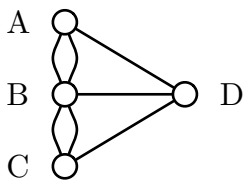
Example 7.8 Some complete bipartite graphs $K_{m,n}$ for small m, n:



Exercise 7.4 Show that the complete bipartite graph $K_{m,n}$ has mn edges.

Sometimes we want to allow a (directed or undirected) graph to have more than one edge between the same vertices v and w. Such a graph is called a **multigraph**. For a multigraph $G = (V, E)$, we define the set of edges as a subset E of $V \times V \times \mathbb{N}$, where $e = (v, w, n)$ means that e is the nth edge from v to w.

Example 7.9 The graph in the Königsberg bridge problem is an undirected multigraph:



The set E contains the edges $(\mathrm{A}, \mathrm{B}, 1)$, $(\mathrm{A}, \mathrm{B}, 2)$, $(\mathrm{A}, \mathrm{D}, 1)$, $(\mathrm{B}, \mathrm{C}, 1)$, $(\mathrm{B}, \mathrm{C}, 2)$, $(\mathrm{B}, \mathrm{D}, 1)$, $(\mathrm{C}, \mathrm{D}, 1)$.

7.2 Paths and circuits

Let $G = (V, E)$ be a (directed or undirected) graph and $v, w \in V$ vertices in G. A **walk of length k** in G is a sequence of vertices

$$v = v_0, v_1, v_2, \ldots, v_{k-1}, v_k = w$$

such that for all $i = 0, \ldots, k-1$

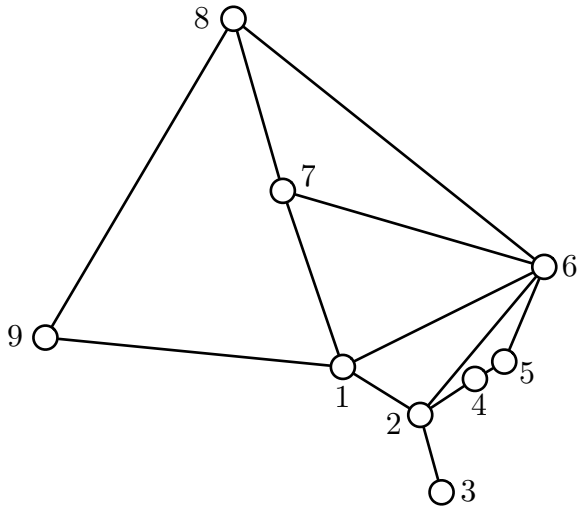
$$(v_i, v_{i+1}) \in E.$$

If the startpoint and endpoint in a walk are identical, $v = w$, then it is called a **circuit**.

Note that the definition of a walk allows some odd behaviour, such as going back and forth between two points along the same edge multiple times. A walk is called **simple** if no edge in it appears twice.

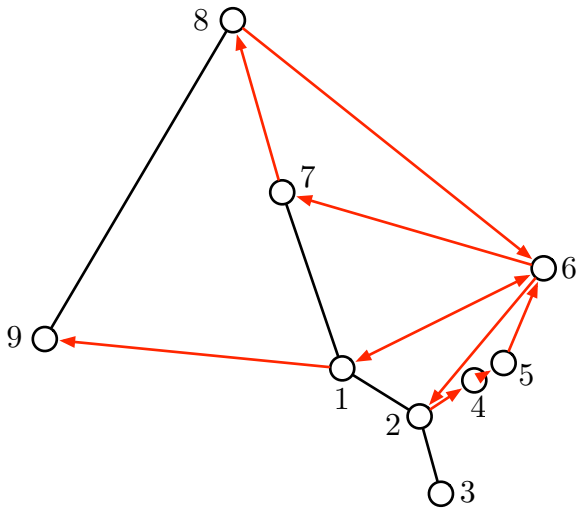
A **path** is a walk in which no vertex in it appears twice (in particular, it is simple), and a circuit in which no vertex other than the endpoints appears twice is called a **cycle**.

Example 7.10 Consider the following graph:



A walk of length 10 in this graph is

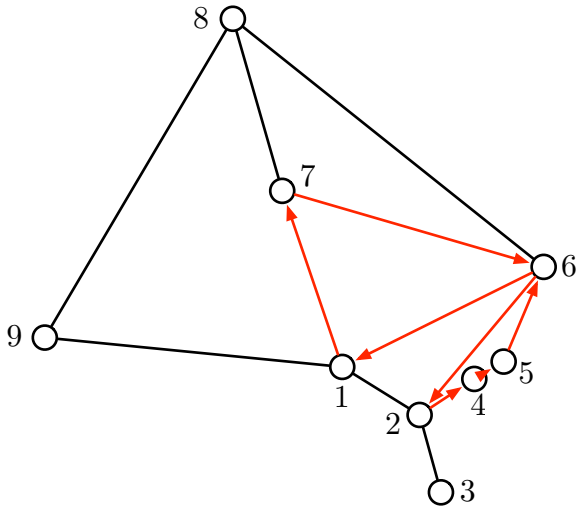
$$1, 6, 7, 8, 6, 2, 4, 5, 6, 1, 9.$$



Note that the edge $(1, 6)$ is traversed twice, so this walk is not simple.

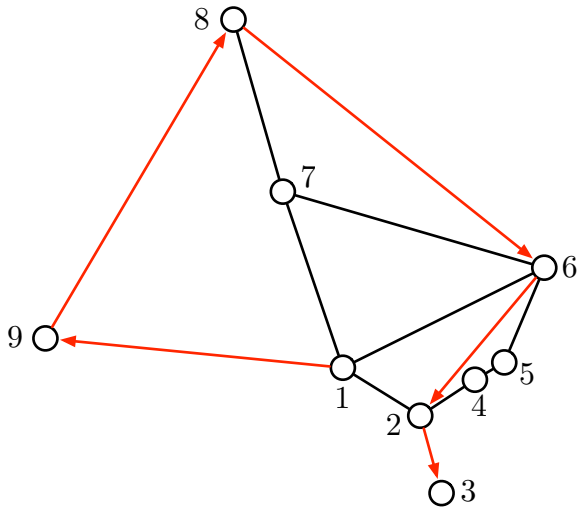
A circuit of length 7 which is simple but not a cycle is

$$1, 7, 6, 2, 4, 5, 6, 1.$$



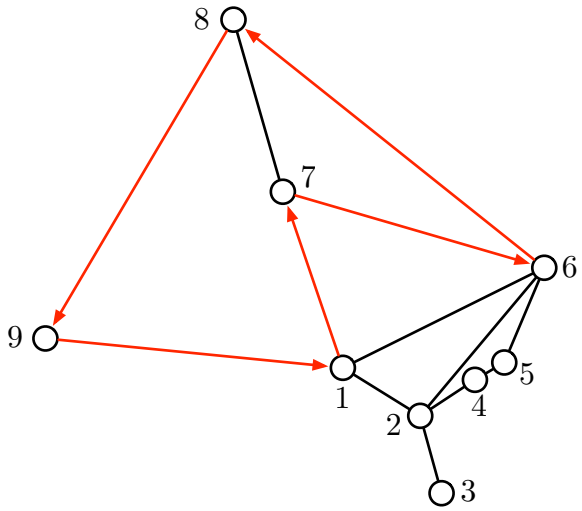
A path of length 5 from 1 to 3 is

$$1, 9, 8, 6, 2, 3.$$



A cycle of length 5 beginning and ending at 1 is

$$1, 7, 6, 8, 9, 1.$$



Exercise 7.5 Show that if $G = (V, E)$ contains a walk from v to w of length k, then it contains a path from v to w of length $\leq k$.

Theorem 7.11 *Let $G = (V, E)$ be a (directed or undirected) finite graph with n vertices and adjacency matrix A. If $k \geq 1$ and $a_{ij}^{(k)}$ denotes the entry in row i, column j of the kth power A^k, then $a_{ij}^{(k)}$ is the number of walks of length k from v_i to v_j.*

PROOF: Induction on $k \geq 1$:

Initial case: $A^1 = A$, and a walk of length 1 consist of just one edge. The coefficient $a_{ij}^{(1)} = a_{ij}$ of the adjacency matrix is 1 precisely if there is an edge from v_i to v_j.

Induction hypothesis: For some number $k \geq 1$, the entry $a_{ij}^{(k)}$ of A^k is the number of walks from v_i to v_j.

Induction step: $A^{k+1} = AA^k$, so by the formula for matrix multiplication,

$$a_{ij}^{(k+1)} = a_{i1}a_{1j}^{(k)} + a_{i2}a_{2j}^{(k)} + \ldots + a_{in}a_{nj}^{(k)}.$$

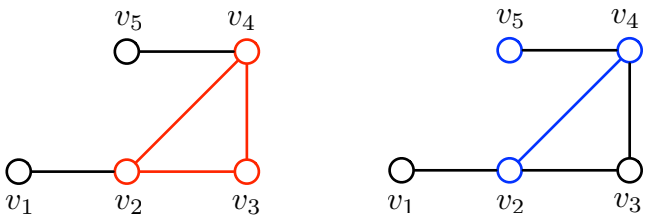
Look at the first summand: $a_{i1} = 1$ if there is an edge from v_i to v_1, and in that case there $a_{1j}^{(k)}$ walks of length k leading from v_1 to v_j (by the induction hypothesis). So combining the edge (v_i, v_1) with any of these walks gives us all the walks from v_i to v_j with v_1 as the second vertex.

On the other hand, if $a_{i1} = 0$ then the product $a_{i1}a_{1j}^{(k)}$ is 0. This reflects the fact that there is no edge and hence no walk from v_i to v_1, and so there cannot be a walk from v_i to v_j with v_1 as second vertex.

In the same way the following summands $a_{im}a_{mj}^{(k)}$ count the number of walks from v_i to v_j with v_m as the second vertex. Thus their sum is the number of walks from v_i to v_j. ■

A **subgraph** $G' = (V', E')$ of $G = (V, E)$ is a graph such that $V' \subseteq V$ and $E' \subseteq \{(v, w) \in E \mid v, w \in V'\}$.

Example 7.12 Two subgraphs of the graph from Example 7.4:



Example 7.13 A path $v_0, v_1, \ldots, v_n$ in G can be identified with a subgraph G' containing all its vertices $V' = \{v_0, \ldots, v_n\}$ and the edges $E' = \{(v_0, v_1), \ldots, (v_{n-1}, v_n)\}$ between them.

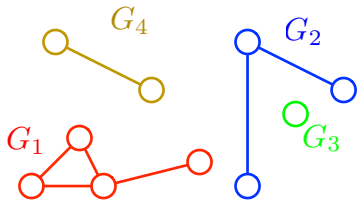
A graph $G = (V, E)$ is **connected** if each pair of vertices is joined by a path.

Given a graph $G = (V, E)$, we can partition the set of vertices into disjoint sets of common vertices

$$V = V_1 \cup V_2 \cup \ldots \cup V_p$$

where $v, w \in V_i$ if and only if v, w are connected by a path. Let E_i denote the subset of E containing the edges between vertices in V_i. The subgraphs $G_i = (V_i, E_i)$ are called the **connected components** of G.

Example 7.14 A graph with four connected components.



Exercise 7.6 Let $G = (V, E)$ be a graph with connected components $G_1, \ldots, G_r$. Let $n_i = |V_i|$. Show that for a suitable labelling of the vertices, the adjacency matrix A of G takes the form

$$A = \begin{pmatrix} \boxed{A_1} & & \\ & \ddots & \\ & & \boxed{A_r} \end{pmatrix},$$

where A_i is an $n_i \times n_i$-matrix and all other entries in A are 0.

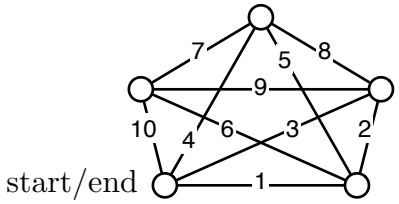
In this example we were looking for a walk on the multigraph which uses each edge exactly once (in particular it must be a simple walk) and returns to the starting vertex (so it must be a circuit). A walk with these properties is called an **Euler circuit**.

Theorem 7.15 *If a connected loop-free finite multigraph $G = (V, E)$ has an Euler circuit, then all its vertices have even degree.*

PROOF: Follow the Euler circuit from vertex to vertex and delete each edge once it has been traversed. This way, for each vertex v_i ($i \geq 1$) in the Euler circuit, we erase one edge (v_{i-1}, v_i) when we arrive at this vertex and one edge (v_i, v_{i+1}) when we depart from this vertex (because we excluded the existence of loops). This reduced the degree of v_i by 2. For the start vertex v_0, we erase the first and last edge of the circuit. Eventually, all edges have been deleted and all vertices have degree 0, so they must have even degree to begin with. ■

It follows that the Königsberg bridge problem has no solution, since there is a vertex with degree 3.

Example 7.16 The complete graph K_5 admits an Euler circuit. The numbers on the edges indicate the order in which they are traversed.

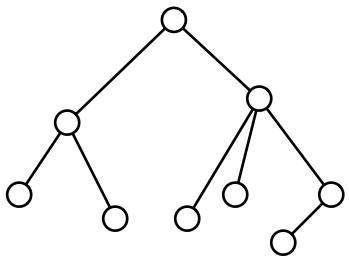


Note that all vertices have even degree 4.

7.3 Trees

A **tree** is a connected graph without cycles. A **forest** is a graph whose connected components are trees.

Example 7.17 A tree.



Theorem 7.18 *If v, w are vertices in a tree $T = (V, E)$ then there is a unique path from v to w.*

PROOF: Since T is connected there is at least one path from v to w. If there were two distinct paths from v to w, then we could traverse the first path from v to w and then the second path backwards from w to v, thus obtaining a circuit. This circuit contains a cycle since v and w are distinct, contradicting the definition of a tree. ■

Theorem 7.19 *In a tree $T = (V, E)$*

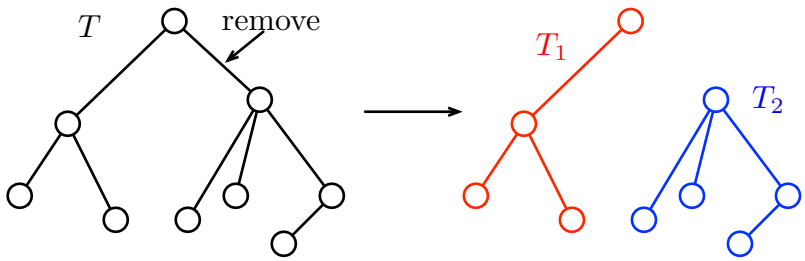
$$|V| = |E| + 1.$$

PROOF: Induction on $|E| = m \geq 0$.

Initial case: For a tree with $m = 0$ edges, there can be only one vertex.

Induction hypothesis: The result is true for all trees with $k \leq m$ edges.

Induction step: Let $T = (V, E)$ be a tree with $m + 1$ edges. Pick any edge $(v, w) \in E$ and remove it.



This splits T into two subtrees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ such that

$$|V| = |V_1| + |V_2|, \quad |E| = 1 + |E_1| + |E_2|.$$

Since both trees T_1 and T_2 have $< m + 1$ edges, the induction hypothesis implies

$$|V_1| = |E_1| + 1, \quad |V_2| = |E_2| + 1.$$

Plugging this into the previous equation for $|V|$, we find

$$|V| = |E_1| + 1 + |E_2| + 1 = (1 + |E_1| + |E_2|) + 1 = |E| + 1,$$

as required. ■

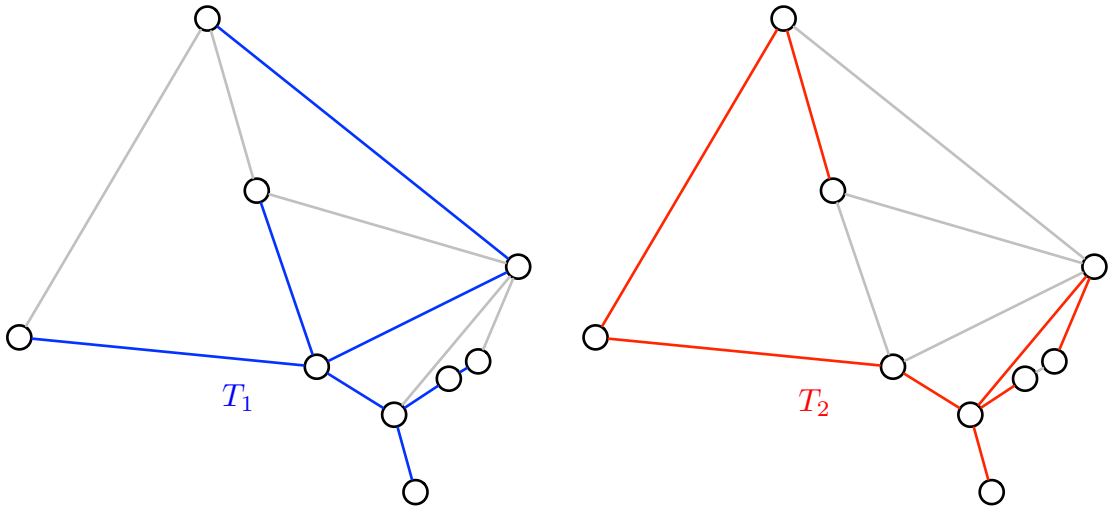
7.4 Spanning trees

We are interested in smallest sets of edges in a graph G that connect all the vertices.

A **spanning tree** of a connected graph $G = (V, E)$ is a subgraph $T = (V, E')$ which is a tree and contains every vertex of G.

A spanning tree is obtained by removing some edges but no vertices of G. Spanning trees are not unique.

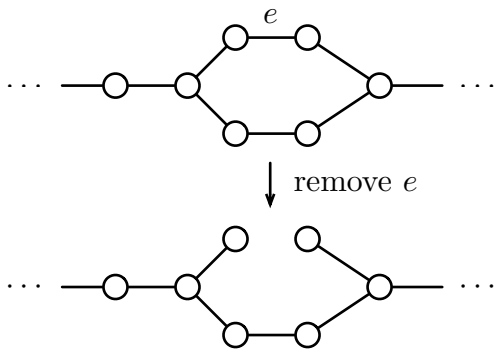
Example 7.20 Two spanning trees of the graph from Example 7.10.



Theorem 7.21 *A finite connected graph $G = (V, E)$ has a spanning tree.*

PROOF: If G has no cycle, then G itself is a tree, and $T = G$.

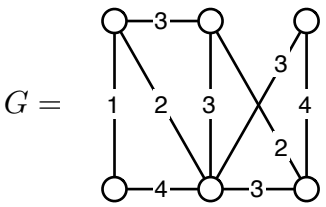
Otherwise, choose a cycle in G and let e be an edge in that cycle. By removing e from the graph, we obtain a subgraph $G' = (V, E')$ with one less cycle and the same vertices.



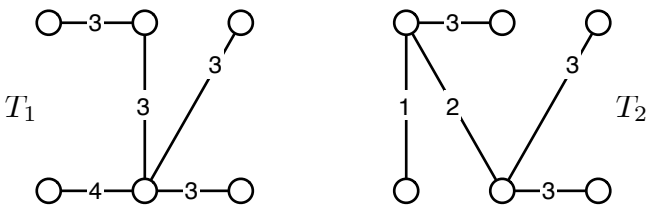
Repeat this procedure with G' until no more cycles are left. What remains is a tree T with vertex set V. ■

The question of spanning trees is especially interesting if the edges in G are **weighted**, that is, if each edge e is assigned a non-negative **weight** $\mathrm{w}(e)$. The weight $\mathrm{w}(G')$ of a subgraph G' of G is the sum of the weights of the edges in G'.

Example 7.22 Consider the following graph G. The numbers on the edges indicate the weights of the edges.



The following trees T_1 and T_2 are both spanning trees of G.



Note that $\mathrm{w}(T_1) = 16$, whereas $\mathrm{w}(T_2) = 12$.

The problem is to find a **minimum spanning tree** T, one for which $\mathrm{w}(T)$ is minimal among all spanning trees of G.

The following algorithm constructs a minimum spanning tree for a weighted graph.

Algorithm 7.23 (Kruskal's algorithm) Given a weighted graph $G = (V, E)$ with m edges $e_1, \ldots, e_m$.

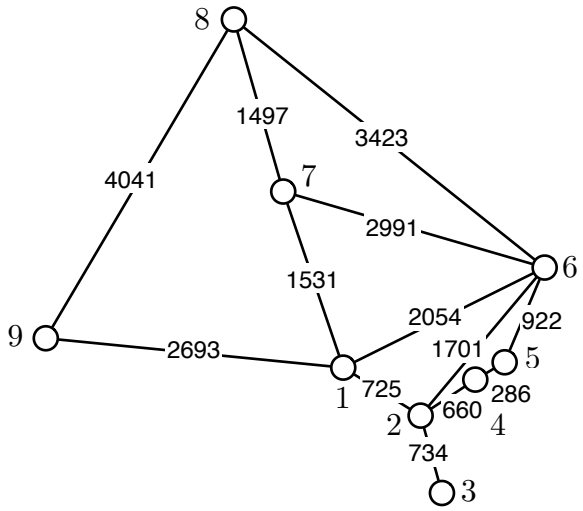
Initialise a subforest $T = (V_T, E_T)$ of G with $V_T = V$ and $E_T = \varnothing$.

In the first step, sort the edges by their weight, so that after relabelling the edges we may assume $\mathrm{w}(e_1) \leq \mathrm{w}(e_2) \leq \ldots \leq \mathrm{w}(e_m)$.

For $i = 1, \ldots, m$, let $e_i = (u_i, v_i)$ and test if the vertices u_i and v_i are in different connected components of T. If yes, replace E_T by $E_T \cup \{e_i\}$ (otherwise ignore e_i and continue with the next edge).

Upon termination, $T = (V, E_T)$ is a minimum spanning tree of G.

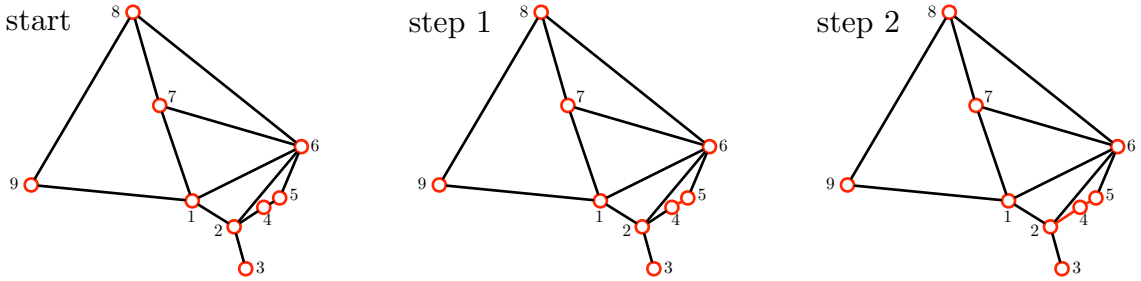
Example 7.24 Consider the following weighted graph $G = (V, E)$ (the numbers on the edges indicate the weights):



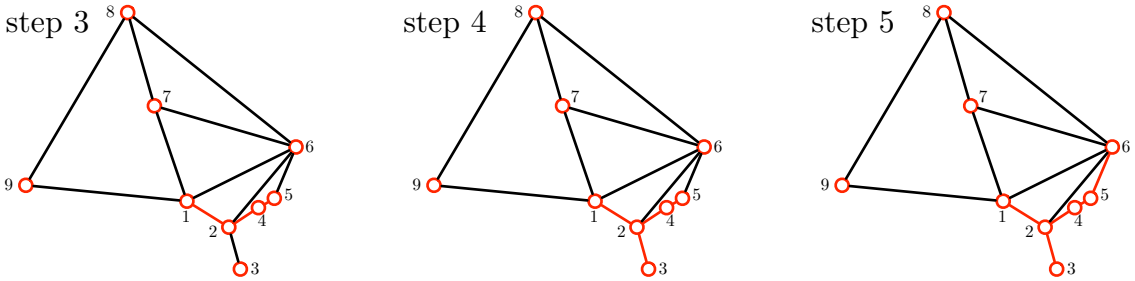
The edges in this graph, ordered by their weight, are

$e_1 = (4, 5), e_2 = (2, 4), e_3 = (1, 2), e_4 = (2, 3), e_5 = (5, 6), e_6 = (7, 8), e_7 = (1, 7),$
$e_8 = (2, 6), e_9 = (1, 6), e_{10} = (1, 9), e_{11} = (6, 7), e_{12} = (6, 8), e_{13} = (8, 9).$

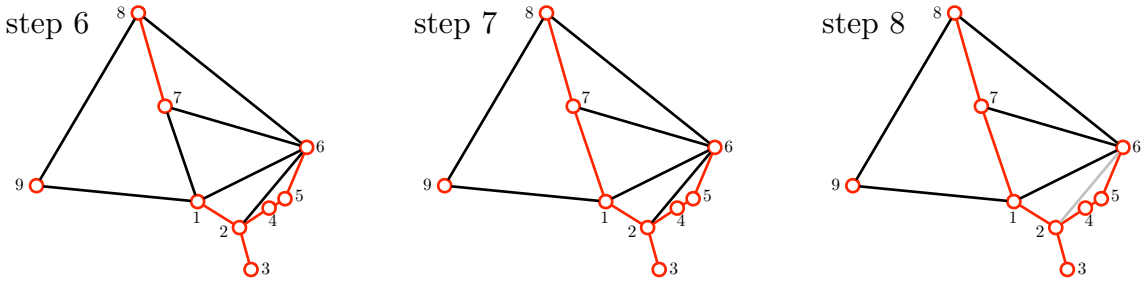
We start with a subgraph T containing all vertices of G but no edges. In the first step, we add the edge $e_1 = (4, 5)$ to the graph. In the second step, we add $e_2 = (2, 4)$.



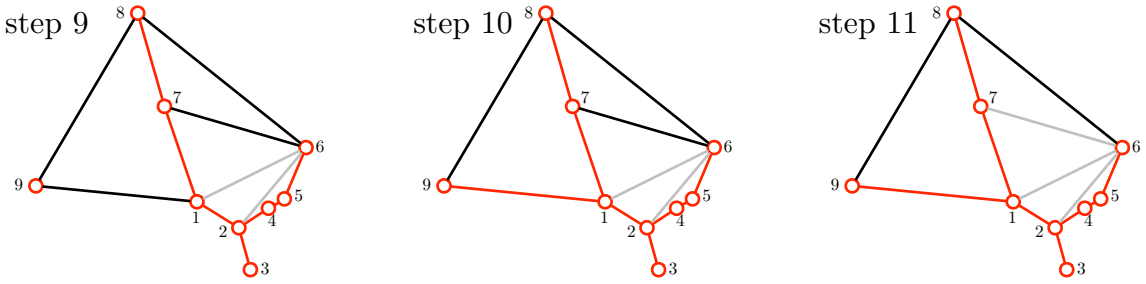
In steps three to five, we add the edges $e_3 = (1, 2)$, $e_4 = (2, 3)$, $e_5 = (5, 6)$.



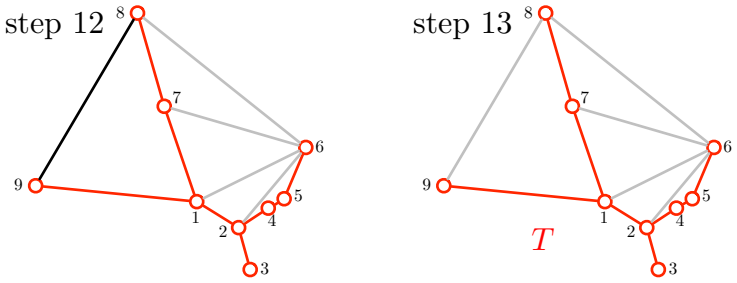
In steps six and seven, we add the edges $e_6 = (7, 8)$ and $e_7 = (1, 7)$. But in step eight, the edge $e_8 = (2, 6)$ is an edge within a connected component of our current T, so we do not add it to T.



In step nine, the edge $e_9 = (1, 6)$ is again within a connected component of T, so we do not add it. In step ten, add the edge $e_{10} = (1, 9)$. In step eleven, the edge $e_{11} = (6, 7)$ is not added.



In steps twelve and 13, neither the edge $e_{12} = (6, 8)$ nor the edge $e_{13} = (8, 9)$ is added.



In the last step we obtain a minimum spanning tree T of G.

Theorem 7.25 *Kruskal's algorithm is correct, that is, it terminates after m steps and returns a minimum spanning tree T of a graph $G = (V, E)$.*

PROOF: Kruskal's algorithm clearly terminates after $m = |E|$ steps. We will prove by induction on m that in each step of the algorithm, $T = (V, E_T)$ is a subforest of a minimum spanning tree.

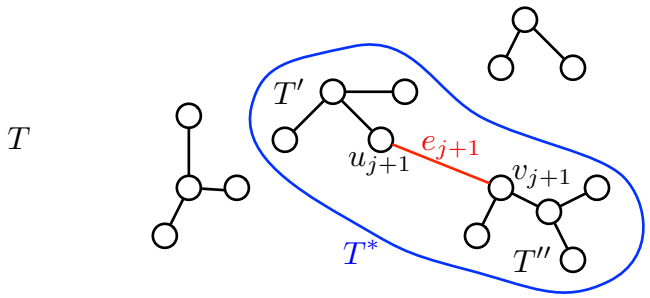
Initial case: $m = 0$. Upon initialisation, T contains all vertices and no edges, $E_T = \varnothing$. Hence T is a subforest of a minimun spanning tree.

Induction hypothesis: In step j, the forest T is contained in a minimum spanning tree $T_{\min}$.

Induction step: Before step $j+1$ is executed, we know that $T = (V, E_T)$ is a subforest of a minimum spanning tree $T_{\min}$ of G by the induction hypothesis. In particular, T is cycle-free. Now consider the edge $e_{j+1} = (u_{j+1}, v_{j+1})$. There are two possibilities:

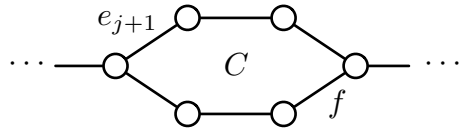
1. If u_{j+1} and v_{j+1} are in the same subtree of T, then we ignore this edge and T is not changed.

2. If u_{j+1} and v_{j+1} are in different connected components T' and T'' of T, then we add the edge e_{j+1}, thus joining these two subtrees of T into a new subgraph T^* of T.



This T^* is connected, since T' and T'' each are connected, and now every vertex in T' can be reached from every vertex in T'' via a path using the edge e_{j+1}, and vice versa. If T^* contains a cycle, then this cycle must contain the edge e_{j+1}, since T' and T'' are trees and thus cycle-free. But this means there exists a second path connecting u_{j+1} to v_{j+1} not using the edge e_{j+1}. This contradicts the assumption that T' and T'' are not connected to each other. So T^* is connected and cycle-free, hence a tree. It follows that after adding the edge e_{j+1} to E_T, T is still cycle-free.

In the first case, T is not changed and therefore still a subforest of $T_{\min}$. We need to show that T is still a subforest of a minimum spanning tree in the second case: Suppose e_{j+1} is an edge in $T_{\min}$. Then T is still a subforest of $T_{\min}$. Otherwise, adding e_{j+1} to $T_{\min}$ creates a cycle C (because $T_{\min}$ is a spanning tree).



Since T is cycle-free, there must be an edge f in C which does not belong to T. If $\mathrm{w}(f) < \mathrm{w}(e_{j+1})$, then Kruskal's algorithm would have considered f before e_{j+1}, and since f was not added to T, it means that the vertices of f are in the same connected component T_f of T. But since f and T_f are contained in $T_{\min}$, this would create a cycle in $T_{\min}$, which contradicts $T_{\min}$ being a tree. Hence

$$\mathrm{w}(e_{j+1}) \leq \mathrm{w}(f).$$

Now obtain $T'_{\min}$ by adding e_{j+1} to $T_{\min}$ and removing f from $T_{\min}$. Then $T'_{\min}$ contains every vertex of G (since $T_{\min}$ does) and has no cycle, hence it is a spanning tree with weight

$$\mathrm{w}(T'_{\min}) = \mathrm{w}(T_{\min}) + \mathrm{w}(e_{j+1}) - \mathrm{w}(f) \leq \mathrm{w}(T_{\min})$$

(in fact, we have $\mathrm{w}(T'_{\min}) = \mathrm{w}(T_{\min})$ because of minimality). So $T'_{\min}$ is a minimum spanning tree containing T after adding the edge e_{j+1}.

This shows that after every step, T is a subforest of a minimum spanning tree. After the last step, T is connected, since G is connected and any edge between connected components of T would have been added by the algorithm. Hence T is a tree containing all vertices of G and therefore must be a minimum spanning tree itself. ∎

Note that Kruskal's algorithm also works for graphs with loops and for multigraphs. Loops are simply removed by the algorithm, and from multiple edges between the same vertices, the one with the lowest weight is chosen.

What remains is to clarify how to test if two vertices u, v are contained in different connected components of T in Kruskal's algorithm. To do this, we need a suitable data structure for the forest T. Recall the connected components $T_1, \ldots, T_k$ form a partition of T. The data structure for this partition must allow to test whether two vertices are in the same subtree and to join two subtrees into one. In the following Section 7.5, we will introduce such data structure. This illustrates how important the construction of good data structures is for the design of algorithms.

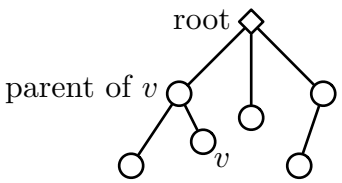
7.5 Addendum: The union-find data structure

In this section we study a data structure that can be used to keep track of the connected components in Kruskal's algorithm. Given a set $V = \{v_1, \ldots, v_n\}$, a **union-find data structure** maintains a partition $V_1 \cup \ldots \cup V_k$ of V (see Section 1.5). We call $V_1, \ldots, V_k$ the **blocks** of the partition. The union-find data structure provides two functions,

- `find`(v) to find a the block V_i of the partition containing v
- and `union`(i, j) to combine V_i and V_j into a new block $V_i \cup V_j$.

Each block is stored as a **rooted tree**, which means the tree has a designated vertex, the **root**, which represents the whole block. The root of each block is then used to represent the block. For every vertex in the rooted tree, the **parent** is the next vertex on a path to the root.

Example 7.26 A rooted tree. The root is indiciated by a diamond.

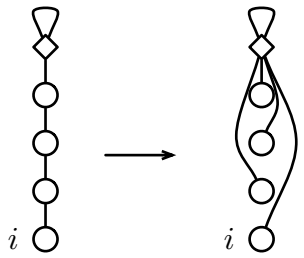


The union-find data structure for a set with n vertices stores an array `parent`, with `parent`$[i]$ being the parent of vertex v_i (we henceforth identify the vertex v_i with its index i). The root vertices of the blocks are their own parents.

The `find` function is implemented as follows:

```
vertex find(i)
    if parent[i] = i then return i
    else
        j := find(parent[i])
        parent[i] := j
        return j
```

The first `if` tests if i itself is a root, in which case i is returned as the representative of its block in the partition. If i is not the root, then `find` is applied recursively to the parent of i until the root is found. While traversing the path from i to its root, every vertex along the path is assigned the root as its parent.

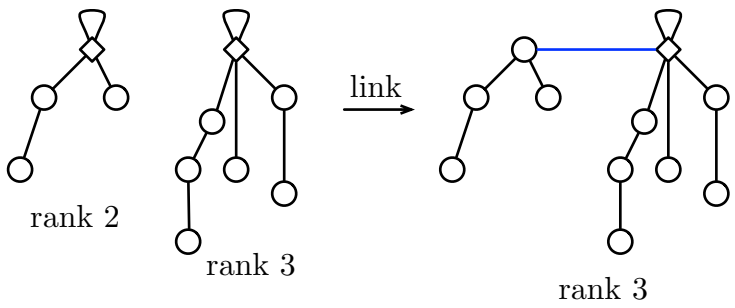


In this way, a chain of parents is never traversed twice. All the vertices visited while executing `find` are redirected to their block's representative, the root.

The union-find data structure also stores an array `rank` of numbers for each root. Whenever two blocks of the partition are joined together, the root of larger rank becomes the root of the combined block. If both have the same rank, the choice is arbitrary, but in this case the rank of the root is increased.

This is implemented in an auxiliary procedure `link`(i, j):

```
void link(i,j)
    if rank[i] < rank[j] then parent[i] := j
    else
        parent[j] := i
        if rank[i] = rank[j] then rank[i] := rank[i]+1
```



Finally, the `union` operation is implemented as follows:

```
void union(i,j)
    if find(i) != find(j) then link(find(i),find(j))
```

So if the `find` function returns different roots for i and j, it means they are in different blocks of the partition. In this case, the `link` procedure is used to join them into a new block.

It can be shown that the implementation of a union-find data structure presented in this section is a very efficient one. For Kruskal's algorithm, initially each vertex is a block of its own in the partition. If we use the union-find data structure to keep track of the subtrees, then the main computational cost factor in Kruskal's algorithm is sorting the edges. This can be done very efficiently, for example by using Algorithm 4.10 (quicksort).

A Matrices and vectors

A $\boldsymbol{n \times k}$**-matrix** is a rectangular array of numbers,

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{pmatrix}$$

with n rows and k columns. The entries a_{ij} of the matrix are all taken from the same set of numbers; usually this will be the real numbers $\mathbb{R}$, but taking entries in other sets such as $\mathbb{Z}_m$ is also possible.

A **vector** is a matrix with one column only,

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}.$$

Vectors are often used as a coordinate representation of points in n-dimensional space.

A.1 Matrix arithmetic

There are three arithmetic operations for matrices: multiplication by a number, addition of two matrices and multiplication of two matrices.

If A is a $n \times k$-matrix whose entries a_{ij} are numbers in a set S, and $c \in S$, then we can multiply A by c,

$$cA = \begin{pmatrix} ca_{11} & \cdots & ca_{1k} \\ \vdots & \ddots & \vdots \\ ca_{n1} & \cdots & ca_{nk} \end{pmatrix}.$$

Example A.1 Let $S = \mathbb{Z}$, $c = 5$ and

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

Then

$$cA = 5\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 5 & 10 & 15 \\ 20 & 25 & 30 \end{pmatrix}.$$

Given two $n \times k$-matrices A and B, we can add them by adding the entries in corresponding lines and columns,

$$A + B = \begin{pmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{pmatrix}.$$

Example A.2 Let

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \quad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}.$$

Both matrices have 2 rows and 3 columns, so we can add them:

$$A + B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 1+7 & 2+8 & 3+9 \\ 4+10 & 5+11 & 6+12 \end{pmatrix} = \begin{pmatrix} 8 & 10 & 12 \\ 14 & 16 & 18 \end{pmatrix}.$$

If A is a $n \times k$-matrix and B is a $p \times q$-matrix, then we can form a matrix product $C = AB$ if $k = p$. Then C is an $n \times q$-matrix with entries

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{ik}b_{kj}.$$

Example A.3 Let

$$A = \begin{pmatrix} 1 & 2 & 0 & -1 \\ 4 & 0 & 2 & 2 \\ -1 & 9 & 2 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 2 \\ 3 & 3 \\ -1 & 5 \\ -2 & 7 \end{pmatrix}.$$

A is a 3×4-matrix and B is a 4×2-matrix. So they can be multiplied. The product must be a 3×2-matrix.

$$C = AB = \begin{pmatrix} 1 & 2 & 0 & -1 \\ 4 & 0 & 2 & 2 \\ -1 & 9 & 2 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 \\ 3 & 3 \\ -1 & 5 \\ -2 & 7 \end{pmatrix} = \begin{pmatrix} 8 & 1 \\ -6 & 32 \\ 23 & 42 \end{pmatrix}.$$

An $n \times n$-matrix A is called a **quadratic matrix**. If A and B are both $n \times n$-matrices, we can form both products AB and BA. In general, matrix multiplication is not commutative:

$$AB \neq BA.$$

Example A.4

$$\begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 0 & -1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 1 & 2 \end{pmatrix}.$$

A.2 Inverses

The **identity matrix** I_n is an $n \times n$-matrix with entries 1 on the diagonal and 0 else:

$$\mathrm{I}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathrm{I}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathrm{I}_n = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & & & 0 \\ \vdots & & \ddots & & \vdots \\ \vdots & & & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

Multiply any $n \times k$-matrix A or $k \times n$-matrix B with I_n:

$$\mathrm{I}_n A = A, \quad B\mathrm{I}_n = B.$$

A quadratic matrix A is called **invertible**, if there exists a matrix A^{-1} such that

$$AA^{-1} = \mathrm{I}_n = A^{-1}A.$$

A^{-1} is called the **inverse** of A.

Example A.5 The matrix

$$A = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix}$$

is invertible. Its inverse is

$$A^{-1} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}.$$

Proof:

$$AA^{-1} = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + (-2) \cdot 0 & 2 \cdot 1 - 2 \cdot 1 \\ 0 \cdot 1 + 1 \cdot 0 & 0 \cdot 2 + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

A.3 Determinants

The **determinant** of a 2×2-matrix is defined as

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{12}a_{21}.$$

Theorem A.6 *A is invertible if and only if* $\det(A) \neq 0$.

Example A.7 We want to know if

$$\begin{pmatrix} 2 & -3 \\ -4 & 6 \end{pmatrix}$$

is invertible. Compute the determinant:

$$\det \begin{pmatrix} 2 & -3 \\ -4 & 6 \end{pmatrix} = 0,$$

so this matrix is not invertible.

A.4 Matrices acting on vectors

By the rule of matrix multiplication, we can multiply a vector $\boldsymbol{x} \in \mathbb{R}^n$ with a quadratic matrix A and obtain a new vector in $\mathbb{R}^n$,

$$A\boldsymbol{x} = \boldsymbol{y}.$$

Multiplication of a vector with a matrix is **linear**: If $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ and $c \in \mathbb{R}$, then

$$A(\boldsymbol{x} + \boldsymbol{y}) = A\boldsymbol{x} + A\boldsymbol{y},$$
$$A(c\boldsymbol{x}) = cA\boldsymbol{x}.$$

The unit vectors are

$$\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Every vector $\boldsymbol{x}$ is of the form

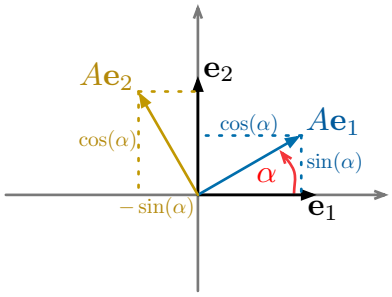
$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2.$$

By linearity, matrix-vector multiplication is determined by the products $A\mathbf{e}_1$ and $A\mathbf{e}_2$:

$$\begin{aligned} A\boldsymbol{x} &= x_1 A\mathbf{e}_1 + x_2 A\mathbf{e}_2 \\ &= x_1 \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \mathbf{e}_1 + x_2 \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \mathbf{e}_2 \\ &= x_1 \begin{pmatrix} a_{11} \\ a_{21} \end{pmatrix} + x_2 \begin{pmatrix} a_{12} \\ a_{22} \end{pmatrix} = \begin{pmatrix} x_1 a_{11} + x_2 a_{12} \\ x_1 a_{21} + x_2 a_{22} \end{pmatrix} \end{aligned}$$

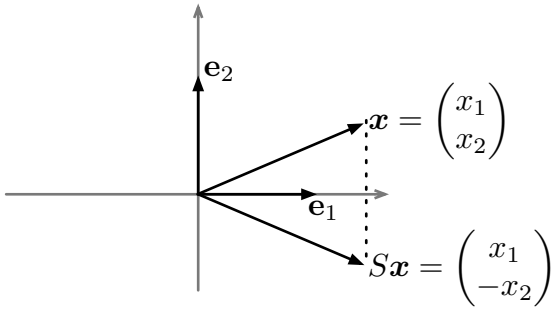
Example A.8 A rotation about the angle α realised by the matrix

$$A = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}.$$



Example A.9 Reflection on the $\mathbf{e}_1$-axis realised by the matrix

$$S = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

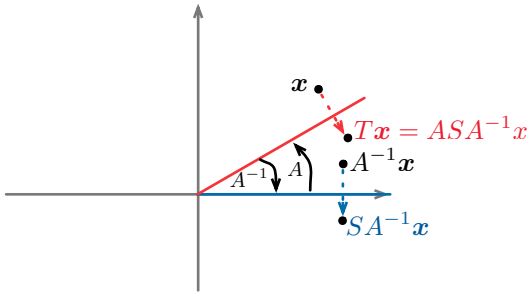


Example A.10 How can we realise a reflection T about an axis tilted by an angle α? Combine rotation and reflection

(i) Tilt axis back to $\mathbf{e}_1$-axis using rotation A^{-1}.

(ii) Use reflection S on $\mathbf{e}_1$-axis.

(iii) Tilt axis back to original position using A.


$\boldsymbol{x}$
$T\boldsymbol{x} = ASA^{-1}x$
$A^{-1}\boldsymbol{x}$
A^{-1}
A
$SA^{-1}\boldsymbol{x}$

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein
Introduction to Algorithms, third edition
MIT Press, 2009

[2] M. Cozzens, S.J. Miller
The Mathematics of Encryption: An Elementary Introduction
American Mathematical Society, 2013

[3] R.L. Graham, D.E. Knuth, O. Patashnik
Concrete Mathematics
Addison-Wesley, 1994

[4] J. Katz, Y. Lindell
Introduction to Modern Cryptography, second edition
Chapman & Hall/CRC, 2014

[5] K.A. Ross, C.R. Wright
Discrete Mathematics, fifth edition
Pearson, 2003

[6] N.J.J. Smith
Logic - The Laws of Truth
Princeton University Press, 2012

[7] W.D. Wallis
A Beginner's Guide to Discrete Mathematics, second edition
Birkhäuser, 2012

Index