Notes on Discrete Mathematics

Miguel A. Lerma

Contents

Introduction

These notes are intended to be a summary of the main ideas in course CS 310: *Mathematical Foundations of Computer Science*. I may keep working on this document as the course goes on, so these notes will not be completely finished until the end of the quarter.

The textbook for this course is Richard Johnsonbaugh:*Discrete Mathematics*, Fifth Edition, 2001, Prentice Hall. With few exceptions I will follow the notation in the book.

These notes contain some questions and "exercises" intended to stimulate the reader who wants to play a somehow active role while studying the subject. They are not homework nor need to be addressed at all if the reader does not wish to. I will recommend exercises and give homework assignments separately.

Finally, if you find any typos or errors, or you have any suggestions, please, do not hesitate to tell me.

Miguel A. Lerma
mlerma@math.northwestern.edu
Northwestern University
Winter 2004

CHAPTER 1

Logic

1.1. Propositions

A *proposition* is a declarative sentence that is either true or false (but not both). For instance, the following are propositions: "Paris is in France" (true), "London is in Denmark" (false), "$2 < 4$" (true), "$4 = 7$ (false)". However the following are not propositions: "what is your name?" (this is a question), "do your homework" (this is a command), "this sentence is false" (neither true nor false), "x is an even number" (it depends on what x represents), "Socrates" (it is not even a sentence). The truth or falsehood of a proposition is called its *truth value*.

1.1.1. Connectives, Truth Tables.

Connectives are used for making *compound propositions*. The main ones are the following (p and q represent given propositions):

Name	**Represented**	**Meaning**
Negation	$\overline{p}$	"not p"
Conjunction	$p \wedge q$	"p and q"
Disjunction	$p \vee q$	"p or q (or both)"
Exclusive Or	$p \veebar q$	"either p or q, but not both"
Implication	$p \to q$	"if p then q"
Biconditional	$p \leftrightarrow q$	"p if and only if q"

The truth value of a compound proposition depends only on the value of its components. Writing F for "false" and T for "true", we can summarize the meaning of the connectives in the following way:

p	q	$\overline{p}$	$p \wedge q$	$p \vee q$	$p \veebar q$	$p \to q$	$p \leftrightarrow q$
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

Note that $\vee$ represents a *non-exclusive* or, i.e., $p \vee q$ is true when any of p, q is true and also when both are true. On the other hand $\veebar$ represents an *exclusive* or, i.e., $p \veebar q$ is true only when exactly one of p and q is true.

1.1.2. Conditional Propositions. A proposition of the form "if p then q" or "p implies q", represented "$p \to q$" is called a *conditional proposition.* For instance: "if John is from Chicago then John is from Illinois". The proposition p is called *hypothesis* or *antecedent*, and the proposition q is the *conclusion* or *consequent.*

Note that $p \to q$ is true always except when p is true and q is false. So, the following sentences are true: "if $2 < 4$ then Paris is in France" (true $\to$ true), "if London is in Denmark then $2 < 4$" (false $\to$ true), "if $4 = 7$ then London is in Denmark" (false $\to$ false). However the following one is false: "if $2 < 4$ then London is in Denmark" (true $\to$ false).

In might seem strange that "$p \to q$" is considered true when p is false, regardless of the truth value of q. This will become clearer when we study *predicates* such as "if x is a multiple of 4 then x is a multiple of 2". That implication is obviously true, although for the particular case $x = 3$ it becomes "if 3 is a multiple of 4 then 3 is a multiple of 2".

The proposition $p \leftrightarrow q$, read "p if and only if q", is called *biconditional.* It is true precisely when p and q have the same truth value, i.e., they are both true or both false.

1.1.3. Logical Equivalence. Note that the compound propositions $p \to q$ and $\overline{p} \vee q$ have the same truth values:

p	q	$\overline{p}$	$\overline{p} \vee q$	$p \to q$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

When two compound propositions have the same truth values no matter what truth value their constituent propositions have, they are called *logically equivalent*. For instance $p \to q$ and $\overline{p} \vee q$ are logically equivalent, and we write it:

$$p \to q \equiv \overline{p} \vee q$$

Example: De Morgan's Laws for Logic. The following propositions are logically equivalent:

$$\overline{p \vee q} \equiv \overline{p} \wedge \overline{q}$$
$$\overline{p \wedge q} \equiv \overline{p} \vee \overline{q}$$

We can check it by examining their truth tables:

p	q	$\overline{p}$	$\overline{q}$	$p \vee q$	$\overline{p \vee q}$	$\overline{p} \wedge \overline{q}$	$p \wedge q$	$\overline{p \wedge q}$	$\overline{p} \vee \overline{q}$
T	T	F	F	T	F	F	T	F	F
T	F	F	T	T	F	F	F	T	T
F	T	T	F	T	F	F	F	T	T
F	F	T	T	F	T	T	F	T	T

Example: The following propositions are logically equivalent:

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

Again, this can be checked with the truth tables:

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

Exercise: Check the following logical equivalences:

$$\overline{p \to q} \equiv p \wedge \overline{q}$$
$$p \to q \equiv \overline{q} \to \overline{p}$$
$$\overline{p \leftrightarrow q} \equiv p \veebar q$$

1.1.4. Converse, Contrapositive. The *converse* of a conditional proposition $p \to q$ is the proposition $q \to p$. As we have seen, the biconditional proposition is equivalent to the conjunction of a conditional proposition an its converse.

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

So, for instance, saying that “John is married if and only if he has a spouse” is the same as saying “if John is married then he has a spouse” *and* “if he has a spouse then he is married”.

Note that the converse is not equivalent to the given conditional proposition, for instance “if John is from Chicago then John is from Illinois” is true, but the converse “if John is from Illinois then John is from Chicago” may be false.

The *contrapositive* of a conditional proposition $p \to q$ is the proposition $\overline{q} \to \overline{p}$. They are logically equivalent. For instance the contrapositive of “if John is from Chicago then John is from Illinois” is “if John is not from Illinois then John is not from Chicago”.

1.2. Quantifiers

1.2.1. Predicates. A *predicate* or *propositional function*[1] is a statement containing variables. For instance "$x + 2 = 7$", "X is American", "$x < y$", "p is a prime number" are predicates. The truth value of the predicate depends on the value assigned to its variables. For instance if we replace x with 1 in the predicate "$x+2 = 7$" we obtain "$1+2 = 7$", which is false, but if we replace it with 5 we get "$5 + 2 = 7$", which is true. We represent a predicate by a letter followed by the variables enclosed between parenthesis: $P(x)$, $Q(x, y)$, etc. An *example* for $P(x)$ is a value of x for which $P(x)$ is true. A *counterexample* is a value of x for which $P(x)$ is false. So, 5 is an example for "$x + 2 = 7$", while 1 is a counterexample.

Each variable in a predicate is assumed to belong to a *domain (or universe) of discourse*, for instance in the predicate "n is an odd integer" 'n' represents an integer, so the domain of discourse of n is the set of all integers. In "X is American" we may assume that X is a human being, so in this case the domain of discourse is the set of all human beings.[2]

1.2.2. Quantifiers. Given a predicate $P(x)$, the statement "for some x, $P(x)$" (or "there is some x such that $p(x)$"), represented "$\exists x\, P(x)$", has a definite truth value, so it is a proposition in the usual sense. For instance if $P(x)$ is "$x + 2 = 7$" with the integers as domain of discourse, then $\exists x\, P(x)$ is true, since there is indeed an integer, namely 5, such that $P(5)$ is a true statement. However, if $Q(x)$ is "$2x = 7$" and the domain of discourse is still the integers, then $\exists x\, Q(x)$ is false. On the other hand, $\exists x\, Q(x)$ would be true if we extend the domain of discourse to the rational numbers. The symbol $\exists$ is called the *existential quantifier*.

[1]The term *propositional function* used by Johnsonbaugh is rather obsolete and I have replaced it here with the more currently used *predicate*.

[2]Usually all variables occurring in predicates along a reasoning are supposed to belong to the *same* domain of discourse, but in some situations (as in the so called *many-sorted* logics) it is possible to use different kinds of variables to represent different types of objects belonging to different domains of discourse. For instance in the predicate "σ is a string of length n" the variable σ represents a string, while n represents a natural number, so the domain of discourse of σ is the set of all strings, while the domain of discourse of n is the set of natural numbers.

Analogously, the sentence “for all x, $P(x)$”—also “for any x, $P(x)$”, “for every x, $P(x)$”, “for each x, $P(x)$”—, represented “$\forall x\, P(x)$”, has a definite truth value. For instance, if $P(x)$ is “$x+2=7$” and the domain of discourse is the integers, then $\forall x\, P(x)$ is false. However if $Q(x)$ represents “$(x+1)^2 = x^2+2x+1$” then $\forall x\, Q(x)$ is true. The symbol $\forall$ is called the *universal quantifier*.

In predicates with more than one variable it is possible to use several quantifiers at the same time, for instance $\forall x \forall y \exists z\, P(x,y,z)$, meaning “for all x and all y there is some z such that $P(x,y,z)$”.

Note that in general the existential and universal quantifiers cannot be permuted, i.e., in general $\forall x \exists y\, P(x,y)$ means something different from $\exists y \forall x\, P(x,y)$. For instance if x and y represent human beings and $P(x,y)$ represents “x is married to y”, then $\forall x \exists y\, P(x,y)$ means that everybody is married to someone, but $\exists y \forall x\, P(x,y)$ means that there is someone to whom everybody else is married (a extreme form of polygamy!).

A predicate can be partially quantified, e.g. $\forall x \exists y\, P(x,y,z,t)$. The variables quantified (x and y in the example) are called *bound* variables, and the rest (z and t in the example) are called *free* variables. A partially quantified predicate is still a predicate, but depending on fewer variables.

1.2.3. Generalized De Morgan Laws for Logic. If $\exists x\, P(x)$ is false then there is no value of x for which $P(x)$ is true, or in other words, $P(x)$ is always false. Hence

$$\overline{\exists x\, P(x)} \equiv \forall x\, \overline{P(x)}\,.$$

On the other hand, if $\forall x\, P(x)$ is false then it is not true that for every x, $P(x)$ holds, hence for some x, $P(x)$ must be false. Thus:

$$\overline{\forall x\, P(x)} \equiv \exists x\, \overline{P(x)}\,.$$

This two rules can be applied in successive steps to find the negation of a more complex quantified statement, for instance:

$$\overline{\exists x \forall y\, p(x,y)} \equiv \forall x \overline{\forall y\, P(x,y)} \equiv \forall x \exists y\, \overline{P(x,y)}\,.$$

Exercise: Write formally the statement “for every real number there is a greater real number”. Write the negation of that statement.

Answer: The statement is: $\forall x\,\exists y\,(x < y)$ (the domain of discourse is the real numbers). Its negation is: $\exists x\,\forall y\,\overline{x < y}$, i.e., $\exists x\,\forall y\,(x \not< y)$. (Note that among real numbers $x \not< y$ is equivalent to $x \geq y$, but formally they are different predicates.)

1.3. Proofs

1.3.1. Mathematical Systems, Proofs. A *Mathematical System* consists of:

1. *Axioms*: propositions that are assumed true.
2. *Definitions*: used to create new concepts from old ones.
3. *Undefined terms*: corresponding to the primitive concepts of the system (for instance in set theory the term "set" is undefined).

A *theorem* is a proposition that has been proved to be true. An argument that establishes the truth of a proposition is called a *proof*.

Example: Prove that if $x > 2$ and $y > 3$ then $x + y > 5$.

Answer: Assuming $x > 2$ and $y > 3$ and adding the inequalities term by term we get: $x + y > 2 + 3 = 5$.

That is an example of *direct proof*. In a direct proof we assume the hypothesis together with axioms and other theorems previously proved and we derive the conclusion from them.

Proof by Contradiction. In a *proof by contradiction* or (*Reductio ad Absurdum*) we assume the hypothesis and the negation of the conclusion, and try to derive a *contradiction*, i.e., a proposition of the form $r \wedge \overline{r}$.

Example: Prove by contradiction that if $x+y > 5$ then either $x > 2$ or $y > 3$.

Answer: We assume the hypothesis $x + y > 5$. From here we must conclude that $x > 2$ or $y > 3$. Assume to the contrary that "$x > 2$ or $y > 3$" is false, so $x \leq 2$ and $y \leq 3$. Adding those inequalities we get $x \leq 2 + 3 = 5$, which contradicts the hypothesis $x + y > 5$. From here we conclude that the assumption "$x \leq 2$ and $y \leq 3$" cannot be right, so "$x > 2$ or $y > 3$" must be true.

A related proof is the *proof by contrapositive*, i.e., instead of proving $p \to q$ we prove the contrapositive $\overline{q} \to \overline{p}$.

1.3.2. Arguments, Rules of Inference. An *argument* is a sequence of propositions $p_1, p_2, \dots, p_n$ called *hypothesis* (or *premises*) followed by a proposition q called *conclusion*. An argument is usually written:

$$\begin{array}{rl} & p_1 \\ & p_2 \\ & \vdots \\ & p_n \\ \hline \therefore & q \end{array}$$

or

$$p_1, p_2, \dots, p_n / \therefore q$$

The argument is called *valid* if q is true whenever $p_1, p_2, \dots, p_n$ are true; otherwise it is called *invalid*.

Rules of inference are certain simple arguments known to be valid and used to make a proof step by step. For instance the following argument is called *modus ponens* or *rule of detachment*:

$$\begin{array}{rl} & p \rightarrow q \\ & p \\ \hline \therefore & q \end{array}$$

In order to check whether it is valid we must examine the following truth table:

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

If we look now at the rows in which both $p \rightarrow q$ and p are true (just the first row) we see that also q is true, so the argument is valid.

Other rules of inference are the following:

1. *Modus Ponens* or *Rule of Detachment*:

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

2. *Modus Tollens*:

$$\begin{array}{rl} & p \to q \\ & \overline{q} \\ \hline \therefore & \overline{p} \end{array}$$

3. *Addition*:

$$\begin{array}{rl} & p \\ \hline \therefore & p \vee q \end{array}$$

4. *Simplification*:

$$\begin{array}{rl} & p \wedge q \\ \hline \therefore & p \end{array}$$

5. *Conjunction*:

$$\begin{array}{rl} & p \\ & q \\ \hline \therefore & p \wedge q \end{array}$$

6. *Hypothetical Syllogism*:

$$\begin{array}{rl} & p \to q \\ & q \to r \\ \hline \therefore & p \to r \end{array}$$

7. *Disjunctive Syllogism*:

$$\begin{array}{rl} & p \vee q \\ & \overline{p} \\ \hline \therefore & q \end{array}$$

Arguments are usually written using three columns. Each row contains a label, a statement and the reason that justifies the introduction of that statement in the argument. That justification can be one of the following:

1. The statement is a *premise*.
2. The statement can be derived from statements occurring earlier in the argument by using a *rule of inference*.

Example: Consider the following statements: "I take the bus or I walk. If I walk I get tired. I do not get tired. Therefore I take the bus.” We can formalize this by calling B = “I take the bus”, W = “I walk” and T = “I get tired”. The premises are $B \vee W$, $W \to T$ and $\overline{T}$, and the conclusion is B. The argument can be described in the following steps:

step	statement	reason
1)	$W \to T$	Premise
2)	$\overline{T}$	Premise
3)	$\overline{W}$	1,2, Modus Tollens
4)	$B \vee W$	Premise
5)	$\therefore B$	4,3, Disjunctive Syllogism

1.3.3. Rules of Inference for Quantified Statements. We state the rules for predicates with one variable, but they can be generalized to predicates with two or more variables.

1. *Universal Instantiation.* If $\forall x\, p(x)$ is true, then $p(a)$ is true for each specific element a in the domain of discourse; i.e.:
$$\begin{array}{r} \forall x\, p(x) \\ \hline \therefore\ \ p(a) \end{array}$$
For instance, from $\forall x\,(x+1 = 1+x)$ we can derive $7+1 = 1+7$.

2. *Existential Instantiation.* If $\exists x\, p(x)$ is true, then $p(a)$ is true for some specific element a in the domain of discourse; i.e.:
$$\begin{array}{r} \exists x\, p(x) \\ \hline \therefore\ \ p(a) \end{array}$$
The difference respect to the previous rule is the restriction in the meaning of a, which now represents some (not any) element of the domain of discourse. So, for instance, from $\exists x\,(x^2 = 2)$ (the domain of discourse is the real numbers) we derive the existence of some element, which we may represent $\pm\sqrt{2}$, such that $(\pm\sqrt{2})^2 = 2$.

3. *Universal Generalization.* If $p(x)$ is proved to be true for a generic element in the domain of discourse, then $\forall x\, p(x)$ is true; i.e.:
$$\begin{array}{r} p(x) \\ \hline \therefore\ \ \forall x\, p(x) \end{array}$$
By “generic” we mean an element for which we do not make any assumption other than its belonging to the domain of discourse. So, for instance, we can prove $\forall x\,[(x+1)^2 = x^2 + 2x + 1]$ (say,

for real numbers) by assuming that x is a generic real number and using algebra to prove $(x+1)^2 = x^2 + 2x + 1$.

4. *Existential Generalization.* If $p(a)$ is true for some specific element a in the domain of discourse, then $\exists x\, p(x)$ is true; i.e.:

$$\begin{array}{rl} & p(a) \\ \hline \therefore & \exists x\, p(x) \end{array}$$

For instance: from $7+1=8$ we can derive $\exists x\,(x+1=8)$.

Example: Show that a counterexample can be used to disprove a universal statement, i.e., if a is an element in the domain of discourse, then from $\overline{p(a)}$ we can derive $\overline{\forall x\, p(x)}$. *Answer*: The argument is as follows:

step	statement	reason
1)	$\overline{p(a)}$	Premise
2)	$\exists x\, \overline{p(x)}$	Existential Generalization
3)	$\overline{\forall x\, p(x)}$	Negation of Universal Statement

1.4. Mathematical Induction

Many properties of positive integers can be proved by mathematical induction.

1.4.1. Principle of Mathematical Induction. Let P be a property of positive integers such that:

1. *Basis Step*: $P(1)$ is true, and
2. *Inductive Step*: if $P(n)$ is true, then $P(n+1)$ is true.

Then $P(n)$ is true for all positive integers.

Remark: The premise $P(n)$ in the inductive step is called *Induction Hypothesis.*

The validity of the Principle of Mathematical Induction is obvious. The basis step states that $P(1)$ is true. Then the inductive step implies that $P(2)$ is also true. By the inductive step again we see that $P(3)$ is true, and so on. Consequently the property is true for all positive integers.

Remark: In the basis step we may replace 1 with some other integer m. Then the conclusion is that the property is true for every integer n greater than or equal to m.

Example: Prove that the sum of the n first odd positive integers is n^2, i.e., $1+3+5+\cdots+(2n-1)=n^2$.

Answer: Let $S(n)=1+3+5+\cdots+(2n-1)$. We want to prove by induction that for every positive integer n, $S(n)=n^2$.

1. *Basis Step*: If $n=1$ we have $S(1)=1=1^2$, so the property is true for 1.
2. *Inductive Step*: Assume (*Induction Hypothesis*) that the property is true for some positive integer n, i.e.: $S(n)=n^2$. We must prove that it is also true for $n+1$, i.e., $S(n+1)=(n+1)^2$. In fact:

$$S(n+1)=1+3+5+\cdots+(2n+1)=S(n)+2n+1\,.$$

But by induction hypothesis, $S(n) = n^2$, hence:

$$S(n+1) = n^2 + 2n + 1 = (n+1)^2 \,.$$

This completes the induction, and shows that the property is true for all positive integers.

Example: Prove that $2n + 1 \leq 2^n$ for $n \geq 3$.

Answer: This is an example in which the property is not true for all positive integers but only for integers greater than or equal to 3.

1. *Basis Step*: If $n = 3$ we have $2n + 1 = 2 \cdot 3 + 1 = 7$ and $2^n = 2^3 = 8$, so the property is true in this case.
2. *Inductive Step*: Assume (*Induction Hypothesis*) that the property is true for some positive integer n, i.e.: $2n + 1 \leq 2^n$. We must prove that it is also true for $n+1$, i.e., $2(n+1)+1 \leq 2^{n+1}$. By the induction hypothesis we know that $2n \leq 2^n$, and we also have that $3 \leq 2^n$ if $n \geq 3$, hence
$$2(n+1) + 1 = 2n + 3 \leq 2^n + 2^n = 2^{n+1} \,.$$

This completes the induction, and shows that the property is true for all $n \geq 3$.

Exercise: Prove the following identities by induction:

- $1 + 2 + 3 + \cdots + n = \dfrac{n\,(n+1)}{2}$.
- $1^2 + 2^2 + 3^2 + \cdots + n^2 = \dfrac{n\,(n+1)\,(2n+1)}{6}$.
- $1^3 + 2^3 + 3^3 + \cdots + n^3 = (1 + 2 + 3 + \cdots + n)^2$.

1.4.2. Strong Form of Mathematical Induction. Let P be a property of positive integers such that:

1. *Basis Step*: $P(1)$ is true, and
2. *Inductive Step*: if $P(k)$ is true for all $1 \leq k \leq n$ then $P(n+1)$ is true.

Then $P(n)$ is true for all positive integers.

Example: Prove that every integer $n \geq 2$ is prime or a product of primes. *Answer*:

1. *Basis Step*: 2 is a prime number, so the property holds for $n = 2$.
2. *Inductive Step*: Assume that if $2 \leq k \leq n$, then k is a prime number or a product of primes. Now, either $n + 1$ is a prime number or it is not. If it is a prime number then it verifies the property. If it is not a prime number, then it can be written as the product of two positive integers, $n + 1 = k_1\, k_2$, such that $1 < k_1, k_2 < n + 1$. By induction hypothesis each of k_1 and k_2 must be a prime or a product of primes, hence $n + 1$ is a product of primes.

This completes the proof.

1.4.3. The Well-Ordering Principle. Every nonempty set of positive integers has a smallest element.

Example: Prove that $\sqrt{2}$ is irrational (i.e., $\sqrt{2}$ cannot be written as a quotient of two positive integers) using the well-ordering principle. *Answer*: Assume that $\sqrt{2}$ is rational, i.e., $\sqrt{2} = a/b$, where a and b are integers. Note that since $\sqrt{2} > 1$ then $a > b$. Now we have $2 = a^2/b^2$, hence $2\,b^2 = a^2$. Since the left hand side is even, then a^2 is even, but this implies that a itself is even, so $a = 2\,a'$. Hence: $2\,b^2 = 4\,{a'}^2$, and simplifying: $b^2 = 2\,{a'}^2$. From here we see that $\sqrt{2} = b/a'$. Hence starting with a fractional representation of $\sqrt{2} = a/b$ we end up with another fractional representation $\sqrt{2} = b/a'$ with a smaller numerator $b < a$. Repeating the same argument with the fraction b/a' we get another fraction with an even smaller numerator, and so on. So the set of possible numerators of a fraction representing $\sqrt{2}$ cannot have a smallest element, contradicting the well-ordering principle. Consequently, our assumption that $\sqrt{2}$ is rational has to be false.

CHAPTER 2

The Language of Mathematics

2.1. Set Theory

2.1.1. Sets. A *set* is a collection of objects, called *elements* of the set. A set can be represented by listing its elements between braces: $A = \{1, 2, 3, 4, 5\}$. The symbol $\in$ is used to express that an element is (or belongs to) a set, for instance $3 \in A$. Its negation is represented by $\notin$, e.g. $7 \notin A$. If the set is finite, its number of elements is represented $|A|$, e.g. if $A = \{1, 2, 3, 4, 5\}$ then $|A| = 5$.

Some important sets are the following:

1. $\mathbb{N} = \{0, 1, 2, 3, \cdots\}$ = the set of natural numbers.[1]
2. $\mathbb{Z} = \{-3, -2, -1, 0, 1, 2, 3, \cdots\}$ = the set of integers.
3. $\mathbb{Q}$ = the set of rational numbers.
4. $\mathbb{R}$ = the set of real numbers.
5. $\mathbb{C}$ = the set of complex numbers.

Is S is one of those sets then we also use the following notations:[2]

1. S^+ = set of positive elements in S, for instance
$$Z^+ = \{1, 2, 3, \cdots\} = \text{ the set of positive integers.}$$
2. S^- = set of negative elements in S, for instance
$$\mathbb{Z}^- = \{-1, -2, -3, \cdots\} = \text{ the set of negative integers.}$$
3. S^* = set of elements in S excluding zero, for instance $\mathbb{R}^*$ = the set of non zero real numbers.

Set-builder notation. An alternative way to define a set, called *set-builder notation*, is by stating a property (predicate) $P(x)$ verified by exactly its elements, for instance $A = \{x \in \mathbb{Z} \mid 1 \leq x \leq 5\}$ = "set of

[1]Note that $\mathbb{N}$ includes zero—for some authors $\mathbb{N} = \{1, 2, 3, \cdots\}$, without zero.

[2]When working with *strings* we will use a similar notation with a different meaning—be careful not to confuse it.

integers x such that $1 \leq x \leq 5$"—i.e.: $A = \{1, 2, 3, 4, 5\}$. In general: $A = \{x \in \mathcal{U} \mid p(x)\}$, where $\mathcal{U}$ is the domain of discourse in which the predicate $P(x)$ must be interpreted, or $A = \{x \mid P(x)\}$ if the domain of discourse for $P(x)$ is implicitly understood. In set theory the term *universal set* is often used in place of "domain of discourse" for a given predicate.[3]

Principle of Extension. Two sets are *equal* if and only if they have the same elements, i.e.:

$$A = B \equiv \forall x \, (x \in A \leftrightarrow x \in B) \,.$$

Subset. We say that A is a *subset* of set B, or A is *contained* in B, and we represent it "$A \subseteq B$", if all elements of A are in B, e.g., if $A = \{a, b, c\}$ and $B = \{a, b, c, d, e\}$ then $A \subseteq B$.

A is a *proper subset* of B, represented "$A \subset B$", if $A \subseteq B$ but $A \neq B$, i.e., there is some element in B which is not in A.

Empty Set. A set with no elements is called *empty set* (or *null set*, or *void set*), and is represented by $\emptyset$ or $\{\}$.

Note that nothing prevents a set from possibly being an element of another set (which is not the same as being a subset!). For instance if $A = \{1, a, \{3, t\}, \{1, 2, 3\}\}$ and $B = \{3, t\}$, then obviously B is an element of A, i.e., $B \in A$.

Power Set. The collection of all subsets of a set A is called the *power set* of A, and is represented $\mathcal{P}(A)$. For instance, if $A = \{1, 2, 3\}$, then

$$\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, A\} \,.$$

Exercise: Prove by induction that if $|A| = n$ then $|\mathcal{P}(A)| = 2^n$.

Multisets. Two ordinary sets are identical if they have the same elements, so for instance, $\{a, a, b\}$ and $\{a, b\}$ are the same set because they have exactly the same elements, namely a and b. However, in some applications it might be useful to allow repeated elements in a set. In that case we use *multisets*, which are mathematical entities similar to sets, but with possibly repeated elements. So, as multisets, $\{a, a, b\}$ and $\{a, b\}$ would be considered different, since in the first one the element a occurs twice and in the second one it occurs only once.

[3]Properly speaking, the domain of discourse of set theory is the collection of all sets (which is not a set).

2.1.2. Venn Diagrams. Venn diagrams are graphic representations of sets as enclosed areas in the plane. For instance, in figure 2.1, the rectangle represents the universal set (the set of all elements considered in a given problem) and the shaded region represents a set A. The other figures represent various set operations.

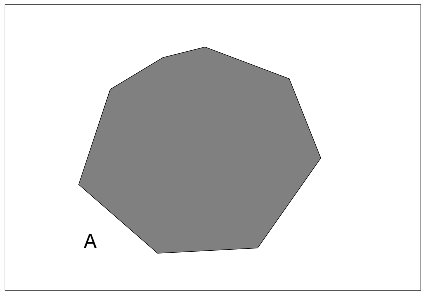


FIGURE 2.1. Venn Diagram.

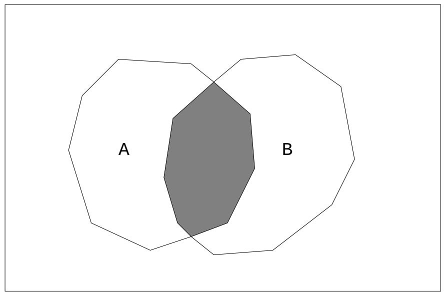


FIGURE 2.2. Intersection $A \cap B$.

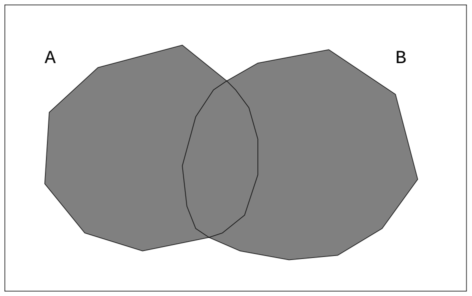


FIGURE 2.3. Union $A \cup B$.

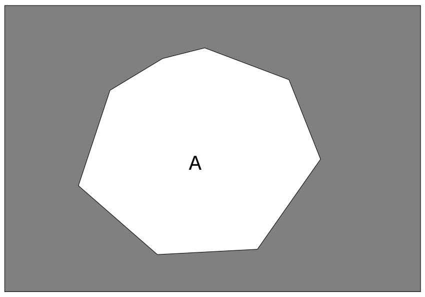


FIGURE 2.4. Complement $\overline{A}$.

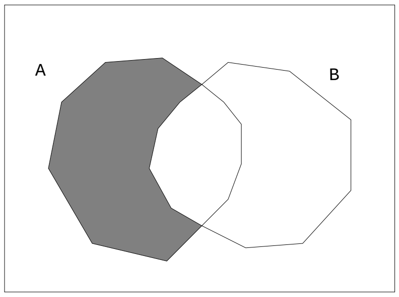


FIGURE 2.5. Difference $A - B$.

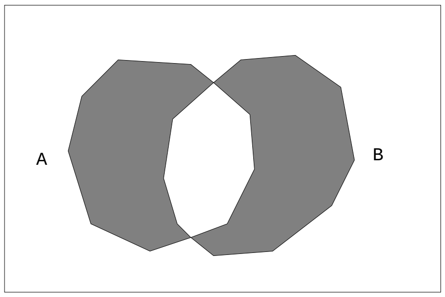


FIGURE 2.6. Symmetric Difference $A \triangle B$.

2.1.3. Set Operations.

1. *Intersection*: The common elements of two sets:
$$A \cap B = \{x \mid (x \in A) \wedge (x \in B)\}.$$
If $A \cap B = \emptyset$, the sets are said to be *disjoint*.

2. *Union*: The set of elements that belong to either of two sets:
$$A \cup B = \{x \mid (x \in A) \vee (x \in B)\}.$$

3. *Complement*: The set of elements (in the universal set) that do not belong to a given set:
$$\overline{A} = \{x \in \mathcal{U} \mid x \notin A\}\,.$$

4. *Difference* or *Relative Complement*: The set of elements that belong to a set but not to another:
$$A - B = \{x \mid (x \in A) \land (x \notin B)\} = A \cap \overline{B}\,.$$

5. *Symmetric Difference*: Given two sets, their symmetric difference is the set of elements that belong to either one or the other set but not both.
$$A \triangle B = \{x \mid (x \in A) \veebar (x \in B)\}\,.$$
It can be expressed also in the following way:
$$A \triangle B = A \cup B - A \cap B = (A - B) \cup (B - A)\,.$$

2.1.4. Counting with Venn Diagrams. A Venn diagram with n sets intersecting in the most general way divides the plane into 2^n regions. If we have information about the number of elements of some portions of the diagram, then we can find the number of elements in each of the regions and use that information for obtaining the number of elements in other portions of the plane.

Example: Let M, P and C be the sets of students taking Mathematics courses, Physics courses and Computer Science courses respectively in a university. Assume $|M| = 300$, $|P| = 350$, $|C| = 450$, $|M \cap P| = 100$, $|M \cap C| = 150$, $|P \cap C| = 75$, $|M \cap P \cap C| = 10$. How many students are taking exactly one of those courses? (fig. 2.7)

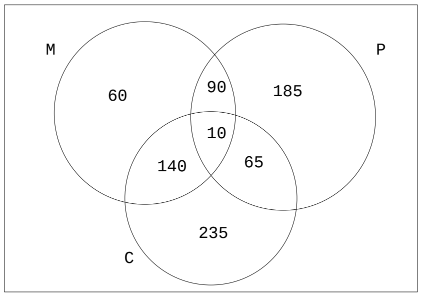


FIGURE 2.7. Counting with Venn diagrams.

We see that $|(M\cap P)-(M\cap P\cap C)| = 100-10 = 90$, $|(M\cap C)-(M\cap P\cap C)| = 150-10 = 140$ and $|(P\cap C)-(M\cap P\cap C)| = 75-10 = 65$.

Then the region corresponding to students taking Mathematics courses only has cardinality $300-(90+10+140) = 60$. Analogously we compute the number of students taking Physics courses only (185) and taking Computer Science courses only (235). The sum $60 + 185 + 235 = 480$ is the number of students taking exactly one of those courses.

2.1.5. Properties of Sets.

The set operations verify the following properties:

1. *Associative Laws:*
$$A \cup (B \cup C) = (A \cup B) \cup C$$
$$A \cap (B \cap C) = (A \cap B) \cap C$$
2. *Commutative Laws:*
$$A \cup B = B \cup A$$
$$A \cap B = B \cap A$$
3. *Distributive Laws:*
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
4. *Identity Laws:*
$$A \cup \emptyset = A$$
$$A \cap \mathfrak{U} = A$$
5. *Complement Laws:*
$$A \cup \overline{A} = \mathfrak{U}$$
$$A \cap \overline{A} = \emptyset$$
6. *Idempotent Laws:*
$$A \cup A = A$$
$$A \cap A = A$$
7. *Bound Laws:*
$$A \cup \mathfrak{U} = \mathfrak{U}$$
$$A \cap \emptyset = \emptyset$$
8. *Absorption Laws:*
$$A \cup (A \cap B) = A$$
$$A \cap (A \cup B) = A$$
9. *Involution Law:*
$$\overline{\overline{A}} = A$$

10. 0/1 *Laws:*

$$\overline{\emptyset} = \mathcal{U}$$
$$\overline{\mathcal{U}} = \emptyset$$

11. *DeMorgan's Laws:*

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

2.1.6. Generalized Union and Intersection. Given a collection of sets $A_1, A_2, \dots, A_N$, their union is defined as the set of elements that belong to at least one of the sets (here n represents an integer in the range from 1 to N):

$$\bigcup_{n=1}^{N} A_n = A_1 \cup A_2 \cup \cdots \cup A_N = \{x \mid \exists n\,(x \in A_n)\}\,.$$

Analogously, their intersection is the set of elements that belong to all the sets simultaneously:

$$\bigcap_{n=1}^{N} A_n = A_1 \cap A_2 \cap \cdots \cap A_N = \{x \mid \forall n\,(x \in A_n)\}\,.$$

These definitions can be applied to infinite collections of sets as well. For instance assume that $S_n = \{kn \mid k = 2, 3, 4, \dots\}$ = set of multiples of n greater than n. Then

$$\bigcup_{n=2}^{\infty} S_n = S_2 \cup S_3 \cup S_4 \cup \cdots = \{4, 6, 8, 9, 10, 12, 14, 15, \dots\}$$
$$= \text{set of composite positive integers}\,.$$

2.1.7. Partitions. A *partition* of a set X is a collection $\mathcal{S}$ of non overlapping non empty subsets of X whose union is the whole X. For instance a partition of $X\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ could be

$$\mathcal{S} = \{\{1, 2, 4, 8\}, \{3, 6\}, \{5, 7, 9, 10\}\}\,.$$

Given a partition $\mathcal{S}$ of a set X, every element of X belongs to exactly one member of $\mathcal{S}$.

Example: The division of the integers $\mathbb{Z}$ into even and odd numbers is a partition: $\mathcal{S} = \{\mathbb{E}, \mathbb{O}\}$, where $\mathbb{E} = \{2n \mid n \in \mathbb{Z}\}$, $\mathbb{O} = \{2n + 1 \mid n \in \mathbb{Z}\}$.

Example: The divisions of $\mathbb{Z}$ in negative integers, positive integers and zero is a partition: $\mathcal{S} = \{\mathbb{Z}^+, Z^-, \{0\}\}$.

2.1.8. Ordered Pairs, Cartesian Product. An ordinary pair $\{a, b\}$ is a set with two elements. In a set the order of the elements is irrelevant, so $\{a, b\} = \{b, a\}$. If the order of the elements is relevant, then we use a different object called *ordered pair*, represented (a, b). Now $(a, b) \neq (b, a)$ (unless $a = b$). In general $(a, b) = (a', b')$ iff $a = a'$ and $b = b'$.

Given two sets A, B, their *Cartesian product* $A \times B$ is the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$:

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

Analogously we can define triples or 3-tuples (a, b, c), 4-tuples (a, b, c, d), $\dots$, n-tuples $(a_1, a_2, \dots, a_n)$, and the corresponding 3-fold, 4-fold,$\dots$, n-fold Cartesian products:

$$A_1 \times A_2 \times \cdots \times A_n = \\ \{(a_1, a_2, \dots, a_n) \mid (a_1 \in A_1) \wedge (a_2 \in A_2) \wedge \cdots \wedge (a_n \in A_n)\}\,.$$

If all the sets in a Cartesian product are the same, then we can use an exponent: $A^2 = A \times A$, $A^3 = A \times A \times A$, etc. In general:

$$A^n = A \times A \times \overset{(n \text{ times})}{\cdots} \times A\,.$$

An example of Cartesian product is the *real plane* $\mathbb{R}^2$, where $\mathbb{R}$ is the set of real numbers ($\mathbb{R}$ is sometimes called *real line*).

2.2. Sequences and Strings

2.2.1. Sequences. A *sequence* is an (usually infinite) ordered list of elements. Examples:

1. The sequence of positive integers:
$$1, 2, 3, 4, \dots, n, \dots$$
2. The sequence of positive even integers:
$$2, 4, 6, 8, \dots, 2n, \dots$$
3. The sequence of powers of 2:
$$1, 2, 4, 8, 16, \dots, n^2, \dots$$
4. The sequence of Fibonacci numbers (each one is the sum of the two previous ones):
$$0, 1, 1, 2, 3, 5, 8, 13, \dots$$
5. The reciprocals of the positive integers:
$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \cdots, \frac{1}{n}, \cdots$$

In general the elements of a sequence are represented with an indexed letter, say $s_1, s_2, s_3, \dots, s_n, \dots$. The sequence itself can be defined by giving a rule, e.g.: $s_n = 2n + 1$ is the sequence:
$$3, 5, 7, 9, \dots$$
Here we are assuming that the first element is s_1, but we can start at any value of the index that we want, for instance if we declare s_0 to be the first term, the previous sequence would become:
$$1, 3, 5, 7, 9, \dots$$
The sequence is symbolically represented $\{s_n\}$ or $\{s_n\}_{n=1}^{\infty}$.

If $s_n \leq s_{n+1}$ for every n the sequence is called *increasing*. If $s_n \geq s_{n+1}$ then it is called *decreasing*. For instance $s_n = 2n+1$ is increasing: $3, 5, 7, 9, \dots$, while $s_n = 1/n$ is decreasing: $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \cdots$.

If we remove elements from a sequence we obtain a *subsequence*. E.g., if we remove all odd numbers from the sequence of positive integers:
$$1, 2, 3, 4, 5 \dots,$$

we get the subsequence consisting of the even positive integers:

$$2, 4, 6, 8, \dots$$

2.2.2. Sum (Sigma) and Product Notation. In order to abbreviate sums and products the following notations are used:

1. *Sum* (or *sigma*) notation:

$$\sum_{i=m}^{n} a_i = a_m + a_{m+1} + a_{m+2} + \cdots + a_n$$

2. *Product* notation:

$$\prod_{i=m}^{n} a_i = a_m \cdot a_{m+1} \cdot a_{m+2} \cdot \cdots \cdot a_n$$

For instance: assume $a_n = 2n + 1$, then

$$\sum_{n=3}^{6} a_n = a_3 + a_4 + a_5 + a_6 = 7 + 9 + 11 + 13 = 40\,.$$

$$\prod_{n=3}^{6} a_n = a_3 \cdot a_4 \cdot a_5 \cdot a_6 = 7 \cdot 9 \cdot 11 \cdot 13 = 9009\,.$$

2.2.3. Strings. Given a set X, a *string over* X is a *finite* ordered list of elements of X.

Example: If X is the set $X = \{a, b, c\}$, then the following are examples of strings over X: aba, $aaaa$, bba, etc.

Repetitions can be specified with a superscripts, for instance: $a^2b^3ac^2a^3 = aabbbaccaaa$, $(ab)^3 = ababab$, etc.

The *length* of a string is its number of elements, e.g., $|abaccbab| = 8$, $|a^2b^7a^3c^6| = 18$.

The string with no elements is called *null string*, represented λ. Its length is, of course, zero: $|\lambda| = 0$.

The set of all strings over X is represented X^*. The set of no null strings over X (i.e., all strings over X except the null string) is represented X^+.

Given two strings α and β over X, the string consisting of α followed by β is called the *concatenation* of α and β. For instance if $\alpha = abac$ and $\beta = baaab$ then $\alpha\beta = abacbaaab$.

2.3. Relations

2.3.1. Relations. Assume that we have a set of men M and a set of women W, some of whom are married. We want to express which men in M are married to which women in W. One way to do that is by listing the set of pairs (m, w) such that m is a man, w is a woman, and m is married to w. So, the relation "married to" can be represented by a subset of the Cartesian product $M \times W$. In general, a *relation* $\mathcal{R}$ from a set A to a set B will be understood as a subset of the Cartesian product $A \times B$, i.e., $\mathcal{R} \subseteq A \times B$. If an element $a \in A$ is related to an element $b \in B$, we often write $a \,\mathcal{R}\, b$ instead of $(a, b) \in \mathcal{R}$.

The set

$$\{a \in A \mid a \,\mathcal{R}\, b \text{ for some } b \in B\}$$

is called the *domain* of $\mathcal{R}$. The set

$$\{b \in B \mid a \,\mathcal{R}\, b \text{ for some } a \in A\}$$

is called the *range* of $\mathcal{R}$. For instance, in the relation "married to" above, the domain is the set of married men, and the range is the set of married women.

If A and B are the same set, then any subset of $A \times A$ will be a *binary relation* in A. For instance, assume $A = \{1, 2, 3, 4\}$. Then the binary relation "less than" in A will be:

$$\begin{aligned} <_A = \{(x, y) \in A \times A \mid x < y\} \\ = \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}\,. \end{aligned}$$

Notation: A set A with a binary relation $\mathcal{R}$ is sometimes represented by the pair $(A, \mathcal{R})$. So, for instance, $(\mathbb{Z}, \leq)$ means the set of integers together with the relation of non-strict inequality.

2.3.2. Representations of Relations.

Arrow diagrams. Venn diagrams and arrows can be used for representing relations between given sets. As an example, figure 2.8 represents the relation from $A = \{a, b, c, d\}$ to $B = \{1, 2, 3, 4\}$ given by $\mathcal{R} = \{(a, 1), (b, 1), (c, 2), (c, 3)\}$. In the diagram an arrow from x to y means that x is related to y. This kind of graph is called *directed graph* or *digraph.*

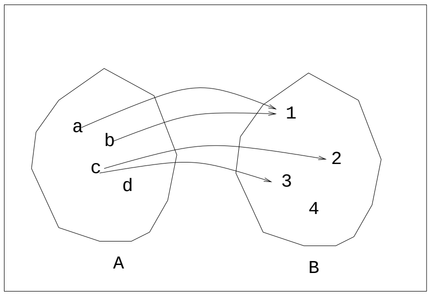


FIGURE 2.8. Relation.

Another example is given in diagram 2.9, which represents the divisibility relation on the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

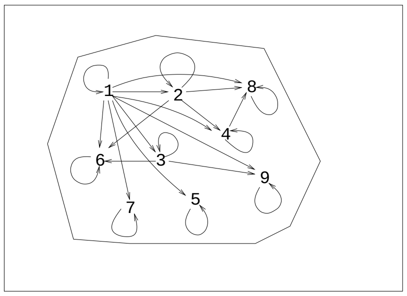


FIGURE 2.9. Binary relation of divisibility.

Matrix of a Relation. Another way of representing a relation $\mathcal{R}$ from A to B is with a matrix. Its rows are labeled with the elements of A, and its columns are labeled with the elements of B. If $a \in A$ and $b \in B$ then we write 1 in row a column b if $a \, \mathcal{R} \, b$, otherwise we write 0. For instance the relation $\mathcal{R} = \{(a, 1), (b, 1), (c, 2), (c, 3)\}$ from $A = \{a, b, c, d\}$ to $B = \{1, 2, 3, 4\}$ has the following matrix:

$$\begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

2.3.3. Inverse Relation. Given a relation $\mathcal{R}$ from A to B, the inverse of $\mathcal{R}$, denoted $\mathcal{R}^{-1}$, is the relation from B to A defined as

$$b \, \mathcal{R}^{-1} \, a \Leftrightarrow a \, \mathcal{R} \, b \, .$$

For instance, if $\mathcal{R}$ is the relation "being a son or daughter of", then $\mathcal{R}^{-1}$ is the relation "being a parent of".

2.3.4. Composition of Relations. Let A, B and C be three sets. Given a relation $\mathcal{R}$ from A to B and a relation $\mathcal{S}$ from B to C, then the composition $\mathcal{S} \circ \mathcal{R}$ of relations $\mathcal{R}$ and $\mathcal{S}$ is a relation from A to C defined by:

$$a\,(\mathcal{S} \circ \mathcal{R})\,c \Leftrightarrow \text{there exists some } b \in B \text{ such that } a\,\mathcal{R}\,b \text{ and } b\,\mathcal{S}\,c\,.$$

For instance, if $\mathcal{R}$ is the relation "to be the father of", and $\mathcal{S}$ is the relation "to be married to", then $\mathcal{S} \circ \mathcal{R}$ is the relation "to be the father in law of".

2.3.5. Properties of Binary Relations. A binary relation $\mathcal{R}$ on A is called:

1. *Reflexive* if for all $x \in A$, $x\,\mathcal{R}\,x$. For instance on $\mathbb{Z}$ the relation "equal to" $(=)$ is reflexive.
2. *Transitive* if for all $x, y, z \in A$, $x\,\mathcal{R}\,y$ and $y\,\mathcal{R}\,z$ implies $x\,\mathcal{R}\,z$. For instance equality $(=)$ and inequality $(<)$ on $\mathbb{Z}$ are transitive relations.
3. *Symmetric* if for all $x, y \in A$, $x\,\mathcal{R}\,y \Rightarrow y\,\mathcal{R}\,x$. For instance on $\mathbb{Z}$, equality $(=)$ is symmetric, but strict inequality $(<)$ is not.
4. *Antisymmetric* if for all $x, y \in A$, $x\,\mathcal{R}\,y$ and $y\,\mathcal{R}\,x$ implies $x = y$. For instance, non-strict inequality $(\leq)$ on $\mathbb{Z}$ is antisymmetric.

2.3.6. Partial Orders. A *partial order*, or simply, an *order* on a set A is a binary relation "$\preccurlyeq$" on A with the following properties:

1. *Reflexive*: for all $x \in A$, $x \preccurlyeq x$.
2. *Antisymmetric*: $(x \preccurlyeq y) \wedge (y \preccurlyeq x) \Rightarrow x = y$.
3. *Transitive*: $(x \preccurlyeq y) \wedge (y \preccurlyeq z) \Rightarrow x \preccurlyeq z$.

Examples:

1. The non-strict inequality $(\leq)$ in $\mathbb{Z}$.
2. Relation of divisibility on $\mathbb{Z}^{+}$: $a|b \Leftrightarrow \exists t,\ b = at$.

3. Set inclusion ($\subseteq$) on $\mathcal{P}(A)$ (the collection of subsets of a given set A).

Exercise: prove that the aforementioned relations are in fact partial orders. As an example we prove that integer divisibility is a partial order:

1. Reflexive: $a = a\,1 \Rightarrow a|a$.
2. Antisymmetric: $a|b \Rightarrow b = at$ for some t and $b|a \Rightarrow a = bt'$ for some t'. Hence $a = att'$, which implies $tt' = 1 \Rightarrow t' = t^{-1}$. The only invertible positive integer is 1, so $t = t' = 1 \Rightarrow a = b$.
3. Transitive: $a|b$ and $b|c$ implies $b = at$ for some t and $c = bt'$ for some t', hence $c = att'$, i.e., $a|c$.

Question: is the strict inequality ($<$) a partial order on $\mathbb{Z}$?

Two elements $a, b \in A$ are said to be *comparable* if either $x \preccurlyeq y$ or $y \preccurlyeq x$, otherwise they are said to be *non comparable.* The order is called *total* or *linear* when every pair of elements $x, y \in A$ are comparable. For instance $(\mathbb{Z}, \leq)$ is totally ordered, but $(\mathbb{Z}^+, |)$, where "$|$" represents integer divisibility, is not. A totally ordered subset of a partially ordered set is called a *chain*; for instance the set $\{1, 2, 4, 8, 16, \dots\}$ is a chain in $(\mathbb{Z}^+, |)$.

2.3.7. Hasse diagrams. A Hasse diagram is a graphical representation of a partially ordered set in which each element is represented by a dot (node or vertex of the diagram). Its immediate successors are placed above the node and connected to it by straight line segments. As an example, figure 2.10 represents the Hasse diagram for the relation of divisibility on $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Question: How does the Hasse diagram look for a totally ordered set?

2.3.8. Equivalence Relations. An *equivalence relation* on a set A is a binary relation "$\sim$" on A with the following properties:

1. *Reflexive*: for all $x \in A$, $x \sim x$.
2. *Symmetric*: $x \sim y \Rightarrow y \sim x$.
3. *Transitive*: $(x \sim y) \wedge (y \sim z) \Rightarrow x \sim z$.

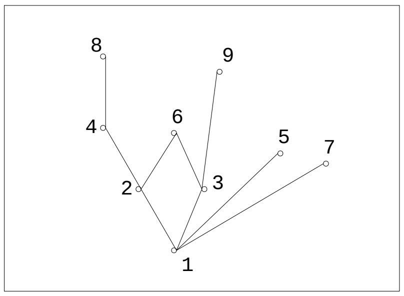


FIGURE 2.10. Hasse diagram for divisibility.

For instance, on $\mathbb{Z}$, the equality $(=)$ is an equivalence relation.

Another example, also on $\mathbb{Z}$, is the following: $x \equiv y \pmod 2$ ("x is congruent to y modulo 2") iff $x-y$ is even. For instance, $6 \equiv 2 \pmod 2$ because $6-2=4$ is even, but $7 \not\equiv 4 \pmod 2$, because $7-4=3$ is not even. Congruence modulo 2 is in fact an equivalence relation:

1. Reflexive: for every integer x, $x-x=0$ is indeed even, so $x \equiv x \pmod 2$.
2. Symmetric: if $x \equiv y \pmod 2$ then $x-y=t$ is even, but $y-x=-t$ is also even, hence $y \equiv x \pmod 2$.
3. Transitive: assume $x \equiv y \pmod 2$ and $y \equiv z \pmod 2$. Then $x-y=t$ and $y-z=u$ are even. From here, $x-z=(x-y)+(y-z)=t+u$ is also even, hence $x \equiv z \pmod 2$.

2.3.9. Equivalence Classes, Quotient Set, Partitions. Given an equivalence relation $\sim$ on a set A, and an element $x \in A$, the set of elements of A related to x are called the *equivalence class* of x, represented $[x] = \{y \in A \mid y \sim x\}$. Element x is said to be a *representative* of class

$[x]$. The collection of equivalence classes, represented $A/\!\sim\; = \{[x] \mid x \in A\}$, is called *quotient set* of A by $\sim$.

Exercise: Find the equivalence classes on $\mathbb{Z}$ with the relation of congruence modulo 2.

One of the main properties of an equivalence relation on a set A is that the quotient set, i.e. the collection of equivalence classes, is a partition of A. Recall that a *partition* of a set A is a collection of non-empty subsets $A_1, A_2, A_3, \ldots$ of A which are pairwise disjoint and whose union equals A:

1. $A_i \cap A_j = \emptyset$ for $i \neq j$,
2. $\bigcup_n A_n = A$.

Example: in $\mathbb{Z}$ with the relation of congruence modulo 2 (call it "$\sim_2$"), there are two equivalence classes: the set $\mathbb{E}$ of even integers and the set $\mathbb{O}$ of odd integers. The quotient set of $\mathbb{Z}$ by the relation "$\sim_2$" of congruence modulo 2 is $\mathbb{Z}/\sim_2 \; = \{\mathbb{E}, \mathbb{O}\}$. We see that it is in fact a partition of $\mathbb{Z}$, because $\mathbb{E} \cap \mathbb{O} = \emptyset$, and $\mathbb{Z} = \mathbb{E} \cup \mathbb{O}$.

Exercise: Let m be an integer greater than or equal to 2. On $\mathbb{Z}$ we define the relation $x \equiv y \pmod{m} \Leftrightarrow m|(y-x)$ (i.e., m divides exactly $y - x$). Prove that it is an equivalence relation. What are the equivalence classes? How many are there?

Exercise: On the Cartesian product $\mathbb{Z} \times \mathbb{Z}^*$ we define the relation $(a,b)\,\mathcal{R}\,(c,d) \Leftrightarrow ad = bc$. Prove that $\mathcal{R}$ is an equivalence relation. Would it still be an equivalence relation if we extend it to $\mathbb{Z} \times \mathbb{Z}$?

2.4. Functions

2.4.1. Correspondences. Suppose that to each element of a set A we assign some elements of another set B. For instance, $A = \mathbb{N}$, $B = \mathbb{Z}$, and to each element $x \in \mathbb{N}$ we assign all elements $y \in \mathbb{Z}$ such that $y^2 = x$ (fig. 2.11).

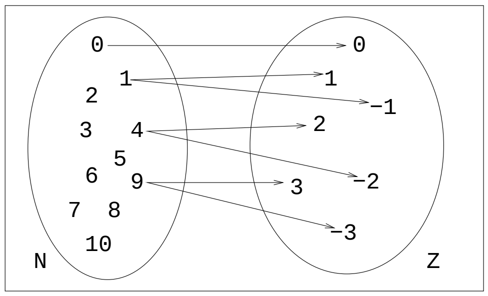


FIGURE 2.11. Correspondence $x \mapsto \pm\sqrt{x}$.

This operation can be interpreted as a relation, but when we want to stress the fact that it is an assignment of some elements to other elements, we call it a *correspondence*.

2.4.2. Functions. A *function* or *mapping* f from a set A to a set B, denoted $f : A \to B$, is a correspondence in which to each element x of A corresponds exactly one element $y = f(x)$ of B (fig. 2.12).

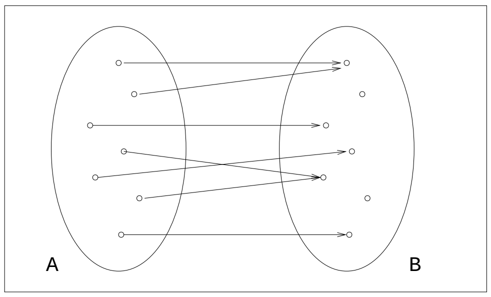


FIGURE 2.12. Function.

Sometimes we represent the function with a diagram like this:

$$f : A \to B \qquad \text{or} \qquad A \xrightarrow{f} B$$
$$x \mapsto y \qquad\qquad\qquad x \mapsto y$$

For instance, the following represents the function from $\mathbb{Z}$ to $\mathbb{Z}$ defined by $f(x) = 2x + 1$:

$$\begin{aligned} f : \mathbb{Z} &\to \mathbb{Z} \\ x &\mapsto 2x + 1 \end{aligned}$$

The element $y = f(x)$ is called the *image* of x, and x is a *preimage* of y. For instance, if $f(x) = 2x + 1$ then $f(7) = 2 \cdot 7 + 1 = 15$. The set A is the *domain* of f, and B is its *codomain*. If $A' \subseteq A$, the image of A' by f is $f(A') = \{f(x) \mid x \in A'\}$, i.e., the subset of B consisting of all images of elements of A'. The subset $f(A)$ of B consisting of all images of elements of A is called the *range* of f. For instance, the range of $f(x) = 2x + 1$ is the set of all integers of the form $2x + 1$ for some integer x, i.e., all odd numbers.

Example: Two useful functions from $\mathbb{R}$ to $\mathbb{Z}$ are the following:

1. The *floor* function:

   $$\lfloor x \rfloor = \text{greatest integer less than or equal to } x\,.$$

   For instance: $\lfloor 2 \rfloor = 2$, $\lfloor 2.3 \rfloor = 2$, $\lfloor \pi \rfloor = 3$, $\lfloor -2.5 \rfloor = -3$.

2. The *ceiling* function:

   $$\lceil x \rceil = \text{least integer greater than or equal to } x\,.$$

   For instance: $\lceil 2 \rceil = 2$, $\lceil 2.3 \rceil = 3$, $\lceil \pi \rceil = 4$, $\lceil -2.5 \rceil = -2$.

Example: The *modulus operator* is the function $\bmod : \mathbb{Z} \times \mathbb{Z}^+ \to \mathbb{Z}$ defined:

$$x \bmod y = \text{ remainder when } x \text{ is divided by } y.$$

For instance $23 \bmod 7 = 2$ because $23 = 3 \cdot 7 + 2$, $59 \bmod 9 = 5$ because $59 = 6 \cdot 9 + 5$, etc.

Graph: The *graph* of a function $f : A \to B$ is the subset of $A \times B$ defined by $G(f) = \{(x, f(x)) \mid x \in A\}$ (fig. 2.13).

2.4.3. Types of Functions.

1. *One-to-One* or *Injective*: A function $f : A \to B$ is called *one-to-one* or *injective* if each element of B is the image of at most one element of A (fig. 2.14):

   $$\forall x, x' \in A,\ f(x) = f(x') \Rightarrow x = x'\,.$$

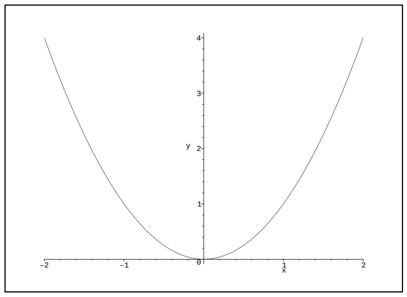


FIGURE 2.13. Graph of $f(x) = x^2$.

For instance, $f(x) = 2x$ from $\mathbb{Z}$ to $\mathbb{Z}$ is injective.

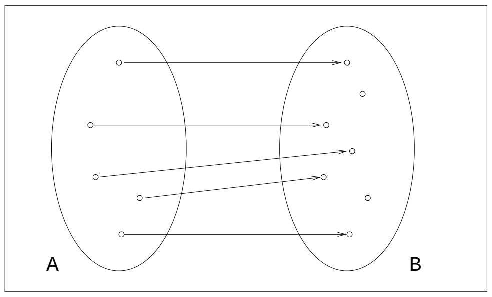


FIGURE 2.14. One-to-one function.

2. *Onto* or *Surjective*: A function $f : A \to B$ is called *onto* or *surjective* if every element of B is the image of some element of A (fig. 2.15):

$$\forall y \in B,\ \exists x \in A \text{ such that } y = f(x)\,.$$

For instance, $f(x) = x^2$ from $\mathbb{R}$ to $\mathbb{R}^+ \cup \{0\}$ is onto.

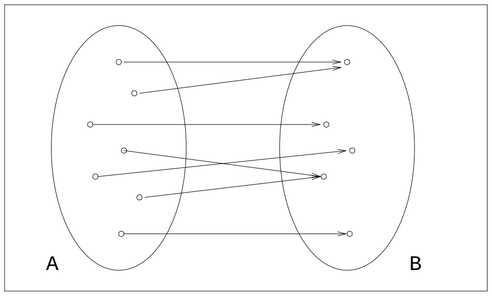


FIGURE 2.15. Onto function.

3. *Bijective Function* or *Bijection*: A function $f : A \to B$ is said to be *bijective* or a *bijection* if it is one-to-one and onto (fig. 2.16). For instance, $f(x) = x + 3$ from $\mathbb{Z}$ to $\mathbb{Z}$ is a bijection.

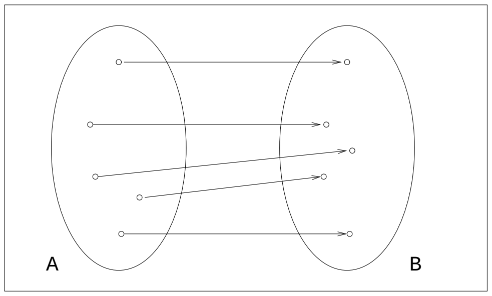


FIGURE 2.16. Bijection.

2.4.4. Identity Function. Given a set A, the function $1_A : A \to A$ defined by $1_A(x) = x$ for every x in A is called the *identity function* for A.

2.4.5. Function Composition. Given two functions $f : A \to B$ and $g : B \to C$, the *composite function* of f and g is the function $g \circ f : A \to C$ defined by $(g \circ f)(x) = g(f(x))$ for every x in A:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, bend left, "g\circ f"] & B \arrow[r, "g"] & C \\ x \arrow[r, mapsto] & y=f(x) \arrow[r, mapsto] & z=g(y)=g(f(x)) \end{tikzcd}$$

For instance, if $A = B = C = \mathbb{Z}$, $f(x) = x + 1$, $g(x) = x^2$, then $(g \circ f)(x) = f(x)^2 = (x+1)^2$. Also $(f \circ g)(x) = g(x) + 1 = x^2 + 1$ (the composition of functions is not commutative in general).

Some properties of function composition are the following:

1. If $f : A \to B$ is a function from A to B, we have that $f \circ 1_A = 1_B \circ f = f$.

2. Function composition is associative, i.e., given three functions

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D \,,$$

we have that $h \circ (g \circ f) = (h \circ g) \circ f$.

Function iteration. If $f : A \to A$ is a function from A to A, then it makes sense to compose it with itself: $f^2 = f \circ f$. For instance, if $f : \mathbb{Z} \to \mathbb{Z}$ is $f(x) = 2x + 1$, then $f^2(x) = 2(2x + 1) + 1 = 4x + 3$. Analogously we can define $f^3 = f \circ f \circ f$, and so on, $f^n = f \circ \overset{(n \text{ times})}{\cdots} \circ f$.

2.4.6. Inverse Function. If $f : A \to B$ is a bijective function, its inverse is the function $f^{-1} : B \to A$ such that $f^{-1}(y) = x$ if and only if $f(x) = y$.

For instance, if $f : \mathbb{Z} \to \mathbb{Z}$ is defined by $f(x) = x + 3$, then its inverse is $f^{-1}(x) = x - 3$.

The arrow diagram of f^{-1} is the same as the arrow diagram of f but with all arrows reversed.

A characteristic property of the inverse function is that $f^{-1} \circ f = 1_A$ and $f \circ f^{-1} = 1_B$.

2.4.7. Operators. A function from $A \times A$ to A is called a *binary operator* on A. For instance the addition of integers is a binary operator $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. In the usual notation for functions the sum of two integers x and y would be represented $+(x, y)$. This is called *prefix* notation. The *infix* notation consists of writing the symbol of the binary operator between its arguments: $x + y$ (this is the most common). There is also a *postfix* notation consisting of writing the symbol after the arguments: $x\, y\, +$.

Another example of binary operator on $\mathbb{Z}$ is $(x, y) \mapsto x \cdot y$.

A *monary* or *unary operator* on A is a function from A to A. For instance the change of sign $x \mapsto -x$ on $\mathbb{Z}$ is a unary operator on $\mathbb{Z}$. An example of unary operator on $\mathbb{R}^*$ (non-zero real numbers) is $x \mapsto 1/x$.

CHAPTER 3

Algorithms

3.1. Algorithms

Consider the following list of instructions to find the maximum of three numbers a, b, c:

1. Assign variable x the value of a.
2. If $b > x$ then assign x the value of b.
3. If $c > x$ then assign x the value of c.
4. Output the value of x.

After executing those steps the output will be the maximum of a, b, c.

In general an algorithm is a finite list of instructions with the following characteristics:

1. *Precision.* The steps are precisely stated.
2. *Uniqueness.* The result of executing each step is uniquely determined by the inputs and the result of preceding steps.
3. *Finiteness.* The algorithm stops after finitely many instructions have been executed.
4. *Input.* The algorithm receives input.
5. *Output.* The algorithm produces output.
6. *Generality.* The algorithm applies to a set of inputs.

Basically an algorithm is the idea behind a program. Conversely, programs are implementations of algorithms.

3.1.1. Pseudocode.

Pseudocode is a language similar to a programming language used to represent algorithms. The main difference respect to actual programming languages is that pseudocode is not required to follow strict syntactic rules, since it is intended to be just read by humans, not actually executed by a machine.

Usually pseudocode will look like this:

```
procedure ProcedureName(Input)
  Instructions...
end ProcedureName
```

For instance the following is an algorithm to find the maximum of three numbers a, b, c:

```
procedure max(a,b,c)
  x := a
  if b>x then
    x := b
  if c>x then
    x := c
  return(x)
end max
```

Next we show a few common operations in pseudocode.

The following statement means "assign variable x the value of variable y:

```
x := y
```

The following code executes "action" if condition "p" is true:

```
if p then
  action
```

The following code executes "action1" if condition "p" is true, otherwise it executes "action2":

```
if p then
  action1
else
  action2
```

The following code executes "action" while condition "p" is true:

```
while p do
  action
```

The following is the structure of a for loop:

```
for var := init to limit do
  action
```

If an action contains more than one statement then we must enclose them in a block:

```
begin
  Instruction1
  Instruction2
  Instruction3
  ...
end
```

Comments begin with two slashes:

```
// This is a comment
```

The output of a procedure is returned with a return statement:

```
return(output)
```

Procedures that do not return anything are invoked with a call statement:

```
call Procedure(arguments...)
```

As an example, the following procedure returns the largest number in a sequence $s_1, s_2, \dots s_n$ represented as an array with n elements: s[1], s[2],…, s[n]:

```
procedure largest_element(s,n)
  largest := s[1]
  for k := 2 to n do
    if s[k] > largest then
      largest := s[k]
  return(largest)
end largest_element
```

3.1.2. Recursiveness.

Recursive Definitions. A definition such that the object defined occurs in the definition is called a *recursive definition.* For instance, consider the *Fibonacci sequence*

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

It can be defined as a sequence whose two first terms are $F_0 = 0$, $F_1 = 1$ and each subsequent term is the sum of the two previous ones: $F_n = F_{n-1} + F_{n-2}$ (for $n \geq 2$).

Other examples:

- Factorial:
  1. $0! = 1$
  2. $n! = n \cdot (n-1)! \qquad (n \geq 1)$
- Power:
  1. $a^0 = 1$
  2. $a^n = a^{n-1}\, a \qquad (n \geq 1)$

In all these examples we have:

1. A *Basis*, where the function is explicitly evaluated for one or more values of its argument.
2. A *Recursive Step*, stating how to compute the function from its previous values.

Recursive Procedures. A *recursive procedure* is a procedure that invokes itself. Also a set of procedures is called *recursive* if they invoke themselves in a circle, e.g., procedure p_1 invokes procedure p_2, procedure p_2 invokes procedure p_3 and procedure p_3 invokes procedure p_1. A *recursive algorithm* is an algorithm that contains recursive procedures or recursive sets of procedures. Recursive algorithms have the advantage that often they are easy to design and are closer to natural mathematical definitions.

As an example we show two alternative algorithms for computing the factorial of a natural number, the first one iterative (non recursive), the second one recursive.

```
procedure factorial_iterative(n)
  fact := 1
  for k := 2 to n do
    fact := k * fact
  return(fact)
end factorial_iterative
```

```
procedure factorial_recursive(n)
  if n = 0 then
    return(1)
  else
    return(n * factorial_recursive(n-1))
end factorial_recursive
```

While the iterative version computes $n! = 1 \cdot 2 \cdot \ldots n$ directly, the recursive version resembles more closely the formula $n! = n \cdot (n-1)!$

A recursive algorithm must contain at least a basic case without recursive call (the case $n = 0$ in our example), and any legitimate input should lead to a *finite* sequence of recursive calls ending up at the basic case. In our example n is a legitimate input if it is a natural number, i.e., an integer greater than or equal to 0. If $n = 0$ then `factorial_recursive(0)` returns 1 immediately without performing any recursive call. If $n >$ then the execution of

```
factorial_recursive(n)
```

leads to a recursive call

```
factorial_recursive(n-1)
```

which will perform a recursive call

```
factorial_recursive(n-2)
```

and so on until eventually reaching the basic case

```
factorial_recursive(0)
```

After reaching the basic case the procedure returns a value to the last call, which returns a value to the previous call, and so on up to the first invocation of the procedure.

Another example is the following algorithm for computing the nth element of the Fibonacci sequence:

```
procedure fibonacci(n)
  if n=0 then
    return(0)
  if n=1 then
    return(1)
  return(fibonacci(n-1) + fibonacci(n-2))
end fibonacci
```

In this example we have two basic cases, namely $n = 0$ and $n = 1$.

In this particular case the algorithm is inefficient in the sense that it performs more computations than actually needed. For instance a call to `fibonacci(5)` contains two recursive calls, one to `fibonacci(4)` and another one to `fibonacci(3)`. Then `fibonacci(4)` performs a call to `fibonacci(3)` and another call to `fibonacci(2)`, so at this point we see that `fibonacci(3)` is being called twice, once inside `fibonacci(5)` and again in `fibonacci(4)`. Hence sometimes the price to pay for a simpler algorithmic structure is a loss of efficiency.

However careful design may yield efficient recursive algorithms. An example is *merge_sort*, and algorithm intended to sort a list of elements. First let's look at a simple non recursive sorting algorithm called *bubble_sort*. The idea is to go several times through the list swapping adjacent elements if necessary. It applies to a list of numbers $s_i, s_{i+1}, \dots, s_j$ represented as an array `s[i], s[i+1],..., s[j]`:

```
procedure bubble_sort(s,i,j)
  for p:=1 to j-i do
    for q:=i to j-p do
      if s[q] > s[q+1] then
        swap(s[q],s[q+1])
end bubble_sort
```

We can see that `bubble_sort` requires $n(n-1)/2$ comparisons and possible swapping operations.

On the other hand, the idea of `merge_sort` is to split the list into two approximately equal parts, sort them separately and then merge them into a single list:

```
procedure merge_sort(s,i,j)
  if i=j then
    return
  m := floor((i+j)/2)
  call merge_sort(s,i,m)
  call merge_sort(s,m+1,j)
  call merge(s,i,m,j,c)
  for k:=i to j do
    s[k] := c[k]
end merge_sort
```

The procedure `merge(s,i,m,j,c)` merges the two increasing sequences $s_i, s_{i+1}, \dots, s_m$ and $s_{m+1}, s_{m+2}, \dots, s_j$ into a single increasing sequence $c_i, c_{i+1}, \dots, c_j$. This algorithm is more efficient than `bubble_sort` because it requires only about $n \log_2 n$ operations (we will make this more precise soon).

The strategy of dividing a task into several smaller tasks is called *divide and conquer*.

3.1.3. Complexity. In general the *complexity* of an algorithm is the amount of time and space (memory use) required to execute it. Here we deal with *time complexity* only.

Since the actual time required to execute an algorithm depends on the details of the program implementing the algorithm and the speed and other characteristics of the machine executing it, it is in general impossible to make an estimation in actual physical time, however it is possible to measure the length of the computation in other ways, say by the number of operations performed. For instance the following loop performs the statement `x := x + 1` exactly n times,

```
for i := 1 to n do
  x := x + 1
```

The following double loop performs it n^2 times:

```
for i := 1 to n do
  for j := 1 to n do
    x := x + 1
```

The following one performs it $1 + 2 + 3 + \cdots + n = n(n + 1)/2$ times:

```
for i := 1 to n do
  for j := 1 to i do
    x := x + 1
```

Since the time that takes to execute an algorithm usually depends on the input, its complexity must be expressed as a function of the input, or more generally as a function of the *size* of the input. Since the execution time may be different for inputs of the same size, we define the following kinds of times:

1. *Best-case time*: minimum time needed to execute the algorithm among all inputs of a given size n.
2. *Wost-case time*: maximum time needed to execute the algorithm among all inputs of a given size n.
3. *Average-case time*: average time needed to execute the algorithm among all inputs of a given size n.

For instance, assume that we have a list of n objects one of which is colored red and the others are colored blue, and we want to find the one that is colored red by examining the objects one by one. We measure time by the number of objects examined. In this problem the minimum time needed to find the red object would be 1 (in the lucky event that the first object examined turned out to be the red one). The maximum time would be n (if the red object turns out to be the last one). The average time is the average of all possible times: $1, 2, 3, \dots, n$, which is $(1+2+3+\cdots+n)/n = (n+1)/2$. So in this example the best-case time is 1, the worst-case time is n and the average-case time is $(n + 1)/2$.

Often the exact time is too hard to compute or we are interested just in how it grows compared to the size of the input. For instance and algorithm that requires exactly $7n^2 + 3n + 10$ steps to be executed on an input of size n is said to be or *order* n^2, represented $\Theta(n^2)$. This justifies the following notations:

Big Oh Notation. A function $f(n)$ is said to be of order at most $g(n)$, written $f(n) = O(g(n))$, if there is a constant C_1 such that

$$|f(n)| \leq C_1 |g(n)|$$

for all but finitely many positive integers n.

Omega Notation. A function $f(n)$ is said to be of order at least $g(n)$, written $f(n) = \Omega(g(n))$, if there is a constant C_2 such that

$$|f(n)| \geq C_2 |g(n)|$$

for all but finitely many positive integers n.

Theta Notation. A function $f(n)$ is said to be of order $g(n)$, written $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Remark: All logarithmic functions are of the same order: $\log_a n = \Theta(\log_b n)$ for any $a, b > 1$, because $\log_a n = \log_b n / \log_b a$, so they always differ in a multiplicative constant. As a consequence, if the execution time of an algorithm is of order a logarithmic function, we can just say that its time is "logarithmic", we do not need to specify the base of the logarithm.

The following are several common growth functions:

Order	Name
$\Theta(1)$	Constant
$\Theta(\log \log n)$	Log log
$\Theta(\log n)$	Logarithmic
$\Theta(n \log n)$	n log n
$\Theta(n)$	Linear
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k)$	Polynomial
$\Theta(a^n)$	Exponential

Let's see now how we find the complexity of algorithms like `bubble_sort` and `merge_sort`.

Since `bubble_sort` is just a double loop its complexity is easy to find; the inner loop is executed

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

times, so it requires $n(n-1)/2$ comparisons and possible swap operations. Hence its execution time is $\Theta(n^2)$.

The estimation of the complexity of `merge_sort` is more involved. First, the number of operations required by the merge procedure is $\Theta(n)$. Next, if we call $T(n)$ (the order of) the number of operations required by `merge_sort` working on a list of size n, we see that roughly:

$$T(n) = 2T(n/2) + n\,.$$

Replacing n with $n/2$ we have $T(n/2) = 2T(n/4) + n/2$, hence

$$T(n) = 2T(n/2) + n = 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n\,.$$

Repeating k times we get:

$$T(n) = 2^k T(n/2^k) + kn\,.$$

So for $k = \log_2 n$ we have

$$T(n) = nT(1) + n\log_2 n = \Theta(n \log n)\,.$$

3.2. The Euclidean Algorithm

3.2.1. The Division Algorithm. The following result is known as *The Division Algorithm*:[1] If $a, b \in \mathbb{Z}$, $b > 0$, then there exist unique $q, r \in \mathbb{Z}$ such that $a = qb + r$, $0 \leq r < b$. Here q is called *quotient* of the *integer division* of a by b, and r is called *remainder*.

3.2.2. Divisibility. Given two integers a, b, $b \neq 0$, we say that b *divides* a, written $b|a$, if there is some integer q such that $a = bq$:

$$b|a \Leftrightarrow \exists q,\ a = bq\,.$$

We also say that b *divides* or is a *divisor of* a, or that a is a *multiple* of b.

3.2.3. Prime Numbers. A *prime* number is an integer $p \geq 2$ whose only positive divisors are 1 and p. Any integer $n \geq 2$ that is not prime is called *composite*. A non-trivial divisor of $n \geq 2$ is a divisor d of n such that $1 < d < n$, so $n \geq 2$ is composite iff it has non-trivial divisors. *Warning*: 1 is not considered either prime or composite.

Some results about prime numbers:

1. For all $n \geq 2$ there is some prime p such that $p|n$.
2. (Euclid) There are infinitely many prime numbers.
3. If $p|ab$ then $p|a$ or $p|b$. More generally, if $p|a_1 a_2 \dots a_n$ then $p|a_k$ for some $k = 1, 2, \dots, n$.

3.2.4. The Fundamental Theorem of Arithmetic. Every integer $n \geq 2$ can be written as a product of primes uniquely, up to the order of the primes.

It is customary to write the factorization in the following way:

$$n = p_1^{s_1}\, p_2^{s_2} \dots p_k^{s_k}\,,$$

where all the exponents are positive and the primes are written so that $p_1 < p_2 < \cdots < p_k$. For instance:

$$13104 = 2^4 \cdot 3^2 \cdot 7 \cdot 13\,.$$

[1]The result is not really an "algorithm", it is just a mathematical theorem. There are, however, algorithms that allow us to compute the quotient and the remainder in an integer division.

3.2.5. Greatest Common Divisor. A positive integer d is called a *common divisor* of the integers a and b, if d divides a and b. The greatest possible such d is called the *greatest common divisor* of a and b, denoted $\gcd(a, b)$. If $\gcd(a, b) = 1$ then a, b are called *relatively prime.*

Example: The set of positive divisors of 12 and 30 is $\{1, 2, 3, 6\}$. The greatest common divisor of 12 and 30 is $\gcd(12, 30) = 6$.

A few properties of divisors are the following. Let m, n, d be integers. Then:

1. If $d|m$ and $d|n$ then $d|(m + n)$.
2. If $d|m$ and $d|n$ then $d|(m - n)$.
3. If $d|m$ then $d|mn$.

Another important result is the following: Given integers a, b, c, the equation

$$ax + by = c$$

has integer solutions if and only if $\gcd(a, b)$ divides c. That is an example of a *Diophantine equation.* In general a Diophantine equation is an equation whose solutions must be integers.

Example: We have $\gcd(12, 30) = 6$, and in fact we can write $6 = 1 \cdot 30 - 2 \cdot 12$. The solution is not unique, for instance $6 = 3 \cdot 30 - 7 \cdot 12$.

3.2.6. Finding the gcd by Prime Factorization. We have that $\gcd(a, b) =$ product of the primes that occur in the prime factorizations of both a and b, raised to their lowest exponent. For instance $1440 = 2^5 \cdot 3^2 \cdot 5$, $1512 = 2^3 \cdot 3^3 \cdot 7$, hence $\gcd(1440, 1512) = 2^3 \cdot 3^2 = 72$.

Factoring numbers is not always a simple task, so finding the gcd by prime factorization might not be a most convenient way to do it, but there are other ways.

3.2.7. The Euclidean Algorithm. Now we examine an alternative method to compute the gcd of two given positive integers a, b. The method provides at the same time a solution to the Diophantine equation:

$$ax + by = \gcd(a, b)\,.$$

It is based on the following fact: given two integers $a \geq 0$ and $b > 0$, and $r = a \bmod b$, then $\gcd(a, b) = \gcd(b, r)$. Proof: Divide a by

b obtaining a quotient q and a reminder r, then

$$a = bq + r\,, \quad 0 \le r < b\,.$$

If d is a common divisor of a and b then it must be a divisor of $r = a - bq$. Conversely, if d is a common divisor of b and r then it must divide $a = bq + r$. So the set of common divisors of a and b and the set of common divisors of b and r are equal, and the greatest common divisor will be the same.

The Euclidean algorithm is a follows. First we divide a by b, obtaining a quotient q and a reminder r. Then we divide b by r, obtaining a new quotient q' and a reminder r'. Next we divide r by r', which gives a quotient q'' and another remainder r''. We continue dividing each reminder by the next one until obtaining a zero reminder, and which point we stop. The last non-zero reminder is the gcd.

Example: Assume that we wish to compute $\gcd(500, 222)$. Then we arrange the computations in the following way:

$$\begin{array}{rcll} 500 & = & 2\cdot 222 + 56 & \rightarrow \quad r = 56 \\ 222 & = & 3\cdot 56 + 54 & \rightarrow \quad r' = 54 \\ 56 & = & 1\cdot 54 + 2 & \rightarrow \quad r'' = 2 \\ 54 & = & 27\cdot 2 + 0 & \rightarrow \quad r''' = 0 \end{array}$$

The last nonzero remainder is $r'' = 2$, hence $\gcd(500, 222) = 2$. Furthermore, if we want to express 2 as a linear combination of 500 and 222, we can do it by working backward:

$$\begin{aligned} 2 = 56 - 1\cdot 54 = 56 - 1\cdot(222 - 3\cdot 56) = 4\cdot 56 - 1\cdot 222 \\ = 4\cdot(500 - 2\cdot 222) - 1\cdot 222 = 4\cdot 500 - 9\cdot 222\,. \end{aligned}$$

The algorithm to compute the gcd can be written as follows:

```
procedure gcd(a,b)
  if a<b then    // make a the largest
    swap(a,b)
  while b ≠ 0 do
    begin
      r := a mod b
      a := b
      b := r
    end
  return(a)
end gcd
```

The next one is a recursive version of the Euclidean algorithm:

```
procedure gcd_recurs(a,b)
  if b=0 then
    return(a)
  else
    return(gcd_recurs(b,a mod b))
end gcd_recurs
```

3.3. Modular Arithmetic, RSA Algorithm

3.3.1. Congruences Modulo m. Given an integer $m \geq 2$, we say that a is congruent to b modulo m, written $a \equiv b \pmod{m}$, if $m|(a-b)$. Note that the following conditions are equivalent

1. $a \equiv b \pmod{m}$.
2. $a = b + km$ for some integer k.
3. a and b have the same remainder when divided by m.

The relation of congruence modulo m is an equivalence relation. It partitions $\mathbb{Z}$ into m equivalence classes of the form

$$[x] = [x]_m = \{x + km \mid k \in \mathbb{Z}\}\,.$$

For instance, for $m = 5$, each one of the following rows is an equivalence class:

...	-10	-5	0	5	10	15	20	...
...	-9	-4	1	6	11	16	21	...
...	-8	-3	2	7	12	17	22	...
...	-7	-2	3	8	13	18	23	...
...	-6	-1	4	9	14	19	24	...

Each equivalence class has exactly a representative r such that $0 \leq r < m$, namely the common remainder of all elements in that class when divided by m. Hence an equivalence class may be denoted $[r]$ or $x + m\,\mathbb{Z}$, where $0 \leq r < m$. Often we will omit the brackets, so that the equivalence class $[r]$ will be represented just r. The set of equivalence classes (i.e., the quotient set of $\mathbb{Z}$ by the relation of congruence modulo m) is denoted $\mathbb{Z}_m = \{0, 1, 2, \dots, m-1\}$. For instance, $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$.

Remark: When writing "r" as a notation for the class of r we may stress the fact that r represents the class of r rather than the integer r by including " $\pmod{p}$" at some point. For instance $8 = 3 \pmod{p}$. Note that in "$a \equiv b \pmod{m}$", a and b represent integers, while in "$a = b \pmod{m}$" they represent elements of $\mathbb{Z}_m$.

Reduction Modulo m: Once a set of representatives has been chosen for the elements of $\mathbb{Z}_m$, we will call "*r reduced modulo m*", written "$r \bmod m$", the chosen representative for the class of r. For instance, if we choose the representatives for the elements of $\mathbb{Z}_5$ in the interval from 0 to 4 ($\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$), then $9 \bmod 5 = 4$. Another possibility is to choose the representatives in the interval from -2 to 2 ($\mathbb{Z}_5 = \{-2, -1, 0, 1, 2\}$), so that $9 \bmod 5 = -1$

In $\mathbb{Z}_m$ it is possible to define an *addition* and a *multiplication* in the following way:

$$[x] + [y] = [x + y]\,; \qquad [x] \cdot [y] = [x \cdot y]\,.$$

As an example, tables 3.3.1 and 3.3.2 show the addition and multiplication tables for $\mathbb{Z}_5$ and $\mathbb{Z}_6$ respectively.

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\cdot$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

TABLE 3.3.1. Operational tables for $\mathbb{Z}_5$

$+$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

$\cdot$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

TABLE 3.3.2. Operational tables for $\mathbb{Z}_6$

A difference between this two tables is that in $\mathbb{Z}_5$ every non-zero element has a multiplicative inverse, i.e., for every $x \in \mathbb{Z}_5$ such that $x \neq 0$ there is an x^{-1} such that $x \cdot x^{-1} = x^{-1} \cdot x = 1$; e.g. $2^{-1} = 4 \pmod 5$. However in $\mathbb{Z}_6$ that is not true, some non-zero elements like 2 have no multiplicative inverse. Furthermore the elements without multiplicative inverse verify that they can be multiply by some other non-zero element giving a product equal zero, e.g. $2 \cdot 3 = 0 \pmod 6$. These elements are called divisors of zero. Of course with this definition zero itself is a divisor of zero. Divisors of zero different from zero are called *proper divisors of zero.* For instance in $\mathbb{Z}_6$ 2 is a proper divisor of zero. In $\mathbb{Z}_5$ there are no proper divisors of zero.

In general:

1. The elements of $\mathbb{Z}_m$ can be classified into two classes:

   (a) *Units*: elements with multiplicative inverse.
   (b) *Divisors of zero*: elements that multiplied by some other non-zero element give product zero.
2. An element $[a] \in \mathbb{Z}_m$ is a unit (has a multiplicative inverse) if and only if $\gcd(a, m) = 1$.
3. All non-zero elements of $\mathbb{Z}_m$ are units if and only if m is a prime number.

The set of units in $\mathbb{Z}_m$ is denoted $\mathbb{Z}_m^*$. For instance:

$$
\begin{aligned}
\mathbb{Z}_2^* &= \{1\} \\
\mathbb{Z}_3^* &= \{1, 2\} \\
\mathbb{Z}_4^* &= \{1, 3\} \\
\mathbb{Z}_5^* &= \{1, 2, 3, 4\} \\
\mathbb{Z}_6^* &= \{1, 5\} \\
\mathbb{Z}_7^* &= \{1, 2, 3, 4, 5, 6\} \\
Z_8^* &= \{1, 3, 5, 7\} \\
Z_9^* &= \{1, 2, 4, 5, 7, 8\}
\end{aligned}
$$

Given an element $[a]$ in $\mathbb{Z}_m^*$, its inverse can be computed by using the Euclidean algorithm to find $\gcd(a, m)$, since that algorithm also provides a solution to the equation $ax + my = \gcd(a, m) = 1$, which is equivalent to $ax \equiv 1 \pmod{m}$.

Example: Find the multiplicative inverse of 17 in $\mathbb{Z}_{64}^*$. *Answer*: We use the Euclidean algorithm:

$$
\begin{array}{rcll}
64 & = & 3 \cdot 17 + 13 & \rightarrow \quad r = 13 \\
17 & = & 1 \cdot 13 + 4 & \rightarrow \quad r = 4 \\
13 & = & 3 \cdot 4 + 1 & \rightarrow \quad r = 1 \\
4 & = & 4 \cdot 1 + 0 & \rightarrow \quad r = 0
\end{array}
$$

Now we compute backward:

$$
\begin{aligned}
1 = 13 - 3 \cdot 4 = 13 - 3 \cdot (17 - 1 \cdot 13) &= 4 \cdot 13 - 3 \cdot 17 \\
&= 4 \cdot (64 - 3 \cdot 17) - 3 \cdot 17 = 4 \cdot 64 - 15 \cdot 17 \,.
\end{aligned}
$$

Hence $(-15) \cdot 17 \equiv 1 \pmod{64}$, but $-15 \equiv 49 \pmod{64}$, so the inverse of 17 in $(\mathbb{Z}_{64}^*, \cdot)$ is 49. We will denote this by writing $17^{-1} = 49$ (mod 64), or $17^{-1} \bmod 64 = 49$.

3.3.2. Euler's Phi Function. The number of units in $\mathbb{Z}_m$ is equal to the number of positive integers not greater than and relatively prime to m, i.e., the number of integers a such that $1 \leq a \leq m$ and $\gcd(a,m) = 1$. That number is given by the so called *Euler's phi function*:

$$\begin{aligned}\phi(m) = {} & \text{number of positive integers not greater than } m \\ & \text{and relatively prime to } m\,.\end{aligned}$$

For instance, the positive integers not greater than and relatively prime to 15 are: $1, 2, 4, 7, 8, 11, 13, 14$, hence $\phi(15) = 8$.

We have the following results:

1. If p is a prime number and $s \geq 1$, then $\phi(p^s) = p^s - p^{s-1} = p^s(1 - 1/p)$. In particular $\phi(p) = p - 1$.
2. If m_1, m_2 are two relatively prime positive integers, then $\phi(m_1 m_2) = \phi(m_1)\,\phi(m_2)$.[1]
3. If $m = p_1^{s_1} p_2^{s_2} \dots p_k^{s_k}$, where the p_k are prime and the s_k are positive, then
$$\phi(m) = m\,(1 - 1/p_1)\,(1 - 1/p_2)\,\dots\,(1 - 1/p_k)\,.$$

For instance

$$\phi(15) = \phi(3 \cdot 5) = \phi(3) \cdot \phi(5) = (3-1) \cdot (5-1) = 2 \cdot 4 = 8\,.$$

3.3.3. Euler's Theorem. If a and m are two relatively prime positive integers, $m \geq 2$, then

$$a^{\phi(m)} \equiv 1 \pmod{m}\,.$$

The particular case in which m is a prime number p, Euler's theorem is called *Fermat's Little Theorem*:

$$a^{p-1} \equiv 1 \pmod{p}\,.$$

For instance, if $a = 2$ and $p = 7$, then we have, in fact, $2^{7-1} = 2^6 = 64 = 1 + 9 \cdot 7 \equiv 1 \pmod{7}$.

A consequence of Euler's Theorem is the following. If $\gcd(a,m) = 1$ then

$$x \equiv y \pmod{\phi(m)} \quad \Rightarrow \quad a^x \equiv a^y \pmod{m}\,.$$

[1]A function $f(x)$ of positive integers such that $\gcd(a,b) = 1 \Rightarrow f(ab) = f(a)f(b)$ is called *multiplicative*.

Consequently, the following function is well defined:

$$\begin{aligned} \mathbb{Z}_m^* \times \mathbb{Z}_{\phi(m)} &\to \mathbb{Z}_m^* \\ ([a]_m, [x]_{\phi(m)}) &\mapsto [a^x]_m \end{aligned}$$

Hence, we can compute powers modulo m in the following way:

$$a^n = a^{n \bmod \phi(m)} \pmod{m},$$

if $\gcd(a, m) = 1$. For instance:

$$\begin{aligned} 3^{9734888} \bmod 100 &= 3^{9734888 \bmod \phi(100)} \bmod 100 \\ &= 3^{9734888 \bmod 40} \bmod 100 = 3^8 \bmod 100 = 6561 \bmod 100 = 61\,. \end{aligned}$$

An even more efficient way to compute powers modulo m is given in Appendix A, paragraph A.1.

3.3.4. Application to Cryptography: RSA Algorithm. The RSA algorithm is an encryption scheme designed in 1977 by Ronald Rivest, Adi Shamir and Leonard Adleman. It allows encrypting a message with a key (the *encryption key*) and decrypting it with a different key (the *decryption key*). The encryption key is public and can be given to everybody. The decryption key is private and is known only by the recipient of the encrypted message.

The RSA algorithm is based on the following facts. Given two prime numbers p and q, and a positive number m relatively prime to p and q, Euler's theorem tells us that:

$$m^{\phi(pq)} = m^{(p-1)(q-1)} = 1 \pmod{pq}\,.$$

Assume now that we have two integers e and d such that $e \cdot d = 1 \pmod{\phi(pq)}$. Then we have that

$$(m^e)^d = m^{e \cdot d} = m \pmod{pq}\,.$$

So, given m^e we can recover m modulo pq by raising to the dth power.

The RSA algorithm consists of the following:

1. Generate two large primes p and q. Find their product $n = pq$.
2. Find two numbers e and d (in the range from 2 to $\phi(n)$) such that $e \cdot d = 1 \pmod{\phi(n)}$. This requires some trial and error. First e is chosen at random, and the Euclidean algorithm is used to find $\gcd(e, m)$, solving at the same time the equation $ex + my = \gcd(e, m)$. If $\gcd(e, m) = 1$ then the value obtained

for x is d. Otherwise, e is no relatively prime to $\phi(n)$ and we must try a different value for e.

3. The *public encryption key* will be the pair (n, e). The *private decryption key* will be the pair (n, d). The encryption key is given to everybody, while the decryption key is kept secret by the future recipient of the message.

4. The message to be encrypted is divided into small pieces, and each piece is encoded numerically as a positive integer m smaller than n.

5. The number m^e is reduced modulo n; $m' = m^e \bmod n$.

6. The recipient computes $m'' = m'^d \bmod n$, with $0 \leq m'' < n$.

It remains to prove that $m'' = m$. If m is relatively prime to p and q, then from Euler's theorem we get that $m'' = m \pmod{n}$, and since both are in the range from 0 to $n - 1$ they must be equal. The case in which p or q divides m is left as an exercise.

CHAPTER 4

Counting

4.1. Basic Principles

4.1.1. The Rule of Sum. If a task can be performed in m ways, while another task can be performed in n ways, and the two tasks cannot be performed simultaneously, then performing either task can be accomplished in $m+n$ ways.

Set theoretical version of the rule of sum: If A and B are disjoint sets ($A \cap B = \emptyset$) then

$$|A \cup B| = |A| + |B| .$$

More generally, if the sets $A_1, A_2, \dots, A_n$ are pairwise disjoint, then:

$$|A_1 \cup A_2 \cup \cdots \cup A_n| = |A_1| + |A_2| + \cdots + |A_n| .$$

For instance, if a class has 30 male students and 25 female students, then the class has $30 + 25 = 45$ students.

4.1.2. The Rule of Product. If a task can be performed in m ways and another independent task can be performed in n ways, then the combination of both tasks can be performed in mn ways.

Set theoretical version of the rule of product: Let $A \times B$ be the Cartesian product of sets A and B. Then:

$$|A \times B| = |A| \cdot |B| .$$

More generally:

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

For instance, assume that a license plate contains two letters followed by three digits. How many different license plates can be printed? *Answer*: each letter can be printed in 26 ways, and each digit can be printed in 10 ways, so $26 \cdot 26 \cdot 10 \cdot 10 \cdot 10 = 676000$ different plates can be printed.

Exercise: Given a set A with m elements and a set B with n elements, find the number of functions from A to B.

4.1.3. The Inclusion-Exclusion Principle.

4.1.3. The Inclusion-Exclusion Principle. The *inclusion-exclusion principle* generalizes the rule of sum to non-disjoint sets.

In general, for arbitrary (but finite) sets A, B:

$$|A \cup B| = |A| + |B| - |A \cap B| \,.$$

Example: Assume that in a university with 1000 students, 200 students are taking a course in mathematics, 300 are taking a course in physics, and 50 students are taking both. How many students are taking at least one of those courses?

Answer: If U = total set of students in the university, M = set of students taking Mathematics, P = set of students taking Physics, then:

$$|M \cup P| = |M| + |P| - |M \cap P| = 300 + 200 - 50 = 450$$

students are taking Mathematics or Physics.

For three sets the following formula applies:

$$|A \cup B \cup C| = \\ |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C| \,,$$

and for an arbitrary union of sets:

$$|A_1 \cup A_2 \cup \cdots \cup A_n| = s_1 - s_2 + s_3 - s_4 + \cdots \pm s_n \,,$$

where s_k = sum of the cardinalities of all possible k-fold intersections of the given sets.

4.2. Combinatorics

4.2.1. Permutations. Assume that we have n objects. Any arrangement of any k of these objects in a given order is called a *permutation* of size k. If $k = n$ then we call it just a *permutation* of the n objects. For instance, the permutations of the letters a, b, c are the following: *abc*, *acb*, *bac*, *bca*, *cab*, *cba*. The permutations of size 2 of the letters a, b, c, d are: *ab*, *ac*, *ad*, *ba*, *bc*, *bd*, *ca*, *cb*, *cd*, *da*, *db*, *dc*.

Note that the order is important. Given two permutations, they are considered equal if they have the same elements arranged in the same order.

We find the number $P(n, k)$ of permutations of size k of n given objects in the following way: The first object in an arrangement can be chosen in n ways, the second one in $n-1$ ways, the third one in $n-2$ ways, and so on, hence:

$$P(n,k) = n \times (n-1) \times \overset{(k \text{ factors})}{\cdots} \times (n-k+1) = \frac{n!}{(n-k)!}\,,$$

where $n! = 1 \times 2 \times 3 \times \overset{(n \text{ factors})}{\cdots} \times n$ is called "n *factorial*".

The number $P(n, k)$ of permutations of n objects is

$$P(n,n) = n!\,.$$

By convention $0! = 1$.

For instance, there are $3! = 6$ permutations of the 3 letters a, b, c. The number of permutations of size 2 of the 4 letters a, b, c, d is $P(4, 2) = 4 \times 3 = 12$.

Exercise: Given a set A with m elements and a set B with n elements, find the number of one-to-one functions from A to B.

4.2.2. Combinations. Assume that we have a set A with n objects. Any subset of A of size r is called a *combination of n elements taken r at a time.* For instance, the combinations of the letters a, b, c, d, e taken 3 at a time are: *abc*, *abd*, *abe*, *acd*, *ace*, *ade*, *bcd*, *bce*, *bde*, *cde*, where two combinations are considered identical if they have the same elements regardless of their order.

The number of subsets of size r in a set A with n elements is:

$$C(n,r) = \frac{n!}{r!\,(n-r)!}\,.$$

The symbol $\binom{n}{r}$ (read "n choose r") is often used instead of $C(n,r)$.

One way to derive the formula for $C(n,r)$ is the following. Let A be a set with n objects. In order to generate all possible permutations of size r of the elements of A we 1) take all possible subsets of size r in the set A, and 2) permute the k elements in each subset in all possible ways. Task 1) can be performed in $C(n,r)$ ways, and task 2) can be performed in $P(r,r)$ ways. By the product rule we have $P(n,r) = C(n,r) \times P(r,r)$, hence

$$C(n,r) = \frac{P(n,r)}{P(r,r)} = \frac{n!}{r!\,(n-r)!}\,.$$

4.3. Generalized Permutations and Combinations

4.3.1. Permutations with Repeated Elements. Assume that we have an alphabet with k letters and we want to write all possible words containing n_1 times the first letter of the alphabet, n_2 times the second letter,..., n_k times the kth letter. How many words can we write? We call this number $P(n; n_1, n_2, \dots, n_k)$, where $n = n_1 + n_2 + \cdots + n_k$.

Example: With 3 a's and 2 b's we can write the following 5-letter words: *aaabb*, *aabab*, *abaab*, *baaab*, *aabba*, *ababa*, *baaba*, *abbaa*, *babaa*, *bbaaa*.

We may solve this problem in the following way, as illustrated with the example above. Let us distinguish the different copies of a letter with subscripts: $a_1a_2a_3b_1b_2$. Next, generate each permutation of this five elements by choosing 1) the position of each kind of letter, then 2) the subscripts to place on the 3 a's, then 3) these subscripts to place on the 2 b's. Task 1) can be performed in $P(5; 3, 2)$ ways, task 2) can be performed in 3! ways, task 3) can be performed in 2!. By the product rule we have $5! = P(5; 3, 2) \times 3! \times 2!$, hence $P(5; 3, 2) = 5!/3!\,2!$.

In general the formula is:

$$P(n; n_1, n_2, \dots, n_k) = \frac{n!}{n_1!\, n_2! \,\dots\, n_k!}\,.$$

4.3.2. Combinations with Repetition. Assume that we have a set A with n elements. Any selection of r objects from A, where each object can be selected more than once, is called a *combination of n objects taken r at a time with repetition.* For instance, the combinations of the letters a, b, c, d taken 3 at a time with repetition are: *aaa*, *aab*, *aac*, *aad*, *abb*, *abc*, *abd*, *acc*, *acd*, *add*, *bbb*, *bbc*, *bbd*, *bcc*, *bcd*, *bdd*, *ccc*, *ccd*, *cdd*, *ddd*. Two combinations with repetition are considered identical if they have the same elements repeated the same number of times, regardless of their order.

Note that the following are equivalent:

1. The number of combinations of n objects taken r at a time with repetition.

2. The number of ways r identical objects can be distributed among n distinct containers.

3. The number of nonnegative integer solutions of the equation:
$$x_1 + x_2 + \cdots + x_n = r\,.$$

Example: Assume that we have 3 different (empty) milk containers and 7 quarts of milk that we can measure with a one quart measuring cup. In how many ways can we distribute the milk among the three containers? We solve the problem in the following way. Let x_1, x_2, x_3 be the quarts of milk to put in containers number 1, 2 and 3 respectively. The number of possible distributions of milk equals the number of non negative integer solutions for the equation $x_1 + x_2 + x_3 = 7$. Instead of using numbers for writing the solutions, we will use strokes, so for instance we represent the solution $x_1 = 2, x_2 = 1, x_3 = 4$, or $2 + 1 + 4$, like this: $||+|+||||$. Now, each possible solution is an arrangement of 7 strokes and 2 plus signs, so the number of arrangements is $P(9; 7, 2) = 9!/7!\,2! = \binom{9}{7}$.

The general solution is:
$$P(n + r - 1; r, n - 1) = \frac{(n + r - 1)!}{r!\,(n - 1)!} = \binom{n + r - 1}{r}\,.$$

4.4. Binomial Coefficients

4.4.1. Binomial Theorem. The following identities can be easily checked:

$$\begin{aligned}
(x+y)^0 &= 1 \\
(x+y)^1 &= x+y \\
(x+y)^2 &= x^2+2\,xy+y^2 \\
(x+y)^3 &= x^3+3\,x^2y+3\,xy^2+y^3
\end{aligned}$$

They can be generalized by the following formula, called the *Binomial Theorem*:

$$\begin{aligned}
(x+y)^n &= \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k \\
&= \binom{n}{0} x^n + \binom{n}{1} x^{n-1} y + \binom{n}{2} x^{n-2} y^2 + \cdots \\
&\qquad\qquad + \binom{n}{n-1} x y^{n-1} + \binom{n}{n} y^n .
\end{aligned}$$

We can find this formula by writing

$$(x+y)^n = (x+y) \times (x+y) \times \overset{(n \text{ factors})}{\cdots} \times (x+y) ,$$

expanding, and grouping terms of the form $x^a y^b$. Since there are n factors of the form $(x+y)$, we have $a+b=n$, hence the terms must be of the form $x^{n-k}y^k$. The coefficient of $x^{n-k}y^k$ will be equal to the number of ways in which we can select the y from any k of the factors (and the x from the remaining $n-k$ factors), which is $C(n,k)=\binom{n}{k}$. The expression $\binom{n}{k}$ is often called *binomial coefficient.*

Exercise: Prove

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \qquad \text{and} \qquad \sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0 .$$

Hint: Apply the binomial theorem to $(1+1)^2$ and $(1-1)^2$.

4.4.2. Properties of Binomial Coefficients. The binomial coefficients have the following properties:

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

2. $\displaystyle\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$

The first property follows easily from $\displaystyle\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The second property can be proved by choosing a distinguished element a in a set A of $n+1$ elements. The set A has $\binom{n+1}{k+1}$ subsets of size $k+1$. Those subsets can be partitioned into two classes: that of the subsets containing a, and that of the subsets not containing a. The number of subsets containing a equals the number of subsets of $A - \{a\}$ of size k, i.e., $\binom{n}{k}$. The number of subsets not containing a is the number of subsets of $A - \{a\}$ of size $k+1$, i.e., $\binom{n}{k+1}$. Using the sum principle we find that in fact $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$.

4.4.3. Pascal's Triangle. The properties shown in the previous section allow us to compute binomial coefficients in a simple way. Look at the following triangular arrangement of binomial coefficients:

$$\begin{array}{ccccccccc}
 & & & & \binom{0}{0} & & & & \\
 & & & \binom{1}{0} & & \binom{1}{1} & & & \\
 & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & \\
 & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & \\
\binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4}
\end{array}$$

We notice that each binomial coefficient on this arrangement must be the sum of the two closest binomial coefficients on the line above it. This together with $\binom{n}{0} = \binom{n}{n} = 1$, allows us to compute very quickly the values of the binomial coefficients on the arrangement:

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

This arrangement of binomial coefficients is called *Pascal's Triangle.*[1]

[1]Although it was already known by the Chinese in the XIV century.

4.5. The Pigeonhole Principle

4.5.1. The Pigeonhole Principle. The *pigeonhole principle* is used for proving that a certain situation must actually occur. It says the following: *If n pigeonholes are occupied by m pigeons and $m > n$, then at least one pigeonhole is occupied by more than one pigeon.*[1]

Example: In any given set of 13 people at least two of them have their birthday during the same month.

Example: Let S be a set of eleven 2-digit numbers. Prove that S must have two elements whose digits have the same difference (for instance in $S = \{10, 14, 19, 22, 26, 28, 49, 53, 70, 90, 93\}$, the digits of the numbers 28 and 93 have the same difference: $8 - 2 = 6$, $9 - 3 = 6$.) *Answer*: The digits of a two-digit number can have 10 possible differences (from 0 to 9). So, in a list of 11 numbers there must be two with the same difference.

Example: Assume that we choose three different digits from 1 to 9 and write all permutations of those digits. Prove that among the 3-digit numbers written that way there are two whose difference is a multiple of 500. *Answer*: There are $9 \cdot 8 \cdot 7 = 504$ permutations of three digits. On the other hand if we divide the 504 numbers by 500 we can get only 500 possible remainders, so at least two numbers give the same remainder, and their difference must be a multiple of 500.

Exercise: Prove that if we select $n + 1$ numbers from the set $S = \{1, 2, 3, \dots, 2n\}$, among the numbers selected there are two such that one is a multiple of the other one.

[1]The *Pigeonhole Principle* (*Schubfachprinzip*) was first used by Dirichlet in Number Theory. The term *pigeonhole* actually refers to one of those old-fashioned writing desks with thin vertical wooden partitions in which to file letters.

4.6. Probability

4.6.1. Introduction. Assume that we perform an experiment such as tossing a coin or rolling a die. The set of possible outcomes is called the *sample space* of the experiment. An *event* is a subset of the sample space. For instance, if we toss a coin three times, the sample space is

$$S = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\}\,.$$

The event "at least two heads in a row" would be the subset

$$E = \{HHH, HHT, THH\}\,.$$

If all possible outcomes of an experiment have the same likelihood of occurrence, then the probability of an event $A \subset \mathcal{S}$ is given by Laplace's rule:

$$P(E) = \frac{|E|}{|S|}\,.$$

For instance, the probability of getting at least two heads in a row in the above experiment is 3/8.

4.6.2. Probability Function. In general the likelihood of different outcomes of an experiment may not be the same. In that case the probability of each possible outcome x is a function $P(x)$. This function verifies:

$$0 \leq P(x) \leq 1 \quad \text{for all } x \in S$$

and

$$\sum_{x\in S} P(x) = 1\,.$$

The probability of an event $E \subseteq S$ will be

$$P(E) = \sum_{x\in E} P(x)$$

Example: Assume that a die is loaded so that the probability of obtaining n point is proportional to n. Find the probability of getting an odd number when rolling that die.

Answer: First we must find the probability function $P(n)$ ($n = 1, 2, \dots, 6$). We are told that $P(n)$ is proportional to n, hence $P(n) = kn$. Since $P(S) = 1$ we have $P(1)+P(2)+\cdots P(6) = 1$, i.e., $k\cdot 1+k\cdot 2+\cdots + k\cdot 6 = 21k = 1$, so $k = 1/21$ and $P(n) = n/21$. Next we want to find the probability of $E = \{2, 4, 6\}$, i.e. $P(E) = P(2)+P(4)+P(6) = \frac{2}{21} + \frac{4}{21} + \frac{6}{21} = \boxed{\frac{12}{21}}$.

4.6.3. Properties of probability. Let P be a probability function on a sample space S. Then:

1. For every event $E \subseteq S$,
$$0 \leq P(E) \leq 1\,.$$
2. $P(\emptyset) = 0$, $P(S) = 1$.
3. For every event $E \subseteq S$, if $\overline{E} =$ is the complement of E ("not E") then
$$P(\overline{E}) = 1 - P(E)\,.$$
4. If $E_1, E_2 \subseteq S$ are two events, then
$$P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2)\,.$$
In particular, if $E_1 \cap E_2 = \emptyset$ (E_1 and E_2 are *mutually exclusive*, i.e., they cannot happen at the same time) then
$$P(E_1 \cup E_2) = P(E_1) + P(E_2)\,.$$

Example: Find the probability of getting a sum different from 10 or 12 after rolling two dice. *Answer*: We can get 10 in 3 different ways: 4+6, 5+5, 6+4, so $P(10) = 3/36$. Similarly we get that $P(12) = 1/36$. Since they are mutually exclusive events, the probability of getting 10 or 12 is $P(10)+P(12) = 3/36+1/36 = 4/36 = 1/9$. So the probability of *not* getting 10 or 12 is $1 - 1/9 = 8/9$.

4.6.4. Conditional Probability. The *conditional probability* of an event E given F, represented $P(E \mid F)$, is the probability of E assuming that F has occurred. It is like restricting the sample space to F. Its value is
$$P(E \mid F) = \frac{P(E \cap F)}{P(F)}\,.$$

Example: Find the probability of obtaining a sum of 10 after rolling two fair dice. Find the probability of that event if we know that at least one of the dice shows 5 points.

Answer: We call $E =$ "obtaining sum 10" and $F =$ "at least one of the dice shows 5 points". The number of possible outcomes is $6 \times 6 = 36$. The event "obtaining a sum 10" is $E = \{(4, 6), (5, 5), (6, 4)\}$, so

$|E| = 3$. Hence the probability is $P(E) = |E|/|S| = 3/36 = 1/12$. Now, if we know that at least one of the dice shows 5 points then the sample space shrinks to

$$F = \{(1,5),(2,5),(3,5),(4,5),(5,5),(6,5),(5,1),(5,2),(5,3),(5,4),(5,6)\}\,,$$

so $|F| = 11$, and the ways to obtain a sum 10 are $E \cap F = \{(5,5)\}$, $|E \cap F| = 1$, so the probability is $P(E \mid F) = P(E \cap F)/P(F) = 1/11$.

4.6.5. Independent Events. Two events E and F are said to be *independent* if the probability of one of them does not depend on the other, e.g.:

$$P(E \mid F) = P(E)\,.$$

In this circumstances:

$$P(E \cap F) = P(E) \cdot P(F)\,.$$

Note that if E is independent of F then also F is independent of E, e.g., $P(F \mid E) = P(F)$.

Example: Assume that the probability that a shooter hits a target is $p = 0.7$, and that hitting the target in different shots are independent events. Find:

1. The probability that the shooter does not hit the target in one shot.
2. The probability that the shooter does not hit the target three times in a row.
3. The probability that the shooter hits the target at least once after shooting three times.

Answer:

1. $P(\text{not hitting the target in one shot}) = 1 - 0.7 = 0.3$.
2. $P(\text{not hitting the target three times in a row}) = 0.3^3 = 0.027$.
3. $P(\text{hitting the target at least once in three shots}) = 1 - 0.027 = 0.973$.

4.6.6. Bayes' Theorem. Suppose that a sample space S is partitioned into n classes $C_1, C_2, \dots, C_n$ which are pairwise mutually exclusive and whose union fills the whole sample space. Then for any event F we have

$$P(F) = \sum_{i=1}^{n} P(F \mid C_i)\, P(C_i)$$

and

$$P(C_j \mid F) = \frac{P(F \mid C_j)\, P(C_j)}{P(F)}\,.$$

Example: In a country with 100 million people 100 thousand of them have disease X. A test designed to detect the disease has a 99% probability of detecting it when administered to a person who has it, but it also has a 5% probability of giving a false positive when given to a person who does not have it. A person is given the test and it comes out positive. What is the probability that that person has the disease?

Answer: The classes are C_1 = "has the disease" and C_2 = "does not have the disease", and the event is F = "the test gives a positive". We have: $|S| = 100{,}000{,}000$, $|C_1| = 100{,}000$, $|C_2| = 99{,}900{,}000$, hence $P(C_1) = |C_1|/|S| = 0.001$, $P(C_2) = |C_2|/|S| = 0.999$. Also $P(F \mid C_1) = 0.99$, $P(F \mid C_2) = 0.05$. Hence:

$$\begin{aligned} P(F) &= P(F \mid C_1) \cdot P(C_1) + P(F \mid C_2) \cdot P(C_2) \\ &= 0.99 \cdot 0.001 + 0.05 \cdot 0.999 = 0.05094\,, \end{aligned}$$

and by Bayes' theorem:

$$\begin{aligned} P(C_1 \mid F) &= \frac{P(F \mid C_1) \cdot P(C_1)}{P(F)} = \frac{0.99 \cdot 0.001}{0.05094} \\ &= 0.019434628 \cdots \approx 2\%\,. \end{aligned}$$

(So the test is really of little use when given to a random person—however it might be useful in combination with other tests or other evidence that the person might have the disease.)

CHAPTER 5

Recurrence Relations

5.1. Recurrence Relations

Here we look at recursive definitions under a different point of view. Rather than definitions they will be considered as equations that we must solve. The point is that a recursive definition is actually a definition when there is one and only one object satisfying it, i.e., when the equations involved in that definition have a unique solution. Also, the solution to those equations may provide a *closed-form* (explicit) formula for the object defined.

The recursive step in a recursive definition is also called a *recurrence relation*. We will focus on *kth-order linear recurrence relations*, which are of the form

$$C_0\, x_n + C_1\, x_{n-1} + C_2\, x_{n-2} + \cdots + C_k\, x_{n-k} = b_n \,,$$

where $C_0 \neq 0$. If $b_n = 0$ the recurrence relation is called *homogeneous*. Otherwise it is called *non-homogeneous*.

The basis of the recursive definition is also called *initial conditions* of the recurrence. So, for instance, in the recursive definition of the Fibonacci sequence, the recurrence is

$$F_n = F_{n-1} + F_{n-2}$$

or

$$F_n - F_{n-1} - F_{n-2} = 0\,,$$

and the initial conditions are

$$F_0 = 0,\ F_1 = 1\,.$$

One way to solve some recurrence relations is by *iteration*, i.e., by using the recurrence repeatedly until obtaining a explicit close-form formula. For instance consider the following recurrence relation:

$$x_n = r\, x_{n-1} \quad (n > 0)\,; \qquad x_0 = A\,.$$

By using the recurrence repeatedly we get:

$$x_n = r\,x_{n-1} = r^2\,x_{n-2} = r^3\,x_{n-3} = \cdots = r^n\,x_0 = A\,r^n\,,$$

hence the solution is $x_n = A\,r^n$.

In the following we assume that the coefficients $C_0, C_1, \ldots, C_k$ are constant.

5.1.1. First Order Recurrence Relations. The homogeneous case can be written in the following way:

$$x_n = r\,x_{n-1} \quad (n > 0)\,; \qquad x_0 = A\,.$$

Its general solution is

$$x_n = A\,r^n\,,$$

which is a *geometric sequence* with *ratio* r.

The non-homogeneous case can be written in the following way:

$$x_n = r\,x_{n-1} + c_n \quad (n > 0)\,; \qquad x_0 = A\,.$$

Using the summation notation, its solution can be expressed like this:

$$x_n = A\,r^n + \sum_{k=1}^{n} c_k\,r^{n-k}\,.$$

We examine two particular cases. The first one is

$$x_n = r\,x_{n-1} + c \quad (n > 0); \qquad x_0 = A\,.$$

where c is a constant. The solution is

$$x_n = A\,r^n + c\sum_{k=1}^{n} r^{n-k} = A\,r^n + c\,\frac{r^n - 1}{r - 1} \qquad \text{if } r \neq 1\,,$$

and

$$x_n = A + c\,n \qquad \text{if } r = 1\,.$$

Example: Assume that a country with currently 100 million people has a population growth rate (birth rate minus death rate) of 1% per year, and it also receives 100 thousand immigrants per year (which are quickly assimilated and reproduce at the same rate as the native population). Find its population in 10 years from now. (Assume that all the immigrants arrive in a single batch at the end of the year.)

Answer: If we call $x_n =$ population in year n from now, we have:

$$x_n = 1.01\, x_{n-1} + 100,000 \quad (n > 0); \qquad x_0 = 100,000,000\,.$$

This is the equation above with $r = 1.01$, $c = 100,000$ and $A = 100,000,00$, hence:

$$\begin{aligned} x_n &= 100,000,000 \cdot 1.01^n + 100,000\, \frac{1.01^n - 1}{1.01 - 1} \\ &= 100,000,000 \cdot 1.01^n + 1000\,(1.01^n - 1)\,. \end{aligned}$$

So:

$$x_{10} = 110,462,317\,.$$

The second particular case is for $r = 1$ and $c_n = c + d\,n$, where c and d are constant (so c_n is an arithmetic sequence):

$$x_n = x_{n-1} + c + d\,n \quad (n > 0)\,; \qquad x_0 = A\,.$$

The solution is now

$$x_n = A + \sum_{k=1}^{n} (c + d\,k) = A + c\,n + \frac{d\,n\,(n+1)}{2}\,.$$

5.1.2. Second Order Recurrence Relations. Now we look at the recurrence relation

$$C_0\, x_n + C_1\, x_{n-1} + C_2\, x_{n-2} = 0\,.$$

First we will look for solutions of the form $x_n = c\,r^n$. By plugging in the equation we get:

$$C_0\, c\, r^n + C_1\, c\, r^{n-1} + C_2\, c\, r^{n-2} = 0\,,$$

hence r must be a solution of the following equation, called the *characteristic equation* of the recurrence:

$$C_0\, r^2 + C_1\, r + C_2 = 0\,.$$

Let r_1, r_2 be the two (in general complex) roots of the above equation. They are called *characteristic roots*. We distinguish three cases:

1. *Distinct Real Roots*. In this case the general solution of the recurrence relation is

   $$x_n = c_1\, r_1^n + c_2\, r_2^n\,,$$

   where c_1, c_2 are arbitrary constants.

2. *Double Real Root.* If $r_1 = r_2 = r$, the general solution of the recurrence relation is
$$x_n = c_1\, r^n + c_2\, n\, r^n\,,$$
where c_1, c_2 are arbitrary constants.

3. *Complex Roots.* In this case the solution could be expressed in the same way as in the case of distinct real roots, but in order to avoid the use of complex numbers we write $r_1 = r\, e^{\alpha i}$, $r_2 = r\, e^{-\alpha i}$, $k_1 = c_1 + c_2$, $k_2 = (c_1 - c_2)\, i$, which yields:[1]
$$x_n = k_1\, r^n\, \cos n\alpha + k_2\, r^n\, \sin n\alpha\,.$$

Example: Find a closed-form formula for the Fibonacci sequence defined by:
$$F_{n+1} = F_n + F_{n-1} \quad (n > 0)\,; \qquad F_0 = 0,\; F_1 = 1\,.$$
Answer: The recurrence relation can be written
$$F_n - F_{n-1} - F_{n-2} = 0\,.$$
The characteristic equation is
$$r^2 - r - 1 = 0\,.$$
Its roots are:[2]
$$r_1 = \phi = \frac{1+\sqrt{5}}{2}; \qquad r_2 = -\phi^{-1} = \frac{1-\sqrt{5}}{2}\,.$$
They are distinct real roots, so the general solution for the recurrence is:
$$F_n = c_1\, \phi^n + c_2\, (-\phi^{-1})^n\,.$$
Using the initial conditions we get the value of the constants:
$$\left\{\begin{array}{lll} (n=0) & c_1 + c_2 & = \; 0 \\ (n=1) & c_1\, \phi + c_2\, (-\phi^{-1}) & = \; 1 \end{array}\right. \Rightarrow \left\{\begin{array}{lcr} c_1 & = & 1/\sqrt{5} \\ c_2 & = & -1/\sqrt{5} \end{array}\right.$$
Hence:
$$F_n = \frac{1}{\sqrt{5}}\, \left\{\phi^n - (-\phi)^{-n}\right\}\,.$$

[1]Reminder: $e^{\alpha i} = \cos\alpha + i \sin\alpha$.

[2]$\phi = \frac{1+\sqrt{5}}{2}$ is the *Golden Ratio.*

CHAPTER 6

Graph Theory

6.1. Graphs

6.1.1. Graphs. Consider the following examples:

1. A road map, consisting of a number of towns connected with roads.

2. The representation of a binary relation defined on a given set. The relation of a given element x to another element y is represented with an arrow connecting x to y.

The former is an example of (undirected) *graph.* The latter is an example of a *directed graph* or *digraph.*

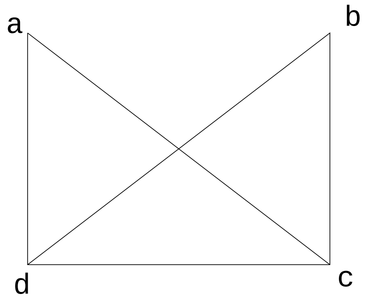


FIGURE 6.1. Undirected Graph.

In general a *graph* G consists of two things:

1. The *vertex set* V, whose elements are called *vertices*, *nodes* or *points*.

2. The *edge set* E or set of *edges* connecting pairs of vertices. If the edges are directed then they are also called *directed edges* or *arcs*. Each edge $e \in E$ is associated with a pair of vertices.

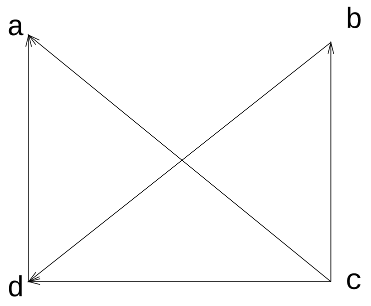


FIGURE 6.2. Directed Graph.

A graph is sometimes represented by the pair (V, E) (we assume V and E finite).

If the graph is undirected and there is a unique edge e connecting x and y we may write $e = \{x, y\}$, so E can be regarded as set of unordered pairs. In this context we may also write $e = (x, y)$, understanding that here (x, y) is not an ordered pair, but the name of an edge.

If the graph is directed and there is a unique edge e pointing from x to y, then we may write $e = (x, y)$, so E may be regarded as a set of ordered pairs. If $e = (x, y)$, the vertex x is called *origin*, *source* or *initial point* of the edge e, and y is called the *terminus*, *terminating vertex* or *terminal point*.

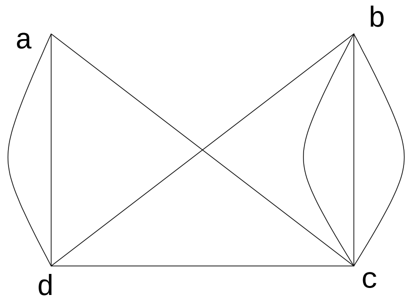


FIGURE 6.3. Graph with parallel edges.

Two vertices connected by an edge are called *adjacent*. They are also the *endpoints* of the edge, and the edge is said to be *incident* to each of its endpoints. If the graph is directed, an edge pointing from vertex x to vertex y is said to be *incident from x* and *incident to y*. An edge connecting a vertex to itself is called a *loop*. Two edges connecting the same pair of points are called *parallel*. A graph with neither loops nor parallel edges is called a *simple graph*.

The *degree* of a vertex v, represented $\delta(v)$, is the number of edges that contain it (loops are counted twice). A vertex of degree zero (not connected to any other vertex) is called *isolated*. A vertex of degree 1 is called *pendant*.

A *path* is a sequence of vertices (v_k) and edges (e_k) of the form $v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n$, where each edge e_k connects v_{k-1} with v_k (and points from v_{k-1} to v_k if the graph is directed).

A *weighted graph* is a graph whose edges have been labeled with numbers. The *length* of a path in a weighted graph is the sum of the weights of the edges in the path.

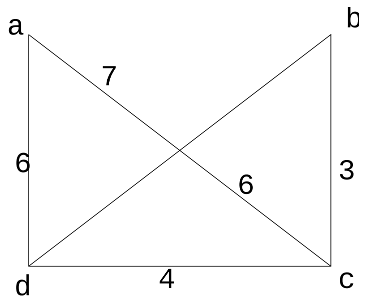


FIGURE 6.4. Weighted Graph.

6.1.2. Special Graphs. Here we examine a few special graphs.

The n-cube: A graph with with 2^n vertices labeled $0, 1, \dots, 2^n - 1$ so that two of them are connected with an edge if their binary representation differs in exactly one bit.

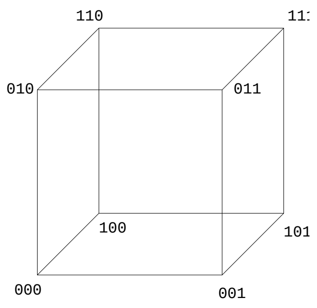


FIGURE 6.5. 3-cube.

Complete Graph: a simple undirected graph G such that every pair of distinct vertices in G are connected by an edge. The *complete graph* of n vertices is represented K_n (fig. 6.6). A *complete directed graph* is a simple directed graph $G = (V, E)$ such that every pair of distinct vertices in G are connected by exactly one edge—so, for each pair of distinct vertices, either (x, y) or (y, x) (but not both) is in E.

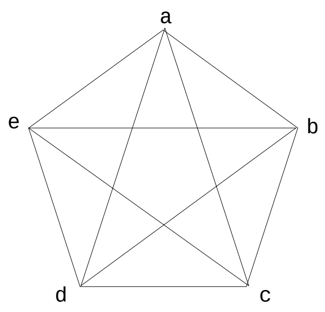


FIGURE 6.6. Complete graph K_5.

Bipartite Graph: a graph $G = (V, E)$ in which V can be partitioned into two subsets V_1 and V_2 so that each edge in G connects some vertex in V_1 to some vertex in V_2. A bipartite simple graph is called *complete* if each vertex in V_1 is connected to each vertex in V_2. If $|V_1| = m$ and $|V_2| = n$, the corresponding complete bipartite graph is represented $K_{m,n}$ (fig. 6.7).

A graph is bipartite iff its vertices can be colored with two colors so that every edge connects vertices of different color.

Question: Is the n-cube bipartite. Hint: color in red all vertices whose binary representation has an even number of 1's, color in blue the ones with an odd number of 1's.

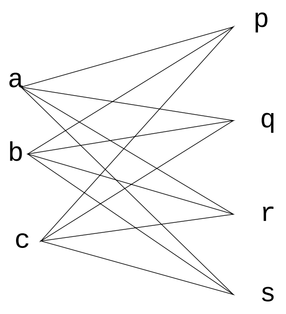


FIGURE 6.7. Complete bipartite graph $K_{3,4}$.

Regular Graph: a simple graph whose vertices have all the same degree. For instance, the n-cube is regular.

6.1.3. Subgraph. Given a graph $G = (V, E)$, a *subgraph* $G' = (V', E')$ of G is another graph such that $V' \subseteq V$ and $E' \subseteq E$. If $V' = V$ then G' is called a *spanning subgraph* of G.

Given a subset of vertices $U \subseteq V$, the subgraph of G *induced* by U, denoted $\langle U \rangle$, is the graph whose vertex set is U, and its edge set contains all edges from G connecting vertices in U.

6.2. Paths and Cycles

6.2.1. Paths. A *path* from v_0 to v_n of length n is a sequence of $n+1$ vertices (v_k) and n edges (e_k) of the form $v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n$, where each edge e_k connects v_{k-1} with v_k. If there are no parallel edges we only need to specify the vertices: $v_0, v_1, v_2, \dots, v_n$.

A *simple path* from v to w is a path from v to w with no repeated vertices. A *cycle* (or *circuit*) is a path of non-zero length from v to v with no repeated edges. A *simple cycle* is a *cycle* with no repeated vertices (except for the beginning and ending vertex).

Remark: If a graph contains a cycle from v to v, then it contains a simple cycle from v to v. Proof: if a given vertex v_i occurs twice in the cycle, we can remove the part of it that goes from v_i and back to v_i. If the resulting cycle still contains repeated vertices we can repeat the operation until there are no more repeated vertices.

6.2.2. Connected Graphs. A graph G is called *connected* if there is a path between any two distinct vertices of G. Otherwise the graph is called *disconnected*. A directed graph is connected if its associated undirected graph (obtained by ignoring the directions of the edges) is connected. A *connected component* of G is any connected subgraph $G' = (V', E')$ of $G = (V, E)$ such that there is not edge (in G) from a vertex in V to a vertex in $V - V'$. Given a vertex in G, the component of G containing v is the subgraph G' of G consisting of all edges and vertices of g contained in some path beginning at v.

6.2.3. The Seven Bridges of Königsberg. This is a classical problem that started the discipline today called *graph theory*.

During the eighteenth century the city of Königsberg (in East Prussia) was divided into four sections, including the island of Kneiphop, by the Pregel river. Seven bridges connected the regions, as shown in figure 6.8. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to the starting point. The first person to solve the problem (in the negative) was the Swiss mathematician Leonhard Euler in 1736. He represented the sections of the city and the seven bridges by the graph of figure 6.9, and proved that it is impossible to find a path in it that transverses every edge of the graph exactly once. In the next section we study why this is so.

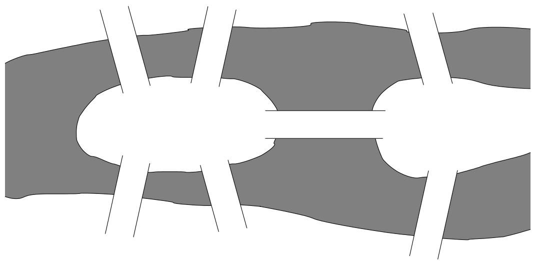


FIGURE 6.8. The Seven Bridges of Königsberg.

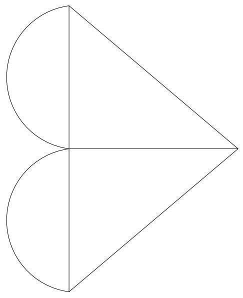


FIGURE 6.9. Graph for the Seven Bridges of Königsberg.

6.2.4. Euler paths and cycles. Let $G = (V, E)$ be a graph with no isolated vertices. An *Euler path* in G is a path that transverses every edge of the graph exactly once. Analogously, an *Euler cycle* in G is a cycle that transverses every edge of the graph exactly once.

The graphs that have an Euler path can be characterized by looking at the degree of their vertices. Recall that the degree of a vertex v, represented $\delta(v)$, is the number of edges that contain v (loops are counted twice). An *even* vertex is a vertex with even degree; an *odd* vertex is a vertex with odd degree. The sum of the degrees of all vertices in a graph equals twice its number of edges, so it is an even number. As a consequence, the number of odd vertices in a graph is always even.

Then G contains an Euler cycle if and only if G is connected and all its vertices have even degree. Also, G contains an Euler path from vertex a to vertex b ($\neq a$) if and only if G is connected, a and b have odd degree, and all its other vertices have even degree.

6.2.5. Hamiltonian Cycles. A *Hamiltonian cycle* in a graph G is a cycle that contains each vertex of G once (except for the starting and ending vertex, which occurs twice). A *Hamiltonian path* in G is a path (not a cycle) that contains each vertex of G once. Note that by deleting an edge in a Hamiltonian cycle we get a Hamilton path, so if a graph has a Hamiltonian cycle, then it also has a Hamiltonian path. The converse is not true, i.e., a graph may have a Hamiltonian path but not a Hamiltonian cycle. *Exercise*: Find a graph with a Hamiltonian path but no Hamiltonian cycle.

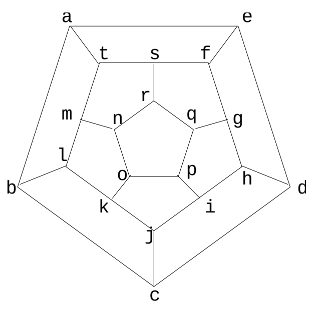


FIGURE 6.10. Hamilton's puzzle.

In general it is not easy to determine if a given graph has a Hamiltonian path or cycle, although often it is possible to argue that a graph has no Hamiltonian cycle. For instance if $G = (V, E)$ is a bipartite graph with vertex partition $\{V_1, V_2\}$ (so that each edge in G connects some vertex in V_1 to some vertex in V_2), then G cannot have a Hamiltonian cycle if $|V_1| \neq |V_2|$, because any path must contain alternatively vertices from V_1 and V_2, so any cycle in G must have the same number of vertices from each of both sets.

Edge removal argument. Another kind of argument consists of removing edges trying to make the degree of every vertex equal two. For instance in the graph of figure 6.11 we cannot remove any edge because that would make the degree of b, e or d less than 2, so it is impossible to reduce the degree of a and c. Consequently that graph has no Hamiltonian cycle.

The Traveling Salesperson Problem. Given a weighted graph, the *traveling salesperson problem* (TSP) consists of finding a Hamiltonian cycle of minimum length in this graph. The name comes from a classical problem in which a salesperson must visit a number of cities and go back home traveling the minimum distance possible. One way to solve the problem consists of searching all possible Hamiltonian cycles and computing their length, but this is very inefficient. Unfortunately no

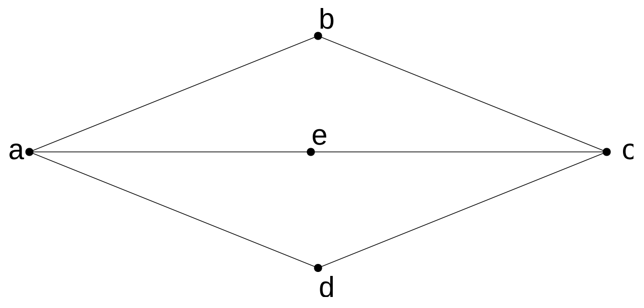


FIGURE 6.11. Graph without Hamiltonian cycle.

efficient algorithm is known for solving this problem (and chances are that none exists).

Remark: (Digression on P/NP problems.) Given a weighted graph with n vertices the problem of determining whether it contains a Hamiltonian cycle of length not greater than a given L is known to be NP-complete. This means the following. First it is a *decision* problem, i.e., a problem whose solution is "yes" or "no". A decision problem is said to be polynomial, or belong to the class P, if it can be solved with an algorithm of complexity $O(n^k)$ for some integer k. It is said to be non-deterministic polynomial, or belong to the class NP, if in all cases when the answer is "yes" this can be determined with a non-deterministic algorithm of complexity $O(n^k)$. A non-deterministic algorithm is an algorithm that works with an extra hint, for instance in the TSP, if G has a Hamiltonian cycle of length not greater than L the hint could consist of a Hamiltonian cycle with length not greater than L—so the task of the algorithm would be just to check that in fact that length is not greater than L.[1] Currently it is not known whether the class NP is strictly larger than the class P, although it is strongly suspected that it is. The class NP contains a subclass called NP-complete containing the "hardest" problems of the class, so that their complexity must be higher than polynomial unless P=NP. The TSP is one of these problems.

Gray Codes. A *Gray code* is a sequence $s_1, s_2, \dots, s_{2^n}$ of n-binary strings verifying the following conditions:

[1]Informally, P problems are "easy to solve", and NP problems are problems whose answer is "easy to check". In a sense the P=NP problem consist of determining whether every problem whose solution is easy to check is also easy to solve.

1. Every n-binary string appears somewhere in the sequence.
2. Two consecutive strings s_i and s_{i+1} differ exactly in one bit.
3. s_{2^n} and s_1 differ in exactly one bit.

For instance: $000, 001, 011, 010, 110, 111, 101, 100,$

The problem of finding a gray code is equivalent to finding a Hamiltonian cycle in the n-cube.

6.2.6. Dijkstra's Shortest-Path Algorithm. This is an algorithm to find the shortest path from a vertex a to another vertex z in a connected weighted graph. Edge (i, j) has weight $w(i, j) > 0$, and vertex x is labeled $L(x)$ (minimum distance from a if known, otherwise ∞). The output is $L(z)$ = length of a minimum path from a to z.

```
procedure dijkstra(w,a,z,L)
  L(a) := 0
  for all vertices x ≠ a do
    L(x) := ∞
  T := set of all vertices
  // T is the set of all vertices whose shortest
  // distance from a has not been found yet
  while z in T do
    begin
      choose v in T with minimum L(v)
      T := T - {v}
      for each x in T adjacent to v do
        L(x) := min{L(x),L(v)+w(v,x)}
    end
  return(L(z))
end dijkstra
```

For instance consider the graph in figure 6.12.

The algorithm would label the vertices in the following way in each iteration (the boxed vertices are the ones removed from T):

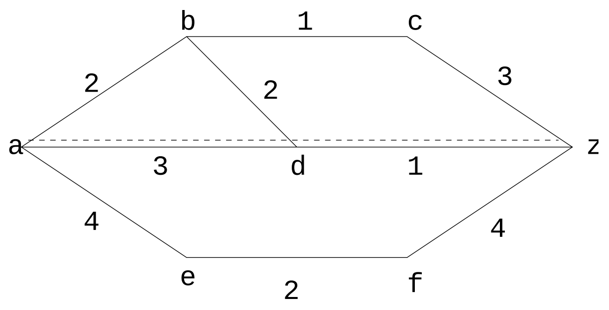


FIGURE 6.12. Shortest path from a to z.

iteration	a	b	c	d	e	f	z
0	0	∞	∞	∞	∞	∞	∞
1	$\boxed{0}$	2	∞	3	4	∞	∞
2	$\boxed{0}$	$\boxed{2}$	3	3	4	∞	∞
3	$\boxed{0}$	$\boxed{2}$	$\boxed{3}$	3	4	∞	6
4	$\boxed{0}$	$\boxed{2}$	$\boxed{3}$	$\boxed{3}$	4	∞	4
5	$\boxed{0}$	$\boxed{2}$	$\boxed{3}$	$\boxed{3}$	$\boxed{4}$	6	4
6	$\boxed{0}$	$\boxed{2}$	$\boxed{3}$	$\boxed{3}$	$\boxed{4}$	6	$\boxed{4}$

At this point the algorithm returns the value 4.

Complexity of Dijkstra's algorithm. For an n-vertex, simple, connected weighted graph, Dijkstra's algorithm has a worst-case run time of $\Theta(n^2)$.

6.3. Representations of Graphs

6.3.1. Adjacency matrix. The *adjacency matrix* of a graph is a matrix with rows and columns labeled by the vertices and such that its entry in row i, column j, $i \neq j$, is the number of edges incident on i and j. If $i = j$ then the entry is twice the number of loops incident on i. For instance the following is the adjacency matrix of the graph of figure 6.13:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix} \end{array}$$

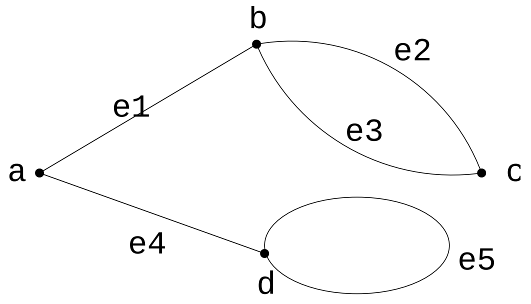


FIGURE 6.13

One of the uses of the adjacency matrix A of a simple graph G is to compute the number of paths between two vertices, namely entry (i, j) of A^n is the number of paths of length n from i to j.

6.3.2. Incidence matrix. The incidence matrix of a graph G is a matrix with rows labeled by vertices and columns labeled by edges, so that entry for row v column e is 1 if e is incident on v, and 0 otherwise. As an example, the following is the incidence matrix of graph of figure 6.13:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

6.3.3. Graph Isomorphism. Two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, are called *isomorphic* if there is a bijection $f : V_1 \to V_2$ and a bijection $g : E_1 \to E_2$ such that an edge e is adjacent to vertices v and w if and only if $g(e)$ is adjacent to $f(v)$ and $f(w)$ (fig. 6.14).

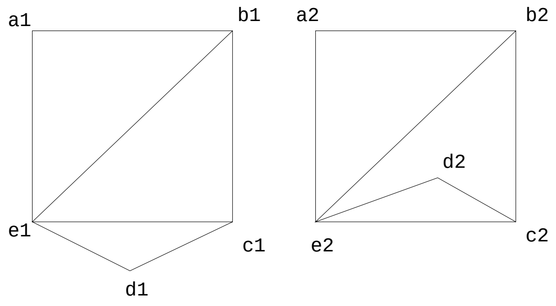


FIGURE 6.14. Two isomorphic graphs.

Two graphs are isomorphic if and only if for some ordering of their vertices their adjacency matrices are equal.

An *invariant* is a property such that if a graph has it then all graphs isomorphic to it also have it. Examples of invariants are their number of vertices, their number of edges, "has a vertex of degree k", "has a simple cycle of length l", etc. It is possible to prove that two graphs are not isomorphic by showing an invariant property that one has and the other one does not have. For instance the graphs in figure 6.15 cannot be isomorphic because one has a vertex of degree 2 and the other one doesn't.

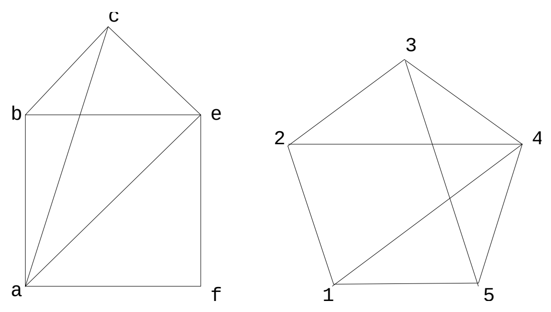


FIGURE 6.15. Non isomorphic graphs.

6.4. Planar Graphs

6.4.1. Planar Graphs. A graph G is *planar* if it can be drawn in the plane with its edges intersecting at their vertices only. One such drawing is called an *embedding* of the graph in the plane.

A particular planar representation of a planar graph is called a *map*. A map divides the plane into a number of regions or faces (one of them infinite).

6.4.2. Graph Homeomorphism. If a graph G has a vertex v of degree 2 and edges (v, v_1), (v, v_2) with $v_1 \neq v_2$, we say that the edges (v, v_1) and (v, v_2) are in *series*. Deleting such vertex v and replacing (v, v_1) and (v, v_2) with (v_1, v_2) is called a *series reduction*. For instance, in the third graph of figure 6.16, the edges (h, b) and (h, d) are in series. By removing vertex h we get the first graph in the left.

Two graphs are said to be *homeomorphic* if they are isomorphic or can be reduced to isomorphic graphs by a sequence of series reductions (fig. 6.16).

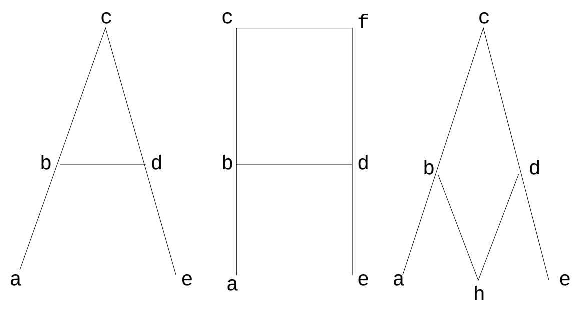


FIGURE 6.16. Three homeomorphic graphs.

Note that if a graph G is planar, then all graphs homeomorphic to G are also planar.

6.4.3. Some Results About Planar Graphs.

1. *Euler's Formula*: Let $G = (V, E)$ be a connected planar graph, and let $v = |V|$, $e = |E|$, and $f =$ number of faces (regions) in which some given embedding of G divides the plane. Then:

$$v - e + f = 2 .$$

Note that this implies that all plane embeddings of a given graph define the same number of faces.

2. Let $G = (V, E)$ be a simple connected planar graph with v vertices, $e \geq 3$ edges and f faces. Then $3f \leq 2e$ and $e \leq 3v - 6$.
3. The graph K_5 is non-planar. Proof: in K_5 we have $v = 5$ and $e = 10$, hence $3v - 6 = 9 < e = 10$, which contradicts the previous result.
4. The graph $K_{3,3}$ is non-planar. Proof: in $K_{3,3}$ we have $v = 6$ and $e = 9$. If $K_{3,3}$ were planar, from Euler's formula we would have $f = 5$. On the other hand, each face is bounded by at least four edges, so $4f \leq 2e$, i.e., $20 \leq 18$, which is a contradiction.
5. *Kuratowski's Theorem*: A graph is non-planar if and only if it contains a subgraph that is homeomorphic to either K_5 or $K_{3,3}$.

6.4.4. Dual Graph of a Map. A map is defined by some planar graph $G = (V, E)$ embedded in the plane. Assume that the map divides the plane into a set of regions $R = \{r_1, r_2, \dots, r_k\}$. For each region r_i, select a point p_i in the interior of r_i. The *dual graph* of that map is the graph $G^d = (V^d, E^d)$, where $V^d = \{p_1, p_2, \dots, p_k\}$, and for each edge in E separating the regions r_i and r_j, there is an edge in E^d connecting p_i and p_j. *Warning*: Note that a different embedding of the same graph G may give different (and non-isomorphic) dual graphs. *Exercise*: Find the duals of the maps shown in figure 6.14, and prove that they are not isomorphic.

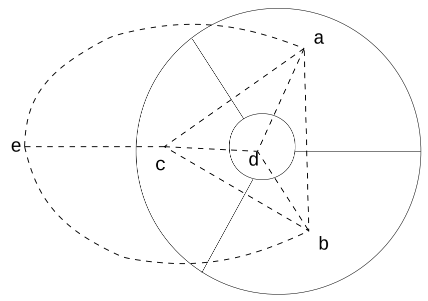


FIGURE 6.17. Dual graph of a map.

6.4.5. Graph Coloring. Consider the problem of coloring a map M in such a way that no adjacent regions (sharing a border) have the same color. This is equivalent to coloring the vertices of the dual graph of M in such a way that no adjacent vertices have the same color.

In general, a *coloring* of a graph is an assignment of a color to each vertex of the graph. The coloring is called *proper* if there are no adjacent vertices with the same color. If a graph can be properly colored with n colors we say that it is *n-colorable*. The minimum number of colors needed to properly color a given graph $G = (V, E)$ is called the *chromatic number* of G, and is represented $\chi(G)$. Obviously $\chi(G) \leq |V|$.

6.4.6. Some Results About Graph Coloring.

1. $\chi(K_n) = n$.
2. Let G be a simple graph. The following statements are equivalent:
   (a) $\chi(G) = 2$.
   (b) G is bipartite.
   (c) Every cycle in G has even length
3. *Five Color Theorem (Kempe, Heawood)* (not hard to prove): Every simple, planar graph is 5-colorable.
4. *Four Color Theorem (Appel and Haken, 1976)*, proved with an intricate computer analysis of configurations: Every simple, planar graph is 4-colorable.

Exercise: Find a planar graph G such that $\chi(G) = 4$.

CHAPTER 7

Trees

7.1. Trees

7.1.1. Terminology. Let T be a graph with n vertices. The following properties are equivalent:

1. T is connected and acyclic (has no cycles).
2. T is connected and has $n-1$ edges.
3. T is acyclic and has $n-1$ edges.
4. If v and w are vertices in T, there is a unique simple path from v to w.

A graph having any of the above equivalent properties is called a *free tree* or simply *tree*. A union of trees, or equivalently a simple graph with no cycles, is called *forest*.

A *rooted tree* is a tree in which a particular vertex is designated as the root.

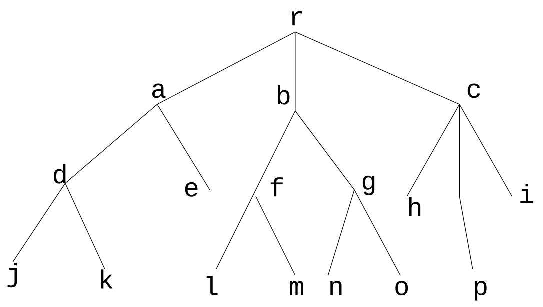


FIGURE 7.1. A rooted tree.

The *level* of a vertex v is the length of the simple path from the root to v. The *height* of a rooted tree is the maximum level of its vertices.

Let T be a tree with root v_0. Suppose that x, y and z are vertices in T and that $(v_0, v_1, \ldots, v_n)$ is a simple path in T. Then:

1. v_{n-1} is the *parent* of v_n.
2. $v_0, v_1, \dots, v_{n-1}$ are *ancestors* of v_n.
3. v_n is a *child* of v_{n-1}.
4. If x is an ancestor of y, y is a *descendant* of x.
5. If x and y are children of z, x and y are *siblings.*
6. If x has no children, it is called a *terminal vertex* or *leaf.*
7. If x is not a terminal vertex, it is an *internal* or *branch vertex.*
8. The *subtree of T rooted at x* is the graph (V, E), where V is x together with its descendants and E = edges of simple paths from x to some vertex in E.

7.1.2. Huffman Codes. Usually characters are represented in a computer with fix length bit strings. *Huffman codes* provide an alternative representation with variable length bit strings, so that shorter strings are used for the most frequently used characters. As an example assume that we have an alphabet with four symbols: $A = \{a, b, c, d\}$. Two bits are enough for representing them, for instance $a = 11$, $b = 10$, $c = 01$, $d = 00$ would be one such representation. With this encoding n-character words will have $2n$ bits. However assume that they do not appear with the same frequency, instead some are more frequent that others, say a appears with a frequency of 50%, b 30%, c 15% and d 5%. Then the following enconding would be more efficient than the fix length encoding: $a = 1$, $b = 01$, $c = 001$, $d = 000$. Now in average an n-character word will have $0.5n$ a's, $0.3n$ b's, $0.15n$ c's and $0.05n$ d's, hence its length will be $0.5n{\cdot}1+0.3n{\cdot}2+0.15n{\cdot}3+0.05n{\cdot}3 = 1.7n$, which is shorter than $2n$. In general the length per character of a given encoding with characters $a_1, a_2, \dots, a_n$ whose frequencies are $f_1, f_2, \dots, f_n$ is

$$\frac{1}{F}\sum_{k=1}^{n} f_k\, l(a_k)\,,$$

where $l(a_k)$ = length of a_k and $F = \sum_{k=1}^{n} f_k$. The problem now is, given an alphabet and the frequencies of its characters, find an optimal encoding that provides minimum average length for words.

Fix length and Huffman codes can be represented by trees like in figure 7.2. The code of each symbol consists of the sequence of labels of the edges in the path from the root to the leaf with the desired symbol.

7.1.3. Constructing an Optimal Huffman Code. An optimal Huffman code is a Huffman code in which the average length of the symbols is minimum. In general an optimal Huffman code can be made

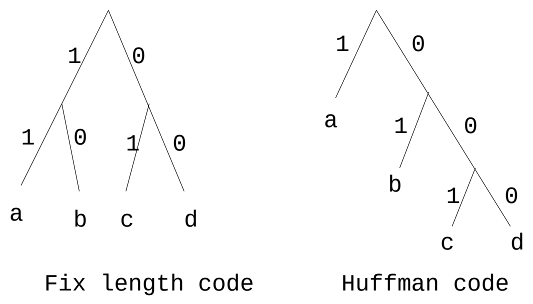


FIGURE 7.2. Fix length code and Huffman code.

as follows. First we list the frequencies of all the codes and represent the symbols as vertices (which at the end will be leaves of a tree). Then we replace the two smallest frequencies f_1 and f_2 with their sum $f_1 + f_2$, and join the corresponding two symbols to a common vertex above them by two edges, one labeled 0 and the other one labeled 1. Than common vertex plays the role of a new symbol with a frequency equal to $f_1 + f_2$. Then we repeat the same operation with the resulting shorter list of frequencies until the list is reduced to one element and the graph obtained becomes a tree.

Example: Find the optimal Huffman code for the following table of symbols:

character	frequency
a	2
b	3
c	7
d	8
e	12

Answer: : The successive reductions of the list of frequencies are as follows:

$$\underbrace{2,3}_{5},7,8,12 \rightarrow \underbrace{5,7}_{12},8,12 \rightarrow 12,8,12$$

Here we have a choice, we can choose to add the first 12 and 8, or 8 and the second 12. Let's choose the former:

$$\underbrace{12,8}_{20},12 \rightarrow \underbrace{20,12}_{32} \rightarrow 32$$

The tree obtained is the following:

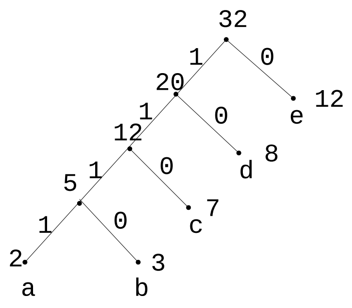


FIGURE 7.3. Optimal Huffman code 1.

The resulting code is as follows:

character	code
a	1111
b	1110
c	110
d	10
e	0

The other choice yields the following:

$$12, \underbrace{8, 12}_{20} \to \underbrace{20, 12}_{32} \to 32$$

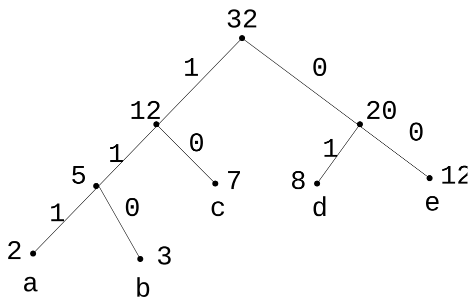


FIGURE 7.4. Optimal Huffman code 2.

character	code
a	111
b	110
c	10
d	01
e	00

7.2. Spanning Trees

7.2.1. Spanning Trees. A tree T is a *spanning tree* of a graph G if T is a subgraph of G that contains all the vertices of G. For instance the graph of figure 7.5 has a spanning tree represented by the thicker edges.

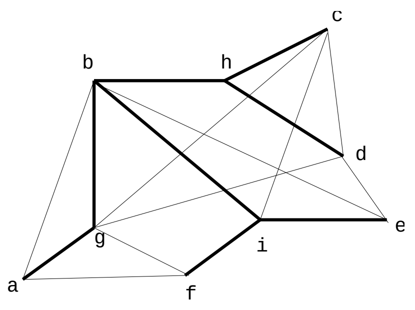


FIGURE 7.5. Spanning tree.

Every connected graph has a spanning tree which can be obtained by removing edges until the resulting graph becomes acyclic. In practice, however, removing edges is not efficient because finding cycles is time consuming.

Next, we give two algorithms to find the spanning tree T of a loop-free connected undirected graph $G = (V, E)$. We assume that the vertices of G are given in a certain order $v_1, v_2, \ldots, v_n$. The resulting spanning tree will be $T = (V', E')$.

7.2.2. Breadth-First Search Algorithm. The idea is to start with vertex v_1 as root, add the vertices that are adjacent to v_1, then the ones that are adjacent to the latter and have not been visited yet, and so on. This algorithm uses a queue (initially empty) to store vertices of the graph. In consists of the following:

1. Add v_1 to T, insert it in the queue and mark it as "visited".
2. If the queue is empty, then we are done. Otherwise let v be the vertex in the front of the queue.
3. For each vertex v' of G that has not been visited yet and is adjacent to v (there might be none) taken in order of increasing subscripts, add vertex v' and edge (v, v') to T, insert v' in the queue and mark it as "visited".
4. Delete v from the queue.

5. Go to step 2.

A pseudocode version of the algorithm is as follows:

```
procedure bfs(V,E)
  S := (v1) // ordered list of vertices of a fix level
  V' := {v1} // v1 is the root of the spanning tree
  E' := {} // no edges in the spanning tree yet
  while true do
    begin
      for each x in S, in order, do
        for each y in V - V' do
          if (x,y) is an edge then
            add edge (x,y) to E' and vertex y to V'
        if no edges were added then
          return(T)
      S := children of S
    end
end bfs
```

Figure 7.6 shows the spanning tree obtained using the breadth-first search algorithm on the graph with its vertices ordered lexicographically: $a, b, c, d, e, f, g, h, i$.

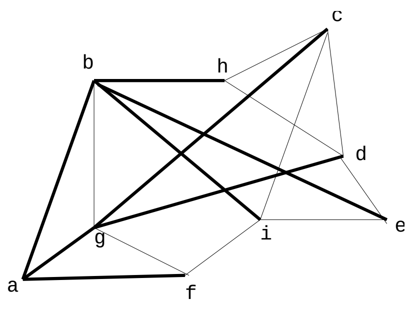


FIGURE 7.6. Breadth-First Search.

7.2.3. Depth-First Search Algorithm. The idea of this algorithm is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.

This algorithm uses a stack (initially empty) to store vertices of the graph. In consists of the following:

1. Add v_1 to T, insert it in the stack and mark it as "visited".

2. If the stack is empty, then we are done. Otherwise let v be the vertex on the top of the stack.
3. If there is no vertex v' that is adjacent to v and has not been visited yet, then delete v and go to step 2 (*backtrack*). Otherwise, let v' be the first non-visited vertex that is adjacent to v.
4. Add vertex v' and edge (v, v') to T, insert v' in the stack and mark it as "visited".
5. Go to step 2.

An alternative recursive definition is as follows. We define recursively a process P applied to a given vertex v in the following way:

1. Add vertex v to T and mark it as "visited".
2. If there is no vertex v' that is adjacent to v and has not been visited yet, then return. Otherwise, let v' be the first non-visited vertex that is adjacent to v.
3. Add the edge (v, v') to T.
4. Apply P to v'.
5. Go to step 2 (*backtrack*).

The *Depth-First Search Algorithm* consists of applying the process just defined to v_1.

A pseudocode version of the algorithm is as follows:

```
procedure dfs(V,E)
  V' := {v1} // v1 is the root of the spanning tree
  E' := {} // no edges in the spanning tree yet
  w := v1
  while true do
    begin
      while there is an edge (w,v) that when added
              to T does not create a cycle in T do
        begin
          Choose first v such that (w,v)
          does not create a cycle in T
          add (w,v) to E'
          add v to V'
          w := v
        end
      if w = v1 then
```

```
17:         return(T)
18:       w := parent of w in T // backtrack
19:     end
20: end
```

Figure 7.7 shows the spanning tree obtained using the breadth-first search algorithm on the graph with its vertices ordered lexicographically: $a, b, c, d, e, f, g, h, i$.

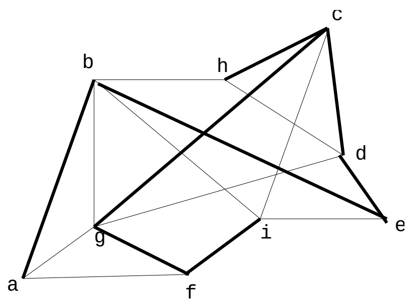


FIGURE 7.7. Depth-First Search.

7.2.4. Minimal Spanning Trees. Given a connected weighted tree G, its *minimal spanning tree* is a spanning tree of G such that the sum of the weights of its edges is minimum. For instance for the graph of figure 7.8, the spanning tree shown is the one of minimum weight.

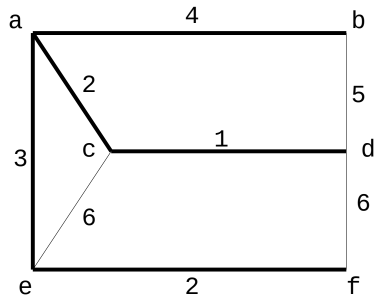


FIGURE 7.8. Minimum Spanning Tree.

Prim's Algorithm. An algorithm to find a minimal spanning tree is *Prim's Algorithm.* It starts with a single vertex and at each iteration adds to the current tree a minimum weight edge that does not complete a cycle.

The following is a pseudocode version of Prim's algorithm. If (x, y) is an edge in $G = (V, E)$ then $w(x, y)$ is its weight, otherwise $w(x, y) = \infty$. The starting vertex is s.

```
procedure prim(V,w,s)
  V' := {s} // vertex set starts with s
  E' = {} // edge set initially empty
  for i := 1 to n-1 do // put n edges in spanning tree
    begin
      find x in V' and y in V - V' with minimum w(x,y)
      add y to V'
      add (x,y) to E'
    end
  return(E')
end prim
```

Prim's algorithm is an example of a *greedy algorithm.* A greedy algorithm is an algorithm that optimized the choice at each iteration without regard to previous choices ("doing the best locally"). Prim's algorithm makes a minimum spanning tree, but in general a greedy algorithm does not always finds an optimal solution to a given problem. For instance in figure 7.9 a greedy algorithm to find the shortest path from a to z, working by adding the shortest available edge to the most recently added vertex, would return acz, which is not the shortest path.

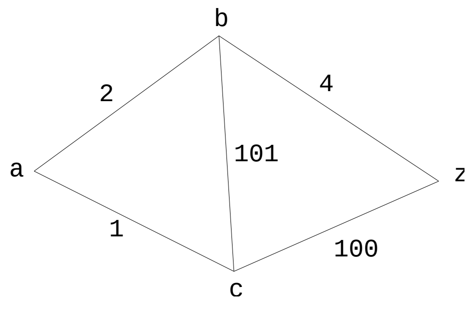


FIGURE 7.9

Kruskal's Algorithm. Another algorithm to find a minimal spanning tree in a connected weighted tree $G = (V, E)$ is *Kruskal's Algorithm.* It starts with all n vertices of G and no edges. At each iteration we add an edge having minimum weight that does not complete a cycle. We stop after adding $n - 1$ edges.

```
procedure kruskal(E,w,n)
  V' := V
  E' := {}
  while |E'| < n-1 do
    begin
      among all edges not completing a cycle in T
      choose e of minimum weight and add it to E
    end
  T' = (V',E')
  return(T')
end kruskal
```

7.3. Binary Trees

7.3.1. Binary Trees. A *binary tree* is a rooted tree in which each vertex has at most two children, designated as *left child* and *right child*. If a vertex has one child, that child is designated as either a left child or a right child, but not both. A *full binary tree* is a binary tree in which each vertex has exactly two children or none. The following are a few results about binary trees:

1. If T is a full binary tree with i internal vertices, then T has $i+1$ terminal vertices and $2i+1$ total vertices.
2. If a binary tree of height h has t terminal vertices, then $t \leq 2^h$.

7.3.2. Binary Search Trees. Assume S is a set in which elements (which we will call "data") are ordered; e.g., the elements of S can be numbers in their natural order, or strings of alphabetic characters in lexicographic order. A *binary search tree* associated to S is a binary tree T in which data from S are associate with the vertices of T so that, for each vertex v in T, each data item in the left subtree of v is less than the data item in v, and each data item in the right subtree of v is greater than the data item in v.

Example: Figure 7.10 contains a binary search tree for the set $S = \{1,2,3,4,5,6,7,8,9,10\}$. In order to find a element we start at the root and compare it to the data in the current vertex (initially the root). If the element is greater we continue through the right child, if it is smaller we continue through the left child, if it is equal we have found it. If we reach a terminal vertex without founding the element, then that element is not present in S.

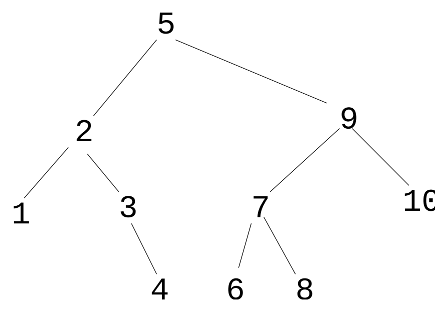


FIGURE 7.10. Binary Search Tree.

7.3.3. Making a Binary Search Tree. We can store data in a binary search tree by randomly choosing data from S and placing it in the tree in the following way: The first data chosen will be the root of the tree. Then for each subsequent data item, starting at the root we compare it to the data in the current vertex v. If the new data item is greater than the data in the current vertex then we move to the right child, if it is less we move to the left child. If there is no such child then we create one and put the new data in it. For instance, the tree in figure 7.11 has been made from the following list of words choosing them in the order they occur: "IN A PLACE OF LA MANCHA WHOSE NAME I DO NOT WANT TO REMEMBER".

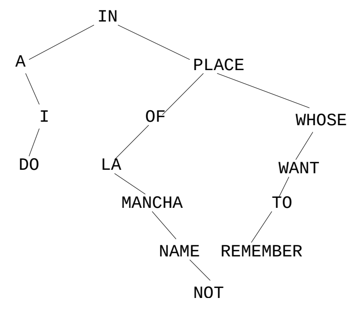


FIGURE 7.11. Another binary Search Tree.

7.3.4. Tree Transversals. In order to motivate this subject, we introduce the concept of Polish notation. Given a (not necessarily commutative) binary operation $\circ$, it is customary to represent the result of applying the operation to two elements a, b by placing the operation symbol in the middle:

$$a \circ b\,.$$

This is called *infix* notation. The *Polish* notation consists of placing the symbol to the left:

$$\circ\, a\, b\,.$$

The *reverse Polish* notation consists of placing the symbol to the right:

$$a\, b\, \circ\ .$$

The advantage of Polish notation is that it allows us to write expressions without need for parenthesis. For instance, the expression $a*(b+c)$ in Polish notation would be $*\,a+b\,c$, while $a*b+c$ is $+*a\,b\,c$. Also, Polish notation is easier to evaluate in a computer.

In order to evaluate an expression in Polish notation, we scan the expression from right to left, placing the elements in a stack.[1] Each time we find an operator, we replace the two top symbols of the stack by the result of applying the operator to those elements. For instance, the expression $* + 2\,3\,4$ (which in infix notation is "$(2+3)*4$") would be evaluated like this:

expression	stack
$* + 2\,3\,4$	
$* + 2\,3$	4
$* + 2$	3 4
$* +$	2 3 4
$*$	5 4
	20

An algebraic expression can be represented by a binary rooted tree obtained recursively in the following way. The tree for a constant or variable a has a as its only vertex. If the algebraic expression S is of the form $S_L \circ S_R$, where S_L and S_R are subexpressions with trees T_L and T_R respectively, and $\circ$ is an operator, then the tree T for S consists of $\circ$ as root, and the subtrees T_L and T_R (fig. 7.12).

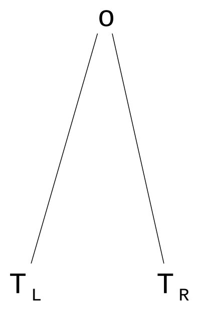


FIGURE 7.12. Tree of $S_1 \circ S_2$.

For instance, consider the following algebraic expression:

$$a + b * c + d \uparrow e * (f + h)\,,$$

where $+$ denotes addition, $*$ denotes multiplication and $\uparrow$ denotes exponentiation. The binary tree for this expression is given in figure 7.13.

[1]A *stack* or *last-in first-out* (LIFO) system, is a linear list of elements in which insertions and deletions take place only at one end, called *top* of the list. A *queue* or *first-in first-out* (FIFO) system, is a linear list of elements in which deletions take place only at one end, called *front* of the list, and insertions take place only at the other end, called *rear* of the list.

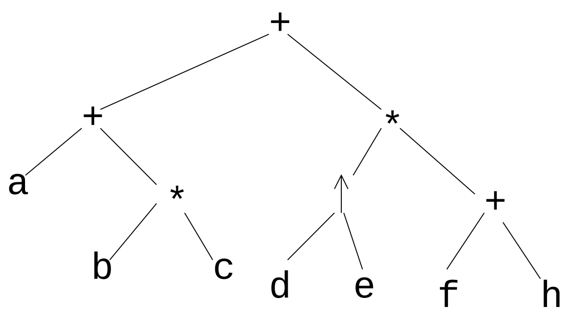


FIGURE 7.13. Tree for $a + b * c + d \uparrow e * (f + h)$.

Given the binary tree of an algebraic expression, its Polish, reverse Polish and infix representation are different ways of ordering the vertices of the tree, namely in *preorder*, *postorder* and *inorder* respectively.

The following are recursive definitions of several orderings of the vertices of a rooted tree $T = (V, E)$ with root r. If T has only one vertex r, then r by itself constitutes the *preorder*, *postorder* and *inorder* transversal of T. Otherwise, let $T_1, \dots, T_k$ the subtrees of T from left to right (fig. 7.14). Then:

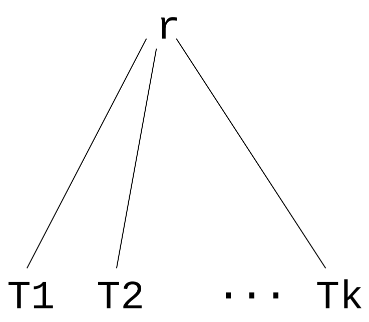


FIGURE 7.14. Ordering of trees.

1. *Preorder Transversal*: $\mathrm{Pre}(T) = r, \mathrm{Pre}(T_1), \dots, \mathrm{Pre}(T_k)$.
2. *Postorder Transversal*: $\mathrm{Post}(T) = \mathrm{Post}(T_1), \dots, \mathrm{Post}(T_k), r$.
3. *Inorder Transversal.* If T is a binary tree with root r, left subtree T_L and right subtree T_R, then: $\mathrm{In}(T) = \mathrm{In}(T_L), r, \mathrm{In}(T_R)$.

7.4. Decision Trees, Tree Isomorphisms

7.4.1. Decision Trees. A *decision tree* is a tree in which each vertex represents a question and each descending edge from that vertex represents a possible answer to that question.

Example: The Five-Coins Puzzle. In this puzzle we have five coins C_1, C_2, C_3, C_4, C_5 that are identical in appearance, but one is either heavier or lighter that the others. The problem is to identify the bad coin and determine whether it is lighter or heavier using only a pan balance and comparing the weights of two piles of coins. The problem can be solved in the following way. First we compare the weights of C_1 and C_2. If C_1 is heavier than C_2 then we know that either C_1 is the bad coin and is heavier, or C_2 is the bad coin and it is lighter. Then by comparing say C_1 with any of the other coins, say C_5, we can determine whether the bad coin is C_1 and is heavier (if C_1 it is heavier than C_5) or it is C_2 and is lighter (if C_1 has the same weight as C_5). If C_1 is lighter than C_2 we proceed as before with "heavier" and "lighter" reversed. If C_1 and C_2 have the same weight we can try comparing C_3 and C_4 in a similar manner. If their weights are the same then we know that the bad coin is C_5, and we can determine whether it is heavier or lighter by comparing it to say C_1. The corresponding decision tree is the following:

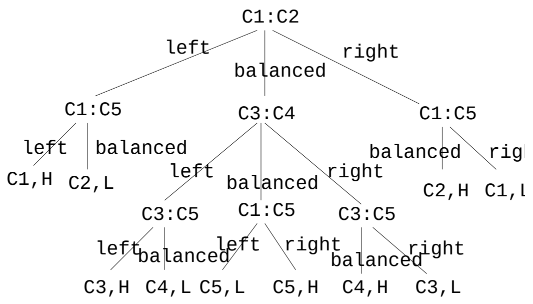


FIGURE 7.15. Decision tree for the 5 coins puzzle.

In each vertex "$C_i : C_j$" means that we compare coins C_i and C_j by placing C_i on the left pan and C_j on the right pan of the balance, and each edge is labeled depending on what side of the balance is heavier. The terminal vertices are labeled with the bad coin and whether it is heavier (H) or lighter (L). The decision tree is optimal in the sense that in the worst case it uses three weighings, and there is no way to solve the problem with less than that—with two weighings we can get at most nine possible outcomes, which are insufficient to distinguish among ten combinations of 5 possible bad coins and the bad coin being heavier or lighter.

7.4.2. Complexity of Sorting. Sorting algorithms work by comparing elements and rearranging them as needed. For instance we can sort three elements a_1, a_2, a_3 with the decision tree shown in figure 7.16

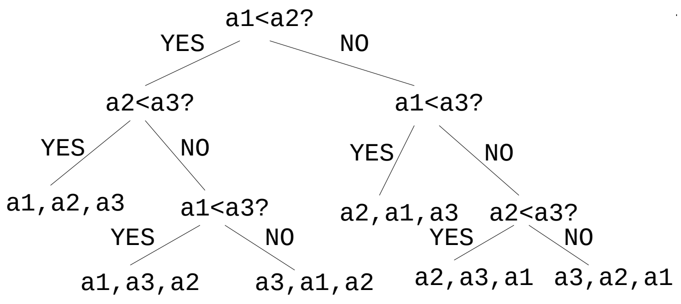


FIGURE 7.16. Sorting three elements.

Since there are $3! = 6$ possible arrangements of 3 elements, we need a decision tree with at least 6 possible outcomes or terminal vertices. Recall that in a binary tree of height h with t terminal vertices the following inequality holds: $t \leq 2^h$. Hence in our case $6 < 2^h$, which implies $h \geq 3$, so the algorithm represented by the decision tree in figure 7.16 is optimal in the sense that it uses the minimum possible number of comparisons in the worst-case.

More generally in order to sort n elements we need a decision tree with $n!$ outcomes, so its height $h(n)$ will verify $n! \leq 2^{h(n)}$. Since $\log_2(n!) = \Theta(n \log_2 n)$,[1] we have $h(n) = \Omega(n \log_2 n)$. So the worse case complexity of a sorting algorithm is $\Omega(n \log_2 n)$. Since the merge-sort algorithm uses precisely $\Theta(n \log_2 n)$ comparisons, we know that it is optimal.

7.4.3. Isomorphisms of Trees. Assume that T_1 is a tree with vertex set V_1 and T_2 is another tree with vertex set V_2. If they are rooted trees then we call their roots r_1 and r_2 respectively. We will study three different kinds of tree-isomorphisms between T_1 and T_2.

[1]According to Stirling's formula, $n! \approx n^n e^{-n} \sqrt{2\pi n}$, so taking logarithms $\log_2 n! \approx n \log_2 n - n \log_2 e + \frac{1}{2} \log_2 (2\pi n) = \Theta(n \log_2 n)$.

1. Usual graph-isomorphism between trees: T_1 and T_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ that preserves adjacency, i.e., $f(v)$ is adjacent to $f(w)$ if and only if v is adjacent to w.

2. Rooted-tree-isomorphism: T_1 and T_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ that preserves adjacency and the root vertex, i.e.:
   (a) $f(v)$ is adjacent to $f(w)$ if and only if v is adjacent to w.
   (b) $f(r_1) = r_2$.

3. Binary-tree-isomorphism: Two binary trees T_1 and T_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ that preserves adjacency, and the root vertex, and left/right children, i.e.:
   (a) $f(v)$ is adjacent to $f(w)$ if and only if v is adjacent to w.
   (b) $f(r_1) = r_2$.
   (c) $f(v)$ is a left child of $f(w)$ if and only if v is a left child of w.
   (d) $f(v)$ is a right child of $f(w)$ if and only if v is a right child of w.

Example: Figure 7.17 shows three trees which are graph-isomorphic. On the other hand as rooted trees T_2 and T_3 are isomorphic, but they are not isomorphic to T_1 because the root of T_1 has degree 3, while the roots of T_2 and T_3 have degree 2. Finally T_2 and T_3 are not isomorphic as binary trees because the left child of the root in T_2 is a terminal vertex while the left child of the root of T_3 has two children.

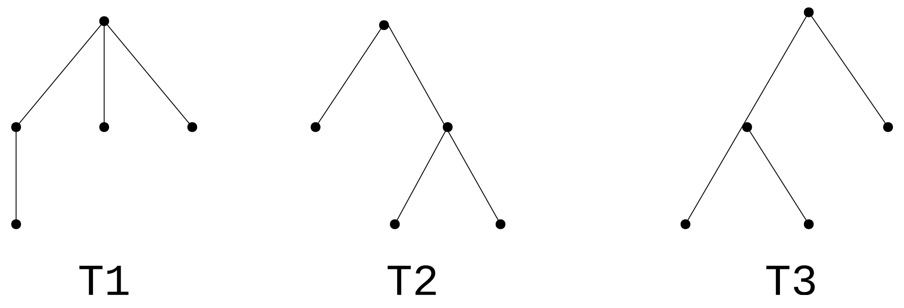


FIGURE 7.17. Trees with different kinds of isomorphisms.

Exercise: Find all non-isomorphic 3-vertex free trees, 3-vertex rooted trees and 3-vertex binary trees. *Answer*: Figure 7.18 shows all 5 non-isomorphic 3-vertex binary trees. As rooted trees T_2–T_5 are isomorphic, but T_1 is not isomorphic to the others, so there are 2 non-isomorphic 3-vertex rooted trees represented for instance by T_1 and T_2. All of them are isomorphic as free trees, so there is only 1 non-isomorphic 3-vertex free tree.

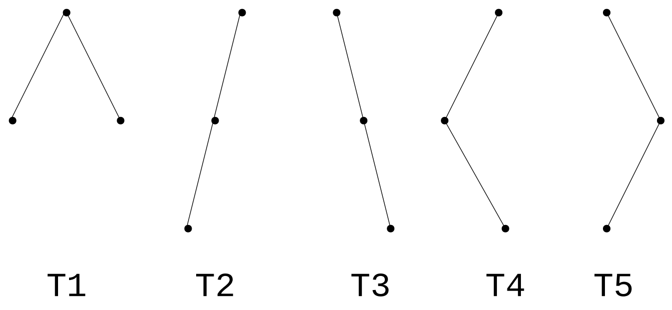


FIGURE 7.18. Non-isomorphic binary trees.

7.4.4. Game Trees. Trees are used in the analysis of some games. As an example we study the following game using a tree: Initially there are two piles with 3 coins and 1 coin respectively. Taking turns two players remove any number of coins from one of the piles. The player that removes the last coin loses. The following tree represents all possible sequences of choices. Each node shows the number of coins in each pile, and each edge represents a possible "move" (choice) from one of the players. The first player is represented with a box and the second player is represented with an circle.

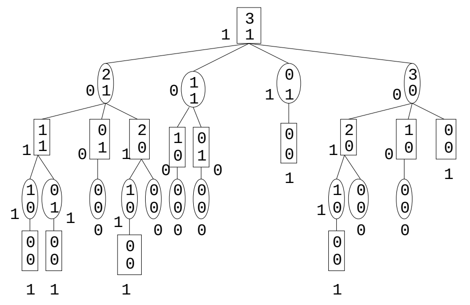


FIGURE 7.19. Tree of a game.

The analysis of the game starts by labeling each terminal vertex with "1" if it represents a victory for the first player and "0" if it represents a victory for the second player. This numbers represent the "value" of each position of the game, so that the first player is interested in making it "maximum" and the second player wants to make it "minimum". Then we continue labeling the rest of the vertices in the following way. After all the children of a given vertex have been labeled, we label the vertex depending on whether it is a "first player" position (box) or a "second player" position (circle). First player positions are labeled with the maximum value of the labels of its children, second player positions are labeled with the minimum value of the labels of its children. This process is called the *minimax procedure*. Every vertex labeled "1" will represent a position in which the first player has advantage and can win if he/she works without making mistakes; on the other hand, vertices labeled "0" represent positions for which the second player has advantage. Now the strategy is for the first player to select at each position a children with maximum value, while the second player will be interested in selecting children with minimum value. If the starting position has been labeled "1" that means that the first player has a winning strategy, otherwise the second player has advantage. For instance in the present game the first player has advantage at the initial position, and only one favorable movement at that point: $\binom{3}{1} \to \binom{0}{1}$, i.e., he/she must remove all 3 coins from the first pile. If for any reason the first player makes a mistake and removes say one coin from the first pile, going to position $\binom{2}{1}$, then the second player has one favorable move to vertex $\binom{0}{1}$, which is the one with minimum "value".

Alpha-beta pruning. In some games the game tree is so complicated that it cannot be fully analyzed, so it is built up to a given depth only. The vertices reached at that depth are not terminal, but they can be "evaluated" using heuristic methods (for instance in chess usually losing a knight is a better choice than losing the queen, so a position with one queen and no knights will have a higher value than one with no queen and one knight). Even so the evaluation and labeling of the vertices can be time consuming, but we can bypass the evaluation of many vertices using the technique of *alpha-beta pruning*. The idea is to skip a vertex as soon as it becomes obvious that its value will not affect the value of its parent. In order to do that with a first player (boxed) vertex v, we assign it an *alpha value* equal to the maximum value of its children evaluated so far. Assume that we are evaluating one of its children w, which will be a second player (circled) position. If at any point a children of w gets a value less than or equal to the alpha value of v then it will become obvious that the value of w is going to be less than the current alpha value of v, so it will not affect the value of v and we can stop the process of evaluation of w (prone the subtree at w). That is called an *alpha cutoff*. Similarly, at a second player (circled) vertex v, we assign a *beta value* equal to the minimum value of its children evaluated so far, and practice a *beta cutoff* when one of its grandchildren gets a value greater than or equal to the current beta value of v, i.e., we prone the subtree at w, where w is the parent of that grandchildren.

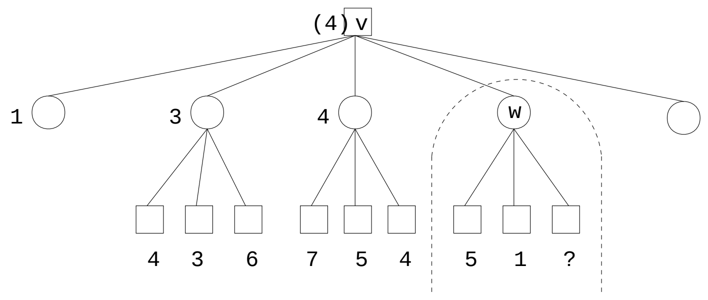


FIGURE 7.20. Alpha cutoff.

CHAPTER 8

Boolean Algebras

8.1. Combinatorial Circuits

8.1.1. Introduction. At their lowest level digital computers handle only binary signals, represented with the symbols 0 and 1. The most elementary circuits that combine those signals are called *gates.* Figure 8.1 shows three gates: OR, AND and NOT.

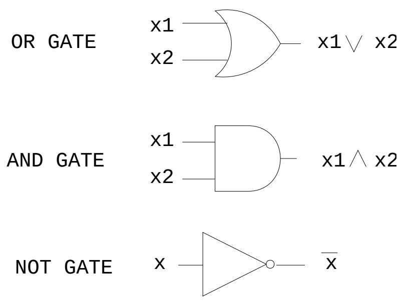


FIGURE 8.1. Gates.

Their outputs can be expressed as a function of their inputs by the following *logic tables*:

x_1	x_2	$x_1 \vee x_2$
1	1	1
1	0	1
0	1	1
0	0	0
OR GATE		

x_1	x_2	$x_1 \wedge x_2$
1	1	1
1	0	0
0	1	0
0	0	0
AND GATE		

x	$\bar{x}$
1	0
0	1
NOT GATE	

These are examples of *combinatorial circuits*. A combinatorial circuit is a circuit whose output is uniquely defined by its inputs. They do not have memory, previous inputs do not affect their outputs. Some combinations of gates can be used to make more complicated combinatorial circuits. For instance figure 8.2 is combinatorial circuit with the logic table shown below, representing the values of the *Boolean expression* $y = \overline{(x_1 \vee x_2) \wedge x_3}$.

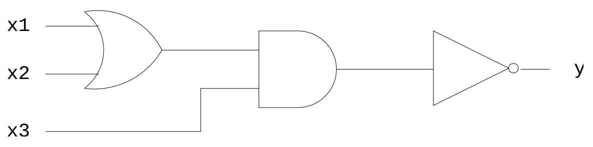


FIGURE 8.2. A combinatorial circuit.

x_1	x_2	x_3	$y = \overline{(x_1 \vee x_2) \wedge x_3}$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	1

However the circuit in figure 8.3 is *not* a combinatorial circuit. If $x_1 = 1$ and $x_2 = 0$ then y can be 0 or 1. Assume that at a given time $y = 0$. If we input a signal $x_2 = 1$, the output becomes $y = 1$, and stays so even after x_2 goes back to its original value 0. That way we can store a bit. We can "delete" it by switching input x_1 to 0.

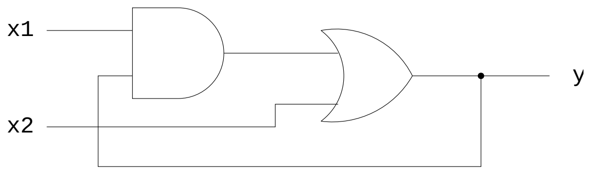


FIGURE 8.3. Not a combinatorial circuit.

8.1.2. Properties of Combinatorial Circuits. Here $\mathbb{Z}_2 = \{0, 1\}$ represents the set of signals handled by combinatorial circuits, and the operations performed on those signals by AND, OR and NOT gates are represented by the symbols $\wedge$, $\vee$ and $^-$ respectively. Then their properties are the following (a, b, c are elements of $\mathbb{Z}_2$, i.e., each represents either 0 or 1):

1. Associative
$$(a \vee b) \vee c = a \vee (b \vee c)$$
$$(a \wedge b) \vee c = a \wedge (b \wedge c)$$
2. Commutative
$$a \vee b = b \vee a$$
$$a \wedge b = b \wedge a$$
3. Distributive
$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$
$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$
4. Identity
$$a \vee 0 = a$$
$$a \wedge 1 = a$$
5. Complement
$$a \vee \bar{a} = 1$$
$$a \wedge \bar{a} = 0$$

A system satisfying those properties is called a *Boolean algebra*.

Two Boolean expressions are defined to be *equal* is they have the same values for all possible assignments of values to their literals. *Example*: $\overline{x \vee y} = \bar{x} \wedge \bar{y}$, as shown in the following table:

x	y	$\overline{x \vee y}$	$\bar{x} \wedge \bar{y}$
1	1	0	0
1	0	0	0
0	1	0	0
0	0	1	1

8.1.3. Boolean Algebras. Here we deal with general Boolean algebras; combinatorial circuits are an example, but there are others.

A Boolean algebra $B = (S, +, \cdot, ', 0, 1)$ is a set S containing two distinguished elements 0 and 1, two binary operators $+$ and $\cdot$ on S, and a unary operator $'$ on S, satisfying the following properties (x, y, z are elements of S):

1. Associative
$$(x + y) + z = x + (y + z)$$
$$(x \cdot y) + z = x \cdot (y \cdot z)$$
2. Commutative
$$x + y = y + x$$
$$x \cdot y = y \cdot x$$
3. Distributive
$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$
$$x + (y \cdot z) = (x + y) \cdot (x + z)$$
4. Identity
$$x + 0 = x$$
$$x \cdot 1 = x$$
5. Complement
$$x + x' = 1$$
$$x \cdot x' = 0$$

Example: $(\mathbb{Z}_2, \vee, \wedge, ^-, 0, 1)$ is a Boolean algebra.

Example: If U is a universal set and $\mathcal{P}(U)$= the power set of S (collection of subsets of S) then $(\mathcal{P}(U), \cup, \cap, ^-, \emptyset, U)$. is a Boolean algebra.

8.1.4. Other Properties of Boolean Algebras.

The properties mentioned above define a Boolean algebra, but Boolean algebras also have other properties:

1. Idempotent
$$x + x = x$$
$$x \cdot x = x$$
2. Bound
$$x + 1 = 1$$
$$x \cdot 0 = 0$$
3. Absorption
$$x + xy = x$$
$$x \cdot (x + y) = x$$
4. Involution
$$(x')' = x$$
5. 0 and 1
$$0' = 1$$
$$1' = 0$$
6. De Morgan's
$$(x + y)' = x' \cdot y'$$
$$(x \cdot y)' = x' + y'$$

For instance the first idempotent law can be proved like this: $x = x + 0 = x + x \cdot x' = (x + x) \cdot (x + x') = (x + x) \cdot 1 = x + x$.

8.2. Boolean Functions, Applications

8.2.1. Introduction. A *Boolean function* is a function from $\mathbb{Z}_2^n$ to $\mathbb{Z}_2$. For instance, consider the *exclusive-or* function, defined by the following table:

x_1	x_2	$x_1 \veebar x_2$
1	1	0
1	0	1
0	1	1
0	0	0

The exclusive-or function can interpreted as a function $\mathbb{Z}_2^2 \to \mathbb{Z}_2$ that assigns $(1,1) \mapsto 0$, $(1,0) \mapsto 1$, $(0,1) \mapsto 1$, $(0,0) \mapsto 0$. It can also be written as a Boolean expression in the following way:

$$x_1 \veebar x_2 = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

Every Boolean function can be written as a Boolean expression as we are going to see next.

8.2.2. Disjunctive Normal Form. We start with a definition. A *minterm* in the symbols $x_1, x_2 \dots, x_n$ is a Boolean expression of the form $y_1 \wedge y_2 \wedge \cdots \wedge y_n$, where each y_i is either x_i or $\bar{x}_i$.

Given any Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ that is not identically zero, it can be represented

$$f(x_1, \dots, x_n) = m_1 \vee m_2 \vee \cdots \vee m_k\,,$$

where $m_1, m_2, \dots, m_k$ are all the minterms $m_i = y_1 \wedge y_2 \wedge \cdots \wedge y_n$ such that $f(a_1, a_2, \dots, a_n) = 1$, where $y_j = x_j$ if $a_j = 1$ and $y_j = \bar{x}_j$ if $a_j = 0$. That representation is called *disjunctive normal form* of the Boolean function f.

Example: We have seen that the exclusive-or can be represented $x_1 \veebar x_2 = (x_1 \wedge \bar{x_2}) \vee (\bar{x_1} \wedge x_2)$. This provides a way to implement the exclusive-or with a combinatorial circuit as shown in figure 8.4.

8.2.3. Conjunctive Normal Form. A *maxterm* in the symbols $x_1, x_2 \dots, x_n$ is a Boolean expression of the form $y_1 \vee y_2 \vee \cdots \vee y_n$, where each y_i is either x_i or $\bar{x}_i$.

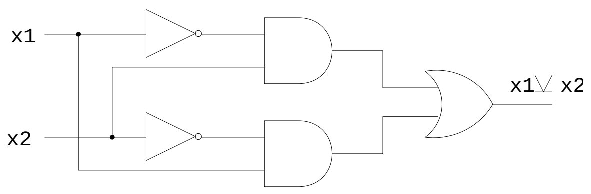


FIGURE 8.4. Exclusive-Or.

Given any Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ that is not identically one, it can be represented

$$f(x_1, \dots, x_n) = M_1 \wedge M_2 \wedge \cdots \wedge M_k \,,$$

where $M_1, M_2, \dots, M_k$ are all the maxterms $M_i = y_1 \vee y_2 \vee \cdots \vee y_n$ such that $f(a_1, a_2, \dots, a_n) = 0$, where $y_j = x_j$ if $a_j = 0$ and $y_j = \bar{x}_j$ if $a_j = 1$. That representation is called *conjunctive normal form* of the Boolean function f.

Example: The conjunctive normal form of the exclusive-or is

$$x_1 \veebar x_2 = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)\,.$$

8.2.4. Functionally Complete Sets of Gates. We have seen how to design combinatorial circuits using AND, OR and NOT gates. Here we will see how to do the same with other kinds of gates. In the following gates will be considered as functions from $\mathbb{Z}_2^n$ into $\mathbb{Z}_2$ intended to serve as building blocks of arbitrary boolean functions.

A set of gates $\{g_1, g_2, \dots, g_k\}$ is said to be *functionally complete* if for any integer n and any function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ it is possible to construct a combinatorial circuit that computes f using only the gates $g_1, g_2, \dots, g_k$. *Example*: The result about the existence of a disjunctive normal form for any Boolean function proves that the set of gates $\{\text{AND}, \text{OR}, \text{NOT}\}$ is functionally complete. Next we show other sets of gates that are also functionally complete.

1. The set of gates $\{\text{AND}, \text{NOT}\}$ is functionally complete. Proof: Since we already know that $\{\text{AND}, \text{OR}, \text{NOT}\}$ is functionally complete, all we need to do is to show that we can compute $x \vee y$ using only AND and NOT gates. In fact:

$$x \vee y = \overline{\bar{x} \wedge \bar{y}}\,,$$

hence the combinatorial circuit of figure 8.5 computes $x \vee y$.

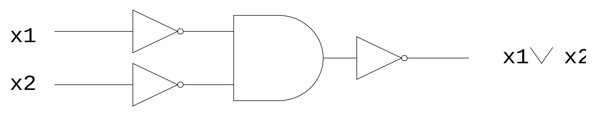


FIGURE 8.5. OR with AND and NOT.

2. The set of gates {OR, NOT} is functionally complete. The proof is similar:
$$x \wedge y = \overline{\bar{x} \vee \bar{y}}\,,$$
hence the combinatorial circuit of figure 8.6 computes $x \vee y$.

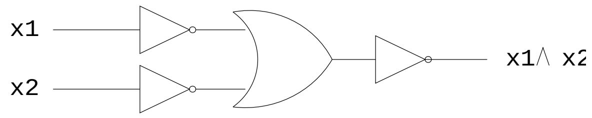


FIGURE 8.6. AND with OR and NOT.

3. The gate NAND, denoted $\uparrow$ and defined as
$$x_1 \uparrow x_2 = \begin{cases} 0 & \text{if } x_1 = 1 \text{ and } x_2 = 1 \\ 1 & \text{otherwise,} \end{cases}$$
is functionally complete.

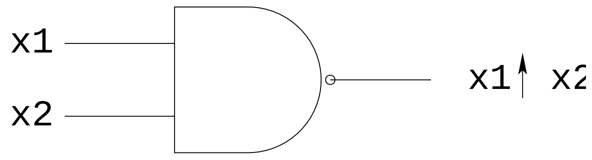


FIGURE 8.7. NAND gate.

Proof: Note that $x \uparrow y = \overline{x \wedge y}$. Hence $\bar{x} = \overline{x \wedge x} = x \uparrow x$, so the NOT gate can be implemented with a NAND gate. Also the OR gate can be implemented with NAND gates: $x \vee y = \overline{\bar{x} \wedge \bar{y}} = (x \uparrow x) \uparrow (y \uparrow y)$. Since the set {OR, NOT} is functionally complete and each of its elements can be implemented with NAND gates, the NAND gate is functionally complete.

8.2.5. Minimization of Combinatorial Circuits. Here we address the problems of finding a combinatorial circuit that computes a given Boolean function with the minimum number of gates. The idea is to simplify the corresponding Boolean expression by using algebraic

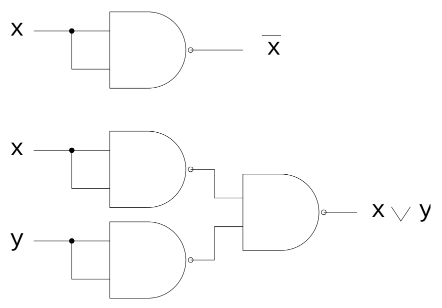


FIGURE 8.8. NOT and OR functions implemented with NAND gates.

properties such as $(E \wedge a) \vee (E \wedge \bar{a}) = E$ and $E \vee (E \wedge a) = E$, where E is any Boolean expression. For simplicity in the following we will represent $a \wedge b$ as ab, so for instance the expressions above will look like this: $Ea \vee E\bar{a} = E$ and $E \vee Ea = E$.

Example: Let $F(x, y, z)$ the Boolean function defined by the following table:

x	y	z	f(x,y,z)
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	0

Its disjunctive normal form is $f(x, y, z) = xyz \vee xy\bar{z} \vee x\bar{y}\bar{z}$. This function can be implemented with the combinatorial circuit of figure 8.9.

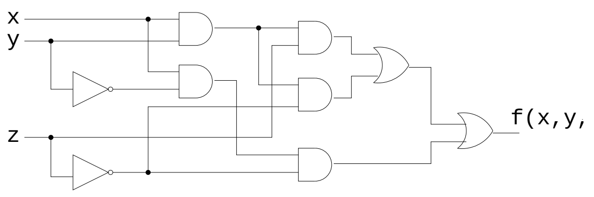


FIGURE 8.9. A circuit that computes $f(x, y, z) = xyz \vee xy\bar{z} \vee x\bar{y}\bar{z}$.

But we can do better if we simplify the expression in the following way:

$$\begin{aligned} f(x,y,z) &= \overbrace{xyz \vee xy\bar{z}}^{xy} \vee x\bar{y}\bar{z} \\ &= xy \vee x\bar{y}\bar{z} \\ &= x(y \vee \bar{y}\bar{z}) \\ &= x(y \vee \bar{y})(y \vee \bar{z}) \\ &= x(y \vee \bar{z})\,, \end{aligned}$$

which corresponds to the circuit of figure 8.10.

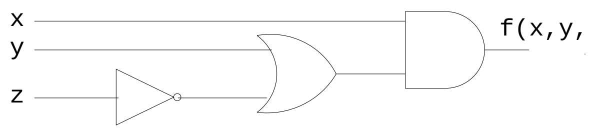


FIGURE 8.10. A simpler circuit that computes $f(x,y,z) = xyz \vee xy\bar{z} \vee x\bar{y}\bar{z}$.

8.2.6. Multi-Output Combinatorial Circuits. *Example*: *Half-Adder*. A half-adder is a combinatorial circuit with two inputs x and y and two outputs s and c, where s represents the sum of x and y and c is the carry bit. Its table is as follows:

x	y	s	c
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

So the sum is $s = x \veebar y$ (exclusive-or) and the carry bit is $c = x \wedge y$. Figure 8.11 shows a half-adder circuit.

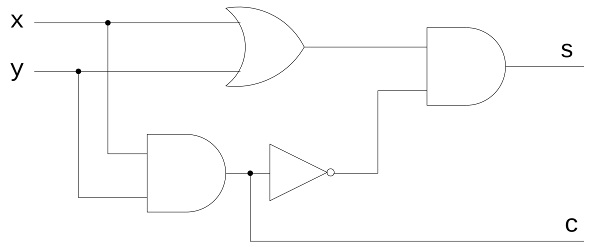


FIGURE 8.11. Half-adder circuit.

CHAPTER 9

Automata, Grammars and Languages

9.1. Finite State Machines

9.1.1. Finite-State Machines. Combinatorial circuits have no memory or internal states, their output depends only on the current values of their inputs. Finite state machines on the other hand have internal states, so their output may depend not only on its current inputs but also on the past history of those inputs.

A *finite-state machine* consists of the following:

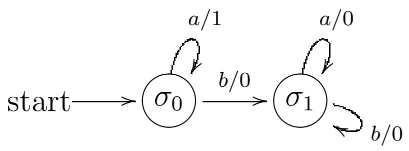
1. A finite set of *input symbols* $\mathcal{I}$.
2. A finite set of *output symbols* $\mathcal{O}$.
3. A finite set of *states* $\mathcal{S}$.
4. A *next-state function* $f : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$.
5. An *output function* $g : \mathcal{S} \times \mathcal{I} \to \mathcal{O}$.
6. An *initial state* $\sigma \in \mathcal{S}$.

We represent the machine $M = (\mathcal{I}, \mathcal{O}, \mathcal{S}, f, g, \sigma)$

Example: We describe a finite state machine with two input symbols $\mathcal{I} = \{a, b\}$ and two output symbols $\mathcal{O} = \{0, 1\}$ that accepts any string from $\mathcal{I}^*$ and outputs as many 1's as a's there are at the beginning of the string, then it outputs only 0's. The internal states are $\mathcal{S} = \{\sigma_0, \sigma_1\}$, where σ_0 is the initial state—we interpret it as not having seeing any "b" yet; then the machine will switch to σ_1 as soon as the first "b" arrives. The next-state and output functions are as follows:

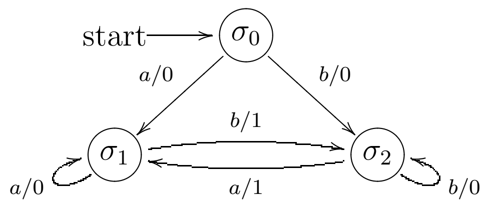
	f		g	
$\mathcal{I}$ / $\mathcal{S}$	a	b	a	b
σ_0	σ_0	σ_1	1	0
σ_1	σ_1	σ_1	0	0

This finite-state machine also can be represented with the following *transition diagram*:

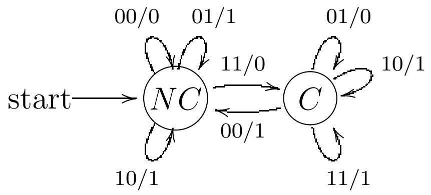


The vertices of the diagram are the states. If in state σ an input i causes the machine to output o and go to state σ' then we draw an arrow from σ to σ' labeled i/o.

Example: The following example is similar to the previous one but the machine outputs 1 only after a change of input symbol, otherwise it outputs 0:



Example: A Serial-Adder. A serial adder accepts two bits and outputs its sum. So the input set is $\mathcal{I} = \{00, 01, 10, 11\}$. The output set is $\mathcal{O} = \{0, 1\}$. The set of states is $\mathcal{S} = \{NC, C\}$, which stands for "no carry" and "carry" respectively. The transition diagram is the following:



9.1.2. Finite-State Automata. A *finite-state automaton* is a finite-state machine with only two output symbols: $\mathcal{O} = \{0, 1\}$. Those states for which the last output is 1 are called *accepting states.*

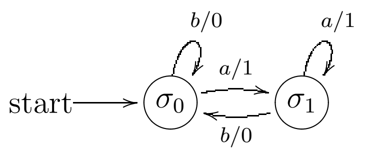
An alternate definition is as follows. A finite-state automaton consists of:

1. A finite set of *input symbols* $\mathcal{I}$.

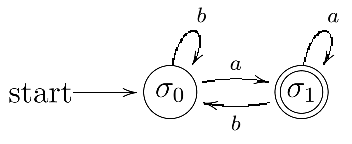
2. A finite set of *states* $\mathcal{S}$.
3. A *next-state function* $f : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$.
4. A subset $\mathcal{A}$ of $\mathcal{S}$ of *accepting states.*
5. An *initial state* $\sigma \in \mathcal{S}$.

We represent the automaton $A = (\mathcal{I}, \mathcal{S}, f, \mathcal{A}, \sigma)$. We say that an automaton *accepts* a given string of input symbols if that strings takes the automaton from the starting state to an accepting state.

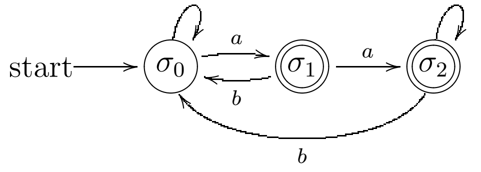
Example: The following transition diagrams represent an automaton accepting any string of a's and b's ending with an a. The first diagram uses the same scheme as with finite-state machines:



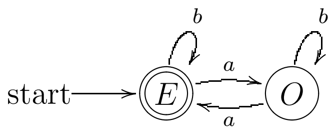
The second kind of diagram omits the outputs and represents the accepting states with double circles:



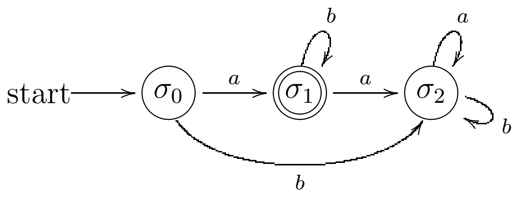
Two finite-state automata that accept exactly the same set of strings are said to be *equivalent.* For instance the following automaton also accepts precisely strings of a's abd b's that end with an a, so it is equivalent to the automaton shown above:



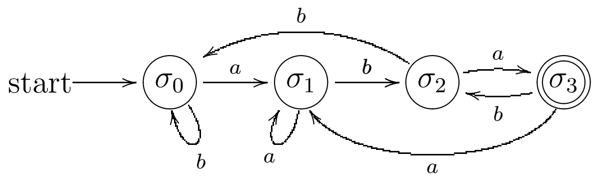
Example: The following automaton accepts strings of a's and b's with exactly an even number of a's:



Example: The following automaton accepts strings starting with one a followed by any number of b's:



Example: The following automaton accepts strings ending with aba:



9.2. Languages and Grammars

9.2.1. Formal Languages. Consider algebraic expressions written with the symbols $A = \{x, y, z, +, *, (,)\}$. The following are some of them: "$x + y * y$", "$y + (x * y + y) * x$", "$(x + y) * x + z$", etc. There are however some strings of symbols that are not legitimate algebraic expressions, because they have some sort of syntax error, e.g.: "$(x + y$", "$z + +y * x$", "$x(*y) + z$", etc. So syntactically correct algebraic expressions are a subset of the whole set A^* of possible strings over A.

In general, given a finite set A (the *alphabet*), a *(formal) language* over A is a subset of A^* (set of strings of A).

Although in principle any subset of A^* is a formal language, we are interested only in languages with certain structure. For instance: let $A = \{a, b\}$. The set of strings over A with an even number of a's is a language over A.

9.2.2. Grammars. A way to determine the structure of a language is with a *grammar*. In order to define a grammar we need two kinds of symbols: *non-terminal*, used to represent given subsets of the language, and *terminal*, the final symbols that occur in the strings of the language. For instance in the example about algebraic expressions mentioned above, the final symbols are the elements of the set $A = \{x, y, z, +, *, (,)\}$. The non-terminal symbols can be chosen to represent a complete algebraic expression (E), or terms (T) consisting of product of factors (F). Then we can say that an algebraic expression E consists of a single term

$E \to T$,

or the sum of an algebraic expression and a term

$E \to E + T$.

A term may consists of a factor or a product of a term and a factor

$T \to F$

$T \to T * F$

A factor may consists of an algebraic expression between parenthesis

$F \to (E)$,

or an isolated terminal symbol

$F \to x$,

$F \to y$,

$F \to z$.

Those expressions are called *productions*, and tell us how we can generate syntactically correct algebraic expressions by replacing successively the symbols on the left by the expressions on the right. For instance the algebraic expression "'$y+(x*y+y)*x$" can be generated like this:

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow y+T \Rightarrow y+T*F \Rightarrow y+F*F \Rightarrow$$

$$y+(E)*F \Rightarrow y+(E+T)*F \Rightarrow y+(T+T)*F \Rightarrow y+(T*F+T)*F \Rightarrow$$

$$y+(F*F+T)*F \Rightarrow y+(x*T+T)*F \Rightarrow y+(x*F+T)*F \Rightarrow$$

$$y+(x*y+T)*F \Rightarrow y+(x*y+F)*T \Rightarrow y+(x*y+y)*F \Rightarrow$$

$$y+(x*y+y)*x\,.$$

In general a *phrase-structure grammar* (or simply, *grammar*) G consists of

1. A finite set N of *nonterminal symbols*.
2. A finite set T of *terminal symbols*, where $N \cap T = \emptyset$.
3. A finite subset P of $[(N \cup T)^* - T^*] \times (N \cup T)^*$ called the set of *productions*.
4. A starting symbol $\sigma \in N$.

We write $G = (N, T, P, \sigma)$.

A production $(A, B) \in P$ is written:

$$A \to B\,.$$

The right hand side of a production can be any combination of terminal and nonterminal symbols. The left hand side must contain at least one nonterminal symbol.

If $\alpha \to \beta$ is a production and $x\alpha y \in (N \cup T)^*$, we say that $x\beta y$ is *directly derivable* from $x\alpha y$, and we write

$$x\alpha y \Rightarrow x\beta y\,.$$

If we have $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ $(n \geq 0)$, we say that α_n is *derivable* from α_1, and we write $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ (by convention also $\alpha_1 \overset{*}{\Rightarrow} \alpha_1$.)

Given a grammar G, the language $L(G)$ associated to this grammar is the subset of T^* consisting of all strings derivable from σ.

9.2.3. Backus Normal Form. The *Backus Normal Form* or *BNF* is an alternative way to represent productions. The production $S \to T$ is written $S ::= T$. Productions of the form $S ::= T_1$, $S ::= T_2$, ..., $S ::= T_n$, can be combined as

$$S ::= T_1 \mid T_2 \mid \cdots \mid T_n\,.$$

So, for instance, the grammar of algebraic expressions defined above can be written in BNF as follows:

$E ::= T \mid E + T$

$T ::= F \mid T * F$

$F ::= (E) \mid x \mid y \mid z$

9.2.4. Combining Grammars. Let $G_1 = (N_1, T_1, P_1, \sigma_1)$ and $G_2 = (N_2, T_2, P_2, \sigma_2)$ be two grammars, where N_1 and N_2 are disjoint (rename nonterminal symbols if necessary). Let $L_1 = L(G_1)$ and $L_2 = L(G_2)$ be the languages associated respectively to G_1 and G_2. Also assume that σ is a new symbol not in $N_1 \cup N_2 \cup T_1 \cup T_2$. Then

1. *Union Rule*: the language *union of L_1 and L_1*

   $$L_1 \cup L_2 = \{\alpha \mid \alpha \in L_1 \text{ or } \alpha \in L_1\}$$

   starts with the two productions

   $$\sigma \to \sigma_1\,, \quad \sigma \to \sigma_2\,.$$

2. *Product Rule*: the language *product of L_1 and L_2*

   $$L_1 L_2 = \{\alpha\beta \mid \alpha \in L_1,\ \beta \in L_1\}$$

where $\alpha\beta =$ string concatenation of α and β, starts with the production

$$\sigma \to \sigma_1\sigma_2\,.$$

3. *Closure Rule*: the language *closure of* L_1

$$L_1^* = L_1^0 \cup L_1^1 \cup L_1^2 \cup \dots$$

were $L_1^0 = \{\lambda\}$ and $L_1^n = \{\alpha_1\alpha_2\dots\alpha_n \mid \alpha_k \in L_1,\ k = 1, 2, \dots, n\}$ $(n = 1, 2, \dots)$, starts with the two productions

$$\sigma \to \sigma_1\sigma\,, \quad \sigma \to \lambda\,.$$

9.2.5. Types of Grammars (Chomsky's Classification). Let G be a grammar and let λ denote the null string.

0. G is a *phrase-structure* (or type 0) *grammar* if every production is of the form:

$$\alpha \to \delta\,,$$

where $\alpha \in (N \cup T)^* - T^*$, $\delta \in (N \cup T)^*$.

1. G is a *context-sensitive* (or type 1) *grammar* if every production is of the form:

$$\alpha A\beta \to \alpha\delta\beta$$

(i.e.: we may replace A with δ in the context of α and β), where $\alpha, \beta \in (N \cup T)^*$, $A \in N$, $\delta \in (N \cup T)^* - \{\lambda\}$.

2. G is a *context-free* (or type 2) *grammar* if every production is of the form:

$$A \to \delta\,,$$

where $A \in N$, $\delta \in (N \cup T)^*$.

3. G is a *regular* (or type 3) *grammar* if every production is of the form:

$$A \to a \quad \text{or} \quad A \to aB \quad \text{or} \quad A \to \lambda\,,$$

where $A, B \in N$, $a \in T$.

A language L is *context-sensitive* (respectively *context-free*, *regular*) if there is a context-sensitive (respectively context-free, regular) grammar G such that $L = L(G)$.

The following examples show that these grammars define different kinds of languages.

Example: The following language is type 3 (regular):

$$L = \{a^n b^m \mid n = 1, 2, 3 \ldots ; m = 1, 2, 3, \ldots\} .$$

A type 3 grammar for that language is the following: $T = \{a, b\}$, $N = \{\sigma, S\}$, with productions:

$$\sigma \to a\sigma \,, \quad \sigma \to aS \,, \quad S \to bS \,, \quad S \to b \,,$$

and starting symbol σ.

Example: The following language is type 2 (context-free) but not type 3:

$$L = \{a^n b^n \mid n = 1, 2, 3, \ldots\} .$$

A type 2 grammar for that language is the following:

$T = \{a, b\}$, $N = \{\sigma\}$, with productions

$$\sigma \to a\sigma b \,, \quad \sigma \to ab \,,$$

and starting symbol σ.

Example: The following language is type 1 (context-sensitive) but not type 2:

$$L = \{a^n b^n c^n \mid n = 1, 2, 3, \ldots\} .$$

A type 1 grammar for that language is the following:

$T = \{a, b, c\}$, $N = \{\sigma, A, C\}$, with productions

$$\begin{array}{ll} \sigma \to abc \,, & \sigma \to aAbc \,, \\ A \to abC \,, & A \to aAbC \,, \\ Cb \to bC \,, & Cc \to cc \,. \end{array}$$

and starting symbol σ.

There are also type 0 languages that are not type 1, but they harder to describe.

9.2.6. Equivalent Grammars. Two grammars G and G' are *equivalent* if $L(G) = L(G')$.

Example: The grammar of algebraic expressions defined at the beginning of the section is equivalent to the following one:

Terminal symbols $= \{x, y, z, +, *, (,)\}$, nonterminal symbols $= \{E, T, F, L\}$, with the productions

$E \to T, \quad E \to E + T,$

$T \to F, \quad T \to T * F$

$F \to (E), \quad F \to L,$

$L \to x, \quad L \to y, \quad L \to z,$

and starting symbol E.

9.2.7. Context-Free Interactive Lindenmayer Grammar. A context-free interactive Lindenmayer grammar is similar to a usual context-free grammar with the difference that it allows productions of the form $A \to B$ where $A \in N \cup T$ (in a context free grammar A must be nonterminal). Its rules for deriving strings also are different. In a context-free interactive Lindenmayer grammar, to derive string β from string α, all symbols in α must be replaced *simultaneously*.

Example: The von Koch Snowflake. The von Koch Snowflake is a fractal curve obtained by starting with a line segment and then at each stage transforming all segments of the figure into a four segment polygonal line, as shown below. The von Koch Snowflake fractal is the limit of the sequence of curves defined by that process.

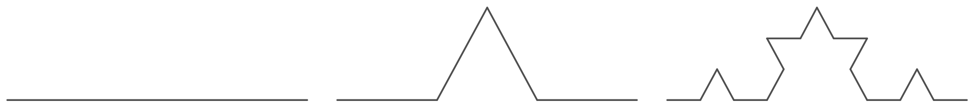


Figure 9.1. Von Koch Snowflake, stages 1–3.

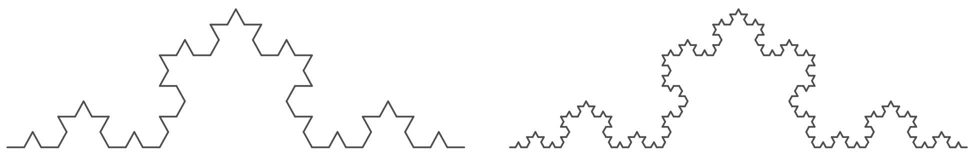


Figure 9.2. Von Koch Snowflake, stages 4–5

A way to represent an intermediate stage of the making of the fractal is by representing it as a sequence of movements of three kinds: 'd'= draw a straight line (of a fix length) in the current direction, 'r'= turn right by 60°, 'l'= turn left by 60°. For instance we start with a single horizontal line d, which we then transform into the polygonal $dldrrdld$, then each segment is transformed into a polygonal according to the rule $d \to dldrrdld$, so we get

$$dldrrdldldldrrdldrrdldrrdldldldrrdld$$

If we represent by D a segment that may no be final yet, then the sequences of commands used to build any intermediate stage of the curve can be defined with the following grammar:

$N = \{D\}$, $T = \{d, r, l\}$, with productions:

$$D \to DlDrrDlD\,, \quad D \to d\,, \quad r \to r\,, \quad l \to l\,,$$

and starting symbol D.

Example: The Peano curve. The Peano curve is a space filling curve, i.e., a function $f : [0,1] \to [0,1]^2$ such that the range of f is the whole square $[0,1]^2$, defined as the limit of the sequence of curves shown in the figures below.

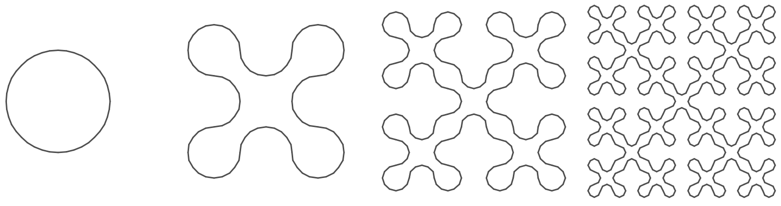


FIGURE 9.3. Peano curve, stages 1–4.

Each element of that sequence of curves can be described as a sequence of 90° arcs drawn either anticlockwise ('l') or clockwise ('r'). The corresponding grammar is as follows:

$T = \{l, r\}$, $N = \{C, L, R\}$, with productions

$$C \to LLLL\,,$$

$$L \to RLLLR\,, \quad R \to RLR\,,$$

$$L \to l\,, \quad R \to r\,, \quad l \to l\,, \quad r \to r\,,$$

and starting symbol C.

9.3. Regular Languages

9.3.1. Properties of Regular Languages. Recall that a regular language is the language associated to a regular grammar, i.e., a grammar $G = (N, T, P, \sigma)$ in which every production is of the form:

$$A \to a \quad \text{or} \quad A \to aB \quad \text{or} \quad A \to \lambda\,,$$

where $A, B \in N$, $a \in T$.

Regular languages over an alphabet T have the following properties (recall that $\lambda =$ 'empty string', $\alpha\beta =$ 'concatenation of α and β', $\alpha^n =$ 'α concatenated with itself n times'):

1. $\emptyset$, $\{\lambda\}$, and $\{a\}$ are regular languages for all $a \in T$.
2. If L_1 and L_2 are regular languages over T the following languages also are regular:

$$\begin{aligned} &L_1 \cup L_2 = \{\alpha \mid \alpha \in L_1 \text{ or } \alpha \in L_2\} \\ &L_1 L_2 = \{\alpha\beta \mid \alpha \in L_1,\ \beta \in L_2\} \\ &L_1^* = \{\alpha_1 \dots \alpha_n \mid \alpha_k \in L_1,\ n \in \mathbb{N}\}\,, \\ &T^* - L_1 = \{\alpha \in T^* \mid \alpha \notin L_1\}\,, \\ &L_1 \cap L_2 = \{\alpha \mid \alpha \in L_1 \text{ and } \alpha \in L_2\}\,. \end{aligned}$$

We justify the above claims about $L_1 \cup L_2$, $L_1 L_2$ and L_1^* as follows. We already know how to combine two grammars (see 9.2.4) L_1 and L_2 to obtain $L_1 \cup L_2$, $L_1 L_2$ and L_1^*, the only problem is that the rules given in section 9.2.4 do no have the form of a regular grammar, so we need to modify them slightly (we use the same notation as in section 9.2.4):

1. *Union Rule*: Instead of adding $\sigma \to \sigma_1$ and $\sigma \to \sigma_2$, add all productions of the form $\sigma \to RHS$, where RHS is the right hand side of some production $(\sigma_1 \to RHS) \in P_1$ or $(\sigma_2 \to RHS) \in P_2$.
2. *Product Rule*: Instead of adding $\sigma \to \sigma_1\sigma_2$, use σ_1 as starting symbol and replace each production $(A \to a) \in P_1$ with $A \to a\sigma_2$ and $(A \to \lambda) \in P_1$ with $A \to \sigma_2$.
3. *Closure Rule*: Instead of adding $\sigma \to \sigma_1\sigma$ and $\sigma \to \lambda$, use σ_1 as starting symbol, add $\sigma_1 \to \lambda$, and replace each production $(A \to a) \in P_1$ with $A \to a\sigma_1$ and $(A \to \lambda) \in P_1$ with $A \to \sigma_1$.

9.3.2. Regular Expressions. Regular languages can be characterized as languages defined by *regular expressions*. Given an alphabet T, a regular expression over T is defined recursively as follows:

1. $\emptyset$, λ, and a are regular expressions for all $a \in T$.
2. If R and S are regular expressions over T the following expressions are also regular: (R), $R + S$, $R \cdot S$, R^*.

In order to use fewer parentheses we assign those operations the following hierarchy (from do first to do last): $*, \cdot, +$. We may omit the dot: $\alpha \cdot \beta = \alpha\beta$.

Next we define recursively the language associated to a given regular expression:

$$\begin{aligned}
&L(\emptyset) = \emptyset\,, \\
&L(\lambda) = \{\lambda\}\,, \\
&L(a) = \{a\} && \text{for each } a \in T, \\
&L(R + S) = L(R) \cup L(S)\,, \\
&L(R \cdot S) = L(R)L(S) && \text{(language product)}, \\
&L(R^*) = L(R)^* && \text{(language closure)}.
\end{aligned}$$

So, for instance, the expression a^*bb^* represents all strings of the form a^nb^m with $n \geq 0$, $m > 0$, $a^*(b + c)$ is the set of strings consisting of any number of a's followed by a b or a c, $a(a + b)^*b$ is the set of strings over $\{a, b\}$ than start with a and end with b, etc.

Another way of characterizing regular languages is as sets of strings recognized by finite-state automata, as we will see next. But first we need a generalization of the concept of finite-state automaton.

9.3.3. Nondeterministic Finite-State Automata. A *nondeterministic finite-state automaton* is a generalization of a finite-state automaton so that at each state there might be several possible choices for the "next state" instead of just one. Formally a nondeterministic finite-state automaton consists of

1. A finite set of *input symbols* $\mathcal{I}$.
2. A finite set of *states* $\mathcal{S}$.
3. A *next-state function* $f : \mathcal{S} \times \mathcal{I} \to \mathcal{P}(\mathcal{S})$.
4. A subset $\mathcal{A}$ of $\mathcal{S}$ of *accepting states*.

5. An *initial state* $\sigma \in \mathcal{S}$.

We represent the automaton $A = (\mathcal{I}, \mathcal{S}, f, \mathcal{A}, \sigma)$. We say that a nondeterministic finite-state automaton *accepts* a given string of input symbols if in its transition diagram there is a path from the starting state to an accepting state with its edges labeled by the symbols of the given string. A path (which we can express as a sequence of states) whose edges are labeled with the symbols of a string is said to *represent* the given string.

Example: Consider the nondeterministic finite-state automaton defined by the following transition diagram:

$$\begin{tikzcd}
\text{start} \arrow[r] & \boxed{\sigma} \arrow[loop above, "b"] \arrow[r, "a"] & \boxed{C} \arrow[loop above, "b"] \arrow[r, "b"] & \boxed{\boxed{F}}
\end{tikzcd}$$

This automaton accepts precisely the strings of the form $b^n a b^m$, $n \geq 0$, $m > 0$. For instance the string *bbabb* is represented by the path $(\sigma, \sigma, \sigma, C, C, F)$. Since that path ends in an accepting state, the string is accepted by the automaton.

Next we will see that there is a precise relation between regular grammars and nondeterministic finite-state automata.

Regular grammar associated to a nondeterministic finite-state automaton. Let A be a non-deterministic finite-state automaton given as a transition diagram. Let σ be the initial state. Let T be the set of inputs symbols and let N be the set of states. Let P be the set of productions

$$S \to xS'$$

if there is an edge labeled x from S to S' and

$$S \to \lambda$$

if S is an accepting state. Let G be the regular grammar

$$G = (N, T, P, \sigma)\,.$$

Then the set of strings accepted by A is precisely $L(G)$.

Example: For the nondeterministic automaton defined above the corresponding grammar will be:

$T = \{a, b\}$, $N = \{\sigma, C, F\}$, with the productions

$$\sigma \to b\sigma\,, \quad \sigma \to aC\,, \quad C \to bC\,, \quad C \to bF\,, \quad F \to \lambda\,.$$

The string *bbabb* can be produced like this:

$$\sigma \Rightarrow b\sigma \Rightarrow bb\sigma \Rightarrow bbaC \Rightarrow bbabC \Rightarrow bbabbF \Rightarrow bbabb\,.$$

Nondeterministic finite-state automaton associated to a given regular grammar. Let $G = (N, T, P, \sigma)$ be a regular grammar. Let

$\mathcal{I} = T$

$\mathcal{S} = N \cup \{F\}\,,$ where $F \notin N \cup T$

$f(S, x) = \{S' \mid S \to xS' \in P\} \cup \{F \mid S \to x \in P\}$

$\mathcal{A} = \{F\} \cup \{S \mid S \to \lambda \in P\}$.

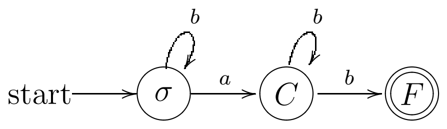
Then the nondeterministic finite-state automaton $\mathcal{A} = (\mathcal{I}, \mathcal{S}, f, \mathcal{A}, \sigma)$ accepts precisely the strings in $L(G)$.

9.3.4. Relationships Between Regular Languages and Automata. In the previous section we saw that regular languages coincide with the languages accepted by nondeterministic finite-state automata. Here we will see that the term "nondeterministic" can be dropped, so that regular languages are precisely those accepted by (deterministic) finite-state automata. The idea is to show that given any nondeterministic finite-state automata it is possible to construct an equivalent deterministic finite-state automata accepting exactly the same set of strings. The main result is the following:

Let $A = (\mathcal{I}, \mathcal{S}, f, \mathcal{A}, \sigma)$ be a nondeterministic finite-state automaton. Then A is equivalent to the finite-state automaton $A' = (\mathcal{I}', \mathcal{S}', f', \mathcal{A}', \sigma')$, where

1. $\mathcal{S}' = \mathcal{P}(\mathcal{S})$.
2. $\mathcal{I}' = \mathcal{I}$.
3. $\sigma' = \{\sigma\}$.
4. $\mathcal{A}' = \{X \subseteq \mathcal{S} \mid X \cap \mathcal{A} \neq \emptyset\}$.
5. $f'(X, x) = \bigcup_{S \in X} f(S, x)\,, \quad f'(\emptyset, x) = \emptyset\,.$

Example: Find a (deterministic) finite-state automaton A' equivalent to the following nondeterministic finite-state automaton A:



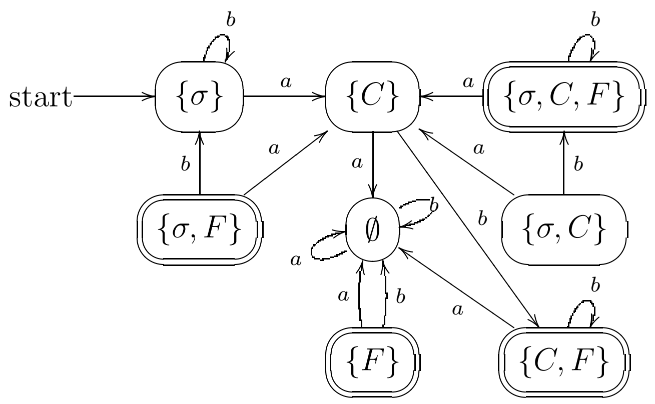
Answer: The set of input symbols is the same as that of the given automaton: $\mathcal{I}' = \mathcal{I} = \{a, b\}$. The set of states is the set of subsets of $\mathcal{S} = \{\sigma, C, F\}$, i.e.:

$$\mathcal{S}' = \{\emptyset, \{\sigma\}, \{C\}, \{F\}, \{\sigma, C\}, \{\sigma, F\}, \{C, F\}, \{\sigma, C, F\}\}\,.$$

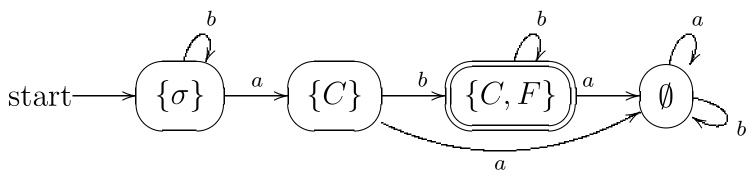
The starting state is $\{\sigma\}$. The accepting states of A' are the elements of $\mathcal{S}'$ containing some accepting state of A:

$$\mathcal{A}' = \{\{F\}, \{\sigma, F\}, \{C, F\}, \{\sigma, C, F\}\}\,.$$

Then for each element X of $\mathcal{S}'$ we draw an edge labeled x from X to $\bigcup_{S \in X} f(S, x)$ (and from $\emptyset$ to $\emptyset$):



We notice that some states are unreachable from the starting state. After removing the unreachable states we get the following simplified version of the finite-state automaton:



So, once proved that every nondeterministic finite-state automaton is equivalent to some deterministic finite-state automaton, we obtain the main result of this section: A language L is regular if and only if there exists a finite-state automaton that accepts precisely the strings in L.

APPENDIX A

A.1. Efficient Computation of Powers Modulo m

We illustrate an efficient method of computing powers modulo m with an example. Assume that we want to compute $3^{547} \bmod 10$. First write 547 in base 2: 1000100011, hence $547 = 2^9 + 2^5 + 2 + 1 = ((2^4+1)\,2^4+1)\,2+1$, so: $3^{547} = ((3^{2^4}\cdot 3)^{2^4}\cdot 3)^2\cdot 3$. Next we compute the expression beginning with the inner parenthesis, and reducing modulo 10 at each step: $3^2 = 9 \pmod{10}$, $3^{2^2} = 9^2 = 81 = 1 \pmod{10}$, $3^{2^3} = 1^2 = 1 \pmod{10}$, $3^{2^4} = 1^2 = 1 \pmod{10}$, $3^{2^4}\cdot 3 = 1\cdot 3 = 3 \pmod{10}$, etc. At the end we find $3^{547} = 7 \pmod{10}$.

The algorithm in pseudocode would be like this:

```
 1: procedure pow_mod(a,x,m) // computes a^x mod m
 2:   p := 1
 3:   bx := binary_array(x) // x as a binary array
 4:   t := a mod m
 5:   for k := 1 to length(bx) do
 6:     begin
 7:       p := (p * p) mod m
 8:       if bx[k] = 1 then
            // if k-th binary digit of x is 1
 9:         p := (p * t) mod m
10:     end
11:   return(p)
12: end pow_mod
```

The following is a program in C implementing the algorithm:

```c
int pow(int a, int x, int m) {
  int p = 1;
  int y = (1 << (8 * size of(int) - 2));

  a %= m;

  while (!(y & x)) y >>= 1;

  while (y) {
    p *= p;
    p %= m;
    if (x & y) {
      p *= a;
      p %= m;
    }
    y >>= 1;
  }
  return p;
}
```

The following is an alternative algorithm equivalent to running through the binary representation of the exponent from right to left instead of left to right:

```
procedure pow_mod(a,x,m) // computes a^x mod m
  p := 1
  t := a mod m
  while x > 0 do
    begin
      if x is odd then
        p := (p * t) mod m
      t := (t * t) mod m
      x := floor(x/2)
    end
  return(p)
end pow_mod
```

A.2. Machines and Languages

A.2.1. Turing Machines. A *Turing machine* is a theoretical device intended to define rigorously the concept of *algorithm*. It consists of

1. An *infinite tape* made of a sequence of cells. Each cell may be empty or may contain a symbol from a given alphabet.
2. A *control unit* containing a finite set of instructions.
3. A *tape head* able to read and write (or delete) symbols from the tape.

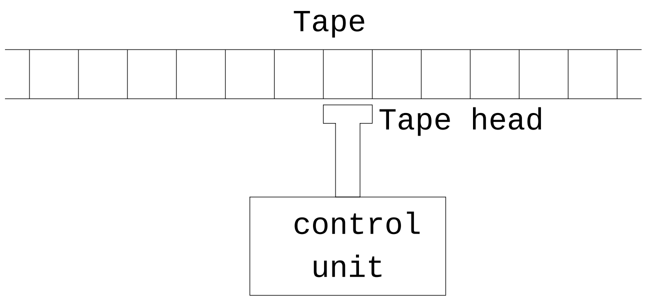


FIGURE A.1. Turing Machine.

Each machine instruction contains the following five parts:

1. The current machine state.
2. A tape symbol read from the current tape cell.
3. A tape symbol to write into the current tape cell.
4. A direction for the tape head to move: L = 'move one cell to the left', R = 'move one cell to the right', S = 'stay in the current cell'.
5. The next machine state.

Turing machines are generalizations of finite-state automata. A finite-state automaton is just a Turing machine whose tape head moves always from left to right and never writes to the tape. The input of the finite-state automaton is presented as symbols written in the tape.

In general we make the following assumptions:

1. An input is represented on the tape by placing the letters of the strings in contiguous tape cells. All other cells contain the blank symbol, which we may denote λ.

2. The tape is initially positioned at the leftmost cell of the input string unless specified otherwise.
3. There is one *start state*.
4. There is one *halt state*, which we denote by “Halt”.

The execution of a Turing machine stops when it enters the Halt state or when it enters a state for which there is no valid move. The output of the Turing machine is the contents of the tape when the machine stops.

We say that an input string is *accepted* by a Turing machine if the machine enters the Halt state. Otherwise the string is *rejected*. This can happen in two ways: by entering a state other than the Halt state from which there is no move, or by running forever (for instance executing an infinite loop).

If a Turing machine has at least two instructions with the same state and input letter, then the machine is *nondeterministic*. Otherwise it is *deterministic*.

Finite-State Automata. A finite-state automata can be interpreted as a Turing machine whose tape head moves only from left to right and never writes to the tape.

Pushdown Automata. A *pushdown automaton* is finite-state automaton with a stack, i.e., a storage structure in which symbols can be put and extracted from it by two operations: *push* (place on the top of the stack) and *pop* (take from the top of the stack)—consequently the last symbol put into the stack is the first symbol taken out. Additionally there is a third operation, *nop*, that leaves the stack intact. The next state function takes into account not only the current state and the symbol read from the input, but also the symbol at the top of the stack. After reading the next input symbol and the symbol at the top of the stack, the automaton executes a stack operation and goes to the next state. Initially there is a single symbol in the stack.

Linearly Bounded Automata. A *linearly bounded automaton* is a Turing machine whose tape is limited to the size of its input string plus two boundary cells that may not be changed.

Computable Functions. Consider a Turing machine T working on symbols from an alphabet of only one symbol $A = \{|\}$ (“stroke”). Let $f : \mathbb{N} \to \mathbb{N}$ the function defined so that $f(n) = m$ means that if the initial input of T consists of a string of $n+1$ strokes, the output of T is a string of $m+1$ strokes. We say that f is *computed* by the Turing machine T. A *computable function* is a function computed by some Turing machine. A computable function $f(n)$ *halts* for a given value of its argument n if T with input $n+1$ strokes halts. A computable function f is *total* if $f(n)$ halts for every n.

An *effective* enumeration of a set is a listing of its elements by an algorithm.

A.2.2. Hierarchy of Languages. Here we mention a hierarchy of languages that includes (and extends) Chomsky's classification, in increasing order of inclusion.

1. *Regular languages.* They are recognized by finite-state automata. *Example*: $\{a^m b^n \mid m, n = 1, 2, 3 \dots\}$.
2. *Deterministic context-free languages*, recognized by deterministic pushdown automata. *Example*: $\{a^n b^n \mid n = 1, 2, 3 \dots\}$.
3. *Context-free languages*, recognized by nondeterministic pushdown automata. *Example*: palindromes over $\{a, b\}$.
4. *Context-sensitive languages*, languages without λ recognized by linearly bounded automata. *Example*: $\{a^n b^n c^n \mid n = 1, 2, 3 \dots\}$
5. *Unrestricted or phrase-structure grammars*, recognized by Turing machines.
6. *Recursively enumerable languages.* A language is recursively enumerable if there is a Turing machine that outputs all the strings of the language. *Example*: $\{a^n \mid f_n(n) \text{ halts}\}$, where $f_0, f_1, f_2, \dots$ is an effective enumeration of all computable functions.
7. *Nongramatical languages*, languages that are not definable by any grammar and cannot be recognized by Turing machines. *Example*: $\{a^n \mid f_n \text{ is total}\}$.