Discrete Mathematics

Iain Phillips

Introduction

This short work introduces the basics of discrete mathematics—in particular, sets, functions and relations. It is intended for beginning students of computer science. They will find that a knowledge of the concepts covered here will aid them in understanding many areas of computer science, for instance, data types, databases, specification, functional and logic programming.

This material was written as lecture notes to accompany a short course on Discrete Mathematics given to first-year students in the Department of Computing at Imperial College. The work is self-contained, though rather concise. I shall be grateful if any inaccuracies are brought to my notice.

These notes are pitched at a fairly abstract level. Abstraction is a powerful tool for the computer scientist, as it allows us to focus on what is important by removing the irrelevant, and thereby make sense of the mass of information associated with a given problem. It allows us to solve a class of problems once for all, instead of having to perform similar reasoning on each of the individual problems. However it can be intimidating to deal with abstract objects until they become more familiar. If at any point things seem to be getting “too abstract” (too many x's and y's), the reader is recommended to make them more concrete by thinking of a particular example, or series of examples.

Further Reading

J.L. Gersting, *Mathematical Structures for Computer Science*, 3rd ed., Freeman 1993.
B.T. Denvir, *Introduction to Discrete Mathematics for Software Engineering*, Macmillan 1986.
J.K. Truss, *Discrete Mathematics for Computer Scientists*, Addison-Wesley 1991.

Many other books cover the material and are suitable.

Revised October 1994

Contents

1. Sets

For the moment assume that sets are formed from a pool of individuals, which are thought of as single and indivisible. A *set* is a collection of such individuals. We write x A to mean that individual x is a *member* of the set A. x A means that x is not a member of A, in other words ¬(x A).

There is a clear correspondence between sets and types in a language such as Miranda. For instance in Miranda there is a predefined type bool (the Booleans, or truth values), and there are two values of type bool, called True and False. We can regard bool as being the set with members True, False, which we write {True, False}. Where in Miranda we might write True::bool, in set notation we write True bool.

1.1 Equality.

Whenever we introduce a structure (in this case sets), an important part of knowing what we mean is to know when two objects of the structure are equal, in other words denote the same thing. Assume that we know when two individuals are equal. When are two sets equal?

Extensionality. Sets are quite transparent; they are nothing more or less than what they contain.

The Principle of Extensionality: A set is determined by its *extension*, i.e. its members.

So two sets are equal if they contain the same objects:

For any sets A,B, A=B if and only if

for every individual x (x A ⇔ x B)

1.2 Subsets.

DEFINITION. Let A, B be any two sets. Then A is a subset of B iff for every x (x A x B)

Notation: A B.

REMARKS (1) We can use a quite general argument: Assume we know A B and a A. Then a B.
For instance, if we know rodents mammals and squirrel rodents, then we can deduce squirrel mammals.

(2) The principle of extensionality tells us that

A=B iff A B and B A

PROPOSITION. Let A,B,C be sets. If A B and B C then A C.

Proof Assume A B and B C. Take any a A. We must show a C. But A B. Hence a B. Also B C. Hence a C as required.

1.3 Defining sets.

There are two basic ways:

(1) *Enumeration.* Just list the elements within braces {}.

For instance {cat,dog}
points = {N,S,E,W}
N = {0,1,2,...}

N is of course an infinite set (the "natural numbers"). The "..." indicates that the remaining elements are enumerated by some rule (in this case "add one to the previous number"). Notice that {dog, cat} and {cat, dog, cat} are the same sets as {cat, dog}, by extensionality. It does not matter if we change the order or repeat elements.

(2) *Comprehension.* Define a set by the property its members possess. Let P(x) be some property, which we here think of as a predicate which may or may not hold for various values of x. Then the corresponding set is all those individuals x for which P(x) holds.

Notation {x | P(x)} or {x: P(x)}

EXAMPLE {x | x is a prime number}

We can also define sets as subsets of sets we already know about: {x A | P(x)} means the set of all those members x of A for which P(x). It is equivalent to {x | x A & P(x)}.

EXAMPLE Primes = {x **N** | x is prime}

Notice that the *empty* (or null) set can be defined by comprehension:

= {x | false}

We shall also use the notation {}.

REMARK The use of completely unrestricted comprehension has the problem that contradictory or paradoxical sets can be defined (such as the Russell set of all those sets which are not members of themselves). There are ways round this, and it need not worry us here.

1.4 Operators on sets.

As well as defining sets directly we can also build them up from sets already defined.

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

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

Difference $A-B = \{x \mid x \in A \text{ and } x \notin B\}$

Symmetric difference $A \Delta B = (A-B) \cup (B-A)$

Clearly $A \cap B \subseteq A \subseteq A \cup B$.

DEFINITION. Two sets A,B are *disjoint* iff $A \cap B = \emptyset$.

Notice that $A \cup B$ is the union of three disjoint sets: A-B, B-A, $A \cap B$.

DEFINITION. For any finite set A, the *cardinality* of A is the number of elements contained in A.

Notation: $|A|$

PROPOSITION. Let A,B be any finite sets. Then $|A \cup B| = |A| + |B| - |A \cap B|$

So if A,B are disjoint then $|A \cup B| = |A| + |B|$.

Notice that it does not make sense to write e.g. $|A| \cup |B|$, since $|A|$ and $|B|$ are numbers, not sets.

1.5 Laws on operators and counterexamples.

$\cup, \cap, -, \Delta$ are all binary operators (they take two arguments). They remind us of the arithmetical operators $+, \times, -, \div$.

There are two very convenient properties which a binary operator * can have:

DEFINITION. A binary operator * is

commutative iff for all x,y, $x*y = y*x$
associative iff for all x,y,z $(x*y)*z = x*(y*z)$

If * is associative we can omit the brackets and write x*y*z.

Which of these laws hold for the arithmetical operators? It is well-known that + and × are commutative and associative. However - and ÷ are neither commutative nor associative. To see that - is not commutative it is enough to observe that $3-4 \neq 4-3$. This is because since the

commutativity property asserts that something is true *for all* numbers x,y, to disprove it it is sufficient to produce *one* example where this fails. This is known as a *counterexample*. We can produce a counterexample to show that - is not associative:

$(3-4)-5 = -6$
$3-(4-5) = 4$

EXERCISE. Show that ÷ is neither commutative nor associative by providing counterexamples.

Other *universal* statements (those beginning "for all ...") can be disproved by counterexample. For instance to show that A B fails (for some particular A,B), exhibit an individual a such that a A but not a B.

DEFINITION. A *universal* statement is a statement of the form "for all x, P(x)", where P(x) is some statement. A *counterexample* is an individual a such that P(a) is false (in other words not P(a)).

Turning to the set operators, one can show that and ∩ are both commutative and associative. Let us check that is associative: Take any sets X,Y,Z. We wish to show that (X Y) Z = X (Y Z). But

(X Y) Z = {x | x X Y or x Z} definition of
= {x | (x X or x Y) or x Z} definition of
X (Y Z) = {x | x X or (x Y or x Z)}

But we know that "or" is associative – it can be checked by truth tables that (A B) C ↔ A (B C). Hence (X Y) Z = X (Y Z). This could also have been shown by Venn diagrams.

It is clear that and ∩ enjoy the same properties on sets that and have on propositions.

Two further laws on operators are of interest:

Distributivity. In arithmetic we know

$x \times (y+z) = x \times y + x \times z$

but in general

$x + (y \times z) \neq (x+y) \times (x+z)$

We say that × *distributes* over + (from the left, though this is immaterial as here × is commutative) but + does not distribute over × .

In the case of sets, for any A,B,C,

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

This may be proved via translation into logic or by Venn diagrams.

Idempotence. For any set A, $A \quad A = A$ and $A \cap A = A$. We say that $\quad$ and $\cap$ are *idempotent*.

1.6 Other set forming operations.

Powerset. Given any set A we can form a new set $\mathbf{P}(A)$ consisting of all the subsets of A:

$$\mathbf{P}(A) = \{X \mid X \quad A\}$$

Notice that the elements of $\mathbf{P}(A)$ are sets.

EXAMPLE. $\mathbf{P}(\{1,2\}) = \{ \quad ,\{1\},\{2\},\{1,2\}\}$

PROPOSITION. If A is a finite set and $|A| = n$ then $|\mathbf{P}(A)| = 2^n$.

To see this let $A = \{a_1,\ldots,a_n\}$. We form a subset X of A by taking each element a_i in turn and deciding whether or not to include it in X. This gives us n independent choices between two possibilities (in or out). The number of different subsets we can form is therefore 2^n (in other words $2\times \ldots \times 2$ [n times]).

In general there are m^n ways of making n independent choices between m options.

Cartesian product.. An ordered pair (a,b) is a pair of objects a and b where the order matters. Thus $(a,b)\neq(b,a)$ unless a=b. Moreover (a,b) is different from {a,b}, since a set is unordered. For any a,b,c,d,

(a,b)= (c,d) iff a=c and b=d

DEFINITION. For sets A,B the *Cartesian product* $A\times B$ is $\{(a,b) \mid a \quad A \text{ and } b \quad B\}$

Notation: We can write A^2 instead of $A\times A$.

EXAMPLES. (1) The real plane $\mathbb{R}^2$. Points are described by their coordinates (x,y)

(2) Computer marriage bureau. Let M be the set of all men registered and W the set of all women registered. Then the set of all possible matches is $M\times W$.

PROPOSITION. If A,B are finite sets then $|A\times B| = |A|.|B|$.

To see this, suppose $A=\{a_1,\ldots,a_m\}$, $B=\{b_1,\ldots,b_n\}$. Draw a table with m rows and n columns of the members of $A\times B$

(a_1,b_1) (a_1,b_2) …
(a_2,b_1) (a_2,b_2) …
…

Such a table clearly has m.n entries.

We also want products of more than two sets. For any n an n-tuple is a sequence $(a_1,\ldots,a_n)$ of n objects where the order (and any repeats) matter. Thus

$(a_1,\ldots,a_n) = (b_1,\ldots,b_n)$ iff for i=1 to n, $a_i=b_i$

Given sets $A_1,\ldots,A_n$ we can form the *n-ary product* $A_1\times\ldots\times A_n$, which is defined to be the set of all n-tuples $(a_1,\ldots,a_n)$ such that $a_i \in A_i$ (i=1,...,n). The product of A with itself n times is written A^n.

EXAMPLES. (1) 3-D space $\mathbb{R}^3$

(2) timetable = day × time × room × courseNo
A typical element: (Monday,15.30,311,140)

PROPOSITION. If $A_1,\ldots,A_n$ are finite sets then $|A_1\times\ldots\times A_n| = |A_1|.|A_2|\ldots .|A_n|$.

Products correspond exactly to the tuple types of Miranda. For instance (num,char) is Miranda's way of forming the product num× char. Our timetable example might be rendered:

```
timetable ::= (day, time, room, courseNo)
(Monday,15.30,311,140) :: timetable
```

There is also a connection with the *record types* of Modula. Suppose we wish to have a database which stores information about people. An array will be unsuitable, since the information, such as height, age, colour of eyes, date of birth, will be of different types. Instead in Modula we can define

```
Person = RECORD
            who: Name;
            height: REAL;
            age: [0..120];
            eyeColour: Colour;
            dateOfBirth: Date
         END
```

Such records are products:

Person = Name × REAL × [0..120] × Colour × Date

Records can be nested, so that in the above example, we might have

```
Date = RECORD
            day: [1..31];
            month: [1..12];
            year: [1900..1990]
       END
```

REMARK. We can now form the product of three sets in three different ways:

$A \times B \times C$ $(A \times B) \times C$ $A \times (B \times C)$

These are all different strictly speaking, so that × as a binary operator is not associative. However there is a clear correspondence between the elements

(a,b,c) ((a,b),c) (a,(b,c))

so that the sets are in a sense equivalent. Nevertheless they are structured differently.

EXAMPLE (Denvir). Each person has three names, drawn from a set N. We can represent the full name either as

Name1 ::= (FirstName, SecondName, Surname)

i.e. N×N×N, or as

Name2 ::= (Forenames, Surname)

where

Forenames ::= (FirstName, SecondName)

i.e. (N×N)×N. The information stored is clearly the same, but the structure is different.

1.7 Partitions

If we divide a set into non-overlapping chunks we get a partition of the set.

DEFINITION. Let S be a set. A *partition* of S is a family $A_1,\ldots,A_n$ of subsets of S such that

- each A_i is non-empty
- the A_i *cover* S, that is, $S = A_1 \ \ldots \ A_n$
- the A_i are *pairwise disjoint*, that is, every pair of sets selected from them is disjoint:
  if $i \neq j$ then $A_i \cap A_j =$

Notation: It is sometimes convenient to write $A_1 \ \ldots \ A_n$ as $\cup_{i<n} A_i$.

EXAMPLES. (1) Humans are partitioned into male and female

(2) Integers are partitioned into even and odd

We shall return to partitions when discussing equivalence relations.

2. Relations

We wish to capture the concept of objects being *related,* e.g.

John **loves** Mary
St Albans **is north of** London
South Kensington **is between** Gloucester Road **and** Sloane Square
$2 < 15$
P **implements** Q

In predicate logic, these are *predicates* or *relations* over individuals.

For our *formal* definition of relations we view a relation as a set. For instance, suppose the set of all people is People. We form a set "loves" consisting of all ordered pairs of people where the first loves the second:

(x,y) loves iff x **loves** y

Thus loves $\subseteq$ People × People.

Informally we shall tend to use the logical notation, writing loves(x,y) rather than (x,y) $\in$ loves.

DEFINITION ($n \geq 1$). An *n-ary relation* is a subset of a Cartesian product $A_1 \times \ldots \times A_n$ of n sets.

We use R,S,... to range over relations. If $R \subseteq A_1 \times \ldots \times A_n$ we say that R is a relation on $A_1 \times \ldots \times A_n$. The *type* of R is $A_1 \times \ldots \times A_n$. Instead of $(a_1,\ldots,a_n) \in R$ we usually write $R(a_1,\ldots,a_n)$. We say that "$a_1,\ldots,a_n$ are related by R".

A case of particular importance is where the relation R is *binary*, that is $R \subseteq A \times B$, some A,B. We say that R is a relation from A to B. If $R \subseteq A \times A$ we say that R is a binary relation on A. Similarly if $R \subseteq A^n$ then R is an n-ary relation on A. Often it will be convenient to use infix notation for binary relations and write aRb instead of R(a,b). For instance, "<" is a binary relation on $\mathbb{R}$ and we usually write x<y rather than <(x,y).

REMARKS. (1) A relation does not have to be "meaningful"; *any* subset of a Cartesian product is a relation. For instance "<" is a binary relation on **N**, and is the set

$< = \{(0,1), (0,2), (1,2), ..., (47,108), ...\}$

But any other set of ordered pairs of natural numbers such as

$R = \{(2,7), (101,12), (13,0)\}$

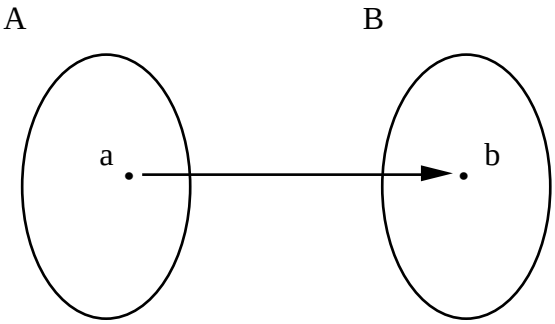
is also a binary relation on **N**.

(2) We have formally defined "relation" in terms of "set". However the idea of a relation is just as fundamental as that of a set, since in order to understand sets we have to have an informal grasp of the membership relation which relates objects to sets.

2.1 Representing relations.

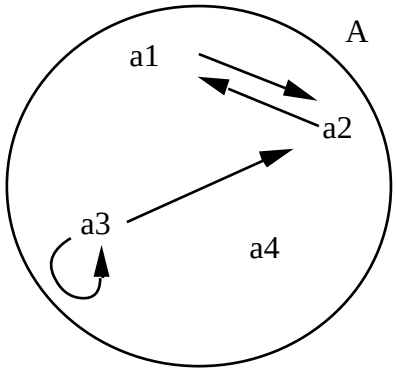
Listing ordered pairs can get tedious and hard to follow. For binary relations R A×B we have other representations.

(1) *Diagram*
(a) We can represent aRb by drawing a line from a to b, as in the diagram to the right.



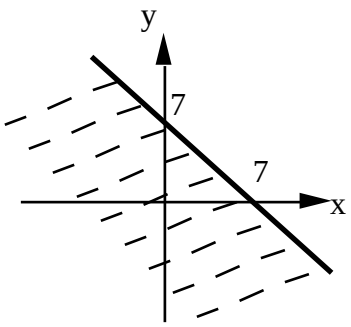
(b) In the case where R is a binary relation on A we can also use a *directed graph*, which consists of a set of *nodes* joined by arrowed lines indicating a relationship between the nodes.

We illustrate by means of an example. Let
$A = \{a_1,\ldots,a_4\}$ and let
$R = \{(a_1,a_2),(a_2,a_1),(a_3,a_2),(a_3,a_3)\}$.



Notice that the direction of the arrows matters. It is of course possible to use a diagram where the source and target sets are drawn separately as in (a).

(c) We can represent a relation on $\mathbb{R}$ as an area in the plane. The accompanying diagram represents R, where xRy iff $x+y \leq 7$.



(2) *Matrix (array).* Suppose that |A|=m, |B|=n. We can represent R by an m×n matrix M of booleans (T,F). Let $A = \{a_1,\ldots a_m\}$, $B = \{b_1,\ldots b_n\}$. Then for i = 1,...,m and j = 1,...,n

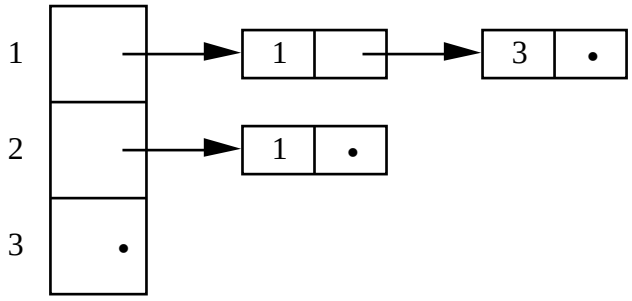
$$M(i,j) = \begin{cases} T & \text{if } a_i R b_j \\ F & \text{otherwise} \end{cases}$$

It is common to use 1,0 instead of T,F. M(i,j) is the entry for the ith row and jth column.

On a computer, storing a relation as an array allows random access and easy manipulation, but can be expensive in space if the relation is much smaller than A × B. With a *sparse* relation, where there are not many ordered pairs, an alternative is an array of linked lists, called an *adjacency list*. As an example, suppose that R is the following binary relation on {1,2,3}:

{(1,1), (1,3), (2,1)}

R has only 3 out of the 9 possible ordered pairs. We create an array of three pointers, one for each element of {1,2,3}, and list for each element which other elements it is related to (adjacent to in the directed graph), as in the following diagram.



2.2 New relations from old

Since relations are sets they inherit certain operators, in particular *union*, *intersection* and *complement*, defined as follows:

DEFINITION. Let R,S $A_1 \times \ldots \times A_n$. Define R S, $R \cap S$, $\overline{R}$ $A_1 \times \ldots \times A_n$ by

$(a_1,\ldots a_n)$ R S iff $(a_1,\ldots a_n)$ R or $(a_1,\ldots a_n)$ S
$(a_1,\ldots a_n)$ $R \cap S$ iff $(a_1,\ldots a_n)$ R and $(a_1,\ldots a_n)$ S
$(a_1,\ldots a_n)$ $\overline{R}$ iff $(a_1,\ldots a_n)$ R

NOTES. (1) To form a union or intersection the relations must be of the same type. The type of the complement $\overline{R}$ is the same as that of R.

(2) The notations R+S, R.S are sometimes used instead of R S, $R \cap S$ respectively.

(3) Union, intersection and complement on relations are the counterparts of the Boolean operations of disjunction, conjunction and negation in logic.

(4) We can also define $\overline{R}$ using set difference: $\overline{R} = A_1 \times \ldots \times A_n - R$

(5) Frequently complement is indicated by crossing out an infix relation: $\neq$, etc.

As well as inheriting structure from sets, relations have operations of their own which do not apply to sets in general. We now define identity, inverse and composition.

DEFINITION. Given any set A, the *identity relation* on A is defined by

$id_A = \{(a,a) \mid a \quad A\}$

Clearly id_A is just "=" on A.

DEFINITION. Given R A×B define the *inverse* of R by

$bR^{-1}a$ iff aRb

Plainly R^{-1} B×A.

NOTES. (1) Inverse is only defined for binary relations. Any binary relation has an inverse.

(2) Inverse should not be confused with complement.

(3) Sometimes inverse is indicated by reversing an infix relation: > is the inverse of <, etc.

(4) In matrix terms inverse is *transpose* (not matrix inverse!)

(5) If we invert a relation twice we recover the original relation: $(R^{-1})^{-1} = R$.

EXAMPLE. The inverse of "is the parent of" is "is the child of":

x is the parent of y iff y is the child of x

DEFINITION (Composition). Given R A×B, S B×C define R∘S A×C by

a R∘S c iff there exists b B. aRb and bSc (any a A,c C)

NOTES. (1) "R∘S" may be read as "R compose S" or "R circle S"

(2) The types of R and S must be compatible or else R∘S is not defined.

(3) One may think of b as being an intermediate point on a path from a to c. Thus a is related to c by R∘S if there is a path from a to c via some member b of B, using R and then S.

(4) In matrix terms ∘ is multiplication where we use instead of × and instead of +. For instance

$$\begin{pmatrix} T & T \\ F & F \end{pmatrix} \begin{pmatrix} T & F \\ T & T \end{pmatrix} = \begin{pmatrix} T & T \\ F & F \end{pmatrix}$$

If we used 0,1 then × would be as usual and + would obey 1+1=1. The same example becomes

$$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

(5) If R is a binary relation on A it makes sense to form $R \circ R^{-1}$. In general this is not the same as id_A.

(6) If R A×B then $id_A \circ R = R \circ id_B = R$

EXAMPLE. grandparent = parent ∘ parent

x is grandparent of y iff z (x is parent of z & z is parent of y)

PROPOSITION. ∘ is associative

Proof We must show (R∘S)∘T = R∘(S∘T) where R A×B, S B×C, T C×D some A,B,C,D.

But x (R∘S)∘T w iff z. x R∘S z T w
iff z y. x R y S z T w
iff y z. x R y S z T w
iff y. x R y S∘T w
iff x R∘(S∘T) w

Notice the advantage of using infix notation in this proof. □

REMARK. We don't really "need" operators like ∘, since we can always replace them by their definitions and work directly (using predicate logic, etc.). But the operators are a convenient shorthand and laws such as associativity give us something of the ease and calculating power of ordinary arithmetic.

We show that two laws do not hold by exhibiting counterexamples:

1. In general $R \neq R^{-1}$. The simplest possible example: Take A = {a,b} (where a≠b) and define R A^2 to be {(a,b)}. Then R^{-1}={(b,a)} which is plainly different from R.

2. Composition is not commutative. We must find R,S such that R∘S≠ S∘R. For both R∘S and S∘R to be defined we must have R A×B, S B×A. Let A=B={a,b}. Let R = {(a,a)}, S = {(a,b)}. Then R∘S= {(a,b)} but S∘R = (a diagram makes this clear). We have the desired counterexample.

EXAMPLE (to illustrate composition of relations). Members of staff occupy various rooms (possibly more than one). They have various keys, and these keys open various rooms. We model this by defining three sets – staff, key, room – and three binary relations

occupies staff × room
hasKey staff × key
opens key × room

We would like to know which staff members can open which rooms. This will be a relation

canOpen staff × room

Plainly s canOpen r iff s hasKey k & k opens r, for some key k. But this means that

canOpen = hasKey ∘ opens

Notice that a member of staff may be able to open a room using more than one key.

A question of interest is whether staff members can open the rooms they occupy. In other words, is it true that

s,r. s occupies r s canOpen r

This is equivalent to asking whether occupies canOpen.

2.3 Applications to relational databases

In this section we describe some simple connections between our work on relations and databases. We define the database operations of join, projection and selection. We only deal

with the static aspects of databases, not concerning ourselves with updating and maintaining integrity.

A relational database is a collection of relations. The records in the database are the *tuples* belonging to the various relations. Each relation has various *attributes*. As an example, a university registry database may have a relation we shall call Student, which stores students' names and addresses, and their examination number, used for purposes of anonymity. The attributes of Student are name, address and number. It is usual to present the tuples of a relation in tables. For instance a table for Student might look like this:

name	*address*	*number*
…	…	…
Brown, B.	5 Lawn Rd.	105
Jackson, B.	1 Oak Dr.	167
Smith, J.	9 Elm St.	156
Walker, S.	4 Ash Gr.	189
…	…	…

Each tuple of the relation corresponds to a row in the table. Each attribute corresponds to a column. Associated with each attribute is a set (or *domain*) from which it takes its values. It is clear that these database relations are just the same as the n-ary relations we have been studying. In our example we may write

Student name_set × address_set × number_set

using an obvious notation for the sets associated with each attribute.

Suppose that the registry database has another relation, called Exam, which records the results for students taking the compilers exam. It has attributes number and grade. A table for Exam might look like this:

number	*grade*
…	…
105	A
156	A
189	C
…	…

Notice that Student and Exam share an attribute, namely number. We can combine the the two realtions using an operation called *join* to get a new relation called Student-Exam which “matches” the two relations on their common attribute:

name	*address*	*number*	*grade*
…	…	…	…
Brown, B.	5 Lawn Rd.	105	A
Smith, J.	9 Elm St.	156	A
Walker, S.	4 Ash Gr.	189	C
…	…	…	…

Notice that candidate 167 did not sit the exam, and therefore does not appear in the join.

In the language of database theory, the new relation might be written

join Student **and** Exam **over** number

We could define the join quite easily using our logical notation:

Student-Exam(n, a, no, g) iff Student(n, a, no) & Exam(no, g)

We might wish to modify Student-Exam by hiding the examination number information to get a new relation Results. This may be done by the database operation of *projection*. The table for Results would look like:

name	*address*	*grade*
…	…	…
Brown, B.	5 Lawn Rd.	A
Smith, J.	9 Elm St.	A
Walker, S.	4 Ash Gr.	C
…	…	…

In database notation Results may be written

project Student-Exam **over** (name, address, grade)

In our logical notation we may write:

Results(n, a, g) iff no. Student-Name(n, a, no, g)

Notice that Results is got from Student and Name by a sort of generalized relational composition. In fact compositions of binary relations can be constructed by a join followed by a projection. Going back to the keys example of the previous section, it can be seen that the relation “canOpen” may be got from “hasKey” and “opens” using database operations as

project (**join** hasKey **and** opens **over** key) **over** (staff, room)

Another thing we might wish to do is to *select* a part of a relation table which is of interest. Suppose in our registry example we wish to have the names of those who should be

considered for the prize, and so we select those candidates who got A in the exam. Starting from the table Results, we get a table which we might call A-Results:

name	*address*	*grade*
…	…	…
Brown, B.	5 Lawn Rd.	A
Smith, J.	9 Elm St.	A
…	…	…

In database notation we have

select Results **where** grade = 'A'

In logical notation we could write

A-Results(n, a, g) iff Results(n, a, g) & g = A

The relation A-Results gives us the names we want, but we could reduce clutter by applying a further projection to get the relation PrizeCands with a single attribute:

name
…
Brown, B.
Smith, J.
…

This is of course

project A-results **over** names

We have now introduced three database operations (join, projection, selection) and seen how each operation has a counterpart in logic.

2.4 Properties of Relations

We now define three standard properties which relations may possess. They will be of use in our later study of relations which express equivalence between objects or ordering between objects.

DEFINITION. Let R be a binary relation on A.

(i) R is *reflexive* iff $\forall x \in A.\ Rxx$
(ii) R is *symmetric* iff $\forall x,y \in A.\ Rxy \Leftrightarrow Ryx$
(iii) R is *transitive* iff $\forall x,y,z \in A.\ Rxy \ \&\ Ryz \Rightarrow Rxz$

NOTES. (1) These are all universal properties

(2) We can define them equivalently in terms of the operations on relations introduced in the previous section:

R is reflexive iff $id_A \subseteq R$
R is symmetric iff $R = R^{-1}$
R is transitive iff $R \circ R \subseteq R$

EXAMPLES. (1) = is reflexive, symmetric and transitive.

(2) ≤ on numbers is reflexive and transitive but not symmetric. The same for $\subseteq$ on sets.

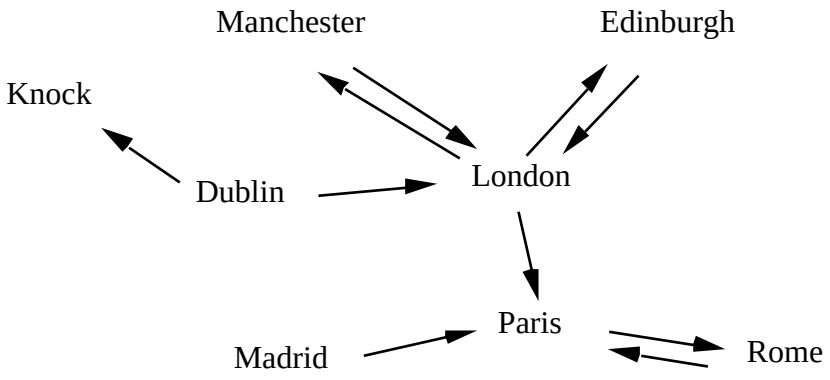
(3) < on numbers is transitive but not reflexive or symmetric.

2.5 Transitive Closure

Consider the following situation. There are various direct flights between various cities. We wish to know for any two cities whether there is a possible trip from one to the other allowing changes of plane. We can model this by defining a set city of cities and a binary relation F on city, such that

aFb iff there is a direct flight from a to b

This relation may be represented as a directed graph with the cities as nodes, as in the following example:



Let F^+ be the relation

a F^+b iff there is a trip from a to b allowing changes

Then clearly a F^+b iff there is some path from a to b in the directed graph. For instance there is a path from Manchester to Rome, but no path from Rome to Manchester. We would like to calculate F^+ from F.

We can express the relation F^+ in terms of F using relational composition. First note that

a F^+b iff there is a path of length n from a to b, some n≥1

Now

aFb iff there is a path of length 1 from a to b
a F∘F b iff there is a path of length 2 from a to b (via some intermediate city)

Define $F^n = F \circ F \circ \ldots \circ F$ (n Fs). Then for any n≥1

a F^nb iff there is a path of length n from a to b

Therefore a F^+b iff n≥1. a F^nb. Moreover

$$F^+ = F \quad F^2 \quad \ldots \quad F^n \quad \ldots$$
$$= \cup_{n\geq 1} F^n$$

DEFINITION. Let R be a binary relation on a set A. The *transitive closure* of R is the binary relation on R defined by

$$R^+ = \cup_{n\geq 1} R^n$$

EXAMPLES. (1) Program modules can import other modules. They can also depend indirectly on modules via some chain of importation, so that for instance M depends on M' if M imports M'' and M'' imports M'. Model this by a set module of modules, and two relations, imports and depends. Clearly

depends = imports$^+$

(2) Two people are related if one is the parent of the other, if they are married, or if there is a chain of such relationships joining them indirectly. Model this by a set people, with three relations: married, parent, and relative. Then

relative = (parent parent^{-1} married)$^+$

Notice that married is symmetric: x married y iff y married x. This is not the case for parent.

R^+ is called the transitive closure because it is transitive and because it is the *smallest* transitive relation containing R. To cast more light on this we now examine an alternative way of building the transitive closure. Imagine that we want to make R transitive, and we want to do this in the most "economical" fashion, by adding as little as possible. If R is already transitive we need do nothing. Otherwise there are a,b,c such that aRbRc but not aRc (so that a,b,c are a counterexample to R being transitive). We add the pair (a,c) into the relation. It is now that much closer to being transitive. We can carry on doing this until there is no need to add further pairs. We now have a transitive relation. Anything we added to R was forced upon us by the

requirement of transitivity, and so we have obtained the smallest transitive relation containing R.

It turns out that the relation we have created is equal to R^+ as defined earlier. We do not prove this here, though we note that it is a straightforward matter to check that R^+ is transitive: Suppose aR^+b, and bR^+c. Then there are paths from a to b and from b to c. Joining these up plainly creates a path from a to c so that aR^+c.

Calculating transitive closures. When calculating "by hand" the easiest method is to draw the directed graph for the given relation R, and then write down the matrix for R^+. To see whether (i,j) should be included, inspect the graph to see if there is a path from the ith node to the jth node. Clearly the word "inspect" is too vague for this to serve as an algorithm, and our informal method is likely to be error prone on large graphs. We now consider how a computer might calculate R^+.

We have defined $R^+ = \cup_{n\geq 1} R^n$. Therefore in order to calculate R^+ we can compute successively

R, R R^2, R R^2 R^3, …

In terms of paths, R R^2 … R^n represents all paths of length between 1 and n. But since R^+ is defined as an infinite union it seems that we will have to carry on computing for ever, which will not do. However the process will come to an end at some finite stage because eventually nothing new will be added. Suppose the set on which R is defined has n elements. Then we need not consider paths of length greater than n since they will involve repeats (visiting the same node of the graph twice). So R^{n+1} is already included in R R^2 … R^n and we need not calculate further as we have found R^+. In fact we often don't have to go as far as n. In the airline example at the beginning of the section there are 8 cities, but the longest paths without repeats are of length 3. Thus we compute F F^2 F^3 and F^4 and we find that F^4 F F^2 F^3, so that we can stop at that point. We may describe our procedure by the following algorithm:

```
Input R
S:= R
T:= R
S:= R∘S
while not S  T do
        T:= T  S
        S:= R∘S
od
Output T
```

In the above whenever the while loop is entered, $S = R^{n+1}$, $T = R \quad R^2 \quad \ldots \quad R^n$ (n=1,2,...). It is not hard to convert the algorithm into a Modula-2 program involving arrays.

There are many ways of improving the algorithm. For instance it is not necessary to store R. A very much more efficient method is *Warshall's algorithm* (see Chapter 16 of *Reasoned Programming*, by K. Broda, S. Eisenbach, H. Khoshnevisan, and S.J. Vickers).

It is sometimes useful to "reverse" the process of finding a transitive closure. In other words, given a transitive relation R, the task is to find a smallest S such that $S^+=R$. The benefit is that S is smaller, while having the same information content as R, since R can be reconstructed from S. However in general there can be many solutions for S. Later on we shall solve this problem in the easier setting of partial orders.

3. Functions

DEFINITION. Given sets A,B, a *function* f from A to B is a method of associating with each $a \in A$ *exactly one* element of B, denoted f(a).

The word "method" is necessarily vague. We intend the notion of function to be grasped directly. Our informal understanding of a function is that it transforms something into something else (a into f(a) in the above definition). This concept is particularly applicable to computer programs, which transform inputs into outputs.

There is a lot of terminology concerning functions, and not all of it is standardised. We let f,g,h range over functions, and we write $f:A \to B$ to mean that f is a function from A to B. A is the *domain* (or *source type*) of f, and B is the *range* (or *co-domain* or *target type*) of f. The expression f(a) is f *applied* to the *argument* a. If f(a) = b, then f is said to *map* a to b. We can also say that b is the *image* of a under f, or that a is a *pre-image* (or *inverse image*) of b under f. Note that elements of the domain always have a single image, but elements of the co-domain can have more than one pre-image or may have none. If the domain A is a product $A_1 \times \ldots \times A_n$ then we write $f(a_1,\ldots,a_n)$ instead of $f((a_1,\ldots,a_n))$. We say that f is a function of n arguments.

When dealing with a function such as $f:\mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^3$, quite often it is convenient not to give it a name, but simply to describe it by its rule for transforming the argument. So in this case we can refer to the function as x^3. If we need to be more careful in indicating the argument we can use the notation $x \mapsto x^3$.

We have defined functions to be *total* (i.e. to have a value for every argument in the domain), following usual mathematical practice. A *partial function* is a function which need not be defined on every member of its domain. It therefore assigns to each element of its domain at

most one element of the range. Miranda programs of course compute functions. On the whole we would like these functions to be total, but it is easy to write programs which do not terminate on some (or all) arguments, so that they define partial functions. Moreover when designing a program P to compute square roots it is quite reasonable to have P return an error message for negative inputs, and to regard P as computing a function which is undefined on negative arguments.

Functions can act on sets other than sets of numbers. For instance let L be (the set of programs in) some programming language. Define a function length: $L \to \mathbf{N}$ by

length(P) = number of lines in P (P an L-program)

A *compiler* transforms programs written in one language L_1 into programs in another language L_2. It may be regarded as a function compile: $L_1 \to L_2$.

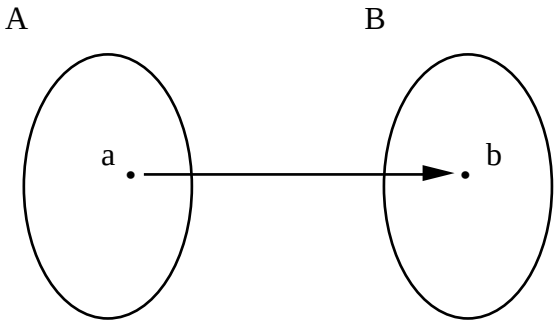
3.1 Representing Functions

Functions can be represented by arrays, diagrams or tables.

(1) *Arrays.* We illustrate by means of an example. Suppose we declare A to be an Array[1..20] of Integer. We mean that the array stores a unique integer for each value from 1 to 20. A therefore represents a function f:[1..20]$\to \mathbb{Z}$ (the mathematical name for the set of integers), where f is defined by

f(i) = A[i] $1 \le i \le 20$

(2) *Diagram.* We can represent f(a) = b by drawing a line from a to b. This is the same diagram as for a relation R from A to B, except that there we were allowed to have a member of A joined to any number of elements of B, which is not allowed for functions.



(3) *Table* . As an example we might record various values of the function age: People$\to \mathbf{N}$. Thus

Name	*Age*
John	21
Kathy	19
Leonard	18

3.2 Properties of Functions.

DEFINITION. Let $f:A \to B$.

(i) f is *onto* iff $\forall b \in B \; \exists a \in A. \; f(a) = b$

(ii) f is *one-one* (1-1) iff $\forall a,a' \in A. \; f(a) = f(a') \quad a=a'$

(iii) f is a *bijection* iff f is both 1-1 and onto (and total)

Part (i) says that every element of the range has a pre-image. Part (ii) can be more easily understood in terms of the following alternative definition

f is *one-one* (1-1) iff $\forall a,a' \in A. \; a \neq a' \quad f(a) \neq f(a')$

In other words no two different elements of the domain can map to the same element of the range. (The second form is the *contrapositive* of the first; in logic $P \to Q$ is equivalent to $\neg Q \to \neg P$).

Notice that 1-1, onto and bijection are universal properties, and so to show that a function does not satisfy them it is enough to produce a counterexample.

EXAMPLE. Define $f:\mathbf{N} \times \mathbf{N} \to N$ by $f(x,y) = x+y$. Show that f is onto but not 1-1.

Proof To show f is onto: Take any $n \in \mathbf{N}$. We must find $(m_1,m_2) \in \mathbf{N} \times \mathbf{N}$ such that $f(m_1,m_2) = n$. But $f(n,0) = n+0 = n$, and so (n,0) is as required.

To show f is not 1-1: We must produce a counterexample, in other words we must find (m_1,m_2), (n_1,n_2) such that $(m_1,m_2) \neq (n_1,n_2)$ but $f(m_1,m_2)= f(n_1,n_2)$. There are many possibilities: For instance $(1,0) \neq (0,1)$ but $f(1,0) = 1+0 = 1$ and $f(0,1) = 0+1 = 1$. In general since + is commutative, $f(m,n) = f(n,m)$ and so (m,n), (n,m) is a counterexample whenever $m \neq n$.

3.3 Functions and Relations

A function $f: A \to B$ can always be regarded as a relation $R \subseteq A \times B$. We simply define

$R(a,b)$ iff $f(a) = b$

But not every relation converts to a function like this. The reason is that with a relation R, for a given $a \in A$ we allow there to be several b's such that R(a,b) or none. But with a function we insist that there must be exactly one b such that f(a) = b.

However there is a different way of converting a relation into a function which always works.

DEFINITION. Suppose $R \subseteq A_1 \times \ldots \times A_n$. The characteristic function $ch_R: A_1 \times \ldots \times A_n \to \{T,F\}$ is defined by

$$ch_R(a_1,\ldots,a_n) = \begin{cases} T & \text{if } R(a_1,\ldots,a_n) \\ F & \text{otherwise} \end{cases}$$

In the binary case the characteristic function is clearly much the same as the corresponding matrix, which was defined by

$$M(i,j) = \begin{cases} T & \text{if } R(a_i,b_j) \\ F & \text{otherwise} \end{cases}$$

3.4 Images of Sets

Functions apply to individuals, but we can also talk of the image of a set X under a function f. This is the collection of all images of members of X under f. If P is a program which computes f, and X is some set of various inputs which are fed to P, then we may wish to know what are the possible outputs.

DEFINITION. Let f: A→B. For any X A, define the *image* of X under f to be

$$f[X] = \{f(a) \mid a \in X\}$$

The set f[A] of all images of f is called the *image set* of f (some authors refer to this as the *range* of f, conflicting with our earlier definition).

Clearly f[X] B. Furthermore f[A] = B iff f is onto. Notice that we effectively have a new function g: **P**A→**P**B defined by

$$g(X) = f[X]$$

3.5 The Pigeonhole Principle and cardinality

Suppose that m objects are to be placed in n pigeonholes, where m>n. Then some pigeonhole will have to contain more than one object. This is called the *Pigeonhole Principle (PP)*.

EXAMPLE. If there are at least 367 people in a room, then at least two share a birthday.

Let us rephrase the PP in the language of functions: Let O be the set of objects and P the set of pigeonholes. An assignment of objects to pigeonholes is described by a function place: O→P. The PP states that if |O| > |P| then place is not 1-1.

PROPOSITION. Let f: A→B, X A. Then $|f[X]| \leq |X|$.

Clearly there cannot be more images of a set X than there are members of X, since each member has at most one image. In fact we can prove the Proposition using the PP:

Suppose for a contradiction that |f[X]| > |X|. Define a place function p: f[X]→X by

p(b) = some a such that f(a) = b

It does not matter which a we choose, but there clearly will be such an a by definition of f[X]. We are placing the members of f[X] in the pigeonholes X. By the PP, some pigeonhole has at least two occupants. In other words there is some a X and b,b' f[X] with p(b) = p(b') = a. But then f(a) = b and f(a) = b', which is a contradiction. □

Given a function f: A → B in general the cardinalities of A and B may be greater than or smaller than each other. However we can relate them as follows:

PROPOSITION. Let f: A → B where A,B are finite.
(a) If f is 1-1 then |A| ≤ |B|
(b) If f is onto then |A| ≥ |B|
(c) If f is a bijection then |A| = |B|

Proof (a) is the contrapositive of the PP. For (b) notice that if f is onto then f[A] = B, so that in particular |f[A]| = |B|. Also |A| ≥ |f[A]| by the preceding Proposition. Therefore |A| ≥ |B| as required. Finally (c) clearly follows from combining (a) and (b). □

3.6 Composition of functions

Composing functions by applying them successively is a well-known procedure. Thus g(f(x)) represents the result of applying f to x and then applying g to the result. In order for this to be defined the value x must be in the domain of f, and f(x) must be in the domain of g. Sometimes it is useful to give the new function taking x to g(f(x)) an explicit name:

DEFINITION. Let f: A → B, g: B → C. A new function g∘f: A → C is defined by

g∘f(x) = g(f(x))

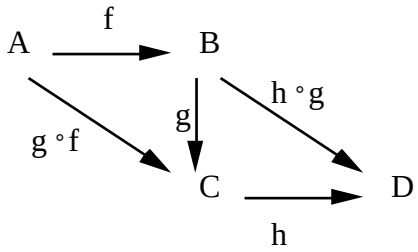
Notice that the range of f has to be equal to the domain of g. Notice also that g∘f represents "f followed by g", unlike in the case of relational composition, where R∘S represented "R followed by S". Just as for relations, function composition is associative but not commutative.

PROPOSITION. Let f: A → B, g: B → C, h: C → D. Then h∘(g∘f) = (h∘g)∘f.

Proof. This is easily established from the definition of functional composition: Take x A. Then

(h∘(g∘f))(x) = h((g∘f)(x)) = h(g(f(x))) = (h∘g)(f(x)) = ((h∘g)∘f)(x)

The proof may be illustrated by this diagram.



Each of the two triangles ABC, BDC "commutes", that is, the result is the same whether one does f followed by g or g∘f, and whether one does g followed by h or h∘g.

Therefore the parallelogram ABDC commutes. □

PROPOSITION. Let f: A→B, g: B→C. If f,g are bijections, then so is g∘f.

Proof. Clearly g∘f is total. It is enough to show:
(1) if f,g are onto then so is g∘f
(2) if f,g are 1-1 then so is g∘f

The result will then follow.

Proof of (1): Assume f,g onto. Take any c C. Since g is onto we can take b B such that g(b) = c. Since f is onto we can take a A such that f(a) = b. But then g∘f(a) = g(f(a)) = g(b) = c. Hence g∘f is onto.

Proof of (2): Assume f,g are 1-1. Take any a_1,a_2 A and suppose g∘f(a_1) = g∘f(a_2). Then

g(f(a_1)) = g(f(a_2))	definition of g∘f
f(a_1) = f(a_2)	since g is 1-1
a_1 = a_2	since f is 1-1

This shows that g∘f is 1-1. □

3.7 Inverses of Functions

The inverse of a function f: A→B is a function g: B→A which undoes the action of f. It turns out that unlike in the case of relations, which always have inverses, a function is only invertible if it is a bijection.

DEFINITION. Let A be a set. Define the *identity function* on A, id_A: A→A, by id_A(a) = a, all a A.

DEFINITION. Let f: A→B. A function g: B→A is an *inverse* of f if

for all a A	g(f(a)) = a
for all b B	f(g(b)) = b

This is clearly equivalent to g∘f = id_A, f∘g = id_B.

PROPOSITION. Let f: A→B be a bijection, and define f^{-1}: B→A by

$f^{-1}(b)$ = the unique a such that f(a) = b

Then f^{-1} is well-defined, and is an inverse of f (in fact *the* inverse in view of the next Proposition).

Proof . There is at least one a such that f(a) = b since f is onto. There cannot be more than one since f is 1-1. Therefore f^{-1} is well-defined. It clearly satisfies the conditions for being an inverse of f. □

PROPOSITION. Let f: A→B. If f has an inverse g, then f is a bijection and the inverse is unique (and is f^{-1} as defined above).

Proof . Assume f has an inverse g.
To show that f is *onto,* take b B. Then f(g(b)) = b. So b is an image of f.
To show that f is *1-1*, take a_1,a_2 A. Suppose $f(a_1) = f(a_2)$. Then $g(f(a_1)) = g(f(a_2))$. But then $a_1 = a_2$.
To show that the inverse is *unique,* suppose that g,g' are both inverses of f. We must show that g=g'. Take any b B. Then f(g(b)) = f(g'(b)) since g,g' are inverses. Hence g(b) = g'(b) since f is 1-1. □

In view of the preceding proposition, one way of showing that a function is a bijection is to show that it has an inverse. Furthermore, if f is a bijection with inverse f^{-1}, then f^{-1} has an inverse, namely f, so that f^{-1} is also a bijection.

EXAMPLES. (1) f:ℝ→ℝ defined by $f(x) = x^3$. The inverse is $f^{-1}(x) = \sqrt[3]{x}$.

(2) f:**N** →**N** defined by

$$f(x) = \begin{cases} x+1 & \text{if } x \text{ even} \\ x-1 & \text{if } x \text{ odd} \end{cases}$$

Then one can check that f∘f(x) = x, considering cases as to whether x is even or odd. So f is its own inverse, and we deduce that f is a bijection.

DEFINITION. Let A be a set. A relation R A× A is an *equivalence relation* iff R is symmetric, reflexive and transitive.

DEFINITION. For any sets X,Y define

X~Y iff there is a bijection from X to Y

We speak of X being *similar* to Y, or being in *1-1 correspondence* with Y.

PROPOSITION. The relation ~ is an equivalence.

Proof. ~ is reflexive: id_X: $X \to X$ is clearly a bijection.
~ is symmetric: If X~Y then there is a bijection f:$X \to Y$. This has an inverse f^{-1} which is also a bijection.
~ is transitive: This comes immediately from the Proposition at the end of Section 3.6. □

3.8 Counting Sets

Let X be a finite set with cardinality n. Then there is a "counting" bijection

$$f: \{1,2,...,n\} \to X$$

Sometimes we express this by enumerating X as $\{a_1,\ldots,a_n\}$. Let X,Y be two finite sets. Then

$$|X| = |Y| \text{ iff } X \sim Y$$

In other words two sets have the same number of elements precisely if there is a 1-1 correspondence between them.

For computing it is convenient to deal with infinite sets, since it is awkward to have for instance a largest natural number. Of course these infinite sets can never be stored in their entirety, and the practical limitations of the machine have to be considered at some stage; but it is natural to consider a program e.g. to multiply numbers without at first insisting that the Maxint of a particular implementation of the programming language be brought into the discussion.

Even though a set such as **N** is infinite, we can build it up by stages

0
0,1
0,1,2
…

in such a way that any number n will appear at some stage. Infinite sets which can be built up in finite portions by stages are particularly nice for computing, since we can hope that our computer will be large enough to store the portion of the set which is needed for the particular input values we wish to process.

Suppose that A is such a set. Then $A = \cup_{n \geq 0} A_n$ where each A_n is finite. We can enumerate A as $\{a_0,a_1,..,a_n,...\}$ by simply adding in the new elements in each A_n, stage by stage. We have a "counting" of A by **N** and A ~ **N**.

DEFINITION. For any set X, X is *countable* iff X is finite or X ~ **N**.

It is in fact the case that X is countable iff there exists a 1-1 function f: $X \to \mathbf{N}$. Thus countable sets are in a sense those which are representable by $\mathbf{N}$.

Surprising things can happen with infinite sets. With finite sets we know that if X is a proper subset of Y, that is, $X \subseteq Y$ and $X \neq Y$, then $|X|<|Y|$ and there can be no bijection between X and Y. But consider the set Even of even natural numbers. Then Even is a proper subset of $\mathbf{N}$. But there is actually a bijection between Even and N, namely $f:N \to$ Even defined by $f(n) = 2n$. In fact any infinite set can be put in bijection with a proper subset of itself.

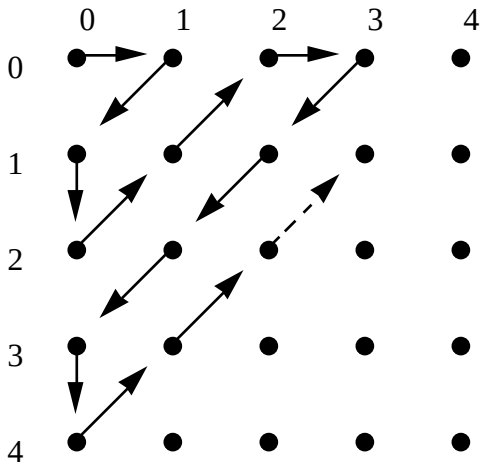
Perhaps more surprisingly still, we can count sets which appear larger than $\mathbf{N}$. For instance the integers $\mathbb{Z}$ may be counted as

0,1,-1,2,-2,3,-3,...

This can be achieved by the bijection $g:\mathbb{Z} \to \mathbf{N}$ defined by

$$g(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ -1-2x & \text{if } x < 0 \end{cases}$$

$\mathbb{Z}$ is like 2 copies of $\mathbf{N}$. We can even count $\mathbf{N}^2$, which is like $\mathbf{N}$ copies of $\mathbf{N}$ (see the following diagram).



Cantor showed that there are *uncountable* sets – sets which are too large to be countable. If X is uncountable then no matter how you try to count it you will always miss out some elements. In other words, if f is any 1-1 map from $\mathbf{N}$ into X then f is not onto.

It is important to realise that a set can still be countable even if a particular attempt to count it fails. For instance we have just seen that $\mathbf{N}^2$ is countable, but the function $f: \mathbf{N} \to \mathbf{N}^2$ defined by $f(n) = (n,0)$ is 1-1 but not onto. If a set X is uncountable then by definition *every* attempt to count it fails, and trying to add more elements to an enumeration will not help.

The most important uncountable set is the reals $\mathbb{R}$. In view of our earlier discussion, we cannot build up the reals by finite stages, and this means that we cannot manipulate actual reals in the way that we can natural numbers. What we have to do is use approximations, such as floating point decimals. There are only countably many such approximations.

FACT. $\mathbf{P}(\mathbf{N})$ is uncountable – in fact so is $\mathbf{P}(X)$ for any infinite X.

For further information on infinite sets and countability see Truss, Section 2.4.

4. More about relations

In this section we discuss two particular kinds of relation, which arise when we wish to *compare* objects, namely *equivalence relations* (which tell us when one object is "as good as" another) and *orderings* (which tell us when one object is "better" than another).

4.1 Equivalence relations

Imagine for example that we have a set of programs and we have various demands to make of them – various properties which they may or may not possess. These might be such things as

- always terminates
- costs less than £100
- computes π to 100 decimal places

etc. We deem two programs to be equivalent if they meet the same demands. If this is the case then even though they are not equal they may as well be, because there is no difference between them as far as our demands are concerned. So what we have is a weakened form of equality. It is not hard to see that this equivalence must be reflexive, symmetric and transitive. This motivates the following definition, which is as in Section 3.7:

DEFINITION. Let A be a set. A relation $R \subseteq A \times A$ is an *equivalence relation* iff R is reflexive, symmetric and transitive.

NOTATION. Equivalence relations are often denoted by ~, E, etc.

EXAMPLES. (1) Let A be any set. The identity (equality) relation id_A is an equivalence relation.

(2) Given a set Student and a map age: Student $\to \mathbf{N}$, the binary relation on Student defined by

$$s_1 \text{ sameAge } s_2 \text{ iff } \mathrm{age}(s_1) = \mathrm{age}(s_2)$$

is an equivalence.

(3) Similarity ~ between sets (see Section 3.7 above).

(4) Logical equivalence between formulas defined by

$A \equiv B$ iff $\models A \leftrightarrow B$

is an equivalence.

(5) Permutation of lists: The relation

$l_1 \, E \, l_2$ iff l_2 is a permutation of l_1

is an equivalence.

Example (2) is a special case of the following:

PROPOSITION. Let $f: A \to B$ and let E be an equivalence relation on B. Then the relation R $A \times A$ defined by

$a_1 \, R \, a_2$ iff $f(a_1) \, E \, f(a_2)$

is also an equivalence.

Proof Exercise □

DEFINITION. Let E be an equivalence relation on a set A. For any a A the *equivalence class* of a (with respect to E) is defined as

$[a]_E = \{x \quad A \mid aEx\}$

We often write [a] where no confusion will arise.

THEOREM. Let E be an equivalence relation on A. Then for all a_1, a_2 A,

$[a_1]=[a_2]$ iff $a_1 E a_2$

Moreover the family of sets [a] (all a A) forms a partition of A.
Conversely, given a partition $A_1, \ldots, A_n$ of A we can define a relation R on A by

xRy iff x,y are in the same component of the partition

and this is an equivalence. □

4.2 Orderings

Orderings on sets of numbers such as $\mathbf{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are familiar. But we can also have orderings on other sets. For instance suppose that we have a set of programs and we wish to compare them as to which are cheaper, or run faster, or are more accurate. We shall use these sort of orderings in Part 2 when we compare the efficiency of algorithms. We can safely assert that any ordering relation should be *transitive*:

if a<b and b<c then a<c

and it will not be *symmetric* (so a<b should not imply b<a) in any interesting case. As to the *reflexive* property we notice that there are two types of orderings, typified by < and ≤ on $\mathbf{N}$. The latter is reflexive, but the former is not – in fact it is *irreflexive* (defined below).

We discuss three notions of ordering – pre-orders, partial orders and linear orders.

DEFINITION. Let R be a binary relation on a set A. Then R is a *pre-order* iff R is reflexive and transitive.

We write (A,R) when we wish to show the underlying set.

EXAMPLES (of pre-orders). (1) Define a relation on formulas by

$A \leq B$ iff $\models A \rightarrow B$

Then ≤ is a pre-order. For instance ff ≤ A ≤ tt (any A), and A≤ A B.

(2) The relation on Student defined by

s_1 noOlderThan s_2 iff $age(s_1) \leq age(s_2)$

PROPOSITION. If $f:A \rightarrow B$ and S is a pre-order on B then R defined by

$a_1 R a_2$ iff $f(a_1) S f(a_2)$

is a pre-order on A. □

Many pre-orders satisfy an additional property:

DEFINITION. Let R A× A. R is *anti-symmetric* iff

x,y A. xRy yRx x=y

R is a *partial order* (po for short) iff R is reflexive, transitive and anti-symmetric.

In other words, a po is an anti-symmetric pre-order. It should be noted that anti-symmetric is not the opposite of symmetric, since a relation can be both symmetric and anti-symmetric (for example the empty relation or the identity relation on any set)

EXAMPLES (of partial orders). (1) $(\mathbf{N},\leq)$ $(\mathbb{Z},\leq)$ $(\mathbb{R},\leq)$

(2) on sets

(3) Suppose $(A,\leq)$ is a po and B A. Then $(B,\leq)$ is a po, where we restrict $\leq$ to B, i.e. take $\leq \cap\ B\times B$. For instance, since $(\mathbf{N},\leq)$ is a po, so is $(\{1,2,3\},\leq)$.

(4) Let $(A,\leq)$ be a po. Define $\geq$ by

$a\geq b$ iff $b\leq a$

(this is just the inverse relation). Then $(A,\geq)$ is a po. Thus is a po on sets.

(5) Let $\mathbb{Z}^+$ be the integers >0. Define m|n iff m divides n exactly. Then $(\mathbb{Z}^+,|)$ is a po. What can you say about where prime numbers occur in this ordering?

DEFINITION. R $A\times A$ is *irreflexive* iff a A. not aRa.

R is a *strict* (strong, irreflexive) po iff R is irreflexive and transitive. The usual po's are called *non-strict* (weak, reflexive) to distinguish them. Strict po's are often denoted by <.

EXAMPLES. $(\mathbf{N},<)$ $(\mathbb{R},<)$

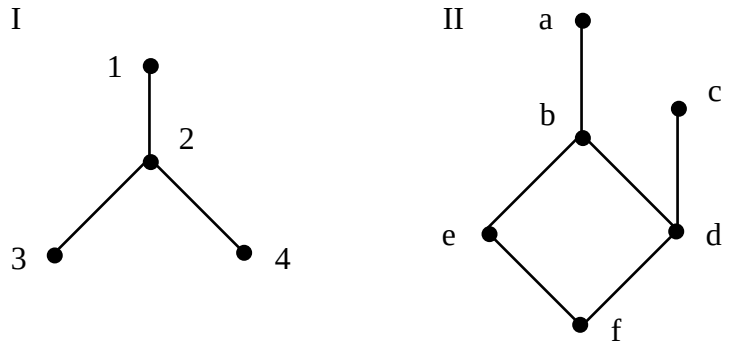
We can convert a strict po < into a non-strict one by defining

$a\leq b$ iff $a<b$ or $a=b$

Conversely we can convert a non-strict po $\leq$ into a strict one by defining

$a<b$ iff $a\leq b$ and $a\neq b$

When we draw diagrams of pos it is natural to use directed graphs, since pos are binary relations. These diagrams get rather cluttered if every arrow is drawn in. So it is usual to omit the arrows from points to themselves and those arrows that can be deduced from knowing that the relation is transitive. Such diagrams are called *Hasse* diagrams. In the following two examples the direction of the lines has been omitted, with the convention that all lines are directed upwards, in other words the upper element is greater than the lower.



In diagram I for instance 3<1, since this can be deduced from 3<2<1. Of course this is not the usual ordering on {1,2,3,4}. We also know 1≤1 even though this is not specified in the diagram.

We can make the connection between the relation in the diagrams and the po's precise as follows:

DEFINITION. Given a po $(A,\leq)$ define the *immediate predecessor* relation by

$$x <_I y \text{ iff } x<y \text{ and there is no } z \text{ such that } x<z<y$$

The immediate predecessor relation is what we usually draw in diagrams instead of ≤. If A is finite it is clear that for any pair (x,y) we can calculate whether $x <_I y$ by simply checking through all possible values of z. The ordering ≤ can be recovered from $<_I$ by taking its reflexive and transitive closure:

$$\leq = (<_I)^* = (id_A \quad <_I^+)$$

(In general the reflexive and transitive closure of a relation R is denoted R*.) We have in fact solved the problem mentioned at the end of Section 2.5, as $<_I$ is the smallest relation S such that $S^+ = <$. Irreflexivity is crucial here. Also this would not hold in general when A was infinite—consider for instance the real numbers with the usual < ordering.

DEFINITION. A po $(A,\leq)$ is a *total* ordering iff $\quad x,y \quad A.\ x\leq y \quad y\leq x.$
This may equivalently be stated: $\quad x,y \quad A.\ x<y \quad y<x \quad x=y.$

Total orderings are also called *linear* orderings, because their diagrams are linear.

EXAMPLES. (1) $(\mathbf{N},<)$ $(\mathbb{R},<)$

(2) If $(A,\leq)$ is total and $B \quad A$ then $(B,\leq)$ is total.

Neither of the two diagrams above is of a linear ordering: in I there is no relationship between 3 and 4, while in II, a and c are unrelated, as are e and d.

DEFINITION. Let $(A,\leq)$ be a po.
a is *minimal* iff for all b $\in$ A. $b \not< a$.
a is *least* iff for all b $\in$ A. $a \leq b$.

An element is minimal if there is nothing below it. It is least if it is below everything else. In diagram I, 3 and 4 are both minimal; neither is least. In diagram II, f is least (as well as minimal).

PROPOSITION. If a is least in a po then a is minimal

Proof Suppose a is least. Suppose for a contradiction that $b<a$. But $a \leq b$ since a is least. This contradicts anti-symmetry. □

PROPOSITION. If A has a least element then it is unique.

Proof . Suppose a and b are least. Then $a \leq b$ and $b \leq a$. Hence $a=b$. □

PROPOSITION. Let $(A,\leq)$ be a finite po. Then A has a minimal element.

Proof Pick any $a_0 \in A$. If a_0 is not minimal we can pick $a_1 < a_0$. If a_1 is not minimal we can pick $a_2 < a_1$. In this way we get a decreasing chain $a_0 > a_1 > a_2 > \ldots$ All the elements of the chain are different, since $x>y>z$ implies $x>z$ and so $x \neq z$. Hence since A is finite the process must stop at some a_n ($n \in \mathbf{N}$), which must be minimal in A. □

Notice that in the above proof we not only show the existence of minimal elements, but also how to find one.

PROPOSITION. Let $(A,\leq)$ be a finite linear ordering. Then A has a least element. □

Suppose we have a set of tasks T to perform. We wish to decide in what order to perform them. We are not totally free to choose, because some tasks have to be finished before others can be started. We can express this prerequisite structure by a partial ordering < on T. We want to find a total order <' on T which respects < in the sense that

if $t<u$ then $t<'u$

Such a process is sometimes called *topological sorting* or *linearisation.* A simple algorithm for topological sorting is based on minimal elements. To generate an appropriate order for the tasks, first find all minimal elements of T (this may be done easily from a matrix representation of (T,<) and remove them from T to get T'. These are done first (in some arbitrary order). Then find all minimal elements of T' and do these next. Carry on until T is exhausted. Plainly there may be many different linearisations.

EXAMPLE. Given orderings on A,B we can define an ordering on the Cartesian product A×B as follows:

$(a,b) \leq (a',b')$ iff $a<a'$ or $(a=a' \ \& \ b \leq b')$

In other words, compare the a's first, and if they are the same then compare the b's. For instance we might wish to sort student records first by year and next alphabetically within a year. (NB There are other possible orderings on A×B.)

EXERCISE. Let $(A,\leq)$, $(B,\leq)$ be linear orderings. Show that $(A\times B,\leq)$ as defined above is a linear ordering.

EXAMPLE. Words in a dictionary (strings of characters) are ordered by the *lexicographic ordering*, which resembles the ordering on A×B defined above. This ordering has the interesting feature that every word w has an immediate successor: $w <_I wa$ (for example $sag <_I saga$). However immediate *predecessors* do not necessarily exist. For instance "cat" does not have one. Why not? Which words do have an immediate predecessor?

4.3 Termination and orderings

How do we establish that a program terminates? The possibility of non-termination usually arises from a while loop or a recursive call. To deal with while loops we use a variant—a counter which is *decreased* on every pass through the loop. To deal with recursive calls we can check that the function or procedure is called with a *smaller* argument. Thus clearly

```
fac(0) = 1
fac(n+1) = n*fac(n)
```

terminates, while

```
nonterm(n) = n*nonterm(n+1)
```

does not.

To use an ordering $(A,\leq)$ to establish termination we need to know that we cannot keep on getting smaller and smaller values in A for ever. For instance the integers **Z** are no use as one can keep on decreasing an integer value for ever. Thus the following attempt at a multiplication program for integers is invalid as evaluation will not terminate:

```
mult((x,y) = y + mult(x-1,y)
```

Plainly we cannot have infinite decreasing chains

$$a_0 > a_1 > a_2 > \ldots > a_n > \ldots$$

DEFINITION. An ordering $(A,\leq)$ is *wellfounded* (wf) if it has no infinite decreasing chain.

Obviously every finite ordering is wellfounded. But there are infinite wf orderings. The basic one is $\mathbf{N}$, and of course this is the one usually used for variants.

EXAMPLE. Ackermann's function $\mathrm{Ack}:\mathbf{N}\times\mathbf{N}\to\mathbf{N}$

$$\mathrm{Ack}(0,y) = y+1$$
$$\mathrm{Ack}(x+1,0) = \mathrm{Ack}(x,1)$$
$$\mathrm{Ack}(x+1,y+1) = \mathrm{Ack}(x,\mathrm{Ack}(x+1,y))$$

The third equation causes this function computed by this program to grow extremely rapidly.

We wish to prove that this program always terminates, and therefore defines a total function. Counting down on x is not good enough, since the third equation does not decrease x+1, because of the embedded Ack(x+1,y). Instead we use the ordering on $\mathbf{N}\times\mathbf{N}$ defined above:

$$(x,y) < (x',y') \text{ iff } x<x' \text{ or } (x=x' \ \&\ y<y')$$

Notice that

$$(x+1,0) > (x,1)$$
$$(x+1,y+1) > (x,\mathrm{Ack}(x+1,y))$$
$$(x+1,y+1) > (x+1,y)$$

and so evaluation takes us down in the ordering. Moreover the ordering is wellfounded—this can be proved using the fact that $\mathbf{N}$ is wf. Even though a member such as (4,3) has infinitely many others below it (e.g. (3,y) for every y), any decreasing chain must be finite.